SOLVING HARD SUBGRAPH PROBLEMS IN PARALLEL

CIARAN MCCREESH

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

SCHOOL OF COMPUTING SCIENCE
COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

16TH JULY 2017

Abstract

This thesis improves the state of the art in exact, practical algorithms for finding subgraphs. We study maximum clique, subgraph isomorphism, and maximum common subgraph problems. These are widely applicable: within computing science, subgraph problems arise in document clustering, computer vision, the design of communication protocols, model checking, compiler code generation, malware detection, cryptography, and robotics; beyond, applications occur in biochemistry, electrical engineering, mathematics, law enforcement, fraud detection, fault diagnosis, manufacturing, and sociology. We therefore consider both the "pure" forms of these problems, and variants with labels and other domain-specific constraints.

Although subgraph-finding should theoretically be hard, the constraint-based search algorithms we discuss can easily solve real-world instances involving graphs with thousands of vertices, and millions of edges. We therefore ask: is it possible to generate "really hard" instances for these problems, and if so, what can we learn? By extending research into combinatorial phase transition phenomena, we develop a better understanding of branching heuristics, as well as highlighting a serious flaw in the design of graph database systems.

This thesis also demonstrates how to exploit two of the kinds of parallelism offered by current computer hardware. Bit parallelism allows us to carry out operations on whole sets of vertices in a single instruction—this is largely routine. Thread parallelism, to make use of the multiple cores offered by all modern processors, is more complex. We suggest three desirable performance characteristics that we would like when introducing thread parallelism: lack of risk (parallel cannot be exponentially slower than sequential), scalability (adding more processing cores cannot make runtimes worse), and reproducibility (the same instance on the same hardware will take roughly the same time every time it is run). We then detail the difficulties in guaranteeing these characteristics when using modern algorithmic techniques.

Besides ensuring that parallelism cannot make things worse, we also increase the likelihood of it making things better. We compare randomised work stealing to new tailored strategies, and perform experiments to identify the factors contributing to good speedups. We show that whilst load balancing is difficult, the primary factor influencing the results is the interaction between branching heuristics and parallelism. By using parallelism to explicitly offset the commitment made to weak early branching choices, we obtain parallel subgraph solvers which are substantially and consistently better than the best sequential algorithms.

Acknowledgements

I would like to thank Patrick Prosser for letting me do this: I hope you have enjoyed it as much as I have. Thanks also to my second supervisor, David Manlove, whose advice and perspectives have made this a better thesis. This work has relied heavily upon access to large amounts of carefully configured hardware: thanks to Douglas MacFarlane and Pete Bailey who went far beyond the call of duty to make this possible, and to Phil Trinder for looking the other way every time I borrowed his toys.

Thanks to Andre Cire, for arranging a visit to the University of Toronto for me, and to Chris Beck for our conversations whilst I was there. Thanks also to Peter Stuckey for hosting me on my trip to the University of Melbourne, and for helping me to decide what to work on next. Whilst at Melbourne I had many enjoyable lunches with Jimmy Lee. Like so many others in the constraint programming community, he was always happy to give helpful advice and encouragement. Closer to home, I am similarly grateful to Özgür Akgün, Jess Enright, Ian Gent, Chris Jefferson, Ian Miguel, Pete Nightingale, and Karen Petrie.

Thanks also to Blair Archibald, Ruth Hoffmann, Lars Kotthoff, Patrick Maier, Samba Ndojh Ndiaye, Craig Reilly, Christine Solnon, Rob Stewart, and James Trimble. I hope I'll be able to work with some of you again. And thanks to Stephen Strowes for wrestling with LaTeX to make everything look shiny and compliant.

To Felix Fischer and Christian Schulte, thank you for making the viva a fun experience. I enjoyed arguing with you.

Finally, in lieu of the customary thanks to God for making this work *possible*, I would like to thank Satan and all his little minions for making it *necessary*.

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/K503058/1], a Jim Gatheral Travel Scholarship, and conference and summer school travel grants from AIJ, CP, SICSA and SoCS.

Contents

Chapter 1

Introduction

A *graph* is a collection of *vertices*, together with *edges* which go between pairs of *adjacent* vertices (we defer formal definitions in the interests of accessibility). Graphs are often drawn using dots or circles for vertices, and lines for edges, as in Figure 1.1; we might associate *labels*, *weights*, or other data with vertices or edges, and sometimes edges are *directed*, in which case they are drawn as arrows rather than lines. Graphs may be used to model relationships, molecules, transportation and financial networks, images, 3D objects, and other structured data.

This thesis looks at problems which involve finding subgraphs with certain properties inside given graphs. A subgraph is a smaller part of a larger graph. Subgraphs come in two forms: for an *induced* subgraph, we look at some of the vertices, but all of the edges between those vertices from the original graph, whereas for a *non-induced* subgraph (sometimes called *partial*), we are allowed to discard both vertices and edges. Sometimes we also require *connected* subgraphs, where between any two distinct vertices, we must be able to find a path of adjacent vertices connecting them. We give examples of each in Figure 1.1.

Finding subgraphs with particular properties is, in general, NP-hard. Very informally, this means we believe that adding just one extra vertex to a graph makes the problem twice as hard in the worst case, so the problems become exponentially more complicated as the graphs get larger. However, we will see that with carefully designed practical algorithms, worst-case

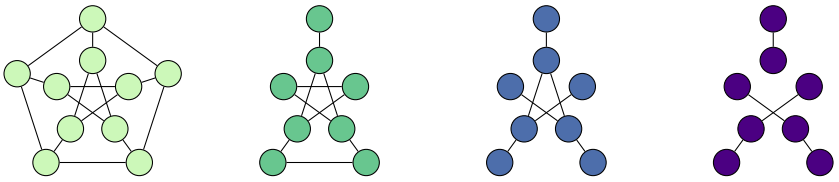


Figure 1.1: On the left, a graph with ten vertices and fifteen edges. The second graph is an induced subgraph of the first, obtained by deleting two vertices, and the four edges involving those two vertices. The third graph is also a subgraph of the first, but is not induced, since we have deleted a further two edges which are not incident upon any deleted vertex. The fourth graph is still a subgraph of the first, but it is neither induced nor connected.

complexity bounds are too pessimistic. The title of this thesis starts with the word "solving", since we are able to produce exact solutions for some instances of these problems involving graphs with tens of thousands of vertices and up to a million edges.

1.1 Hard Subgraph Problems

We begin with an overview of the hard subgraph problems that we will solve in this thesis. These problems fall into three families: clique problems, subgraph isomorphism problems, and maximum common subgraph problems.

1.1.1 Maximum Clique

The *maximum clique* problem is to find a largest possible *complete* subgraph—that is, if the vertices in our graph represent people, and the edges represent friendships, then we want to find the largest set of people where everyone in this set is friends with everyone else in the set. We give an example in Figure 1.2. Finding cliques can be useful in its own right, but the maximum clique problem can also show up as an intermediate step in solving another problem. For example, suppose we have a set of resources, only some of which are compatible with others. We can represent this as a graph, with a vertex for each resource and edges between compatible resources. A maximum clique in this graph tells us the largest set of mutually compatible resources we can select. Because of this, maximum clique algorithms have been used for applications in bioinformatics and biochemistry, for community detection, for document clustering, in computer vision, in electrical engineering and communications, for fault diagnosis, in mathematics, for image comparison, and in robotics. We review these applications and take a detailed look at maximum clique algorithms in Chapters 2 and 3; an alternative recent review is provided by Wu and Hao (2015).

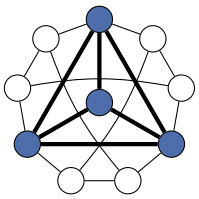


Figure 1.2: A graph, with its unique maximum clique of four vertices highlighted.

The main algorithm we investigate for the maximum clique problem is reasonably flexible, and is capable of working with dense graphs (those where a high proportion of possible edges are present). This allows us to use the same algorithm to solve other problems. Most simply, a *maximum independent set* is a largest-possible subset of vertices, no two of which are adjacent. To find independent sets using a clique algorithm, we simply work with the *complement* of the input graph, replacing edges with non-edges and vice-versa. A different change to the input graph lets us solve a distance-based relaxation of the clique problem, called k*-clique*, which just requires members of the "clique" to be within a certain distance k of each other, rather

than adjacent—this can be useful in social network analysis, where a conventional clique is too strict a requirement (Luce, 1950). We discuss this problem in Chapter 4. More complex encodings allow us to reuse clique algorithms for subgraph isomorphism and maximum common subgraph problems, which we discuss below.

By adapting the algorithm, we can handle certain kinds of side constraints. For example, in the *maximum labelled clique* variant due to Carrabs, Cerulli, and Dell'Olmo (2014), edges have labels, and we may only use a certain different number of labels in the solution—this is useful in telecommunications and in social network analysis. Further adaptations allow us to search for certain other kinds of highly regular pattern—we will illustrate this using a kind of graph called a *balanced induced biclique*. We study both of these problems in Chapter 4. Searching for an arbitrary pattern leads us to the subgraph isomorphism family of problems, which we discuss next.

1.1.2 Subgraph Isomorphism

What if we are interested in a different shape of subgraph? In the *subgraph isomorphism* problem, we must find a given *pattern* graph inside a larger *target* graph (or detect that the pattern does not occur). We discuss this problem in Chapters 5 and 6, where we see that subgraph isomorphism is used in computer vision applications, for law enforcement and fraud detection, for pattern recognition, for model checking, for malware detection, for code comparison, in compiler code generation systems, and particularly in biological and chemical applications, where the target graphs typically represent molecules or proteins and the pattern graphs describe desired or undesired properties.

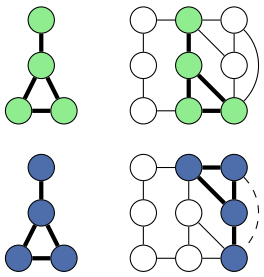


Figure 1.3: The two kinds of subgraph isomorphism. Above, the small pattern graph can be found inside the larger target graph, as shown. This is an induced subgraph isomorphism, whereas the second example is not, due to the extra dashed edge which is present in the target but not the pattern.

The problem comes in both induced and non-induced variants, as we show in Figure 1.3. Additionally, sometimes we must preserve labels on vertices, edges, or both, and sometimes edges are directed. Finally, in the case that the pattern does not occur, we may wish to know how much of the pattern can be found—this leads us to the maximum common subgraph problem.

1.1.3 Maximum Common Subgraphs

The final problems we discuss in this thesis are *maximum common subgraph* problems, which are the subject of Chapter 7. In these problems we must find a largest-possible graph which

Figure 1.4: The maximum common subgraph problem. On the left, a maximum common subgraph has eight vertices. However, if we require the common subgraph to be connected as on the right, then only seven vertices may be selected.

is a subgraph of two given graphs simultaneously. Maximum common subgraph problems are the key step in comparing graphs: to determine the difference between two graphs, we first find what they have in common (Kriege, 2015). We will list applications in biology and chemistry, in computer vision, in the analysis of source code and binary programs, in circuit designs, in computer-aided manufacturing, in crisis management, in deanonymising datasets, and in character recognition problems; the biochemical applications alone are extensive and important enough to justify extensive research into the problem (Ehrlich and Rarey, 2011). Some of these applications, including several from biochemistry, require that the selected graph be connected—we show how this can make a difference in Figure 1.4.

1.1.4 Why are These Problems Hard?

So far we have illustrated problems with carefully selected small examples that make the solutions obvious, and then selected a layout for the graphs which emphasises the solution visually. This may give the incorrect impression that such problems are easy (at least for a human). However, graphs do not usually come equipped with a helpful visual layout, and these problems can quickly become very difficult as the graphs grow. The reader is encouraged to try to solve the problem in Figure 1.5 manually, to get a feel for why even small instances of these problems can be so difficult. In the absence of obvious visual clues, one must resort to considering every possible combination of vertex mappings.

Although *finding* a subgraph isomorphism is difficult, if we are given a solution showing that a subgraph isomorphism exists, *verifying* it is straightforward. The same property holds for the decision versions ("does a solution of size x exist?") of the maximum clique and maximum common subgraph problems: if I want to convince you that a graph contains a clique with 10 vertices, I may simply tell you what those vertices are, and you can easily verify

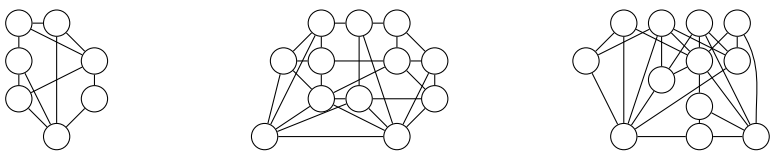


Figure 1.5: There is a non-induced subgraph isomorphism from the pattern on the left, to one of the two target graphs on the right, but not to the other. Attempting to solve these two instances by hand gives a good intuition for why this problem is hard.

that I am telling the truth. Problems where, if the answer is yes, then there is always a proof which may be verified in polynomial time, are common, and are called NP (*nondeterministic polynomial*).

On the other hand, there is no simple way to convince you that a graph does *not* contain a clique of size x. Although for certain graphs there may be tricks leading to a short proof, in general, it is widely believed that it is impossible to do substantially better than showing you all sets of x vertices, and asserting that none of them form a clique. Problems where, if the answer is no, then there is always a short proof, are called coNP. For example, to show that a given clique is *not* the largest in a graph, one may always prove this by listing the vertices of a larger clique.

A final property of all of the problems we study is that they are in some sense equally difficult: if an algorithm exists which can solve either clique decision, or subgraph isomorphism, or common subgraph decision, in polynomial time, then an algorithm exists to solve any NP problem in polynomial time. Problems with this property are called NP-complete. In general NP-completeness is established by showing that an existing known NP-complete problem may be *reduced* to another.[1] Such proofs are trivial for every problem considered in this thesis, and so we do not give them. However, in Chapter 7 we will look at using reductions to clique as a practical technique for solving maximum common subgraph problems.

Problems where a solution may be *found* in polynomial time are said to be in P. We believe that having easily verified yes-instances does not make a problem easier to solve, or even help with verifying no-instances: that is, there are problems in NP (including all NP-complete problems) which are not in P or coNP. The standard introduction for topics involving NP is Garey and Johnson (1979), and Hromkovič (2004) provides a reference for subsequent developments. Determining the exact relationship between NP, coNP, NP-complete, and P problems is outwith the scope of this thesis; given the lack of evidence to the contrary, we assume that the P, NP and coNP complexity classes are all different, and that the apparent hardness of NP-complete problems is in fact genuine.

1.1.5 Definitions and Notation

We now introduce common notation for graph concepts which occur throughout this thesis; further notation will be introduced as it is needed.

Let $G = (V, E)$ be a graph with vertex set $\mathrm{V}(G) = V$ and edge set $\mathrm{E}(G) = E$. We always use the term *vertex* when referring to elements of V, and reserve the term *node* to refer to a vertex in a search tree.

The *order* of G is the cardinality of V, and the *size* the cardinality of E. We write $v \sim_G w$ to indicate that vertex v is adjacent to vertex w. Except where otherwise noted, graphs in this

[1] The egg came first, in the form of the *circuit satisfiability* problem.

thesis are undirected, so $v \sim_G w$ if and only if $w \sim_G v$. A *loop* is a vertex which is adjacent to itself. When working with the maximum clique problem, graphs do not have loops, but for subgraph isomorphism and maximum common subgraph problems, loops are permitted.

The *subgraph induced by* a set of vertices $S \subseteq \mathrm{V}(G)$, written $G[S]$, is the graph with vertex set S, and every edge from G with both endpoints in S. A subgraph is *induced* if it is induced by some subset. The *complement* of a graph G is the graph $\overline{G}$: adjacent distinct vertices in G are non-adjacent in $\overline{G}$, whilst non-adjacent vertices in G are adjacent in $\overline{G}$. If G may have loops, we instead talk about the *loop complement*, $\overline{G}^{\circlearrowleft}$: the construction of $\overline{G}^{\circlearrowleft}$ is similar to that of $\overline{G}$, but whenever a vertex has a loop in G it does not in $\overline{G}^{\circlearrowleft}$ and vice versa.

The *neighbourhood* of a vertex v, written $\mathrm{N}(G, v)$, is the set of vertices adjacent to v, and the cardinality of this set is the *degree* of v. A *path* is a sequence of distinct vertices, where every consecutive pair is adjacent. We also allow the start and end vertices of a path to be the same, in which case we have a *cycle*. The *distance* between two vertices is the length of a shortest path between them. A graph is *connected* if there is a path between every pair of vertices. A connected graph with no cycles is called a *tree*.

If S and T are sets, we write $S \setminus T$ to mean the elements in S which are not also in T. If v is an element of a set, we write $S + v$ and $S - v$ to mean $S \cup \{v\}$ and $S \setminus \{v\}$ respectively. Finally, we write $\vee$ for logical "or", and $\wedge$ for logical "and".

1.2 Modelling with Constraints

Each of the problems from the previous section can be seen as a kind of *constraint satisfaction problem*, or CSP. In a CSP we have a set of *variables*, each of which has a *domain* of possible *values*. We also have a set of *constraints*, each of which specifies an allowed combination of values for some subset of the variables. We then seek to find a way of giving each variable a value from its domain, such that every constraint is satisfied. In a *constraint optimisation problem*, we extend the problem to look for a best solution, with respect to some scoring function.

The most widely known CSP is probably Sudoku. In this popular puzzle, we are given a 9 by 9 grid. Some of the boxes contain a number between 1 and 9, and the objective is to place a number between 1 and 9 in each remaining box, such that each number appears exactly once in each row, once in column, and once in each of the nine 3 by 3 subgrids. To view this as a CSP, we create a variable for each of the $9 \times 9 = 81$ boxes, and give each variable a domain containing the numbers 1 through 9. We then have a constraint for each row, column, and subgrid, saying that all nine variables must be different. (Since there are nine values and nine variables, "use each value once" and "each value must be different" say the same thing.) Finally, for each of the pre-filled boxes, we have a constraint saying that the appropriate box must by equal to its specified value. Note that "proper" Sudoku puzzles are

supposed to contain exactly one solution; in contrast, a CSP may contain many solutions, or none at all.

To encode non-induced subgraph isomorphism as a CSP, we have a variable for each vertex in the pattern graph. The domain of each variable has a value for each vertex in the target graph. We then have a constraint for each pair of vertices which are adjacent in the pattern (that is, for each edge), saying that together these two vertices may only be mapped to a pair of target vertices which are also adjacent. Finally, we have a constraint saying that each variable must be given a different value.

Another view of this model is that we are creating a *function*, which provides a *mapping* from pattern vertices to target vertices. The all-different constraint ensures that the function is *injective*, whilst the adjacency constraints ensure a property known as *homomorphism*. For an accessible introduction to modelling with constraints, including more detail on function-type problems, we refer to Stuckey and Coffrin (2016).

1.3 Solving with Constraints

One way to solve a CSP is simply to *generate* every possible assignment of values to variables in turn, and then *test* to see if all the constraints are satisfied. However, the number of combinations involved usually makes such an approach impractical, and we must move to more intelligent[2] algorithms to be able to handle non-trivial problems. Most of the algorithms we will look at in this thesis are based around *inference*, *bounds*, and *heuristic*-driven *search*. A full discussion of these topics is given in F. Rossi, van Beek, and Walsh (2006); we now give a brief overview of the most important aspects.

1.3.1 Inference

It is sometimes possible to deduce that certain variables can never be given a particular value—for example, in subgraph isomorphism, a pattern vertex which has n neighbours can never be matched to a target vertex which has fewer than n neighbours. We show this in Figure 1.6. Since there is no point in generating any combination which includes a forbidden assignment, this kind of reasoning can be used to reduce the amount of work we may have to perform. To represent this knowledge algorithmically, for each variable, we can keep track of the values remaining in its domain. We then remove any value which our deductions rule out—like when the remaining possible numbers are written in Sudoku boxes, and then crossed out as facts are deduced. We call an algorithm which eliminates infeasible values a *propagator*.

[2]Or sophisticated, if the reader believes that an algorithm can only be intelligent if we do not understand why it works.

Figure 1.6: When trying to find the smaller pattern graph inside the larger target graph, we notice that the central vertex c in the pattern is adjacent to four other vertices. Thus it may only be matched with one of the two shaded vertices in the target which also have at least four neighbours, so the domain of c can be reduced to $\{4, 5\}$. Similarly, on the right, the two pattern vertices a and d which have two neighbours cannot be mapped to any of the four target vertices which have only one neighbour, and so their domains can be reduced to $\{4, 5, 6, 7\}$. This kind of reasoning allows us to eliminate some assignments of values to variables.

Sometimes we will end up with a variable with only one value left in its domain. We can often use this to eliminate further values—for example, in subgraph isomorphism, if we know that a particular pattern vertex v can only be mapped to one target vertex w, then for each variable corresponding to a vertex adjacent to v, we can eliminate any value representing target vertices which are not adjacent to w. This can potentially have a cascade effect, allowing further deletions. It is also sometimes possible to do reasoning involving combinations of domains which each have more than one value remaining. For example, in subgraph isomorphism (and other problems like Sudoku where a set of variables must all take different values), if we can find a set of n variables that only have n different values between them, then we can eliminate those n values from other domains. An example using these kinds of inference in a somewhat ad-hoc manner is given in Figure 1.7; systematic approaches such as that of Régin (1994) are discussed in Chapter 5.

Stronger levels of consistency can sometimes be calculated (Mackworth, 1977; Sabin and Freuder, 1994). For example, in some problems we may be able to infer further information if we check *support* for each value v in a domain—that is, if we look at each constraint in turn which involves v's variable, and verify that at least one way exists of giving values to the other variables involved in that constraint that is consistent with v. If every value in a set of domains is supported in this way, the domains are said to be *arc consistent* (or *generalised arc consistent* if constraints involve more than two variables). Formal notions of consistency do not play a large part of this thesis; instead, we are interested in performing inference quickly, even if this means not achieving a particular theoretical level of consistency.

1.3.2 Search

Despite the range of inference techniques available to us, often no reasonable amount of filtering will be sufficient to find a solution, or to prove that none exists. In such a case we must guess: we pick some variable, and force it to take one of the values remaining in its domain. This assignment will hopefully allow further propagation. In the event that this is still not enough to find a solution, we repeat the process, giving a recursive search. Eventually,

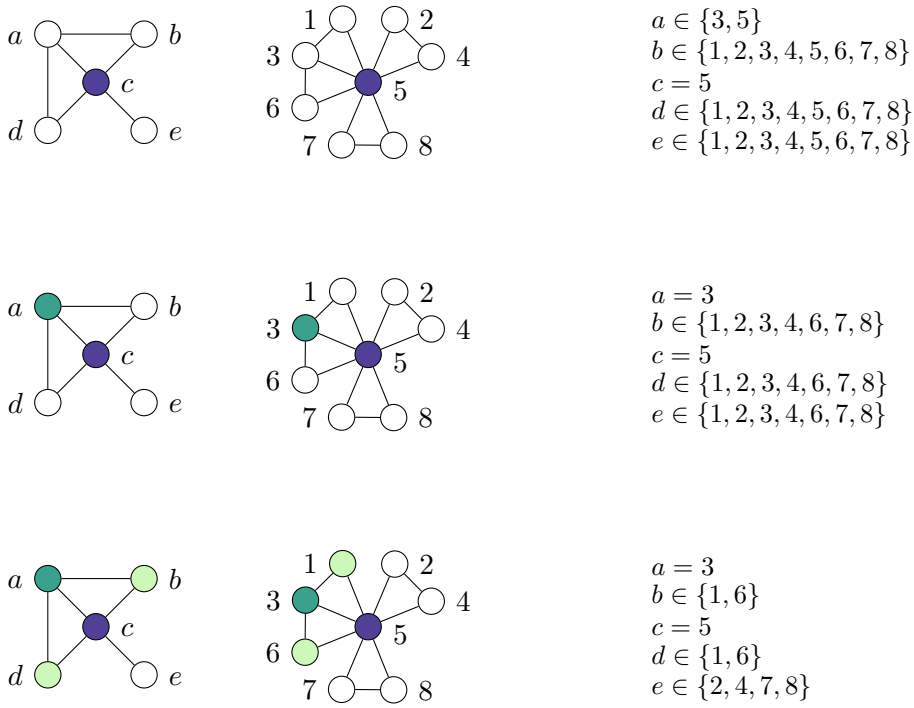


Figure 1.7: Initially, by counting neighbours, we see that the top left vertex a in the pattern can only be mapped to pattern vertices 3 or 5, and the central vertex c in the pattern can only be mapped to the central vertex 5 in the target. Secondly, since $c = 5$ is forced, no other pattern vertex can take the value 5; this leaves only 3 in the domain for a. Thirdly, we remove 3 from every other domain. Then, because we must map adjacent vertices to adjacent vertices, the bottom left and top right pattern vertices b and d are left with only two possible destinations, 1 and 6. We no longer have any forced assignments, but we *can* conclude that because the two variables b and d only have the two values 1 and 6 between them, the variable e could not take either of these values.

this may lead to a variable's domain becoming empty—in this case, we have made a mistake (or no solution exists), so we must backtrack and try another value for our most recently guessed variable instead. We could also run out of values for a variable when branching this way—then we must backtrack to an earlier guessed variable, or if we are on the first variable, then we have proved that no solution exists.

Interleaving search and basic constraint propagation over domains in this way is known as *forward checking*. If arc consistency is maintained at each level of search, the algorithm is known as *maintaining arc consistency*. Simpler algorithms which do not store domains at all, and which do not detect the lack of an available value for a variable until an assignment is attempted, are known as *conventional backtracking*.

When interleaving search and inference, the efficiency of the inference stage is extremely important. Determining a correct and principled order in which to perform inference for sets of constraints has been the subject of extensive research. For best performance, a fully general constraint propagation algorithm must determine not just when it is necessary to

propagate a constraint, but also when *not* to look at a constraint, to avoid wasted effort. This requires knowledge of whether individual propagators guarantee properties such as *idempotence* (does running the propagator twice consecutively always give the same results as running it just once?) and *monotonicity* (does the propagator guarantee that it will never eliminate fewer values when applied to reduced domains?), as well as having estimates of the relative execution costs of different propagators (Schulte and Stuckey, 2008; Schulte and Tack, 2009). Cohen, Jefferson, and Petrie (2016) and Tack (2009) give two different perspectives on the design of such systems. The algorithms we discuss do not require this level of generality: all of our constraints are known up-front, which allows us to avoid these complications and always use a particular predetermined processing order.

1.3.3 Heuristics

When branching during search, the choice of variable and value to guess first has a huge impact on the amount of searching required. Good general principles exist for variable selection: for example, picking the domain with the smallest number of values left first often works well (Haralick and Elliott, 1980), as does picking whichever domain is most constrained. This kind of rule is called a *heuristic*, since it is not guaranteed to give the best choice, but empirically tends to give good choices most of the time. Value selection can be more difficult—in graph problems, we usually end up talking about the number of neighbours a vertex has. Unfortunately, value selection heuristics for subgraph problems tend not to be particularly reliable, particularly when (as often happens) many vertices have the same number of neighbours.

1.3.4 Bounds

For optimisation problems such as maximum clique, an additional form of inference can come from a *bound* function. Suppose we want to find a largest possible clique in a graph. We can use inference and search to find a *feasible* solution (that is, any clique), which we call the *incumbent*. We then continue searching, but are only interested in finding cliques which are strictly larger than the one we have already found. To do this, we use the incumbent in a special kind of filtering which eliminates portions of the search space which we can prove cannot contain a better solution. Conceptually, each time we find a solution, we add a new constraint saying "the solution must be better than this new incumbent", and we then continue with search until no (better) solution exists.

We give an example of a bound in Figure 1.8. A *colouring* of a graph assigns a colour to each vertex, with the rule that adjacent vertices *must* be given different colours. If we can colour a graph using k colours, we know that it cannot contain a clique of more than k vertices (because each vertex in a clique must be given a different colour). However, as the second

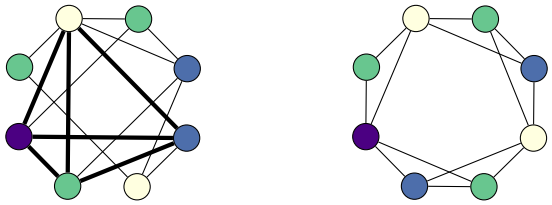


Figure 1.8: The relationship between cliques and colourings. The graph on the left can be coloured using four colours, and contains a clique of size four. The graph on the right also requires four colours, but contains no clique with more than three vertices.

graph of Figure 1.8 shows, this bound may not be tight, even if we produce a best-possible colouring. In fact, there are graphs where the gap between the clique and colour numbers is arbitrarily large (Mycielski, 1955).

Colouring, then, gives us a bound of how large a clique can be in the initial graph. But in Chapter 2 we go further. Suppose, during search, that we have already accepted three vertices, and rejected two others. We can then colour the subgraph given just by vertices which have neither been accepted or no rejected so far, and add the number of colours used to the three vertices accepted, to give us a dynamic bound.

1.3.5 Smart Versus Fast

> Complex methods to reduce the size of the search tree often do not lead to corresponding reductions in actual execution time, because of the additional work needed at each node.
>
> S. A. Cook and Mitchell (1996)

There is a trade-off to be made between the amount of propagation done, and the amount of search that must be performed. Even when strong filtering is theoretically capable of eliminating more values, it is not always worth trying as hard as possible to eliminate values through propagation: expensive propagation algorithms may not lead to additional deletions in practice, or if they do, those deletions may not substantially reduce the amount of search required. The right balance must also be found when selecting variable and value ordering heuristics—for example, rather than trying to find a vertex with most neighbours remaining in other variables (which must be calculated dynamically), it may be more profitable to simply select a vertex which had most neighbours at the start of search. Similarly, we could choose to put a lot of effort into obtaining very good bounds at the expense of making each step more expensive, and we can decide how much effort is spent pruning symmetries.

The effects of such trade-offs form a recurring theme throughout this thesis: efficiency in regularly-performed operations becomes particularly important when we are dealing with graphs with tens of thousands of vertices or hundreds of thousands of edges. For example, in Chapter 2, we introduce a new search heuristic which is stronger but too costly to be practical,

and then propose a cheaper surrogate with nearly all of the benefits and none of the overheads. Similarly, in Chapter 4, we introduce an inference rule, and show how to use laziness to avoid a costly initialisation step. In Chapter 5, we propose and evaluate a new all-different propagator which drops some consistency guarantees in return for much faster execution, and replace expensive but effective dynamic path calculations with a cheaper static alternative. And in Chapter 7, we compare two similar algorithms, one of which is slower but uses strong inference, and another which can carry out many more recursive calls per second but which explores a larger search space.

1.3.6 General Purpose Solvers

Constraint programming toolkits provide implementations of many common inference, heuristic, and search algorithms. We do not make heavy use of toolkits in this thesis, other than in preliminary experiments to reassure ourselves that more complex algorithms produce the same answers. These general-purpose solvers use data structures and algorithms which have not been tailored specifically for graph problems, and for every problem in this thesis we are able to select better alternatives that make our most common operations much cheaper to perform. Prior research has evaluated constraint programming approaches for the three main problems in this thesis, and in each case dedicated algorithms have already comprehensively beaten general-purpose solvers.

Nonetheless, we find constraint programming extremely valuable as a philosophy, to help with understanding why these algorithms work. For example, in Chapter 2, we use constraint programming concepts to explain why a branching rule in a special-purpose maximum clique algorithm is so effective in practice.

1.3.7 Microstructure

This kind of backtracking search is not the only way to solve a CSP. Another way, which we will use twice during this thesis (once explicitly, in Chapter 7, and once in disguise, in Chapter 4), is to encode the problem using its *microstructure* (Jégou, 1993). The microstructure encoding of a CSP is a way of representing a problem instance as a graph, such that a clique of a particular size corresponds to a solution. Microstructure is mostly studied in constraint programming for its theoretical properties (Cohen, Cooper, et al., 2012; Cohen, Jeavons, et al., 2006; Cooper, Jeavons, and Salamon, 2010; Jégou, 1993), but we will show that it is sometimes also useful as a practical solving technique.

1.4 Empirical Algorithmics

> Within theoretical computer science algorithms are usually studied within highly simplified models of computation and evaluated by metrics such as their asymptotic worst-case running time or their competitive ratio. These metrics can be indicative of how algorithms are likely to perform in practice, but they are not sufficiently accurate to predict actual performance. The situation can be improved by using models that take into account more details of system architecture and factors such as data movement and interprocessor communication, but even then considerable experimentation and fine-tuning is typically required to get the most out of a theoretical idea. Efforts must be made to ensure that promising algorithms discovered by the theory community are implemented, tested and refined to the point where they can be usefully applied in practice. This can only happen if the theory of computing community comes to recognize algorithm engineering—the experimental testing and tuning of algorithms—as integral to its mission.
>
> Aho, Johnson, Karp, Kosaraju, and McGeoch (1996)

> It is important to observe that, with the exception of planar graphs, none of the polynomial algorithms mentioned above has been implemented in software.
>
> Piperno (2008)

Since we are not looking to improve worst-case complexity, we need a different way of evaluating our algorithms. The ultimate aim of designing and implementing these algorithms is to solve actual problem instances. Thus we evaluate our algorithms and implementations by measuring how many of these instances we may solve, and how long it takes to solve them.

1.4.1 Implementation Notes

The algorithms introduced in this thesis are implemented in C++. Four factors motivated this decision: the need to write performance-competitive code, the availability of both low-level and medium-level implementation options for parallelism, having access to detailed performance measurements, and the author's familiarity with the language. All experiments are carried out on Linux systems—hardware availability forced this decision.

In this thesis, backtracking is always implemented by recursion, using the implicit compiler-generated stack. This has the advantages of simplicity and readability, as well as avoiding the need for dynamic memory allocation for some data structures (we discuss this further below). However, current C++ compilers generate code which requires the size of the stack to be decided before the program is launched. The default stack sizes on Linux systems is sometimes too small for larger problem instances, and it is sometimes necessary to

raise the size limit (using `ulimit -s` in the Bash shell, for example). A separate, explicitly-maintained stack would avoid this complication, at the expense of much harder to read code. Another potential advantage of explicit stacks is in reduced function call overheads, although the recursive functions in each of our algorithms involve sufficient work that this is not a measurable problem, and automatic inlining takes care of small helper functions.

1.4.2 Runtimes and Other Performance Metrics

Accurately measuring runtimes is not entirely straightforward: McGeoch (2012, Chapter 3) discusses this topic in depth. Fortunately, modern C++ provides us with access to a *steady clock*, which avoids most of the pitfalls associated with timing, even when dealing with threaded code on multi-core hardware.

Every measurement we give when using our own code in this thesis uses a raw steady clock, which is (supposed to be) guaranteed to be monotonically increasing, is unaffected by clock changes or adjustments, and which does not exhibit complications when threads migrate between processor cores. This clock limits us to millisecond resolutions (and two executions which take exactly the same amount of time can have reported runtimes differing by up to one millisecond as a result). As far as possible, we exclude file input times from measurements, since these can be extremely inconsistent, particularly on networked filesystems. In cases where naturally we would read in a file and perform non-trivial computations along the way to convert it to an internal in-memory format, we split this code into two parts and time only the latter.

When using other people's code, we do not always have the option of taking steady-clock measurements, either because we cannot modify the source code, or because the implementation is in a programming language which does not support accurate timing measurements. In particular, several of the solvers we use measure CPU time rather than real time. As McGeoch notes, some experimenters prefer this measurement, since it (approximately) excludes time spent when a process is not running. This is sometimes felt to be "more accurate", because it masks some of the complications of running on real hardware. However, measuring CPU time is completely inappropriate in a multi-threaded environment, since it excludes some (but not all) time spent by threads in a blocked state, and will hide lock contention.

Sometimes we will adopt alternative measures, such as search tree sizes (Bessiere, Zanuttini, and Fernàndez, 2004); however, such measurements will only be used to provide insight, and are not taken as directly indicative of performance.

1.4.3 Cumulative Plots, Scatter Plots, and Heatmaps

Having measured runtimes, how do we interpret and report them? Sometimes we will present a large table of results, showing runtimes and other measurements for every problem instance

individually. For example, we do this in Chapters 2 and 3, where the set of benchmark instances is small enough to make this feasible, and where algorithm behaviour on individual instances illuminates interesting behaviour. A further benefit of such a table is that if search tree size measurements are included, then independent implementations can verify that they produce the same-sized search tree as an indication of correctness (Korf, 2014).

However, producing a table for every instance is not environmentally friendly if we have many thousands of instances, as we do in Chapters 5 and 7. We therefore need ways of summarising data. Taking a mean or median runtime hides many interesting details, and when a timeout is used, there is a further complication of how to handle instances which do not complete with one algorithm. Instead, we make widespread use of *empirical cumulative distribution function* plots, which we simply call *cumulative* plots. Such a plot has time (or some other measure of size) along the x-axis, and one plotted line for each algorithm being measured. The y value at point x is the number of instances whose runtime is less than or equal to x. Usually we use a log scale on the x axis, and scale the y axis linearly from zero to the total number of instances in the dataset. An example of such a plot can be seen in Figure 5.2 on page 135.

Note that some publications use the alternative choice of x and y axes, treating "number of instances solved" as the dependent variable. Such plots are sometimes called *cactus* plots.

Because cumulative plots are the main way of comparing algorithm performance in this thesis, we emphasise the following **extremely important point which must not be misunderstood**: the cumulative plot value is *not* the total number of instances which can be solved in time x. Each instance is considered individually, and we do not in any way add together runtimes from more than one instance. Thus, these plots give the number of successes we would encounter if every single instance in the dataset is run independently, each with its own timeout of x.

Usually when reading a cumulative plot, we look at which line is highest up on the plot to see which algorithm is best: the vertical difference between two lines shows how many more instances one algorithm can solve than the other, for a given choice of timeout. Comparing the horizontal difference between two lines can give an indication of how many times faster one algorithm is than another, bearing in mind that the sets of instances solved by the two algorithms may be complete different. Cumulative plots only show aggregate performance, and tell us nothing about performance on individual instances. In an extreme case, algorithm A could be extremely good at instances from problem family A and poor at instances from problem family B, whilst algorithm B could show the opposite behaviour. In this case, a cumulative plot will show that algorithm A is the clear winner if there are more instances from family A than family B in the test set, or that algorithm B is best if the opposite holds.

To avoid such a misleading conclusion, we also make use of scatter plots, to compare two algorithms on an instance by instance basis. As far as possible, we follow the convention

in this thesis that "points below the diagonal line are better" for whichever algorithm or technique is being proposed. Figure 5.3 on page 135 gives an example, which presents same the results as Figure 5.2 but with a different perspective on the data. By convention, points drawn along the top or right of a plot represent instances which timed out with one algorithm but not the other (so, for example, in the left-hand plot of Figure 5.3 on page 135, we see many instances which timed out with VF2 but not our algorithm, and one which timed out with our algorithm but not with VF2).

Finally, we sometimes use heatmaps, which we find to be more readable than three dimensional plots. A heatmap uses the colour at the point (x, y) to show the value of a function at that point. This function can either be a direct measurement, such as in Figure 6.2 on page 149, or to show the density of points around a location when a scatter plot would be hard to read due to there being many close-together data points, as in Figure 7.3 on page 178. As far as possible, we have selected colours which emphasise the behaviour we wish to illustrate. For example, the top row of Figure 6.2 uses the lightest colour for the point where the function takes the value 0.5, and diverging darker colours for 0 and 1, because the half-way point is where an interesting phenomenon occurs. The subsequent rows of this figure instead use darker colours to represent longer runtimes, and lighter colours to represent shorter runtimes. We refer to Borland and Taylor II (2007) and Janert (2009) for more details on the effects of colour palettes; the schemes in this thesis were created using the Chroma.js Colour Scale Helper (Aisch, 2013).

1.4.4 Instance Selection

Evaluating an implementation using problem instances, rather than by considering its worst-case complexity, runs the risk of drawing overly broad conclusions based upon a limited number of inputs. To reduce bias (at least, of the kind we could introduce ourselves), where possible, we will be using other people's choices of instances for evaluations. Fortunately, because the problems we consider have real-world applications, we often have access to collections of real-world problem instances for evaluation. The flexibility of these problems mean we are usually able to use datasets with different characteristics from multiple sources, which further reduces the chances of overfitting.

We will also be using randomly generated instances. The main advantage of this approach is that we can generate large numbers of instances—this again reduces the chances of misinterpreting the significance of results. Another advantage is that certain interesting but rare phenomena are most easily visible when we can spot outliers after having looked at hundreds or thousands of instances which *should* give very similar performance: an early observation of rare exceptionally hard problem instances by Gent and Walsh (1994) has had a large impact on the design of subsequent algorithms, leading to improved performance on all kinds of instances.

A potential weakness of randomly generated instances is that real-world graphs rarely *look* random—Gent and Walsh (1995) give an example of an exam timetabling graph which contains an unexpected ten-vertex clique, which would be extremely unlikely in a randomly generated graph with a similar order and density. The complete lack of structure in random graphs means that some inference techniques which appear to be completely useless on random graphs are extremely helpful on the kinds of problem instances which people actually want to solve. McGeoch (2012, Chapter 2) contains further discussion on instance selection.

1.4.5 Are Hard Problems Hard?

A further interesting property of randomly generated instances is that, depending upon the parameters used, they can give either overly optimistic or overly pessimistic views of difficulty and scalability. Both of these extremes are explored in Chapter 6. This chapter begins with a close look at a suite of random instances which are often used to support the claim that subgraph isomorphism algorithms can easily scale to patterns of many hundreds and targets of many thousands of vertices. By using a slightly different random model, we create instances with thirty and one hundred and fifty vertices respectively which cannot be solved with a reasonable timeout by any solver.

The existence of *really hard* instances is perhaps reassuring. It is widely believed that algorithms for NP-complete problems necessarily have exponential worst-case complexity (or at least, the best algorithms we know about have exponential worst-case behaviour), yet it is rare to encounter anything approaching these worst-case bounds in practice. The *phase transition* phenomena discussed in Chapters 2 and 6 give us a predictable way of generating families of problem instances which force the worst case to occur.

This is not purely of theoretical interest. Although real-world instances are usually very different from the hardest random instances, understanding algorithm behaviour on hard random instances helps us to improve algorithms, and also systems built upon these algorithms. This is the topic of the second half of Chapter 6, which re-evaluates a line of research in graph database systems using an improved understanding of when subgraph isomorphism is hard. This understanding highlights a serious design problem with a popular technique known as “filter / verify”.

1.5 Other Approaches to Hardness

The focus in this thesis is upon tackling hard problems through *practical* algorithms—that is, algorithms which work well in practice on relevant problem instances, as determined by scientific experiments on an implementation, as opposed to through mathematical analysis of worst-case or average-case complexity. We now briefly discuss some common alternatives.

1.5.1 Fixed-Parameter Tractability

> The algorithms resulting from this theory are most unlikely to be useful in practice.
>
> McKay and Piperno (2014)

A different perspective on per-instance hardness comes from *fixed-parameter tractability* theory. Fixed-parameter tractability gives a way of limiting the exponential behaviour of an algorithm by restricting properties of the inputs. For example, in Chapter 7 we will see that there are special cases where a connected common subgraph problem is hard in general, but if the maximum degree of each vertex in a graph is restricted to some constant k, then the problem is of polynomial complexity (but exponentially difficult in k).

Although fixed-parameter tractability receives a lot of theoretical attention, its practical significance for subgraph problems has not been established. Piperno (2008) notes that almost none of the special polynomial cases for graph isomorphism have ever been implemented, and McKay and Piperno (2014) argue that there is little point in doing so. Sharmin (2014, Chapter 10) did implement and evaluate a fixed-parameter tractable algorithm for maximum clique: her results are generally exceedingly poor, except for one family of crafted benchmark instances, where they are orders of magnitude better than standard solvers. Another success is claimed by Akiba and Iwata (2016) for certain kinds of vertex cover problems; a closer look by Strash (2016) shows the need for caution, by demonstrating that the actual benefits came from input preprocessing rather than from exploiting fixed-parameter tractability during search. In any case, such theories do not appear ready to explain the behaviour we witness in Chapters 2 and 6.

1.5.2 Approximation Algorithms and Heuristics

What if, instead of requiring a maximum clique or maximum common subgraph, we instead only want to find a large clique or common subgraph? Two very different approaches to this problem are *approximation algorithms* and *heuristics*. An approximation algorithm provides an answer which is guaranteed not to be more than a certain ratio below optimal; unless $\mathsf{P} = \mathsf{NP}$, no such algorithm exists for the maximum clique problem (Zuckerman, 2006), which rules out this kind of approach for general subgraph problems.

Heuristic approaches, in contrast, provide no guarantees beyond empirical evidence that they tend to give good solutions in practice. To avoid confusing with variable- and value-ordering heuristics, we refer to such approaches as *inexact*. Although the main focus of this thesis is in guaranteeing optimal solutions, in Chapter 2 we discuss the benefits of augmenting an exact solver with solutions from inexact approaches.

1.6 Exploiting Parallel Hardware

> In my future work, I will focus more attention on applying parallelism to NP-complete problems. Somebody with a very severe theoretical point of view could say, "That's hopeless, you can never reduce the run time from exponential to polynomial by throwing processors at a problem, unless you have an exponential number of processors." On the other hand, even though you may never be able to go from exponential to polynomial, it's also clear that there is tremendous scope for parallelism on those problems, and parallelism may really help us curb combinatorial explosions.
>
> Richard M. Karp, interviewed by Frenkel (1986)

> To me, it looks more or less like the hardware designers have run out of ideas, and that they're trying to pass the blame for the future demise of Moore's Law to the software writers by giving us machines that work faster only on a few key benchmarks! I won't be surprised at all if the whole multithreading idea turns out to be a flop [...]. Let me put it this way: During the past 50 years, I've written well over a thousand programs, many of which have substantial size. I can't think of even five of those programs that would have been enhanced noticeably by parallelism or multithreading.
>
> Donald E. Knuth, interviewed by Binstock (2008)

Processor clock speeds are no longer increasing substantially. To exploit modern hardware to its full potential, programs must be able to make good use of vector parallelism, multiple cores, and cache, which requires major changes to how algorithms are designed and implemented (Sutter, 2005; Sutter and Larus, 2005).

For the "in parallel" part of the title of this thesis, we will be taking good subgraph algorithms and improving them to exploit two kinds of parallelism. The first is vector or "single instruction multiple data" parallelism, where bitsets are used to carry out the same operation on multiple vertices simultaneously. The second is shared-memory thread parallelism to exploit multiple cores, which we use both for parallel preprocessing, and for parallel search.

There are two other potential sources of parallelism from modern hardware, which do not play a significant role in this thesis—we could use more than one computer at once, and communicate using message passing over a network, or we could use special parallel hardware such as graphics processing units. The former is left as future work, due to the additional complexity of programming for these systems; the latter can give large benefits to throughput-oriented floating point computations, but are not designed with constraint-based search algorithms in mind.

1.6.1 Bit Parallelism

> We believe, however, that bit parallel optimization is not just an implementation trick but has a full right to exist as an independent discipline.
>
> San Segundo, Rodríguez-Losada, Galán, Matía, and Jiménez (2007)

The use of bit-parallelism to accelerate graph algorithms has a long history. For example, Levi (1973) used it in a maximum common subgraph algorithm, and San Segundo, Rodríguez-Losada, Galán, et al. (2007) started a long chain of research into bit-parallel maximum clique algorithms which we review in Chapter 2. With modern hardware, exploiting bit-parallelism is becoming increasingly beneficial.

Bitset representations allow us to operate on 64 (or even more) vertices in a single instruction. For example, taking the intersection of two sets of vertices represented using bitsets is a simple bitwise-and operation upon two integers. When working with bitsets in algorithms, we use set notation, so $A \cap B$ means "perform a bitwise-and operation on bitsets A and B, corresponding to the set intersection operation".

To work with larger sets, we use a flat array of integers and a simple loop. The maximum size of every bitset we use in this thesis is known in advance (usually it is equal to the order of the input graph, sometimes with a small number of "extra" bits for wildcard values). Because bitsets are performance-critical in some places (such as the two nested loops in the `colourOrder` function in Algorithm 2.1), it can be beneficial to compile multiple copies of performance-critical functions to work with bitsets of different array lengths, before switching to dynamic allocation for larger arrays. This can be done automatically using C++ templates, without affecting the readability of the body of the function: the bitset data type can be parameterised by the number of words it contains, and can transparently either use a static array or a dynamic array as appropriate.

At best, bit-parallelism gives a constant factor improvement, which is completely ignored by abstract measures of complexity. Sometimes the theoretical behaviour is even worse—for example, finding the first true bit in a bitset is an $O(m)$ operation, where m is the potential number of bits which could be in the set, whilst finding the leftmost element of a tree is a $O(\log n)$ operation, where n is the number of elements actually present.[3] However, on modern processors there is a dedicated hardware instruction for finding the first set bit in an integer, and in practice bitsets can remain faster than tree or hash sets even when m is many thousands (corresponding to the number of vertices in the largest graphs we consider).

There is also a dedicated hardware instruction for finding the population count of an integer (that is, the number of bits in its binary representation which are one). This corresponds to the cardinality of a bitset, and in algorithms we write $|B|$ to mean "calculate the cardinality of the

[3] Complicating matters, the C++ standard requires both this operation and iteration to be $O(1)$ on all containers, so C++ ordered sets are typically implemented with a combined linked list and balanced tree.

bitset B, using population count instructions". Again, this operation has worse worst-case complexity than for a tree when the set is sparse, but in practice is much faster. Tarhio, Holub, and Giaquinta (2016) provide an interesting perspective on the merits of algorithmic improvements versus hardware instructions.

An additional advantage of bitset parallelism is that the associated data structures are very memory- and cache-friendly. Memory latency is not improving significantly on modern hardware: "first word" latency has not changed between the first DDR memory (c. 2000) and current DDR4 memory. In contrast, memory bandwidth has risen by more than an order of magnitude in the same period, even before taking the increasing numbers of memory channels available into account. These trends should influence algorithm design and implementation: Stroustrup (2012) gives an example of deleting elements from a linked list versus a flat vector, where the improved theoretical complexity of using a list gives worse practical performance on modern hardware when using fewer than half a million elements due to differing memory access patterns. Stroustrup's three initial recommendations for designing software are to avoid storing data unnecessarily, to keep data compact, and to access memory in a predicable manner: in cases where they can be used, bitsets follow the second and third recommendations perfectly.

We make use of bitsets throughout this thesis: other than one exception in Chapter 7, every best-performing algorithm in our experiments exploits bit-parallelism in some way. Our "default" data structure for representing a graph is an adjacency matrix using bit vectors (although in some cases we convert between representations for particular operations), and we will also discuss using bitsets when we must deal with additional data such as labels or connectivity information.

1.6.2 Thread-Parallel Propagation and Preprocessing

Some of the algorithms we introduce and study spend a substantial amount of time in a preprocessing stage. This preprocessing usually involves iterating over every vertex in an input graph at least once. Parallelising such a loop is usually routine, and so we do not spend much time discussing it, beyond verifying that it is beneficial. We refer to McCool, Reinders, and Robison (2012) for an overview of structured parallel programming in general (which is sufficient to cover the techniques we use for parallel preprocessing, but not parallel search) and to Williams (2012) for C++-specific issues.

There could also be a limited amount of potential for carrying out inference in parallel by propagating different constraints simultaneously (Nguyen and Deville, 1998). We make use of bit-parallelism for inference, which could be seen as a way of propagating many binary constraints simultaneously. Other than this, we do not make use of parallel propagation—none of our algorithms use a constraint queue, since we are always able to determine a static propagation order at design time, and the granularity of our propagation is typically much

too fine for thread parallelism. Even if we could solve the granularity problem, there are also theoretical limitations to such an approach (Kasif, 1990).

1.6.3 Thread-Parallel Search

Although parallel preprocessing can help, on difficult problem instances, most of the time is spent doing search. Since we are not making use of multi-core parallelism for propagation, we must identify a different source of parallelism to be able to make good use of modern processors. We do this by parallelising the search process—this is typically the largest source of improvement that we are able to get from parallelism. This topic is introduced in Chapter 3.

1.6.4 Mutexes, Atomics, and Queues

In a multi-threaded environment, care must be taken when multiple threads access the same variable if at least one of these threads may be writing to it (C++ mostly guarantees that simultaneous *reads* are safe, both for primitives and for standard library data structures). The simplest way of allowing concurrent reads and writes is to protect the variable with a *mutex*. Any thread may *lock* the mutex, perform some work, and then unlock the mutex. Only one lock per mutex can be held at a time, and any other thread attempting to lock that mutex will block until it is unlocked. When concurrent reads are extremely common and writes rare, we instead make use of an *atomic*. Atomics are special variables which support only a small number of operations, but which can be read from and written to simultaneously. Williams (2012) explains these concepts in more detail.

We also make use of queues. The term *queue* in this thesis always refers to an unbounded thread-safe queue, where items can be enqueued and dequeued concurrently, and where dequeue operations may block.

1.6.5 Measuring Parallel Improvements

Naïvely, one might think that given ten processing cores, we should be able to solve a problem ten times faster than if we only had one processing core. We might anticipate some overheads: perhaps we should only reasonably expect things to be nine times faster with ten processing cores? Unfortunately matters are not this simple even for ideal algorithms, and with the state-of-the-art subgraph algorithms in this thesis the situation becomes much more complicated.

By a *speedup*, we mean "how many times faster is a parallel algorithm?" (recalling our earlier discussion on measuring times). An *absolute* speedup is a speedup over a tuned sequential algorithm; a *relative* speedup is over a parallel algorithm run with a single thread (or with a smaller number of threads). In this thesis, whenever we refer to a speedup over a

sequential algorithm, we mean an absolute speedup over a sequential algorithm which we have tuned with the same care given to parallel code—that is, we are measuring genuine improvements over the best we could do sequentially. When we give a speedup from, say, increasing from 8 to 16 threads, this is obviously a relative speedup. A *linear* speedup is a speedup of (approximately) n from n threads. As Hamadi and Wintersteiger (2012) note, there are myriad reasons why linear speedups are an unrealistic goal. Amongst the most relevant for this thesis are:

- The increased number of cores in modern Intel and AMD processors do not come with a corresponding increase in memory bandwidth. If our algorithms are constrained by accesses to either main memory or shared caches, then multi-core hardware does not give a linear increase in the critical resource.

- Memory bandwidth is not the only hardware complication. Hyper-threading is a feature on Intel processors where each "real" core is presented as two cores, with control switching between cores at very high granularity (such as when there is a stall for a memory access). For some applications hyper-threading gives an additional benefit of twenty to thirty percent, whilst for others it does nothing. Additionally, sequential execution usually proceeds more slowly if two threads are running on the same "real" core. We leave hyper-threading enabled when measuring absolute parallel performance, since our goal is to see the benefits of making use of whatever features our hardware offers. We disable hyper-threading when we are trying to get a careful picture of scalability effects.

- Another hardware feature, referred to by Intel as "Turbo Boost" and by AMD as "Turbo Core", causes the processor clock speed to increase when operating below power, current and temperature limits. This feature is unlikely to trigger when parallel code is running, but sometimes gives a benefit for sequential code. We leave this feature disabled, to avoid having to account for the effects of the Scottish summer when performing experiments.

- Often we must change the sequential algorithm to allow for the introduction of parallelism, taking a penalty to the starting point in the hopes of more than recovering the lost performance through parallelism. For example, in most of our algorithms we have to make copies of certain data structures to allow for the possibility of parallelism (even if that possibility does not occur), whereas sequentially these data structures could be updated in-place using reversible operations. In no cases do we "cheat" and deliberately design our sequential algorithms with unnecessary copying—this plays a particularly large role for the final algorithm discussed in Chapter 7.

- Sometimes we are only parallelising part of the execution. Suppose an algorithm proceeds in two parts, which sequentially take one and three seconds respectively. If we only

parallelise the second part, the highest speedup we could possibly get with any number of processors is four.

With the final point in mind, ordinarily a work on parallelism would next talk about various laws named after distinguished researchers. However, these laws are not applicable in a speculative parallelism setting, and so we pointedly ignore them; Chapter 3 discusses more suitable alternatives.

Indeed, we believe that parallelism laws, together with an overly zealous focus upon linear speedups, have stood in the way of progress. Multi-core hardware is now the standard even on low-power laptops, and new server hardware typically has many tens of cores. Complaining that we cannot make *perfect* use of this parallelism should not stop us from making *some* use of it. Our first measure of success, therefore, is simply whether parallelism is *beneficial*. That is, can we see that our best parallel algorithm is clearly better than the best sequential algorithm, preferably without needing to resort to any statistical tests more complex than a cumulative plot?

1.6.6 Anomalies and Risk-Free, Reproducible Parallel Search

> Our experimental results indicate that such anomalous behaviour will be rarely witnessed in practice
>
> Lai and Sahni (1984)

> In practice the above described anomalies, especially detrimental anomalies, are so common that we can regard them as facts of life. The word 'anomaly' is thus not very appropriate, and we would rather think of them as *limited speedup*, *super speedup*, and *slowdown* respectively. Even though we do not agree with the terminology in the literature, we will adhere to it in the remainder of this thesis in order to prevent confusion.
>
> Trienekens (1990)

A further complication is that parallel search is *speculative*: it works by using parallelism to pre-compute results which *might* be used in the future. There is therefore no expectation of a linear speedup at all even under ideal circumstances. Instead, we could see no speedup, a superlinear speedup (that is, a speedup of more than n from n cores), or even an absolute slowdown. Such behaviours are called *anomalies* in the literature, which is unfortunate, since they are not in any way anomalous. We explain this topic fully in Chapter 3. For now, we simply introduce three further properties which we desire from parallel search:

- Parallelism should be *risk-free*. We would like to guarantee that parallel runtimes will not be (substantially) worse than sequential runtimes. A precise definition of this property is

difficult to formulate: practically, we do not mind small amounts of overhead, or an increase in runtimes from one millisecond to ten millisecond on trivial instances. However, we would prefer that the amount of "work done" does not increase by a larger factor than the number of cores used, and would like to guarantee that we will never introduce exponential slowdowns.

- We would also like our parallel algorithms to be *scalable*, in the following sense. If we increase the number of cores available to us, we would like runtimes to decrease, or at least not increase (again, ignoring small amounts of overhead).

- Finally, parallel runtimes should be *reproducible*: if we run the same implementation on the same instance with the same hardware more than once, we should get similar runtimes.

Although each of these properties may seem obvious and reasonable, in Chapter 3 we observe that no existing constraint programming mechanism for parallel search provides all of these guarantees simultaneously. A central contribution of this thesis is the design and implementation of a work splitting technique which is risk-free, scalable, and reproducible, both in theory and in practice. This technique is also clearly beneficial, thanks to a key observation: whilst work balance can be a problem, for each kind of subgraph problem we discuss, there is an interaction between parallelism and value-ordering heuristics which must be understood and controlled to obtain consistently good results.

1.7 Overview of the Thesis

The remainder of this thesis is structured as follows.

In Chapter 2 we study a bit-parallel algorithm for the maximum clique problem. We look at and explain its behaviour on random graphs and real-world problem instances. Next, we analyse the algorithm in more detail, using analogies with constraint programming to explain why a widely used but poorly understood choice of branching order is so successful. Parts of this chapter have previously been published by McCreesh and Prosser (2014b) as "Reducing the Branching in a Branch and Bound Algorithm for the Maximum Clique Problem".

Then, in Chapter 3 we compare and contrast several ways of parallelising this algorithm. This chapter introduces parallel search, and highlights the connection between performance and interactions between work splitting and variable-ordering heuristics which inform the remainder of the thesis. This chapter is extended from McCreesh and Prosser (2015c), "The Shape of the Search Tree for the Maximum Clique Problem and the Implications for Parallel Branch and Bound".

In Chapter 4 we discuss three variants of the maximum clique problem. Firstly, we look at the maximum k-clique problem, and investigate whether a well-known but previously

unimplemented reduction to the maximum clique problem (which is, in effect, a microstructure encoding) leads to a workable algorithm in practice. Secondly, we give a new approach to the maximum labelled clique problem, which is many orders of magnitude better than previously published results. Finally, we present the first algorithm for the maximum balanced induced biclique problem, which includes a static kind of symmetry breaking. Much of this chapter is derived from three publications: McCreesh and Prosser (2016), "Finding Maximum k-Cliques Faster Using Lazy Global Domination", McCreesh and Prosser (2015b), "A parallel branch and bound algorithm for the maximum labelled clique problem", and McCreesh and Prosser (2014a), "An Exact Branch and Bound Algorithm with Symmetry Breaking for the Maximum Balanced Induced Biclique Problem".

Having covered the maximum clique problem, we then move on to subgraph isomorphism problems. Chapter 5 introduces a novel approach to the problem, exploiting bit-parallelism to achieve both strong inference and fast propagation; this algorithm is derived from an earlier publication, McCreesh and Prosser (2015a), "A Parallel, Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs". Experiments over a large set of problem instances introduced in Kotthoff, McCreesh, and Solnon (2016), "Portfolios of Subgraph Isomorphism Algorithms", show that this new algorithm is the single strongest solver. The algorithm is then extended to make use of thread parallelism for preprocessing and search, drawing from the lessons of Chapter 3. Further experiments confirm that thread parallelism is beneficial, risk-free, scalable, and reproducible.

Despite subgraph isomorphism being NP-complete, the algorithms compared in Chapter 5 can work comfortably with instances with many hundreds of vertices in the pattern graph, and up to six thousand vertices in the target graph. Chapter 6 begins by taking a critical look at the problem instances used to reach this conclusion, and introduces a new method for creating "really hard" subgraph isomorphism instances. This method also helps to justify the variable- and value-ordering heuristics we selected. This first half of the chapter is derived from McCreesh, Prosser, and Trimble (2016), "Heuristics and Really Hard Instances for Subgraph Isomorphism Problems".

Chapter 6 continues by extending this method to labelled graphs, where we witness unusually poor behaviour from Cordella et al.'s (2004) widely used VF2 subgraph isomorphism algorithm. Analysing this behaviour in more detail uncovers a fundamental flaw in the design of graph database systems, and casts doubt on fifteen years of research into indexing and a technique known as *filter / verify*.

The final part of the thesis looks at maximum common subgraph problems. In Chapter 7 we attempt to determine what "the best" maximum common subgraph algorithm is, comparing a constraint programming approach to a reduction to maximum clique. We also consider the connected version of the problem, introducing a clique-inspired algorithm which is extremely strong on edge-labelled instances. These experiments were published as McCreesh, Ndiaye,

et al. (2016), “Clique and Constraint Models for Maximum Common (Connected) Subgraph Problems”. This chapter then describes two new approaches to the problem: the first was introduced in Hoffmann, McCreesh, and Reilly (2017), “Between Subgraph Isomorphism and Maximum Common Subgraph”, and the second is forthcoming work due to James Trimble. Depending upon the instances being considered, either of these two new approaches or the clique approach could be the best choice of solver. We therefore finish the chapter by parallelising all three.

Finally, in Chapter 8 we conclude, recapping our work, and giving broader perspectives on empirical complexity, on constraint programming, and on programming language features for parallelism.

Chapter 2

The Maximum Clique Problem

We begin with the simplest kind of subgraph problem: finding a clique in a graph. Recall that a clique is a subset of vertices, each of which is adjacent to every other vertex in the subset. We are usually interested in finding a largest possible clique in a given graph (as in Figure 2.1), which is called the maximum clique problem. The size of a maximum clique is denoted ω.

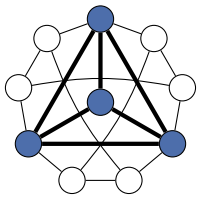


Figure 2.1: A graph, with its unique maximum clique of four vertices highlighted.

In this chapter we look at a bit-parallel algorithm for the maximum clique problem which is designed for use on dense graphs. We begin by discussing the origins of this algorithm, and reviewing related work. We then conduct experiments on random graphs: although solving the maximum clique problem on random graphs is not generally useful in practice, it does provide us with insights into the behaviour of the algorithm. We then consider applications and real problem instances.

Next, we analyse the algorithm in more detail. In particular, we explain why a widely used but poorly understood choice of branching order is so successful, and use our new understanding to further improve the algorithm. Finally, we discuss some open questions.

Parts of this chapter have previously been published by McCreesh and Prosser (2014b) as “Reducing the Branching in a Branch and Bound Algorithm for the Maximum Clique Problem”. The description of Algorithm 2.1 is based upon McCreesh and Prosser (2016), “Finding Maximum k-Cliques Faster Using Lazy Global Domination”.

2.1 Algorithms for the Maximum Clique Problem

Exact algorithms for clique-finding date back at least as far as Harary and Ross (1957). Early approaches with computational experiments include a simple backtracking (but non-recursive, due to the use of Fortran 77) search algorithm by Carraghan and Pardalos (1990) which uses the number of remaining vertices as a bound, and a quadratic approach by Pardalos and

Rodgers (1992); both papers use experiments on randomly generated graphs. We refer to Pardalos and Xue (1994) for more ancient history.

The maximum clique problem was one of three selected for the Second DIMACS Implementation Challenge (Johnson and Trick, 1993), along with graph colouring and Boolean satisfiability. The challenge lead to new exact algorithms (Goldberg and Rivenburgh, 1993; Mannino and Sassano, 1993) and quasi-exact algorithms (Balas and Niehaus, 1993), exact integer programming approaches (Mercure et al., 1993; Pataki et al., 1993), and a number of heuristic techniques which we do not cover in this thesis. The challenge introduced a suite of benchmark instances, which we discuss below. Unfortunately, the challenge also introduced a rescaling mechanism, where results are taken from one paper and then rescaled according to relative runtimes obtained by running a standard program on the experimenter's computer; this unreliable and unscientific technique is still in common use, which makes it particularly hard to compare reported results.

Subsequently, better results were obtained by Wood (1997) using colouring as a bound and a degree-based branching strategy, and Östergård (2002) using Russian dolls search. Fahle (2002) and Régin (2003) both investigated explicit constraint programming approaches, the latter with a matching-based bound. All use a subset of the DIMACS instances to evaluate their approaches.

However, the most promising line of research is based upon a series of exact branch and bound algorithms using a greedy graph colouring both as a bound and a branching strategy (Tomita, 2017; Tomita and Kameda, 2007; Tomita and Seki, 2003; Tomita, Sutani, et al., 2010; Tomita, Yoshida, et al., 2016). A study by Prosser (2012) reviewed, reimplemented, and compared these algorithms, and confirmed the benefits of bit-parallel versions of these algorithms proposed by San Segundo, Matía, et al. (2013) and San Segundo, Rodríguez-Losada, and Jiménez (2011) by a direct comparison rather than by rescaling results. We will now describe the core concepts underlying these algorithms; further enhancements are discussed in Sections 2.4 and 2.5.

2.1.1 A Basic Colour-Based Branch and Bound Algorithm

In Algorithm 2.1 we describe a basic maximum clique algorithm which uses a colour-based branch and bound strategy. The algorithm we describe is the variant Prosser (2012) calls "MCSa1": this is Tomita, Sutani, et al.'s (2010) "MCS" algorithm, without the colour repair step, and using a degree-based ordering at the top of search.

Colouring The algorithm we discuss uses branch and bound, with a greedy graph colouring as both the bound and an ordering strategy. Recall that a *colouring* of a graph is an assignment of colours to vertices, such that adjacent vertices are given different colours; if we can colour

Algorithm 2.1: Solving the maximum clique problem.

```
1  maxClique :: (Graph G) → Vertex Set
2  begin
3      permute G into non-increasing degree order
4      global incumbent ← ∅
5      expand(∅, V(G))
6      return incumbent (unpermuted)
7  expand :: (Vertex Set solution, Vertex Set remaining)
8  begin
9      (order, bounds) ← colourOrder(remaining)
10     for i ← |remaining| downto 1 do
11         if |solution| + bounds[i] ≤ |incumbent| then return
12         v ← order[i]
13         solution′ ← solution + v
14         remaining′ ← remaining ∩ N(G, v)
15         if remaining′ ≠ ∅ then expand(solution′, remaining′)
16         else if |solution′| > |incumbent| then incumbent ← solution′
17         remaining ← remaining − v
18 colourOrder :: (Vertex Set remaining) → (Vertex Array, Int Array)
19 begin
20     (order, bounds) ← ([], [])
21     uncoloured ← remaining
22     currentColour ← 1
23     while uncoloured ≠ ∅ do
24         colourable ← uncoloured
25         while colourable ≠ ∅ do
26             v ← the first vertex of colourable
27             append v to order
28             append currentColour to bounds
29             uncoloured ← uncoloured − v
30             colourable ← colourable \ N(G, v)
31         currentColour ← currentColour + 1
32     return (order, bounds)
```

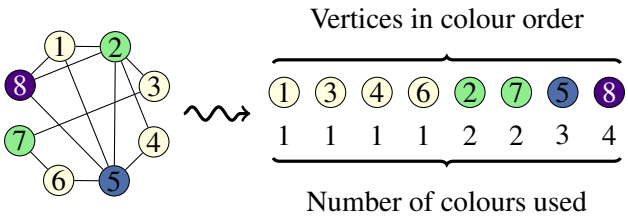


Figure 2.2: The graph on the left has been coloured greedily, using four colours: vertices 1, 3, 4 then 6 were given the first colour, then vertices 2 then 7 were given the second colour, then vertex 5 was given the third colour, and vertex 8 the fourth colour. The *order* array, on top, contains the vertices in the order they were coloured; the ith entry of the *bounds* array, below, contains the number of colours used to colour the first i vertices of *order*.

a graph using c colours, then the graph cannot contain a clique of size greater than c (each vertex in a clique must be given a different colour).

Obtaining a minimal colouring is NP-hard, but we may create a greedy colouring in polynomial time. This is done by the `colourOrder` routine: we start the first colour (line 22), and while there are uncoloured vertices remaining (line 23), we try to give each vertex in turn the current colour (lines 24 to 30). When we cannot colour any further vertices, we start a new colour (line 31).

The distinguishing feature of these algorithms is that they use a constructive colouring as more than just a bound—the `colourOrder` routine does not just return the number of colours used. Instead, it returns a pair of arrays, $order$ and $bounds$. The $order$ array contains vertices, in the order in which they were coloured. The ith entry of the $bounds$ array contains the colour number used for the ith vertex in $order$. We illustrate this in Figure 2.2. Crucially, $bounds$ is non-decreasing (i.e. $bounds[i+1] \geq bounds[i]$), and for any i we may colour the subgraph induced by the first i vertices of $order$ using $bounds[i]$ colours.

An alternative perspective is that `colourOrder` returns a list of *colour classes* in the order they were created, where each colour class is a list of vertices in the order they were given that colour. Although the use of arrays is critical for performance reasons, for descriptive purposes we will sometimes treat the return type as a list of lists.

The order in which vertices are selected for colouring can have a large effect upon performance. Various initial vertex orderings have been considered for the maximum clique problem. As a starting point, we colour vertices in a static non-increasing degree order, by permuting the graph at the top of search (line 3); we discuss this more below.

Branching and recursing We may now describe the main recursive part of the algorithm. If v is a vertex, then a clique in G either contains only v and vertices adjacent to v, or does not contain v. This allows us to grow cliques by repeatedly picking a vertex, and branching upon whether or not to include it. Our growing clique is stored in the variable $solution$, which is initially empty (line 5). We also track which vertices may still be added to $solution$ in the variable $remaining$, which initially contains every vertex (line 5). The `expand` procedure picks a vertex v (line 12), then considers adding v to $solution$ (lines 13 to 15): we create a new $remaining'$ from $remaining$ (line 14) by rejecting vertices which are not adjacent to v (and thus every vertex in $remaining'$ is adjacent to *every* vertex in $solution$). If vertices remain in $remaining'$, we recurse (line 15). We then take the opposite branch choice, and consider rejecting from $remaining$ and $solution$ (line 17). Finally, we loop, and pick a new v.

Integrating the colour bound We keep track of the best solution we have found so far: this is stored in $incumbent$, which is initially empty (line 4). Whenever we find a new clique, we compare its size to that of $incumbent$, and if it is better, the incumbent is unseated

(line 16). Now we may make use of the colour bound. At the start of the recursive procedure (line 9), we use `colourOrder` to produce a constructive greedy colouring of the subgraph induced by $remaining$ into the array $order$, with the colour numbers placed in $bounds$. When selecting v, we iterate over $bounds$ from right to left (line 10). Thus, on line 11 we know that the largest possible clique we could find at the current location has size no greater than $|solution| + bounds[i]$, and so if this cannot unseat the incumbent then we may abandon search and backtrack.

2.1.2 Bit Parallelism

As San Segundo, Rodríguez-Losada, and Jiménez (2011) and San Segundo, Matía, et al. (2013) observe, this algorithm is suitable for bit-parallelism. When permuting G on line 3, the graph should be re-encoded as an array of adjacency bitsets. (It is not helpful to do this before constructing G.) Now as discussed in Section 1.6.1, the intersection on line 14 becomes a simple bitwise "and" operation, and the intersection with complement on line 30 is a bitwise "and not" operation. Recall also that finding the first set bit in a bitset (needed on line 26) is a dedicated hardware instruction in modern processors.

The benefits to `colourOrder` are particularly important: except on the sparsest of graphs, nearly all of the runtime is spent producing colourings. Indeed, the particular colouring algorithm we have described is due to San Segundo, Rodríguez-Losada, and Jiménez (2011); the original MCS algorithm colours vertex by vertex, rather than colour class by colour class. It can easily be seen that the two strategies produce the same output, but the algorithm we describe is much more amenable to bit-parallelism.

2.2 Maximum Cliques in Random Graphs

In this section we evaluate our C++ implementation of Algorithm 2.1 on random graphs. We do this with a view to better understanding its behaviour, rather than as a "horse race" to try to demonstrate that this algorithm is best. We use the Erdős-Rényi probability model: by $G(n, p)$ we mean a random graph with n vertices, having an edge between each distinct pair of vertices with probability p. Our experiments are performed on systems with dual Intel Xeon E5-2697A v4 processors and 512GBytes RAM, running Ubuntu Linux 16.04. Our compiler is GCC 5.4.0.

In Figure 2.3 we reproduce some of the experiments performed by Prosser (2012): we look at random graphs with 100 and 150 vertices, and vary the edge probability from 0.40 to 1.00 in steps of 0.01 for 100 vertex graphs, and steps of 0.05 for 150 vertex graphs. For each point we use 100 samples. The shaded curves plot the mean runtime in microseconds.[1]

[1] Our steady-clock timer only has a granularity of one millisecond; however, using microseconds allows us

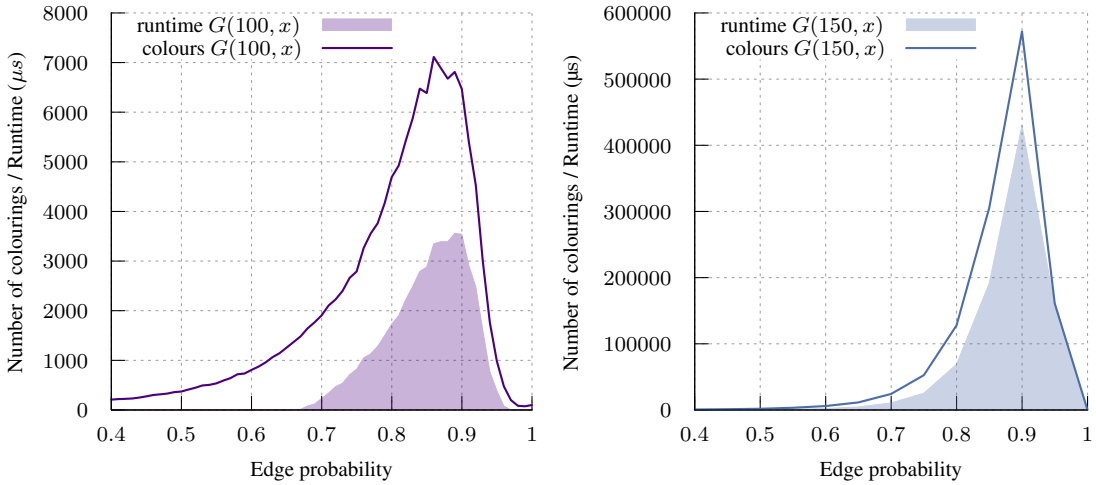


Figure 2.3: The maximum clique problem in random graphs of 100 (left) and 150 (right) vertices, with a low resolution and small sample size. On the x-axis we vary the edge probability from 0.40 to 1.00 in steps of 0.01 (100 vertices) or 0.05 (150 vertices). On the y-axis, we plot the mean number of colourings (equivalently, number of recursive calls) required, and the mean runtime in microseconds, using one hundred samples with Algorithm 2.1. These plots reproduce some of the experiments in Figure 6 of Prosser (2012); we argue that this data is insufficiently detailed to provide a complete understanding of the situation, and refer to Figure 2.4 for a better picture.

We also plot the number of recursive calls to `expand` made on the same axis. This gives us a machine-independent measurement of how much work the algorithm is performing, and such a measurement will help us to understand the algorithm's behaviour. Bessiere, Zanuttini, and Fernàndez (2004) argue that number of recursive calls is not an ideal measurement; however, in this case we are not using an algorithm based around a propagation queue, and the number of recursive calls corresponds directly to the number of colourings performed. Since nearly all of the algorithm's runtime is spent in the `colourOrder` routine, number of recursive calls is a meaningful measurement for this algorithm.

Our results mirror those of Prosser: the problem is very easy in extremely sparse or extremely dense graphs, but for graphs with density over 0.7 the problem becomes difficult. The difficulty results appear pleasing and look like the kind of curve that a scientific paper should contain: if we do not inspect the data too closely, it plausibly resembles a skewed (asymmetric) bell curve, with a slightly jagged peak that can be put down to small sample size.

We also see that the runtimes and number of recursive calls made correspond closely: the more recursive calls needed, the longer the algorithm takes to run. Our runtimes are much lower than Prosser's (2012) (non-bitset) runtimes, however: our peak mean runtimes are $320\,\mu\text{s}$ for 100 vertices and $437\,\text{ms}$ for 150 vertices, compared to around $250\,\text{ms}$ and $7500\,\text{ms}$ respectively. If we were following the DIMACS conventions, at this point we would resort to

to plot recursive calls and search nodes on the same axis.

rescaling results to try to account for hardware differences; Prosser also says that his bitset implementation is typically around twice as fast, so we could also take that into account. Fortunately however, Prosser's (2012) source code is available (although sadly the same cannot be said for the papers that introduced these algorithms, nor for most of the other papers discussed in the following sections). Thus we can reliably measure that on our machine and on random instances with 150 vertices and density 0.9, our code runs thirty to forty times faster, and uses exactly the same number of recursive calls, due to consistent tiebreaking rules.

However, there is much more to maximum clique algorithms on random graphs than Figure 2.3 suggests: the low resolution and small sample size is making the behaviour appear much simpler than it actually is. With a view to increasing our ignorance, the first use of parallelism in this thesis is simply to run an awful lot of experiments and produce Figure 2.4 as an extremely detailed replacement for Figure 2.3. Now we are using steps of 0.0001, and a thousand (200 vertices) or ten thousand (100 and 150 vertices) samples. This lets us determine that the jagged peak is indeed an artifact of sample size. However, the difficulty curves are not simple: there are wobbles all the way along the curves, or at least up to density 0.6 (after which it becomes hard to tell even at this sample size, but we will go on and justify why there are wobbles along the entire length).

We show the wobbles in more detail in the zoom boxes: the straight light grey lines plot the best fit of the form $y = e^{a+bx}$ over the subset of data selected. We can clearly see that the difficulty curves alternate going above and below the fit lines, with the period of alternation increasing as the density increases. This conclusively demonstrates that Figure 2.3 is misleading, and that previous sets of experiments have not produced the full story.

To explain these curves, we will first look at the clique decision problem ("does a given graph contain a clique with k vertices?"). The behaviour of decision problems on randomly generated problem instances is a well-studied topic, beginning with investigations into graph colouring, Hamiltonian circuits, travelling salesman, and Boolean satisfiability by Cheeseman, Kanefsky, and Taylor (1991) and Mitchell, Selman, and Levesque (1992). For example, for the graph colouring decision problem, Cheeseman, Kanefsky, and Taylor showed that sparse and dense instances are trivially satisfiable and unsatisfiable respectively, and are both computationally easy, whilst the instances in the narrow density region where there is a mix of satisfiable and unsatisfiable instances were hard for their algorithm. Subsequent research suggests that this behaviour is common to many (but not all) NP-complete problems, is algorithm-independent (or at least, that no known algorithm finds the "really hard" instances easy, although bad algorithms can find "easy" instances hard), and cannot be avoided by reductions.

We therefore ask: does the clique decision problem behave like the decision problems for colouring? We can modify Algorithm 2.1 to solve the decision problem, by priming the size of the incumbent to be $k - 1$ rather than zero, and allowing the algorithm to exit as soon as a

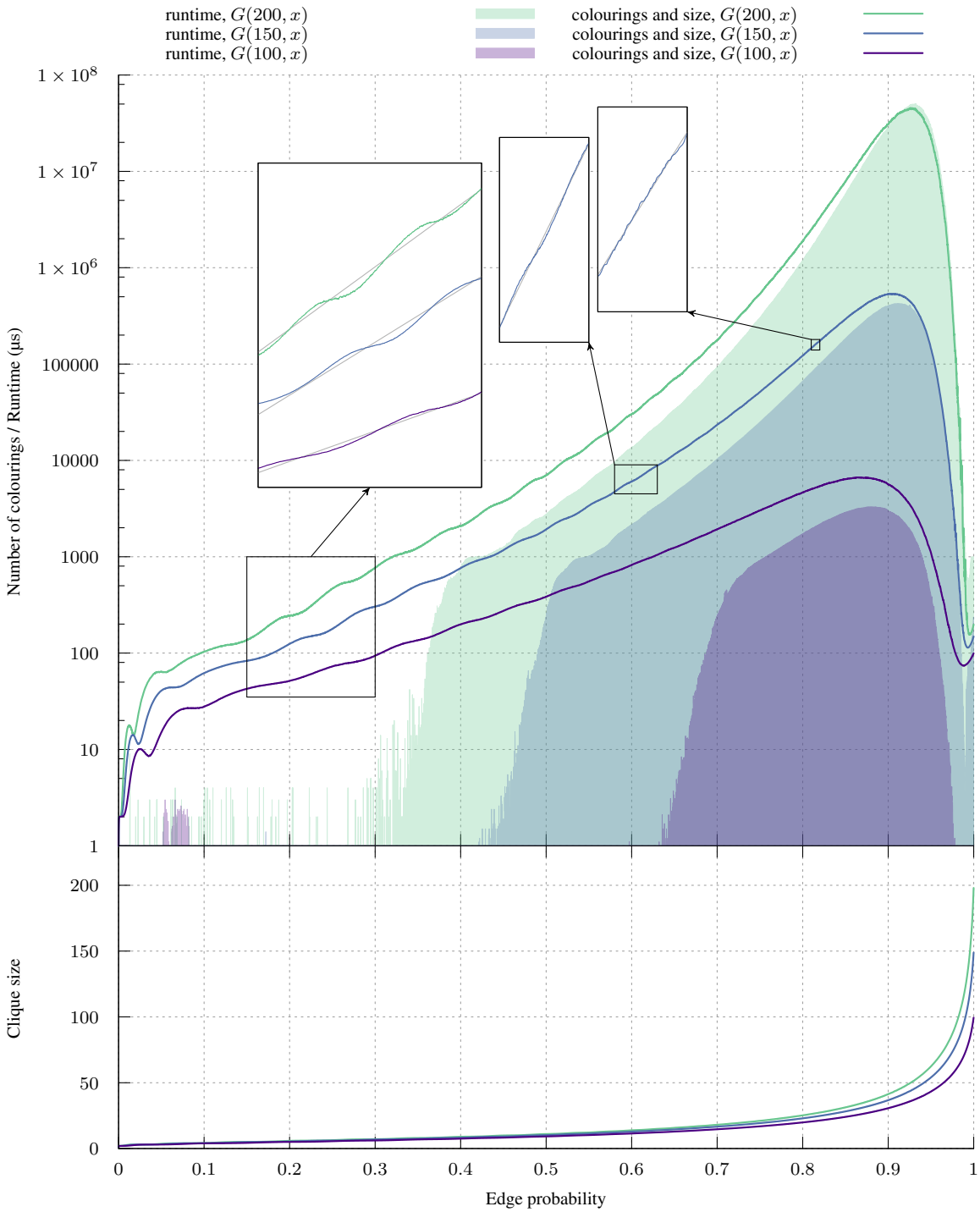


Figure 2.4: The maximum clique problem in random graphs of 100, 150, and 200 vertices. On the x-axis we vary the edge probability from 0.0000 to 1.0000 in steps of 0.0001. On the y-axis of the top plot, we plot the mean number of colourings required, and the mean runtime in microseconds, using one thousand (200 vertices) or ten thousand (100 and 150 vertices) samples with Algorithm 2.1. The inserts give a zoomed in look at some parts of the curves; the grey straight lines show the best fit of the form $y = e^{a+bx}$ over the subset of data selected. On the bottom plot y-axis, we plot the mean size of the maximum clique.

clique of size k is found. This allows us to produce Figure 2.5, which plots the $k = 20$ and $k = 25$ decision problem in graphs of 150 vertices and varying edge probabilities.

What we see are typical phase transitions: below a certain edge probability, all instances are unsatisfiable, above a certain edge probability all instances are satisfiable, and in the middle there is a narrow mushy region (Smith, 1994). We also see familiar easy-hard-easy difficulty peaks, with instances near the mushy region being by far the most difficult to solve. This matches our intuition: obviously a graph with only a few edges will not contain a large clique, but a graph with only a few edges missing will, and it is only in the middle that it is hard to tell. Our plots do have one oddity: there is an upwards kick at the end of the difficulty curve. This is simply because our algorithm only checks whether it can exit when it has found a clique which cannot be extended, and so at the far right where we have a complete graph, the algorithm requires 150 recursive calls before it terminates—if we were using one of the other measurement methods discussed by Bessiere, Zanuttini, and Fernàndez (2004), this artifact would disappear (but we would lose the close correspondence between number of recursive calls and runtimes).

But does this help us to understand the optimisation problem? In Figure 2.6 we plot the difficulty curves for every decision problem on a single chart, using dotted lines. Using solid lines, we also show the mean total difficulty of the optimisation problem, as well as the mean difficulty if the algorithm is primed with the size of an optimal solution, and the mean difficulty if the algorithm only has to find an optimal solution but is allowed to exit without proving there is nothing larger.

All three lines are clearly wobbly, but the peaks of the wobbles do not line up. The cost of proving optimality largely follows the difficulty of the most difficult decision problem at any given density, and as the gaps between the complexity peaks for subsequent decision problems start off far apart and get closer as density increases, this explains both the shape and why it is hard to tell whether wobbles exist for higher densities. The cost of finding an optimal solution but not proving its optimality also wobbles, but is out of phase: instances to the right of a complexity peak are likely to be satisfiable, and the further to the right of the peak we go, the higher the density of solutions is likely to be, making them easier to find (and as Smith and Dyer (1996) note, if we consider the enumeration problem rather than the decision problem, there is no peak and the cost instead continues to rise). However, if we go too far to the right, then the size of the optimal solution increases and we enter the "rare and hard" region of the subsequent decision problem. Finally, as expected, the total cost is close to the sum of the two other costs.

Viewing our data in another way supports this understanding. In Figure 2.7 we redraw the difficulty curve for graphs with 150 vertices. This time, we also draw, for each ω, a curve which considers the difficulty only of instances for which the solution is that ω; for these curves we use colours to show the sample size, with darker indicating a larger sample.

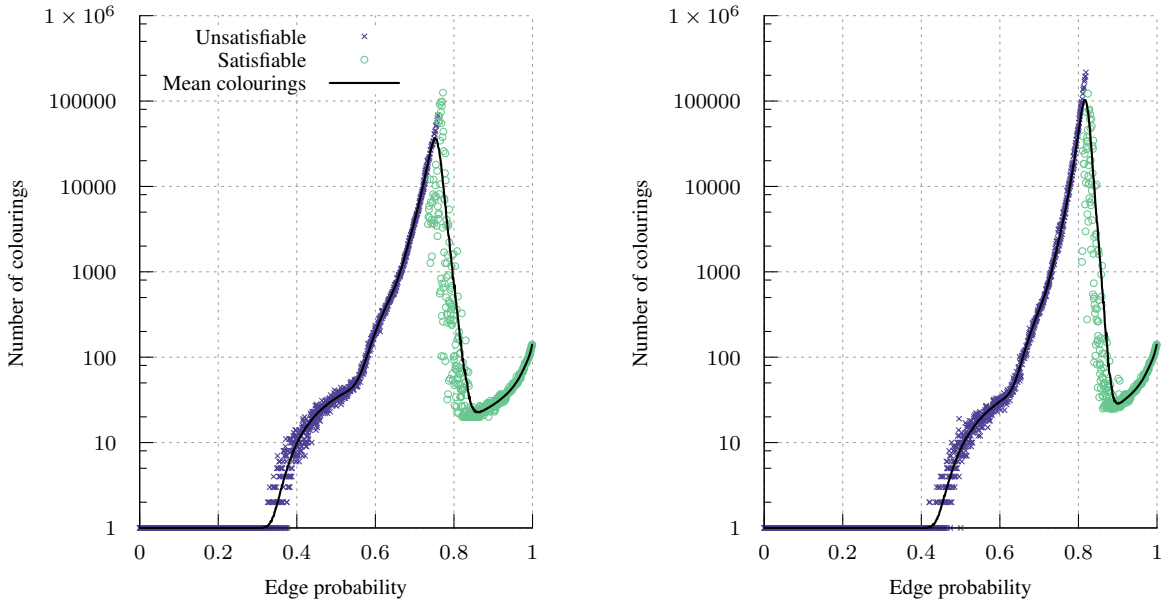


Figure 2.5: A phase transition in the clique decision problem. We plot the mean difficulty of the decision problem for cliques of order 20 (left) and 25 (right) in random graphs with 150 vertices and varying edge probabilities. We also plot a subset of the instances as individual points, showing whether that instance is satisfiable or unsatisfiable.

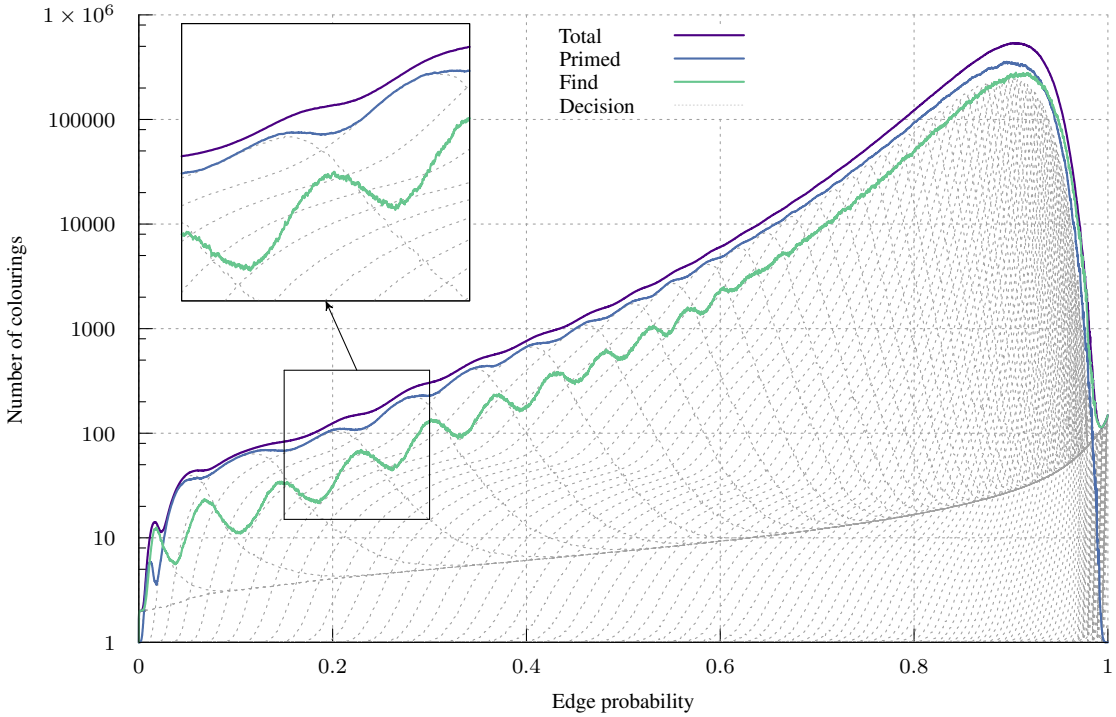


Figure 2.6: The difficulty of the maximum clique problem in random graphs with 150 vertices, and varying edge probabilities. The solid lines show the mean number of recursive calls for the optimisation problem, in total, if the incumbent is primed to the actual solution, and if the algorithm may exit as soon as the actual solution is found; each dotted line shows the mean number of recursive calls for one particular value of k for the decision problem.

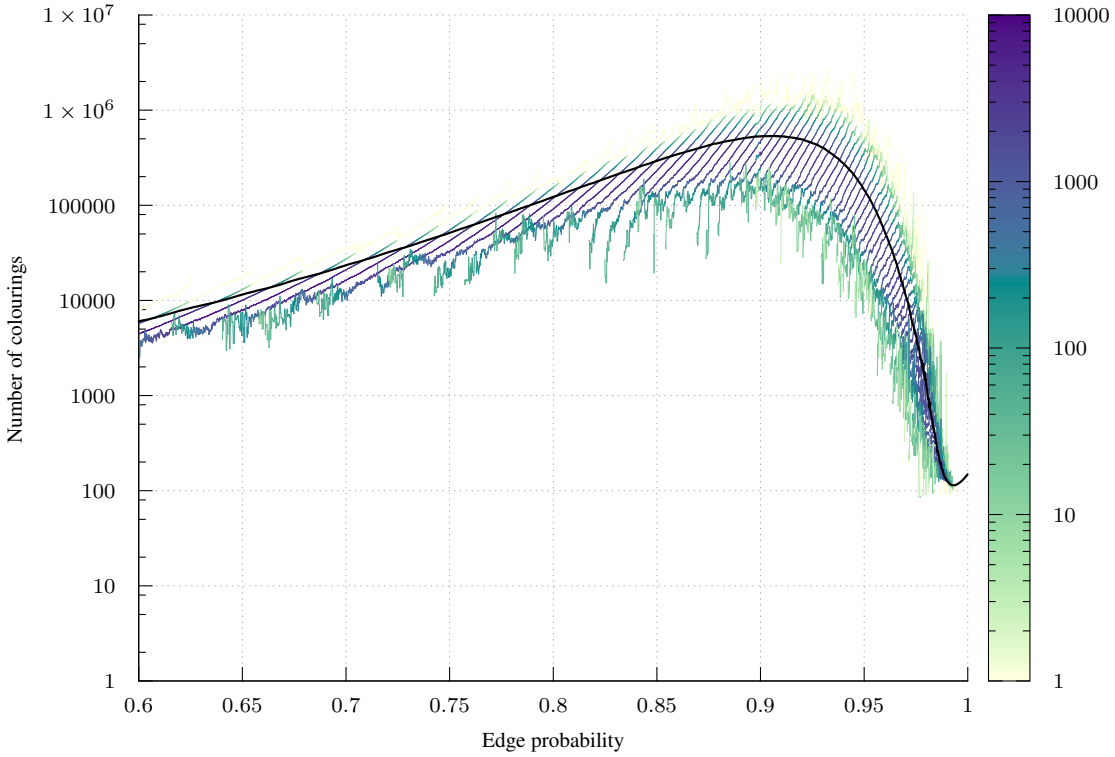


Figure 2.7: The difficulty of finding a maximum clique in random graphs of 150 vertices with varying edge probabilities. The black line shows mean search effort. Each coloured line shows mean search effort only for instances for a particular value of ω, with colour indicating sample size.

We can now see the following: for any particular ω, we occasionally encounter atypically sparse graphs with that ω as a solution, and these instances tend to be many times easier than average. We also sometimes encounter atypically dense graphs, and these graphs tend to be many times harder than average. In other words, if a graph contains a larger maximum clique than it "should" considering its density, then it is unusually easy (because having a stronger incumbent makes an optimality proof easier), and if it contains a smaller maximum clique than it "should", it is unusually difficult (conversely, having a weak incumbent makes the optimality proof harder).

There is one further piece of information which would allow us to explain the entire shape of the curve for the optimisation problem: why does the most difficult decision problem (when we are allowed to pick any ω and any density) occur with a density slightly higher than 0.9? Sadly we do not have the answer to this final part of the puzzle. It is not even clear whether the most difficult density is genuinely algorithm-independent—it is certainly possible to write algorithms which find densities higher than 0.9 even harder, but are there algorithms which are substantially better than Algorithm 2.1 on the hardest densities (or good reasons to believe that such algorithms cannot exist)? We return to these kinds of question in Chapter 6.

2.3 Benchmark Instances

Although random graphs allow us to gain insights into the behaviour of the algorithm, we are not primarily interested in finding cliques in random graphs. We now describe the standard benchmark instances used to evaluate maximum clique algorithms, together with some less well-known datasets.

In Table 2.1 we provide both the number of search nodes and runtimes using Algorithm 2.1 to give a rough feel for the difficulty of these instances; where substantially better runtimes may be obtained by using one of the techniques discussed in Section 2.5, we also discuss this. As previously, runtimes are from systems with dual Intel Xeon E5-2697A v4 processors. We also give primed runtimes (that is, runtimes where we initialise the incumbent to be the size of a maximum clique, rather than zero): this is to provide insight into the structure of these instances, and is not a claim that such a runtime is realistic.

2.3.1 The Second DIMACS Implementation Challenge

The Second DIMACS Implementation Challenge (Johnson and Trick, 1993) introduced a suite of problem instances which have been very widely used for benchmarking maximum clique algorithms—indeed, many papers look only at these instances (or a subset thereof). It is conventional to just provide a table of results looking at these instances, without considering what the graphs are; we will take the unusual step of discussing each family in more detail, since we believe that some of the conclusions reached based upon these datasets are questionable due to the nature of certain instances.

Random graphs from the "C", "DSJC" and "p_hat" families: The "C" and "DSJC" families contain randomly generated instances. For the "C" family, the ".5" instances (which have density 0.5 and order up to 4,000) have all been solved. The ".9" graphs with 500 vertices and upwards are believed to be open. Both "DSJC" graphs, which have density 0.5 and 500 or 1,000 vertices, have been solved.

The "p_hat" family are also random graphs, but with an unusually large degree spread, created using the $\hat{p}$ generator (Gendreau, Soriano, and Salvail, 1993; Soriano and Gendreau, 1993). The largest and densest of these, "p_hat1500-3", was first solved by McCreesh and Prosser (2013) in 128 days using 32 hardware threads.

Random graphs from the "brock", "gen", and "san(r)" families: The "brock" family of instances is due to Brockington and Culberson (1993). They are an attempt at camouflaging a known clique in a quasi-random graph for cryptographic purposes, in a way that was resistant to certain early heuristic attacks. There are three subfamilies, "brock200", "brock400" and "brock800"; the number denotes the number of vertices in the graphs. Roughly speaking, for

Table 2.1: Properties of the DIMACS benchmark instances. We give non-rounded figures for recursive calls to aid verification of other implementations. The "primed" columns show the ratio compared to the base algorithm, if the incumbent is initialised with the size of a maximum clique.

Instance	$\|V\|$	D	ω	Colourings		Runtime (ms)	
				Base	Primed	Base	Primed
Randomly generated							
C125.9	125	0.90	34	50 240	0.536	57	0.579
C250.9	250	0.90	44	1 082 441 593	0.895	1 126 147	0.899
C500.9	500	0.90	≥ 57		Open		
C1000.9	1 000	0.90	≥ 68		Open		
C2000.5	2 000	0.50	16	18 189 648 267	1.000	38 832 285	1.011
C2000.9	2 000	0.90	≥ 80		Open		
C4000.5	4 000	0.50	18	See McCreesh and Prosser (2013)			
DSJC500_5	500	0.50	13	1 153 043	0.948	733	0.958
DSJC1000_5	1 000	0.50	15	76 981 458	0.997	87 921	0.993
Randomly generated with large degree spread							
p_hat300-1	300	0.24	8	1 480	0.871	1	1.000
p_hat300-2	300	0.49	25	4 256	0.665	7	0.571
p_hat300-3	300	0.74	36	624 947	0.394	666	0.432
p_hat500-1	500	0.25	9	9 777	0.992	9	0.667
p_hat500-2	500	0.50	36	114 009	0.347	174	0.420
p_hat500-3	500	0.75	50	39 260 458	0.397	70 925	0.441
p_hat700-1	700	0.25	11	26 649	0.607	27	0.667
p_hat700-2	700	0.50	44	750 903	0.504	1 830	0.576
p_hat700-3	700	0.75	62	282 412 276	0.567	935 998	0.599
p_hat1000-1	1 000	0.24	10	176 576	0.992	147	1.007
p_hat1000-2	1 000	0.49	46	34 473 978	0.632	94 978	0.657
p_hat1000-3	1 000	0.74	68	130 317 818 368	0.295	470 938 212	0.322
p_hat1500-1	1 500	0.25	12	1 184 526	0.809	1 799	0.871
p_hat1500-2	1 500	0.51	65	2 006 796 270	0.545	13 209 931	0.575
p_hat1500-3	1 500	0.75	94	See McCreesh and Prosser (2013)			
Randomly generated with large hidden solutions							
brock200_1	200	0.75	21	524 723	0.583	316	0.630
brock200_2	200	0.50	12	3 826	0.674	4	0.500
brock200_3	200	0.61	15	14 565	0.997	14	0.571
brock200_4	200	0.66	17	58 730	0.538	47	0.617
brock400_1	400	0.75	27	198 359 829	0.590	184 360	0.644
brock400_2	400	0.75	29	145 597 994	0.332	133 714	0.415
brock400_3	400	0.75	31	120 230 513	0.138	106 134	0.202
brock400_4	400	0.75	33	54 440 888	0.141	51 592	0.214
brock800_1	800	0.65	23	2 227 634 634	0.790	3 080 308	0.837
brock800_2	800	0.65	24	2 235 803 416	0.586	3 083 366	0.671
brock800_3	800	0.65	25	2 146 717 172	0.327	2 890 338	0.415
brock800_4	800	0.65	26	640 444 536	0.795	1 075 174	0.858
gen200_p0.9_44	200	0.90	44	1 774 374	0.084	1 818	0.106
gen200_p0.9_55	200	0.90	55	170 254	0.014	178	0.022
gen400_p0.9_55	400	0.90	55	2 353 914 262 613	0.204	3 585 700 777	0.231
gen400_p0.9_65	400	0.90	65	175 757 037 249	0.041	280 143 536	0.058
gen400_p0.9_75	400	0.90	75	104 883 350 585	0.001	157 187 491	0.002

continued on next page...

Instance	|V|	D	ω	Colourings		Runtime (ms)	
				Base	Primed	Base	Primed
san200_0.7_1	200	0.70	30	13 399	0.017	15	0.000
san200_0.7_2	200	0.70	18	464	0.002	1	0.000
san200_0.9_1	200	0.90	70	87 329	$<10^{-3}$	81	0.000
san200_0.9_2	200	0.90	60	229 567	0.005	260	0.008
san200_0.9_3	200	0.90	44	6 815 145	0.062	6 189	0.094
san400_0.5_1	400	0.50	13	2 453	$<10^{-3}$	8	0.125
san400_0.7_1	400	0.70	40	119 356	0.083	147	0.190
san400_0.7_2	400	0.70	30	889 125	0.094	1 313	0.166
san400_0.7_3	400	0.70	22	521 410	0.129	859	0.193
san400_0.9_1	400	0.90	100	4 536 723	0.073	15 285	0.101
san1000	1 000	0.50	15	150 725	$<10^{-5}$	1 178	0.009
sanr200_0.7	200	0.70	18	152 882	0.824	110	0.818
sanr200_0.9	200	0.90	42	14 921 850	0.683	14 323	0.760
sanr400_0.5	400	0.50	13	320 110	0.612	185	0.659
sanr400_0.7	400	0.70	21	64 412 015	0.993	48 319	1.008
Randomly generated with known solution sizes							
frb30-15-1	450	0.82	30	292 095 125	0.745	456 067	0.784
frb30-15-2	450	0.82	30	557 252 809	0.596	811 184	0.639
frb30-15-3	450	0.82	30	167 116 178	0.600	249 959	0.643
frb30-15-4	450	0.82	30	991 460 271	0.423	1 363 978	0.467
frb30-15-5	450	0.82	30	282 763 799	0.627	402 948	0.688
frb35-17-1	595	0.84	35	13 273 030 824	0.517	30 541 366	0.568
frb35-17-2	595	0.84	35	23 358 937 783	0.674	55 226 300	0.699
frb35-17-3	595	0.84	35	8 248 153 344	0.752	20 187 054	0.771
frb35-17-4	595	0.84	35	8 850 406 216	0.746	22 558 406	0.784
frb35-17-5	595	0.84	35	58 010 454 258	0.575	123 320 497	0.604
Fault diagnosis							
c-fat200-1	200	0.08	12	24	0.125	0	1.000
c-fat200-2	200	0.16	24	24	0.042	0	1.000
c-fat200-5	200	0.43	58	139	0.194	1	0.000
c-fat500-1	500	0.04	14	14	0.071	0	1.000
c-fat500-2	500	0.07	26	26	0.038	1	0.000
c-fat500-5	500	0.19	64	64	0.016	2	0.500
c-fat500-10	500	0.37	126	126	0.008	3	0.333
Coding theory							
hamming6-2	64	0.90	32	32	0.031	0	1.000
hamming6-4	64	0.35	4	82	0.988	0	1.000
hamming8-2	256	0.97	128	128	0.008	2	0.500
hamming8-4	256	0.64	16	36 452	1.000	39	0.769
hamming10-2	1 024	0.99	512	512	0.002	64	0.750
hamming10-4	1 024	0.83	40	Known by construction, open as a clique instance			
johnson8-2-4	28	0.56	4	24	0.958	0	1.000
johnson8-4-4	70	0.77	14	126	0.913	0	1.000
johnson16-2-4	120	0.76	8	256 100	1.000	55	0.618
johnson32-2-4	496	0.88	16	Known by construction, open as a clique instance			
Keller conjecture							
keller4	171	0.65	11	13 725	0.997	10	0.500
keller5	776	0.75	27	50 707 104 364	1.000	90 244 319	1.001
keller6	3 361	0.82		See Debroni et al. (2011), open as a clique instance			

continued on next page...

Instance	|V|	D	ω	Colourings		Runtime (ms)	
				Base	Primed	Base	Primed
Steiner triple problem							
MANN_a9	45	0.93	16	71	0.845	0	1.000
MANN_a27	378	0.99	126	38 019	0.994	172	0.994
MANN_a45	1 035	1.00	345	2 851 572	0.998	123 226	1.063
MANN_a81	3 321	1.00	1 100	See McCreesh and Prosser (2013)			
Proteins							
1KZKA_3KT2A_78	271	0.99	247	247	0.004	3	0.333
1allA_3dbjC_41	451	0.97	346	675	0.551	19	0.263
1f82A_1zb7A_5	655	0.97	500	716	0.411	27	0.481
2FDVC_1PO5A_83	750	0.96	556	1 348	0.108	35	0.486
2UV8I_2J6IA_13107	200	0.86	69	4 263	0.108	8	0.125
2W00B_3H1TA_10858	346	0.91	143	777 428	0.158	1 939	0.199
2W4JA_2A2AD_0	563	0.98	447	890	0.011	16	0.375
3HRZA_2HR0A_476	905	0.94	563	934 965	0.346	17 899	0.386
3P0KA_3GWLB_0	138	0.94	89	90	0.033	0	1.000
3ZY0D_3ZY1A_110	61	0.98	52	52	0.019	0	1.000

a good exact maximum clique algorithm on modern hardware, members of the "brock200" family can be solved in well under a second, the "brock400" family take a few minutes, and the "brock800" family take around an hour. In each case, there is a unique maximum clique in these graphs (except for "brock200_1", which has two equally sized hidden cliques), and the clique is larger than one would expect from a random graph with the same density.

The "gen" and "san(r)" instances use a different technique for hiding a large clique of known size in a graph (Sanchis, 1992; Sanchis, 1995). Again, they are an attempt to create challenging instances with a known optimal solution. This has implications for the behaviour of algorithms: as we can see from Table 2.1, once an optimal solution has been found, these instances become easy. This is a more extreme version of the effect discussed in Section 2.2. Corrêa et al. (2014), Maslov, Batsyn, and Pardalos (2014), Batsyn et al. (2014) and Tomita, Yoshida, et al. (2016) all claim huge speedups on some of these instances and use it as justification for their new algorithms, but none discuss the peculiar nature of the large hidden cliques in these graphs.

Fault diagnosis via the "c-fat" family: These graphs are related to fault diagnosis for distributed systems (Berman and Pelc, 1990). All are computationally trivial.

Coding theory with the "hamming" and "johnson" families: The "hamming" and "johnson" graphs model problems from coding theory (Bomze et al., 1999). The solutions to these instances are all known through non-clique means, but proofs of optimality for "hamming10-4" and "johnson32-2-4" are beyond current maximum clique algorithms.

Mathematical problems with the "keller" and "MANN" families: These families encode mathematical conjectures. The "keller" instances encode a geometric conjecture. All of these instances have been solved, using special knowledge of the structure of the graphs (Debroni et al., 2011). Using general purpose maximum clique algorithms, the "keller6" graph remains unsolved.

The MANN family is made from clique formulations of the Steiner triple problem, due to Mannino and Sassano (1995). All have been solved: "MANN_a9" and "MANN_a27" are not challenging, and "MANN_a45" takes minutes; "MANN_a81" was first solved as a general maximum clique problem by McCreesh and Prosser (2013) in 31 days using 24 hardware threads (although the solution was already known by other means). These graphs are very dense, which sometimes gives atypical behaviour from algorithms.

2.3.2 Benchmarks with Hidden Optimal Solutions

The "frb" family of instances (K. Xu, 2014) are another attempt at generating hard instances with known solution sizes. They are produced using a generator described by K. Xu, Boussemart, et al. (2005) and K. Xu and W. Li (2006). Algorithm 2.1 finds these difficult: the instances with solution size 30 take around ten minutes, and the 35 family instances are within reach in a day, but larger instances are intractable.

Interestingly, the modifications by San Segundo, Nikolaev, Batsyn, and Pardalos (2016) make the instances with solution size 30 trivial (but larger instances are not reported); C. Li, Z. Fang, and K. Xu (2013) do well on larger instances too. Using a fixed parameter tractable algorithm, Sharmin (2014) does extremely well on BHOSLIB instances, which is especially interesting since her algorithm cannot solve *any* of the DIMACS instances within sixty minutes.

2.3.3 Protein Product Graphs

Depolli et al. (2013) use the maximum clique problem to compare proteins, using a product graph encoding which we discuss in much more detail in Chapter 7. The instances can all be solved very quickly, with only two taking over one second. Interestingly, Depolli et al. show that these instances are beyond the reach of naïve algorithms. It is also worth noting that these graphs are real-world instances (that is, they are not randomly generated, and the result has a real-world meaning), but the graphs are not sparse. As we will see in Chapter 7, the product graph of two sparse graphs is not itself sparse.

2.3.4 Solving Other Problems via Maximum Clique

Other maximum clique instances will arise in Chapter 4, where they are used to solve the maximum k-clique problem, and in Chapter 7, where they are used to solve the maximum common subgraph problem.

2.3.5 Other Applications

Beyond the application areas covered by the problem instances we have seen so far, maximum clique problems arise in bioinformatics (Eblen et al., 2012), in biochemistry and genomics (Butenko and Wilhelm, 2006; Fukagawa et al., 2011; Konc and Janežič, 2007a), in community detection (B. Yan and Gregory, 2009), in document clustering (Okubo and Haraguchi, 2006), in telecommunications (Balasundaram and Butenko, 2006), in computer vision and electrical engineering (Bomze et al., 1999), in image comparison (San Segundo et al., 2010), and in controlling hordes of flying robots for the upcoming apocalypse (Regula and Lantos, 2013; Walsh, 2015). Bomze et al. (1999) and Wu and Hao (2015) discuss further applications and give broader perspectives on clique problems.

2.4 Explaining the Iteration Order

We now return to Algorithm 2.1, to explain an aspect of its behaviour which is not properly addressed in the literature: on line 10, why do we select vertices in colour class order, and why select from right to left? This has an efficient implementation using a pair of arrays, and allows colourings to be reused when iterating. However, recursing from left to right (and thus selecting from the first colour class first, rather than the last colour class first) may be implemented equally efficiently, so why use a reverse order? Tomita and Kameda (2007) claim that vertices in the rightmost colour class are "generally expected [to have a] high probability of belonging to a maximum clique". This claim was not tested experimentally, beyond verifying that the reverse ordering gives much worse performance.

In Figure 2.8 we test this claim experimentally. Returning to 150 vertex random graphs, the top solid line plots the difficulty of Algorithm 2.1, and the top dotted line plots the difficulty if instead of iterating in reverse order, we iterate selecting the first-created colour class first. Due to the extremely long execution times, we use increments of 0.001 and 100 samples for the "forwards" lines, which is why they appear wavier (and the jumps at high densities are caused by three outliers). We see that iterating in reverse order is substantially better—so far so good. However, the second-top solid and dotted lines repeat the experiments, priming search with the incumbent set to the size of a maximum clique. The reverse order remains better than the forward order, by a very similar factor. Tomita and Kameda's (2007) claimed explanation, then, cannot tell the entire story: with a primed incumbent, we are only

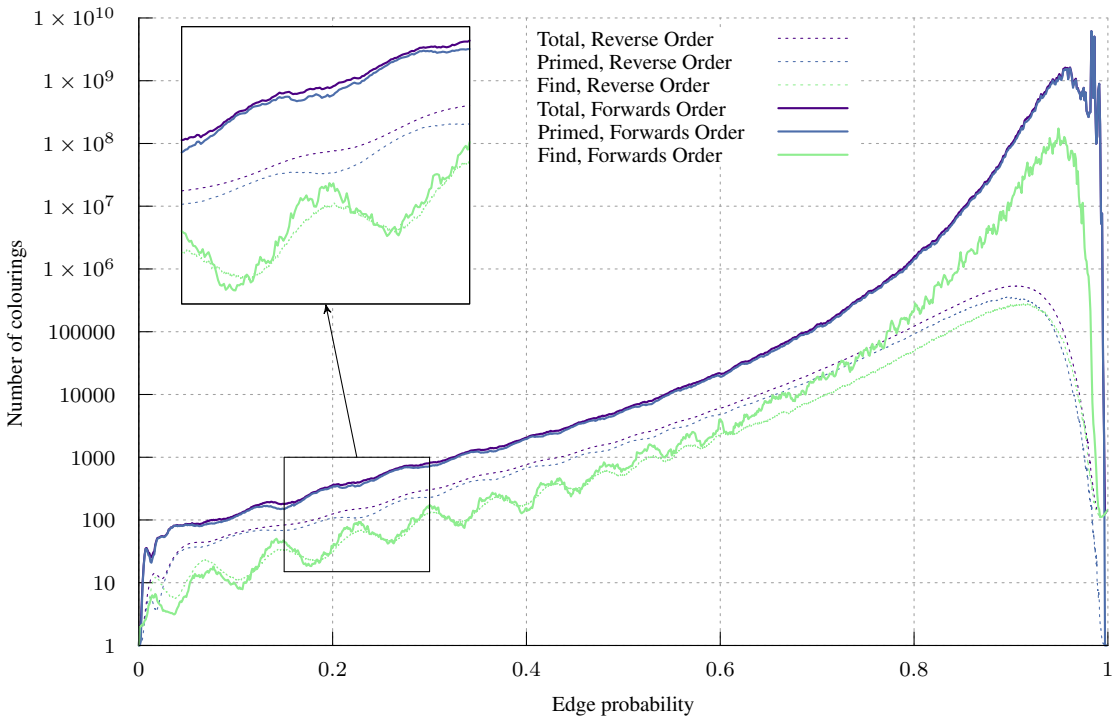


Figure 2.8: The difficulty of the maximum clique problem in random graphs of 150 vertices, iterating in either reverse colour class order (the default), or in forwards colour class order (the opposite). Forwards measurements use a smaller sample size and lower resolution.

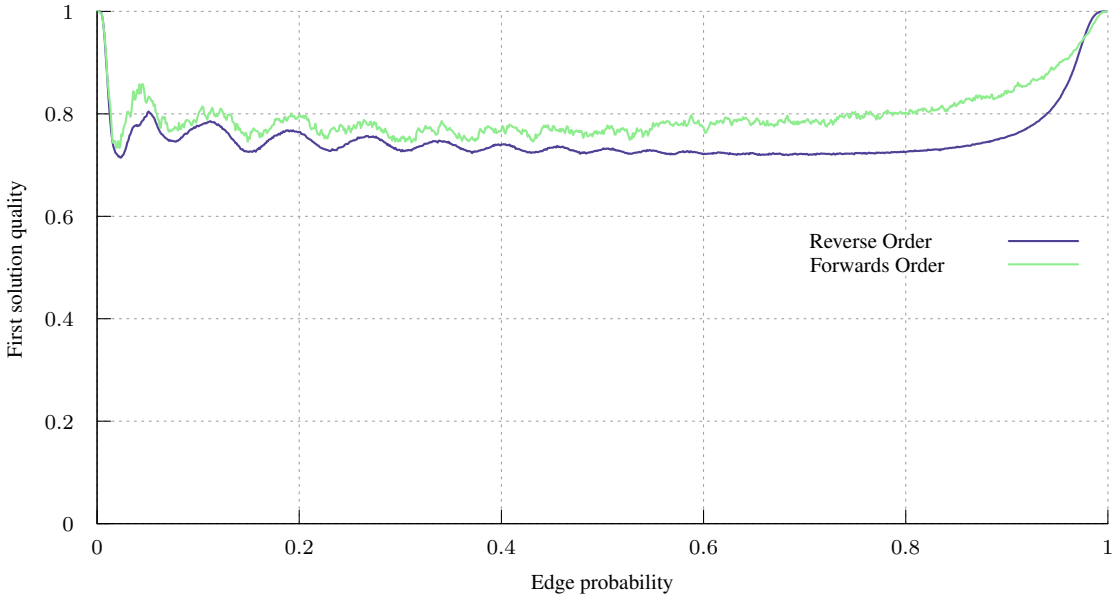


Figure 2.9: The size of the first solution found, as a proportion of the optimal solution, for the maximum clique problem in random graphs of 150 vertices, iterating in either reverse colour class order (the default), or in forwards colour class order (the opposite). Forwards measurements use a smaller sample size and lower resolution.

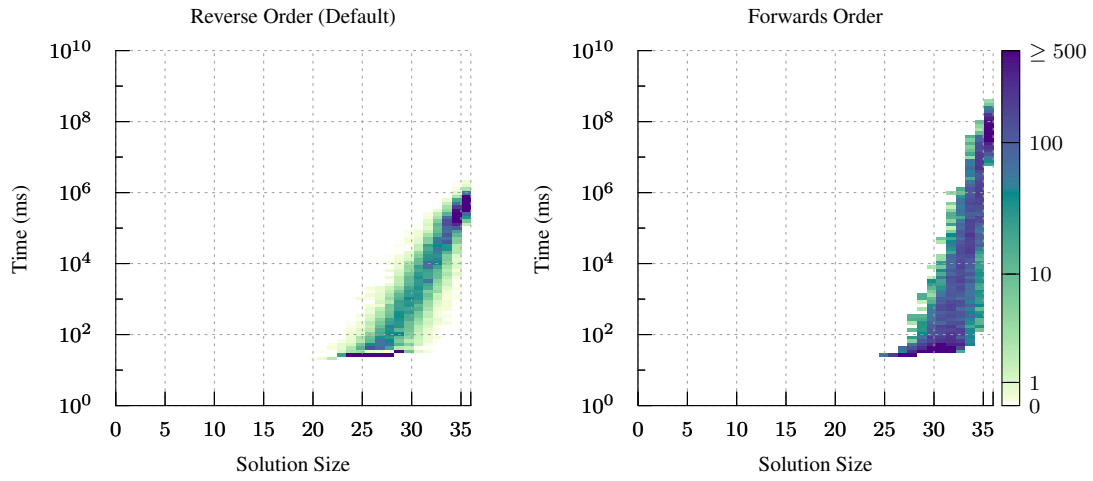


Figure 2.10: The solution quality over time, for instances of $G(150, p)$ where $\omega = 35$. On the left, the default reverse ordering, which finds weaker solutions initially but finishes faster; on the right, the forwards ordering, which finds better solutions initially but takes longer overall.

proving optimality, not finding a maximum clique, and so the "probability of belonging to a maximum clique" is irrelevant when selecting vertices.

Finally in Figure 2.8, the lowest solid and dotted lines plot time to find, but not prove, an optimal solution. If rightmost vertices were most likely to be in a maximum clique, we would expect the solid line to be below the dotted line. In fact, the two lines cross. Roughly speaking, for densities up to 0.5, the forwards order line is a slightly amplified version of the reverse order line, having taller peaks (being worse for densities where finding an optimal solution is relatively hard), but also shallower troughs (being better for densities where finding an optimal solution is relatively easy).

We can also directly measure the size of the first solution found by the two orderings. Figure 2.9 displays this information, using the initial solution size as a proportion of the optimal solution on the y-axis. The results show that (except for a very narrow region with extremely dense graphs) iterating forwards finds *better* initial solutions, even though it takes much longer to solve problems overall. Inspecting individual instances in more detail suggests that in general, iterating forwards finds better solutions quickly, but then takes much longer to improve those solutions and then prove optimality, leading to longer runtimes overall. Figure 2.10 confirms this: from the 5,142 instances of $G(150, p)$ which had $\omega = 35$, we record a point for each time the incumbent was unseated, using the new solution size on the x-axis and the time it was found on the y-axis. We also record a point with $x = 36$ for the time to complete (that is, to prove that no solution with $\omega \geq 36$ exists). This data is shown as a heatmap, with darker colours indicating a higher density of individual points at a location.

Tomita and Kameda's (2007) claimed explanation, then, is not supported by experimental evidence. However, reverse iteration clearly *is* the better strategy. We must therefore seek a different explanation to understand the success of the reverse selection order. Our inspiration

will come from variable ordering heuristics in constraint programming.

Let us reflect upon the greedy colouring process. Intuitively, one might suspect that early colour classes are likely to be larger: colour classes are filled greedily, with vertices being placed in the first available colour class. Selecting from small colour classes first is beneficial: consider Figure 2.2 on page 31, and suppose $incumbent = 3$. If we select v from the rightmost colour class (which contains only one vertex) first, we make only a single recursive call which cannot be eliminated by the bound. But if we were to select from the leftmost colour class, we would have to make four recursive calls before our bound would decrease. (This also shows why we commit entirely to a selected colour class: we want to eliminate colour classes as quickly as possible.)

In constraint programming terms, if we view colour classes (rather than vertices) as variables, with each domain having an additional wildcard value meaning "nothing from this colour class", then selecting from small colour classes first is a "smallest domain first" variable selection heuristic (albeit a slightly unusual one, since we produce a new set of variables with each new colouring at each level of search). Haralick and Elliott (1980) show that such a heuristic tends to give a low branching factor locally (that is, it reduces the number of recursive calls made). This does not necessarily produce the best possible search tree globally, but we will demonstrate that it is generally beneficial in this context.

2.4.1 Are Colour Classes Roughly Sorted by Size?

We will now test our intuition, by augmenting Algorithm 2.1 to take measurements inside the search. The hypothesis we are testing is as follows: is there a correlation between the position a colour classes is in, and the position it would be in if colour classes were sorted by size (largest first)? To measure this, we use the Kendall tau test—this will give us a value of 1 if there is a perfect monotonically increasing relationship, -1 if it is perfectly monotonically decreasing, and a value in-between otherwise.

We performed this test for each colouring produced, over 100 samples of random graphs $G(150, 0.9)$. The results are plotted in the top left graph of Figure 2.11. For the x-axis, we use the number of colour classes used. For the y-axis, rather than show the average, we show the distribution of the results of the statistical test (so the colours in each column sum to 1). For comparison purposes, the bottom left graph shows what we would see if the colour classes were in no particular order (we shuffle the colour classes before running the test), and the bottom right graph shows the color classes fully sorted (i.e. SDF). These results confirm our suspicions that colour classes are "roughly" sorted by size, as a side effect of the greedy colouring process: the top left graph is much more heavily weighted towards 1 (sorted) than the shuffled graph. In other words, the greedy colouring process and backwards iteration is approximating an SDF heuristic.

The top right graph shows the effects of our "domains of size two first" heuristic, which

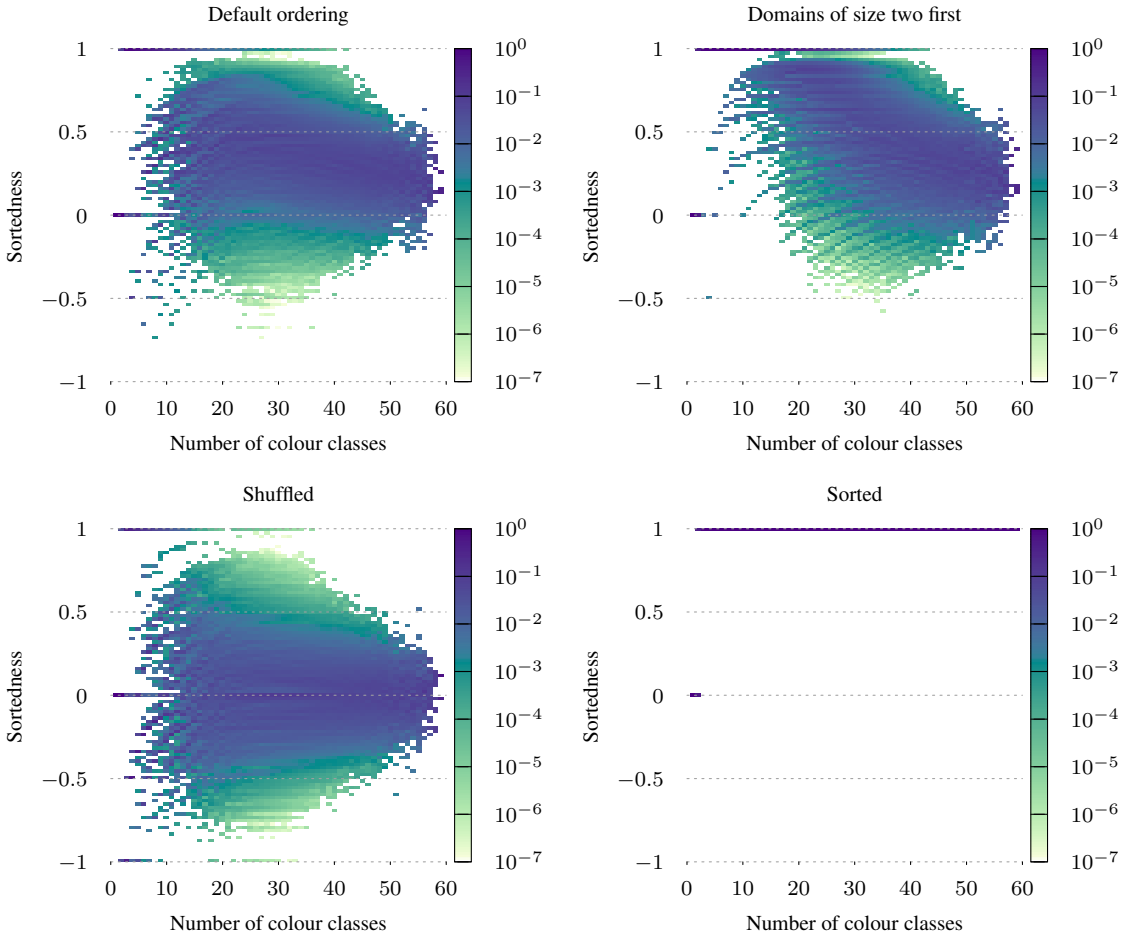


Figure 2.11: Are colour classes sorted? The top graphs shows the distribution of the Kendall measure on the colour classes generated during search, with the x-axis being the number of colour classes generated. The bottom graphs provide points of comparison.

we describe below. As its name suggests, we use a partial sort to increase the degree to which colour classes are sorted by size, but does not sort them fully—it is a cheap surrogate for SDF.

2.4.2 Reordering Colour Classes

We have established empirically that smaller colour classes tend to be picked earlier by greedy colourings algorithms, and explained theoretically why this is beneficial. Now ask what would happen if we increased this effect. We consider two approaches.

The "sorted", or "smallest domains first" variation. We could explicitly select from the smallest colour class (domain) first, by adding a (stable) sort step to the colouring routine.

The "partially sorted", or "domains of size two first" (2DF) variation. We also consider a potentially cheaper alternative: instead of fully sorting colour classes by size, we propose a partial sort that moves colour classes containing only one vertex (which we call *singleton*

colour classes) to the end of the list of colour classes, so that they are selected first. In other words, we are picking from domains with two values (a single vertex, plus the "nothing" option) first.

2.4.3 Tie-breaking

But why are we preserving the relative order of the partially sorted colour classes—that is, why do we specify a stable sort, or why is it important to put the last singleton colour class at the end of the list of colour classes? Suppose Algorithm 2.1 produced the colour classes shown in Figure 2.12. Due to the greediness of the colouring, vertices 5, 6, 7, 8 and 9 must all be adjacent to vertex 4 (the only member of the purple colour class). Thus if Algorithm 2.1 selects colour class $\{4\}$ in preference to the other singleton colour class $\{7\}$, the new candidate set $remaining'$ will contain *some* of the vertices from the set $\{1, 2, 3\}$, and *all* of the vertices from the sets $\{5, 6\}$, $\{7\}$ and $\{8, 9\}$. However, by the same kind of reasoning, if the colour class $\{7\}$ is selected before $\{4\}$, $remaining'$ will contain *some* of the vertices from the sets $\{1, 2, 3\}$ and $\{5, 6\}$ and *all* of the vertices in the sets $\{4\}$ and $\{8, 9\}$, so the new candidate set will potentially be smaller. This is why we preserve the order: selecting from the latest-coloured singleton colour class first can increase the amount of filtering done on $remaining$, giving a smaller $remaining'$ in the recursive call, further reducing the branching in the search process.

2.4.4 Does Reordering Help?

Next we verify that reordering colour classes actually leads to a reduction in search space size. In Figure 2.13 we compare the three ordering strategies in random graphs of 150 vertices and varying densities. The results show that both reordering heuristics reduce the size of the search space by around twenty percent on the most difficult instances, and that "domains of size two first" is nearly as effective as "smallest domain first". The "domains of size two first" heuristic also leads to a reduction in runtimes by a similar factor. However, the "smallest

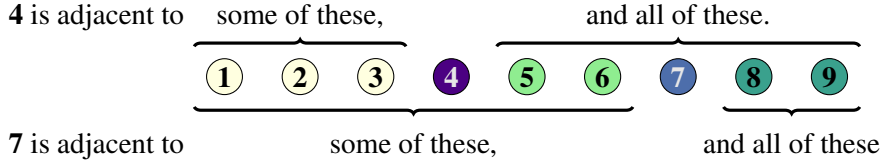


Figure 2.12: Due to the greedy colouring, singleton colour classes are not equally powerful from a filtering perspective. For any singleton colour class, its vertex is adjacent to every vertex with a later colour, but only some vertices with an earlier colour. Here, branching on vertex 7 rather than vertex 4 is likely to lead to more filtering, giving a smaller subproblem at the next recursive call.

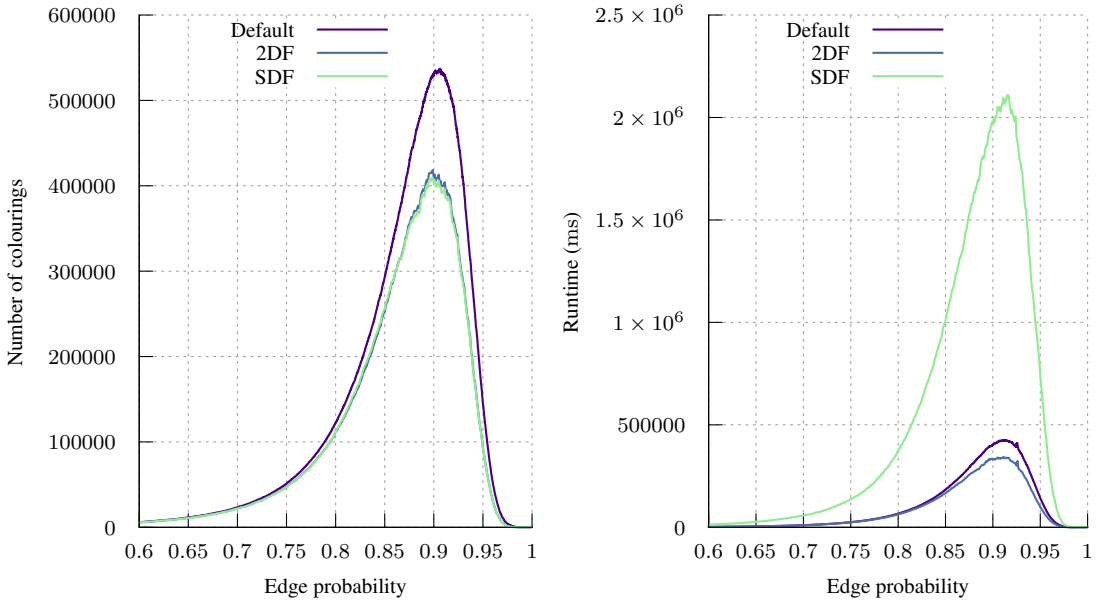


Figure 2.13: The maximum clique problem in random graphs of 150 vertices, with colour class reordering. On the left, search space size, and on the right, runtimes.

domains first" heuristic takes much longer, introducing a slowdown of three to four, despite the reduction in the search space.

What about on non-random instances? In the top row of Figure 2.14 we show the cumulative number of instances solved over time (right) or as a function of search space size (left). The differences are hard to see, as we would expect: because of the huge range of difficulties involved, we are using a log scale, but if random graphs are any indication, we expect to see a small constant factor difference. However, the "2DF" line is usually above the "default" line in both plots. In the second row, we give instance by instance comparisons of the search space sizes. Again, because of the log scale, the differences are hard to see, but most of the points are slightly below the diagonal line, indicating a small improvement in search space size. We see a similar result for runtimes with "2DF" on the final row.

Because of the relatively small number of instances available, we are able to present Table 2.2, which lists a full set of results. Overall, the trend is similar to that in random graphs: "2DF" usually gives a small improvement to both search space size and runtimes, whilst "SDF" usually gives a slightly larger improvement to search space size, but substantially increases runtimes.

As a way of improving the algorithm, then, reordering colour classes is at best a very limited success. However, as a way of improving our understanding the behaviour of Algorithm 2.1, it is much more significant: future attempts at improvements to this algorithm need no longer be complicated by the incorrect claims made by Tomita and Kameda (2007) as to why the reverse colour ordering should be used. For example, if Algorithm 2.1 is to be used for the clique decision problem, then it is worth considering Walsh's (1998) suggestion

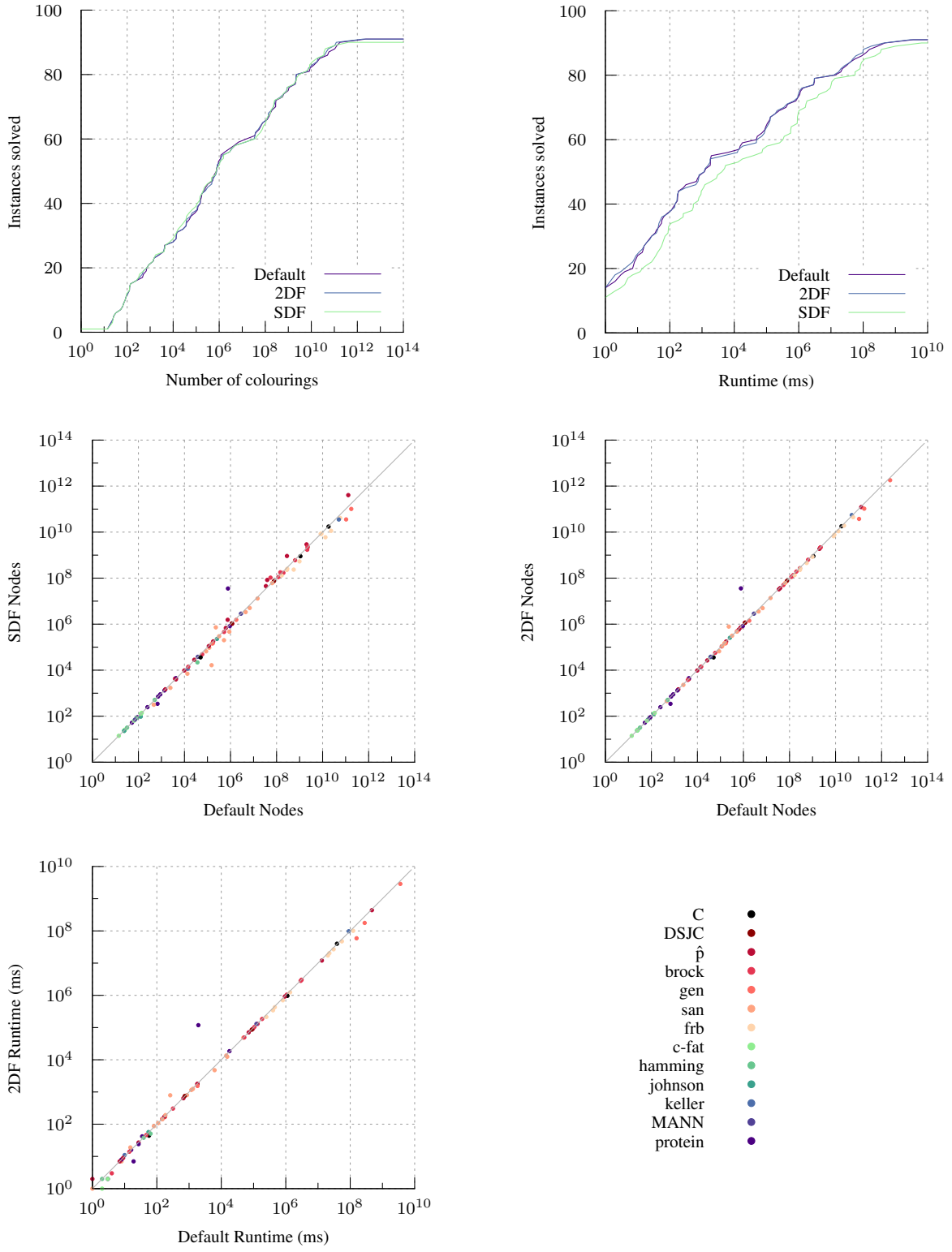


Figure 2.14: On the top row, cumulative number of instances solved as a function of search nodes (left) and runtimes (right), using different domain ordering strategies. Below, instance-by-instance comparisons. As discussed in the text, the improvements are hard to see, particularly on a log scale; Table 2.2 presents these results in tabular form.

Table 2.2: The effects of reordering colour classes, on the easier clique problem instances. We show the number of colourings (equivalently, recursive calls) required using Algorithm 2.1, then, as a ratio, the number when using domains of size two first or smallest domain first heuristics. We then show runtimes, again using ratios for the second and third columns.

Instance	Colourings			Runtime (ms)		
	Default	2DF	SDF	Default	2DF	SDF
Randomly generated						
C125.9	50 240	0.715	0.717	57	0.772	4.561
C250.9	1 082 441 593	0.831	0.828	1 126 147	0.859	6.422
C2000.5	18 189 648 267	0.989	0.956	38 832 285	1.028	2.303
DSJC500_5	1 153 043	0.986	0.936	733	1.020	4.139
DSJC1000_5	76 981 458	0.988	0.961	87 921	0.975	3.700
Randomly generated with large degree spread						
p_hat300-1	1 480	0.997	1.001	1	2.000	5.000
p_hat300-2	4 256	0.962	0.935	7	1.000	5.286
p_hat300-3	624 947	0.929	1.096	666	0.959	7.455
p_hat500-1	9 777	0.993	0.994	9	1.000	3.556
p_hat500-2	114 009	0.950	0.979	174	0.989	5.454
p_hat500-3	39 260 458	0.936	2.116	70 925	0.999	14.404
p_hat700-1	26 649	0.994	1.063	27	1.000	3.407
p_hat700-2	750 903	0.957	2.058	1 830	0.973	9.810
p_hat700-3	282 412 276	0.953	3.241	935 998	0.985	14.291
p_hat1000-1	176 576	0.996	1.007	147	0.986	3.646
p_hat1000-2	34 473 978	0.948	1.311	94 978	0.956	6.181
p_hat1000-3	130 317 818 368	0.938	3.117	470 938 212	0.936	13.766
p_hat1500-1	1 184 526	0.997	0.874	1 799	0.991	2.285
p_hat1500-2	2 006 796 270	0.940	1.457	13 209 931	0.908	4.253
Randomly generated with large hidden solutions						
brock200_1	524 723	0.938	0.875	316	0.968	6.079
brock200_2	3 826	0.984	1.119	4	0.750	4.750
brock200_3	14 565	0.979	0.966	14	1.000	5.071
brock200_4	58 730	0.964	0.832	47	0.979	4.000
brock400_1	198 359 829	0.970	0.853	184 360	1.000	4.875
brock400_2	145 597 994	0.936	1.230	133 714	0.974	6.796
brock400_3	120 230 513	0.943	0.928	106 134	0.976	5.291
brock400_4	54 440 888	0.922	1.931	51 592	0.958	10.948
brock800_1	2 227 634 634	0.973	1.046	3 080 308	0.971	3.479
brock800_2	2 235 803 416	0.973	0.970	3 083 366	0.989	3.284
brock800_3	2 146 717 172	0.973	0.802	2 890 338	0.972	3.289
brock800_4	640 444 536	0.979	0.938	1 075 174	0.974	3.880
gen200_p0.9_44	1 774 374	0.802	0.873	1 818	0.846	7.107
gen200_p0.9_55	170 254	0.862	0.859	178	0.933	6.180
gen400_p0.9_55	2 353 914 262 613	0.767	0.635	3 585 700 777	0.803	4.048
gen400_p0.9_65	175 757 037 249	0.595	0.581	280 143 536	0.632	3.598
gen400_p0.9_75	104 883 350 585	0.357	0.341	157 187 491	0.376	2.165
san200_0.7_1	13 399	1.001	0.524	15	1.267	3.267
san200_0.7_2	464	0.976	0.679	1	1.000	3.000
san200_0.9_1	87 329	0.761	0.754	81	1.074	6.062
san200_0.9_2	229 567	3.409	3.168	260	3.031	20.762
san200_0.9_3	6 815 145	0.728	0.741	6 189	0.766	6.252

continued on next page...

Instance	Colourings			Runtime (ms)		
	Default	2DF	SDF	Default	2DF	SDF
san400_0.5_1	2 453	0.932	0.693	8	0.875	1.500
san400_0.7_1	119 356	0.942	0.794	147	0.966	5.150
san400_0.7_2	889 125	0.899	0.528	1 313	0.947	2.135
san400_0.7_3	521 410	0.907	0.386	859	0.932	1.077
san400_0.9_1	4 536 723	0.788	0.739	15 285	0.813	4.710
san1000	150 725	0.983	0.109	1 178	0.982	0.075
sanr200_0.7	152 882	0.956	0.938	110	0.982	4.900
sanr200_0.9	14 921 850	0.915	0.872	14 323	0.943	7.325
sanr400_0.5	320 110	0.985	0.933	185	1.032	4.341
sanr400_0.7	64 412 015	0.954	0.942	48 319	1.003	5.297
Randomly generated with known solution sizes						
frb30-15-1	292 095 125	0.886	0.868	456 067	0.939	3.953
frb30-15-2	557 252 809	0.818	0.422	811 184	0.866	2.058
frb30-15-3	167 116 178	0.807	0.785	249 959	0.855	3.659
frb30-15-4	991 460 271	0.852	0.540	1 363 978	0.910	2.700
frb30-15-5	282 763 799	0.798	0.808	402 948	0.853	3.974
frb35-17-1	13 273 030 824	0.827	0.449	30 541 366	0.880	1.835
frb35-17-2	23 358 937 783	0.823	0.488	55 226 300	0.860	2.009
frb35-17-3	8 248 153 344	0.808	1.014	20 187 054	0.855	3.961
frb35-17-4	8 850 406 216	0.866	0.957	22 558 406	0.902	3.636
frb35-17-5	58 010 454 258	0.765	0.710	123 320 497	0.819	2.975
Fault diagnosis						
c-fat200-1	24	1.000	0.958	0	1.000	1.000
c-fat200-2	24	1.000	1.000	0	1.000	1.000
c-fat200-5	139	1.000	1.000	1	0.000	2.000
c-fat500-1	14	1.000	1.000	0	1.000	1.000
c-fat500-2	26	1.000	1.000	1	0.000	0.000
c-fat500-5	64	1.000	1.000	2	0.500	1.000
c-fat500-10	126	1.000	1.000	3	0.667	1.667
Coding theory						
hamming6-2	32	1.000	1.000	0	1.000	1.000
hamming6-4	82	1.000	1.000	0	1.000	1.000
hamming8-2	128	1.000	1.000	2	1.000	2.000
hamming8-4	36 452	1.009	0.600	39	0.974	2.769
hamming10-2	512	1.000	1.000	64	0.797	1.109
johnson8-2-4	24	1.000	0.958	0	1.000	1.000
johnson8-4-4	126	1.000	0.762	0	1.000	1.000
johnson16-2-4	256 100	1.000	0.888	55	1.036	4.764
Keller conjecture						
keller4	13 725	0.987	0.841	10	1.100	2.700
keller5	50 707 104 364	1.088	0.697	90 244 319	1.081	2.608
Steiner triple problem						
MANN_a9	71	1.000	1.000	0	1.000	1.000
MANN_a27	38 019	1.000	1.000	172	1.000	7.122
MANN_a45	2 851 572	1.000	1.000	123 226	1.056	2.876
Proteins						
1KZKA_3KT2A_78	247	1.000	1.000	3	0.667	2.333

continued on next page...

Instance	Colourings			Runtime (ms)		
	Default	2DF	SDF	Default	2DF	SDF
1allA_3dbjC_41	675	0.513	0.513	19	0.368	0.737
1f82A_1zb7A_5	716	0.994	0.994	27	0.889	2.074
2FDVC_1PO5A_83	1 348	0.978	0.978	35	1.200	2.629
2UV8I_2J6IA_13107	4 263	1.048	1.053	8	1.000	7.500
2W00B_3H1TA_10858	777 428	46.179	45.645	1 939	61.076	494.224
2W4JA_2A2AD_0	890	0.999	0.999	16	1.000	2.625
3HRZA_2HR0A_476	934 965	0.879	0.879	17 899	1.031	4.283
3P0KA_3GWLB_0	90	1.000	1.000	0	1.000	∞
3ZY0D_3ZY1A_110	52	1.000	1.000	0	1.000	1.000

of reversing heuristics away from trying to fail quickly when an instance is expected to be satisfiable. Another implication of these results is discussed in the following section.

2.5 Other Enhancements

Algorithm 2.1 shows the key features of most recent maximum clique algorithms: candidate solutions are constructed by accepting then rejecting vertices, and a colouring is used both as a bound and as an ordering function. We discussed how to reorder colour classes to improve the performance slightly. This algorithm has been adapted and enhanced in various other ways—we now give a brief overview of some recent research. None of these variations is a clear winner universally, but each gives improvements on certain instances.

Initial vertex ordering Algorithm 2.1 begins by permuting vertices by degree order. This in turn alters the colouring produced, by determining the order in which vertices are coloured on line 26. The justification for using a non-increasing degree order is that such an approach tends to produce reasonably tight greedy colourings. Using a static degree order which is computed at the top of search is much cheaper computationally than recalculating degrees dynamically.

Other vertex orderings have been proposed. Prosser (2012) considers three: the simple non-increasing degree order we describe, a minimum-width or degeneracy ordering (Eppstein and Strash, 2011; Freuder, 1982; Matula and Beck, 1983), and non-increasing degree with tiebreaking on the accumulated degree of neighbours. There is no clear winner: the best algorithm varies between both problem families and problem instances, and is also dependent upon the exact choice of colouring algorithm used.

After Prosser's (2012) study, San Segundo, Lopez, and Batsyn (2014) proposed a vertex ordering which is between degree and minimum width. Their approach has a magic parameter k controlling which ordering dominates, which must be selected to be "neither too small, nor too big". With an appropriate Goldilocks value for k, the results appear promising, offering a modest improvement for most of the instances considered. Building upon this, San Segundo,

Lopez, Batsyn, et al. (2016) propose selecting between different colouring procedures based upon a combination of the density of the graph, and the quality of colourings produced at the top of search.

It is not entirely clear what the purpose of these orderings is: there have been several attempts at determining good strategies for producing greedy colourings (Brélaz, 1979; Kubale and Jackowski, 1985; San Segundo, 2012; Sewell, 1993), and we might think that producing a tight colouring at the top of search should give best results. However, this is not the case: producing a better initial colouring will often increase the overall size of the search space, rather than reduce it. This could be because of the close coupling between branching and the order of vertices in colour classes, or because producing a good colouring initially does not imply we will continue to produce good colourings after branching and rejecting some vertices. To avoid the latter, we could also consider a dynamic vertex ordering: we could recalculate degrees in the subgraph induced by $remaining$ before each colouring, rather than reusing a single ordering throughout search. However, maintaining degree information during search is prohibitively expensive. To offset this cost, Tomita, Yoshida, et al. (2016) suggest recalculating a vertex ordering at each level near the root of the search tree; an empirically derived formula is given to determine when this should be done.

We must also be wary of interactions between parts of the algorithm: we cannot consider vertex orderings independently of the mechanism used to produce the colouring. Prosser (2012) shows that, all other things being unchanged, a minimum width ordering produces better results than either degree-based ordering on random graphs when using the greedy colouring routine which we described in Algorithm 2.1, but worse results when using a slightly different procedure due to Tomita and Kameda (2007).

A further complication in comparing results comes from how exactly the ordering is calculated. The degree spread in many graphs is fairly low, and even with tiebreaking mechanisms, the final order is usually not uniquely determined by the sorting criteria. Prosser (2012) explicitly uses vertex number as a final tiebreaking rule, to ensure that results are not dependent upon quirks of whichever unstable sorting algorithm is used. We adopt this convention; other authors have not done so, and have used random tiebreaking. This can make a large difference in practice: for example, on the brock400 family of instances (whose vertices are not "naturally" ordered in any way), tiebreaking in reverse vertex order is around twice as bad on two instances, 20% worse on a third, and only 12% better on the fourth. Because of the relatively small number of instances in each of the DIMACS families, if we only looked uncritically at values in a table of results, we could mistakenly conclude that this tiebreaking strategy is a genuine improvement for certain kinds of graph.

Tighter colourings We could also try to produce tighter colourings by going beyond a simple greedy colouring. Tomita, Sutani, et al. (2010) propose a repair mechanism, which

works as follows. When performing a vertex-by-vertex greedy colouring, if we must open a new colour class for the active vertex, we first check why. If there is only a single conflict between the active vertex and some colour class, and if this conflicting vertex may be placed in a later colour class without causing a new conflict, then we perform this exchange instead and do not open a new colour class. San Segundo, Matía, et al. (2013) explain how to adapt this approach to a bit-parallel setting.

Prosser's (2012) study suggests that, when compared independently of other changes, the colour repair mechanism is often able to reduce the size of the search space somewhat, but often at the expense of runtimes (particularly on larger or denser graphs).

Alternatives to colourings Going further in this direction, we could consider alternatives to using colouring as a bound. One possibility is to use MaxSAT, which is able to detect inconsistencies between colour classes and identify (for example) cases where three vertices are differently coloured but do not form a triangle. This can produce bounds which are tighter than a greedy colouring, and which are potentially tighter than even an optimal colouring.

When using a MaxSAT solver, the results are not generally competitive due to overheads and the size of the SAT representation. Also, using a solver restricts us to producing a single colouring for the encoding at the top of search, and we cannot modify the encoding using new colourings as vertices are rejected during search. However, using MaxSAT-inspired bounding inside a colour class algorithm can produce better results for certain classes of graph (C. Li, Z. Fang, and K. Xu, 2013; C. Li, Hua Jiang, and Manyà, 2017; C. Li, Hua Jiang, and R. Xu, 2015; C. Li and Quan, 2010a; C. Li and Quan, 2010b). Recent work by San Segundo, Artieda, León, et al. (2016), San Segundo, Nikolaev, and Batsyn (2015), and San Segundo, Nikolaev, Batsyn, and Pardalos (2016) suggests that a very limited detection of inconsistent (non-clique-forming) subsets of coloured vertices using a weakened form of MaxSAT reasoning may be a better approach. The gains from doing so are typically modest, reducing both the search space and runtimes by less than a factor of two; however, when combined with a heuristic lower bound, this technique can close the smallest set of BHOSLIB instances without search.

There are other possibilities: Balas and Xue (1996) and Wood (1997) used fractional colourings as a bound, which can be tighter than a conventional colouring. Another interesting bound is from the Lovász theta function (Knuth, 1994), which always lies in between the clique and colouring numbers, and is computable in polynomial time; however, the polynomial and associated constant factors appear not to be practical.

Faster colourings Another possible avenue for improvement is in performing colourings faster. San Segundo, Tapia, and Lopez (2013) describe a scheme inspired by watched literals to avoid recalculating information during the colouring process. Batsyn et al. (2014) observe

that statically preallocating space can avoid the cost of memory allocations (the various iterations of our parallel implementation have always done this as a consequence of the data structures and compilation methods used for bitsets which we described in Section 1.6.1, but the technique was not explicitly reported).

Because nearly all of the execution time is spent performing colourings on non-trivial instances, we could try to put less effort into the colouring process, even if this also increases the size of the search space. Konc and Janežič (2007b) reuse colourings at certain levels of the search, rather than producing a new colouring on every iteration. Similarly, Nikolaev, Batsyn, and San Segundo (2015) show that, with a newer algorithm, reusing a colouring under certain circumstances can reduce computational time by an average of around 30%, without severely increasing the search tree size.

We could also select different strengths of colouring dynamically. San Segundo and Tapia (2014) avoid computing a full colouring for colour classes which are over a branching threshold, and Tomita, Yoshida, et al. (2016) use a lighter colouring process near the leaves of the search tree, giving an empirically derived formula to decide when to do this.

Priming In Algorithm 2.1 we initialise the incumbent to be empty. Maslov, Batsyn, and Pardalos (2014) instead apply an iterated local search (ILS) heuristic to generate an initial solution. They run the heuristic 100,000 times, except for two instances (which are known to be difficult, but which had been solved exactly previously, and where 100,000 runs of the heuristic did not find the known solution) where they run the heuristic 60 million times instead (which was sufficient to find the known maximum clique). This strategy was reused by Batsyn et al. (2014). It is extremely effective on certain harder instances, for reasons which we will investigate further in the following chapter, but adds considerable overhead to trivial instances. When using such an approach we must also know in advance how many times we should run the heuristic—in general we do not have the benefit of knowing *a priori* what the solution is, then being able to select an appropriate number of times to run the heuristic such that that solution is found. Thus, although the effectiveness of this technique is interesting, we consider its use to be somewhat unprincipled.

This technique was also adopted by Tomita, Yoshida, et al. (2016), who use a different form of local search to prime the incumbent. They give a formula for how many iterations to run which is based upon the number of vertices and the density of the graph; the formula was derived empirically by considering a small set of instances, and it does not generalise to other families such as those in Chapter 4.

This chapter's results on reordering colour classes shed more light on this technique: previously experimenters assumed that Algorithm 2.1 tried to find strong solutions quickly and that heuristic solutions were a way of helping out. In fact, we saw that the algorithm tries hard to fail as soon as possible, because doing so leads to less time spent proving optimality

(which is often the most expensive part of the search). Priming, then, could provide a way of offsetting potential penalties of having a weaker incumbent when trying harder to fail sooner.

Fast clique detection Batsyn et al. (2014) observe that if we colour the first k vertices using k colours, then those vertices must form a clique. This can be used to avoid making recursive calls and performing new colourings during parts of the search.

Alternatives to backtracking search Östergård (2002) used a technique now commonly known as Russian dolls search (Verfaillie, Lemaître, and Schiex, 1996) to obtain a bound. In clique terms, the idea is as follows: we write the vertices out in some order $v_1, v_2, \ldots, v_n$. Now we find the maximum clique in the subgraph induced by v_1 (which necessarily is of size 1), then the subgraph induced by $\{v_1, v_2\}$ (which is either the same, or one higher and using only vertex v_2 and its neighbours), then $\{v_1 \ldots v_3\}$, and so on, until we arrive at the overall solution. The benefit of doing so is that at each stage, we may use the previously-solved subproblems to give a bound on the solution size: if v_k is the highest numbered undecided vertex which has not yet been rejected, then we cannot grow our current candidate solution by more than the solution we found for the subgraph induced by the first k vertices.

Corrêa et al. (2014) resurrect this idea and combine it with a colour bound. The reported results are somewhat faster on some DIMACS instances, although the results are compared to rescaled results from another paper. The reported runtimes for two of the "gen" instances are vastly superior, since their technique finds the hidden solutions quickly. Araujo Tavares (2016) shows that Russian dolls search is particularly effective for the maximum weight clique problem.

Bergman et al. (2014) propose the use of decision diagram search. This technique allows for the merging of equivalent or weaker states during search, pruning the search space. However, their results are extremely poor: with 256 processor cores, every single runtime reported is worse than the single core runtimes of Algorithm 2.1, usually by many orders of magnitude.

Lazy global domination In Chapter 4 we propose a different technique based upon eliminating redundant or subsumed states, which we call *lazy global domination*. As this technique gives no improvement to the problem instances discussed here, and appears only to be useful for exploiting unusual properties problem instances created by a reduction from a different problem, we defer discussion until then.

Parallel search As well as bit-parallelism, these algorithms can be parallelised to make use of multiple threads or processors. We discuss this in detail in the following chapter.

Other problems In Chapters 4 and 7 we will look at using clique algorithms to solve distance relaxations of clique, and maximum common subgraph problems. Chapter 4 also discusses modifying these algorithms to solve a clique problem with side constraints, and a bipartite version of the clique problem.

2.5.1 Other Related Work

We now briefly discuss other related work which is relevant for the remainder of this thesis. Wu and Hao (2015) give a broader review, including work on non-exact methods which we do not study.

Sparse graphs Because of the use of an adjacency matrix and a quadratic (or higher cost) bounding operation, Algorithm 2.1 and its variants are only suitable for relatively small, dense graphs (although we will see in Chapter 4 that "relatively small" can still be many tens of thousands of vertices). There is considerable interest in an algorithm which is more suitable for large, sparse graphs—see, for example, recent work by Hua Jiang, C. Li, and Manyà (2016), Pattabiraman et al. (2013), Pattabiraman et al. (2015), San Segundo, Artieda, Batsyn, et al. (2017), San Segundo, Lopez, Artieda, et al. (2017), and San Segundo, Lopez, and Pardalos (2016).

The justification for these algorithms is that real-world graphs are sparse. It is worth noting, however, that most uses of clique algorithms appear to be on encoded graphs, rather than on graphs that directly represent real-world phenomena, and encoded graphs often do not have similar properties to their inputs. For example, the encodings we use in Chapter 4 and Chapter 7 both turn sparse input graphs into dense graphs. In practice, even the simplest of polynomial reductions can severely impact solver behaviour. For example, the reductions from vertex cover and independent set to clique change sparse graphs into dense graphs—this goes some way towards explaining why Akiba and Iwata (2016) saw such poor performance from a clique algorithm on vertex cover instances in large sparse graphs.

Preprocessing In a more detailed look at the results of Akiba and Iwata (2016), Strash (2016) observes that preprocessing input graphs can sometimes be hugely beneficial, even when simplification rules are only applied at the top of search. In Chapter 4 we explore this further, and look at using laziness to avoid paying upfront costs of simplification.

Enumeration The maximal clique enumeration problem is to list every maximal clique in a graph (that is, every clique which cannot be made larger by adding a vertex). Bron and Kerbosch (1973) provided the first algorithm for this problem, and variants of this algorithm remain the state of the art. An interesting related problem is *maximum* clique enumeration, which is discussed by Eblen et al. (2012). Although we do not consider enumeration problems

directly, one of the algorithms in Chapter 4 performs two passes to find a multi-objective solution, and the second pass in many ways resembles maximum clique enumeration.

2.6 Conclusion

We have looked at sequential algorithms for the maximum clique problem. We introduced Algorithm 2.1, which plays an important part in the remainder of this thesis. In Chapter 3 we will look at how to parallelise it to make use of multiple cores, and in Chapter 4 we will adapt the algorithm to solve three other problems. We also return to cliques when solving the maximum common subgraph problem in Chapter 7.

Our experiments have helped us uncover more about how modern maximum clique algorithms work. Building upon research on phase transitions and complexity peaks for decision problems, we now understand some of the shape of the complexity curve for the optimisation problem. It is known how to locate the clique phase transition analytically (Frieze, 1990). Following on from this, an interesting open question is whether we can explain the relative difficulty of different decision problems, thus obtaining a complete explanation for the shape of the complexity curve.

We now understand why the right-to-left branching order is used. We demonstrated that the explanation in the literature for this rule is incorrect, and used constraint programming concepts to propose an alternative justification which stands up to experimental scrutiny. This justification allowed us to design a slightly better branching rule. Another open question is whether further enhancements are possible, in particular with respect to picking vertices within colour classes (corresponding to value-ordering heuristics).

We looked in detail at the DIMACS benchmark instances usually used to compare maximum clique algorithms. On the one hand, having a diverse set of instances that are used across papers makes it easier to judge progress. On the other hand, upon closer inspection we saw that many of these instances are effectively random instances in disguise, sometimes with large hidden solutions. This suggests that we should treat massive speedups in the "brock", "gen" and "san(r)" families with a little skepticism—in particular, the best algorithm for the "gen" instances is simply whichever algorithm happens to find an optimal solution most quickly. This leads to real concerns with the design of arbitrary tie-breaking rules which just happen to favour these instances, and suggests that overfitting may be a problem in algorithm design. We could address this by creating larger numbers of instances using the original generators (some of which are publicly available), but this puts even more emphasis on random instances in benchmarking: will we end up simply measuring which algorithms happen to find hidden solutions quickly?

The "frb" family from the BHOSLIB suite does little to improve the situation: these are also crafted instances which are designed to be difficult. It is interesting that with the

right level of inference these instances suddenly become easy, in that being able to generate hard instance with known properties has potential applications in cryptography (Brockington and Culberson, 1993). However, it is unclear as to when stronger inference is more useful in general: more complicated bounding functions can raise the computational cost of each recursive call substantially. In Chapter 4 we will be looking at much larger problem instances where more complex inference is prohibitively expensive, to the point that we do not want to use even a single $O(n^3)$ step in the algorithm.

Despite this somewhat pessimistic view, it is worth emphasising that considerable progress *has* been made. Many of the instances that we find easy, such as the diagnostic and protein graphs (as well as the maximum common subgraph instances in Chapter 7), are only easy because of advances made in algorithm design, and many of those advances have come from studying hard synthetic instances. We will also be introducing new instances from other applications in subsequent chapters, and will see that with a few modifications, modern clique algorithms perform better than old clique algorithms on these new families. With this in mind, the emphasis on this chapter has primarily been about learning more about the basic algorithm, rather than simply introducing an improvement and demonstrating that it runs faster on some instances. Indeed, the main interest for our 2DF branching rule is not that it is a bit faster, but rather that it helps us understand why the algorithm works so well in practice.

Chapter 3

Parallel Maximum Clique

When parallelising an algorithm, we seek to divide work into a suitable number of equally sized, independent subproblems which may be evaluated simultaneously. Sequential branch and bound algorithms do not lend themselves to this approach, and parallel branch and bound algorithms must cope with extremely irregular subproblem sizes, speculative evaluation, and communication patterns which cannot be predicted in advance. Nonetheless, parallel branch and bound can be beneficial in practice: this chapter takes the sequential maximum clique algorithm from the previous chapter, and evaluates several multi-core parallel implementations. We will see that on modern multi-core hardware, any one of these parallel implementations is clearly better than the original sequential algorithm.

Our aim is not just to produce a faster maximum clique implementation, however; we look in detail at the choices available when parallelising a branch and bound algorithm, in the hopes of learning more general lessons. We show that different work splitting strategies can have a substantial effect upon performance, and that a careful study of the underlying sequential algorithm can help direct our design. We make use of the extra programming flexibility offered by multi-core systems to take measurements "inside the search" to demonstrate *why* some implementation choices give better or worse performance on some problems. We make three claims, and justify them experimentally. Firstly, we show that static work splitting can lead to balance problems. Secondly, we show that different search orders and work splitting mechanisms often have a large effect upon speedups, and that this is due to changes in "the amount of work done", rather than just imbalances and overheads. Thirdly, we show that an understanding of how ordering heuristics behave can explain why a simple fixed-depth work splitting mechanism usually does so well, despite occasional balance problems.

Using these results, we discuss how we may improve work balance whilst retaining the search-order benefits of a simpler work splitting mechanism. We present a novel work splitting mechanism which explicitly diversifies at the top of search, to offset poor early heuristic advice, and that resplits work later to improve balance. We show that, as our explanations predict, this gives better and more consistent speedups than other mechanisms.

This chapter is based upon McCreesh and Prosser (2015c), "The Shape of the Search Tree for the Maximum Clique Problem and the Implications for Parallel Branch and Bound". Prior to that work, we implemented a threaded version of a maximum clique algorithm similar to Algorithm 2.1 (McCreesh and Prosser, 2013). This closed three previously open DIMACS instances: the best-known cliques in "MANN_a81", "C4000.5" and "p_hat1500-3" were all shown to be optimal. Independently, Depolli et al. (2013) produced another threaded maximum clique implementation. Both approaches started with a similar sequential algorithm, both were implemented using C++11 native threads to explore subtrees in parallel, and both performed experiments on similar multi-core hardware. At a glance, both sets of results are similar: good speedups are achievable in practice (but the speedups achieved vary substantially between problem instances), and superlinear speedups (greater than n from n processors) sometimes happen—in other words, both papers show that adding parallel tree-search to a strong maximum clique algorithm is worth doing.

A close inspection of both sets of results shows that for some particular problem instances, there are substantial differences: for example, we obtained roughly linear speedups for the DIMACS graph "MANN_a45", whereas Depolli et al. achieved a speedup of below 4 regardless of the number of processor cores available. We will explain why this happened, and how this may be addressed without compromising other results (in particular by preserving the superlinear speedups seen on some instances).

We start by explaining how branch and bound algorithms may be viewed as a tree search process, and how we can treat behaviour "inside search" as different kinds of subtrees. We use this to explain the fundamental concepts underlying parallel branch and bound, and discuss the potential for speedups.

In Section 3.2 we look in more detail at the effects of design choices in practice. We show that the reason for Depolli et al.'s speedup limit of 4 for "MANN_a45" is due to poor load balancing. We then look in more detail at further problems. We will see that that balance does not severely restrict speedups in most cases, and that steps taken to improve the balance will often give much worse performance (and not just due to overheads). Our view of branch and bound as a tree, together with some knowledge of how heuristics behave, lets us explain this. We show that diversity (that is, using parallelism to hedge against weak early heuristic choices), not balance, is usually the primary contributing factor to performance.

Finally, in Section 3.3 we discuss how to get "the best of both worlds". Driven by a deeper understanding of the search process, we present a new work splitting technique which explicitly adds diversity early in search, to offset the weakest heuristic advice, followed by a low-overhead resplitting mechanism to even out balance problems later on in search. This approach is more than just beneficial: recalling the properties introduced in Section 1.6.6, it is also risk-free, scalable, and reproducible. It also generalises to other problems: in Chapters 4, 5 and 7, we show that this strategy remains beneficial for other subgraph problems.

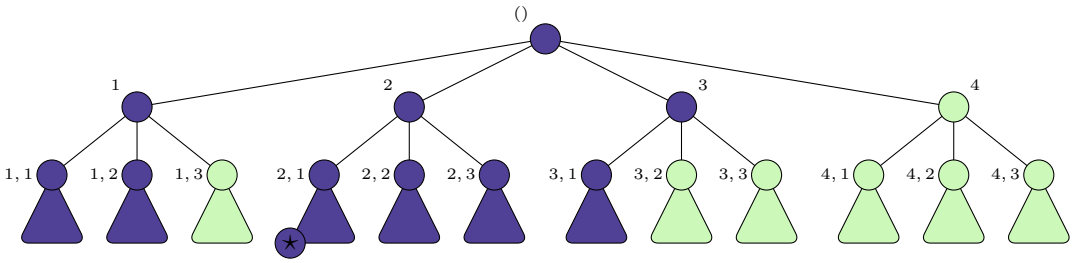


Figure 3.1: A possible search space of a branch and bound algorithm, viewed as a tree—we assume a depth-first search, iterating over children from left to right. Each node represents a recursive call; triangles represent large subtrees. The optimal solution is marked $\star$, at location $(2, 1 \ldots)$. Nodes which need not be explored if this optimal solution has already been found, which are said to be eliminable, are shaded lightly; nodes which must always be explored to prove optimality are shaded darkly. Here, the light subtree $(1, 3)$ is avoidable—it would be explored in the sequential run despite being eliminable.

3.1 Branch and Bound as a Tree

We may view the recursive calls made by a branch and bound algorithm as forming a tree, as in Figure 3.1. Here, each node in the tree represents a recursive call; triangles represent large subtrees (recall that we use *node* for search trees, and *vertex* for graphs). We mark by $\star$ the location of an optimal solution. Nodes shaded darkly are those which cannot be eliminated by the bound, regardless of the strength of the incumbent—we say such nodes are *ineliminable*. Nodes shaded lightly are those which could be eliminated by the bound, if $\star$ has been found—such nodes are *eliminable*. The tree is traversed in a manner similar to a depth-first search, exploring subtrees from left to right. Note that in a sequential run, the leftmost light-shaded subtree will *not* be eliminated by the bound, since the search will not yet have found $\star$. In other words, not all eliminable nodes are necessarily eliminated. We call eliminable nodes that are not eliminated in the sequential run *avoidable*.

In maximum clique terms, $\star$ is the location of a maximum clique, whose size is denoted ω. The dark shaded nodes are then the nodes which must be explored to prove that there is no clique of size $\omega + 1$ in the graph. Light shaded nodes are those that would be eliminated by the bound, if the algorithm were to be initialised with an incumbent size of ω rather than 0. For now we assume there is a unique optimum, and that the optimum must be known before any subtree may be eliminated by the bound. We will see later that this assumption is slightly too simple in some cases.

We may label nodes with lists of numbers as follows: the root node is labelled with the empty list, $()$. We label children of the root node from left to right as (1), (2), (3), etc. The children of (1) are labelled $(1, 1)$, $(1, 2)$, $(1, 3)$, and so on. Thus, in our figure, the optimal solution $\star$ is found at the location $(2, 1 \ldots)$, which corresponds to taking the second branch at the top of search followed by the first branch thereafter.

3.1.1 Parallel Branch and Bound

Usually when we wish to parallelise an algorithm, we look for independent units of work which may be evaluated independently. With branch and bound we cannot take this simple approach, and we must resort to speculatively executing non-independent subproblems to be able to exploit parallel hardware. We view branch and bound as a tree search, ignore left-to-right dependencies, and explore different subtrees in parallel. There is a single incumbent which is shared between every worker. This technique in general is well known—see, for example, discussions by Bader, Hart, and Phillips (2005) and Crainic, Le Cun, and Roucairol (2006). For the maximum clique problem, it was first attempted by Pardalos, Rappe, and Resende (1998), where message passing was used on a cluster of four machines, and later, by McCreesh and Prosser (2013) and Depolli et al. (2013) using threads. Another recent parallel maximum clique algorithm by R. A. Rossi et al. (2013) does the same, starting with an algorithm designed for large sparse graphs; again, superlinear speedups were observed. Parallel tree-search was also used by Debroni et al. (2011) to close the Keller maximum clique problem, using an algorithm which exploited special properties of these graphs.

Other approaches have been considered for maximum clique. Xiang, Guo, and Aboulnaga (2013) used MapReduce rather than conventional parallel branch and bound. Their emphasis is upon the scalability of their solution: they demonstrate linear speedups (over a parallel implementation, not a sequential one) on three graphs from one DIMACS family, and on one further graph. We will see that other families of graphs have properties which can interfere with this approach. Bergman et al. (2014) used parallel decision diagrams, and showed excellent scalability—however, their sequential runtimes were much worse than state-of-the-art algorithms, and with 256 cores their results never beat the sequential Algorithm 2.1 on a single core.

3.1.2 The Potential for Speedup

With the parallel tree-search approach, we are not dividing a fixed amount of work between processor cores. Thus we should not always expect to gain a speedup approaching n from n cores (such a speedup is said to be *linear*). Instead, the speedup obtained will depend not just upon the algorithm used, but also the nature of the input. There are various possible outcomes, which we illustrate in Figure 3.2:

- The search space consists entirely of nodes that cannot be eliminated—discovering the optimal solution does not provide any benefit to proving optimality. In this case, we *are* dividing a fixed amount of work up between workers, and may hope for a linear speedup.

- The search space contains many eliminable nodes that are eliminated on the sequential run, but are not eliminated in the parallel run. Our additional workers end up contributing

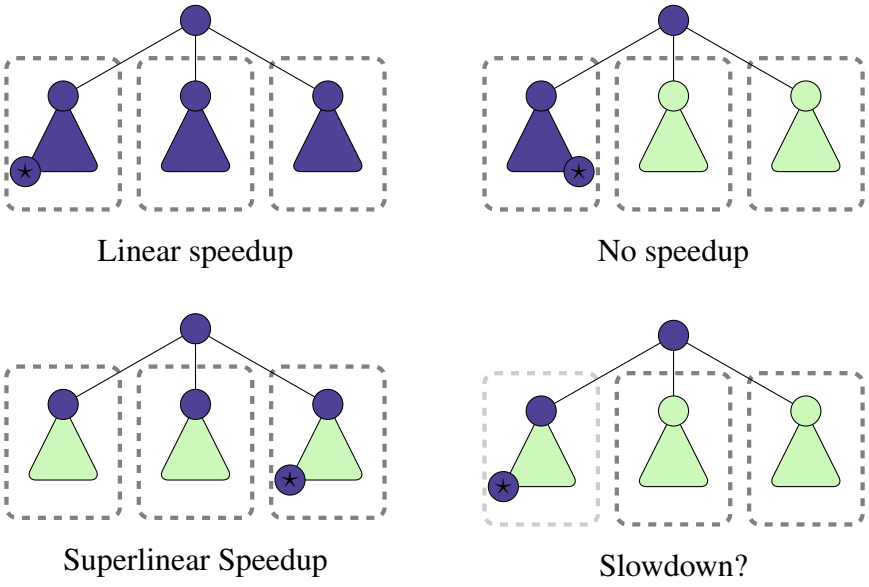


Figure 3.2: Possibilities for speedup in parallel branch and bound. Here we show the search space being divided between three workers. If there are no eliminable nodes, we are dividing up a fixed amount of work, and may hope for a linear speedup. If our workers spend their time exploring avoidable nodes, as in the top right example, we would get no speedup at all. Conversely, a worker may find an optimal solution much more quickly than in the sequential run. This may lead to avoidable nodes being eliminated in the parallel run, giving a superlinear speedup. Finally, the bottom right example shows the perils of exploring the tree "in a different order" to the sequential run: if only two workers are available, and exploring the leftmost subtree is deferred, we would explore many more nodes in total than in a sequential run.

nothing to the solution, so we get no speedup at all.

- The search space contains many avoidable nodes (eliminable nodes that are not eliminated on the sequential run) that are eliminated in parallel due to one of the additional workers finding an optimal solution quickly. Here we may get a superlinear speedup (Lai and Sahni, 1984; Trienekens, 1990).

- We could start by allocating all our workers to explore portions of the search space that would be eliminated in the sequential run. This could lead to a slowdown. Intuitively, one might think this is due to the tree being explored "in a different order" in parallel. This possibility is indeed avoidable—we must ensure that incumbents are discovered at least as quickly in parallel as they are in the sequential run, and that newly discovered incumbents are shared between threads. We refer to works by de Bruin, Kindervater, and Trienekens (1995), G. Li and Wah (1984), G. Li and Wah (1986), and Trienekens (1990); the injectivity requirements are not relevant for the maximum clique problem, since all cliques of a given size are equivalent for inference purposes.

Both our approach and that by Depolli et al. meet the conditions for avoiding a slowdown (with one technicality: it is possible for colourings of a subproblem to be worse than colourings

of a parent problem—we have been unable to observe this having an effect in practice, although it can easily be worked around). The approach by Xiang, Guo, and Aboulnaga (2013) does *not* meet these conditions: they do not preserve sequential search order, and they do not share newly discovered incumbents until a subproblem has finished. We will see in the following section why this was not a problem for the three DIMACS graphs that they considered.

Note that these properties may only be categorised after the fact—when the algorithm finds the optimal solution, it does not yet know that there is nothing better. Furthermore, in practice, we would not expect problems to fall neatly into one of these categories—we could both explore some additional eliminable nodes before a solution is found, and avoid some avoidable nodes. The matter is further complicated by the possibility of multiple solutions, and of "quite good" solutions that allow some but not all of the eliminable nodes to be eliminated. Our outcomes only represent the extremes in behaviour that we might observe.

3.2 Do Details of Parallel Algorithm Design Matter?

Having discussed what could happen in theory, we now show that these concerns are often relevant in practice. But first we briefly discuss our experimental setup and test data.

3.2.1 Experimental Setup and Data

Both McCreesh and Prosser (2013) and Depolli et al. (2013) considered problem instances from the Second DIMACS Implementation Challenge. We also considered random graphs and graphs from BHOSLIB, whilst Depolli et al. introduced a series of protein product graphs. We discussed these instances in the previous chapter, and will use all three families in this chapter too.

For scalability experiments, we work with a machine with four AMD Opteron 6366 HE processors and 512GBytes RAM, running Fedora Linux 18. Each processor has eight modules, each containing two cores, for a total of 64 cores. (Note that resources are shared between pairs of cores in a module, so it is not the case that we have 64 times as much processing power as is found in a single core used on its own.) The AMD Turbo Core feature was disabled, to allow us to investigate scalability effects. We emphasise that each core individually is not especially fast, compared to the systems previously used by either McCreesh and Prosser (2013) or Depolli et al. in their experiments—our goal with these experiments is not to solve new instances or to solve existing instances faster, but rather to look at the consequences of algorithm design choices. Speedups are given over a sequential implementation, not a parallel implementation with one thread. Our compiler is GCC 4.9.0, and we use C++11 with native threads.

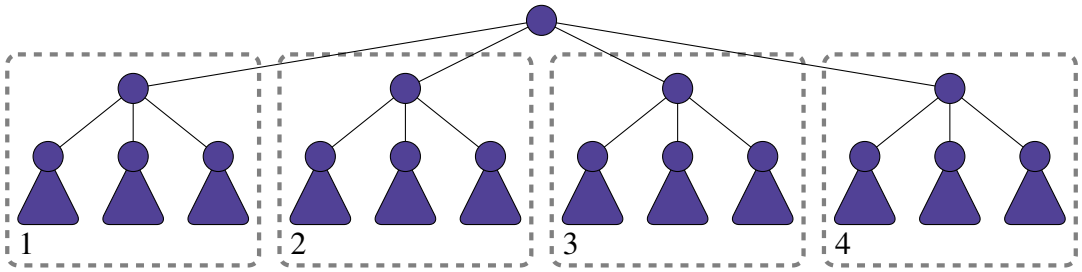


Figure 3.3: Splitting work at the top: we divide the work into four jobs by splitting the tree at distance 1 from the root. Workers tackle the subproblems in the order shown.

3.2.2 The Importance of Good Work Splitting

We have assumed so far that we are able to divide work between processors; we now discuss how to do this. A number of approaches are possible. Both McCreesh and Prosser and Depolli et al. started by splitting work "at the top" of search, as in Figure 3.3. Since the number of vertices in a graph is expected to be much larger than the number of cores available, this produces more jobs than there are workers. We explicitly placed each subtree of distance 1 from the root onto a queue (using an additional thread, and blocking when the queue contained too many items), and had workers process subtrees from this queue in order. Depolli et al. instead started by processing the root node, and when branching, would process the "take v" case in one worker, and pass the "do not take v" case on to a one-item queue to be processed by the next available worker. (We also had a work donation mechanism for further splitting later on in search; we ignore this for now.)

Such an approach assumes that subtrees will be sufficiently numerous and of similar size to allow an even work distribution. We could imagine a situation like the one shown in Figure 3.4 occurring, where the cost of evaluating one subtree dominates the runtime. Informally, we say that a workload is *balanced* if each thread has roughly the same runtime. We suggest that balance is generally a good thing.[1]

Depolli et al. claim their solution offers "low idle time and good load-balancing", and for most problem instances this is indeed the case. However, we will now show that a balance problem is the reason they were unable to achieve a speedup of 4 for the DIMACS graph "MANN_a45" regardless of the number of processors used. This contradicts their speculation that the density and large maximum clique size of the graph was to blame for the atypically poor speedup.

Figure 3.5 plots speedups obtained by our implementation as the number of worker threads increases (speedups are measured over a good sequential implementation, not a parallel implementation run with one thread). The bottom line shows what happens when splitting work at distance 1, without the work donation mechanism—we see that our speedups

[1] Of course, one could cheat and achieve a supposedly perfect balance by having threads that would otherwise be idle perform useless work instead. Thus, like counting recursive calls, measures of balance should be used for enlightenment, not for comparisons.

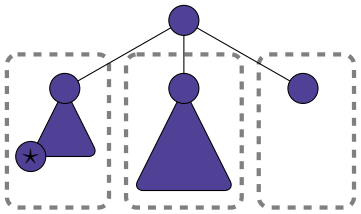


Figure 3.4: Subtrees can be highly irregularly sized. Here the search space contains no eliminable nodes, so if we divided the work between three processor cores as shown we might expect a speedup approaching three. But if work is not split dynamically, the runtime may be dominated by the cost of exploring the large subtree in the middle.

appear to be capped at around four.[2]

The first graph in Figure 3.6 shows the gap between the runtime of the shortest running and the longest running threads, as the number of threads increases. We would expect the gap between the two lines to be small if a good balance has been achieved; here we see that splitting at distance 1 gives a reasonable balance only for up to four threads.

The bottom right graph shows runtimes of individual threads, when 32 threads are used. A perfectly balanced work distribution would give a horizontal line. Here we see the longest running thread working for $107\,\mathrm{s}$, with the second longest finishing after $79\,\mathrm{s}$, and there are only six further threads with runtimes over $10\,\mathrm{s}$. The sequential runtime is $438\,\mathrm{s}$, and our speedup is capped at ${}^{438}/_{107} \approx 4.1$.

One possible workaround is to split the tree further from the root, in the hopes that this will give a better balance. For example, it is trivial to modify our implementation to split at distance 2 from the root, as in Figure 3.7. Figures 3.5 and 3.6 shows that doing so solves the balance problem for "MANN_a45" for up to fifteen threads, beyond which again the runtime is determined by the size of the largest subproblem. Going one step further, and splitting at distance 3 from the root (also shown) gives a sufficiently balanced distribution to allow effective use of 64 cores. This approach is also taken by Xiang, Guo, and Aboulnaga (2013), with a splitting distance chosen at the start of search based upon estimates of the sizes of subtrees.

The principle underlying this approach is that we may solve the irregularity problem by statically breaking the problem up into many more pieces than we have cores at the top of search, and then distributing work dynamically, so that an even balance is obtained automatically. A similar approach known as "embarrassingly parallel search" has been used in a constraint programming setting by Malapert, Régin, and Rezgui (2016), Menouer et al. (2016), Régin, Rezgui, and Malapert (2013), and Régin, Rezgui, and Malapert (2014), with very favourable results on a range of standard constraint programming problems (but

[2]Strictly speaking, it is a coincidence that our limit of "around 4" and Depolli et al.'s figure of "between 3 and 4" both involve the number 4. The two sequential algorithms used different initial vertex orderings, and the implementations give different rates of nodes per second at different depths in the graph, so we should not expect to see exactly the same limit in both cases.

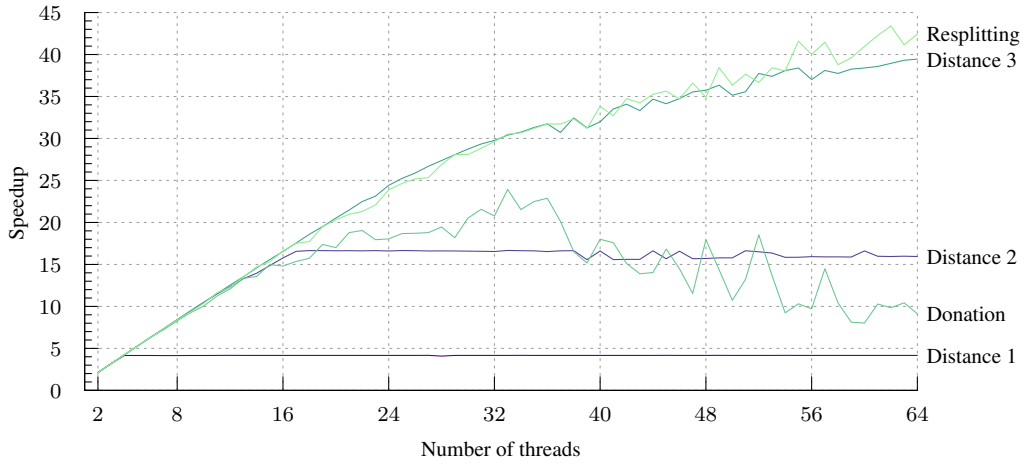


Figure 3.5: Speedup obtained as the number of threads increases for the DIMACS instance "MANN_a45". We see that with no work donation, splitting at distance 1 limits the speedup to around 4 regardless of the number of threads used, and splitting at distance 2 limits the speedup to 16. Splitting at distance 3 gives a typical speedup curve. Also shown are results for the work donation approach proposed by McCreesh and Prosser (2013), which evidently has scalability limits at this range with this hardware, and a more scalable approach described at the end of this chapter. The sequential runtime is 438 s.

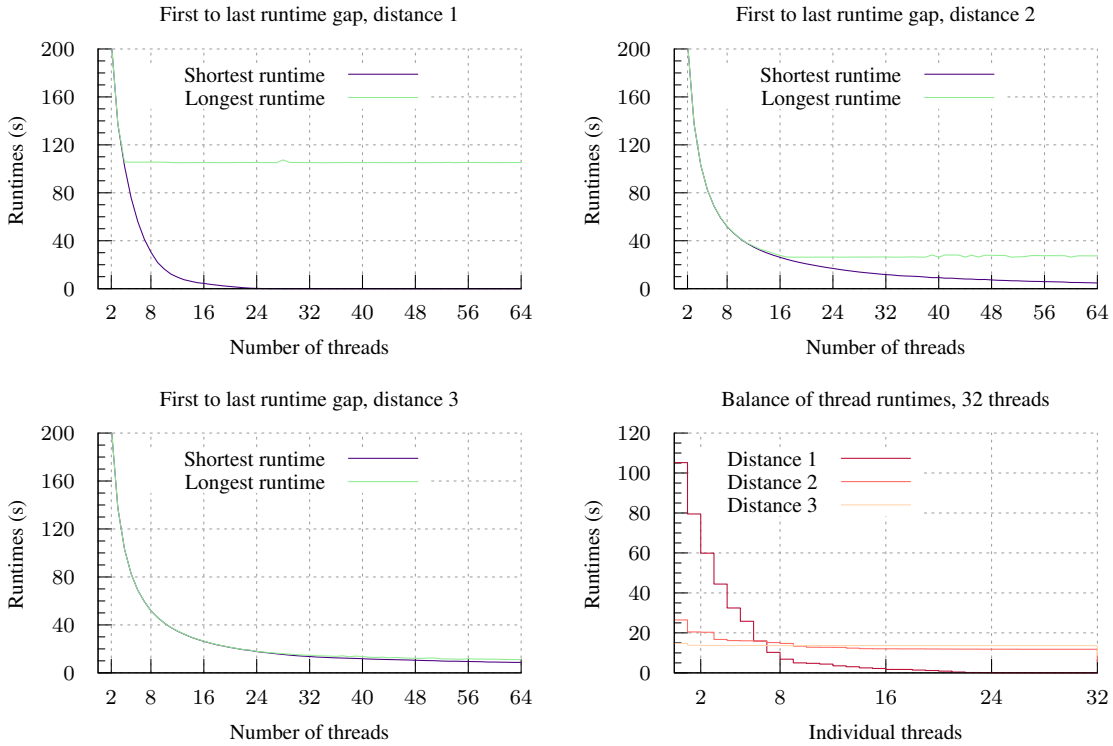


Figure 3.6: The first three graphs show runtimes of individual worker threads as the number of threads increases for the DIMACS instance "MANN_a45", with different splitting distances. The top line shows the runtime of the longest thread, and the bottom line the shortest. The fourth graph shows the runtimes of each thread when using 32 threads. We see that as the splitting distance increases, the balance improves.

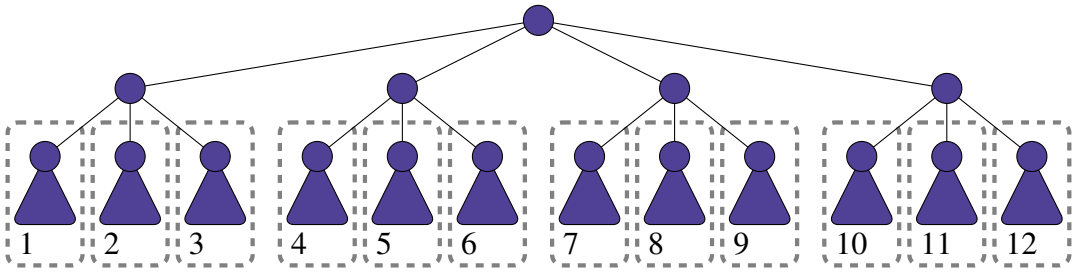


Figure 3.7: Splitting work at level 2. We might hope that this would lead to a more even work distribution than in Figure 3.3. But we will see that doing so does not avoid the problems with irregularly sized subtrees, and removes one of the benefits of splitting work at the top.

maximum clique was not considered).

We could also consider splitting work entirely dynamically (Sanders, 1995). If the algorithm is refactored to perform a binary search, this is conceptually simple: before recursing down any "left" branch, the implementation can check whether any thread is out of work, and if one is, then it is given the "right" branch to evaluate. More generally, the splitting could be left to a high level library or language feature—we will investigate this in Section 3.3.2, using Intel's Cilk Plus. It is important to note that the splitting here is non-deterministic: we will see that this can lead to large variations in runtimes when solving the same problem instance several times.

Finally, we could try to estimate in advance how long each subproblem will take, as Xiang, Guo, and Aboulnaga (2013) do, and use these estimates to partition work statically. However, accurately estimating the runtime of a subproblem is extremely difficult. Otten and Dechter (2017) demonstrates the state of the art in this area: they solve large and/or branch and bound problems on a computational grid which has severe limits on communications, and must put much more effort into obtaining a good balance up-front than approaches where dynamic balancing is possible.

Unfortunately, for branch and bound algorithms all of these alternatives have extremely undesirable effects. Looking ahead to Figure 3.8, we see the opposite of what is shown in Figure 3.5: for other instances, splitting at distance 3 is much worse than splitting at distance 1. This is because deeper splitting does not just affect balance; as a side effect, the search order is also changed. When splitting at distance 1, we explore the subtrees at (1), (2), (3) and so on, in that (parallel) order. But when splitting at distance 2, we would instead start by exploring the subtrees $(1, 1)$, $(1, 2)$, $(1, 3)$ and so on. In a hypothetical situation such as Figure 3.1 where a solution is located at $(2, 1, 1 \ldots)$, splitting at distance 1 will lead to a solution being found immediately, but splitting at distance 2 will not. In practice, changing the parallel search order can often have much more significant effects than a potentially improved balance. The remainder of this chapter explains this in more detail, and shows how to get close to "the best of both worlds" (these are the remaining lines in the speedup plots).

3.2.3 Does Parallel Search Order Matter?

Different parallel search orders only matter if we are not dividing up a fixed amount of work—that is, if we are not in the situation shown in the top left of Figure 3.2. If there are not many eliminable nodes, exploring the search tree in any order which does not violate Trienekens's (1990) conditions for avoiding a slowdown is acceptable. However, we saw in Chapter 2 there are in fact many *avoidable* search nodes in most instances. Thus, even after ensuring that a slowdown cannot happen, parallel search order is an important consideration.

In Table 3.1 we redisplay some of the information from Table 2.1 on page 41, listing the easier DIMACS and BHOSLIB instances, and Depolli et al.'s protein product graph instances. As before, we show the size of a maximum clique, ω, and the number of search nodes (that is, the number of recursive calls made). Next is the number of search nodes that it takes to prove that the graph does not contain a clique of size $\omega + 1$; we find this total by rerunning the algorithm with the size of $incumbent$ initialised to ω rather than 0. In other words, this is the number of nodes in ineliminable subtrees. We then present two new pieces of information: we show the proportion of the search space which consists of avoidable nodes, and the location of the solution, which we discuss below.

For some DIMACS instances (e.g. "hamming8-4", "johnson16-2-4", the "keller" and "MANN" graphs and "sanr400_0.7"), we see that there are very few avoidable nodes, so parallel search order does not matter. The three DIMACS graphs considered by Xiang, Guo, and Aboulnaga are also in this category.

But for more than half of the DIMACS instances, at least a third of the search space is avoidable. For "san400_0.5_1" and "san1000", the entire search space is avoidable—that is, the bound at the top of search is sufficient to prove optimality, if a clique of size ω has already been found. For the remainder of the "san" family (but not "sanr"), at least $87\,\%$ of the search space is avoidable. The "brock", "gen" and "p_hat" graph families also contain many members with substantial avoidable proportions. The BHOSLIB instances are similar: the avoidable proportion is between $25\,\%$ and $58\,\%$. Finally, the protein product graphs all have very high avoidable proportions, although in some cases these are coupled with a low total node count. Thus we should expect that parallel search order will often, but not always, be an important factor in determining speedups.

3.2.4 The Quality of Heuristics, and What This Implies

W. D. Harvey and Ginsberg's (1995) limited discrepancy search is a general tree-search technique which is based upon two principles. Firstly, that when a search fails to find a solution immediately, it is likely that it only made a small number of "wrong turns" (a discrepancy is when search does not follow a heuristic, in an attempt to correct one of these wrong turns). Secondly, they claim that "for many problems the heuristics are *least* reliable

Table 3.1: Properties of the sequential search space for selected problem instances. Shown is the size of a maximum clique ω, then the total number of search nodes. Next is the number of search nodes required to prove there is no clique of size $\omega + 1$, and the percentage of the search space which is avoidable. Finally, we give the location of the first maximum clique.

Instance	ω	Total	Prove	Avoid	Location
Randomly generated					
C125.9	34	5.02×10^{4}	2.69×10^{4}	46.4	$7, 5, 5, 3, 1, 1, 2, 2, 1\times26$
C250.9	44	1.08×10^{9}	9.68×10^{8}	10.5	$4, 32, 4, 9, 3, 5, 2\times3, 5, \ldots$
C2000.5	16	1.82×10^{10}	1.82×10^{10}	0.0	$1, 21, 19, 26, 5, 1\times11$
DSJC500_5	13	1.15×10^{6}	1.09×10^{6}	5.2	$8, 38, 3, 1\times10$
DSJC1000_5	15	7.70×10^{7}	7.67×10^{7}	0.3	$1, 96, 1\times13$
Randomly generated with large degree spread					
p_hat300-1	8	1.48×10^{3}	1.29×10^{3}	12.9	$18, 12, 1\times6$
p_hat300-2	25	4.26×10^{3}	2.83×10^{3}	33.5	$7, 18, 4, 4, 1\times21$
p_hat300-3	36	6.25×10^{5}	2.46×10^{5}	60.6	$69, 21, 2, 3, 2\times3, 1\times5, 2, 1\times23$
p_hat500-1	9	9.78×10^{3}	9.70×10^{3}	0.8	$3, 18, 1\times7$
p_hat500-2	36	1.14×10^{5}	3.96×10^{4}	65.3	$114, 10, 10, 5, 2, 4, 1\times10, 2, 1\times19$
p_hat500-3	50	3.93×10^{7}	1.56×10^{7}	60.3	$96, 64, 12, 8, 2\times3, 1\times4, 2, 1\times38$
p_hat700-1	11	2.66×10^{4}	1.62×10^{4}	39.3	$259, 4, 2, 1\times8$
p_hat700-2	44	7.51×10^{5}	3.79×10^{5}	49.6	$62, 35, 18, 6, 2, 3, 1\times38$
p_hat700-3	62	2.82×10^{8}	1.60×10^{8}	43.3	$147, 62, 42, 11, 6, 8, 2, 2, \ldots$
p_hat1000-1	10	1.77×10^{5}	1.75×10^{5}	0.8	$6, 29, 5, 1\times7$
p_hat1000-2	46	3.45×10^{7}	2.18×10^{7}	36.8	$171, 36, 25, 5, 2, 2, 1, 2, \ldots$
p_hat1000-3	68	1.30×10^{11}	3.85×10^{10}	70.5	$368, 30, 99, 6, 10, 3, 2, 1, \ldots$
p_hat1500-1	12	1.18×10^{6}	9.59×10^{5}	19.1	$226, 75, 4, 1\times9$
p_hat1500-2	65	2.01×10^{9}	1.09×10^{9}	45.5	$280, 136, 25, 29, 8, 1, 3, 3, \ldots$
Randomly generated with large hidden solutions					
brock200_1	21	5.25×10^{5}	3.06×10^{5}	41.7	$22, 4, 10, 6, 1\times17$
brock200_2	12	3.83×10^{3}	2.58×10^{3}	32.6	$10, 7, 1\times10$
brock200_3	15	1.46×10^{4}	1.45×10^{4}	0.3	$1, 1, 3, 1\times12$
brock200_4	17	5.87×10^{4}	3.16×10^{4}	46.2	$36, 20, 5, 1\times14$
brock400_1	27	1.98×10^{8}	1.17×10^{8}	41.0	$20, 2, 10, 11, 4, 2, 1\times21$
brock400_2	29	1.46×10^{8}	4.84×10^{7}	66.8	$13, 8, 10, 2, 1, 2, 1\times23$
brock400_3	31	1.20×10^{8}	1.66×10^{7}	86.2	$14, 10, 1, 4, 2, 1\times26$
brock400_4	33	5.44×10^{7}	7.67×10^{6}	85.9	$9, 3, 4, 3, 1\times29$
brock800_1	23	2.23×10^{9}	1.76×10^{9}	21.0	$16, 9, 9, 16, 3, 1\times18$
brock800_2	24	2.24×10^{9}	1.31×10^{9}	41.4	$23, 65, 1\times22$
brock800_3	25	2.15×10^{9}	7.03×10^{8}	67.3	$46, 2, 24, 6, 2, 1\times20$
brock800_4	26	6.40×10^{8}	5.09×10^{8}	20.5	$4, 32, 7, 2, 1\times22$
gen200_p0.9_44	44	1.77×10^{6}	1.49×10^{5}	91.6	$8, 4, 2, 2, 1\times40$
gen200_p0.9_55	55	1.70×10^{5}	2.32×10^{3}	98.6	$4, 3, 1\times53$
gen400_p0.9_65	65	1.76×10^{11}	7.29×10^{9}	95.9	$11, 7, 3, 2, 1\times61$
gen400_p0.9_75	75	1.05×10^{11}	1.24×10^{8}	99.9	$17, 8, 3, 4, 1\times71$
san200_0.7_1	30	1.34×10^{4}	227	98.3	$2\times3, 1\times27$
san200_0.7_2	18	464	1	99.8	$7, 3, 3, 1\times15$
san200_0.9_1	70	8.73×10^{4}	18	100.0	$2, 1\times69$
san200_0.9_2	60	2.30×10^{5}	1.06×10^{3}	99.5	$5, 1, 3, 1\times57$
san200_0.9_3	44	6.82×10^{6}	4.19×10^{5}	93.8	$3, 4, 2, 1\times41$
san400_0.5_1	13	2.45×10^{3}	1	100.0	$20, 5, 3, 1\times10$
san400_0.7_1	40	1.19×10^{5}	9.94×10^{3}	91.7	$13, 2, 3, 1\times37$

continued on next page...

Instance	ω	Total	Prove	Avoid	Location
san400_0.7_2	30	8.89×10^5	8.32×10^4	90.6	$27, 1, 2, 1, 2, 1\times25$
san400_0.7_3	22	5.21×10^5	6.70×10^4	87.1	$3, 4, 2, 1, 1, 2, 2, 1\times15$
san400_0.9_1	100	4.54×10^6	3.33×10^5	92.7	$8, 1\times99$
san1000	15	1.51×10^5	1	100.0	$42, 39, 1\times13$
sanr200_0.7	18	1.53×10^5	1.26×10^5	17.6	$5, 4, 8, 6, 2, 1\times13$
sanr200_0.9	42	1.49×10^7	1.02×10^7	31.7	$8, 15, 2, 4, 2\times6, 1\times32$
sanr400_0.5	13	3.20×10^5	1.96×10^5	38.8	$95, 12, 2, 2, 1\times9$
sanr400_0.7	21	6.44×10^7	6.40×10^7	0.7	$1, 22, 4, 11, 6, 1, 3, 2, 1\times13$
Randomly generated with known solution sizes					
frb30-15-1	30	2.92×10^8	2.18×10^8	25.5	$12, 5, 13, 7, 4, 2, 1\times3, 3, \ldots$
frb30-15-2	30	5.57×10^8	3.32×10^8	40.4	$20, 13, 7, 4, 11, 1, 3, 2, \ldots$
frb30-15-3	30	1.67×10^8	1.00×10^8	40.0	$13, 3, 8, 4, 4, 1, 1, 4, \ldots$
frb30-15-4	30	9.91×10^8	4.20×10^8	57.7	$55, 10, 12, 4, 2, 1, 2, 1, \ldots$
frb30-15-5	30	2.83×10^8	1.77×10^8	37.3	$9, 17, 7, 4, 3, 2, 2, 1\times3, \ldots$
Fault diagnosis					
c-fat200-1	12	24	3	87.5	$5, 1\times11$
c-fat200-2	24	24	1	95.8	1×24
c-fat200-5	58	139	27	80.6	$28, 1\times57$
c-fat500-1	14	14	1	92.9	1×14
c-fat500-2	26	26	1	96.2	1×26
c-fat500-5	64	64	1	98.4	1×64
c-fat500-10	126	126	1	99.2	1×126
Coding theory					
hamming6-2	32	32	1	96.9	1×32
hamming6-4	4	82	81	1.2	1×4
hamming8-2	128	128	1	99.2	1×128
hamming8-4	16	3.65×10^4	3.64×10^4	0.0	1×16
hamming10-2	512	512	1	99.8	1×512
johnson8-2-4	4	24	23	4.2	1×4
johnson8-4-4	14	126	115	8.7	1×14
johnson16-2-4	8	2.56×10^5	2.56×10^5	0.0	1×8
Keller conjecture					
keller4	11	1.37×10^4	1.37×10^4	0.3	$1, 4, 1, 5, 1\times7$
keller5	27	5.07×10^{10}	5.07×10^{10}	0.0	$1, 2, 1, 13, 1, 1, 3, 1\times20$
Steiner triple problem					
MANN_a9	16	71	60	15.5	1×16
MANN_a27	126	3.80×10^4	3.78×10^4	0.6	$1\times6, 3, 1\times119$
MANN_a45	345	2.85×10^6	2.85×10^6	0.2	$1\times5, 6, 1\times4, 6, 1\times334$
Proteins					
1KZKA_3KT2A_78	247	247	1	99.6	1×247
1allA_3dbjC_41	346	675	372	44.9	1×346
1f82A_1zb7A_5	500	716	294	58.9	1×500
2FDVC_1PO5A_83	556	1.35×10^3	146	89.2	$2, 1\times555$
2UV8I_2J6IA_13107	69	4.26×10^3	461	89.2	$5, 3, 1, 1, 3, 2, 1\times63$
2W00B_3H1TA_10858	143	7.77×10^5	1.23×10^5	84.2	$6, 6, 1\times141$
2W4JA_2A2AD_0	447	890	10	98.9	$1\times5, 2, 1\times441$
3HRZA_2HR0A_476	563	9.35×10^5	3.23×10^5	65.4	$1, 2, 2, 1, 2, 1\times4, 2, 1\times14, \ldots$
3P0KA_3GWLB_0	89	90	3	96.7	1×89
3ZY0D_3ZY1A_110	52	52	1	98.1	1×52

early in the search, before making decisions that reduce the problem to a size for which the heuristics become reliable". There is a long-standing tradition of focusing upon the first of these two claims, and ignoring the second (Korf, 1996; Moisan, Gaudreault, and Quimper, 2013; Prosser and Unsworth, 2011; Walsh, 1997). Here we will buck the trend and emphasise the second claim.

One observation that suggests that the second claim might hold for the maximum clique problem is that the heuristics use degree information at the top of search. For many of the DIMACS graphs the degrees of most vertices are fairly similar (which motivates many of the tie-breaking strategies which we discussed in Chapter 2), and in some cases they are intentionally designed to be misleading. We consider the possibility that using parallelism to avoid a strong commitment to the first choice made by a weak early heuristic may be a more fruitful alternative than trying to wring even more information out of the graph at the top of search. Explicitly splitting at distance 1 maximises this effect—by assigning additional workers to go against initial heuristic advice, we minimise early commitments.

We return to Table 3.1 to justify this. The final column shows the location in the search space of the maximum clique found by the sequential algorithm. If heuristics were strong at the top of search, we would expect the first number to be 1 in most cases, and low in the remainder. Sometimes this is indeed the case—for example, for the graphs "hamming8-4", taking the first heuristic choice sixteen times in a row leads immediately to an optimal solution. Similarly, the heuristic is perfect for "johnson16-2-4" (taking the first heuristic choice at each level again finds a solution immediately), and for "MANN_a27" and "MANN_a45" it is accurate for the first few levels. For these graphs, the cost is almost entirely in proving optimality. This suggests that having at least one thread preserve search order to avoid a slowdown is worthwhile (although in fact for all of these graphs, there are multiple maximum cliques).

On the other hand, for most graphs the first number in the last column of Table 3.1 is not a 1, or even a small number. The table shows that W. D. Harvey and Ginsberg's second claim holds for the DIMACS, BHOSLIB and non-trivial protein instances for the maximum clique problem: heuristic information is weak at the top of search, and strong commitment to that information can result in reduced performance. Combined with the high proportion of avoidable nodes in many of these cases, we should expect parallel search order to matter, and for explicit diversity early in search to be beneficial compared to strong early commitment. (Additionally, early diversity remains as good as any other solution in cases where heuristics are strong at all levels, or where there are few avoidable nodes.)

3.2.5 Selected Results in Depth

We will now look in more detail at the behaviour of different work splitting mechanisms for selected instances, to justify our claims about parallel algorithm design.

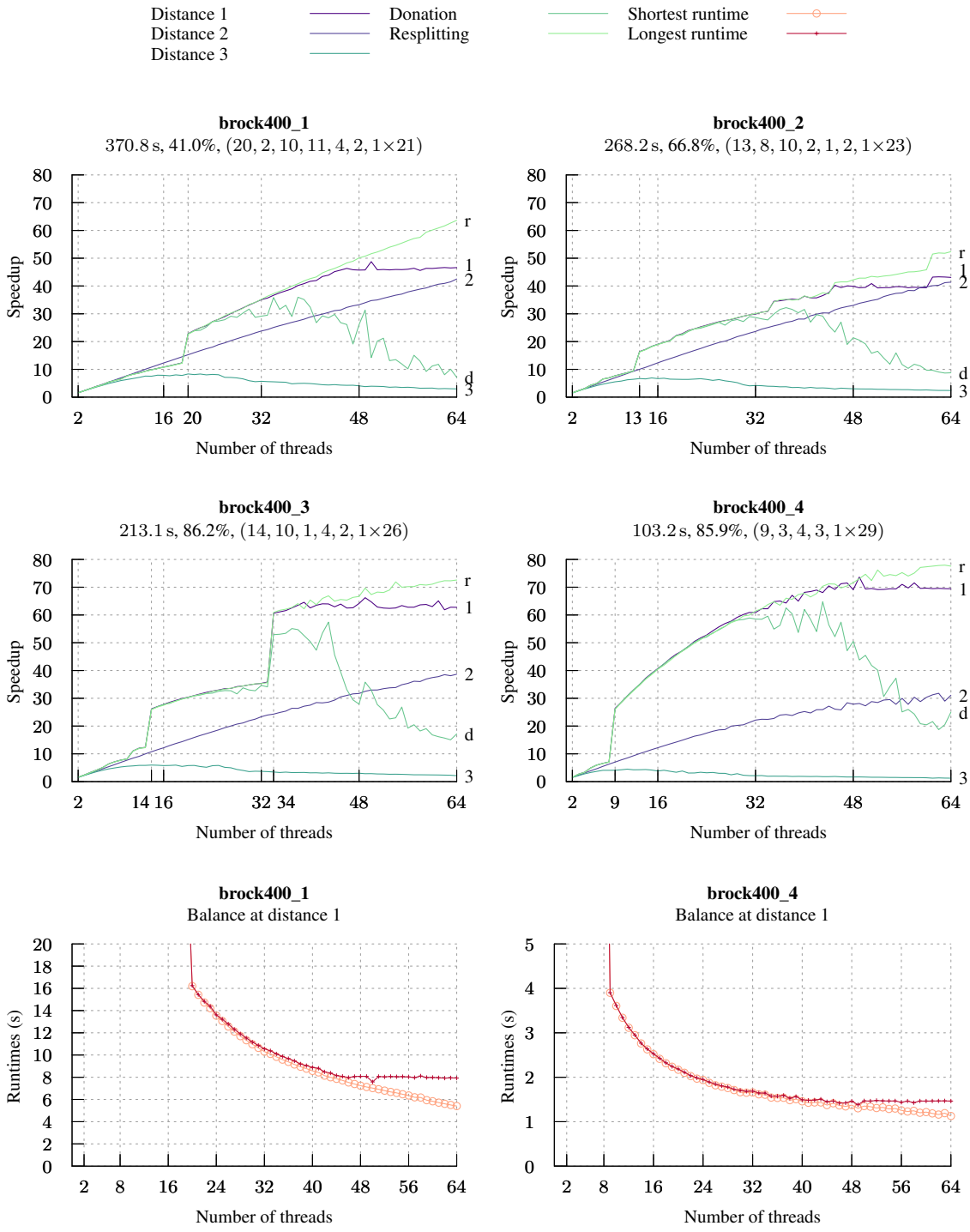


Figure 3.8: The first four graphs show speedup obtained as the number of threads increases for the "brock400" family of DIMACS instances. Measurements are from a 32 module / 64 core system. The title shows the instance, the sequential runtime, the proportion of avoidable nodes, and the location of the solution found by the sequential algorithm. The dotted vertical lines show where we might expect speedup jumps to occur when splitting at distance 1; the second dotted vertical line in the third graph at position 34 is explained in the text. The bottom two graphs show the balance when splitting at distance 1 with varying number of threads for "brock400_1" (where balance becomes a problem) and "brock400_4" (where balance is better).

In Figure 3.8 we show the speedups obtained as the number of threads increases from 2 to 64 for the four members of the DIMACS "brock400" family. We see straight away that we are getting very different behaviour to that shown for "MANN_a45" in Figure 3.5. For "MANN_a45", splitting at distance 3 was clearly the best option, but here in each case splitting at distance 1 is best, and distance 3 is very poor. This shows that balance is not the deciding factor for these instances.

For each of these graphs we see a sudden jump in speedups from being approximately linear to being visibly superlinear when splitting at distance 1. For "brock400_1", the jump occurs when going from 19 threads to 20, and the solution is located at $(20, 2, 10, 11, 4, 2, 1{\times}21)$. Thus with distance 1 splitting, the 20th thread very quickly finds an optimal solution and allows much of the search tree to be eliminated. The same behaviour occurs for the other three members of the "brock400" family—in each case, the jump occurs with k threads, where k is the first number in the location of an optimal solution. There is no jump when splitting at distance 2 or distance 3.

For "brock400_3" ($\omega = 31$) we get a second jump as we move to 34 threads. We might think perhaps there is a second optimal solution located at, say, $(34, 1{\times}30)$. In fact it is not this clear-cut: the solutions for all four of the "brock400" graphs are unique, but for "brock400_3" there is a strong (but not optimal) incumbent of size 29 located at $(34, 5, 8, 1, 3, 3, 1{\times}23)$. With 33 threads, the best incumbent found by the equivalent time in the search only has size 24.

Balance does have some effect for "brock400_1", with over 40 threads—we show this in the fifth graph. The sixth graph shows "brock400_4", where the balance is even better (the other two graphs are in-between). But in each case, splitting at distance 1 gives a reasonable work distribution and best overall performance. We should not find this surprising: Depolli et al. did not encounter hard speedup limits with most instances, so we would not expect balance problems to be common, particularly with smaller numbers of threads.

One further observation is that in each case, for up to around eight threads it does not matter much which mechanism is used—each gives a roughly linear speedup.

But are these result typical? Figure 3.9 shows the same information for four further DIMACS graphs. Each shows different interesting behaviour.

For "san400_0.9_1", where the solution is located at $(8, 1{\times}99)$, there is a very sharp jump as we go from seven to eight threads when splitting at distance 1. Here the eighth thread finds the solution immediately, and this allows over $90\,\%$ of the search space to be avoided. We see speedups from nearly 100 to over 300 as we go from 8 to 64 threads. Note that our parallel runtimes go considerably below one second—within this region, our implementation is sensitive to scheduling effects, startup costs and the time spent in the initial sequential part of the problem, and so our speedup lines become rather unstable.

An even stronger superlinear speedup is obtained for "gen400_p0.9_75", where the

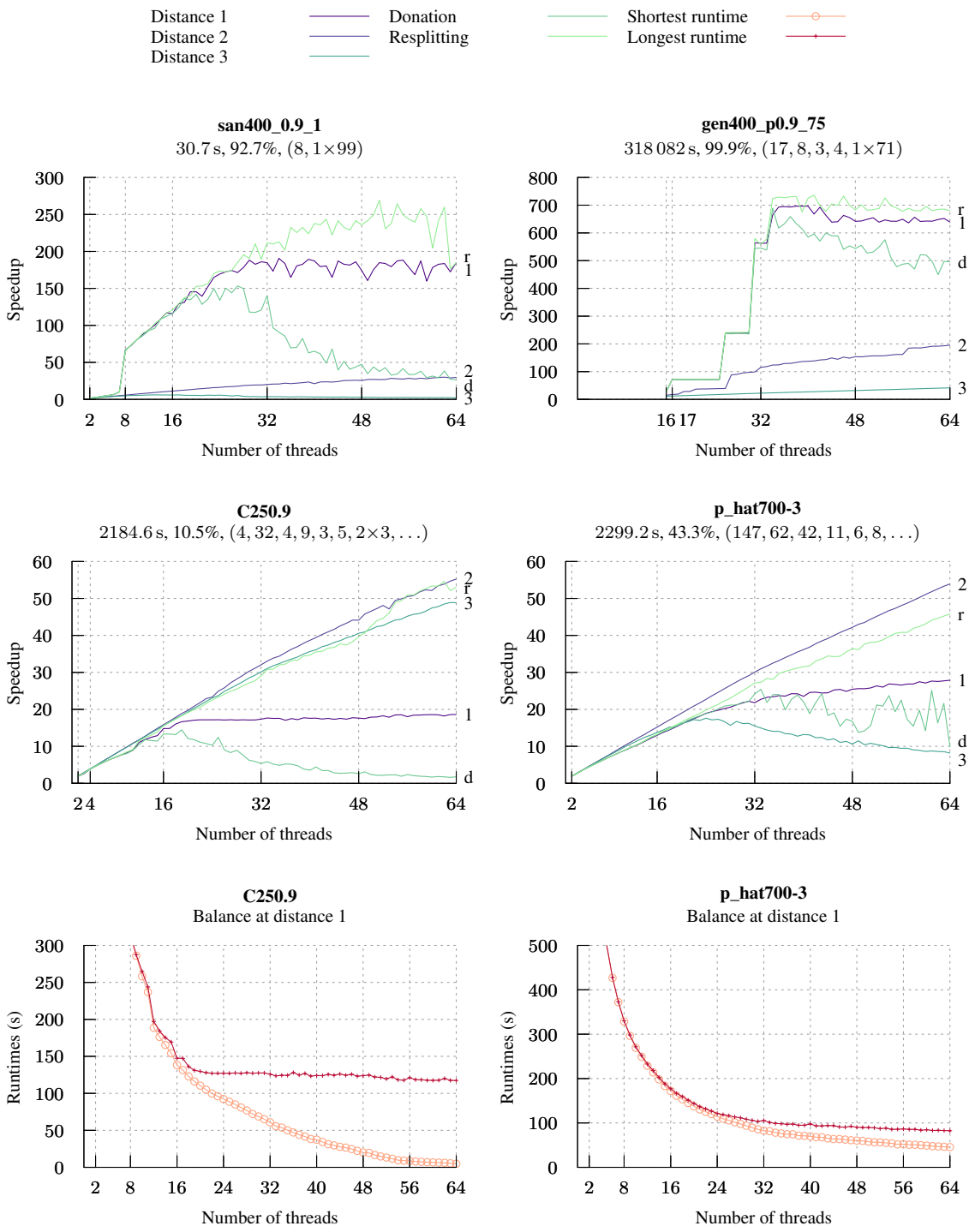


Figure 3.9: The first four graphs show speedup obtained as the number of threads increases for four further DIMACS instances. Measurements are from a 32 module / 64 core system. The title shows the instance, the sequential runtime, the proportion of avoidable nodes, and the location of the solution found by the sequential algorithm. The dotted vertical lines show where we might expect speedup jumps to occur when splitting at distance 1. For "gen400_p0.9_75", results using fewer than 16 threads are omitted due to the long sequential runtime. The bottom two graphs show the balance when splitting at distance 1 with varying number of threads for "C250.9" and "p_hat700-3".

speedup from distance 1 splitting is between 800 and 1000 from 32 or more threads. We should expect that this could happen: $99.9\,\%$ of the search space is avoidable. But although there is a small jump when going to 17 threads, as we might predict, there are two much larger jumps at 26 and 31 threads. The jump at 17 threads is small because the heuristic is badly wrong at both the top of search, and the second level, so the 17th thread still takes a long time to find a maximum clique. However, the 26th thread quickly finds an incumbent of size 55 at $(26, 1{\times}3, 2{\times}3, 1, 2, 1{\times}46)$, and a little later the 31st thread finds an incumbent of size 59 at $(31, 1, 3, 4, 1{\times}3, 2, 1{\times}51)$. These are followed by larger incumbents being found by the 17th and 26th threads, and then the 31st thread finds an incumbent of size 72 at $(31, 4, 3, 1{\times}4, 2, 1{\times}64)$. This eliminates enough of the search tree that the maximum clique of size 75 is then found by the 17th thread as we would expect, and search finishes two seconds later. This is an unusually complicated picture, whose behaviour is not captured by simple measurements which assume a single maximum clique and nothing else—we must look more carefully inside the search process to explain what is going on.

This is not the only peculiar behaviour visible in the graph. We can also explain why the speedup decreases slightly as the number of threads goes from 32 to 64. Firstly, we note that there is almost no improvement to the number of nodes required to find an optimal solution when going over 31 threads, so each additional thread could contribute at most a linear extra improvement to the runtime. Secondly, we remind the reader that although our hardware is marketed as having 64 cores, some resources are shared between cores—thus, it does not have 64 times as much processing power when used perfectly in parallel. Nor does the bandwidth available for memory and cache increase by a factor of 64 when using all 64 cores. The result is that although we have more *total* processing power when using all 64 cores, each individual core is penalised somewhat: with 32 threads, stronger incumbents are found slightly sooner than with 64 threads. Since time to find a maximum clique is the most important factor for this instance, the overall speedup is reduced. This is a slightly odd case of what de Bruin, Kindervater, and Trienekens (1995) describe as the "[danger of increasing] the processing power of a system by adding a less powerful processing element"; this should also serve as a warning against attempting to gain the benefits of increased diversity by using more threads than there are cores.

For "C250.9" there is no jump. The reasons for this are twofold. Firstly, although the solution is found by taking the fourth heuristic choice at the top of search, the second heuristic decision (where we are not explicitly diversifying) is also poor. Thus, even with distance 1 splitting, we do not find a better incumbent quickly. Secondly, the proportion of avoidable nodes for this instance is relatively low, so finding a stronger incumbent quickly does not provide much benefit. This graph also has a speedup limit from imbalances when splitting at distance 1, which we show in the fifth graph.

Nor is there a jump for "p_hat700-3" (ω = 62). Here splitting at distance 2 beats splitting

at distance 1 by a small amount at 16 threads, and a much larger amount by 64 threads. Balance is one factor here, but only from 24 threads onwards. Splitting at distance 1 finds incumbents of sizes up to 53 most quickly, but splitting at distance 2 finds incumbents of sizes 54 and higher in slightly less time. Although $43.3\,\%$ of the search space is avoidable, the heuristics in this case are sufficiently poor at the first six levels that we are unable to find an optimal solution quickly regardless of how the work is split.

Overall, we see that balance is sometimes a problem with larger numbers of cores, but that increasing the likelihood of finding an optimal solution quickly is usually far more important. As we predicted, the early diversity offered by splitting at distance 1 is often helpful with this.

3.3 Getting the Best of Both Worlds

In view of the previous section, we should look for a work splitting mechanism which gives a good balance, particularly if we are targeting larger numbers of cores. But the results clearly reinforce that splitting at distance 2 or 3 rather than distance 1 has practical consequences beyond the balance of subproblems, and that in most cases parallel search order rather than balance is the dominating factor. We need an approach which gives the best of both worlds.

It is generally known that parallel search order is important if a strong incumbent is not available at the start of search, but selecting a parallel search order explicitly designed to improve our chances of finding a solution quickly has not been given the attention it deserves. For example, Clausen (1997) notes that "most implementations . . . focus on workload distribution methods to increase efficiency of the parallel algorithm". Clausen discusses balance, and how to assess the efficiency of a parallel algorithm, and observes that "different selection strategies may lead to different search trees", but does not consider where the solutions are actually likely to be in the search space, and how this can be used to determine a work splitting mechanism. Similarly, in the design of an algorithmic skeleton for parallel branch and bound, Poldner and Kuchen (2008) state that

> "the number of problems considered by the parallel skeleton differs enormously over several runs with the same inputs. This number largely depends on the fact whether a subproblem leading to the optimal solution is picked up early or late. Note that the parallel algorithm behaves non-deterministically in the way the search-space tree is explored. In order to get reliable results, we have repeated each run 100 times and computed the average runtimes."

The search-order issue also occurs in constraint programming. Schulte, Tack, and Lagerkvist (2016) list seven pitfalls with the randomised work-stealing system implemented in the Gecode constraint programming toolkit. The most relevant to our discussion are as follows.

> "As work-stealing is indeterministic (depending on how threads are scheduled, machine load, and other factors), the work that is stolen varies over different runs for the very same problem: an idle worker could potentially steal different subtrees from different busy workers. As different subtrees contain different solutions, it is indeterministic which solution is found first.
>
> When using parallel search one needs to take the following facts into account:
>
> - The order in which solutions are found might be different compared to the order in which sequential search finds solutions. Likewise, the order in which solutions are found might differ from one parallel search to the next. This is just a direct consequence of the indeterministic nature of parallel search. Naturally, the amount of search needed to find a first solution might differ both from sequential search and among different parallel searches. Note that this might actually lead to superlinear speedup (for n workers, the time to find a first solution is less than $1/n$ the time of sequential search) or also to real slowdown.
> - For best solution search, the number of solutions until a best solution is found as well as the solutions found are indeterministic. First, any better solution is legal (it does not matter which one) and different runs will sometimes be lucky (or not so lucky) to find a good solution rather quickly. Second, as a better solution prunes the remaining search space the size of the search space depends crucially on how quickly good solutions are found.
> - As a corollary to the above items, the deviation in runtime and number of nodes explored for parallel search can be quite high for different runs of the same problem."

These pitfalls are not confined to Gecode's work stealing implementation. Caniou et al. (2011) note that "solutions may be not uniformly distributed in the search space", and that this has an affect on parallelism, but they do not consider adapting the search process to improve the chances of finding a solution quickly. The "embarrassingly parallel" approach introduced by Régin, Rezgui, and Malapert (2013) "relies on the assumption that the resolution time of disjoint subproblems is equivalent to the resolution time of the union of these subproblems"—we have seen that this assumption does not hold here. Indeed, Malapert, Régin, and Rezgui (2016) state that

> "We will ignore the problem of finding a first feasible solution because the parallel speedup can be completely uncorrelated to the number of workers, making the results hard to analyze. We will consider optimization problems for which the same variability can be observed, but at a lesser extent because the optimality

> proof is required. The variability for unsatisfiable and enumeration instances is lowered, and therefore, they are often used as a test bed for parallel computing."

We argue that being hard to analyse is not a reason to avoid doing the analysis, particularly given the rich history of search strategies and ordering heuristics in constraint programming. Here we *do* analyse the results, explain the source of the variability, and use it to improve the decomposition method. This is not a common attitude—more frequently the issue is ignored entirely, or artificially eliminated from the results. For example, when proposing a new work splitting mechanism, Fischetti, Monaci, and Salvagnin (2014) state that

> "Since we are interested in measuring the scalability of our method, we considered only instances which are either infeasible or in which we are required to find all feasible solutions (the parallel speedup for finding a first feasible solution can be completely uncorrelated to the number of workers, making the results hard to analyze)."

Leroy et al. (2014) do the same in a branch and bound setting:

> "Therefore, we chose to always initialize our B&B by the optimal solution of the instance to be solved. With this initialization, we are sure that the number of explored subproblems is the same in both approaches."

We believe that both of these approaches to measurement and comparison are flawed—parallelism and search order are linked, and should be considered together. In particular, we disagree with Rayward-Smith, Rush, and McKeown's (1993) assertion that speedup is "only a very crude measure of the success of parallelism" and their notion of "pseudo efficiency". Changes in the amount of work done are inherent to parallelism, and should *not* be disregarded when evaluating a parallelism strategy.

Changes to the amount of work done and the expected locations of solutions *are* considered by Chu, Schulte, and Stuckey's (2009) confidence-based work stealing. However, their experimental work does not evaluate the strengths of confidence heuristics based upon domain knowledge, and their only experiment using non-fixed confidence intervals relies upon already knowing where solutions are found. In Chu, Schulte, and Stuckey's terms, we will show that we can specify a good, non-constant confidence heuristic for our underlying sequential algorithm, and that we may exploit it using a simple queueing mechanism. We believe that the simplicity aspect is important: to the best of our knowledge, despite promising experimental results, confidence-based work stealing has no public implementation and has never been available in a production-ready constraint programming toolkit. A further difference is that our approach is both reproducible and scalable (we introduced these properties in Section 1.6.6), which we believe are very useful properties when scientifically verifiable results are required—we will see in the next section the effects of not guaranteeing these properties.

We also desire a stronger notion of reliability than Poldner and Kuchen (2008). For non-trivial problem instances, when splitting at a fixed depth, runtimes are very consistent between executions; we consider it desirable to preserve this property. Langer et al. (2013) look at parallel branch and bound for integer programming, and explain that "reducing idle time by eagerly exploring as much of the tree as possible might be counter-productive by using compute resources for exploring sub-trees that might have been easily pruned later". They evaluate two designs for work allocation, and observe that "even though Design A has better repeatability, the worst performance using Design B is better than the best performance using Design A", and conclude that "Design B is the design of choice". We do not wish to have to make this kind of trade-off.

3.3.1 A Low-Overhead, High-Diversity Parallelism Mechanism

We must ensure early diversity (that is, weak heuristic commitment at the top of search), but retain a way of rebalancing subproblems. McCreesh and Prosser (2013) described (without detailed justification) a work donation mechanism, whereby subproblems are requeued when threads become idle. This mechanism worked very well on the 12 core Intel machine used in previous experiments, but it is clear that the approach is unsuitable for a 64 core AMD machine. We now introduce an alternative work splitting mechanism which is more suitable for larger numbers of cores.

Initially we split work at distance 1, as before, and place items in order onto a queue. When the queue is empty, and a thread first becomes idle, this thread then steals and requeues unstarted work from every other thread, splitting at a distance of 2. Finally, when the queue again becomes empty and a thread becomes idle, work is stolen and requeued with splitting at a distance of 3. This gives us all the search order benefits of distance 1 splitting, and the balance benefits of splitting at distance 3. In addition, if w is the number of workers and $|V|$ the number of vertices in the graph, it limits the queue size to be at worst $w \times |V|$ rather than $|V|^3$, and avoids enqueueing lots of eliminable nodes early on.

This scheme may be implemented with very little overhead, as follows. Each thread publishes three integers, describing its current position in search at the first three levels, and three corresponding flags. When work stealing takes place, the appropriate flag is set; when returning from a recursive call at the first three levels, the flag is checked to see whether the remainder of the work has already been stolen and requeued. In particular, this approach introduces no overhead below depth 3 (where most of the search time is spent), and is very unlikely to have any contended critical sections.

To share the incumbent, we follow the approach of McCreesh and Prosser (2013) and use an atomic integer variable to store its size. Using a mutex instead introduces severe scalability limits: the incumbent's size is accessed during every recursive call, but is updated very rarely. The actual vertices making up the incumbent do not need to be shared, and can be stored on a

thread-by-thread basis, with a simple reduction operation at the end of search to select the largest.

Results for this approach are shown in our speedup plots—we see that in each case it is at least nearly as good as whichever other mechanism is best, and in many cases it is slightly better than the best alternative. (Since speedups are being given over a good sequential implementation, not over a parallel implementation run with one thread, it is legitimate to compare lines directly.) For all of the "brock400" graphs, "san400_0.9_1" and "gen400_p0.9_75", we get the same superlinear speedup jumps as when splitting at distance 1. But "brock400_1", "MANN_a45" and "C250.9" show that this approach is successful in addressing imbalances. With "p_hat700-3", we initially behave similarly to splitting at distance 1, but as the number of threads increases and resplitting starts to have an effect earlier in search, our behaviour approaches that of distance 2 splitting. The graphs also clearly show that this technique scales well to 64 cores, and does not introduce noticeable overheads compared to static splitting mechanisms.

These experiments also suggest that with 64 cores, splitting at distance 3 is sufficient, and there is no benefit of continuing this to an arbitrary distance. However, this is not an intrinsic limitation, and a larger maximum depth can be used if necessary for larger numbers of cores or if unusually-shaped problem instances are found (indeed, this turns out to be necessary in the following chapter). Régin, Rezgui, and Malapert (2013) suggest that the number of subproblems created only needs to grow linearly with the number of cores. By increasing the effective splitting depth dynamically by resplitting we instead gain a polynomial increase for only linear cost.

3.3.2 Comparison to Off-the-Shelf Work Stealing

We now compare our new approach with an off-the-shelf randomised work stealing system, using the GCC implementation of Intel's Cilk Plus extensions to C++. We are now interested in performance rather than insight, so we return to the dual Xeon E5-2697A v4 machines used in the previous chapter. In Table 3.2 we present sequential and parallel runtimes for the larger solvable DIMACS instances, and the smallest group of BHOSLIB instances (these are selected so that parallel runtimes are at least half a second—below this point, measurements have considerable noise). Each parallel measurement is the average of ten runs: we show both runtimes, and the number of recursive calls made. For parallel measurements, we also give the standard deviation and the range of values observed, both expressed as a proportion.

In all but five cases, resplitting takes less time on average, sometimes by more than a factor of ten. In some of these cases resplitting does more work: sometimes this is a genuine difference, and sometimes it is simply because resplitting recomputes parts of the search space, and we include this in node count measurements. The overheads between the two approaches differ: Cilk Plus does not recompute work, but requires extra copying. The better

Table 3.2: A comparison of sequential and parallel runtimes for larger problem instances, using either resplitting or Cilk Plus for work stealing, with 32 or 64 threads on a 32 core hyper-threaded dual Intel Xeon E5-2697A v4 system. Parallel figures are an average of ten samples. The error is the standard deviation, and the figures in parentheses are the range, both expressed as a proportion. A $\star$ indicates the better average.

T	Resplitting			Cilk Plus		
	Runtime $\pm$ RSD (Range)	Speedup	Work	Runtime $\pm$ RSD (Range)	Speedup	Work
2W00B_3H1TA_10858: sequential 1.9 s						
32	$\star$136.8 ms $\pm$ 0.29 (0.42-1.45)	14.17	$\star$ 0.26	416.2 ms $\pm$ 0.20 (0.69-1.39)	4.66	0.69
64	$\star$160.5 ms $\pm$ 0.27 (0.70-1.52)	12.08	$\star$ 0.23	350.1 ms $\pm$ 0.29 (0.56-1.45)	5.54	1.04
3HRZA_2HR0A_476: sequential 17.9 s						
32	2.3 s $\pm$ 0.04 (0.92-1.05)	7.83	0.51	$\star$1.0 s $\pm$ 0.13 (0.77-1.23)	17.89	$\star$ 0.50
64	2.4 s $\pm$ 0.05 (0.93-1.13)	7.56	$\star$ 0.47	$\star$1.8 s $\pm$ 0.21 (0.61-1.45)	10.16	0.77
C250.9: sequential 1126.1 s						
32	$\star$39.3 s $\pm$ 0.02 (0.99-1.05)	28.64	$\star$ 1.09	63.5 s $\pm$ 0.03 (0.95-1.05)	17.72	1.14
64	$\star$24.1 s $\pm$ 0.03 (0.96-1.05)	46.70	$\star$ 1.06	37.1 s $\pm$ 0.05 (0.93-1.09)	30.39	1.12
C2000.5: sequential 10.8 h						
32	$\star$1171.7 s $\pm$ 0.00 (1.00-1.01)	33.14	1.00	2149.6 s $\pm$ 0.01 (0.98-1.02)	18.06	$\star$ 1.00
64	$\star$806.2 s $\pm$ 0.00 (1.00-1.00)	48.17	$\star$ 1.00	1433.6 s $\pm$ 0.02 (0.98-1.03)	27.09	1.00
DSJC500_5: sequential 733 ms						
32	$\star$60.3 ms $\pm$ 0.16 (0.71-1.19)	12.16	$\star$ 0.98	169.2 ms $\pm$ 0.33 (0.44-1.45)	4.33	0.99
64	$\star$143.0 ms $\pm$ 0.79 (0.30-2.94)	5.13	1.03	307.4 ms $\pm$ 0.26 (0.59-1.55)	2.38	$\star$ 1.00
DSJC1000_5: sequential 87.9 s						
32	$\star$2.9 s $\pm$ 0.02 (0.98-1.04)	30.02	1.02	4.5 s $\pm$ 0.07 (0.87-1.09)	19.52	$\star$ 1.02
64	$\star$2.2 s $\pm$ 0.02 (0.97-1.04)	39.80	$\star$ 1.01	3.0 s $\pm$ 0.03 (0.94-1.04)	28.91	1.02
MANN_a45: sequential 123.2 s						
32	$\star$4.5 s $\pm$ 0.02 (0.97-1.03)	27.20	1.04	5.2 s $\pm$ 0.03 (0.94-1.05)	23.93	$\star$ 1.03
64	$\star$3.2 s $\pm$ 0.05 (0.91-1.07)	38.81	1.07	3.8 s $\pm$ 0.05 (0.92-1.10)	32.37	$\star$ 1.05
brock400_1: sequential 184.4 s						
32	$\star$4.2 s $\pm$ 0.01 (0.98-1.01)	44.09	$\star$ 0.64	8.0 s $\pm$ 0.08 (0.89-1.18)	23.16	0.85
64	$\star$3.0 s $\pm$ 0.02 (0.98-1.06)	62.17	$\star$ 0.66	5.3 s $\pm$ 0.08 (0.86-1.10)	35.07	0.83
brock400_2: sequential 133.7 s						
32	$\star$3.6 s $\pm$ 0.03 (0.96-1.07)	36.80	0.79	4.6 s $\pm$ 0.06 (0.91-1.10)	29.34	$\star$ 0.63
64	$\star$2.6 s $\pm$ 0.04 (0.96-1.09)	51.06	0.83	3.1 s $\pm$ 0.11 (0.83-1.19)	43.84	$\star$ 0.62
brock400_3: sequential 106.1 s						
32	$\star$2.5 s $\pm$ 0.01 (0.97-1.02)	43.24	0.64	3.4 s $\pm$ 0.11 (0.79-1.15)	31.49	$\star$ 0.54
64	$\star$1.5 s $\pm$ 0.04 (0.94-1.09)	69.31	0.56	2.2 s $\pm$ 0.08 (0.87-1.14)	48.36	$\star$ 0.53
brock400_4: sequential 51.6 s						
32	$\star$773.2 ms $\pm$ 0.07 (0.92-1.14)	66.73	$\star$ 0.32	1.0 s $\pm$ 0.16 (0.79-1.31)	51.51	0.32
64	$\star$715.6 ms $\pm$ 0.09 (0.81-1.17)	72.10	0.45	838.1 ms $\pm$ 0.17 (0.79-1.35)	61.56	$\star$ 0.30
brock800_1: sequential 3080.3 s						
32	$\star$83.1 s $\pm$ 0.00 (1.00-1.01)	37.05	$\star$ 0.83	136.4 s $\pm$ 0.21 (0.81-1.29)	22.59	0.87
64	$\star$64.7 s $\pm$ 0.00 (1.00-1.00)	47.64	0.87	82.9 s $\pm$ 0.02 (0.97-1.04)	37.17	$\star$ 0.87

continued on next page...

T	Resplitting			Cilk Plus		
	Runtime ± RSD (Range)	Speedup	Work	Runtime ± RSD (Range)	Speedup	Work
brock800_2: sequential 3083.4 s						
32	⋆87.5 s ± 0.00 (1.00-1.01)	35.22	⋆ 0.89	177.6 s ± 0.03 (0.96-1.04)	17.37	0.92
64	⋆57.1 s ± 0.00 (1.00-1.00)	54.02	⋆ 0.73	78.6 s ± 0.02 (0.97-1.03)	39.23	0.81
brock800_3: sequential 2890.3 s						
32	⋆73.2 s ± 0.00 (1.00-1.01)	39.51	⋆ 0.79	153.9 s ± 0.03 (0.94-1.04)	18.78	0.81
64	⋆29.3 s ± 0.00 (0.99-1.01)	98.63	⋆ 0.35	56.4 s ± 0.06 (0.90-1.09)	51.23	0.60
brock800_4: sequential 1075.2 s						
32	⋆44.4 s ± 0.00 (0.99-1.01)	24.23	1.46	70.9 s ± 0.03 (0.94-1.03)	15.17	⋆ 1.17
64	⋆27.2 s ± 0.00 (0.99-1.00)	39.54	⋆ 1.12	36.8 s ± 0.06 (0.90-1.09)	29.25	1.24
frb30-15-1: sequential 456.1 s						
32	⋆12.6 s ± 0.01 (0.99-1.01)	36.11	⋆ 0.81	16.7 s ± 0.03 (0.97-1.06)	27.39	0.83
64	⋆8.8 s ± 0.01 (0.99-1.02)	51.95	0.86	10.8 s ± 0.02 (0.97-1.03)	42.35	⋆ 0.84
frb30-15-2: sequential 811.2 s						
32	⋆22.9 s ± 0.01 (0.99-1.02)	35.38	⋆ 0.84	31.3 s ± 0.05 (0.92-1.06)	25.91	0.89
64	⋆17.7 s ± 0.00 (0.99-1.01)	45.93	0.98	20.5 s ± 0.07 (0.88-1.10)	39.57	⋆ 0.90
frb30-15-3: sequential 250.0 s						
32	⋆6.6 s ± 0.02 (0.97-1.04)	37.80	0.76	8.3 s ± 0.04 (0.91-1.06)	29.96	⋆ 0.73
64	⋆4.4 s ± 0.01 (0.98-1.02)	56.39	0.77	5.4 s ± 0.03 (0.93-1.06)	46.34	⋆ 0.73
frb30-15-4: sequential 1364.0 s						
32	⋆45.1 s ± 0.00 (1.00-1.01)	30.28	1.00	58.9 s ± 0.02 (0.97-1.03)	23.15	⋆ 0.99
64	⋆20.9 s ± 0.00 (1.00-1.00)	65.31	⋆ 0.68	38.3 s ± 0.03 (0.95-1.04)	35.60	0.99
frb30-15-5: sequential 402.9 s						
32	⋆15.3 s ± 0.01 (0.99-1.04)	26.35	1.15	16.4 s ± 0.04 (0.91-1.06)	24.57	⋆ 0.91
64	11.9 s ± 0.01 (0.96-1.01)	33.96	1.37	⋆11.3 s ± 0.07 (0.90-1.12)	35.52	⋆ 0.99
frb35-17-1: sequential 8.5 h						
32	⋆693.6 s ± 0.00 (1.00-1.00)	44.03	⋆ 0.70	1359.0 s ± 0.04 (0.95-1.11)	22.47	0.86
64	⋆665.6 s ± 0.00 (1.00-1.00)	45.88	0.89	916.8 s ± 0.06 (0.95-1.16)	33.31	⋆ 0.82
frb35-17-2: sequential 15.3 h						
32	⋆1711.2 s ± 0.00 (1.00-1.01)	32.27	1.02	2552.5 s ± 0.04 (0.94-1.06)	21.64	⋆ 0.91
64	⋆1718.8 s ± 0.00 (1.00-1.00)	32.13	1.35	1741.5 s ± 0.05 (0.89-1.09)	31.71	⋆ 0.90
frb35-17-3: sequential 5.6 h						
32	⋆706.3 s ± 0.00 (1.00-1.01)	28.58	1.16	957.3 s ± 0.04 (0.94-1.09)	21.09	⋆ 0.95
64	723.0 s ± 0.00 (1.00-1.00)	27.92	1.57	⋆638.9 s ± 0.06 (0.92-1.12)	31.60	⋆ 0.91
frb35-17-4: sequential 6.3 h						
32	⋆656.4 s ± 0.00 (1.00-1.00)	34.37	0.95	967.5 s ± 0.02 (0.96-1.05)	23.32	⋆ 0.87
64	⋆510.1 s ± 0.00 (1.00-1.00)	44.23	0.95	661.4 s ± 0.03 (0.96-1.06)	34.11	⋆ 0.86
frb35-17-5: sequential 34.3 h						
32	⋆3038.6 s ± 0.00 (1.00-1.01)	40.58	⋆ 0.79	1.5 h ± 0.05 (0.93-1.12)	22.58	0.84
64	⋆2863.7 s ± 0.00 (1.00-1.00)	43.06	0.98	1.1 h ± 0.05 (0.90-1.08)	31.77	⋆ 0.86
gen200_p0.9_44: sequential 1.8 s						
32	⋆115.2 ms ± 0.46 (0.36-1.81)	15.78	⋆ 0.38	369.6 ms ± 0.47 (0.26-1.60)	4.92	0.40
64	⋆202.3 ms ± 0.64 (0.32-2.14)	8.99	⋆ 0.45	355.5 ms ± 0.31 (0.37-1.50)	5.11	0.49

continued on next page...

T	Resplitting			Cilk Plus		
	Runtime ± RSD (Range)	Speedup	Work	Runtime ± RSD (Range)	Speedup	Work
gen400_p0.9_55: sequential 996.0 h						
32	⋆13.0 h ± 0.00 (1.00-1.00)	76.84	⋆ 0.40	27.1 h ± 0.15 (0.78-1.26)	36.70	0.65
64	⋆10.1 h ± 0.00 (0.99-1.01)	98.42	⋆ 0.46	17.9 h ± 0.11 (0.86-1.17)	55.80	0.64
gen400_p0.9_65: sequential 77.8 h						
32	⋆1.7 h ± 0.00 (1.00-1.00)	45.78	⋆ 0.70	2.5 h ± 0.21 (0.53-1.30)	31.71	0.78
64	⋆2612.5 s ± 0.00 (1.00-1.00)	107.23	⋆ 0.41	1.1 h ± 0.29 (0.44-1.64)	72.57	0.50
gen400_p0.9_75: sequential 43.7 h						
32	⋆194.2 s ± 0.00 (1.00-1.00)	809.44	⋆ 0.03	1.4 h ± 0.11 (0.74-1.17)	31.57	0.73
64	⋆203.6 s ± 0.00 (0.99-1.01)	772.05	⋆ 0.05	3204.1 s ± 0.25 (0.62-1.37)	49.06	0.72
keller5: sequential 25.1 h						
32	⋆2780.9 s ± 0.00 (1.00-1.00)	32.45	⋆ 1.00	1.4 h ± 0.02 (0.97-1.04)	18.13	1.00
64	⋆2186.7 s ± 0.00 (1.00-1.00)	41.27	⋆ 1.00	3567.8 s ± 0.02 (0.98-1.03)	25.29	1.00
p_hat300-3: sequential 666 ms						
32	⋆132.9 ms ± 0.21 (0.66-1.29)	5.01	1.17	362.8 ms ± 0.24 (0.55-1.30)	1.84	⋆ 1.15
64	⋆169.3 ms ± 0.52 (0.36-1.84)	3.93	⋆ 1.19	332.8 ms ± 0.32 (0.33-1.31)	2.00	1.26
p_hat500-3: sequential 70.9 s						
32	⋆2.8 s ± 0.04 (0.95-1.06)	25.22	1.10	4.0 s ± 0.06 (0.89-1.08)	17.93	⋆ 1.08
64	⋆1.7 s ± 0.07 (0.89-1.12)	41.75	⋆ 0.90	2.4 s ± 0.03 (0.96-1.07)	29.91	1.06
p_hat700-2: sequential 1.8 s						
32	⋆227.6 ms ± 0.19 (0.80-1.32)	8.04	1.23	471.6 ms ± 0.22 (0.52-1.51)	3.88	⋆ 1.11
64	⋆280.6 ms ± 0.22 (0.56-1.31)	6.52	1.54	384.8 ms ± 0.14 (0.79-1.20)	4.76	⋆ 1.19
p_hat700-3: sequential 936.0 s						
32	⋆32.9 s ± 0.01 (0.99-1.01)	28.48	⋆ 1.12	50.8 s ± 0.01 (0.98-1.03)	18.43	1.21
64	⋆26.7 s ± 0.01 (0.98-1.02)	35.09	⋆ 1.18	29.8 s ± 0.02 (0.96-1.04)	31.42	1.18
p_hat1000-2: sequential 95.0 s						
32	⋆3.5 s ± 0.01 (0.98-1.03)	27.25	1.13	5.8 s ± 0.04 (0.93-1.09)	16.25	⋆ 1.04
64	⋆2.9 s ± 0.03 (0.96-1.04)	33.18	1.19	2.9 s ± 0.04 (0.96-1.08)	32.36	⋆ 1.07
p_hat1000-3: sequential 130.8 h						
32	⋆4.1 h ± 0.00 (1.00-1.01)	31.97	⋆ 1.03	5.8 h ± 0.01 (0.99-1.01)	22.64	1.04
64	⋆3.0 h ± 0.00 (1.00-1.00)	43.12	⋆ 0.98	4.2 h ± 0.01 (0.98-1.01)	31.43	1.05
p_hat1500-1: sequential 1.8 s						
32	⋆173.5 ms ± 0.20 (0.70-1.26)	10.37	1.03	419.9 ms ± 0.18 (0.62-1.23)	4.28	⋆ 1.01
64	⋆197.8 ms ± 0.30 (0.71-1.78)	9.10	1.06	318.1 ms ± 0.23 (0.52-1.26)	5.66	⋆ 1.01
p_hat1500-2: sequential 3.7 h						
32	⋆451.4 s ± 0.00 (0.99-1.01)	29.26	1.11	532.8 s ± 0.01 (0.99-1.01)	24.79	⋆ 1.08
64	⋆315.0 s ± 0.01 (1.00-1.01)	41.94	1.26	356.2 s ± 0.01 (0.98-1.02)	37.09	⋆ 1.14
san200_0.9_3: sequential 6.2 s						
32	⋆196.0 ms ± 0.15 (0.78-1.26)	31.58	0.21	471.8 ms ± 0.33 (0.24-1.55)	13.12	⋆ 0.17
64	⋆204.6 ms ± 0.28 (0.58-1.61)	30.25	⋆ 0.25	400.9 ms ± 0.33 (0.40-1.61)	15.44	0.26
san400_0.7_2: sequential 1.3 s						
32	⋆116.6 ms ± 0.32 (0.34-1.34)	11.26	⋆ 0.48	435.1 ms ± 0.21 (0.65-1.28)	3.02	0.57
64	⋆173.5 ms ± 0.38 (0.33-1.59)	7.57	⋆ 0.56	306.9 ms ± 0.28 (0.61-1.37)	4.28	0.92

continued on next page...

T	Resplitting			Cilk Plus		
	Runtime ± RSD (Range)	Speedup	Work	Runtime ± RSD (Range)	Speedup	Work
san400_0.7_3: sequential 859 ms						
32	⋆108.4 ms ± 0.44 (0.46-1.93)	7.92	0.53	336.4 ms ± 0.29 (0.64-1.70)	2.55	⋆ 0.26
64	⋆178.1 ms ± 0.33 (0.39-1.40)	4.82	0.90	304.4 ms ± 0.29 (0.61-1.35)	2.82	⋆ 0.89
san400_0.9_1: sequential 15.3 s						
32	⋆163.1 ms ± 0.19 (0.58-1.24)	93.72	⋆ 0.08	425.0 ms ± 0.18 (0.72-1.28)	35.96	0.13
64	⋆186.8 ms ± 0.24 (0.71-1.53)	81.83	⋆ 0.08	306.3 ms ± 0.38 (0.37-1.76)	49.90	0.16
sanr200_0.9: sequential 14.3 s						
32	⋆1.2 s ± 0.06 (0.89-1.11)	11.99	1.50	1.6 s ± 0.11 (0.80-1.16)	9.13	⋆ 1.33
64	1.1 s ± 0.04 (0.95-1.05)	12.84	⋆ 1.13	⋆1.1 s ± 0.06 (0.91-1.10)	13.55	1.34
sanr400_0.7: sequential 48.3 s						
32	⋆1.8 s ± 0.02 (0.96-1.03)	27.35	1.01	3.3 s ± 0.07 (0.88-1.10)	14.81	⋆ 1.00
64	⋆1.2 s ± 0.05 (0.93-1.07)	38.79	1.01	2.0 s ± 0.08 (0.87-1.13)	24.29	⋆ 1.00
san1000: sequential 1.2 s						
32	⋆159.0 ms ± 0.23 (0.79-1.42)	7.41	0.86	257.0 ms ± 0.49 (0.35-1.83)	4.58	⋆ 0.79
64	⋆233.1 ms ± 0.47 (0.22-1.75)	5.05	1.11	299.3 ms ± 0.54 (0.34-1.96)	3.94	⋆ 0.80

approach depends upon relative costs. For large, dense graphs, copying may be the better option—this is highlighted in the protein graph "3HRZA_2HR0A_476", where the amounts of work are similar, but where Cilk Plus is approximately twice as fast due to colourings being expensive on these graphs. However, the opposite is true for, for example, "C2000.5", where resplitting achieves a higher rate of nodes per second.

It may be possible to reduce the overheads incurred with either approach, through careful tuning. However, the difference between the average runtimes is not the most striking part of the results. When looking at the standard deviations and ranges, we see a more interesting picture: for many instances, the runtimes for Cilk Plus vary substantially between repeat runs, whereas the resplitting runtimes are much more consistent. (This is why we have not included Cilk Plus runtimes on our speedup graphs: the speedups are too chaotic to give meaningful data.) This is particularly visible with the "brock" and "gen" instances, which we have seen are very sensitive to the time taken to find the solution: for "gen400_p0.9_65", our shortest observed runtime with Cilk Plus was 1681 s, and our longest was 6317 s. By contrast, with resplitting, the shortest runtime was 2602 s and the longest was 2621 s. We do not consider the variability seen with Cilk Plus to be ideal behaviour.

Despite the lack of explicit diversity, we *do* often see superlinear speedups with Cilk Plus. We can explain this. The GCC Cilk Plus implementation tends to steal "earliest created" jobs first: behind the scenes, each worker has a deque, and jobs are enqueued and processed by the worker at one end (LIFO), but are stolen from the other (FIFO); victims for stealing are selected randomly. The rationale is that this is less likely to introduce contention, and that earlier-created jobs are likely to be larger (McCool, Reinders, and Robison, 2012). However, this has another benefit here: it introduces at least some diversity early in the search. Assuming

the (1) subtree is non-trivial, the (2) subtree will always be stolen immediately, and a third thread will then either steal $(1, 2)$ or (3), nondeterministically. This is an implementation detail, and not a specified behaviour, so it could change with a new release of the compiler—this could have disastrous effects for parallel branch and bound.

This accidental diversity also usually requires a higher number of threads before the incumbent can be found quickly. With 16 threads as opposed to 64, Cilk Plus is slower for every instance except "p_hat1500-2" (where it wins by less than $1\,\%$), due to it requiring more work to find the solution.

One point in favour of Cilk Plus is the relative ease of implementation. It remains to be seen whether tailored work splitting strategies could be made as easy to program as Cilk Plus already is—recent work by Archibald et al. (2017) is an interesting first step in this direction.

3.3.3 Other Approaches

We have seen how this approach compares to that of Depolli et al. (2013): we improve the balance, whilst retaining the diversity. Compared to an off-the-shelf work stealing implementation, we often get better runtimes, and our performance is much more consistent between runs.

Compared to Xiang, Guo, and Aboulnaga (2013), we do not need to estimate up-front how large subtrees might be to obtain balance, and we do not need to do any calibration. Unfortunately Xiang, Guo, and Aboulnaga only considered three of the DIMACS graphs, all from the same family; by strange coincidence, none of these graphs have many avoidable nodes, and so the effects we discuss here are not seen in their results. This explains their consistently linear speedups (which are presented over a parallel algorithm, not a sequential one). Their approach also does not preserve any sequential order, and only shares updated incumbents when starting a new subproblem—when there are many avoidable nodes, this can cause a slowdown, both in theory and in practice. It is thus unclear what would happen with their approach on other DIMACS graph families, where scalability is not the only concern.

Compared to Moisan, Gaudreault, and Quimper's (2013) parallel discrepancy search, we are emphasising early diversity, not total number of discrepancies—that is, we believe W. D. Harvey and Ginsberg's second claim (that heuristics are weak early on) is important for this problem, not their first (that the total number of wrong turns made is low). Another form of parallel limited discrepancy search, in a constraint programming setting, is discussed by Michel, See, and Van Hentenryck (2009). They report that superlinear speedups are common for some problems when comparing parallel limited discrepancy search to sequential depth first search. However, such an approach risks introducing a slowdown (which they discuss), and again they consider the number of discrepancies rather than where those discrepancies occur.

Compared to the works by Batsyn et al. (2014), Maslov, Batsyn, and Pardalos (2014), and

Tomita, Yoshida, et al. (2016) on priming search by using a heuristic (which we discussed in the previous chapter), we are trying to obtain a strong incumbent earlier in the search by using multiple paths through a single search tree, rather than a two stage process. This means that all of our work done is potentially contributing to a proof of optimality. We also do not need to select a pre-determined arbitrary time to run the first stage, so our approach does not require special tuning before it can be used on "unseen" graphs. Note however that these approaches are not mutually exclusive—for problems which are expected to be particularly difficult, it would be possible to sacrifice one thread during early stages of our search process to perform a search using a non-exact algorithm.

Another approach using multiple search trees is cooperating local search (Clearwater, Huberman, and Hogg, 1991), which has been demonstrated in a maximum clique context by Pullan, Mascia, and Brunato (2011). Here the aim is to combine multiple non-exact algorithms in parallel in the hopes of gaining the strengths of each; no attempt is made to prove optimality. This technique has also been used in graph colouring, where Lewandowski and Condon (1993) observe that "in general, having the processors work in parallel yields better colorings faster than simply using multiple independent runs".

This theme also shows up in the ManySAT parallel SAT solver (Hamadi, Jabbour, and Sais, 2009), where parallelism is used to counter the sensitivity of SAT solvers to initial tuning parameters. Hamadi, Jabbour, and Sais remark that "the performance of parallel solvers is usually better on SAT problems than on UNSAT ones". This is because multiple search trees may make it easier to find a solution, but they are less helpful in proving that there is no solution (although things are not this simple for SAT, where learned clauses are shared). In contrast, in cases where a good solution can be found quickly, our approach allows all of the work done by every thread to contribute to a proof of optimality—this is why we typically get at least near-linear speedups. Most modern SAT solvers also make use of randomised restarts (Gomes, Selman, and Kautz, 1998), again with the aim of avoiding heavy commitment to any particular portion of the search space; the emphasis here is upon finding a solution to a satisfiable instance more quickly (although again, learning complicates this analysis).

Finally, one could use these experiments to support the view that *sequential* maximum clique algorithms should use some form of early discrepancy search. The author does not dispute this, although firstly more complicated sequential search algorithms introduce overheads which cannot be offset during the proof of optimality stage of search, and secondly, either way, every modern processor has multiple cores and we should be making use of this. In any case, doing so would not invalidate this approach—we would simply see some superlinear speedups become linear speedups over a faster sequential baseline.

3.4 Conclusion

We have considered some of the design choices available when parallelising a state-of-the-art branch and bound algorithm for the maximum clique problem. We have shown that the irregularity of subproblem sizes can cause workload balance problems—this should not be surprising. But we also saw that different parallel search orders will often produce substantially different speedups, to the extent that balance is usually *not* the deciding factor for performance.

This sheds new light on Lai and Sahni's (1984) claim that "anomalous behaviour will be rarely witnessed in practice". Both McCreesh and Prosser (2013) and Depolli et al. (2013) *did* commonly encounter superlinear speedups, but only because our parallel search orders encouraged this. We also saw superlinear speedups when using Cilk Plus, but only because of an unintended effect of an unspecified implementation detail. We used our understanding of the behaviour of heuristics to provide an explanation: parallelism was introducing diversity into the search, avoiding a strong commitment to weak early heuristic advice. We then showed how to preserve this diversity whilst improving load balancing.

More generally, these experiments demonstrate that obtaining a parallel algorithm which seems to behave well most of the time should only be the first step in the algorithm design process. For heuristics, Hooker (1995) advocates a scientific approach to algorithm evaluation, rather than simple performance comparisons:

> "Based on one's insight into an algorithm, for instance, one might expect good performance to depend on a certain characteristic. How to find out? Design a controlled experiment that checks how the presence or absence of this characteristic affects performance. Even better, build an explanatory mathematical model that captures the insight, as is done routinely in other empirical sciences, and deduce from it precise consequences that can be put to the test."

Such an approach has been helpful here too. By taking measurements inside search, and looking in depth at individual results, we gained insight into how to improve a parallel algorithm. We explained *why* McCreesh and Prosser (2013) and Depolli et al. (2013) both saw so many superlinear speedups, and how to preserve these speedups when modifying the parallel algorithm. We also showed why in certain cases Depolli et al. had parallelism limits, and why priming search is ineffective for some instances.

Later on in this thesis, this technique is re-applied to other problems. In some ways, maximum clique is an "ideal" problem for this approach, and other problems may not have the same properties which could complicate matters. It is not obvious whether W. D. Harvey and Ginsberg's (1995) second claim (that heuristics are worst early in search) will always hold—in cases where it does not, it may be necessary to switch to a more complex mechanism, along

the lines of confidence-based work stealing (Chu, Schulte, and Stuckey, 2009), to continue obtaining high quality results.

Chapter 4

Other Clique-Like Problems

In this chapter we discuss three problems which are closely related to finding a maximum clique. In each case we adapt Algorithm 2.1 from Chapter 2, showing its flexibility, and then verify that the parallelism techniques discussed in Chapter 3 remain useful in the new setting.

Firstly, we look at the maximum k-clique problem. This is a distance relaxation, originating in social network analysis. We show that solving the maximum k-clique problem by reduction to maximum clique is a practical approach. We then augment the algorithm with a new inference rule can give much better results on these instances in some cases. This part of the chapter has been published by McCreesh and Prosser (2016) as "Finding Maximum k-Cliques Faster Using Lazy Global Domination".

Secondly, we provide a new algorithm for the maximum labelled clique problem. This problem involves graphs which have labels on edges, with restrictions on how many labels may be used. Previous computational results have used integer programming models; we show that adapting our dedicated maximum clique algorithm to handle side constraints provides results which are several orders of magnitude faster. This work has been published by McCreesh and Prosser (2015b) as "A parallel branch and bound algorithm for the maximum labelled clique problem".

Finally, we look at the maximum balanced induced biclique problem, which involves looking for a different shape of subgraph. An earlier version of this work was published by McCreesh and Prosser (2014a) as "An Exact Branch and Bound Algorithm with Symmetry Breaking for the Maximum Balanced Induced Biclique Problem".

4.1 Maximum k-Cliques

When analysing real-world data, a clique may be too strong a requirement. A k-clique (or sometimes n-clique or s-clique—in an unfortunate clash of notation, "k-clique problem" is sometimes used elsewhere for the decision version of the clique problem, to distinguish it from the maximum clique problem) is a relaxed form of clique, where instead of requiring

each pair of vertices to be directly adjacent, we only require that they be connected by a path of length at most k (Luce, 1950). Thus a 1-clique *is* a clique, a 2-clique may be thought of as "a group of people, all of whom either know each other or have a mutual acquaintance", and so on. We illustrate this in Figure 4.1. Determining the size of a maximum k-clique is NP-hard for any fixed k (Bourjolly, Laporte, and Pesant, 2002).

A related relaxation is a k-club, which tightens the requirement of a k-clique as follows (Mokken, 1979). In a k-clique, each pair of vertices is connected by a path of length at most k, but that path may use any vertices in the original graph. In a k-club, each pair of vertices must be connected by a path of length at most k using only vertices that are also in the club (or equivalently, a k-club is an induced subgraph of diameter at most k). Thus the 2-clique in Figure 4.1 is *not* a 2-club (obviously, every k-club is a k-clique).

A recent survey by Shahinpour and Butenko (2013) discusses algorithms and results for k-clique and k-club problems. We adopt their notation of $\tilde{\omega}_k$ for the size of a maximum k-clique; the use of ω for the size of a maximum clique is standard. They note that "unlike the maximum clique problem, the maximum s-clique problem has not been the subject of extensive research and we are not aware of any computational results for this problem to date". This is in contrast to the k-club problem, for which a wide range of computational results are available (Bourjolly, Laporte, and Pesant, 2000; Bourjolly, Laporte, and Pesant, 2002; Carvalho and Almeida, 2016; Chang et al., 2013; Hartung, Komusiewicz, and Nichterlein, 2015; Mahdavi Pajouh and Balasundaram, 2012; Picker, 2015; Shahinpour and Butenko, 2013; Wotzlaw, 2014).

A maximum clique algorithm can easily be adapted to find a maximum k-clique in a graph G by considering the graph G^k: this graph has the same vertex set as G, and edges between any two distinct vertices v_1 and v_2 iff there is a path of length at most k between v_1 and v_2 in G. It is easy to see that maximum cliques in G^k correspond with maximum k-cliques in G (Balasundaram, Butenko, and Trukhanov, 2005). However, it is not obvious that this is a viable approach: even if G is sparse, G^k may not be, and the maximum clique problem on dense graphs can be very challenging computationally. Using Algorithm 2.1, we investigate whether this approach is feasible in practice. We modify the algorithm to include a new lazy "global domination" inference step—this technique provides no benefit for typical maximum

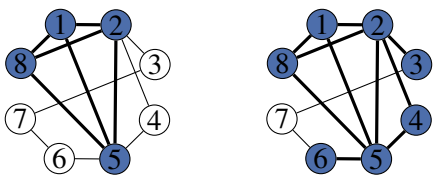


Figure 4.1: On the left, a graph, with its unique maximum clique $\{1, 2, 5, 8\}$ of size 4 highlighted. On the right, the same graph, with a maximum 2-clique $\{1, 2, 3, 4, 5, 6, 8\}$ of size 7 highlighted. This is not a 2-club, since the only path of length 2 between vertices 3 and 6 goes through vertex 7. A 3-clique covers the entire graph.

clique problems, but for maximum k-clique graphs it sometimes gives improvements of several orders of magnitude. We present computational results for the maximum k-clique problem on a range of benchmark and real-world graphs. We finish with a detailed look at random graphs.

4.1.1 Adapting a Maximum Clique Algorithm

Algorithm 4.1 is an adaptation of Algorithm 2.1 for solving the maximum k-clique problem. As in Chapter 2, we continue to use colouring as both a bound and an ordering heuristic. Vertices are coloured in a static non-increasing degree order, which is done by permuting the graph at the top of search (line 4). This algorithm does *not* use any of the more expensive dynamic tie-breaking mechanisms which were introduced in Chapter 2: although doing so can sometimes be beneficial for small dense graphs in a maximum clique context, for the larger graphs considered in this chapter, the cubic cost is prohibitively expensive. For the same reason, and additionally because our colour classes typically contain many vertices, we use a simple greedy colouring and do not use any of the colour repair steps, stronger MaxSAT-based inference, or colour class reordering techniques discussed in Chapter 2. Algorithm 4.1 differs from Algorithm 2.1 in just two ways: the input modification step, and the introduction of a new inference rule.

Reduction The first step (line 3) is to replace our input graph G with G^k. This graph may be constructed using a bounded breadth-first search: Chang et al. (2013) describe how to implement this quickly in practice.

Lazy global domination Aside from the G^k step, and making design choices which avoid cubic complexity operations (because we expect to be working with very large graphs), we make one further change to the maximum clique algorithm: we introduce a new "lazy global domination" rule which performs additional inference during search. This rule is not specific to the maximum k-clique problem, and is also valid for the maximum clique problem, but does not appear to be helpful for the usual clique instances.

Let v and w be distinct vertices in a graph G. We say that v *dominates* w if the neighbourhood of w, excluding v, is a (possibly non-strict) subset of the neighbourhood of v, excluding w. This is the tree-search / decision diagram state-oriented definition of domination in a clique context, rather than the usual graph definition: from a maximum clique perspective, this means that v is "better than" w. If v and w are adjacent, any clique containing w may always be extended by the inclusion of v; if v and w are non-adjacent, replacing w with v in any clique containing w cannot reduce the amount by which the clique may be grown.

Suppose a graph does contain one or more pairs of dominating vertices. We could make use of this fact during search in at least two ways. Firstly, when accepting a vertex w, we

Algorithm 4.1: Solving the maximum k-clique problem. The `colourOrder` function is the same as in Algorithm 2.1.

```
1  maxKClique :: (Graph G, Integer k) → Vertex Set
2  begin
3      G ← G^k
4      permute G into non-increasing degree order
5      global incumbent ← ∅
6      expand(∅, V(G))
7      return incumbent (unpermuted)
8  expand :: (Vertex Set solution, Vertex Set remaining)
9  begin
10     (order, bounds) ← colourOrder(remaining)
11     v_rej ← unset
12     for i ← |remaining| downto 1 do
13         if |solution| + bounds[i] ≤ |incumbent| then return
14         if v_rej ≠ unset then
15             remaining ← remaining \ dominated(v_rej)
16         v ← order[i]
17         if v ∈ remaining then
18             solution' ← solution + v
19             remaining' ← remaining ∩ N(G, v)
20             if remaining' ≠ ∅ then expand(solution', remaining')
21             else if |solution'| > |incumbent| then incumbent ← solution'
22             remaining ← remaining − v
23         v_rej ← v
24 dominated :: (Vertex v) → Vertex Set
25 begin
26     return {w ∈ V(G) : N(G, w) − v ⊆ N(G, v) − w}
```

may also unconditionally accept any vertex v which both dominates and is adjacent to w. Secondly, when rejecting a vertex v, we may also unconditionally reject any vertex w which is dominated by v. We could also choose to calculate domination globally (i.e. with respect to G^k, or even the original G), or locally (i.e. with respect to the subgraph of G^k induced by $solution \cup remaining$).

Detecting whether one vertex dominates another may be done in linear time (we discuss this further below), but finding all vertices dominated by a particular vertex is quadratic, and finding all dominations is cubic. This is a heavy price to pay, if there are no dominating vertices. This is why such a rule has not previously been used in the maximum clique context: the DIMACS graphs do not contain dominating vertices, and other graphs that do are too easy computationally for the step to be worthwhile.

However, some of the graphs we consider in the following section *do* contain dominating vertices, and although the maximum clique problem is trivial on these graphs, the maximum

k-clique problem is not for some values of k. Preliminary experiments suggested that the use of a domination rule could be extremely beneficial in certain circumstances, but that in cases where it had little effect, doing such a calculation introduced a substantial penalty to runtimes. Moreover, even in graphs where dominating vertices are present, knowing this fact is sometimes not useful: it is common for an optimal solution to be found straight away, and for the bound to be strong enough to prove optimality immediately, so no branching occurs.

This motivates the design of a lazy global domination rule. We perform our domination checks globally, with respect to G^k (which may contain more dominating vertices than G), and we remember and reuse the results of any domination checks we perform. We also only perform inference on the "reject" case, to avoid introducing any cost when a solution is found and proven optimal without branching.

The lines highlighted in Algorithm 4.1 show how this is done. When a vertex v_{rej} is rejected, we remove from $remaining$ any vertex that is dominated (with respect to G^k) by v_{rej}. This is line 15; the set of dominated vertices calculated here should be cached. One might expect that this calculation would appear after line 22. However, this introduces a cost if the bound allows the next choice of v to be eliminated. Thus we simply remember that we have rejected v by storing it in v_{rej} (line 23), and lazily postpone the filtering until after the bound has been checked.

Finally, note that we do not perform a new colouring when we reject dominated vertices—doing so typically does not lead to a smaller bound, since most colour classes contain many vertices. Thus when we select a v from $order$, it is now possible that v has already been rejected. We check for this on line line 17.

4.1.2 Experimental Results

We now give experimental results on a range of standard benchmarks, and on real-world and random graphs. Experiments were run on machines with dual Intel Xeon E5-2697A v4 processors and 512GBytes RAM running Ubuntu Linux 16.04; single-threaded runtimes are given, except in Table 4.2, where 64 threads were used (these machines have 32 real cores, and hyper-threading). Our software was implemented in C++, using C++11 native threads, and was compiled using GCC 5.4.0. The time taken to read in the graph from a file is excluded, but preprocessing time (including the construction of G^k and the bitset encoding) is included. We use the term *nodes* to refer to the number of recursive calls made by the branch-and-bound part of the algorithm (or equivalently, the number of times `colourOrder` is called).

We begin with a selection of real-world and standard benchmark graphs. We look at k equal to 2, 3 and 4 in every case—this is a standard practice for the k-club problem.

Erdős collaboration graphs In the first part of Table 4.1 we present experimental results from Erdős collaboration graphs from the Pajek dataset (Batagelj and Mrvar, 2006) which

Table 4.1: Experimental results for Erdős collaboration graphs, DIMACS clique graphs with diameter greater than two, and the smaller DIMACS clustering and partitioning graphs. For each graph, we consider k equal to 2, 3 and 4. In each case we show the density of G^k, the size of a maximum k-clique, and then for both the unmodified algorithm and the algorithm with our lazy global domination step, the number of nodes required, and the runtime.

Instance	k	D	$\tilde{\omega}_k$	Unmodified		With Domination	
				Nodes	Runtime	Nodes	Runtime
Erdős collaboration graphs							
Erdos971	2	0.09	42	42	5 ms	42	4 ms
$\|V\| = 472$	3	0.31	117	121	7 ms	119	9 ms
$\|E\| = 1314$	4	0.56	235	468	17 ms	468	12 ms
Erdos972	2	0.01	258	258	1.1 s	258	1.2 s
$\|V\| = 5488$	3	0.09	517	537	1.5 s	521	1.4 s
$\|E\| = 8972$	4	0.35	1509	10976953	>1.0 h	8197	3.8 s
Erdos981	2	0.09	43	43	7 ms	43	5 ms
$\|V\| = 485$	3	0.31	123	358	9 ms	354	9 ms
$\|E\| = 1381$	4	0.57	245	246	14 ms	246	13 ms
Erdos982	2	0.01	274	274	1.2 s	274	1.4 s
$\|V\| = 5822$	3	0.09	547	555	1.5 s	547	1.7 s
$\|E\| = 9505$	4	0.35	1594	11563239	>1.0 h	618826	196.2 s
Erdos991	2	0.09	43	44	6 ms	44	5 ms
$\|V\| = 492$	3	0.31	126	375	9 ms	374	9 ms
$\|E\| = 1417$	4	0.57	246	491	15 ms	491	15 ms
Erdos992	2	0.01	277	277	1.1 s	277	953 ms
$\|V\| = 6100$	3	0.09	562	573	1.1 s	562	1.2 s
$\|E\| = 9939$	4	0.35	1643	10762095	>1.0 h	202543	68.7 s
Erdos02	2	0.02	508	508	915 ms	508	1.8 s
$\|V\| = 6927$	3	0.20	1014	1022	2.6 s	1015	2.7 s
$\|E\| = 11850$	4	1.00	6927	6927	12.1 s	6927	12.5 s
DIMACS clique graphs							
c-fat200-1	2	0.13	18	41	1 ms	35	1 ms
$\|V\| = 200$	3	0.19	24	74	2 ms	48	1 ms
$\|E\| = 1534$	4	0.24	30	134	2 ms	65	2 ms
c-fat200-2	2	0.27	35	35	2 ms	35	2 ms
$\|V\| = 200$	3	0.39	46	488	3 ms	102	3 ms
$\|E\| = 3235$	4	0.50	57	1496	6 ms	128	4 ms
c-fat200-5	2	0.71	87	11513	25 ms	257	6 ms
$\|V\| = 200$	3	1.00	200	200	11 ms	200	9 ms
$\|E\| = 8473$	4	1.00	200	200	13 ms	200	13 ms
c-fat500-1	2	0.06	21	52	5 ms	43	5 ms
$\|V\| = 500$	3	0.09	28	28	5 ms	28	5 ms
$\|E\| = 4459$	4	0.11	35	35	6 ms	35	7 ms
c-fat500-2	2	0.12	39	134	7 ms	79	7 ms
$\|V\| = 500$	3	0.17	52	52	10 ms	52	10 ms
$\|E\| = 9139$	4	0.22	65	65	13 ms	65	12 ms
c-fat500-5	2	0.31	96	10133	49 ms	196	26 ms
$\|V\| = 500$	3	0.44	128	128	41 ms	128	41 ms
$\|E\| = 23191$	4	0.56	159	1357762269	>1.0 h	326	79 ms

continued on next page...

Instance	k	D	$\tilde{\omega}_k$	Unmodified		With Domination	
				Nodes	Runtime	Nodes	Runtime
c-fat500-10	2	0.62	189	1021506910	>1.0 h	560	59 ms
$\|V\| = 500$	3	0.87	252	252	105 ms	252	94 ms
$\|E\| = 46627$	4	1.00	500	500	139 ms	500	135 ms
p-hat300-1	2	1.00	299	299	11 ms	299	11 ms
$\|V\| = 300$	3	1.00	300	300	34 ms	300	38 ms
$\|E\| = 10933$	4	1.00	300	300	34 ms	300	36 ms
DIMACS partitioning graphs							
3elt	2	0.00	10	340	735 ms	340	801 ms
$\|V\| = 4720$	3	0.01	16	1582	935 ms	1582	1.0 s
$\|E\| = 13722$	4	0.01	27	911	978 ms	911	1.1 s
4elt	2	0.00	11	486	9.3 s	486	10.6 s
$\|V\| = 15606$	3	0.00	20	717	9.3 s	717	11.5 s
$\|E\| = 45878$	4	0.00	36	345	9.5 s	345	10.1 s
add20	2	0.04	124	124	213 ms	124	163 ms
$\|V\| = 2395$	3	0.25	671	671	309 ms	671	276 ms
$\|E\| = 7462$	4	0.67	1454	1454	595 ms	1454	649 ms
add32	2	0.00	32	32	1.0 s	32	1.1 s
$\|V\| = 4960$	3	0.01	99	286	782 ms	194	774 ms
$\|E\| = 9462$	4	0.03	268	268	804 ms	268	1.1 s
bcsstk29	2	0.01	72	9752	6.8 s	963	10.9 s
$\|V\| = 13992$	3	0.02	132	27188730	1725.7 s	7781	17.0 s
$\|E\| = 302748$	4	0.04	210	27977976	>1.0 h	21689	23.1 s
bcsstk30	2	0.01	219	224	41.5 s	219	41.8 s
$\|V\| = 28924$	3	0.03	496	509	45.1 s	496	45.4 s
$\|E\| = 1007284$	4	0.05	843	854	54.6 s	845	52.1 s
bcsstk31	2	0.00	189	189	68.8 s	189	67.4 s
$\|V\| = 35588$	3	0.01	278	605	71.4 s	369	70.7 s
$\|E\| = 572914$	4	0.02	428	119640	136.8 s	6588	81.1 s
bcsstk33	2	0.03	141	141	3.8 s	141	3.8 s
$\|V\| = 8738$	3	0.08	228	26033	9.7 s	1744	6.4 s
$\|E\| = 291583$	4	0.15	435	1974622	434.6 s	52779	21.1 s
crack	2	0.00	10	2894	4.8 s	2894	11.0 s
$\|V\| = 10240$	3	0.00	17	4996	5.0 s	4987	13.5 s
$\|E\| = 30380$	4	0.01	31	2173	5.1 s	2173	10.2 s
cs4	2	0.00	6	5780	25.1 s	5780	96.5 s
$\|V\| = 22499$	3	0.00	12	7812	25.7 s	7812	104.3 s
$\|E\| = 43858$	4	0.00	18	29032	26.9 s	29032	179.7 s
cti	2	0.00	7	8918	11.6 s	8918	79.1 s
$\|V\| = 16840$	3	0.00	15	6406	13.7 s	6406	59.0 s
$\|E\| = 48232$	4	0.01	26	62316	15.9 s	62316	148.1 s
data	2	0.01	18	638	300 ms	617	342 ms
$\|V\| = 2851$	3	0.02	32	4982	292 ms	4913	388 ms
$\|E\| = 15093$	4	0.04	52	40095	566 ms	36089	626 ms
fe-4elt2	2	0.00	13	61	6.9 s	61	5.7 s
$\|V\| = 11143$	3	0.00	20	389	5.7 s	389	7.8 s
$\|E\| = 32818$	4	0.01	32	448	5.7 s	446	6.8 s
fe-pwt	2	0.00	16	95	64.9 s	95	74.4 s
$\|V\| = 36519$	3	0.00	29	167	70.5 s	167	70.5 s
$\|E\| = 144794$	4	0.00	52	224	72.8 s	224	75.4 s

continued on next page...

Instance	k	D	$\tilde{\omega}_k$	Unmodified		With Domination	
				Nodes	Runtime	Nodes	Runtime
fe-sphere	2	0.00	7	14173	14.4 s	14173	72.4 s
$\|V\| = 16386$	3	0.00	12	34328	14.9 s	34328	140.8 s
$\|E\| = 49152$	4	0.00	19	73632	16.4 s	73632	189.0 s
memplus	2	0.02	574	574	16.7 s	574	19.1 s
$\|V\| = 17758$	3	0.26	8057	8061	54.0 s	8058	59.4 s
$\|E\| = 54196$	4	0.74	8963	8963	101.7 s	8963	98.2 s
uk	2	0.00	5	433	767 ms	433	1.1 s
$\|V\| = 4824$	3	0.00	8	1891	875 ms	1891	1.2 s
$\|E\| = 6837$	4	0.01	14	2168	942 ms	2168	1.1 s
vibrobox	2	0.02	121	302	7.4 s	302	9.4 s
$\|V\| = 12328$	3	0.08	408	1984	10.5 s	1984	11.8 s
$\|E\| = 165250$	4	0.26	$\geq$1094	6574637	>1.0 h	6521169	>1.0 h
whitaker3	2	0.00	9	1222	2.0 s	1222	6.2 s
$\|V\| = 9800$	3	0.00	15	3724	3.3 s	3724	8.9 s
$\|E\| = 28989$	4	0.01	23	6530	2.9 s	6530	11.9 s
wing-nodal	2	0.01	29	648	4.0 s	648	6.4 s
$\|V\| = 10937$	3	0.02	54	13091	5.5 s	13039	17.7 s
$\|E\| = 75488$	4	0.04	114	60230499	2302.3 s	60230217	2325.3 s
DIMACS clustering graphs							
adjnoun	2	0.50	50	50	0 ms	50	0 ms
$\|V\| = 112$	3	0.91	83	164	0 ms	164	1 ms
$\|E\| = 425$	4	0.99	107	107	1 ms	107	1 ms
as-22july06	2	0.04	2391	2391	14.5 s	2391	14.8 s
$\|V\| = 22963$	3	0.36	8455	345770	>1.0 h	94497	977.9 s
$\|E\| = 48436$	4	0.79	14911	14911	212.0 s	14911	216.3 s
astro-ph	2	0.01	361	365	6.8 s	362	15.4 s
$\|V\| = 16706$	3	0.10	1553	1567	16.3 s	1560	25.7 s
$\|E\| = 121251$	4	0.35	$\geq$4040	987476	>1.0 h	954213	>1.0 h
celegans-meta	2	0.44	238	238	8 ms	238	7 ms
$\|V\| = 453$	3	0.89	371	371	24 ms	371	21 ms
$\|E\| = 2025$	4	0.98	432	432	20 ms	432	20 ms
celegansneural	2	0.55	135	135	3 ms	135	5 ms
$\|V\| = 297$	3	0.95	245	245	11 ms	245	9 ms
$\|E\| = 2148$	4	1.00	295	295	12 ms	295	18 ms
cond-mat	2	0.00	108	108	5.6 s	108	5.7 s
$\|V\| = 16726$	3	0.01	250	1403	7.2 s	844	6.7 s
$\|E\| = 47594$	4	0.05	720	8464437	>1.0 h	674453	307.0 s
cond-mat-2003	2	0.00	203	204	27.2 s	204	26.0 s
$\|V\| = 31163$	3	0.02	$\geq$629	7459261	>1.0 h	7477245	>1.0 h
$\|E\| = 120029$	4	0.12	$\geq$2605	1810714	>1.0 h	1604058	>1.0 h
cond-mat-2005	2	0.00	279	279	56.0 s	279	79.4 s
$\|V\| = 40421$	3	0.03	$\geq$1060	1882237	>1.0 h	1993460	>1.0 h
$\|E\| = 175691$	4	0.16	$\geq$4185	518583	>1.0 h	517273	>1.0 h
dolphins	2	0.32	14	14	0 ms	14	0 ms
$\|V\| = 62$	3	0.59	30	30	0 ms	30	0 ms
$\|E\| = 159$	4	0.77	40	40	0 ms	40	0 ms
email	2	0.09	72	72	34 ms	72	30 ms
$\|V\| = 1133$	3	0.45	233	19031	299 ms	19031	298 ms
$\|E\| = 5451$	4	0.86	654	6854	277 ms	6676	270 ms

continued on next page...

Instance	k	D	$\tilde{\omega}_k$	Unmodified		With Domination	
				Nodes	Runtime	Nodes	Runtime
football	2	0.45	17	147	0 ms	145	0 ms
$\lvert V\rvert = 115$	3	0.95	69	70	1 ms	70	0 ms
$\lvert E\rvert = 613$	4	1.00	115	115	1 ms	115	1 ms
hep-th	2	0.00	51	51	3.1 s	51	2.3 s
$\lvert V\rvert = 8361$	3	0.01	125	239	2.3 s	176	2.5 s
$\lvert E\rvert = 15751$	4	0.04	347	158164	24.3 s	23714	6.0 s
jazz	2	0.69	103	107	2 ms	107	3 ms
$\lvert V\rvert = 198$	3	0.95	174	174	8 ms	174	7 ms
$\lvert E\rvert = 2742$	4	0.99	192	192	7 ms	192	7 ms
karate	2	0.61	18	18	0 ms	18	0 ms
$\lvert V\rvert = 34$	3	0.86	25	25	0 ms	25	0 ms
$\lvert E\rvert = 78$	4	0.99	33	33	0 ms	33	0 ms
lesmis	2	0.43	37	37	0 ms	37	0 ms
$\lvert V\rvert = 77$	3	0.85	58	58	0 ms	58	0 ms
$\lvert E\rvert = 254$	4	0.99	75	75	0 ms	75	0 ms
netscience	2	0.01	35	35	45 ms	35	45 ms
$\lvert V\rvert = 1589$	3	0.01	54	54	47 ms	54	47 ms
$\lvert E\rvert = 2742$	4	0.02	85	85	48 ms	85	48 ms
PGPgiantcompo	2	0.00	206	206	3.7 s	206	3.9 s
$\lvert V\rvert = 10680$	3	0.02	423	843	5.9 s	841	6.0 s
$\lvert E\rvert = 24316$	4	0.07	1161	1161	8.5 s	1161	7.3 s
polblogs	2	0.27	352	352	86 ms	352	97 ms
$\lvert V\rvert = 1490$	3	0.58	776	2210	492 ms	2177	457 ms
$\lvert E\rvert = 16715$	4	0.66	1127	1537	683 ms	1166	747 ms
polbooks	2	0.37	28	28	0 ms	28	1 ms
$\lvert V\rvert = 105$	3	0.64	54	54	0 ms	54	0 ms
$\lvert E\rvert = 441$	4	0.86	68	68	1 ms	68	1 ms
power	2	0.00	20	20	1.1 s	20	697 ms
$\lvert V\rvert = 4941$	3	0.00	30	30	999 ms	30	984 ms
$\lvert E\rvert = 6594$	4	0.01	61	61	1.1 s	61	922 ms

we introduced at the beginning of this chapter. We were able to solve all of these problems in under four minutes (and all but three in under four seconds) when using the domination rule. However, using the unmodified maximum clique algorithm, three of these results did not finish running within one hour. Note that for $k = 4$, a k-clique covers all of "Erdos02".

In several cases, the algorithm found and proved an optimal solution immediately ($\tilde{\omega}_k$ is equal to the number of search nodes). This illustrates the necessity of laziness: if we simply computed dominating pairs upfront, we would be paying a cubic preprocessing cost for an algorithm which is effectively quadratic in practice.

By comparing these results with the k-club results of Chang et al. (2013), we see that in all but four cases the k-clique and k-club numbers are equal; all of these differences occur when $k = 4$. (Chang et al. did not investigate the "Erdos02" graph, but Wotzlaw confirmed privately that the k-clique and k-club numbers are the same here too.) On the other hand, the k-clique numbers are sometimes much easier to find, both algorithmically and computationally.

Clique graphs In the second part of Table 4.1 we present results from the "clique" graphs from the Second DIMACS implementation challenge, which we discussed in Chapter 2. Nearly all of these graphs have diameter 2, so a 2-clique covers the entire graph—these instances are therefore excluded. The only exceptions are the "c-fat" family (all of which are trivial for a maximum clique solver), and one of the "p_hat" graphs.

With the domination rule, we solve all of these problems within a tenth of a second. Without, two of the results take over an hour, and the rest remain trivial. Note that in several cases, for some values of k a k-clique covers the entire graph. Again using Chang et al.'s results, we see that for the first six graphs in this table the k-clique and k-club numbers are the same for each value of k (Chang et al. did not investigate "c-fat500-10" or "p-hat300-1").

Partitioning graphs The third part of Table 4.1 presents results from the smallest 20 partitioning graphs from the 10th DIMACS Implementation Challenge (Walshaw, 2016). Many of these graphs are considerably larger than those typically considered for the maximum clique problem, and we might expect our $O(|V|^2)$ memory requirements to cause problems. Nonetheless, with the domination rule there is only one instance which we were unable to solve within an hour (and without the domination rule, there are two).

On the other hand, we sometimes see a significant cost where the domination rule does not help, and where the proof of optimality is not immediate: in "3elt" and "4elt", our runtimes increase by 10%, and for "cs4" and "cti" the slowdown sometimes approaches a factor of ten. Thus laziness can be costly when the rule is used, but useless. However, the instance where both variants timed out shows that for long-running instances, the rate of nodes per second does not change substantially (although it is much lower than it is for the instances in Chapter 2).

Five of these graphs were considered for the k-club problem by Wotzlaw (2014). In every case, the k-clique and k-club numbers are the same. However, the k-clique number was again consistently much easier to find.

Clustering graphs The final part of Table 4.1 presents results from the smallest 20 clustering graphs from the 10th DIMACS Implementation Challenge (Meyerhenke, 2011). Again, from a maximum clique perspective these would be considered unusually large graphs. However, only five were unsolvable within an hour (plus a further two when the domination rule was not used), and over half of the problems took under two seconds.

Seven of these graphs were considered for the k-club problem by Wotzlaw (2014). In these cases, the 2-clique and 2-club numbers are the same, except for "football" where the 2-club number is 16 but the 2-clique number is 17; for $k = 3$ and $k = 4$ there are some differences. There is a large difference in computational difficulty between the k-clique and k-club problems: for "polblogs" with $k = 3$ and $k = 4$, Wotzlaw was unable to prove

				Sequential		Parallel	
Instance	k	D	$\tilde{\omega}_k$	Nodes	Runtime	Nodes	Runtime
astro-ph	2	0.01	361	365	15.4 s	101642	16.8 s
$\|V\| = 16706$	3	0.10	1553	1567	25.7 s	52576	26.8 s
$\|E\| = 121251$	4	0.35	≥4125	987476	>1.0 h	1160769758	>24.0 h
cond-mat	2	0.00	108	108	5.7 s	16981	5.9 s
$\|V\| = 16726$	3	0.01	250	1403	6.7 s	51222	16.1 s
$\|E\| = 47594$	4	0.05	720	8464437	307.0 s	89288	12.1 s
cond-mat-2003	2	0.00	203	204	26.0 s	40145	47.5 s
$\|V\| = 31163$	3	0.02	634	7459261	>1.0 h	467793281	1.8 h
$\|E\| = 120029$	4	0.12	≥2609	1810714	>1.0 h	1896826741	>24.0 h
cond-mat-2005	2	0.00	279	279	79.4 s	84011	90.8 s
$\|V\| = 40421$	3	0.03	1060	1882237	>1.0 h	1117533785	12.1 h
$\|E\| = 175691$	4	0.16	≥4271	518583	>1.0 h	544501020	>24.0 h
vibrobox	2	0.02	121	302	9.4 s	13249	4.7 s
$\|V\| = 12328$	3	0.08	408	1984	11.8 s	27061	19.6 s
$\|E\| = 165250$	4	0.26	≥1107	6574637	>1.0 h	7374855043	>24.0 h
wing-nodal	2	0.01	29	648	6.4 s	11601	7.2 s
$\|V\| = 10937$	3	0.02	54	13091	17.7 s	23917	16.4 s
$\|E\| = 75488$	4	0.04	114	60230499	2325.3 s	67418	18.1 s

Table 4.2: Experimental results for multi-threaded search on harder instances, using 64 threads. For each graph, we consider k equal to 2, 3 and 4. In each case we show the density of G^k, the size of a maximum k-clique, and then for both the sequential algorithm and the parallel algorithm (with lazy global domination in both cases), the number of nodes required, and the runtime.

optimality within an hour, but we required less than two seconds to do so. In both of these cases the k-clique and k-club numbers are the same.

4.1.3 Parallel Search

What about our parallel search? We selected the four graphs which we were unable to solve (for some values of k), along with two of the challenging graphs which required substantial amounts of search (at least half a million nodes) to solve, and repeated the experiments using parallel search with 64 threads and the domination rule enabled, using the resplitting strategy described in Chapter 3. To make laziness thread-safe, a simple mutex lock can be used to protect each entry of the `dominated` table.

The results are shown in Table 4.2. Parallel search allowed us to close two more instances, and gave improved bounds on the four remaining instances within 24 hours. However, to get sufficient work balance, we had to allow for work splitting to depth 10 rather than depth 3. A close inspection of the search patterns showed that simpler static decomposition approaches such as those proposed by Depolli et al. (2013), Malapert, Régin, and Rezgui (2016), and San Segundo, Lopez, and Pardalos (2016) would give little to no speedup on many harder k-clique instances, unless it were somehow possible to generate $\mathcal{O}(|V|^{10})$ subproblems.

For the easier instances where the sequential runtime is known, the parallel search did more work—this is to be expected, since the work distribution approach from Section 3.3.1 recomputes parts of the search space (to allow for more efficient mutable data structures to be used during search), and speculative parallelism is unlikely to contribute to the solution when the sequential search tree is small. However, despite the extra work, and despite having to introduce overhead into early stages of the algorithm to allow for work stealing, and despite having more processing power but not more memory bandwidth, in no cases were the parallel runtimes vastly longer than the sequential runtimes (although in many cases they were not better either). We also did not parallelise the construction of G^k or the preprocessing stage of the algorithm, which in many cases dominated the runtime: the efficient sequential algorithm by Chang et al. (2013) is not obviously parallelisable, and it would be interesting future work to tackle this problem.

As well as improving the results on the instances we could not solve sequentially, parallel search sometimes gave large improvements for the easier instances. For "wing-nodal" with $k = 4$, the parallel run did much less work than the sequential run: a speedup of over 100 was obtained from 64 threads (in fact three threads suffices to see this). A similar effect occurred with "cond-mat" and $k = 4$. This is because the work splitting mechanism we used explicitly diversifies at the top of search first, where branching heuristics are least likely to be correct, which leads to an initial incumbent being found faster—this is in line with our observations in Chapter 3 that tailored work stealing should be favoured over randomised work stealing for combinatorial search problems.

4.1.4 Random Graphs

Recall that an Erdős-Rényi random graph $G(n, p)$ has n vertices, and an edge between each distinct pair of vertices with probability p, chosen independently. We now investigate the size of a maximum k-clique in such graphs, and the complexity of finding it. In each case, we use an average over 10,000 samples for every point.

In the left-hand plot of Figure 4.2 we illustrate the average value of $\tilde{\omega}_k$ in $G(200, p)$ for different values of k, and a range of values of p for the x-axis. We see that even for very low edge probabilities, a maximum k-clique quickly covers the entire graph. (This is in contrast to the maximum clique problem, where a maximum clique does not even cover a quarter of the graph for edge probabilities below 0.75.) On the right we show the average size of the search space (number of nodes, or recursive calls made) for the same problem. We see that there is a complexity peak for each k, although the peak is much smaller for $k = 4$ than it is for $k = 3$, which is in turn much smaller than it is for $k = 2$. The peak also occurs for lower edge probabilities as k increases. For contrast, for the maximum clique problem, the peak occurs at around edge probability 0.9, and is two orders of magnitude larger.

In Figure 4.3 we show the effect of changing n and fixing $k = 2$. As n increases from

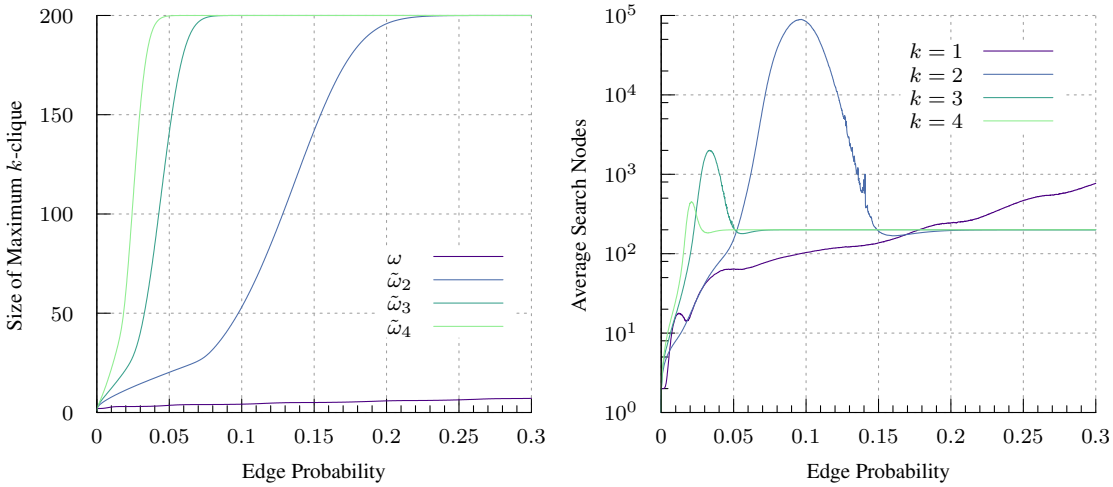


Figure 4.2: On the left, values of $\tilde{\omega}_k$ for random graphs $G(200, p)$, with varying edge probabilities. We see that even for very low edge probabilities, a maximum k-clique quickly covers the entire graph when $k > 1$. On the right, search space size. We see that 4-clique is easier than 3-clique in practice, which in turn is easier than 2-clique. (The complexity peak for maximum clique occurs at around edge probability 0.9, and requires approximately 15 million search nodes.)

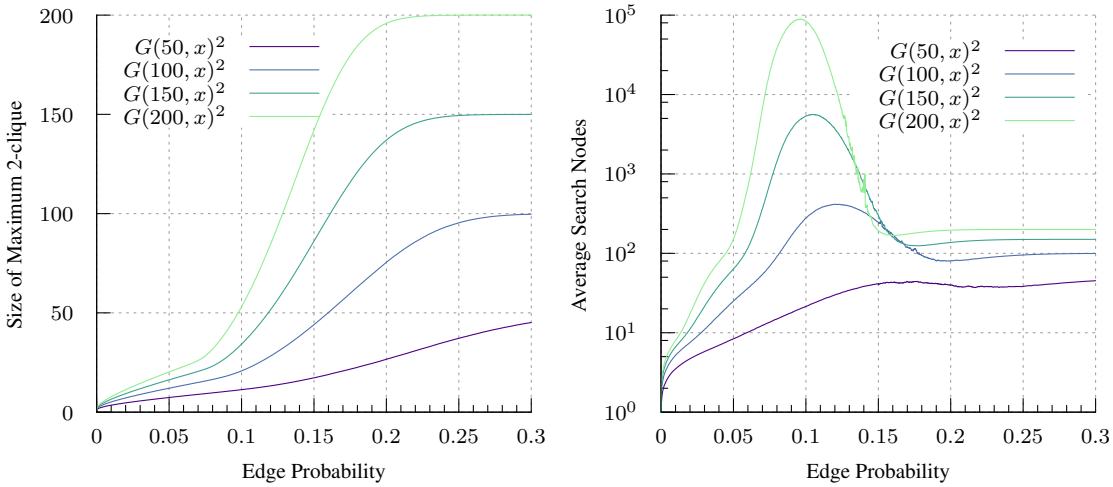


Figure 4.3: On the left, the size of a maximum 2-clique in random graphs $G(n, p)$ with varying edge probabilities, and different values of n. For $G(50, p)$, a 2-clique has size average 50 from $p = 0.42$ onwards. On the right, the search space size. As n increases, the complexity peak grows and moves slowly to the left.

50 to 200, the complexity peak becomes much more pronounced, and shifts slightly towards the left (lower edge probabilities). As expected, a constant increase in the number of vertices gives a roughly exponential increase in the difficulty of the hardest instances.

Interestingly, we do not see evidence of the "wobbly line" behaviour that was present for the maximum clique problem in Chapter 2. The shape of the maximum k-clique curves is also different to the maximum k-clique curves, with a flattening growth rate at higher densities. It would be interesting to have an analytic explanation of these behaviours.

4.2 Maximum Labelled Cliques

Having seen that Algorithm 2.1 is flexible enough to solve a related problem by reduction, we now turn to a clique problem with side constraints. Carrabs, Cerulli, and Dell'Olmo (2014) introduced a variant of the maximum clique problem called the *maximum labelled clique problem*, and gave example applications involving telecommunications and social network analysis. In this variant, each edge in the graph has a label, and we are given a budget b: we seek to find as large a clique as possible, but the edges in our selected clique may not use more than b different labels in total (representing, for example, a set of interconnected data centres relying upon only a small number of telecommunications providers, or a close-knit group of friends who additionally share a small number of common interests). In the case that there is more than one such maximum, we must find the one using fewest different labels. We illustrate these concepts in Figure 4.4, using an example graph due to Carrabs, Cerulli, and Dell'Olmo; our four labels are shown using different styled edges.

Carrabs, Cerulli, and Dell'Olmo proposed a mathematical programming approach to solving the problem, and used CPLEX to provide experimental results on a range of graph instances. Here we introduce the first dedicated algorithm for the maximum labelled clique problem, and then describe how it may be parallelised. We evaluate our implementation experimentally, and show that it is consistently faster than that of Carrabs, Cerulli, and Dell'Olmo, sometimes by four or five orders of magnitude.

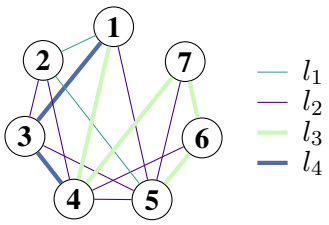


Figure 4.4: A graph with maximum clique $\{1, 2, 3, 4, 5\}$, using all four edge labels. If our budget is only three, a maximum feasible clique has size four. There are several such cliques, but $\{4, 5, 6, 7\}$ is optimal since it uses only two labels, whilst every other uses at least three.

4.2.1 Definitions

In this section we are working with graphs which have a label on each edge. The *cost* of a clique is the cardinality of the union of the labels associated with all of its edges. A clique is *feasible* if it has cost not greater than the budget. We say that a feasible clique C is *better than* a feasible clique C' if either it has larger cardinality, or if it has the same cardinality but lower cost. The *maximum labelled clique problem* is to find a feasible clique which is either better than or equal to any other feasible clique in a given graph—that is, of all the maximum feasible cliques, we seek the cheapest.

The hardness of the maximum clique problem immediately implies that the maximum labelled clique problem is also NP-hard. Carrabs, Cerulli, and Dell'Olmo showed that the problem remains hard even for complete graphs, where the maximum clique problem is trivial.

4.2.2 A Branch and Bound Algorithm

In Algorithm 4.2 we present the first dedicated algorithm for the maximum labelled clique problem. Like Algorithm 2.1, this is a branch and bound algorithm, using a greedy colouring for the bound, but it has been extended to recognise labels, and uses two passes to handle the objective.

Labels and the budget: On the first pass ($first$ = **true**, from line 5), we concentrate on finding the largest feasible clique, but do not worry about finding the cheapest such clique. To do so, we store the labels currently used in $solution$ in the variable $labels$. When we add a vertex v to $solution$, we create from $labels$ a new label set $labels'$ and add to it any additional labels used (line 15). Now we check whether we have exceeded the budget (line 16), and only proceed with this value of $solution$ if we have not. As well as storing $incumbent$, we also keep track of the labels it uses in $incumbentLabels$.

On the second pass ($first$ = **false**, from line 6), we already have the size of a maximum feasible clique in $|incumbent|$, and we seek to either reduce the cost $|incumbentLabels|$, or prove that we cannot do so. Thus we repeat the search, starting with our existing values of $incumbent$ and $incumbentLabels$, but instead of using the budget to filter labels on line 16, we use $|incumbentLabels| - 1$ (which can become smaller as cheaper solutions are found). We must also change the bound condition slightly: rather than looking only for solutions strictly larger than $incumbent$, we are now looking for solutions with size equal to $incumbent$ (line 12). Finally, when potentially unseating the incumbent (line 18), we must check to see if either $solution$ is larger than $incumbent$, or it is the same size but cheaper.

This two-pass approach is used to avoid spending a long time trying to find a cheaper clique of size $|incumbent|$, only for this effort to be wasted when a larger clique is found. The additional filtering power from having found a clique containing only one additional vertex is often

Algorithm 4.2: An algorithm for the maximum labelled clique problem. The `colourOrder` function is the same as in Algorithm 2.1.

```
1  maxLabelledClique :: (Graph G, Int budget) → Vertex Set
2  begin
3      permute G so that vertices are in non-increasing degree order
4      global (incumbent, incumbentLabels) ← (∅, ∅)
5      expand(true, ∅, V(G), ∅)
6      expand(false, ∅, V(G), ∅)
7      return incumbent (unpermuted)

8  expand :: (Boolean first, Vertex Set solution, Vertex Set remaining, Label Set labels)
9  begin
10     (order, bounds) ← colourOrder(remaining)
11     for i ← |remaining| downto 1 do
12         if |solution| + bounds[i] < |incumbent| ∨
                    (first ∧ |solution| + bounds[i] = |incumbent|) then return
13         v ← order[i]
14         solution ← solution + v
15         labels′ ← labels ∪ the labels of edges between v and any vertex in solution
16         if |labels′| ≤ (budget if first, otherwise |incumbentLabels| − 1) then
17             if (solution, labels′) is better than (incumbent, incumbentLabels) then
18                 (incumbent, incumbentLabels) ← (solution, labels′)
19             remaining′ ← the vertices in remaining that are adjacent to v
20             if remaining′ ≠ ∅ then expand(first, solution, remaining′, labels′)
21         solution ← solution − v
22         remaining ← remaining − v
```

extremely beneficial. On the other hand, label-based filtering using $|incumbentLabels| - 1$ rather than the budget is not possible until we are sure that $incumbent$ cannot grow further, since it could be that larger feasible maximum cliques have a higher cost.

Bit parallelism: As in Algorithm 2.1, $remaining$ should be a bitset. Additionally, the operations performed on $labels$ are all bitset-friendly—recall from Section 1.6.1 that determining the cardinality of a bitset is also a dedicated hardware instruction in modern processors.

Note that $solution$ should *not* be stored as a bitset, to speed up line 15. Instead, it should be an array. Adding a vertex to $solution$ on line 14 may be done by appending to the array, and when removing a vertex from $solution$ on line 21 we simply remove the last element—this works because $solution$ is used like a stack.

Thread parallelism: We must be slightly careful when introducing thread parallelism to this algorithm. The incumbent is accessed regularly and updated rarely. To avoid requiring mutex locking, which showed severe scalability limits in preliminary experiments, we use a single atomic variable as follows: we map both $incumbent$ and $incumbentLabels$ into a large

unsigned integer, allocating the higher order bits to $|incumbent|$ and the lower order bits to the bitwise complement of $|incumbentLabels|$. This respects the natural comparison order for unsigned integers, allowing both variables to be compared and updated simultaneously.

A further complication is that in the first pass, we could find an equally sized but more costly incumbent than we would find sequentially. Thus we cannot even guarantee that parallel search not cause a slowdown in certain cases, and so our guarantees from Section 1.6.6 do not necessarily hold.

4.2.3 Experimental Results

We now evaluate an implementation of our sequential and parallel algorithms experimentally. Our implementation was coded in C++, and for parallelism, C++11 native threads were used. The bitset encoding was used in both cases. Because we find this problem very easy, experimental results are produced on a desktop machine with an Intel i5-3570 CPU and 12GBytes of RAM. This is a dual core machine, with hyper-threading, so for parallel results we use four threads (but should not expect an ideal-case speedup of 4). Sequential results are from a dedicated sequential implementation, not from a parallel implementation run with a single thread. Timing results include preprocessing time and thread startup costs, but not the time taken to read in the graph file and generate random labels.

Standard benchmark problems In Table 4.3 we present results from the same set of benchmark instances as Carrabs, Cerulli, and Dell'Olmo (2014). These are some of the smaller graphs from the Second DIMACS Implementation Challenge, with randomly allocated labels. Carrabs, Cerulli, and Dell'Olmo used three samples for each measurement, and presented the average; we use one hundred. Note that our CPU is newer than that of Carrabs, Cerulli, and Dell'Olmo, and we have not attempted to scale their results for a "fair" comparison.

The most significant result is that none of our parallel runtime averages are above seven seconds, and none of our sequential runtime averages are above twenty four seconds (our worst sequential runtime from any instance is 32.3 seconds, and our worst parallel runtime is 8.4 seconds). This is in stark contrast to Carrabs, Cerulli, and Dell'Olmo, who aborted some of their runs on these instances after three hours. Most strikingly, the keller4 instances, which all took Carrabs, Cerulli, and Dell'Olmo at least an hour, took under 0.1 seconds for our parallel algorithm. We are using a different model CPU, so results are not directly comparable, but we strongly doubt that hardware differences could contribute to more than one order of magnitude improvement in the runtimes.

We also see that parallelism is in general useful, and is never a penalty, even with very low runtimes. We see a speedup of between 3 and 4 on the non-trivial instances. This is despite the initial sequential portion of the algorithm, the cost of launching the threads, the general

$\|L\|$	25% budget					50% budget					75% budget				
	Size	Cost	Seq	Par	Enh	Size	Cost	Seq	Par	Enh	Size	Cost	Seq	Par	Enh
johnson8-2-4															
4	3.13	1.00	0.00	0.00	0.03	4.00	1.87	0.00	0.00	0.02	4.00	1.87	0.00	0.00	0.01
8	3.51	1.50	0.00	0.00	0.03	4.00	2.48	0.00	0.00	0.01	4.00	2.48	0.00	0.00	0.02
12	4.00	2.85	0.00	0.00	0.01	4.00	2.85	0.00	0.00	0.02	4.00	2.85	0.00	0.00	0.02
MANN_a9															
11	5.64	2.76	0.01	0.00	3.18	8.89	5.93	0.02	0.01	13.98	13.34	8.99	0.01	0.00	6.47
21	6.61	5.60	0.02	0.01	12.43	9.74	10.68	0.08	0.02	46.52	13.32	15.73	0.03	0.01	15.55
32	7.00	7.79	0.04	0.01	17.61	10.26	14.99	0.19	0.05	108.13	14.12	23.28	0.03	0.01	13.18
hamming6-4															
6	3.99	1.97	0.00	0.00	0.32	4.00	1.99	0.00	0.00	0.33	4.00	1.99	0.00	0.00	0.35
11	4.00	2.64	0.00	0.00	0.43	4.00	2.64	0.00	0.00	0.42	4.00	2.64	0.00	0.00	0.37
17	4.00	2.98	0.00	0.00	0.41	4.00	2.98	0.00	0.00	0.43	4.00	2.98	0.00	0.00	0.44
hamming6-2															
15	6.07	3.82	0.04	0.01	41.32	9.87	7.86	0.76	0.21	382.73	15.42	12.00	2.44	0.66	711.20
29	7.17	7.12	0.41	0.11	221.77	11.01	14.73	6.25	1.70	2368.78	15.86	21.84	11.73	3.14	2079.50
44	8.00	10.81	1.35	0.37	429.21	12.00	21.49	20.19	5.47	4955.96	17.13	32.65	23.15	6.28	2170.84
johnson8-4-4															
14	5.99	3.93	0.01	0.00	49.82	7.98	6.94	0.02	0.01	85.73	11.09	10.89	0.01	0.00	16.36
27	6.30	5.95	0.03	0.01	102.60	9.07	12.81	0.02	0.01	110.53	12.18	20.29	0.00	0.00	11.80
40	7.01	8.97	0.05	0.01	177.13	10.00	19.01	0.02	0.01	60.79	12.97	28.93	0.00	0.00	2.39
johnson16-2-4															
23	6.50	5.28	0.15	0.04	4248.29	8.00	8.91	0.14	0.04	1943.33	8.00	8.91	0.14	0.04	2066.26
46	7.75	11.17	0.29	0.08	5660.29	8.00	12.21	0.22	0.06	3044.75	8.00	12.21	0.22	0.06	3290.52
69	8.00	14.23	0.28	0.08	3699.71	8.00	14.23	0.28	0.08	4227.76	8.00	14.23	0.28	0.08	3909.15
keller4															
28	6.98	6.89	0.28	0.07	$> 3h$	9.04	12.68	0.10	0.03	$> 3h$	11.00	18.75	0.02	0.01	3304.30
55	8.00	12.85	0.32	0.09	$> 3h$	11.00	26.98	0.02	0.01	4081.97	11.00	26.98	0.02	0.01	4173.34
83	9.00	19.82	0.14	0.04	$> 3h$	11.00	31.88	0.02	0.01	4827.99	11.00	31.88	0.02	0.01	5028.16

Table 4.3: Experimental results. For each graph, we use three different label set sizes and three different budgets, with randomly allocated labels, and show averages over 100 runs. In each case, we show the average size and cost of the result, the sequential runtime in seconds, the parallel runtime in seconds (2 cores, 4 threads) and then the "Enhanced" times reported by Carrabs, Cerulli, and Dell'Olmo (2014).

$\|L\|$	budget = 2			budget = 3			budget = 4		
	Size	Cost	Seq	Size	Cost	Seq	Size	Cost	Seq
Erdos971									
3	5.20	1.98	0.00	7.00	3.00	0.00			
4	4.61	1.86	0.00	5.71	2.78	0.00	7.00	3.98	0.00
5	4.24	1.94	0.00	5.10	2.81	0.00	6.05	3.91	0.00
Erdos972									
3	5.30	1.98	0.12	7.00	3.00	0.12			
4	4.84	1.90	0.12	5.84	2.88	0.12	7.00	3.97	0.12
5	4.35	1.80	0.12	5.18	2.76	0.12	6.04	3.82	0.12
Erdos981									
3	5.18	1.99	0.00	7.00	3.00	0.00			
4	4.68	1.91	0.00	5.78	2.88	0.00	7.00	3.99	0.00
5	4.18	1.82	0.00	5.21	2.89	0.00	6.10	3.82	0.00
Erdos982									
3	5.29	1.99	0.14	7.00	3.00	0.14			
4	4.80	1.85	0.14	5.95	2.95	0.14	7.00	3.96	0.14
5	4.26	1.69	0.14	5.19	2.82	0.14	6.19	3.88	0.14
Erdos991									
3	5.18	1.97	0.00	7.00	3.00	0.00			
4	4.64	1.89	0.00	5.84	2.91	0.00	7.00	3.98	0.00
5	4.23	1.93	0.00	5.19	2.82	0.00	6.16	3.86	0.00
Erdos992									
3	5.36	2.00	0.15	8.00	3.00	0.15			
4	4.92	1.92	0.15	6.06	2.98	0.15	8.00	3.99	0.15
5	4.27	1.84	0.15	5.32	2.86	0.15	6.38	3.88	0.15
Erdos02									
3	5.86	2.00	0.19	8.00	3.00	0.19			
4	5.04	1.99	0.19	6.43	2.98	0.19	8.00	3.99	0.19
5	4.66	1.82	0.19	5.69	2.83	0.19	6.82	3.91	0.19

Table 4.4: Experimental results on Erdős collaboration graphs. For each instance, we use three different label set sizes and three different budgets, with randomly allocated labels, and show averages over 100 runs. In each case, we show the average size and cost of the result, and the sequential runtime in seconds.

complications involved in parallel branch and bound, and the hardware providing only two "real" cores.

Large sparse graphs In Table 4.4 we present results using the Erdős collaboration graphs from the Pajek dataset by Vladimir Batagelj and Andrej Mrvar (Batagelj and Mrvar, 2006). We have chosen these datasets because of the potential "social network analysis" application suggested by Carrabs, Cerulli, and Dell'Olmo, where edge labels represent a particular kind of common interest, and we are looking for a clique using only a small number of interests.

For each instance we use 3, 4 and 5 labels, with a budget of 2, 3 and 4. The "3 labels, budget 4" cases are omitted, but we include the "3 labels, budget 3" and "4 labels, budget 4" cases—although the clique sizes are the same (and are equal to the size of a maximum unlabelled clique), we see in a few instances the costs do differ where the budget is 4. Again, we use randomly allocated labels and a sample size of 100.

Despite their size, none of these graphs are at all challenging for our algorithm, with average sequential runtimes all being under 0.2 seconds. However, no benefit at all is gained from parallelism—the runtimes are dominated by the cost of preprocessing and encoding the graph, not the search.

4.3 Maximum Balanced Induced Bicliques

Having looked at using Algorithm 2.1 to solve the k-clique problem by reduction, and to solve the maximum labelled clique problem by adding in side constraints, we now adapt it to find a subgraph of a different shape. We look at the maximum balanced induced biclique problem, which is in some ways a bipartite version of the maximum clique problem. We introduce a simple and effective symmetry elimination technique. We also discuss one particular class of graphs where the algorithm's bound is ineffective, and show how to detect this situation and fall back to a simpler but faster algorithm. We are not aware of any direct applications of the maximum balanced induced biclique problem; our interest is rather in seeing whether Algorithm 2.1 can reasonably be adapted to other problems.

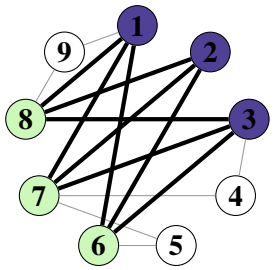


Figure 4.5: A graph, with its unique maximum balanced induced biclique of size six, $\{\{1,2,3\},\{6,7,8\}\}$, shown shaded in light and dark.

Let $G = (V, E)$ be a graph with vertex set V and edge set E. We say G is *bipartite* if its vertices may be partitioned into two disjoint independent sets. A *biclique*, or complete bipartite subgraph, is a pair of (possibly empty) disjoint subsets of vertices $\{A, B\}$ such that $\{a, b\} \in E$ for every $a \in A$ and $b \in B$. A biclique is *balanced* if $|A| = |B|$, and *induced*

if no two vertices in A are adjacent and no two vertices in B are adjacent. The maximum balanced induced biclique problem is to find a balanced induced biclique of maximum size in an arbitrary graph. We illustrate an example in Figure 4.5.

Finding such a maximum is NP-hard (Garey and Johnson, 1979, Problem GT24), both in bipartite and arbitrary graphs. A naïve exponential algorithm could simply enumerate every possible solution to find a maximum. Here we develop a branch and bound algorithm with symmetry elimination that substantially reduces the search space.

Recall that an *independent set* is the complement of a clique—that is, a set of vertices, no two of which are adjacent. A *clique cover* is a partition of the vertices in a graph into sets, each of which is a clique—this is the complement of a colouring. We introduce the symbol $\ddot{\omega}$ for the size (i.e. $|A| + |B|$) of a maximum balanced induced biclique, which is always even.

4.3.1 A Simple Branch and Bound Algorithm

A very simple branch and bound algorithm for the maximum induced biclique problem is given in Algorithm 4.3. The algorithm works by recursively building up two sets A and B such that $\{A, B\}$ is a biclique. At each stage, $remaining_a$ contains those vertices which may be added to A whilst keeping a feasible solution (i.e. each $v \in remaining_a$ is individually adjacent to every $b \in remaining_b$ and nonadjacent to every $a \in A$), and similarly $remaining_b$ contains vertices which may be added to B. Initially, A and B are both empty, and $remaining_a$ and $remaining_b$ both contain every vertex in the graph (line 4).

At each recursive call to `expand`, a vertex v is chosen from $remaining_a$ (line 8) and moved to be in A instead (lines 10 and 11). The algorithm then considers the implications of $v \in A$ (lines 12 to 16). A new $remaining'_a$ is constructed on line 12 by filtering from $remaining_a$ those vertices adjacent to v (since A must be an independent set), and a new $remaining'_b$ is constructed on line 13 by filtering from $remaining_b$ those vertices *not* adjacent to v (everything in B must be adjacent to everything in A).

Now if $remaining'_b$ is not empty, we may grow B further. Thus we repeat the process with a recursive call on line 16, swapping the roles of A and B—we are adding vertices to the two sides of the growing biclique in alternating order.

Having considered the possibility of $v \in A$, we then consider $v \notin A$ (line 17). The algorithm loops back to line 8, selecting a new v from $remaining_a$, until $remaining_a$ is empty. Finally, we backtrack by returning from the recursive call.

We keep track of the largest feasible solution $\{A^\star, B^\star\}$ that we have found so far. Initially the incumbent is empty (line 3). Whenever we find a potential solution, we compare it to the incumbent (line 14), and if our new solution is larger then the incumbent is unseated (line 14). Note that at this point, the balance condition must be checked explicitly, since either $|A| = |B|$, or $|A| = |B| + 1$ could be true.

Knowing the size of the incumbent allows us to avoid exploring some of the search

Algorithm 4.3: A simple, alternating branch and bound algorithm for the maximum balanced induced biclique problem.

```
1  simpleMaxBiclique :: (Graph G) → (Vertex Set, Vertex Set)
2  begin
3      (A*, B*) ← (∅, ∅)
4      expand(G, ∅, ∅, V(G), V(G), A*, B*)
5      return (A*, B*)

6  expand :: (Graph G, Set A, Set B, Set remaining_a, Set remaining_b, Set A*, Set B*)
7  begin
8      for v ∈ remaining_a do
9          if |remaining_a| + |A| > |A*| and |remaining_b| + |B| > |B*| then
10             A ← A + v
11             remaining_a ← remaining_a − v
12             remaining'_a ← remaining_a ∩ N̄(G, v)
13             remaining'_b ← remaining_b ∩ N(G, v)
14             if |A| = |B| and |A| > |A*| then (A*, B*) ← (A, B)
15             if remaining'_b ≠ ∅ then
16                 expand(G, B, A, remaining'_b, remaining'_a, B*, A*)
17             A ← A − v
```

space—this is the bound part of branch and bound. The condition on line 9 checks how much further we can grow A and B: if there are not enough vertices available to potentially unseat the incumbent, search at the current position can be abandoned. (This is not a very good bound, and is only for illustrative purposes. We discuss a more sophisticated bound below.)

4.3.2 Improving the Algorithm

We now adapt Algorithm 4.3 to incorporate symmetry elimination, an improved bound based upon clique covers, and an initial sort order. The end result is Algorithm 4.4. Like Algorithm 2.1, this algorithm is intended to be bit-parallel.

Symmetry elimination The search space for Algorithm 4.3 is larger than it should be: it explores legal *ordered* pairs (A, B) of vertex sets rather than unordered pairs $\{A, B\}$. Having explored every possible solution with $v \in A$, the search then considers $v \notin A$. But there is nothing to stop it from then considering a new $v' \in A$, and later placing $v \in B$. This is wasted effort, since if such a solution existed we would already have considered an equivalent with A and B reversed.

We may eliminate this symmetry as follows: if, at the top of search, we have considered every possibility with $v \in A$ then we may eliminate v from $remaining_b$ to avoid considering $v \in B$. The modified `expand` function in Algorithm 4.4 includes this rule: lines 22 to 23

Algorithm 4.4: An improved alternating branch and bound algorithm for the maximum balanced induced biclique problem.

```
1  improvedMaxBiclique :: (Graph G) → (Vertex Set, Vertex Set)
2  begin
3      (A*, B*) ← (∅, ∅)
4      permute G so that the vertices are in non-increasing degree order
5      expand(G, ∅, ∅, V(G), V(G), A*, B*)
6      return (A*, B*) (unpermuted)

7  expand :: (Graph G, Set A, Set B, Set remaining_a, Set remaining_b, Set A*, Set B*)
8  begin
9      (bounds, order) ← cliqueOrder(G, remaining_a)
10     for i ← |remaining_a| downto 1 do
11         if bounds[i] + |A| > |A*| and |remaining_b| + |B| > |B*| then
12             v ← order[i]
13             A ← A + v
14             remaining_a ← remaining_a − v
15             remaining'_a ← remaining_a ∩ N̄(G, v)
16             remaining'_b ← remaining_b ∩ N(G, v)
17             if |A| = |B| and |A| > |A*| then
18                 (A*, B*) ← (A, B)
19             if remaining'_b ≠ ∅ then
20                 expand(G, B, A, remaining'_b, remaining'_a, B*, A*)
21             A ← A − v
22             if B = ∅ then
23                 remaining_b ← remaining_b − v

24     cliqueOrder :: (Vertex Set remaining) → (Vertex Array, Int Array)
25     begin
26         (order, bounds) ← ([], [])
27         uncliqued ← remaining
28         clique ← 1
29         while uncliqued ≠ ∅ do
30             cliqueable ← uncliqued
31             while cliqueable ≠ ∅ do
32                 v ← the first vertex of cliqueable
33                 append v to order
34                 append clique to bounds
35                 uncliqued ← uncliqued − v
36                 cliqueable ← cliqueable ∩ N(G, v)
37             clique ← clique + 1
38         return (order, bounds)
```

remove symmetric solutions.

This technique may be seen as a special case of the standard lex symmetry elimination technique used in constraint programming (Crawford et al., 1996; Gent, Petrie, and Jean-François Puget, 2006). A constraint programmer would view A and B as binary strings, and impose the constraint $B \leq A$ (or the other way around—after all, the order of A and B is arbitrary). We are doing the same thing, by saying that if the first n bits of A are 0 then the first n bits of B must also be 0. Unlike adding a lex constraint, this approach does not interfere with the search order and does not introduce the risk of disrupting ordering heuristics (Gent, W. Harvey, and Kelsey, 2002). Additionally, this constraint always removes symmetric solutions from the search tree as early as possible (Backofen and Will, 2002).

Bounding We know that A and B must be independent sets, which are complements of cliques. We may therefore adapt the `colourOrder` bound from Algorithm 2.1 by replacing every concept with its complement, as follows. If we can cover a graph G using n cliques, we know that G cannot contain an independent set of size greater than n (since each element in an independent set must be in a different clique). This gives us a bound on $remaining_a$ which can be much better than simply considering $|remaining_a|$: we construct a greedy clique cover of the subgraph induced by $remaining_a$, and consider its size instead.

The `cliqueOrder` function in Algorithm 4.4, then, is simply the complement of Algorithm 2.1's `colourOrder`. The $bounds$ array it produces contains bounds on the size of a maximum independent set: the subgraph induced by vertices 1 to n of $order$ cannot have a maximum independent set of size greater than $bounds[n]$. The $order$ array contains the vertices of $remaining$ in some order, and is to be traversed from right to left, repeatedly removing the rightmost value for the choice branching vertex v.

To integrate this bound, we make the following changes: we begin by using `cliqueSort` to obtain the $bounds$ and $order$ variables (line 9). As previously, we explicitly iterate over $order$ from right to left (lines 10 and 12), rather than drawing v from $remaining_a$ arbitrarily. Finally, we make use of the bound on $remaining_a$, rather than using $|remaining_a|$ (line 11).

Vertex ordering We use a static ordering for constructing clique covers, so the initial order of vertices must also be considered—experiments show that, as for the maximum clique problem, a static non-increasing degree order fixed at the top of search is a good choice. We achieve this ordering by permuting the graph.

Detecting when the bound is useless Our bound considers how far A can grow, based upon what is in $remaining_a$, and how far B can grow based upon what is in $remaining_b$. If both $remaining_a$ and $remaining_b$ are independent sets, this does not help, and constructing the clique cover ordering is a substantial overhead. This situation occurs in particular if the

input is a bipartite graph, or close to one. We can at least detect when $remaining_a$ is an independent set: this happens precisely if $bounds[i] = i$ (assuming $bounds$ is 1-indexed), since if the graph contains at least two non-adjacent vertices then at least one such pair will be placed in the same clique (Batsyn et al., 2014, Proposition 2).

Ideally we would be able to switch to a better bound in the case that both $remaining_a$ and $remaining_b$ are (potentially overlapping) independent sets. However, we have been unable to find a better bound which is sufficiently cheap to compute to provide a benefit—approaches which reduce the search space but increase runtime include the use of degrees, indirect colouring, or the fact that finding an (unbalanced) induced biclique in a bipartite graph can be done in polynomial time via a matching algorithm. We may still decay to a version of the algorithm which includes symmetry elimination and uses cardinality bounds as in Algorithm 4.3—we do not demonstrate this technique in Algorithm 4.4, but it is simple to incorporate.

Parallel search Finally, we may adopt the parallel search strategy introduced in Chapter 3. We continue to prioritise stealing work created highest in the search tree. For the purposes of determining depth, we treat each recursive call equally, and do not do anything special to handle the alternating nature of A and B.

4.3.3 Computational Experiments

Since we are not aware of any useful applications of this problem, we return to the graphs from the Second DIMACS implementation challenge, which we introduced in Chapter 2. These experiments are performed on machines with dual Intel Xeon E5-2697A v4 processors and 512GBytes RAM, running Ubuntu Linux 16.04, and software was implemented in C++ and compiled using GCC 5.4.0. Results are presented in Table 4.5: only four instances could not be closed within four hours. Recall that these machines have 32 cores and hyper-threading, so parallel results use 64 threads.

For very easy instances, the parallel results are often worse than the sequential results, due to overheads. However, for harder instances, speedups between twenty and fifty are common. The amount of work done sequentially and in parallel is usually rather similar, which suggests either a high solution density, or very strong value-ordering heuristics, or a relatively weak bound function.

The final columns of the table show that the symmetry elimination technique is successful in reducing both runtimes and the size of the search space. In many instances the gain approaches 50% (this is expected: halving the number of solutions should not necessarily halve the size of the search space). In other cases the interaction of the bound and symmetry elimination reduces the benefit (sometimes to zero, when the bound can already eliminate symmetric solutions), but it is never a penalty.

Table 4.5: Experimental results for the maximum balanced induced biclique problem on the DIMACS clique graphs, sequentially and using 64 threads on a 32 core hyper-threaded system. The "no symmetry" results are expressed as a ratio of the sequential nodes and times.

Instance	$\ddot{\omega}$	Sequential		Parallel		No Symmetry	
		Nodes	Runtime	Nodes	Runtime	Nodes	Runtime
Randomly generated							
C125.9	8	920	0 ms	997	55 ms	1.495×	1.000×
C250.9	8	12448	7 ms	12697	73 ms	1.642×	1.429×
C500.9	10	107553	66 ms	190652	195 ms	1.456×	1.379×
C1000.9	10	7358668	5.5 s	7359667	341 ms	1.797×	1.588×
C2000.5	≥16	14513728903	>4.0 h	583381867522	>4.0 h	1.022×	>4.0 h
C2000.9	12	318154620	494.4 s	311435268	10.3 s	1.956×	1.781×
C4000.5	≥18	7965967412	>4.0 h	353042345893	>4.0 h	1.005×	>4.0 h
DSJC500_5	14	67503338	23.5 s	67662915	722 ms	1.780×	1.785×
DSJC1000_5	16	8883052113	1.4 h	8911483409	120.0 s	1.895×	1.856×
Randomly generated with large degree spread							
p_hat300-1	12	283383	73 ms	302062	219 ms	1.616×	1.027×
p_hat300-2	12	283126	75 ms	264981	117 ms	1.596×	1.560×
p_hat300-3	12	228584	87 ms	213655	57 ms	1.678×	1.655×
p_hat500-1	12	3924453	917 ms	3898725	219 ms	1.916×	1.797×
p_hat500-2	14	5881265	2.0 s	5792747	221 ms	1.444×	1.553×
p_hat500-3	12	6438257	3.0 s	6438756	244 ms	1.744×	1.742×
p_hat700-1	12	42615280	11.4 s	42624623	505 ms	1.747×	1.807×
p_hat700-2	14	35235961	17.0 s	36490628	708 ms	1.613×	1.606×
p_hat700-3	14	31214587	21.9 s	31190088	672 ms	1.715×	1.647×
p_hat1000-1	14	254937521	82.5 s	258899834	2.3 s	1.649×	1.706×
p_hat1000-2	16	364675172	208.0 s	354565372	5.1 s	1.415×	1.502×
p_hat1000-3	14	558151141	445.9 s	558145499	10.2 s	1.634×	1.555×
p_hat1500-1	16	5176905110	2941.1 s	5198892111	66.5 s	1.463×	1.532×
p_hat1500-2	16	6761389937	2.0 h	6790987829	142.8 s	1.532×	1.522×
p_hat1500-3	16	5460156144	2.3 h	5169287144	155.7 s	1.559×	1.561×
Randomly generated with large hidden solutions							
brock200_1	10	57931	16 ms	58130	23 ms	1.722×	1.625×
brock200_2	12	171207	39 ms	171406	210 ms	1.908×	1.795×
brock200_3	12	118038	31 ms	125090	92 ms	1.764×	1.677×
brock200_4	12	102484	27 ms	73252	78 ms	1.515×	1.481×
brock400_1	12	1774969	757 ms	1775368	201 ms	1.916×	1.694×
brock400_2	12	1760558	748 ms	1760957	160 ms	1.914×	1.878×
brock400_3	12	1812188	790 ms	1812587	168 ms	1.906×	1.657×
brock400_4	12	1772650	749 ms	1773049	165 ms	1.917×	1.844×
brock800_1	14	951765108	548.6 s	951756573	13.5 s	1.749×	1.731×
brock800_2	14	913607053	541.1 s	913622400	13.0 s	1.742×	1.685×
brock800_3	14	962767080	554.9 s	962785885	13.7 s	1.746×	1.723×

continued on next page...

Instance	$\ddot{\omega}$	Sequential		Parallel		No Symmetry	
		Nodes	Runtime	Nodes	Runtime	Nodes	Runtime
brock800_4	14	936273493	541.7 s	936275823	13.3 s	1.743×	1.720×
gen200_p0.9_44	10	2628	1 ms	2964	176 ms	1.758×	1.000×
gen200_p0.9_55	8	4201	1 ms	4400	109 ms	1.746×	2.000×
gen400_p0.9_55	16	8562	8 ms	8796	113 ms	1.825×	1.375×
gen400_p0.9_65	14	11709	9 ms	12018	105 ms	1.761×	1.444×
gen400_p0.9_75	12	16388	12 ms	18778	78 ms	1.708×	1.417×
san200_0.7_1	14	4330	2 ms	4529	67 ms	1.748×	1.500×
san200_0.7_2	24	1939	1 ms	2103	76 ms	2.180×	2.000×
san200_0.9_1	8	1850	1 ms	2049	80 ms	1.813×	1.000×
san200_0.9_2	8	3540	1 ms	3739	95 ms	1.847×	2.000×
san200_0.9_3	10	2085	1 ms	2284	57 ms	1.754×	1.000×
san400_0.5_1	62	1315	2 ms	1895	74 ms	4.953×	3.000×
san400_0.7_1	20	16229	18 ms	16628	119 ms	1.850×	1.611×
san400_0.7_2	28	7973	11 ms	8373	47 ms	2.210×	1.909×
san400_0.7_3	38	10361	11 ms	11015	116 ms	2.194×	2.091×
san400_0.9_1	10	19054	12 ms	20195	159 ms	1.788×	1.500×
san1000	134	10778	47 ms	13582	129 ms	4.019×	3.106×
sanr200_0.7	10	127080	32 ms	127279	82 ms	1.805×	1.688×
sanr200_0.9	8	4095	1 ms	4294	127 ms	1.739×	2.000×
sanr400_0.5	14	13683397	4.3 s	11980734	311 ms	1.691×	1.677×
sanr400_0.7	14	3370144	1.5 s	4389363	245 ms	1.572×	1.445×
Fault diagnosis							
c-fat200-1	2	214	0 ms	413	50 ms	1.117×	1.000×
c-fat200-2	2	353	0 ms	552	23 ms	1.187×	1.000×
c-fat200-5	2	927	1 ms	1126	56 ms	1.846×	2.000×
c-fat500-1	2	523	1 ms	1022	86 ms	1.069×	1.000×
c-fat500-2	2	619	1 ms	1118	172 ms	1.233×	1.000×
c-fat500-5	2	1398	3 ms	1897	65 ms	1.687×	1.667×
c-fat500-10	2	4219	12 ms	4718	180 ms	1.911×	2.000×
Coding theory							
hamming6-2	4	4	0 ms	67	32 ms	1.000×	1.000×
hamming6-4	14	1896	0 ms	1959	49 ms	2.006×	1.000×
hamming8-2	4	4	0 ms	259	73 ms	1.000×	1.000×
hamming8-4	32	303	0 ms	558	36 ms	1.000×	1.000×
hamming10-2	4	4	6 ms	4	22 ms	1.000×	1.000×
hamming10-4	40	45013477	118.5 s	45073409	2.6 s	1.832×	1.800×
johnson8-2-4	6	460	0 ms	487	47 ms	2.128×	1.000×
johnson8-4-4	10	211	0 ms	280	13 ms	1.621×	1.000×
johnson16-2-4	14	2216795	162 ms	2216914	91 ms	1.977×	1.957×
johnson32-2-4	≥30	93089451466	>4.0 h	3206339586495	>4.0 h	1.019×	>4.0 h

continued on next page...

Instance	$\ddot{\omega}$	Sequential		Parallel		No Symmetry	
		Nodes	Runtime	Nodes	Runtime	Nodes	Runtime
Keller conjecture							
keller4	18	82646	30 ms	82816	144 ms	1.731×	1.400×
keller5	32	3623257109	2374.9 s	3650000093	58.0 s	1.939×	1.928×
keller6	≥62	3102133022	>4.0 h	141963515329	>4.0 h	1.002×	>4.0 h
Steiner triple problem							
MANN_a9	6	32	0 ms	63	101 ms	1.125×	1.000×
MANN_a27	6	1407	1 ms	888	59 ms	1.064×	1.000×
MANN_a45	6	9852	15 ms	9294	166 ms	1.050×	1.000×
MANN_a81	6	53902	318 ms	49928	244 ms	1.025×	1.063×

Detecting when the bound is useless and decaying to a simpler algorithm provides a measurable benefit for several of the "p_hat" family of graphs and for "san1000", but does not generally make a substantial difference. On the other hand, for random bipartite graphs, this technique avoids a factor of five slowdown from the overhead of calculating a useless bound.

4.4 Conclusion

This chapter illustrated that Algorithm 2.1 is reasonably flexible: it can be adapted to handle a distance relaxation by reduction, a problem with side constraints and a multi-criteria objective, and a different kind of search pattern which includes an additional symmetry. In each case, the parallelism strategy introduced in Chapter 3 remained beneficial.

4.4.1 Maximum k-Cliques

The first part of this chapter shows for the first time that using a maximum clique algorithm to solve the maximum k-clique algorithm is feasible in practice. This is despite G^k being dense even if G is sparse—this ruled out the use of maximum clique algorithms which are designed for sparse graphs, and we were working with graphs with many more vertices than is typical for dense maximum clique algorithms.

For better performance on these graphs, we introduced a new lazy global domination rule. This was sometimes extremely beneficial, leading to exponential reductions in the search space—without this rule, we would have been unable to solve nine of the problem instances we considered, and many others would have taken much longer. However, even with laziness there is still sometimes a cost to pay when this rule does nothing. This rule is thus harmful (although only by a polynomial factor) for the graphs typically considered for the maximum clique problem, and we see the benefit of tailoring algorithms to the problem being solved. We suggest that a similar rule may also be useful for the maximum k-club problem.

We were able to use parallelism to close two further instances, although this required a much finer level of task granularity than usual. We suspect further progress could be made by tailoring the initial vertex ordering based upon what we know about the graphs, or by increasing the number of recursive calls per second by making the colouring stage cheaper, for example by reusing colour classes (Nikolaev, Batsyn, and San Segundo, 2015).

Quite often, we saw k-clique numbers and k-club numbers being the same. However, solving the maximum k-clique problem is much easier, both in terms of the algorithm and computationally. Thus it is worth checking whether the simpler model would be sufficient for practical applications before trying to solve the k-club problem.

In random graphs, we saw that $G(n,p)^k$ is easier than $G(n,p')$ with some higher probability p'. We also saw that as k increases, the problem gets easier—this was not typically the case for some of the real world graphs. It would be interesting to know why this is the case, and to gain a broader understanding of this behaviour in general.

Our results suggest that k is a very coarse grained parameter. We saw that often a 2-clique or 3-clique would cover the entire graph. In these circumstances the increased restrictions for k-club are of no benefit. It is not obvious if somehow allowing a "fractional" value of k could give more fine-grained control. Thus it may be worth considering other clique relaxations not based upon distance. However other models also have problems: a density-based relaxation known as quasi-clique, for example, can allow vertices with only a single edge to be added to a "clique" (Abello, Resende, and Sudarsky, 2002).

4.4.2 Maximum Labelled Cliques

Next, we saw that adapting Algorithm 2.1 to handle side constraints was both viable and much faster than a mathematical programming solution. This is not surprising. However, the extent of the performance difference was unexpected: we were able to solve multiple problems in under a tenth of one second that previously took over an hour, and we never took more than ten seconds to solve any of Carrabs, Cerulli, and Dell'Olmo's (2014) instances. We were also able to work with large sparse graphs without difficulty.

Of course, a more complicated mathematical programming model could close the performance gap. One possible route would be to treat colour classes as variables. But this would require a pre-processing step, and would lose the "ease of use" benefits of a mathematical programming approach. It is also not obvious how the label constraints would map to this kind of model, since equivalently coloured vertices are no longer equal.

On the other hand, adapting a dedicated maximum clique algorithm for this problem did not require major changes.

There is likely scope for improving the algorithm. For example, rather than doing two full passes, it is possible to start the second pass at the point where the last unseating of the incumbent occurred in the first pass. In the sequential case, this is conceptually simple

but messy to implement: viewing the recursive calls to `expand` as a tree, we could store the location whenever the incumbent is unseated. For the second pass, we could then skip portions of the search space "to the left" of this point. In parallel, this is much trickier: it is no longer the case that when a new incumbent is found, we have necessarily explored every subtree to the left of its position.

There are also variations on the problem which we could consider. In the formulation by Carrabs, Cerulli, and Dell'Olmo, each edge has exactly one label. What if instead edges may have multiple labels? If taking an edge requires paying for all of its labels, this is just a trivial modification to our algorithm. But if taking an edge requires selecting and paying for only one of its labels, it is not obvious what the best way to handle this would be. One possibility would be to branch on edges as well as on vertices (but only where none of the available edges matches a label which has already been selected).

This modification could be useful for real-world problems: for Carrabs, Cerulli, and Dell'Olmo's example where labels represent different relationship types in a social network graph, it is plausible that two people could both be members of the same club and be colleagues. Similarly, for the Erdős datasets, we could use labels either for different journals and conferences, or for different topic areas (combinatorics, graph theory, etc.). When looking for a clique of people using only a small number of different relationship types, it would make sense to allow only one of the relationships to count towards the cost. However, we suspect that this change could make the problem substantially more challenging.

4.4.3 Maximum Balanced Induced Bicliques

The third problem we considered involved a different shape of pattern. Although unlikely to be practically applicable, these results suggest that Algorithm 2.1's techniques can generalise to other graph-related problems involving heavily regular shapes of graph. For example, we have heard rumours that finding a maximum *pair* of cliques could be used to solve the problem of finding minimal generating sets for the monoid of all $n \times n$ Boolean matrices. We also showed a simple symmetry elimination technique, which could be useful for other problems.

However, attempting to adapt Algorithm 2.1 for fully general patterns is likely going too far; the next two chapters instead investigate fully general patterns using a different kind of algorithm.

Chapter 5

Subgraph Isomorphism Problems

The subgraph isomorphism family of problems involve "finding a copy of" a pattern graph inside a larger target graph; applications arise in bioinformatics (Bonnici et al., 2013), chemistry (Régin, 1995), computer vision (Damiand et al., 2011; Solnon et al., 2015), law enforcement (Coffman, Greenblatt, and Marcus, 2004), model checking (Sevegnani and Calder, 2015), malware detection (Bruschi, Martignoni, and Monga, 2006), compilers (Blindell et al., 2015; Murray, 2012; Murray and Franke, 2012), pattern recognition (Conte, Foggia, Sansone, et al., 2004; Foggia, Percannella, and Vento, 2014), program similarity comparison (Dalla Preda and Vidali, 2017), and graph databases (discussed in the following chapter). These problems have natural constraint programming models: we have a variable for each vertex in the pattern graph, with the vertices of the target graph being the domains. The exact constraints vary depending upon which variation of the problem we are studying (which we discuss in the following section), but generally there are rules about preserving adjacency, and an all-different constraint across all the variables to express injectivity. We illustrate the problem in Figure 5.1.

This constraint-based search approach dates back to works by McGregor (1979), McGregor (1982), and Ullmann (1976). Further improvements were introduced by Régin (1995), and in the LV (Larrosa and Valiente, 2002), ILF (Zampelli, Deville, and Solnon, 2010), LAD (Solnon, 2010), SND (Audemard, Lecoutre, et al., 2014), and PathLAD (Kotthoff, McCreesh, and Solnon, 2016) algorithms. The trend is towards "deep thinking": Régin introduced all-different propagation on vertices, LV additionally reasons about the size of neighbourhoods, ILF also reasons using the degrees of vertices in a neighbourhood, LAD

Figure 5.1: On the left, an induced subgraph isomorphism. On the right, a non-induced subgraph isomorphism: the extra dashed edge is not present in the pattern graph.

further adds all-different propagation on neighbourhoods, and SND and PathLAD additionally use paths to reason about non-adjacent vertices.

An alternative approach is used in the VF2 algorithm (Cordella et al., 2004) and variants (Carletti, 2016; Carletti, Foggia, and Vento, 2015). Although they still work by recursively building up an assignment of pattern to target vertices, these algorithms do not store or track domains. Instead, the basic VF2 algorithm works by computing and iterating over a set of candidate assignments, performing a number of feasibility checks, and then recursing with each assignment applied in turn until a solution is found. The candidate assignments at the top level of search consist of the lowest-numbered vertex in the pattern graph being assigned to each vertex in the target graph in numerical order. At subsequent levels, the set of candidate assignments is extended to include vertices adjacent to an already-made assignment (again, in input order). In other words, VF2 attempts to grow a connected solution by branching only on vertices adjacent to a choice already made. The feasibility checks consist of simple adjacency checks, together with a limited form of lookahead which checks the cardinalities of neighbourhoods.

VF2 often exhibits extremely poor behaviour on instances that other solvers find easy—we discuss this in more detail in the following chapter. However, VF2 and related approaches *do* have the advantage of being able to explore the state space very quickly, and have lower RAM requirements when working with large target graphs. Although much stronger in aggregate, LAD and SND sometimes make less than one recursive call per second with larger target graphs, and cannot always explore enough of the search space to find a solution in time. This motivates an alternative constraint programming approach, which is the main contribution of this chapter: on the same hardware, we will be making 10^4 to 10^6 recursive calls per core per second, whilst maintaining most of the strong reasoning properties of LAD and SND. The main features of this new algorithm are:

1. We represent graphs and domains as bitsets, and only use inference algorithms which can run very quickly with this representation.

2. We introduce a counting all-different propagator. This propagator is stronger than simple value elimination, but does not guarantee full generalised arc consistency.

3. Rather than calculating paths dynamically during search, as SND does, we introduce a concept we call *supplemental graphs*. Roughly, the idea is to create additional pattern / target graph pairs, in such a way that any subgraph isomorphism we find must additionally be a subgraph isomorphism between each supplemental pair. This approach is potentially weaker than SND's inference, in that supplemental graphs are calculated statically at the top of search, but they are also potentially more powerful, since we can perform degree-based reasoning on supplemental graphs too. For additional filtering power, we use path counts, rather than distances.

4. We refine existing variable-ordering heuristics with tie-breaking, and introduce value-ordering heuristics.

5. We use thread-parallel preprocessing and search, to make better use of modern multi-core hardware. As in Chapter 3, we use explicit, non-randomised work stealing to offset the difficulty of early value-ordering heuristic choices.

Although weaker propagation goes against the trend established by LV, ILF, LAD, and SND, and is contrary to the established wisdom for constraint programming in general (Bessière and Régin, 1996), here this approach usually pays off. In Section 5.3 we show that over a large collection of instances commonly used to compare subgraph isomorphism algorithms, our solver is the single best. Additionally, we verify that the parallelism we introduce is reproducible, risk-free, scalable, and beneficial.

Parts of this chapter have been published by McCreesh and Prosser (2015a) as "A Parallel, Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs", which used slightly different inference and search rules, and had a different parallel search implementation. The experiments reported in this chapter use a larger set of benchmark instances which were introduced in Kotthoff, McCreesh, and Solnon (2016), "Portfolios of Subgraph Isomorphism Algorithms".

5.1 Definitions, Notation, and a Proposition

In this chapter, our graphs are finite, undirected, and do not have multiple edges between pairs of vertices, but may have loops (an edge from a vertex to itself). A *non-induced subgraph isomorphism* is an injective mapping $i : P \rightarrowtail T$ from a graph P to a graph T which preserves adjacency—that is, if $v \sim_P w$ then we require $i(v) \sim_T i(w)$ (and thus if v has a loop, then $i(v)$ must have a loop). The *non-induced subgraph isomorphism problem* is to find such a mapping from a given pattern graph P to a given target graph T.

The *induced subgraph isomorphism problem* additionally requires that if $v \not\sim_P w$ then $i(v) \not\sim_T i(w)$; we use the notation $P \hookrightarrow T$ to refer to the induced variant. We focus primarily on the non-induced problem, since it more commonly appears in application-oriented papers. Variants of the problem also exist for labelled and directed graphs, where the mapping must preserve labels and the direction of edges respectively. We return to these three variations in Chapters 6 and 7.

The *neighbourhood degree sequence* of a vertex v in a graph G, denoted $\mathrm{S}(G, v)$, is the sequence consisting of the degrees of every neighbour of v, from largest to smallest. If R and S are sequences of integers, we write $R \preceq S$ if there exists a subsequence of S with length equal to that of R, such that each element in R is less than or equal to the corresponding

element in S. Observe that if a subgraph isomorphism $i : P \rightarrowtail T$ maps $i(v) = t$ then $\mathrm{S}(P, v) \preceq \mathrm{S}(T, i(v))$ must necessarily hold (Zampelli, Deville, and Solnon, 2010).

As in Chapter 4, we write G^d for the graph with vertex set $\mathrm{V}(G)$, and edges between v and w if the distance between v and w in G is at most d. We introduce the notation $G^{n,\ell}$ for the graph with vertex set $\mathrm{V}(G)$, and edges between vertices v and w (not necessarily distinct) precisely if there are at least n paths of length exactly ℓ between v and w in G. The following proposition may easily be verified by observing that subgraph isomorphisms preserve paths:

Proposition 5.1. Let $i : P \rightarrowtail T$ be a subgraph isomorphism. Then i is also

1. a subgraph isomorphism $i^d : P^d \rightarrowtail T^d$ for any $d \geq 1$, and
2. a subgraph isomorphism $i^{n,\ell} : P^{n,\ell} \rightarrowtail T^{n,\ell}$ for any $n, \ell \geq 1$.

The (contrapositive of the) first of these facts is used by SND, which dynamically performs distance-based filtering during search. We will instead use the second fact, at the top of search, to generate implied constraints; a similar approach has been adopted for PathLAD.

5.2 A New Algorithm

Algorithm 5.1 describes our approach. We begin (line 3) with a simple check that there are enough vertices in the pattern graph for an injective mapping to exist. We then (line 4) discard isolated vertices in the pattern graph—such vertices may be greedily assigned to any remaining target vertices after a solution is found. This reduces the number of variables which must be copied when branching. Next we construct the supplemental graphs (line 5) and initialise domains (line 6). We then (line 7) use a counting-based all-different propagator to reduce these domains further. Finally, we perform a backtracking search (line 7). Each of these steps is elaborated upon below.

5.2.1 Preprocessing and Initialisation

Following Proposition 5.1, in line 5 we construct a sequence of supplemental graph pairs from our given pattern and target graph. We will then search for a mapping which is simultaneously a mapping from each pattern graph in the sequence to its paired target graph—this gives us implied constraints, leading to additional filtering during search.

Our choice of supplemental graphs is derived experimentally, rather than in a principled manner. Filtering at a distance of 2 is clearly beneficial, and distances of greater than 3 rarely give additional filtering power. Choosing between distance 2 or distance 3 is more difficult: as we show below, there are instances where distance 3 supplemental graphs make the problem much easier, but enumerating paths of length 3 can be prohibitively expensive for large target

Algorithm 5.1: A non-induced subgraph isomorphism algorithm

```
1  nonInducedSubgraphIsomorphism (Graph 𝒫, Graph 𝒯) → Bool
2  begin
3      if |V(𝒫)| > |V(𝒯)| then return false
4      Discard isolated vertices in 𝒫
5      L ← [(𝒫, 𝒯), (𝒫^{1,2}, 𝒯^{1,2}), (𝒫^{2,2}, 𝒯^{2,2}), (𝒫^{3,2}, 𝒯^{3,2})]
6      D ← init(V(𝒫), V(𝒯), L)
7      return countingAllDifferent(D) ∧ search(L, D)

8  init (Vertices V, Vertices R, GraphPairs L) → Domains
9  begin
10     repeat
11         foreach v ∈ V do
12             D_v ← ⋂_{(P,T)∈L} { w ∈ R : v ~_P v ⇒ w ~_T w ∧ S(P, v) ⪯ S(T[R], w) }
13         R ← ⋃_{v∈V} D_v
14     until R is unchanged
15     return D

16 search (GraphPairs L, Domains D) → Boolean
17 begin
18     if D = ∅ then return true
19     D_v ← a domain in D with minimum size, tiebreaking on descending static degree in 𝒫
20     foreach v' ∈ D_v ordered by ascending static degree in 𝒯 do
21         D' ← clone(D)
22         if assign(L, D', v, v') ∧ search(L, D' − D_v) then return true
23     return false

24 assign (GraphPairs L, Domains D, Vertex v, Vertex v') → Boolean
25 begin
26     D_v ← {v'}
27     foreach D_w ∈ D − D_v do
28         D_w ← D_w − v'
29         foreach (P, T) ∈ L do
30             if v ~_P w then D_w ← D_w ∩ N(T, v')
31         if D_w = ∅ then return false
32     return countingAllDifferent(D)

33 countingAllDifferent (Domains D) → Boolean
34 begin
35     (H, A, n) ← (∅, ∅, 0)
36     foreach D_v ∈ D from smallest cardinality to largest do
37         D_v ← D_v \ H
38         (A, n) ← (A ∪ D_v, n + 1)
39         if D_v = ∅ ∨ |A| < n then return false
40         if |A| = n then (H, A, n) ← (H ∪ A, ∅, 0)
41     return true
```

graphs. As for the count, checking for path counts of at least one, two, and three appears to be a good tradeoff. Looking at up to three paths of length exactly two work reasonably well in general on the wide range of benchmark instances we consider, but there is room to improve the algorithm by better selection on an instance by instance basis (Kotthoff, McCreesh, and Solnon, 2016; Malitsky, 2014).

The `init` function is responsible for initialising domains. We have a variable for each vertex in the (original) pattern graph, with each domain being the vertices in the (original) target graph. It is easy to see that a vertex of degree d in the pattern graph P may only be mapped to a vertex in the target graph T of degree d or higher: this allows us to perform some initial filtering. As noted in the previous section, we may use compatibility of neighbourhood degree sequences for further filtering: v may only be mapped to w if $\mathrm{S}(P, v) \preceq \mathrm{S}(T, w)$ (Zampelli, Deville, and Solnon, 2010). Because any subgraph isomorphism $P \rightarrowtail T$ is also a subgraph isomorphism $F(P) \rightarrowtail F(T)$ for any of our supplemental graph constructions F, we may further restrict initial domains by considering only the intersection of filtered domains using each supplemental graph pair individually (line 12). At this stage, we also enforce the "loops must be mapped to loops" constraint.

Following this filtering, some target vertices may no longer appear in any domain, in which case R will be reduced on line 13. If this happens, we iteratively repeat the domain construction, but do not consider any target vertex no longer in R when calculating degree sequences. (Note that for performance reasons, we do not recompute supplemental graphs when this occurs.)

5.2.2 Search and Inference

The `search` function is our main recursive procedure. If every variable has already been assigned, we succeed (line 18). Otherwise, we pick a variable (line 19) to branch on by selecting the variable with smallest domain, tiebreaking on descending static degree only in the original pattern graph (we justify this in the following chapter). For each value in its domain in turn, ordered by ascending static degree in the target graph (also justified in the following chapter), we try assigning that value to the variable (line 22). If we do not detect a failure, we recurse (line 22).

For assignment and inference, the `assign` function gives the value v' to the domain D_v (line 26), and then infers which values may be eliminated from the remaining domains. Firstly, no other domain may now be given the value v' (line 28). Secondly, for each supplemental graph pair, any domain for a vertex adjacent to v may only be mapped to a vertex adjacent to v' (line 30). If any domain gives a wipeout, then we fail (line 31).

To enforce the all-different constraint, it suffices to remove the assigned value from every other domain, as we did in line 28. However, it is often possible to do better. If we can find a set of n variables whose domains include only n values between them, then those values may

be removed from the domains of any other variable. Such a set of variables is called a *Hall set* (Gent, Miguel, and Nightingale, 2008; Quimper and Walsh, 2005). We can also sometimes detect that an assignment is impossible even if values remain in each variable's domain: if we can find a set of n variables whose domains include strictly less than n values between them, then no solution exists. The canonical solution is to use Régin's (1994) matching-based propagator, which detects and filters every Hall set in polynomial time.

However, matching-based filtering is expensive and may do relatively little, particularly when domains are large, and the payoff may not always be economical. Various approaches to offsetting this cost while maintaining the filtering power have been considered (Gent, Miguel, and Nightingale, 2008). Since we are not maintaining arc consistency in general, we instead use an intermediate level of inference which is not guaranteed to identify every Hall set: this can be thought of as a heuristic towards the matching approach. This is described in the `countingAllDifferent` function.

The algorithm works by performing a linear pass over each domain in turn, from smallest cardinality to largest (line 36). The H variable contains the union of the values of every Hall set detected so far; initially it is empty. The A set accumulates the union of domains seen so far, and n contains the number of domains contributing to A. For each new domain we encounter, we eliminate any values present in previous Hall sets (line 37). We then add that domain's values to A and increment n (line 38). If we have too few values between the domains seen so far, we fail (line 39). If we detect a Hall set, we add its values to H, reset A and n, and keep going (line 40).

It is important to note that this approach may fail to identify some Hall sets, if the initial ordering of domains is imperfect. This propagator is not idempotent (Schulte and Stuckey, 2008): running the algorithm twice in succession could result in additional deletions. Nor is it monotonic (Schulte and Tack, 2009): removing additional values from domains before running the propagator can weaken its effects. The correctness of Algorithm 5.1 as a whole relies upon the single value propagation in line 28, and `countingAllDifferent` is used only as an additional step which might remove some infeasible values.

Although it lacks the theoretical benefits of stronger propagators, this approach runs very quickly in practice: the sorting step is $\mathcal{O}(v \log v)$ (where v is the number of remaining variables), and the loop has complexity $\mathcal{O}(vd)$ (where d is the cost of a bitset operation over a target domain, which we discuss below). We validate this trade-off experimentally in the following section.

We have been unable to find this `countingAllDifferent` propagator described elsewhere in the literature, although a sort- and counting-based approach has been used to achieve bounds consistency (Jean-Francois Puget, 1998)—but subgraph isomorphism domains are not naturally ordered—and as a preprocessing step (Quimper and Walsh, 2005). Bitsets, which we discuss below, have also been used to implement the matching part of the

generalised arc-consistency algorithm (Kessel and Quimper, 2012).

5.2.3 Bit-Parallelism

We saw in Chapter 2 that bitsets are used in many maximum clique algorithms. They are equally beneficial in a subgraph isomorphism context, where their use dates back to at least Ullmann (1976), and remains an active area of research (San Segundo, Rodríguez-Losada, Galán, et al., 2007; Ullmann, 2010). We use bitsets here too: graphs are stored as arrays of bit vectors, domains are stored as bit vectors, the neighbourhood intersection in line 30 is a bitwise-and operation, the unions on lines 38 and 40 are bitwise-or operations, and the cardinality check in line 40 is a population count (recall from Section 1.6.1 that this is a dedicated hardware instruction in modern CPUs).

5.2.4 Thread-Parallel Search

As in Chapter 3, we often spend a lot of time searching for solutions. We therefore parallelise the search function, exploring different assignments using multiple threads. The mechanism we use is similar to the resplitting mechanism introduced and evaluated in Chapter 3. However, for the maximum clique problem, the extremely high number of recursive calls per second and the need to avoid copying meant that publishing positions and recomputing parts of the search space was necessary. A slightly different implementation is simpler for this algorithm: we run one thread following the sequential search path, and allow other threads to pre-compute future iterations of *any* of the in-progress **foreach** loops from line 20. We continue to prioritise parallelising the top level of the search tree, and then the second level of the search tree (tiebreaking on left-to-right ordering), and so on, limiting ourselves to splitting to depth five. Our experiments will confirm that the highest-first splitting strategy is beneficial for subgraph isomorphism for the same reasons that it was for the maximum clique problem.

In this chapter we are dealing with a decision problem, not an optimisation problem, so there is no bound function or incumbent—however, we can use an atomic shared Boolean variable as the "incumbent", taking the value false until a solution is found, and if this variable is set to true, every thread may terminate immediately. Thus we continue to provide both the theoretical and empirical performance guarantees proposed in Section 1.6.6: this approach will not introduce exponential slowdowns either when going from sequential to parallel or when increasing the number of threads used, and will be reproducible.

5.2.5 Thread-Parallel Preprocessing

Although search sometimes takes a long time (and is therefore worth parallelising), for some instances nearly all of the time is spent in initialisation. To speed this step up, there are two

further opportunities for parallelism. Firstly, we may parallelise the outer **for** loops involved in calculating neighbourhood degree sequences and in initialising the domains of variables. This step is entirely routine. Secondly, constructing each supplemental graph involves an outer **for** loop, iterating over each vertex in the input graph. These loops may also be parallelised, with one caveat: we must be able to add edges to (but not remove edges from) the output graph safely, in parallel. This may be done using an atomic "or" operation, since we are only ever enabling additional bits, never disabling them.

5.3 Experimental Evaluation

We now evaluate a C++ implementation of Algorithm 5.1 on machines with dual Intel Xeon E5-2640 v2 processors (for a total of 16 cores, and 32 hardware threads via hyper-threading) and 64GBytes RAM, running Ubuntu Linux 14.04. We compiled using GCC 5.3.0, and for parallelism we use C++11 native threads.

We consider a large benchmark set of 5,725 instances used in a recent evaluation of algorithm portfolios for subgraph isomorphism (Kotthoff, McCreesh, and Solnon, 2016). This is a superset of the instances used to evaluate LAD (Solnon, 2010), SND (Audemard, Lecoutre, et al., 2014), and the published variant of this algorithm (McCreesh and Prosser, 2015b). All these instances are available in a simple text format (Solnon, 2016). The instances come from a range of sources:

- A database containing various kinds of graph gathered by Larrosa and Valiente (2002) from the Stanford Graph Database. The instances are divided into two sets. The first contains small instances generated from the first 50 graphs of the database; the second, larger instances with pattern graphs from the first 50 graphs of the database and target graphs from the next 50 graphs. Readers of the deluxe colour edition of this thesis will see instances from the first set plotted in purple in the scatter plots below, and those from the second set are in blue; satisfiable instances are shown as circles, and unsatisfiable instances as crosses.

- Randomly generated instances. One family, introduced by Zampelli, Deville, and Solnon (2010), comes from scale-free graphs. The remainder use bounded degree, regular mesh, and uniform random models (Cordella et al., 2004). All of the latter instances are satisfiable, and we spend the first half of Chapter 6 discussing this dataset's flaws. Thse instances are plotted in green.

- Randomly generated instances close to the phase transition, which we introduce in the following chapter. These instances are plotted in black.

- Instances representing segmented images (Damiand et al., 2011; Solnon et al., 2015), and models of 3D objects (Damiand et al., 2011). These instances are plotted in red and orange respectively.

These instances come in a wide range of orders, sizes, and densities, although sparse graphs are most common. The pattern graphs with the largest order and size have 900 vertices and 24,820 edges respectively, and target graphs go up to 6,671 vertices and 418,000 edges.

5.3.1 Comparison with Other Solvers

We begin by comparing our implementation of Algorithm 5.1 to Solnon's C implementations of LAD (Solnon, 2010) and PathLAD (Kotthoff, McCreesh, and Solnon, 2016), and the VFLib C implementation of VF2 (Cordella et al., 2004). (The versions of each of these solvers we used could support loops in graphs correctly—for VF2 this is handled by labelling.)

In Figure 5.2 we show the cumulative performance of each algorithm. The value of the line at a given time for an algorithm shows the total number of instances which, individually, were solved in at most that amount of time. Algorithm 5.1 is the single strongest solver for timeouts of over 0.2 seconds, but is weakest with timeouts below the twenty millisecond range. This is largely due to the cost of producing supplemental graphs at the top of search, which incurs overhead even on trivial instances.

We also give an instance-by-instance scatter plot in Figure 5.3, comparing with VF2 (left) and PathLAD (right). VF2 often performs better on instances which both algorithms find easy, but there are many instances which Algorithm 5.1 finds easy which VF2 either finds difficult or cannot solve at all. There is only one instance which VF2 can solve within the timeout which we cannot, and two more where VF2's performance is much better than ours. All three instances are satisfiable, and so from Chapter 3, we might guess that these instances are due to an early value-ordering mistake made by Algorithm 5.1—indeed, we see below that when using parallel search to offset early heuristic choices, our solver achieves strong superlinear speedups on all three of these instances. In contrast, Algorithm 5.1's performance when compared to PathLAD is more mixed: although it is stronger overall, there are many instances that PathLAD finds easy which it does not, and vice-versa.

For a comparison against SND, we refer to McCreesh and Prosser (2015b), which shows that SND has extremely high startup costs but beats LAD (but not our algorithm) for runtimes of over one hour. The implementation by Audemard, Lecoutre, et al. (2014) requires graphs to be preprocessed and converted to a solver-specific format, and so we were unable to run it on the full set of 5,725 instances.

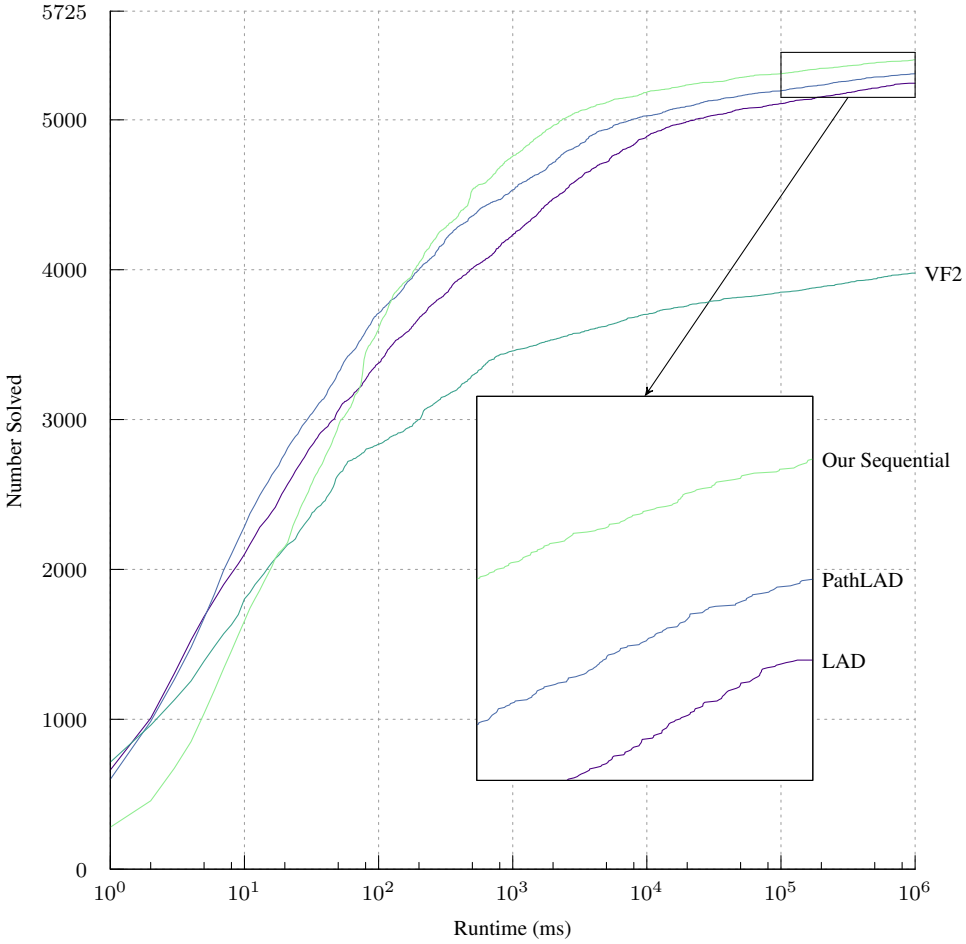


Figure 5.2: Cumulative number of benchmark instances solved within a given time, for a sequential Algorithm 5.1, versus other solvers.

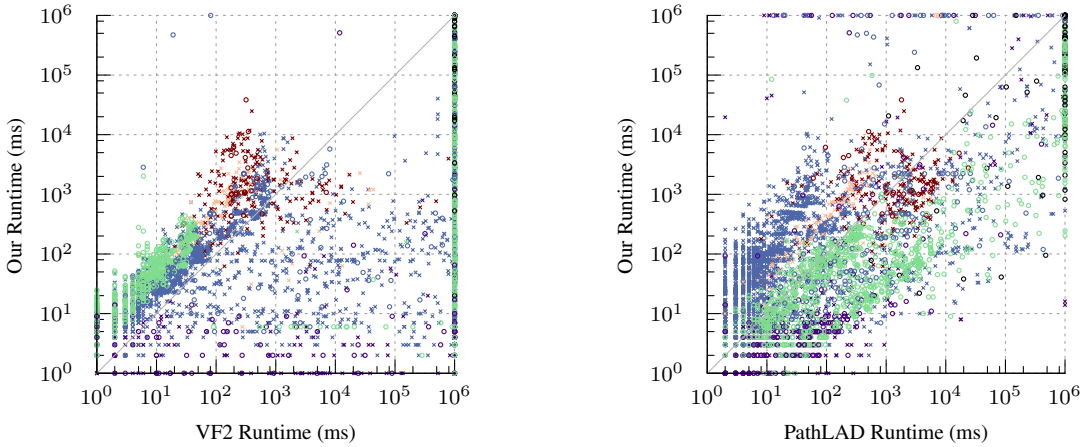


Figure 5.3: An instance by instance comparison of sequential Algorithm 5.1 to VF2 (left) and PathLAD (right).

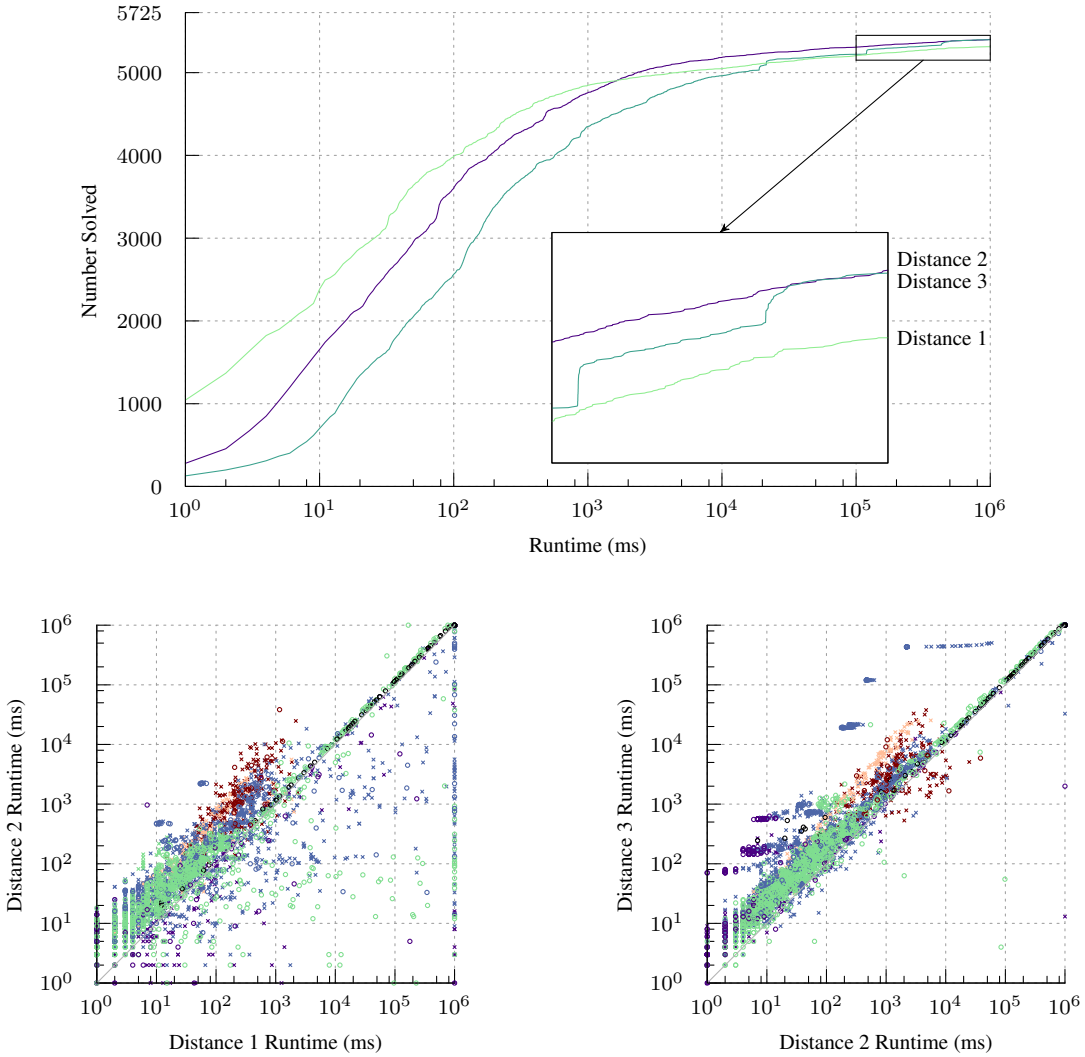


Figure 5.4: The effects of using different distances for filtering in Algorithm 5.1. On top, cumulative number of instances solved, and below, the difference between simple adjacency and distance 2 filtering (left), and distance 2 and distance 3 filtering (right).

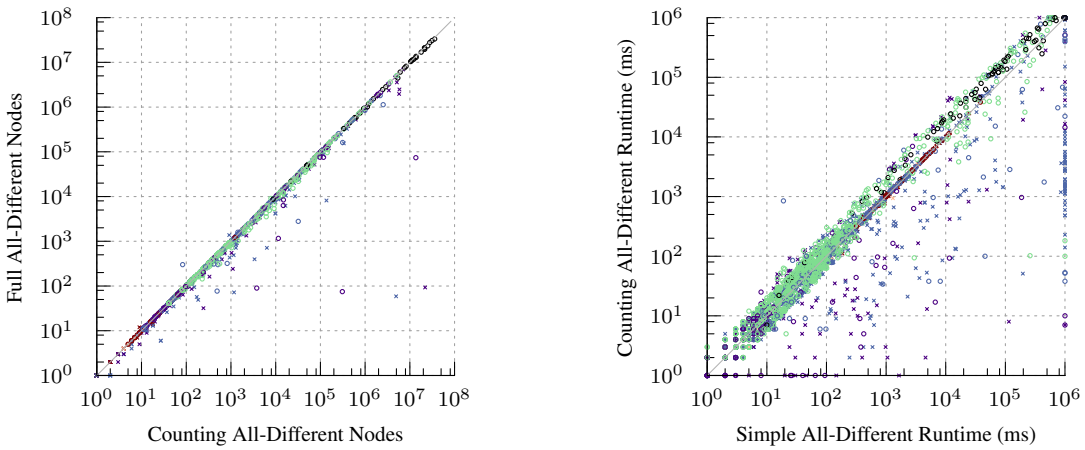


Figure 5.5: On the left, how much smaller is the search space when using a global arc consistent all-different propagator? On the right, the benefits of using a counting all-different propagator over simple value elimination.

5.3.2 Algorithm Design Choices

Having seen that Algorithm 5.1 is a strong algorithm overall, we now look in more detail at some of the choices made during its design. For example, why do we opt for looking at paths of length 2, but not length 3? Figure 5.4 justifies this decision: the cumulative plot shows filtering using paths of length 2 is better than using only adjacency (at least if we expect the runtime to be more than a few seconds). However, using paths of length 3 incurs considerable overheads for little gain—although with sufficiently long timeouts, using longer paths does eventually pay off.

The scatter plots present an alternative perspective: there are many instances which become much easier with distance two filtering, whilst the overheads are rarely more than an order of magnitude, and are much less when looking at longer runtimes. Meanwhile, using distance three makes far fewer instances much easier, and can incur extreme costs in some cases. Note that most of the instances where distance three filtering is much worse are the large real-world graphs—this motivates further experiments with per-instance algorithm selection (Kotthoff, McCreesh, and Solnon, 2016).

In Figure 5.5 we justify our use of the counting all-different propagator. In the left-hand plot we show the benefits to the size of the search space that would be gained if we used Régin's algorithm at every step instead of our counting propagator (we use a longer runtime of 10,000 seconds for the stronger propagator, and plot only instances solved by both algorithms). The results show that even if we were able to maintain generalised arc consistency for no greater cost than the counting propagator, substantial improvements would be rare. (The two cases where stronger filtering makes matters worse is due to dynamic variable ordering heuristics.)

Thus, either our counting propagator is nearly always as good as maintaining arc consistency, or neither propagator does very much at all in this setting. To show that the former is true, in the right-hand plot of Figure 5.5 we show the benefits to runtime that are gained from using counting, rather than simply deleting a value from every other domain on assignment. The large number of points below the diagonal line confirm that going beyond simple value deletion for all-different propagation is worthwhile, whilst (again excluding one case where dynamic variable ordering heuristics complicate matters) the constant overhead of this bit-parallel propagator is less than a factor of two.

5.3.3 Thread Parallelism

Now we evaluate the benefits of thread parallelism. In Figure 5.6 we show the cumulative and per-instance benefits of introducing thread parallelism by comparing results using 32 threads (recall that we work with 16 core, hyper-threaded machines) to sequential results. We show results both for distance 2 filtering, and distance 3 filtering. In both cases, the cumulative plots

show that the parallel results are clearly and substantially better than the sequential results.

The left-hand highlight box in the cumulative plot zooms in on a sudden jump in the distance 3 results, which in parallel occurs both to the left of (i.e. sooner) and above (i.e. with a higher number of instances solved) where it is in the sequential curve. The jump occurs because the Cordella et al. (2004) and Larrosa and Valiente (2002) datasets include several groups of instances all with the same large number of target vertices, many of which are trivial once the supplemental graphs have been constructed. This indirectly demonstrates the benefits of parallelising the preprocessing as well as search: without parallel supplemental graph construction, we would get no speedup at all on these instances, and this jump would not shift to the left. Similar effects do occur when only using distance 2 reasoning, but they are less pronounced.

The scatter plots reveal more interesting behaviour. Superlinear speedups are common, and there are many instances in both cases where the parallel implementation finds a solution almost immediately whilst the sequential implementation times out. As our theoretical guarantees predict, we never introduce an exponential slowdown. For some satisfiable instances parallel search provides little or no benefit, and we see low speedups because of wasted work and overheads—in a very small number of cases these lead to a small absolute slowdown. This does not go against our theoretical model: remember that we only have sixteen "real" cores, that parallelism decreases the amount of memory bandwidth available to each core, and that we must modify the sequential code slightly to allow for parallel search. We observe occasional poor speedups on unsatisfiable instances, due to work balance problems, although usually unsatisfiable instances approach a linear speedup as we would expect (an unsatisfiable instance for a decision problem behaves like a branch and bound instance where an optimal solution is found instantly). Interestingly, we observe a similar roughly-linear speedup for many *satisfiable* instances. For these instances, we are performing a lot of work which we also carry out in the sequential run before a solution is eventually found.

In Figure 5.7 we compare parallel Algorithm 5.1 to sequential VF2 and sequential PathLAD. This is not necessarily a fair comparison if we are simply interested in which algorithm runs fastest (although one could argue that VF2 and PathLAD should be penalised for not being able to make use of the multiple cores available in all modern hardware). The main point of these plots is to see a change in behaviour in comparison to Figure 5.3. Recall that there were three satisfiable instances which VF2 found relatively easy and which our sequential solver found hard. All three are gone in this plot, confirming our suspicions that poor value-ordering choices were to blame. In contrast, PathLAD remains competitive for some instances, suggesting that much stronger reasoning is sometimes a better choice.

What about the scalability and reproducibility properties that we claim to guarantee? In Figure 5.8 we show what happens going from sequential to four threads, from four threads

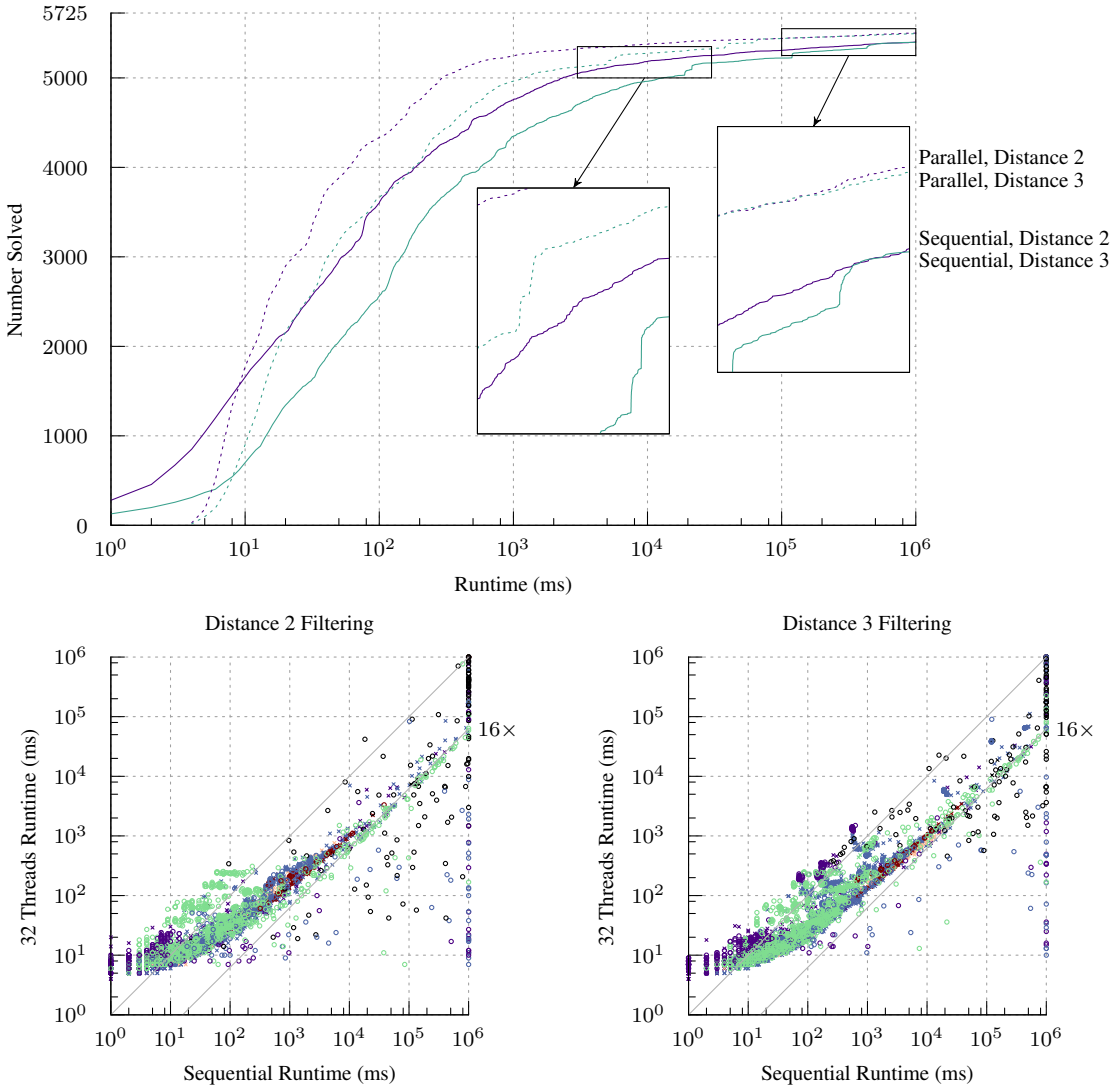


Figure 5.6: Above, cumulative number of benchmark instances solved within a given time, comparing sequential and parallel Algorithm 5.1 implementations. Below, per-instance comparisons. Results use 32 threads on a 16 core hyper-threaded system.

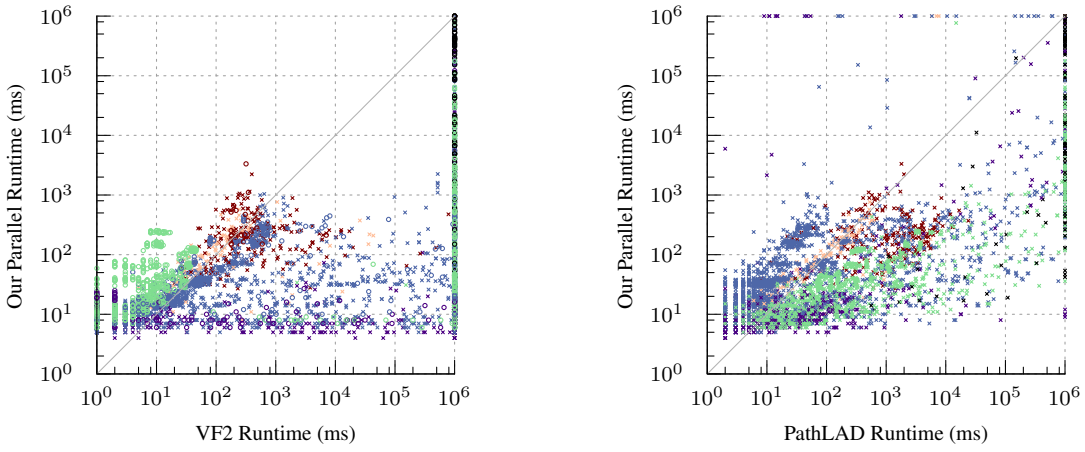


Figure 5.7: Comparing parallel Algorithm 5.1 with sequential VF2 and sequential PathLAD.

to eight, from eight threads to sixteen, and from sixteen threads to thirty two. In each case we see new instances exhibiting superlinear speedups; it is worth emphasising that once we have a superlinear speedup, increasing the number of threads never makes this go away. For the first three scatter plots we also see a common improvement for unsatisfiable instances, showing that increasing the number of threads increases the rate at which work is done.

Going from sixteen threads to thirty two is more erratic: we see an increase in superlinear speedups, but also an increase in overheads. This is not surprising, since when hyper-threading, each core in effect runs more slowly, in return for a little extra total processing power. The final scatter plot shows what happens when we oversubscribe even further, and run sixty four threads. Again we see new superlinear speedups, but we also see constant-factor slowdowns on many instances. Interestingly, the cumulative plot shows that oversubscribing is slightly better overall: although there are overheads, if we are prepared to risk a constant factor slowdown on some instances, then the increase in superlinear speedups from the additional diversity given by the extra threads is beneficial in aggregate. These plots also suggest that we are not approaching a scalability limit, and that if we had more cores available, we would be able to make good use of them.

Reassuringly, in both Figures 5.6 and 5.8, all the superlinear speedups are on satisfiable instances (which are plotted using a circle, rather than a cross). In Chapter 3 we saw for the maximum clique problem that superlinear speedups were common due to a combination of value-ordering heuristics being poor at the top of search, and a work-splitting mechanism which explicitly prioritised branching at the top level of search first. The same effect causes the superlinear speedups here. We might anticipate this: we are using degree as a value-ordering heuristic (and we return to this issue in the following chapter), and many target graphs do not exhibit a particularly broad degree spread. Thus, even if picking by degree were a totally reliable rule (which it is not), we often have many vertices of the same degree and no way to choose between them, and so diversifying remains important.

Finally, in Figure 5.9, we show that our reproducibility guarantee holds in practice, by plotting repeated runs with sixteen and thirty two threads. The results are not quite as clean as those for a repeated sequential run (which we show in the first plot), but in no cases do we see more than a small difference in runtimes, and we never see exponential changes, despite superlinear speedups being common. By comparison, if we use Intel Cilk Plus for randomised work-stealing (which we show in the final plot), on satisfiable instances we sometimes see several orders of magnitude difference in runtimes.

5.4 Other Problem Variants

So far we have only discussed the non-induced variant of the problem on unlabelled, undirected graphs. To handle the induced problem, it suffices to notice that an induced subgraph

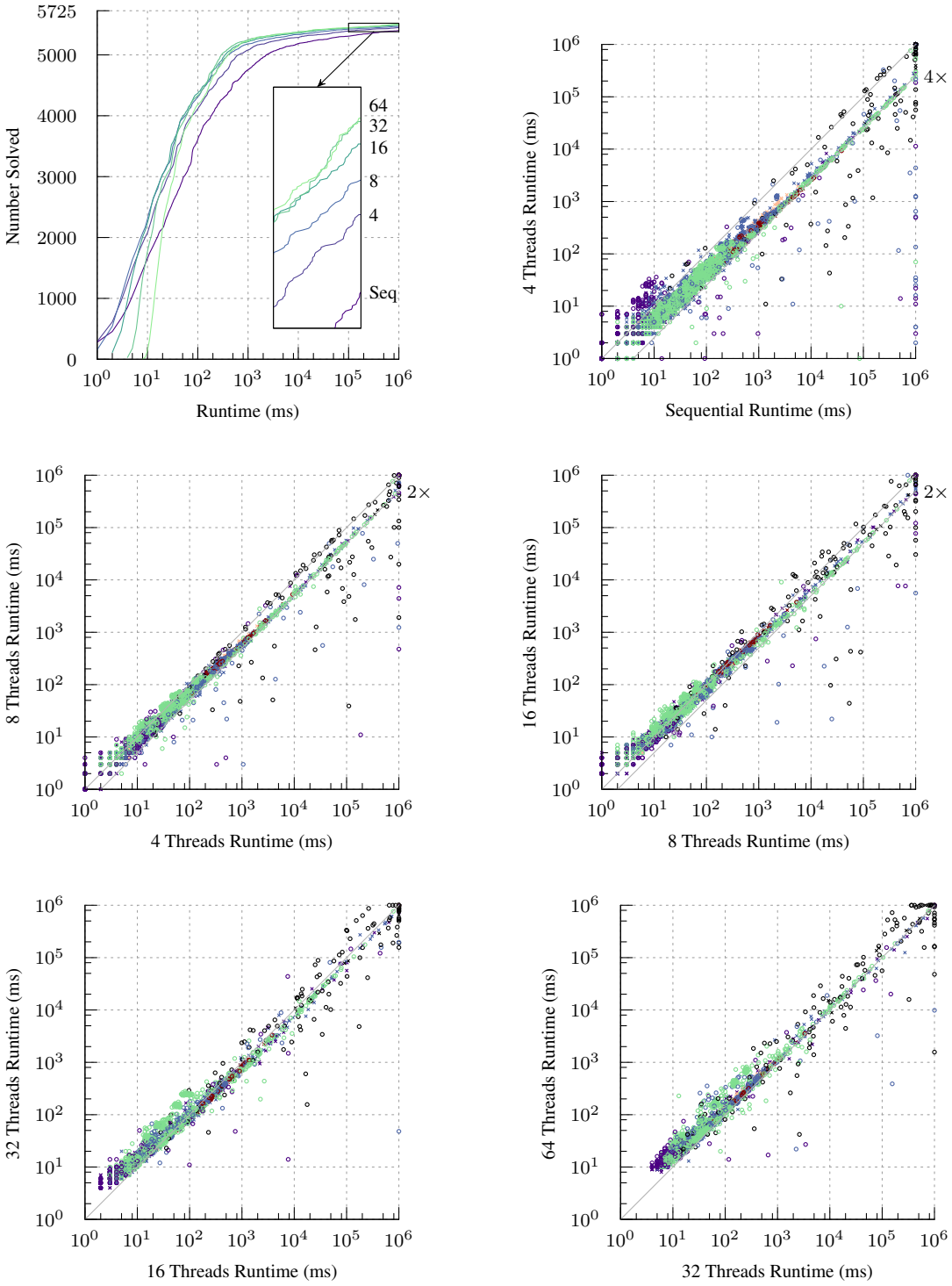


Figure 5.8: The top left plot shows the cumulative number of instances solved over time, with varying numbers of threads, on a 16 core hyper-threaded system. The scatter plots show the effects of going from sequential to parallel, and then increasing the number of threads used.

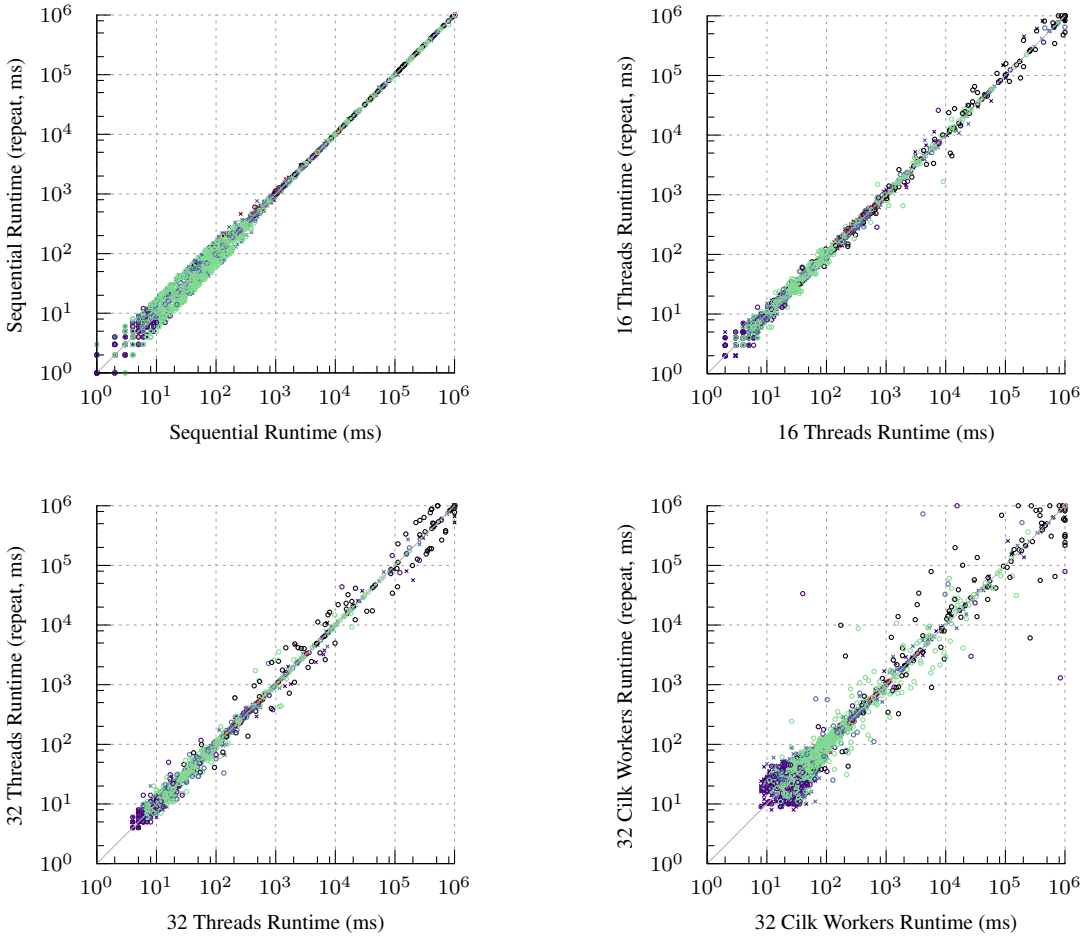


Figure 5.9: Verifying that parallel runtimes are reproducible: Algorithm 5.1 runtimes, plotted against a repeat run with the same parameters, using 16 and 32 threads respectively. For comparison, the first plot shows repeated sequential runs, and the last plot uses Intel Cilk Plus with 32 threads for work allocation.

isomorphism is a non-induced subgraph isomorphism which is also a non-induced subgraph isomorphism between the loop complements of the pattern and target graphs. Thus, we may modify Algorithm 5.1 simply by adding an additional supplemental graph pair for the loop complements, and removing the "strip isolated vertices" rule. We look more at induced isomorphisms in the remainder of this thesis.

In the following two chapters we will also be looking at labelled graphs, where vertices and / or edges must be mapped to equally labelled vertices and / or edges. For vertices this is a simple domain reduction operation at the top of search (although there is room to improve neighbourhood degree sequence filtering to make more use of labels). Edge labels can be handled by propagation, but we discuss better models in Chapter 7. Handling directed edges is also straightforward.

Richer labelling schemes also exist. For example, temporal subgraph isomorphism (Redmond and Cunningham, 2013; Redmond and Cunningham, 2016) has edges labelled with times, and the problem can be to find a pattern, all of whose edge times occur within a

certain duration or period.

Other variants of the subgraph isomorphism problem alter the difference constraints. For example, the subgraph epimorphism problem (Gay et al., 2014) is to find a surjective mapping rather than an injective one, whilst the subgraph homomorphism problem drops the injectivity requirement. Graph isomorphism (where the pattern and the target must have the same number of vertices and edges) is typically handled differently to exploit much stronger invariants, by using a repeated label refinement scheme (McKay and Piperno, 2014). The adjacency rule also has other flavours: sometimes the rule is that the preimage of an edge must be an edge, and sometimes adjacent vertices may be mapped to the same target vertex (even if the target vertex has no loop). Each of these problems allows for a similar algorithm, although the isolated vertex, neighbourhood degree sequence, and supplemental graph rules must be modified. Finally, maximum common subgraph problems, which we discuss in Chapter 7, can be thought of as relaxations of subgraph isomorphism problems.

5.5 Conclusion

Although the trend in subgraph isomorphism algorithms has been towards stronger and stronger reasoning, we have seen that cheap surrogates for all-different and distance-based filtering can give an even more effective algorithm. We also saw the importance of not going too far in the quest for simplicity—disabling distance filtering or all-different propagation entirely made our algorithm considerably worse. This balance between speed and simplicity is investigated in more detail in the following chapter.

We introduced parallelism for both preprocessing and search, and showed that it is risk-free, scalable, reproducible and beneficial—as in Chapter 3, controlling the interaction between work-splitting and value-ordering heuristics was key to this success. Recall that Malapert, Régin, and Rezgui (2016) said that they "ignore the problem of finding a first feasible solution because the parallel speedup can be completely uncorrelated to the number of workers, making the results hard to analyze"; with the performance guarantees that our approach to work splitting offer, we do not have any trouble analysing our results, or seeing that they are favourable.

There is plenty of scope for extensions of and improvements to our algorithm. For example, per-instance algorithm selection works well in this setting (Kotthoff, McCreesh, and Solnon, 2016), and it is likely that special domain- or instance-specific supplemental graphs could improve things further. It is also possible to make supplemental graphs and neighbourhood degree sequence filtering stronger in labelled, induced, and directed settings.

We also believe that our algorithm could benefit from some form of conflict analysis. In McCreesh and Prosser (2015a) we showed that conflict-directed backjumping was sometimes extremely beneficial in a similar algorithm, and that it could be parallelised whilst maintaining

performance guarantees by treating the **for** loop as a parallel fold with left zero (Lobachev, 2012). Preliminary experiments suggest that clause-learning could be even stronger. However, the all-different propagator often leads to overly-specific learned clauses compared to what a human would produce in the same situation, and these clauses tend not to be reusable as a result. It is also not clear how learned clauses could be integrated into the sequential algorithm without introducing huge overheads (the standard two watched literals scheme still requires iterating over every value removed from a domain to trigger any watches, whereas the current algorithm does nothing on value removals unless a domain wipeout occurs). If clause-learning can be made to work efficiently in this setting, it also paves the way for *subgraph modulo theories* problems and hybrid solver strategies—we consider this to be a very promising avenue for future research.

In the following chapter we take a deeper look at why both Algorithm 5.1 and LAD are so successful: although subgraph isomorphism is NP-complete, we have been working with graphs with thousands of vertices without too much difficulty. As we might suspect from Chapter 2, there are in fact small, "really hard" subgraph isomorphism instances which we cannot solve; understanding these instances also justifies our choices of variable- and value-ordering heuristics. Then, as part of Chapter 7, we investigate whether Algorithm 5.1 can be adapted for maximum common subgraph problems.

Chapter 6

When is Subgraph Isomorphism Really Hard?

Although subgraph isomorphism is NP-complete, the algorithms we introduced and compared in the previous chapter were working comfortably with instances with many hundreds of vertices in the pattern graph, and over six thousand vertices in the target graph. However, subgraph isomorphism algorithms cannot handle *arbitrary* instances involving this many vertices. Experimental evaluations of subgraph isomorphism algorithms (including our own in the previous chapter) are usually performed using a mix of real-world graphs, graphs that encode biochemistry and computer vision problems, and randomly generated graph pairs. Using random instances to evaluate algorithm behaviour can be beneficial, because it provides a way of generating many instances cheaply, and reduces the risk of over-fitting when tuning design parameters. The random instances used typically come from common datasets (De Santo et al., 2003; Zampelli, Deville, and Solnon, 2010), which were generated by taking a random subgraph of a random (Erdős-Rényi, scale-free, bounded degree, or mesh) graphup to and permuting the vertices. Such instances are guaranteed to be satisfiable—Anton and Olson (2009) exploited this property to create large sets of random satisfiable Boolean satisfiability instances. This is the most common approach to generating random subgraph isomorphism instances, meaning existing benchmark suites contain relatively few non-trivial unsatisfiable instances (although a few of the patterns in the instances by Zampelli, Deville, and Solnon have had extra edges added, making them unsatisfiable). Also, the satisfiable instances tend to be computationally fairly easy, with most of the difficulty being in dealing with the size of the model. This has lead to bias in algorithm design, to the extent that some proposed techniques, such as those of Battiti and Mascia (2007), will *only* work on satisfiable instances.

The first contribution of this chapter is to present and evaluate new methods for creating random pattern / target pairs. The method we introduce generates both satisfiable and unsatisfiable instances, and can produce computationally challenging instances with only a few tens of vertices in the pattern, and 150 vertices in the target. This is not entirely

straightforward—the lack of unsatisfiable instances for testing purposes cannot be addressed simply by taking a pattern graph from one of the existing random suites with the "wrong" target graph, as this tends to give either a trivially unsatisfiable instance, or a satisfiable instance. (In particular, it is *not* the case that a relatively small random graph is unlikely to appear in a larger random graph.) Sections 6.2 and 6.3 have been published by McCreesh, Prosser, and Trimble (2016) as "Heuristics and Really Hard Instances for Subgraph Isomorphism Problems"; the pseudo-Boolean, SAT and MIP encodings are due to that paper's third author.

This work builds upon the phase transition phenomena observed in satisfiability and graph colouring problems first described by Cheeseman, Kanefsky, and Taylor (1991) and Mitchell, Selman, and Levesque (1992), which we discussed for clique problems in Chapter 2. For subgraph isomorphism we identify three relevant control parameters: we can independently alter the edge probability of the pattern graph, the edge probability of the target graph, and the relative orders (number of vertices) of the pattern and target graphs. For non-induced isomorphisms, with the correct choice of parameters we see results very similar to those observed with Boolean satisfiability problems: there is a phase transition (whose location we can predict) from satisfiable to unsatisfiable, and we see a solver-independent complexity peak near this phase transition. Additionally, understanding this behaviour helps us to select better variable- and value-ordering heuristics—this is the second contribution of this chapter.

For certain choices of parameters for induced isomorphisms, there are two phase transitions, going from satisfiable to unsatisfiable, and then from unsatisfiable back to satisfiable. Again, when going from satisfiable to unsatisfiable (from either direction), instances go from being trivial to really hard to solve. However, each of the three solvers we tested also finds the central unsatisfiable region to be hard, despite it not being near a phase transition. To show that this is not a simple weakness of current subgraph isomorphism algorithms, we verify that this region is also hard when using a pseudo-Boolean encoding, and under reduction to the clique problem. Interestingly, the constrainedness measure proposed by Gent, MacIntyre, Prosser, and Walsh (1996) *does* predict this difficult region—the third contribution of this chapter is to use these instances to provide evidence in favour of constrainedness, rather than proximity to a phase transition, being an accurate predictor of difficulty, and to show that constrainedness is not simply a refinement of a phase transition prediction.

When labels on vertices are introduced, as is commonly seen in real-world applications and in graph database systems, richer behaviour emerges, particularly when moving away from a uniform labelling scheme, and we see that VF2 behaves much worse than the Glasgow and LAD algorithms in certain cases. A close look shows that these cases *should* be easy to solve.

This is not simply a theoretical curiosity. A labelled version of subgraph isomorphism is one of the key components in supporting complex queries in graph databases—typically, these systems store a fixed set of target graphs, and for a sequence of pattern queries, they must

return every target graph which contains that pattern. One common approach to this problem is to combine a subgraph isomorphism algorithm with an indexing system in a so-called "filter / verify" model, where invariants are used to attempt to pre-exclude unsatisfiable instances to avoid the cost of a subgraph isomorphism call. In this context, the terms *matching* and *verification* are often used for the subgraph isomorphism step (or a slightly broader problem, for example permitting wildcards).

We look at some of the datasets commonly used to test graph database systems, which the literature suggests are extremely hard to solve. Simple experiments show that the entire perceived difficulty of each of these datasets comes down to the widespread use of especially poor subgraph isomorphism algorithms. Our fourth contribution is to show that the use of forward-checking and the right variable ordering heuristic, even without the more sophisticated recent advances in subgraph algorithms, makes the entire graph database filter / verify paradigm unnecessary. Finally, we explain why filter / verify (and other recently proposed techniques) cannot be beneficial even on more challenging instances except if an extremely poor choice of subgraph isomorphism algorithm is made.

6.1 Experimental Setup

The experiments in this chapter are performed on systems with Intel Xeon E5-4650 v2 CPUs and 768GBytes RAM, running Scientific Linux release 6.7. We will be working with three subgraph isomorphism solvers: the version of the algorithm from the previous chapter described in McCreesh and Prosser (2015a), which we call the Glasgow solver; LAD (Solnon, 2010); and VF2 (Cordella et al., 2004). The first of these algorithms is implemented in C++, and the other two in C. Each was compiled using GCC 4.9.

Recall that the Glasgow and LAD solvers use backtracking search to build up an assignment of target vertices (values) to pattern vertices (variables), but differ in terms of inference and ordering heuristics. The approach used by VF2 is similar, although the domains of variables are not stored (in the style of conventional backtracking, rather than forward-checking), and so domain wipeouts are not detected until an assignment is made. Both the Glasgow and LAD solvers use domain sizes to guide search, whilst VF2 uses adjacency to previously-assigned vertices.

In this chapter we measure only the number of recursive calls (guessed assignments) made, not runtimes. We are not aiming to compare absolute performance between solvers; rather, we are looking for solver-independent patterns of difficulty. All experiments use a timeout of 1,000 seconds, which is enough for the Glasgow solver to solve nearly all instances (whose orders were selected with this timeout in mind), although we may slightly overestimate the proportion of unsatisfiable instances for extremely sparse or dense pattern graphs. The LAD and VF2 solvers experienced many more failures with this timeout, so our picture of just how

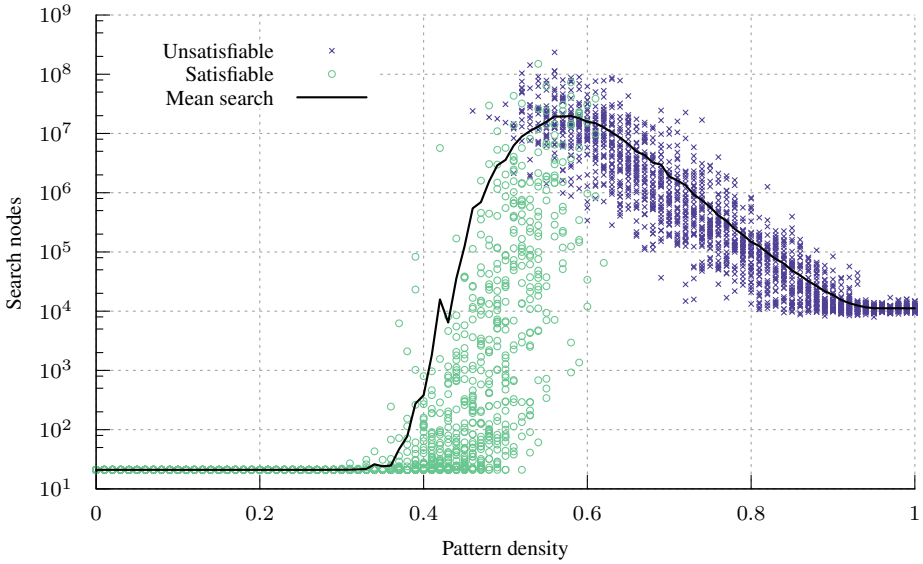


Figure 6.1: With a fixed pattern graph order of 20, a target graph order of 150, a target edge probability of 0.40, and varying pattern edge probability, we observe a phase transition and complexity peak with the Glasgow solver in the non-induced variant. Each point represents one instance. The lines show mean search effort and mean proportion satisfiable.

hard the hardest instances are with these solvers is less detailed.

6.2 Non-Induced Subgraph Isomorphisms

Suppose we arbitrarily decide upon a pattern graph order of 20, a target graph order of 150, a fixed target edge probability of 0.40, and no vertex labels. As we vary the pattern edge probability from 0 to 1, we would expect to see a shift from entirely satisfiable instances (with no edges in the pattern, we can always find a match) to entirely unsatisfiable instances (a maximum clique in this order and edge probability of target graph will usually have between 9 and 12 vertices). The move from green circles (satisfiable) to blue crosses (unsatisfiable) in Figure 6.1 shows that this is the case. For densities of 0.67 or greater, no instance is satisfiable; with densities of 0.44 or less, every instance is satisfiable; and with a density of 0.55, roughly half the instances are satisfiable.

The line plots mean search effort using the Glasgow solver: for sparse patterns, the problem is trivial, for dense patterns proving unsatisfiability is not particularly difficult, and we see a complexity peak around the point where half the instances are satisfiable. We also plot the search cost of individual instances, as points. The behaviour we observe looks remarkably similar to the clique decision problem and to random 3SAT problems—compare, for example, Figure 2.5 on page 38, or Figure 1 of Leyton-Brown et al. (2014). In particular, satisfiable instances tend to be easier, but show greater variation than unsatisfiable instances, and there are exceptionally hard satisfiable instances (Smith and Grant, 1997). (Recall from

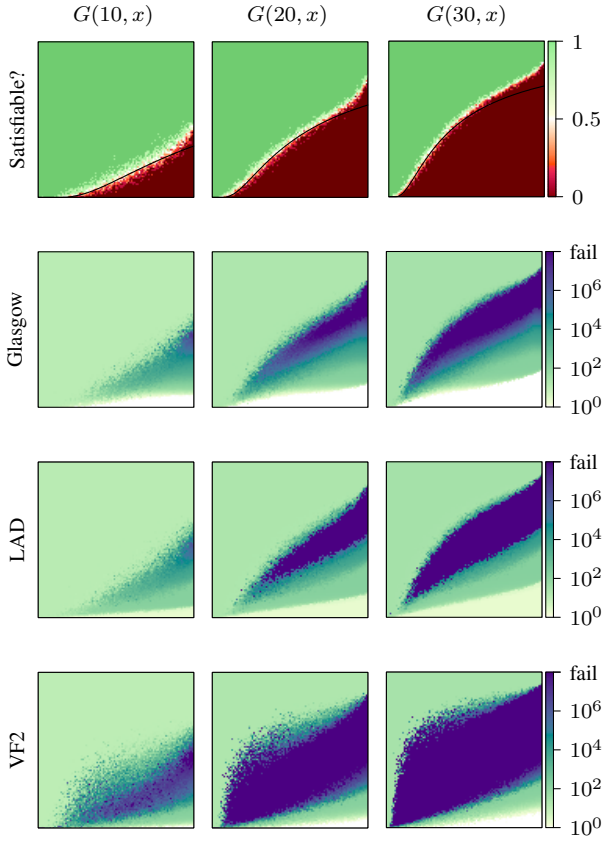


Figure 6.2: Behaviour of algorithms on the non-induced variant. For each plot, the x-axis is the pattern edge probability and the y-axis is the target edge probability, both from 0 to 1. Along the top row, we show the proportion of instances which are satisfiable; the white bands show the phase transitions, and the black lines are the predictions using equation (6.1) of where the phase transition will occur. On the subsequent three rows, we show the number of search nodes used by the Glasgow, LAD and VF2 solvers; the dark regions indicate "really hard" instances.

Chapter 5 that the Glasgow solver supports parallel search with a work-stealing strategy explicitly designed to reduce or eliminate these. We are not using parallel search in this chapter to avoid dealing with the complexity of search tree measurements under parallelism.)

What if we alter the edge probabilities for both the pattern graph and the target graph? In the top row of Figure 6.2 we show the satisfiability phase transition for the non-induced variant, for patterns of order 10, 20 and 30, targets of order 150, and varying pattern (x-axis) and target (y-axis) edge probabilities. Each axis runs over 101 edge probabilities, from 0 to 1 in steps of 0.01. For each of these points, we generate ten random instances. The colour denotes the proportion of these instances which were found to be satisfiable. Inside the red region, at the bottom right of each plot, every instance is unsatisfiable—here we are trying to find a dense pattern in a sparse target. In the green region, at the top left, every instance is satisfiable—we are looking for a sparse pattern in a dense target (which is easy, since we only have to preserve adjacency, not non-adjacency). The white band between the regions

shows the location of the phase transition: here, roughly half the instances are satisfiable. (We discuss the black line below.)

On subsequent rows, we show the average number of search nodes used by the different algorithms. In general, satisfiable instances are easy, until very close to the phase transition. As we hit the phase transition and move into the unsatisfiable region, we see complexity increase. Finally, as we pass through the phase transition and move deeper into the unsatisfiable region, instances become easier again. This behaviour is largely solver-independent, although VF2 has a larger hard region than Glasgow or LAD. VF2 also finds some instances extremely hard when the target is empty but the pattern is not—this turns out to be extremely important, and we return to it in Sections 6.4 and 6.5. Thus, although we have moved away from a single control parameter, we still observe the easy-hard-easy pattern seen in many NP-complete problems.

6.2.1 Locating the Phase Transition

We can approximately predict the location of the phase transition by calculating (with simplifications regarding rounding and independence) the expected number of solutions for given parameters. Since we are trying to find an *injective* mapping from a pattern $P = G(p, d_p)$ to a target $T = G(t, d_t)$, there are

$$t^{\underline{p}} = t \cdot (t-1) \cdot \ldots \cdot (t-p+1)$$

possible assignments of target vertices to pattern vertices. If we assume the pattern has exactly $d_p \cdot \binom{p}{2}$ edges, we obtain the probability of all of these edges being mapped to edges in the target by raising d_t to this power, giving an expected number of solutions of

$$\langle Sol \rangle = t^{\underline{p}} \cdot {d_t}^{d_p \cdot \binom{p}{2}}. \tag{6.1}$$

This formula predicts a very sharp phase transition from $\langle Sol \rangle \ll 1$ to $\langle Sol \rangle \gg 1$, which may easily be located numerically. We plot where this occurs using black lines in the first row of Figure 6.2.

This prediction is generally reasonably accurate, except that for very low and very high pattern densities, we overestimate the satisfiable region. This is due to variance: although an expected number of solutions much below one implies a high likelihood of unsatisfiability, it is not true that a high expected number of solutions implies that any particular instance is likely to be satisfiable. (Consider, for example, a sparse graph which has several isolated vertices. If one solution exists, other symmetric solutions can be obtained by permuting the isolated vertices. Thus although the expected number of solutions may be one, there cannot be exactly one solution.) A similar behaviour is seen with random constraint satisfaction

problems (Smith and Dyer, 1996).

6.2.2 Variable and Value Ordering Heuristics

Various general principles have been considered when designing variable and value ordering heuristics for backtracking search algorithms—one of these is to try to maximise the expected number of solutions inside any subproblem considered during search (Gent, MacIntyre, Prosser, Smith, et al., 1996). This is usually done by cheaper surrogates, rather than direct calculation. When branching, both LAD and Glasgow pick a variable with fewest remaining values in its domain: doing this will generally reduce the first part of the $\langle Sol \rangle$ equation by as little as possible. When two or more domains are of equal size, LAD simply breaks ties lexicographically, whereas Glasgow will pick a variable corresponding to a pattern vertex of highest degree. This strategy was determined empirically, but could have been derived from the $\langle Sol \rangle$ formula: picking a pattern vertex of high degree will make the remaining pattern subgraph sparser, which will decrease the exponent in the second half of the formula, maximising the overall value. LAD does not apply a value ordering heuristic, but Glasgow does: it prefers target vertices of lowest degree. Again, this was determined empirically, but it has the effect of increasing $\langle Sol \rangle$ by increasing the remaining target density. The VF2 heuristics, in contrast, are based around preserving connectivity, which gives very little discrimination except on the sparsest of inputs.

6.3 Induced Subgraph Isomorphisms

In the first four rows of Figure 6.3 we repeat our experiments, finding induced isomorphisms. With a pattern of order 10, we get two independent phase transitions: the bottom right half of the plots resemble the non-induced results, and the top left half is close to a mirror image. The central satisfiable region, which is away from either phase transition, is computationally easy, but instances near the phase transition are hard.

For larger patterns of order 20 and 30, we have a large unsatisfiable region in the middle. Despite not being near either phase transition, instances in the centre remain computationally challenging. We also plot patterns of orders 14, 15 and 16, to show the transition between the two behaviours.

We might expect these complexity plots to be symmetric along the diagonal, since for the induced problem, if we replace both inputs with their complements, the solutions remain the same. For the Glasgow solver, this is the case. This should be expected, because this complement property is precisely how the Glasgow solver handles the induced variant (although the heuristics may differ between the two, which we discuss below). For LAD, some of the very dense patterns are slightly harder than their diagonal opposites (LAD reasons

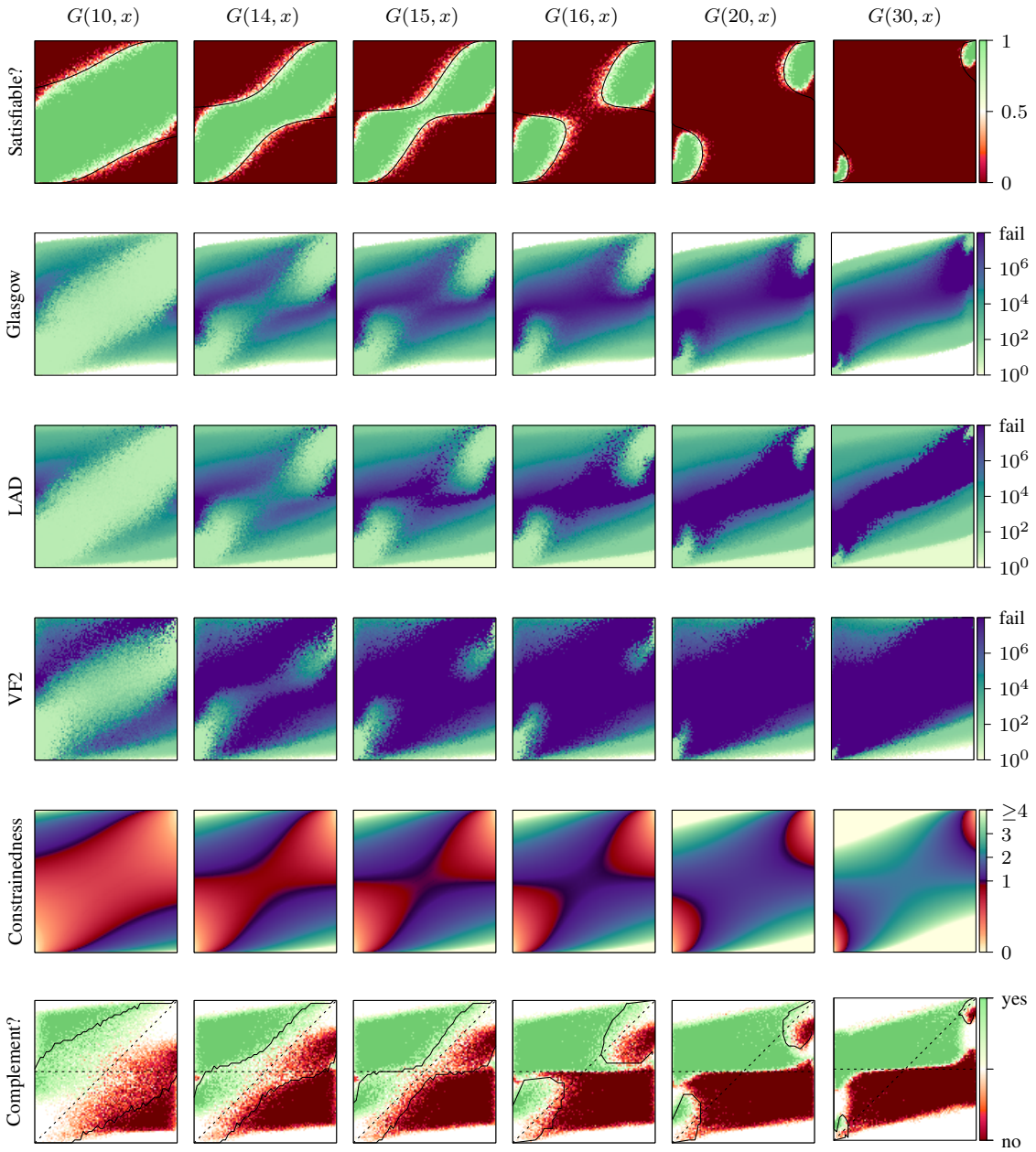


Figure 6.3: Behaviour of algorithms on the induced variant, shown in the style of Figure 6.2. The first row shows the empirical phase transition locations, together with the theoretical predictions from equation (6.2), and the next three rows show empirical hardness. The fifth row plots constrainedness using equation (6.3): the darkest region is where $\kappa = 1$, and the lighter regions show where the problem is either over- or under-constrained. The final row shows when the Glasgow algorithm performs better when given the complements of the pattern and target graphs as inputs—the solid lines show the location of the phase transition, and the dotted lines are $t_d = 0.5$ and the $p_d = t_d$ diagonal.

about degrees, but not about complement-degrees); it is interesting to note that for larger target graphs, VF2 finds *all* dense pattern graphs difficult.

6.3.1 Predictions and Heuristics

To predict the location of the induced phase transition, we repeat the argument for locating the non-induced phase transition and additionally consider non-edges, to get an expected number of solutions of

$$\langle Sol \rangle = t^{\underline{p}} \cdot {d_t}^{d_p \cdot \binom{p}{2}} \cdot (1 - d_t)^{(1-d_p) \cdot \binom{p}{2}}. \tag{6.2}$$

We plot this using black lines on the top row of Figure 6.3—again, our prediction is accurate except for very sparse or very dense patterns.

We might guess that degree-based heuristics would just not work for the induced problem: for any claim about the degree, the opposite will hold for the complement constraints. However, empirically, this is not the case: on the final row of Figure 6.3, we show whether it is better to use the original pattern and target as the input to the Glasgow algorithm, or to take the complements. (The only steps performed by the Glagsow algorithm which differ under taking the complements are the degree-based heuristics. LAD and VF2 are not symmetric in this way: LAD performs a filtering step using degree information, but does not consider the complement degree, and VF2 uses connectivity in the pattern graph.)

For patterns of order 10, it is always better to try to move towards the satisfiable region: if we are in the bottom right diagonal half, we are best retaining the original heuristics (which move us towards the top left), and if we are in the top left we should use the complement instead. This goes against a suggestion by Walsh (1998) that switching heuristics based upon an estimate of the solubility of the problem may offer good performance.

For larger patterns, more complex behaviour emerges. If we are in the intersection of the bottom half and the bottom right diagonal of the search space, we should always retain the original heuristic, and if we are in the intersection of the top half and the top left diagonal, we should always use the complements. This behaviour can be predicted by taking the partial derivatives of $\langle Sol \rangle$ in the $-p_d$ and t_d directions. However, when inside the remaining two eighths of the parameter space, the partial derivatives of $\langle Sol \rangle$ disagree on which heuristic to use, and using directional derivatives is not enough to resolve the problem. A close observation of the data suggests that the actual location of the phase transition may be involved (and perhaps Walsh's suggestion applies only in these conditions). In any case, $\langle Sol \rangle$ is insufficient to explain the observed behaviour in these two eighths of the parameter space.

In practice, this is unlikely to be a problem: most real-world instances are extremely sparse and are usually easy. In this situation, these experiments justify reusing the non-induced heuristics on induced problems.

6.3.2 Is the Central Region Genuinely Hard?

The region in the parameter space where both pattern and target have medium density is far from a phase transition, but nevertheless contains instances that are hard for all three solvers. We would like to know whether this is due to a weakness in current solvers (perhaps our solvers cannot reason about adjacency and non-adjacency simultaneously?), or whether instances in this region are inherently difficult to solve. Thus we repeat the induced experiments on smaller pattern and target graphs, using different solving techniques. Although these techniques are not competitive in absolute terms, we wish to see if the same pattern of behaviour occurs. The results are plotted in Figure 6.4.

Our pseudo-Boolean (PB) encoding is as follows. For each pattern vertex v and each target vertex w, we have a binary variable which takes the value 1 if and only if v is mapped to w. Constraints are added to ensure that each pattern vertex maps to exactly one target vertex, that each target vertex is mapped to by at most one pattern vertex, that adjacent vertices are mapped to adjacent vertices, and that non-adjacent vertices are mapped to non-adjacent vertices. We use the Clasp solver (Gebser et al., 2011) version 3.1.3 to solve the pseudo-Boolean instances. The instances that are hard for the Glasgow solver remain hard for the PB solver, including instances inside the central region, and the easy satisfiable instances remain easy. Similar results are seen with the Glucose SAT solver (Audemard and Simon, 2014) using a direct encoding of the cardinality constraints. We also show an integer program encoding: the Gurobi solver is only able to solve some of the trivial satisfiable instances, and was almost never able to prove unsatisfiability within the time limit.

The *association graph encoding* of a subgraph isomorphism problem is a reduction to the clique decision problem. We discuss this encoding in much more detail in Chapter 7; briefly, it constructed by creating a new graph with a vertex for each pair (p, t) of vertices from the pattern and target graphs respectively. There is an edge between vertex (p_1, t_1) and vertex (p_2, t_2) if mapping p_1 to t_1 and p_2 to t_2 simultaneously is permitted, i.e. p_1 is adjacent to p_2 if and only if t_1 is adjacent to t_2. A clique of size equal to the order of the pattern graph exists in the association graph if and only if the problem is satisfiable (Levi, 1973). We used this encoding with Algorithm 2.1, modified to solve the decision problem rather than the optimisation problem. Again, our results show that the instances in the central region remain hard, and additionally, some of the easy unsatisfiable instances become hard.

Together, these experiments suggest that the central region may be genuinely hard, despite not being near a phase transition. The clique results in particular rule out the hypothesis that subgraph isomorphism solvers only find this region hard due to not reasoning simultaneously about adjacency and non-adjacency, since the constraints in the association graph encoding consider compatibility rather than adjacency and non-adjacency.

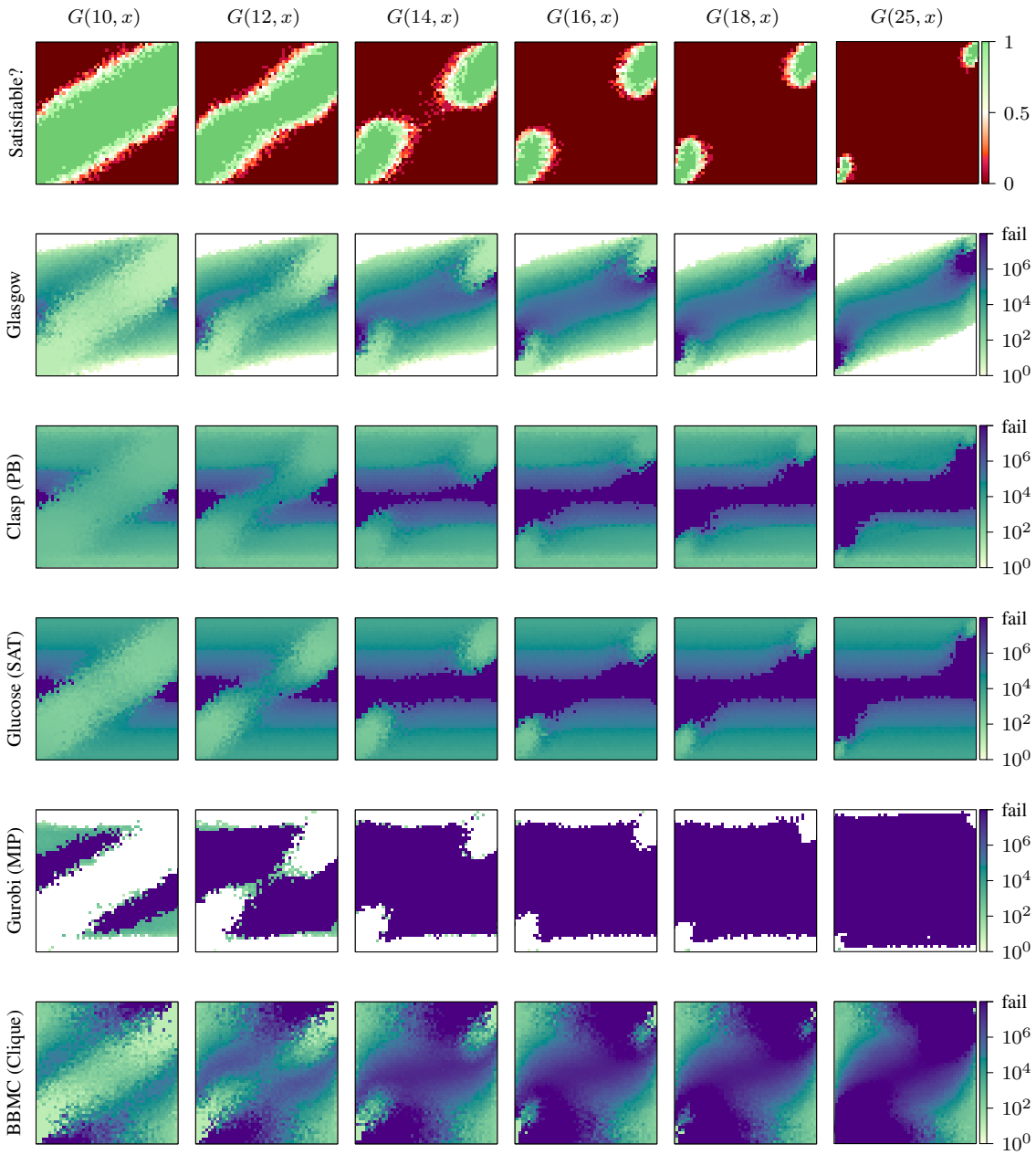


Figure 6.4: Behaviour of other solvers on the induced variant on smaller graphs, shown in the style of Figure 6.2. The second row shows the number of search nodes used by the Glasgow algorithm, the third and fourth rows show the number of decisions made by the pseudo-Boolean and SAT solvers, the fifth shows the number of search nodes used on the MIP encoding, and the final row the clique encoding.

6.3.3 Constrainedness

Constrainedness, denoted κ, is an alternative measure of difficulty designed to refine the phase transition concept, and to generalise hardness parameters across different combinatorial problems (Gent, MacIntyre, Prosser, and Walsh, 1996). A problem with $\kappa < 1$ is said to be underconstrained, and is likely to be satisfiable; a problem with $\kappa > 1$ is overconstrained, and is likely to be unsatisfiable. Empirically, problems with κ close to 1 are hard, and problems where κ is very small or very large are usually easy. By handling injectivity as a restriction on the size of the state space rather than as a constraint, we derive

$$\kappa = 1 - \frac{\log\left(t\underline{^p} \cdot {d_t}^{d_p \cdot \binom{p}{2}} \cdot (1-d_t)^{(1-d_p)\cdot\binom{p}{2}}\right)}{\log t\underline{^p}} \tag{6.3}$$

for induced isomorphisms, which we plot on the fifth row of Figure 6.3. We see that constrainedness predicts that the central region will still be relatively difficult for larger pattern graphs: although the problem is overconstrained, it is less overconstrained than in the regions the Glasgow and LAD solvers found easy. Thus it seems that rather than just being a unification of existing generalised heuristic techniques, constrainedness also gives a better predictor of difficulty than proximity to a phase transition—our method generates instances where constrainedness and "close to a phase transition" give very different predictions, and constrainedness gives the better prediction.

Unfortunately, constrainedness does not help us with heuristics: minimising constrainedness gives the same predictions as maximising the expected number of solutions.

6.4 Labelled Graphs

So far, we have looked at unlabelled graphs. What happens when labels on vertices are introduced? This is common in real-world applications—for example, when working with graphs representing chemical molecules, mappings are typically expected only to match carbon atoms with carbon atoms, hydrogen atoms with hydrogen atoms, and so on. We will look at the non-induced variant, as this seems to be more common in the literature.

6.4.1 Predictions and Empirical Hardness

Suppose our labels are drawn randomly from a set $L = \{1, \ldots, k\}$, where k is reasonably small compared to p. Let $\ell(v)$ be the label of vertex v. By defining

$$\mathrm{V}(P)|_n = \{v \in \mathrm{V}(P) : \ell(v) = n\}$$

to be the set of vertices with label k, we can partition the pattern vertices by label into disjoint sets

$$\{\mathrm{V}(P)|_1, \ldots, \mathrm{V}(P)|_k\},$$

each of which is expected to contain p/k vertices. Similarly, we may partition the target vertices into disjoint sets $\{\mathrm{V}(T)|_1, \ldots, \mathrm{V}(T)|_k\}$.

Without labels, there are $t^{\underline{p}} = t \cdot (t-1) \cdot \ldots \cdot (t-p+1)$ possible injective assignments of target vertices to pattern vertices. With labels, observe that for any label x, vertices in $\mathrm{V}(P)|_x$ may only be mapped to vertices in $\mathrm{V}(T)|_x$. Thus for each label x, we have an expected p/k variables, each of whose domains contain t/k values. We would like to say that the size of the state space is now

$$|S| = \left((t/k)^{\underline{p/k}}\right)^k,$$

but to do this we must state what $a^{\underline{b}}$ means when b is fractional. The gamma function $\Gamma(n)$ is equal to $(n-1)!$ for integers $n \geq 1$, but is also defined for positive real numbers, obeying the identify $\Gamma(x+1) = x\Gamma(x)$. By noting that $t^{\underline{p}} = t!/(t-p)!$, we may obtain a reasonable continuous extension by taking

$$|S| = \left(\frac{\Gamma\left(t/k + 1\right)}{\Gamma\left(t/k - p/k + 1\right)}\right)^k,$$

As before, we expect the pattern to have $d_p \cdot \binom{p}{2}$ edges, and so if we simplify by assuming the pattern will have *exactly* this many edges, we obtain the probability of all of these edges being mapped to edges in the target by raising d_t to this power, giving an estimate of

$$\langle Sol \rangle = \left(\frac{\Gamma\left(t/k + 1\right)}{\Gamma\left(t/k - p/k + 1\right)}\right)^k \cdot {d_t}^{d_p \cdot \binom{p}{2}}. \tag{6.4}$$

So how good are these predictions? The black lines on the first row of heatmaps in Figure 6.5 plot where we calculate $\langle Sol \rangle = 1$ will occur. For small numbers of labels, our predictions are slightly better than in the unlabelled case: there seems to be less of a variance problem for very dense patterns. Even as the numbers of labels becomes relatively large, the prediction of the phase transition still occurs in the right place, but with ten labels we start to see sporadic unsatisfiable instances deep inside the satisfiable region. With twenty labels, we instead get a kind of phase transition from "all unsatisfiable" to "mixed satisfiable and unsatisfiable". We can understand this intuitively: with sufficiently many labels, we might generate, say, a red vertex adjacent to a blue vertex in the pattern, but not in the target. With twenty labels, we even sometimes generate no vertices with a particular label in the target at all. This in many ways resembles "flaws" generated by certain random constraint satisfaction problem instance generators (Achlioptas et al., 2001; Gent, MacIntyre, Prosser, Smith, et al., 2001).

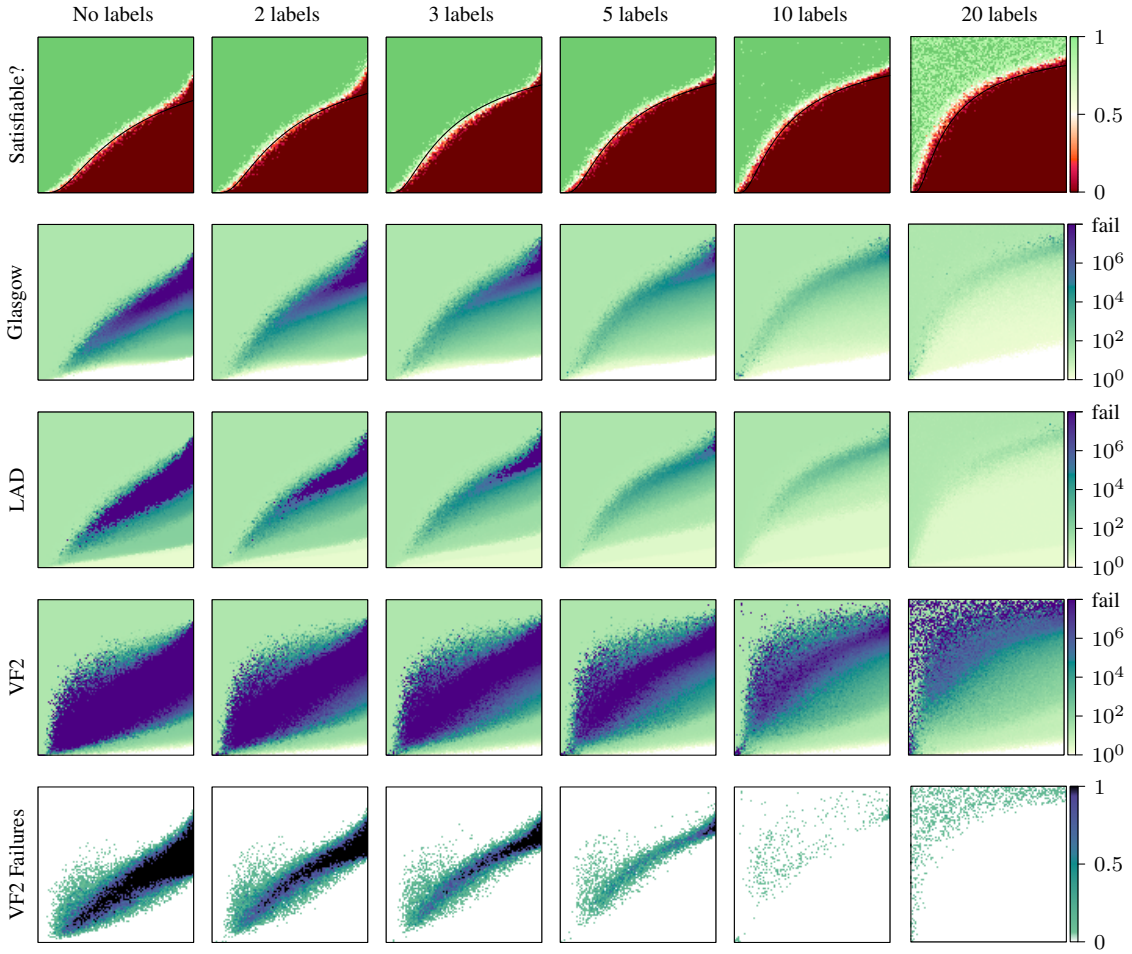


Figure 6.5: On the top row, predicted and actual location of the phase transition for labelled non-induced random subgraph isomorphism, with a pattern order of 20, a target order of 150, varying pattern (x-axis) and target (y-axis) density, and varying numbers of labels. On subsequent rows, the average number of search nodes needed to solve an instance, for three different solvers, and on the final row, the proportion of instances aborted due to a timeout with VF2.

What about empirical hardness? As before, some instances on the phase transition are hard for all solvers (although as the number of labels increases, the hardest instances become easier). Instances far from the phase transition are easy, except for VF2, which occasionally finds some instances with larger numbers of labels very difficult. This behaviour occurs both on satisfiable instances, and on the flawed unsatisfiable instances deep inside the "satisfiable" region. It is interesting to observe that all such unsatisfiable instances that VF2 finds difficult have a very small proof of unsatisfiability, with neither Glasgow nor LAD requiring more than one hundred recursive calls.

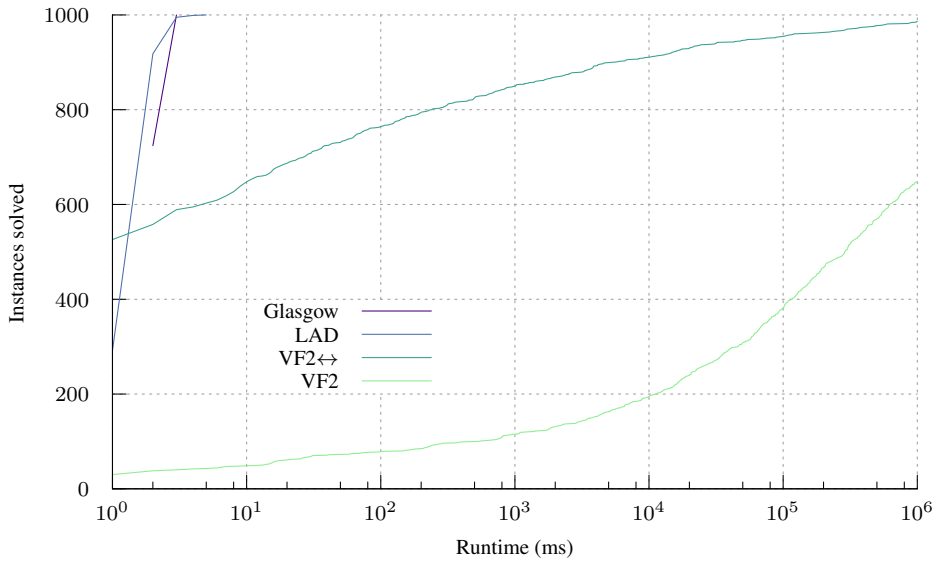


Figure 6.6: Cumulative number of instances solved over time, using the richer model of randomness described in Section 6.4.2.

6.4.2 Richer Label Models, and VF2's Deficiencies

Why does VF2 find some of these instances so hard? VF2 does not track domains, like other algorithms, and does not detect a domain wipeout until it branches on a particular vertex. It also cannot detect small domains, and only uses "adjacency to an existing assignment" as a branching heuristic. This can make it very hard for VF2 to detect that it is in an obviously failed state, or that an instance or subproblem is trivially unsatisfiable.

To illustrate this point further, we will now look at some instances created using a slightly more structured random model. We create a family of one thousand labelled instance pairs, as follows. To create a pattern graph, we create ten vertices with label zero, with edges between these vertices with probability 0.2. We then add another ten vertices, with labels chosen randomly between one and thirty, and add edges between these vertices and the zero-labelled vertices with probability 0.1. To generate a target, we follow a similar process: we have fifty vertices with label zero and edge probability 0.2, fifty vertices with labels randomly between one and thirty, and edges from the first set of vertices to the second set with probability 0.3. This model was selected and the parameters tuned to provide a demonstration of particular behaviours of VF2, not because of any natural property (although inspiration came from seeking a very crude approximation of chemical graphs, which often contain a lot of carbon in the centre, and other atoms around the outside). From our set of one thousand such instances, sixty six are satisfiable.

We plot the cumulative number of instances solved over time for these instances in Figure 6.6. Both Glasgow and LAD find all of these instances trivial, but VF2 finds many of them extremely difficult. The "VF2↔" line shows what happens with VF2 if the graphs are permuted, so that the non-zero labelled vertices are given lower vertex numbers rather than

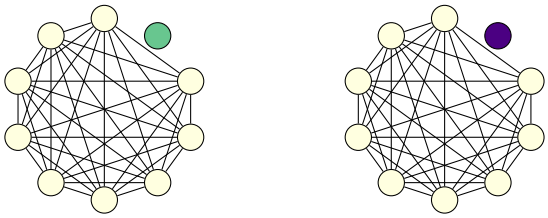


Figure 6.7: A trivially unsatisfiable subgraph isomorphism instance which VF2 finds exponentially difficult. The pattern (left) and target (right) both consist of a clique, plus one extra vertex which has a different label in each graph.

higher vertex numbers. This makes VF2 behave better, but still extremely poorly compared to the other two solvers; we return to this in reference to work by Katsarou, Ntarmos, and Triantafillou (2017) in the following section. It is also important to note for later that both VF2 variations find some satisfiable and some unsatisfiable instances extremely hard.

An even more extreme example of VF2's misbehaviour can be seen in Figure 6.7. Here we have a pattern graph and a target graph, both of which are cliques plus one isolated vertex, and the isolated vertices have different unique labels. This is trivially unsatisfiable (and Glasgow and LAD detect this without search), but unless the labelled vertex is given the lowest vertex number (so it is branched on first), VF2 takes nearly a million recursive calls to detect this: VF2 will try to map every adjacent vertex in the clique before considering the difficult isolated vertex. This example can also be extended slightly to fool any simple static heuristic, or any simple label counting mechanism.

These experiments further highlight that VF2 occasionally finds some instances which should be easy hard. But does this cause problems in practice? The literature suggests that it does. For example, Grömping (2014) uses VF2 in a package for the R statistics language, and states

> "There are (not so many) instances for which creation of a clear design is prohibitively slow in the current implementation that evaluates subgraph isomorphism with the VF2 algorithm ... Recent experiences with a few of these showed that the LAD algorithm was very fast in ruling out impossible matches, where VF2 took a long time."

Similarly, Murray (2012, Chapter 4) uses VF2 inside a compiler, and observes extremely variable (and prohibitively high) compile times in some cases due to the expense of subgraph isomorphism calls—given the heavily labelled nature of these graphs, we conjecture that any domain-based algorithm would eliminate this cost.

However, by far the biggest problem is in graph databases. The following section shows how widespread use of VF2 has not just lead to overly pessimistic conclusions regarding performance, but has misdirected the design of larger systems.

6.5 Querying Graph Databases

A particularly common use of subgraph isomorphism algorithms is inside graph databases. The general problem these systems solve is, for a set of target graphs, to process a pattern query and return every target graph which is subgraph-isomorphic to that pattern. The set of target graphs is usually seen as fixed, or at least rarely-changing, whilst the patterns arrive dynamically. This has lead to the development of systems which perform extensive computations on the target graphs beforehand, in the hopes of reducing the response times for individual pattern queries. The most popular of these strategies is a form of indexing which is often named *filter / verify*.

6.5.1 The Filter / Verify Paradigm

The filter / verify approach has an interesting history, of which we now give a very selective and incomplete overview. Our description is biased by a general modern understanding of the empirical hardness of NP-complete problems, which was not widely known at least at the time of the earlier papers we discuss. The common theme of all of the following papers is that pre-computed information is used to eliminate certain unsatisfiable instances from consideration, without performing a subgraph isomorphism test. For example, an index might contain a bit of information expressing whether a target graph contains two red vertices. When a pattern graph with two red vertices is used as a query, any target whose feature set does not have this bit set would not be considered, and so a subgraph isomorphism call would not be made for that pattern / target pair. (In practice, feature hashing is often used, which can lead to false positives, but this is not relevant to our discussion.)

An early graph database system by Shasha, J. T. Wang, and Giugno (2002) uses a filtering heuristic to eliminate unsatisfiable instances based upon simple structural elements. It is not clear whether the aim is to reduce I/O costs, or to reduce the number of queries which must be tested, and the experiments do not answer whether the indexing is effective. However, the work was influenced by a commercial graph database system, whose documentation (Daylight Chemical Information Systems, Inc., 2011, Section 7.1) states that indexing is used to minimise disk accesses.

Subsequently, in a widely cited paper, X. Yan, P. S. Yu, and Han (2004) introduce an indexing system called gIndex. Again, this system handles queries by first producing a set of candidates by eliminating certain unsatisfiable instances, this time by using substructures. They argue that the query response time, which is to be minimised, is governed by the equation

$$T = T_{search} + |C_q| \cdot T_{iso\_test},$$

where T_{search} is the time taken to search for a candidate set of potential solutions of size $|C_q|$,

and T_{iso_test} is the cost of a subgraph isomorphism test. They reason that since isomorphism testing is NP-complete, by making $|C_q|$ as small as possible, the query response time will be reduced. (Importantly, it is *not* time taken to load graphs from disk which contributes to the per-candidate cost, but rather the time to perform a subgraph isomorphism call.) They conclude that "graph indexing plays a critical role at efficient query processing in graph databases".

This equation is repeated and expanded upon by X. Yan, P. S. Yu, and Han (2005) to explicitly separate I/O and isomorphism testing costs, obtaining a query response time of

$$T_{search} + |C_q| \cdot (T_{io} + T_{iso\_test}).$$

The authors explicitly state that "the value of T_{iso_test} does not change much for a given query". The argument presented is that

> "Sequential scan is very costly because one has to not only access the whole graph database but also check subgraph isomorphism. It is known that subgraph isomorphism is an NP-complete problem. Clearly, it is necessary to build graph indices in order to help processing graph queries."

With what we now know about the behaviour of modern subgraph isomorphism algorithms, and the nature of solving NP-complete problems in general, we should immediately be suspicious of these claims. We do not expect T_{iso_test} to be anything like a constant, even if the orders of the input graphs are similar. In particular, any instance which can be excluded based upon filtering must have a very small proof of unsatisfiability. These instances *should* be trivial with any decent subgraph isomorphism algorithm. Thus, all filtering *should* be doing is eliminating the startup costs of a trivial subgraph isomorphism call. The fact that filtering was successful empirically should make us wonder whether the subgraph isomorphism algorithm being used was excessively primitive. Indeed, the subgraph isomorphism algorithm used is described only as "the simplest approach" in the paper. Additionally, the experiments focus on reducing the size of the candidate set, without considering the time taken to verify different candidate set instances. The claim that T_{iso_test} does not change much is not justified experimentally, and no consideration is given as to whether this would hold true for other subgraph matching algorithms.

Moving forwards, the (simpler form of the) query response time equation is repeated by Zhao, J. X. Yu, and P. S. Yu (2007). Again, the work has a focus on reducing the candidate set size through indexing. The authors appear to believe that the cost of the isomorphism test is not a major factor in influencing the result, and focus on the remaining terms in the equation. They use the "average cost" of a subgraph isomorphism test as a constant, without considering that the average cost could be influenced by the character of the candidate set.

The equation is also used by Haoliang Jiang et al. (2007), who claim that "usually the

verification time dominates the Query Response Time [s]ince the computational complexity of T_{iso_test} is NP-Complete". They note that

> "Approximately, the value of T_{iso_test} does not change too much with the difference of query. Thus, the key to reducing query response time is to minimize the size of the candidate answer set".

This claim becomes understandable when one examines the subgraph matching algorithm chosen for the verification step (Ullmann, 1976): as the algorithm predates techniques like arc-consistent all-different propagation (Régin, 1994), it cannot immediately detect unsatisfiability in simple cases like there being two red vertices in the pattern but only one in the target.[1]

Without using the equation, Cheng et al. (2007) state that since subgraph isomorphism is NP-complete, processing by a sequential scan is infeasible. They introduce new filtering techniques to try to avoid the subgraph isomorphism step. A similar claim is made by Zhang, Hu, and Yang (2007) in an introduction of another indexing technique: "obviously it is inefficient to perform a sequential scan on every graph in the database, because the subgraph isomorphism test is expensive".

Muddying the waters slightly, a survey by Han et al. (2007) states that "large volumes of data" (not NP-completeness) is the reason for using indexing in these systems. However, in a description of a system tailored to biological networks, Zhang, S. Li, and Yang (2009) state that

> "Since the size of the raw database graph is small, it can be easily fit in the main memory. However, the query (matching) time will be very long due to the NP-hard complexity."

They suggest that indexing is a way of avoiding this.

Cao et al. (2011) propose a privacy-preserving cloud graph database system using filter / verify, stating that "checking subgraph isomorphism is NP-complete, and therefore it is infeasible to employ such a costly solution" which simply checks every graph for a match. G. Wang et al. (2012) look at indexing large sparse graphs. They state that "because subgraph isomorphism is an NP-complete problem, a filter-and-verification method is usually employed to speed up the search efficiency of graph similarity matching over a graph set". Similarly, after reviewing the literature, Yuan and Mitra (2013) argue that subgraph querying is costly because it is NP-complete, and that indices can improve the performance of graph database queries. Again, new indexing techniques are introduced. Subsequently, Katsarou, Ntarmos, and Triantafillou (2015) perform a comprehensive comparison of graph database filtering techniques. They state that performing a query against each graph in the dataset "obviously does not scale, as subgraph isomorphism is NP-complete". In describing a system which

[1] Although interestingly, this algorithm effectively does forward-checking and has a variable-ordering heuristic, before these concepts appeared in the constraints literature.

returns a special subset of graphs which match a query, Zheng et al. (2016) suggest that it is "NP-hard to check the graph isomorphism" (meaning subgraph isomorphism), and "in order to improve the time efficiency" they use an indexing system to "avoid as many costly subgraph isomorphism checkings as possible". Their index takes tens of thousands of seconds to build, and they suggest that Ullmann's algorithm and VF2 are state of the art for verification. And Peng et al. (2016) state that "obviously, it is impossible to employ some subgraph isomorphism algorithm, such as Ullmann or VF2", and argue that "in order to speed up query processing", they need to create indices.

The filter / verify paradigm also influences other research. For example, it is used by Tian and Patel (2008) in an approximate subgraph searching system: they state that Ullmann's algorithm "is prohibitively expensive for querying against [a] database with a large number of graphs", and that indices are used "to filter out graphs that do not match the query". More recently, Yuan, Mitra, and Giles (2013) continue a line of supergraph search work, again using a filtering step to avoid subgraph isomorphism calls (in the opposite direction, so queries are now target graphs). Hong et al. (2015) look at graph database subgraph matching with an additional set-similarity constraint, and state that Ullmann's algorithm and VF2 are "usually costly for large graphs" because they "do not utilize any index structure". They propose an indexing structure, which takes over 2,000 seconds to construct, and uses nearly 2GBytes of space. There is also research into maintaining indices when the set of target graph changes: for example Yuan, Mitra, H. Yu, et al. (2015) look at algorithms for updating graph indices. And in describing a system for reusing results of queries which are sub- or super-graphs of previous queries, J. Wang, Ntarmos, and Triantafillou (2016) state that querying is a "very costly operation as it entails the NP-complete problem of subgraph isomorphism", and place "an emphasis on the number of unnecessary subgraph isomorphism tests".

After some early ambiguity, then, it becomes clear that the intent behind filter / verify systems is to reduce the number of subgraph isomorphism calls, and that the cost of loading graphs from disk is not considered to be problematic. It is worth noting that the entire test datasets from most of these papers will comfortably fit in RAM on a modern desktop machine, even when an adjacency matrix representation is used.

Thus we can see that there are two critical beliefs underlying all of this work—firstly, that subgraph isomorphism is necessarily hard because it is NP-complete, and secondly, that there are ways of identifying unsatisfiable instances using short proofs that a subgraph isomorphism algorithm will not detect, but that an indexing system can. Throughout, the cost models used assume that the time for a subgraph isomorphism query does not particularly depend upon the instance, and nowhere is it considered that a good subgraph isomorphism algorithm should be able to eliminate obviously-unsatisfiable instances with a similar time requirement to an indexing system.

These beliefs are not entirely unfounded: none of the subgraph matching algorithms

considered in these papers will immediately detect if a pattern contains two red vertices, whilst the target graph contains only one. This kind of flaw *should* be picked up at the top of search by an all-different propagator (Régin, 1994). However, as Katsarou, Ntarmos, and Triantafillou (2015) note, some variation on VF2 (Cordella et al., 2004) is the usual matching algorithm of choice for graph database systems, although Ullmann's algorithm is sometimes chosen. Other approaches have been considered, albeit not with algorithms strong enough to establish all-difference. For example, Shang et al. (2008) propose an algorithm which makes use of the frequency of various features to guide search; Lee et al. (2012) determine experimentally that this technique tends to be very effective, even on families of graphs for which it was not designed.

We therefore believe it would be unlikely to cause too much astonishment if we suggested that a better subgraph matching algorithm could be dropped in as a black box replacement in graph databases systems to improve their performance. This is not our claim. Instead, this chapter shows that better algorithms invalidate the flawed premise underlying the entire filter / verify approach. We will now demonstrate that filter / verify is simply a poor workaround for the kinds of deficiency in VF2 demonstrated towards the end of the previous section.

6.5.2 Is Filtering Necessary?

To demonstrate that a pure subgraph isomorphism approach is feasible, with no indexing or supporting preprocessing, we look at four datasets commonly used in graph indexing evaluations (Giugno et al., 2013). We do not claim that these are high-quality datasets or that the associated queries are sensible, merely that they are widely used.

- The AIDS dataset contains graphs representing 40,000 chemical molecules. These graphs are labelled, and are fairly small (an average of 45 vertices) and sparse. Following Giugno et al., the queries are compounds with 8, 16 or 32 edges.

- The PDBS dataset (He et al., 2002) contains 30 labelled graphs representing DNA, RNA, and proteins, each duplicated twenty times. These can be relatively large, having up to tens of thousands of vertices, but are extremely sparse. The queries are randomly selected connected subgraphs and do not have a real-world meaning.

- The PCM dataset (Vehlow et al., 2011) contains 50 protein contact maps, each duplicated four times. These graphs have under a thousand vertices and below twenty thousand edges; they are slightly less sparse than the previous two datasets. As for PDBS, the queries are randomly generated and do not have a particular meaning.

- The PPI dataset contains 20 protein interaction networks, with up to ten thousand vertices. The queries have either four or eight vertices.

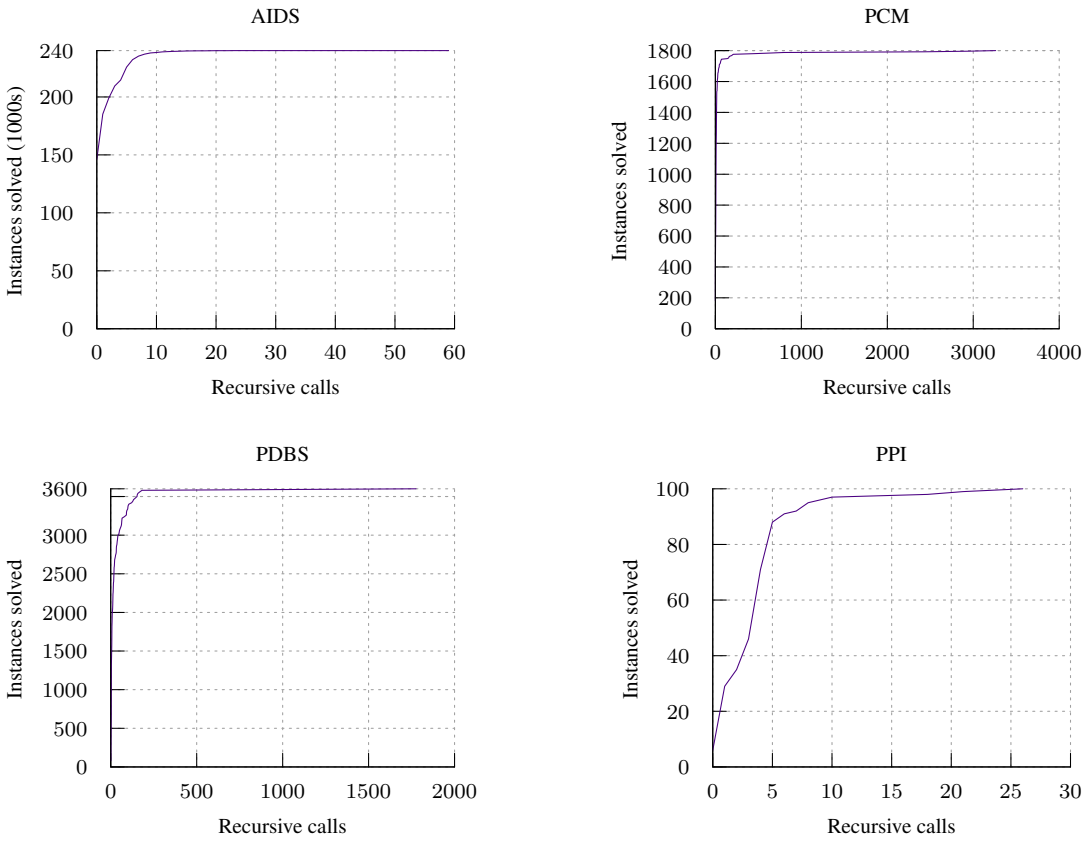


Figure 6.8: Cumulative number of instances solved as a function of search space size, using four graph database datasets and a simple domain-based subgraph isomorphism algorithm. Note that the x-axis shows recursive calls, and does *not* use a log scale.

Rather than the modern techniques discussed in Chapter 5, we deliberately select a very simple starting point: a forward-checking algorithm which uses the smallest domain first heuristic, the simple counting all-different propagator, and degree and neighbourhood degree sequence filtering at the top of search. Even this simple algorithm finds every instance nearly trivial, and no exponential behaviour is observed. As we show in Figure 6.8, the *hardest* instance for the AIDS dataset requires 59 recursive calls, and 146,175 of the instances can be solved without search (we measure recursive calls to demonstrate hardness, rather than algorithm tuning and good constant factors; recall from Chapter 5 that 10^4 to 10^6 recursive calls per second is feasible). For PDBS, the hardest problem requires 1,776 recursive calls (which occurs 20 times, due to duplicated queries), and none of the rest require more than 182 calls. For PCM the hardest problem requires 3,261 calls, and for PPI, 26. In other words, none of these instances are in any way computationally hard even for a simple domain-based algorithm, even before introducing stronger filtering, inference, or heuristics. Furthermore it is certainly not the case that a sequential scan is infeasible as so many filter / verify papers claim. The exponential behaviour seen by others in these instances is purely down to a bad choice of subgraph isomorphism algorithm, and these instances are not inherently "really hard".

There are at least four benefits beyond simplicity towards abolishing filter / verify:

- Domain filtering is useful on both satisfiable and unsatisfiable instances, whilst filtering can only eliminate trivially unsatisfiable instances, and is entirely wasted on satisfiable instances. Domain filtering is also useful even on relatively hard instances, since the information cuts down the search space rather than providing a simple "yes" or "no".
- Domain-based heuristics are much stronger than VF2's adjacency branching rules. Picking from small domains dynamically allows search to focus on the hardest part of the problem.
- This approach automatically combines features. For example, a pattern / target pair may have matching label features, and matching degree features, but if the only red pattern vertex has degree three whilst no red target vertex has degree more than two, domain filtering will detect this immediately. When combined with all-different propagation and maintained during search, this effect is even stronger.
- Finally, as indexing systems get more and more complex in an attempt to filter out a few more instances where VF2 performs poorly, the cost of index construction and maintenance is considerable. Indices proposed in the literature often take many hours to build, and can consume much more space than the original graphs.

In other words, using domain-based algorithms would not simply be a viable alternative to filter / verify with VF2, but rather would be a much better solution.

6.5.3 Rethinking Graph Matching for Database Systems

The evidence so far clearly shows a need for graph database systems to be redesigned making use of knowledge from constraint programming and artificial intelligence, and that this knowledge must inform the systems as a whole, and not just parts of them. Having demonstrated that filter / verify is both theoretically and practically a flawed approach to designing graph databases based upon a repeated misconception that NP-completeness means long runtimes, we now briefly discuss how future graph database systems should be designed, and how such systems can help support strong subgraph isomorphism algorithms.

First, though, we briefly consider whether other lessons can be learned from constraint programming: unfortunately indexing is not the only incorrect design choice being in these systems. For example, recent work by Katsarou, Ntarmos, and Triantafillou (2017) looks at what they call "straggler" queries, which are the small subset of queries that they observe taking much longer than others to solve. The authors present an approach to address this perceived problem that they describe as novel: "instead of trying to come up with new algorithms for sub-iso testing, we utilize isomorphic query rewritings and existing alternative algorithms in parallel". Their first claim is that permuting graphs before running the subgraph

isomorphism algorithm can lead to "wildly different" execution times. They note that VF2 does not define "a strict order in which the nodes of the query are matched", and so permuting the graphs (for example, by using degree or label frequency information) before search can sometimes give orders of magnitude improvements. The connection to variable- and value-ordering heuristics is not noted, and no consideration is given to even the simplest dynamic ordering heuristics like smallest domain first (Haralick and Elliott, 1980). A little thought shows that of the orderings proposed, those involving placing rare labels first are effectively poor approximations to a static "smallest domain at top of search first" ordering, whilst the remainder use degree as we did earlier in this chapter. However, Katsarou, Ntarmos, and Triantafillou do not investigate any algorithm which employs domains, let alone strong variable- or value-ordering heuristics or all-different propagation, and do not consider that their apparent successes could be due to VF2's weaknesses rather than an inherent property of NP-completeness. We saw in the previous section that although permuting input graphs could improve VF2's behaviour on some instances, doing so does not make its performance come close to that of domain-based algorithms. Katsarou, Ntarmos, and Triantafillou say they "hope that our findings will open up new research directions, striving to find appropriate, per-query, isomorphic rewritings, in combination with alternate per-query sub-iso algorithms that can yield large improvements". We believe it is important instead to emphasis previous research directions that have already solved most of this problem, before any further research effort is wasted.

Katsarou, Ntarmos, and Triantafillou's (2017) second claim is that "different algorithms find different queries hard". To address this, they run many subgraph isomorphism algorithms and input permutations in parallel; the extensive literature on parallel portfolios in general (Gomes and Selman, 2001), and the approach by Battiti and Mascia (2007) for subgraph isomorphism in particular, is not noted (and nor are portfolios mentioned when they say that "using machine learning models to predict which version ... to employ per query is of high interest"). The evidence so far in this chapter suggests that we should be wary. It is certainly true that there are some instances that all algorithms find hard, and there are good theoretical reasons to believe that these instances genuinely are really hard. Furthermore, algorithm portfolios are also well-known to be a successful technique, and are beneficial even with modern subgraph isomorphism algorithms (Kotthoff, McCreesh, and Solnon, 2016). However, in Section 6.4 we saw that there are many instances that VF2 finds hard that should not be hard, and that are not hard for other algorithms. If permuting graphs sometimes addresses the difficulty of some of these instances, then it is likely that they are not genuinely hard instances at all, and perhaps it would be better simply to start using an algorithm with domain tracking and domain-based ordering heuristics, rather than assembling a portfolio of bad algorithms in the hopes that at least one of them will often get lucky.

To test this suggestion, we return briefly to the experiments on the datasets discussed

earlier in this section. What if we had not used a smallest domain first heuristic, and instead used only degree-based heuristics explained earlier in this chapter? Although we still find all four of these datasets easy, and the changes to the runtimes are hard to detect (more time is spent in initialisation than in search), the worst case in the AIDS dataset now takes 4,051 recursive calls as opposed to 59. How about if, additionally, we do not detect domain wipeouts until attempting to branch on an empty variable? Suddenly three of the four datasets become extremely hard, with many instances now timing out after making hundreds of millions of recursive calls; the PPI dataset now requires up to 168,451 calls for some queries. Furthermore, slight changes to the input vertex ordering can now make some of these "hard" instances easy again.

This backs up our suspicions: the apparent success of Katsarou, Ntarmos, and Triantafillou's (2017) technique is due to the ease of sometimes getting better results out of a poor algorithm, rather than being an inherent hardness property, and it would not help on really hard problem instances. We point to Gent's (1998) admonition on how heuristics are benchmarked as being similarly relevant here:

> "Benchmark problems should be hard. I report on the solution of the five open benchmark problems introduced . . . for testing bin packing problems. Since the solutions were found either by hand or by using very simple heuristic methods, these problems would appear to be easy. In four cases I give improved packings to refute conjectures that previously reported packings were optimal, and I give a proof that the fifth conjecture was correct. . . . Future experimenters should be careful to perform tests on problems that can reasonably be regarded as hard."

Claiming huge improvements from an algorithm portfolio consisting only of variations of poor algorithms does not demonstrate a genuine improvement over the state of the art.

We have seen, then, that domain tracking and domain-based ordering heuristics are critical, and we have seen that these alone are enough to make indexing and other techniques irrelevant. There is therefore no point in continued research such as that by Carletti (2016) and Carletti, Foggia, and Vento (2015) into tweaking VF2's connectivity-based heuristics. However, we do not claim that the ultimate "big data" subgraph matching algorithm already exists: on the contrary, there is likely to be plenty of room for future improvement now that we understand the importance of getting algorithm design right. For example, a major disadvantage of using domains is the relatively expensive initialisation costs, which quickly add up when dealing with large numbers of trivial instances. Employing a presolver is an obvious approach—and VF2 is actually good in this role (Kotthoff, McCreesh, and Solnon, 2016)—but there are other possibilities. For example, minimal or lazy forward-checking (Bacchus and Grove, 1995; Dent, 1996; Dent and Mercer, 1994; Larrosa and Meseguer, 1998) avoids constructing every domain upfront, although adopting this require alternatives to smallest domain first and to

all-different. Such an approach may also be beneficial for huge target graphs, where having domains range over the entire target is impractical.

There is also scope for precalculating supporting information about target graphs. For example, neighbourhood degree sequences and supplemental graphs could both be pre-computed and stored. The aim here is to reduce the initialisation costs of a good subgraph isomorphism algorithm, and not to provide indexing (although additionally using this information as an index may not hurt, if initialisation is still costly).

Our central message, though, is not just to say that graph database systems must use a better algorithm. Instead, this chapter shows that such systems need to be designed with a better understanding of the empirical hardness of NP-complete problems, that subgraph isomorphism algorithms must not be treated as a black box, and that lessons learned in constraint programming and artificial intelligence must not go unheeded in other domains.

6.6 Conclusion

We have shown how to generate small but hard instances for the non-induced and induced subgraph isomorphism problems, which will help offset the bias in existing datasets. For non-induced isomorphisms, behaviour was as in many other hard problems, but for induced isomorphisms we uncovered several interesting phenomena: there are hard instances far from a phase transition, constrainedness predicts this, and existing general techniques for designing heuristics do not work in certain portions of the parameter space.

When labels were introduced, we saw that VF2 finds some instances hard which other algorithms find easy. We looked at this kind of instance in more detail, and argued that this was due to flaws in VF2's design, rather than an interesting property of NP-completeness. Although inevitably there will be instances that every algorithm finds hard, we see no excuse for an algorithm to exhibit exponential behaviour in the case when there are two red vertices in a pattern and only one in a target.

Unfortunately, we saw that this aspect of VF2's performance has had practical consequences, most notably being the misdesign of graph database systems. By not trying even the simplest constraint programming techniques from the literature, and ignoring all that is known about the empirical hardness of NP-complete problems, members of the graph databases community have misled themselves into believing that extensive research into supporting techniques is important, whereas really all they are doing is working around some but not all of the defects in their choices of subgraph isomorphism algorithms. With this new understanding of what does and does not make subgraph isomorphism hard, it is time for a radical rethink of how graph database systems work.

Chapter 7

Maximum Common Subgraph Problems

So far we have looked at finding a clique in a graph, and then finding an arbitrary pattern. The final set of problems discussed in this thesis involve finding a subgraph which is common to two graphs simultaneously. We illustrate two variants of this problem in Figure 7.1—in both cases we are finding an induced subgraph and maximising the number of vertices selected, but in the second variant the common subgraph must be connected.

Finding a maximum common subgraph is the key step in measuring the similarity or difference between two graphs (Bunke, 1997; Fernández and Valiente, 2001; Kriege, 2015): to determine the difference between two graphs, we find what they have in common, and then take everything left over. Because of this, maximum common subgraph problems frequently arise in biology and chemistry (Ehrlich and Rarey, 2011; Gay et al., 2014; Raymond and Willett, 2002) where graphs represent molecules, but also in applications including computer vision (Combier, Damiand, and Solnon, 2013; D. J. Cook and Holder, 1994), computer-aided manufacturing (Luo et al., 2017), crisis management (Delavallade et al., 2016), deanonymising datasets (Sharad and Danezis, 2013), the analysis of source code (Djoko, D. J. Cook, and Holder, 1997), binary programs (Gao, Reiter, and Song, 2008), and circuit designs (D. J. Cook and Holder, 1994), in malware detection (Park, Reeves, and Stamp, 2013), social network analysis (M. Fang et al., 2015), graph database query explanations (Vasilyeva et al., 2016), and in character recognition problems (Lu, Ren, and Suen, 1991); Shasha, J. T. Wang, and

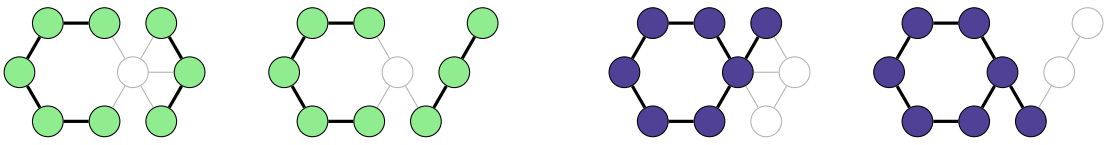


Figure 7.1: A maximum common induced subgraph of the first two graphs has eight vertices, shaded. However, if we require that the common subgraph be connected, only seven vertices may be selected—one way to do this is shown in the third and fourth graphs.

Giugno (2002) give a more detailed review.

The ultimate aim of this chapter is to parallelise a state-of-the-art algorithm for maximum common subgraph problems, and to see whether we agree with Minot, Ndiaye, and Solnon's (2015) conclusion that doing so is unusually challenging. To this end, we must establish what the state of the art is. In Section 7.1, we start by reviewing existing approaches for solving the maximum common subgraph problem. Conventional wisdom is that algorithms based upon constraint programming give the best results, whilst reducing the problem to finding a maximum clique in an association graph is considered weaker. However, previous experimental evaluations of the clique approach have used simple maximum clique algorithms, or even enumeration algorithms—for example, Vismara and Valery (2008) compare a modified form of the Bron and Kerbosch (1973) maximal clique enumeration algorithm with a constraint optimisation approach. Therefore, in Section 7.2, we re-evaluate the clique-based approach using Algorithm 2.1, and see that it outperforms constraint programming on labelled graphs, and that it is competitive with constraint programming on unlabelled graphs, contradicting conventional wisdom.

In Section 7.3 we move on to the maximum common *connected* subgraph problem. There are two ways this could be handled with constraint programming. The traditional approach would be to add a global connectedness constraint to the model. Alternatively, a special branching rule introduced by Vismara and Valery (2008) can be used to grow connected subgraphs only. These two techniques may be combined, and experiments show that the best results are in fact obtained when combining both. When solving the connected problem with a clique-based approach, neither of these techniques seems directly viable with an association graph encoding. However, it is possible to adapt the combined branching and bounding rule used by Algorithm 2.1 to maintain connectedness during search. A comparison of the clique-based approach with the best constraint programming variant for the connected problem shows that it outperforms constraint programming on labelled graphs, but is outperformed by constraint programming on unlabelled graphs.

The experimental results in these first sections present a sobering reality: maximum common subgraph problems might actually be *hard*. In previous chapters we have been working with graphs with thousands of vertices, and now we find pairs of graphs with only 35 vertices each which are challenging for the state of the art. To make matters worse, both the constraint programming and clique approaches are extremely memory-intensive: whilst using $O(|\mathrm{V}(G)|^2\,|\mathrm{V}(H)|^2)$ memory is not a problem for 35 vertex graphs, it is far from feasible when dealing with the subgraph isomorphism instances with which we have worked previously (which may or may not be computationally challenging for the maximum common subgraph problem, but which are too large to fit in RAM with such encoding and propagation overheads). We therefore consider two new ways to tackle these problems.

Section 7.4 introduces a new problem called k-less subgraph isomorphism. For $k = 0$,

this problem is the same as subgraph isomorphism, but as k increases we are allowed to throw away parts of the pattern. This allows us to adapt and weaken the subgraph isomorphism algorithm from Chapter 5, which can work with larger graphs. This is useful in applications such as computer vision, where finding "most of" a pattern graph inside a target graph corresponds closely with an approximate visual match.

Then in Section 7.5 we take a first look at an upcoming algorithm jointly developed with James Trimble and Patrick Prosser, which reimplements the constraint programming approach using different domain-store data structures and filtering algorithms. Although unable to handle certain labelling schemes and side constraints, in cases where it can be used, it is considerably more than an order of magnitude faster than conventional constraint programming, and can operate comfortably on graphs with thousands of vertices without memory problems.

Finally, Section 7.6 introduces and evaluates parallel versions of many of these algorithms. As in previous chapters, the parallel algorithms we introduce are clearly and substantially better than their associated sequential versions. Additionally, we demonstrate that parallelism is risk-free, reproducible, and scalable, and that exploiting the relationship between work splitting and value-ordering heuristics remains a useful strategy.

Parts of this chapter are collaborative work. The experimental comparisons in Sections 7.1 to 7.3 have been published by McCreesh, Ndiaye, et al. (2016) as "Clique and Constraint Models for Maximum Common (Connected) Subgraph Problems"; the constraint programming implementations are due to Ndiaye and Solnon, and the clique implementation and the clique-inspired algorithm for the connected problem are the author's own. The algorithm discussed in Section 7.4 was published by Hoffmann, McCreesh, and Reilly (2017) as "Between Subgraph Isomorphism and Maximum Common Subgraph", and the theoretical results are joint work with those co-authors (the implementation is the author's own). The ideas, algorithm, and implementation discussed in Section 7.5 are to appear by McCreesh, Prosser, and Trimble (2017) as "A Partitioning Algorithm for Maximum Common Subgraph Problems", and are collaborative work with the algorithm being designed and implemented by Trimble; the experimental evaluation was performed by the author. The parallel implementations discussed in Section 7.6 are all the author's own; the parallel splitting algorithm is an adaptation of Trimble's code with the author's parallelisation functions, not a from-scratch implementation.

7.1 Background

Given two graphs G and H, a common subgraph is a graph C together with two induced subgraph isomorphisms $C \hookrightarrow G$ and $C \hookrightarrow H$. The *maximum common subgraph* problem is to find a C with as many vertices as possible. An alternative formulation, which gives a more convenient constraint programming model, is to find a largest-possible subset $C \subseteq \mathrm{V}(G)$,

together with an induced subgraph isomorphism $G[C] \hookrightarrow H$. Notice that we speak only of *induced* isomorphisms and subgraphs. In the non-induced case, we could simply pick every vertex from the smaller graph, and none of the edges. A non-induced variant of the problem where we must maximise the number of edges selected, rather than the number of vertices, is usually called the *maximum common partial subgraph* problem.

There are two competitive approaches for solving the maximum common subgraph problem. The first approach (described in Section 7.1.1) is based on constraint programming, whilst the second (described in Section 7.1.2) involves a reduction to the maximum clique problem. Both approaches are described for undirected, unlabelled graphs; their extension to richer graphs is discussed in Section 7.1.3. Other approaches have been tried, including mixed integer programming (Bahiense et al., 2012; Piva and de Souza, 2012) and inexact methods (Englert and Kovács, 2015); we saw in Chapter 6 that Boolean satisfiability encodings struggle even for subgraph isomorphism. Some special cases also have practical polynomial time algorithms (Droschinsky, Kriege, and Mutzel, 2016; Droschinsky, Kriege, and Mutzel, 2017).

7.1.1 Constraint Programming Models

McGregor (1982) proposed a branch and bound algorithm: each branch of the search tree corresponds to the matching of two vertices, and a bounding function evaluates the number of vertices that still may be matched so that the current branch is pruned as soon as this bound becomes lower than the size of the largest known common subgraph. Krissinel and Henrick (2004) refined McGregor's (1982) algorithm with a better search strategy. Constraint programming approaches may be viewed as enhancements of these branch and bound algorithms.

Vismara and Valery (2008) introduced the first explicit constraint programming model. Given two graphs G and H, this model associates a variable D_v with every vertex v of G, and the domain of this variable contains all vertices of H, plus an additional value $\perp$: variable D_v is assigned to $\perp$ if vertex v is not matched to any vertex of H; otherwise D_v is assigned to the vertex of H to which it is matched. Edge constraints ensure that variable assignments preserve edges and non-edges between matched vertices:

$$\forall u, v \in \mathrm{V}(G), (i(u) = \perp) \vee (i(v) = \perp) \vee ((u, v) \in \mathrm{E}(G) \Leftrightarrow (i(u), i(v)) \in \mathrm{E}(H)),$$

where $i(v)$ represents the value assigned to variable D_v. Difference constraints ensure that each vertex of H is assigned to at most one variable, i.e.

$$\forall u, v \in \mathrm{V}(G) \text{ distinct}, (i(u) = \perp) \vee (i(v) = \perp) \vee (i(u) \neq i(v)).$$

This constraint programming model was improved by Ndiaye and Solnon (2011) by replac-

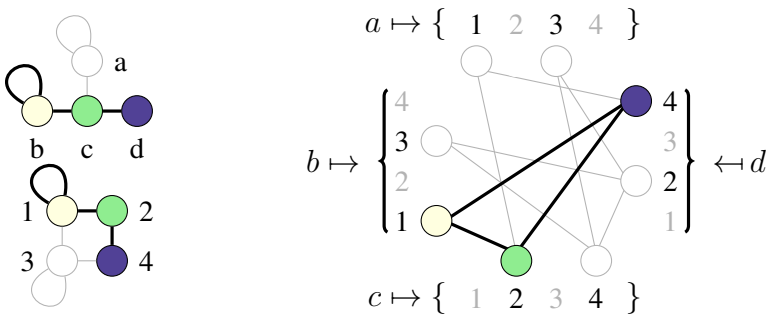


Figure 7.2: A maximum common induced subgraph between the two graphs on the left has three vertices—one solution is highlighted. On the right, the association graph encoding: the highlighted clique of size three shows the same solution. The "missing" vertices correspond to assignments which are impossible due to the presence or absence of loops.

ing binary difference constraints with a soft global all-different constraint which maximizes the number of D_u variables that are assigned to values different from $\perp$, while ensuring they are all different when they are not assigned to $\perp$. They found that maintaining arc consistency (Sabin and Freuder, 1994) on the edge constraints obtains the best results on labelled graphs, whilst forward checking obtains the best results on unlabelled graphs, and both outperform prior approaches. The bound used in both cases uses a matching algorithm to check whether it is possible to assign distinct values to enough D_u variables to surpass the best cost found so far—this is a weaker version of Petit, Régin, and Bessière's (2001) global arc consistent soft all-different constraint.

7.1.2 Reformulation to a Maximum Clique Problem

An alternative approach to solving the maximum common subgraph problem is to reduce the problem to finding a maximum clique in an association graph (Balas and C. S. Yu, 1986; Durand et al., 1999; Levi, 1973; Raymond and Willett, 2002). An association graph (or compatibility graph, or weak modular product) of two graphs G and H is an undirected graph $G \triangledown H$ with vertex set

$$\mathrm{V}(G \triangledown H) = \{(v, v') \in \mathrm{V}(G) \times \mathrm{V}(H) : (v, v) \in \mathrm{E}(G) \Leftrightarrow (v', v') \in \mathrm{E}(H)\}.$$

To avoid confusing vertices of $G \triangledown H$ with vertices of the two original graphs, we call vertices of $G \triangledown H$ *matching nodes*, as each vertex (u, u') of $G \triangledown H$ denotes the matching of u with u'. The edges of $G \triangledown H$ connect matching nodes which denote compatible assignments, so two matching nodes (u, u') and (v, v') are adjacent if $u \neq v$ and $u' \neq v'$, and if they preserve both edges and non-edges, so $(u, v) \in \mathrm{E}(G) \Leftrightarrow (u', v') \in \mathrm{E}(H)$. We illustrate this in Figure 7.2.

A clique in an association graph corresponds to a set of compatible matchings. Therefore, such a clique corresponds to a common subgraph, and a maximum clique of $G \triangledown H$ is a

maximum common subgraph of G and H. It follows that any method able to find a maximum clique in a graph can be used to solve the maximum common subgraph problem.

Note that the association graph is a subgraph of the microstructure (Jégou, 1993) associated with the constraint programming model of Vismara and Valery (2008): the microstructure has more matching nodes than the association graph because it has a matching node $(u, \perp)$ for each vertex u of G. Each clique of size $|\mathrm{V}(G)|$ in the microstructure corresponds to a common subgraph, the size of which is defined by the number of matching nodes that do not contain $\perp$.

7.1.3 Extension to Labelled or Directed Graphs

In some applications, labels may be associated with vertices or edges. We denote by $\ell(u)$ and $\ell((u, v))$ the label of a vertex u and an edge (u, v), respectively. Where graphs are labelled, any isomorphism f must additionally preserve labels, so we require $\ell(f(v)) = \ell(v)$ for any vertex v, and $\ell((f(u), f(v))) = \ell((u, v))$ for any edge (u, v). This kind of label compatibility constraint is handled in a straightforward way in both constraint programming and clique-based approaches. For constraint programming, we restrict the domain of every variable D_u to vertices with compatible labels, and ensure that edge labels are preserved in edge constraints. For clique-based approaches, label compatibility is handled through the definition of the association graph, by restricting the set of matching nodes to pairs of vertices with compatible labels, and the set of matching edges to pairs of edges with compatible labels. The extension to directed graphs, where isomorphisms must preserve directed edges, is similarly straightforward.

Labels and directed edges usually simplify the solution process, both for constraint programming and clique-based approaches: vertex labels reduce domain sizes for constraint programming, and the number of matching nodes in association graphs; edge labels tighten edge constraints for constraint programming, and make the association graph sparser for clique-based approaches. It is worth noting that edge constraints do not help constraint programming approaches to do more filtering so long as $\perp$ remains in variable domains: every pair of variables (D_i, D_j) having $\perp \in D_j$ is arc consistent, since $\perp$ is a support for any value $u \in D_i$. However, as soon as $\perp$ is removed from domains (i.e. when the number of variables assigned to $\perp$ has reached the best known bound on the size of the solution), maintaining arc consistency may filter values, and then tighter constraints increase the opportunities for filtering.

7.2 Re-Evaluating the Clique Model

Previous experimental evaluations of the association graph model have used either maximal clique enumeration algorithms, even when the maximisation problem was being considered (Koch, 2001; Vismara and Valery, 2008), or very simple maximum clique algorithms (Bunke et al., 2002; Conte, Foggia, and Vento, 2007). As a result, their conclusions may now be overly pessimistic. Thus we re-evaluate the approach using a modern maximum clique algorithm. Association graphs are dense, even if the input is sparse, so we will be using our C++ implementation of Algorithm 2.1. We compare this to the "FC+Bound" and "MAC+Bound" (simply referred to as FC and MAC) constraint programming implementations of Ndiaye and Solnon (2011). Experiments are performed on machines with Intel Xeon E5-2640 v2 CPUs and 64GBytes RAM running Ubuntu 14.04; software was compiled using GCC 5.3.0, and a timeout of one thousand seconds was used.

We work with a randomly generated database (Conte, Foggia, and Vento, 2007; De Santo et al., 2003) commonly used for benchmarking maximum common subgraph problems. The dataset contains different classes of graphs: randomly connected graphs with different densities; 2D, 3D, and 4D regular and irregular meshes; regular bounded valence graphs, and irregular bounded valence graphs. For each pair of graphs, there are 3 different labellings such that the number of different labels is approximately equal to 33%, 50% or 75% of the number of vertices. These experiments will look at unlabelled graphs (labels are ignored), and with 33% labellings either just on vertices, or on both vertices and edges (the problem becomes very easy with larger numbers of labels). We select the first ten instances from each class. For unlabelled graphs, we further restrict our tests to the graph pairs where the number of vertices in each graph is no more than 50, for a total of 4,110 pairs; for labelled graphs, which we find less computationally challenging, we select all 8,140 graph pairs, to include graphs with up to 100 vertices.

Many of the criticisms from Chapter 6 also apply to this dataset. However, because we are dealing with an optimisation problem rather than a decision problem, and because the dataset includes instances with a wide range of solution sizes, the situation is not so dire—indeed, solution size stands out as being the best indicator of how hard instances are in practice. Also, this dataset is challenging, so we are not in danger of jumping to overly-optimistic conclusions about what maximum common subgraph algorithms are capable of.

The left-hand plots of Figure 7.3 display the cumulative number of instances solved with respect to time. When graphs are labelled, either just with vertex labels or with both vertex and edge labels, the clique-based approach clearly outperforms either constraint programming model, and MAC has a slight advantage over FC. (Recall that edge labels decrease the density of the association graph, which is typically very beneficial for clique algorithms, but do not help constraint programming until $\perp$ is removed from domains. Vertex labels help both models.) For unlabelled graphs, the three approaches are broadly comparable, and ultimately

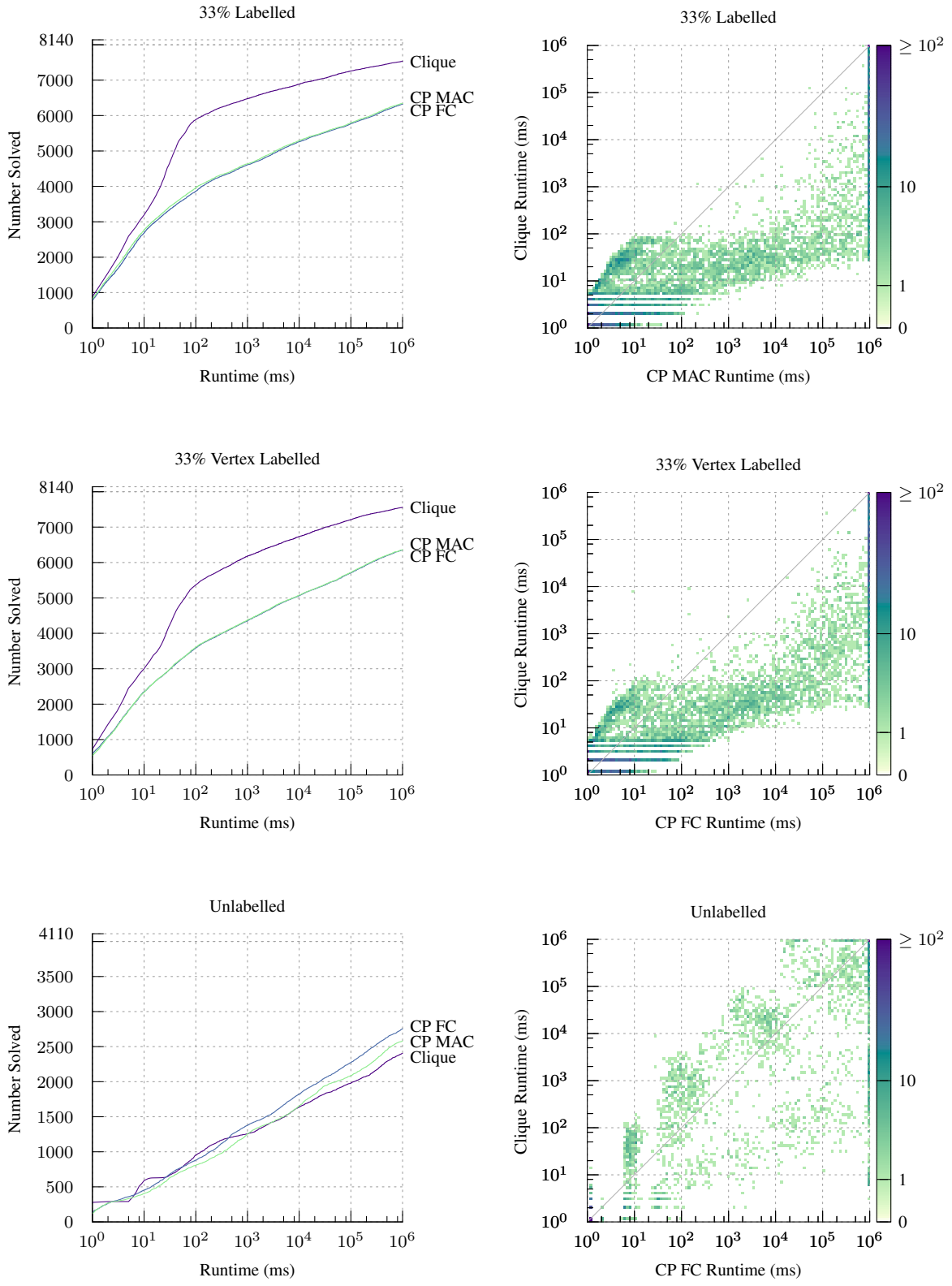


Figure 7.3: On the left, cumulative number of instances solved in under a certain time. On the right, comparisons between the clique model and the best constraint programming model.

FC beats MAC, which beats the clique approach. The right-hand column gives a per-instance comparison of the best constraint programming approach with the clique approach: the heatmaps are similar to scatter plots, but due to the large number of instances, we colour each point according to the density of solutions around that point. For labelled graphs, the clique approach comes close to dominating MAC on non-trivial instances (which suggests that there is unlikely to be scope for per-instance algorithm selection here). For unlabelled graphs, there is still a broad correlation between the runtimes; the clique approach rarely wins by more than one order of magnitude, but is sometimes much worse.

A closer inspection of the data suggests that the different randomness models used to generate instances have little effect on the runtimes for either approach. However, the relative size of the solution matters, particularly for the clique algorithm: if the solution is large (i.e. the two input graphs are very similar), the clique approach finds nearly every labelled instance trivial.

7.3 Maximum Common Connected Subgraphs

Sometimes we seek a common subgraph which must satisfy additional constraints. This is usually handled by propagation in a constraint programming setting. For clique-based approaches, some constraints may be implemented by modifying the definition of the association graph—for example, constraints on pairs of vertices that may be matched are handled by removing inconsistent pairs from $\mathrm{V}(G \triangledown H)$. However, non-decomposable global constraints cannot usually be handled by modifying the association graph.

In this section, we focus on the connectedness constraint, which occurs in many applications (Ehrlich and Rarey, 2011; Koch, 2001; Luo et al., 2017; Raymond and Willett, 2002; Vismara and Valery, 2008). Adding the connectedness requirement makes certain special cases solvable in polynomial time, including outerplanar graphs of bounded degree (Akutsu and Tamura, 2013) and trees (Droschinsky, Kriege, and Mutzel, 2016), but the general case remains NP-hard. As illustrated in Figure 7.1, the maximum common connected subgraph cannot be deduced from the maximum common subgraph: we need to ensure connectedness during search. Section 7.3.1 shows two ways this may be done in constraint programming, and Section 7.3.2 evaluates both options. Then, in Section 7.3.3, we introduce a new way of ensuring connectedness in a clique-based approach. Finally, we compare constraint programming and the clique-based approach in Section 7.3.4.

For the connected problem we consider only undirected graphs (and so directed edges in the inputs are treated as being undirected). For directed graphs, there is more than one notion of connectivity, and it is not clear which should be selected—the approaches we discuss extend easily to weakly connected directed graphs, but not to the strongly connected case (for which no applications are known).

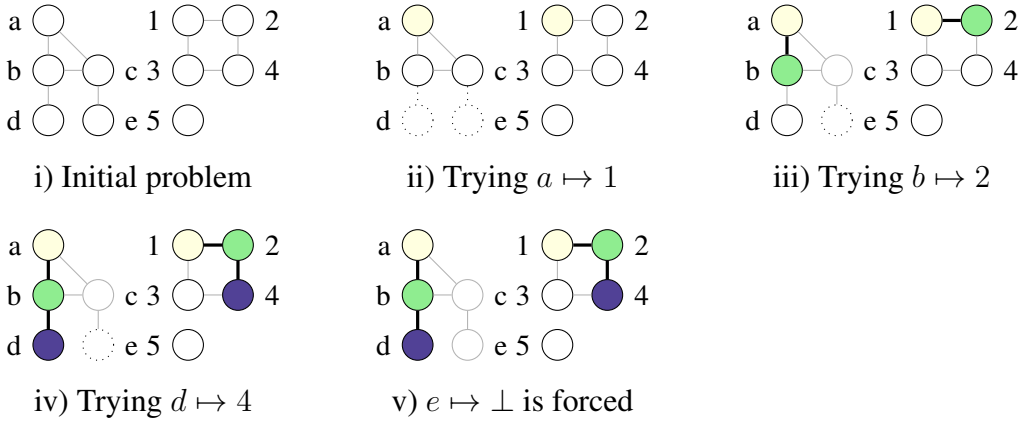


Figure 7.4: Suppose we are looking for a connected common subgraph, using the graph on the left for variables and the graph on the right (which has an isolated vertex) for values. We initially consider $a \mapsto 1$. Our restricted branching rule requires us to select either variable b or variable c subsequently, not d or e. We try $b \mapsto 2$, which adds d to the branchable variables, and forces $c \mapsto \bot$. We may now only branch on d, and we try $d \mapsto 4$. Now the only remaining variable is unbranchable, and so $e = \bot$ is forced, even though 5 remains in its domain and does not violate any constraints.

7.3.1 Ensuring Connectedness with Constraint Programming

Vismara and Valery (2008) implemented the connectedness constraint by using a branching rule which selects the next variable to be assigned. Let A be the set of variables already assigned to values different from $\bot$. The next variable to be assigned is chosen from the set of unassigned variables which are adjacent to at least one vertex of A. When this set is empty, all remaining unassigned variables are assigned to $\bot$. We illustrate this in Figure 7.4.

A more traditional constraint programming approach would be to express connectedness as a conventional constraint. For example, Dooms, Deville, and Dupont (2005) introduced graph domain variables and enforce connectivity via a *reachable* constraint, ensuring that there is a path from a specified vertex to a specified set of vertices. One such constraint could be posted for each of the vertices in the graph, encoding the transitive closure of the graph. Brown et al. (2005) explored the use of constraint programming in the generation of connected graphs with specified degree sequences. Two constraints were combined: the graphical constraint (a backtrackable implementation of the Havel-Hakimi algorithm), and a connectivity constraint implemented using sets of vertices, where vertex sets A and B are combined when there exists a pair of vertices $v \in A$ and $w \in B$ and an edge $(v, w) \in E$. Residual degree counts are maintained on components and vertices to enforce graphicality and connectivity. Prosser and Unsworth (2006) proposed a connectivity constraint for connected graph generation where decision variables are edges (the search process accepts and rejects edges). The constraint employed depth first search to maintain the set of tree edges and back edges, associating path counters on these edges. The counters were then used to detect the existence of cut-edges and protects these by forcing edges.

In all these previous works, the goal was to ensure that a given set of vertices is connected. Here the problem is slightly different: we have to ensure that the number of connected vertices that may be matched (in both graphs) is greater than the size of the largest common subgraph previously found. Therefore, we introduce a new filtering algorithm to ensure connectedness consistency. Let us consider two graphs G and H, and let D be the current domains (we suppose that D_u is a singleton when u is assigned). Let S and T be the sets of vertices of G and H respectively which may belong to the common subgraph, i.e. $S = \{u \in \mathrm{V}(G) : D_u \neq \{\bot\}\}$, and $T = \cup_{u \in \mathrm{V}(G)} D_u - \bot$. Connectedness consistency ensures that both $G[S]$ and $H[T]$ are connected graphs.

Connectedness consistency is ensured only once a first variable has been assigned, rather than at the root of search. Let D_u be the first assigned variable, and v the value assigned to D_u. To ensure connectedness consistency, we perform a traversal of G (respectively, H), starting from u (respectively, v), and we initialize S (respectively, T) with all visited vertices. Then, for each vertex $v \in \mathrm{V}(G) \setminus S$, we set D_v to $\bot$, and for each $w \in \mathrm{V}(H) \setminus T$, we remove w from all domains to which it belongs.

During search, each time a variable is assigned to $\bot$, we remove the corresponding vertex from S and perform a new traversal of $G[S]$ starting from the initial vertex u. For each vertex $w \in S$ that is not visited by the traversal, we remove w from S and assign D_w to $\bot$. Also, each time a value is removed from a domain so that this value no longer belongs to any domain, we remove it from T, and perform a new traversal of $H[T]$ starting from the initial vertex v. For each vertex w that is not visited by the traversal, we remove w from T, and remove w from all domains to which it belongs.

Finally, the two approaches for ensuring connectedness (branching and filtering) are complementary and may be combined: at each step of the search, we select the next variable to be assigned within the neighbors of A, and each time a vertex of H is removed from a domain we filter domains to ensure connectedness consistency. In the example in Figure 7.4, after the first assignment, filtering alone would remove 5 from every domain but would allow branching on any remaining variable, whilst branching alone would force the next variable to be either b or c but would not immediately eliminate 5 from the domains of d and e.

7.3.2 Comparison of Connectedness Techniques

Figure 7.5 compares the three approaches for ensuring connectedness in constraint programming: by branching, by filtering, or by combining both branching and filtering. Results are shown only using the best variant for each class—that is, MAC for labelled graphs, and FC for unlabelled graphs (the other results are very similar). On labelled graphs, we see many instances which are solved very quickly by branching but not at all by filtering, and vice versa. However, combining both is rarely much worse than just doing one or the other, and is often much better, even if on average it is slightly slower. On unlabelled graphs, the three variants

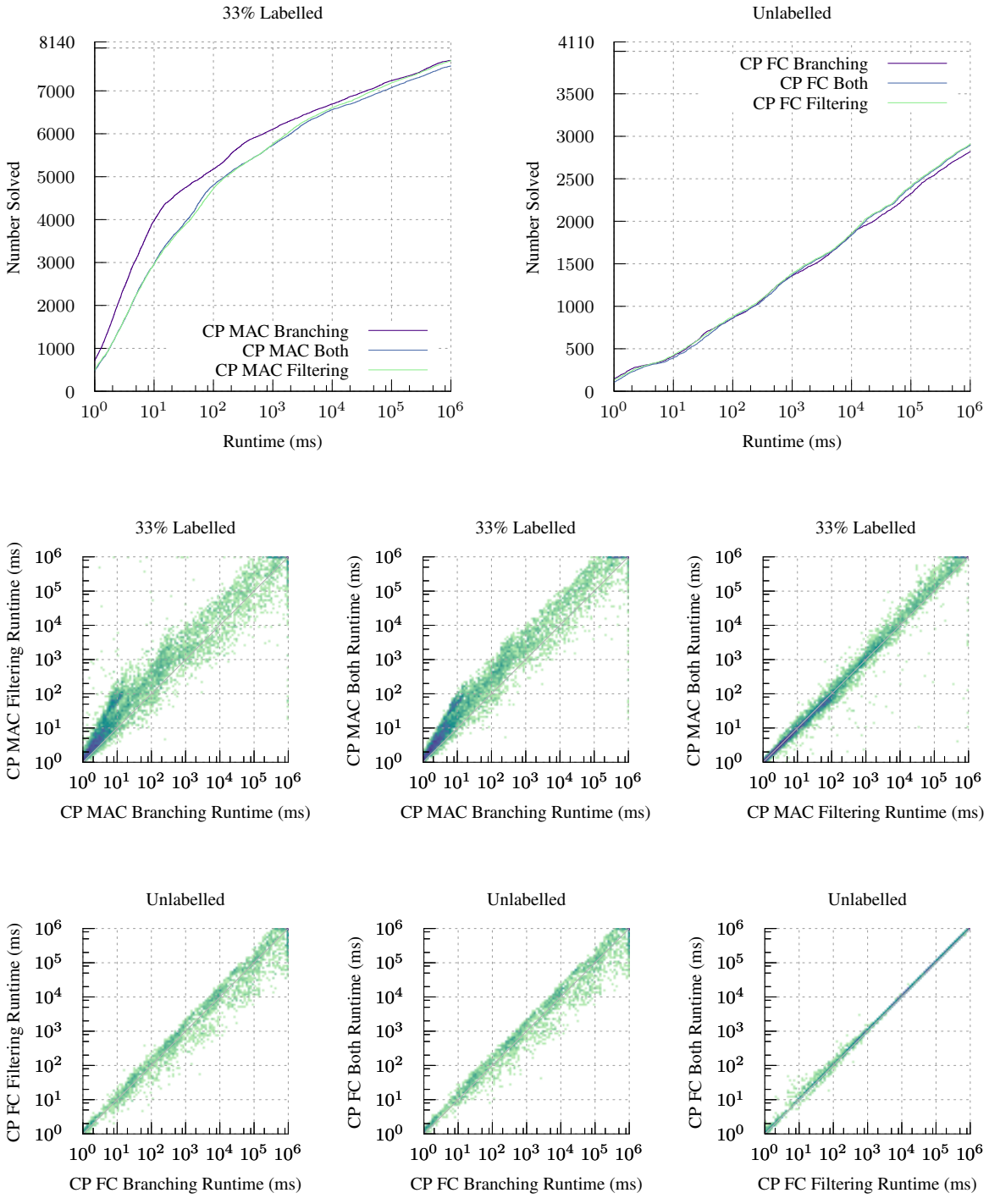


Figure 7.5: On top, the cumulative number of connected instances solved in under a certain time using different constraint programming techniques, for 33% labelled (left) and unlabelled undirected (right) graphs. Below, instance-by-instance comparisons.

Algorithm 7.1: An algorithm for a maximum common connected induced subgraph isomorphism via an association graph.

```
1  associationMCCIS :: (Graph G_1, Graph G_2) → Map
2  begin
3      global incumbent ← ∅
4      G ← G_1 ▽ G_2
5      search(G, ∅, ∅, V(G))
6      return incumbent
7  search :: (Graph G, Set solution, Set connected, Set remaining)
8  begin
9      colourClasses ← concatenate(
           colourOrder(G, remaining \ connected),
           colourOrder(G, remaining ∩ connected))
10     while length(colourClasses) > 0 do
11         foreach v ∈ last(colourClasses) in reverse order do
12             if |solution| + length(colourClasses) ≤ |incumbent| then return
13             if v ∉ connected ∧ solution ≠ ∅ then return
14             solution' ← solution + v
15             if |solution'| > |incumbent| then incumbent ← solution'
16             connected' ← connected ∪ {w ∈ G : first(w) ∈ N(G, first(v))}
17             remaining' ← remaining ∩ N(G, v)
18             if remaining' ≠ ∅ then search(G, solution', connected', remaining')
19         removeLast(colourClasses)
```

have rather similar performance.

7.3.3 Ensuring Connectedness in a Clique-Based Approach

It is not possible to determine connectedness from a raw association graph. Nor is it possible to encode connectedness by modifying the association graph encoding. However, we can take a maximum clique algorithm and mimic the constraint programming branching strategy if we have access to the underlying graphs and can determine the "meaning" of the association graph vertices.

This is not straightforward. Recall that Algorithm 2.1 uses a greedy graph colouring both as a bound and as a branching heuristic: vertices are selected in reverse order from their colour classes in turn, starting with the last colour class created. Because of this coupling of branching and the bound (which we saw in Chapter 2 was important in practice), if we were to select only a subset of vertices for branching at each stage inside a clique algorithm, we would lose completeness. Thus we must adapt the bound in a non-trivial way to take into account restricted branching.

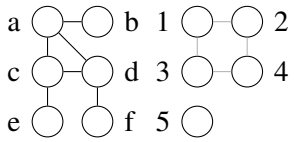
In Algorithm 7.1 we introduce a novel clique-inspired algorithm which finds a maximum

common connected induced subgraph via an association graph. If the additional branching restrictions are removed, the core of the algorithm is the same as Algorithm 2.1. The way we extend this algorithm for connectedness differs considerably from that of Koch (2001) and Vismara and Valery (2008): these earlier approaches worked by classifying labels in the association graph based upon whether a common vertex is shared, and then constructing cliques with particular edge properties—this is harder to integrate with a strong bound function.

The algorithm begins by building the association graph (line 4). The main part of the algorithm then works by building up candidate cliques in the $solution$ variable, by recursive calls to the `search` procedure—starting from the empty set (line 5), each recursive subcall adds one vertex to $solution$ (line 14) in such a way that $solution$ is always a clique which corresponds to a connected common subgraph. The $remaining$ set contains the set of vertices which are adjacent to every vertex in $solution$, and which have not yet been accepted or rejected (and so initially it contains every vertex). The main loops in the `search` procedure (lines 10 and 11) have the effect of iterating over each vertex in this set in a particular order—each vertex v is selected in turn, and then a recursive call is made to consider the effects of including v in $solution$ (line 18), followed by the next iteration where v is instead rejected. When v is accepted, we add it to the new $solution'$ (line 14), and create a new $remaining'$ containing only the vertices in $remaining$ which are adjacent to v (line 17).

The $connected$ set contains the set of matching nodes which correspond to vertices adjacent to an already-accepted vertex in the first input graph—in constraint programming terms, it is the set of assignments which could be made next which maintain connectedness. (Using only one of the two input graphs is sufficient for correctness, and has the advantage that the connectedness set may be determined by a simple lookup into a precomputed array which maps each vertex in the first input graph to a bitset.) At the top of search, this set is empty, and is not used (our first vertex selection is special, and does not care about connectedness). At subsequent depths, we may only accept vertices which are in this set, and if no such vertices remain then we return immediately (line 13). When recursing, we extend $connected$ with the new vertices permitted by our acceptance of the branching v (line 16). Note that we are assuming that inside the main loops, we encounter every vertex in $remaining \cap connected$ before any vertex in $remaining \setminus connected$.

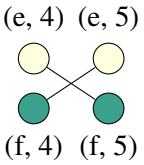
As we proceed, we keep track of the best solution we have found so far—this is stored in the $incumbent$ variable (lines 3 and 15). We use the incumbent, together with a colour bound, to prune portions of the search space which cannot contain a better solution. The colour bound operates as follows: at each entry to the `search` procedure, we produce a greedy colouring of the vertices in $remaining$ (line 9, discussed further below). This greedy colouring gives us a list of colour classes, each of which is a list of pairwise non-adjacent vertices. The two loops (lines 10 and 11) then iterate over each colour class, from last to first, and then over



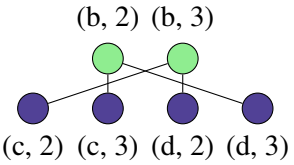
i) Initial problem

$$\begin{array}{ll} \textit{solution} & \{(a,1)\} \\ \textit{connected} & \{b,c,d\} \times \{1\ldots5\} \\ \textit{remaining} & (\{b,c,d\} \times \{2,3\}) \cup (\{e,f\} \times \{4,5\}) \end{array}$$

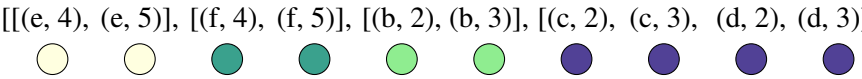
ii) Search variables after guessing $a \mapsto 1$



iii) $\textit{remaining} \setminus \textit{connected}$



iv) $\textit{remaining} \cap \textit{connected}$



v) The resulting $\textit{colourClasses}$ variable.

Figure 7.6: Solving a maximum common connected problem using an association graph. Suppose we have already mapped vertex a to vertex 1, giving the assignments on the right. Now we have two subgraphs to colour. We need two colours for $\textit{remaining} \setminus \textit{connected}$, and we place these two colour classes first in the $\textit{colourClasses}$ variable. We can also colour $\textit{remaining} \cap \textit{connected}$ using two colours, since we cannot simultaneously map c to 2 and d to 3, or vice-versa. Thus $\textit{colourClasses}$ becomes a list of four colour classes. This tells us that if we hope to extend the current common subgraph by another four vertices, we must pick one assignment from each of the four colour classes (which is not actually possible, so the bound here gives an overestimate). The algorithm thus guesses $d \mapsto 3$ as its next assignment, and if that fails, $d \mapsto 2$, and so on; once $b \mapsto 3$ is reached, the bound decreases by one, and if $f \mapsto 5$ were reached we would stop due to a lack of remaining connected association nodes.

each vertex in that colour class, again from last to first. (This should use a pair of immutable flat arrays, rather than actually using a list of lists and removing items, as in Algorithm 2.1.) Finally, if at any point the number of remaining colour classes plus the number of vertices currently present in $\textit{solution}$ is not strictly greater than the size of the incumbent, then we may backtrack immediately (line 12).

Finally, we describe the colouring process—an example is shown in Figure 7.6. We cannot simply reuse the `colourOrder` function from Algorithm 2.1: the colourings it produces will not give us the required property that vertices in $\textit{remaining} \cap \textit{connected}$ come last (so they are selected first by the reverse branching order). Thus we produce *two* greedy sequential colourings (which we view as a list of colour classes, each of which contains a list of vertices), first considering the non-branching vertices in $\textit{remaining} \setminus \textit{connected}$, followed by the branching vertices, and concatenate them (line 9). This produces a valid colouring, since we do not merge any colour classes between the two stages, although it may use more

colours than a single colouring would. As before, we produce greedy sequential colourings, and we use a simple static degree ordering at the top of search. It is possible that special properties of the association graph could be exploited to improve this step—for example, it is always possible to colour the initial association graph using $\min(|\mathrm{V}(G_1)|, |\mathrm{V}(G_2)|)$ colours, but with certain vertex orderings, a greedy sequential colouring will sometimes use many more colours.

(What if we did not guarantee that vertices in $remaining \cap connected$ came last, and just used a conventional colouring with the branching rule? Suppose we had four vertices in $remaining$, and produced a colouring $[[v_1, v_2], [v_3], [v_4]]$, and suppose that extending $solution$ with $\{v_1, v_3, v_4\}$ gives an optimal solution. If v_4 was not $connected$ yet, we would not branch on that subtree, and the bound could eliminate branching on v_3 and v_1, so we would miss the solution. Thus we cannot simply add the branching rule without also adapting the combined bound and ordering heuristic.)

7.3.4 Comparison of the Two Approaches

In Figure 7.7 we compare the clique-based approach to the connected problem with the two CP (with both branching and filtering) approaches. The trend is broadly similar to the unconnected problem: for labelled graphs, the clique-based approach is the clear winner, but for unlabelled graphs the clique approach lags somewhat.

The heatmaps show a more detailed picture. As before, in the unlabelled case, the association approach is almost never more than an order of magnitude better, and is often much worse. In the labelled case, however, there are now a number of instances (along the top of the heatmap) where the constraint programming approach does much better than the association approach, despite the association approach remaining much better overall. (With Chapter 3 in mind, it might be tempting to suggest that these could be instances where the clique branching rule behaves particularly poorly at the top of search, and that parallel search might make them go away. Unfortunately, although this is true for a small number of these instances, experimental evidence does not support such a simple explanation in general.)

7.4 k-Less Subgraph Isomorpism

Compared to previous chapters, the graphs with which we have worked for maximum common subgraph problems so far are small: we have been limited to 100 vertex graphs, and there are instances with only 35 vertices which we cannot solve. Both the clique encoding and the constraint programming inference algorithms are extremely memory-intensive, and larger instances cannot fit in memory. This is unfortunate: suppose a subgraph isomorphism between a pattern graph and a target graph does not exist. We may wish to find “as much as possible”

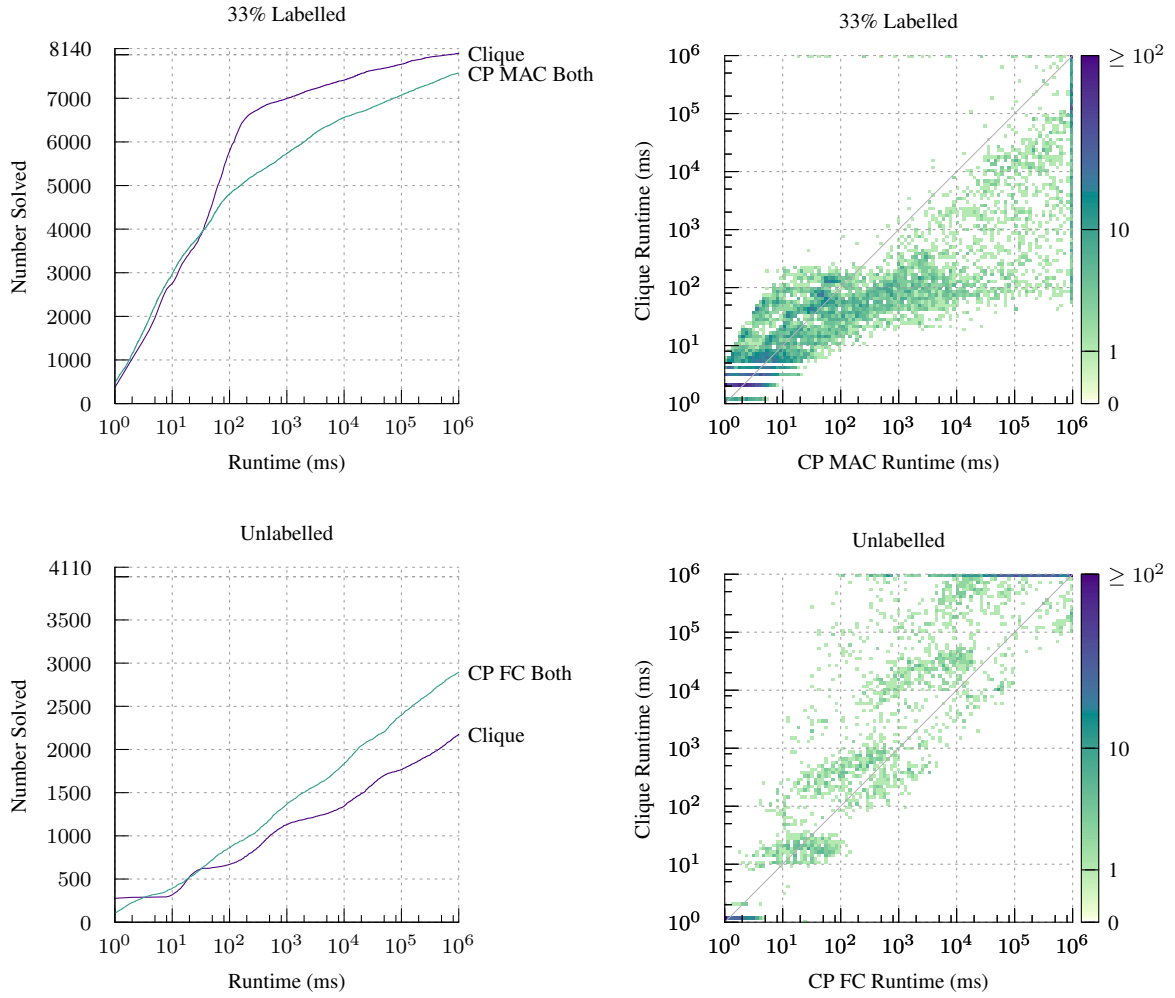


Figure 7.7: The cumulative number of connected instances solved in under a certain time: on the left, 33% labelled undirected graphs with up to 100 vertices, and then unlabelled and undirected graphs with up to 35 vertices. On the right, an instance-by-instance comparison of the association and CP (with both branching and filtering) approaches, with 33% labelled graphs on top, and unlabelled and undirected graphs below.

of the pattern graph inside the target. In the induced case, this is precisely the maximum common subgraph problem (we discuss the non-induced case below). With that in mind, this section discusses a new way of looking at the maximum common subgraph problem which allows us to work with much larger graphs.

This new perspective is as follows: say we are given a pattern graph and a target graph, then we must find a way to map all but k vertices of the pattern graph into the target. We show an example in Figure 7.8. This in some ways resembles the approximate subgraph matching model of Zampelli, Deville, and Dupont (2005), although we allow any vertex to be removed. When k is reasonably small (say, between 1 and 5), weakened forms of the degree- and path-based filterings which we introduced in Chapter 5 are still effective in pruning the initial search space and in providing additional constraints respectively. Additionally, combining these techniques leads to a practical algorithm which can scale to work with the families of

Figure 7.8: The pattern graph on the left cannot be found in the target graph on the right, but if the central vertex in the pattern is removed, then a subgraph isomorphism exists.

graphs we used to benchmark subgraph isomorphism algorithms: depending upon the family, we can close a substantial portion of the instances, and in many more cases, we can at least obtain an upper bound. This is a significant improvement over previous methods discussed in this chapter, which cannot even fit many of these instances in 64GBytes of RAM. Finally, we will see that in the induced case, starting with $k = 0$ and iteratively increasing k gives a competitive algorithm for the maximum common subgraph problem.

7.4.1 Additional Definitions and Notation

Recall that a *non-induced subgraph isomorphism* from a graph P (called the *pattern*) to a graph T (the *target*) is an injective mapping $\mathrm{V}(P) \rightarrowtail \mathrm{V}(T)$ which maps adjacent vertices to adjacent vertices, and that an *induced subgraph isomorphism* $P \hookrightarrow T$ additionally maps non-adjacent vertices to non-adjacent vertices. We define a *k-less subgraph isomorphism* from P to T to be a subgraph isomorphism from all but k vertices of P to T; this may be non-induced or induced, written $P \,{}_k\!\!\rightarrowtail T$ and $P \,{}_k\!\!\hookrightarrow T$ respectively. We write $p \,{}_k\!\!\mapsto t$ to mean that the pattern vertex p is mapped to the target vertex t under either kind of mapping.

Further recall that the *loop complement* of a graph P, written $\overline{P}^\circ$, is the graph where adjacent vertices in P are non-adjacent, non-adjacent vertices in P are adjacent, and vertices in $\overline{P}^\circ$ have loops precisely if they do not in P. The following propositions follow directly from the definitions.

Proposition 7.1. Let i be an assignment of vertices of T to vertices of P. Then i satisfies the definition of $P \hookrightarrow T$ if and only if i satisfies $P \rightarrowtail T$ and $\overline{P}^\circ \rightarrowtail \overline{T}^\circ$ simultaneously. Similarly, i satisfies the definition of $P \,{}_k\!\!\hookrightarrow T$ if and only if i satisfies both $P \,{}_k\!\!\rightarrowtail T$ and $\overline{P}^\circ \,{}_k\!\!\rightarrowtail \overline{T}^\circ$.

Proposition 7.2. An induced k-less subgraph isomorphism $P \,{}_k\!\!\hookrightarrow T$ is equivalent to a common induced subgraph of P and T with $|\mathrm{V}(P)| - k$ vertices.

The non-induced case is different, however. Recall that to avoid the problem of a maximum common non-induced subgraph allowing us to select every vertex in the smaller of the two graphs and none of the edges, it is traditional to change the objective to maximise the number of edges selected. This is not what we will be discussing in this section: maximum common subgraph problems are symmetric in their inputs, but when discussing the non-induced case we are allowing extra edges only in the target graph, not in the pattern.

7.4.2 Constraint Models and Algorithms

There are three easy ways we might try to extend existing algorithms to handle the k-less problem. A non-induced k-less subgraph isomorphism from P to T is equivalent to a subgraph isomorphism between P and T, with k extra universally-adjacent vertices added to T, and so we could try solving subgraph isomorphism with a modified target graph. However, using such an approach is not ideal, because it would introduce symmetries; for induced k-less subgraph isomorphisms, an approach based around adding vertices to T cannot work at all. Another algorithmic approach could be to try each way of removing k vertices from the pattern graph, and solving each subgraph isomorphism problem in turn. This approach might be feasible for $k = 1$, although it would involve a lot of duplication of search effort, but for larger values of k the number of searches which would have to be made would grow with $\binom{k}{\mathrm{V}(P)}$. Finally, for the induced case we could try adapting maximum common subgraph algorithms to solve the decision problem. However, our main aim with this problem is to avoid the prohibitive memory requirements that come with the maximum common subgraph algorithms discussed earlier in this chapter.

Instead, we will discuss a new algorithm, inspired by the subgraph isomorphism algorithm introduced in Chapter 5. This algorithm requires only $O(|\mathrm{V}(P)|^2 |\mathrm{V}(T)|)$ space (which can be thousands of times less than maximum common subgraph approaches, once the constant factors and orders of T in these instances are considered).

7.4.3 Experimental Setup and Instances

We continue to perform our experiments on systems with dual Intel Xeon CPU E5-2640 v2 processors with 64GBytes of RAM. For datasets, we will switch to the 5,725 instances used in Chapter 5 for the subgraph isomorphism problem. Note that many of these instances are *much* larger than the maximum common subgraph instances we have looked at so far: this dataset contains graphs with up to 6,671 vertices. For the constraint programming forward-checking algorithm, 1,560 of the instances cannot fit in the amount of RAM we have available—we treat these instances as having timed out. The situation for the clique encoding is even worse, and 3,653 of these instances do not fit in 64GBytes of RAM. In contrast, for the k-less algorithm, every instance fits comfortably.

7.4.4 Domain Filtering Using Degrees

Let p be a vertex in a graph P. The degree of vertices gives us an invariant, which may be used to eliminate some infeasible values from domains as follows.

Proposition 7.3. Let p be a vertex in P and t a vertex in T. For both non-induced and induced k-less subgraph isomorphisms, if $p \,{}_k\!\mapsto t$ then $\deg(p) - k \leq \deg(t)$.

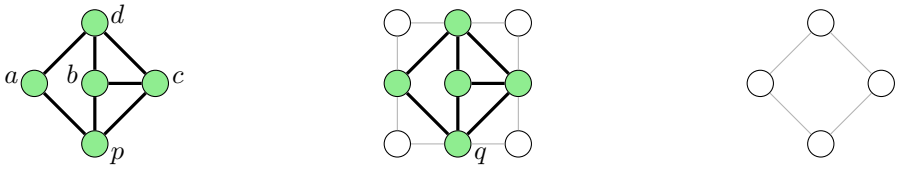


Figure 7.9: There is a non-induced isomorphism from the first graph to the second, using the highlighted vertices. No such isomorphism exists between the first and third graphs, but there is a 1-less non-induced isomorphism which omits vertex b.

Proof. Let p be a vertex in P, and t a vertex in T, with $p \mapsto t$. Then by the definition of subgraph isomorphism, $\deg(p) \leq \deg(t)$. Let P' be P less k vertices and p a vertex in P'. Then

$$\deg_P(p) - k \leq \deg_{P'}(p) \leq \deg_P(p) \leq \deg(t). \qquad \square$$

Recall that the *neighbourhood degree sequence* of a vertex p, $\mathrm{S}(G, p)$, is the (non-ascending) sequence of degrees of its neighbours; in this section we omit G where it is clear from the context. As discussed in Chapter 5, this may be used for filtering in subgraph isomorphism. We extend this for the k-less setting as follows.

Let $S = (s_1, \ldots, s_n)$ and $T = (t_1, \ldots, t_m)$ be two sequences. As in Chapter 5, we say that $S \preceq T$ if $n \leq m$ and $\forall s_i \in S$ there exists a distinct $t_j \in T$ with $s_i \leq t_j$. When considering a k-less subgraph isomorphism we say that $S \,{}_k\!\preceq T$ if $n - k \leq m$, and if there exists some subsequence S_k of S containing up to k members such that $\forall s_i \in S \setminus S_k$, there exists a distinct $t_j \in T$ with $s_i - k \leq t_j$.

Proposition 7.4. If $p \,{}_k\!\mapsto t$, then $\mathrm{S}(p) \,{}_k\!\preceq \mathrm{S}(t)$.

Proof. Let $p \,{}_k\!\mapsto t$. Then $\deg(p) - k \leq \deg(q)$, by Proposition 7.3, which implies that $|\mathrm{S}(p)| - k \leq |\mathrm{S}(t)|$. Also, $p \,{}_k\!\mapsto t$ implies that for each $q \in N(p) \setminus P_k$, where P_k is some subset of the vertices of P with $|P_k| \leq k$, we have $q \,{}_k\!\mapsto u$, where $u \in N(t)$ and each u is distinct. Then $\deg(q) - k \leq \deg(u)$, by Proposition 7.3. Hence $\mathrm{S}(p) \,{}_k\!\preceq \mathrm{S}(t)$. $\square$

For example, consider the pattern graph P and the two target graphs T and U shown in Figure 7.9. The neighbourhood degree sequence of pattern vertex p is $\mathrm{S}(p) = (3, 3, 2)$. We highlight a subgraph in T which is non-induced isomorphic to P. The vertex p can be mapped to q in the target graph T, as $\mathrm{S}(q) = (5, 5, 4, 2, 2)$. There is a non-induced mapping of the pattern graph P into U by removing b of P. This removal changes the neighbourhood degree sequence of p in the k-less version of P to $\mathrm{S}(p) = (2, 2)$.

Figure 7.9 also illustrates the three cases possible when filtering by neighbourhood degree sequence. Removing vertex d causes each entry in $\mathrm{S}(p)$ to be reduced; removing vertex a removes an entry from $\mathrm{S}(p)$; and removing either of vertex b or c causes both the size of $\mathrm{S}(p)$ to be reduced and an entry in $\mathrm{S}(p)$ to be reduced.

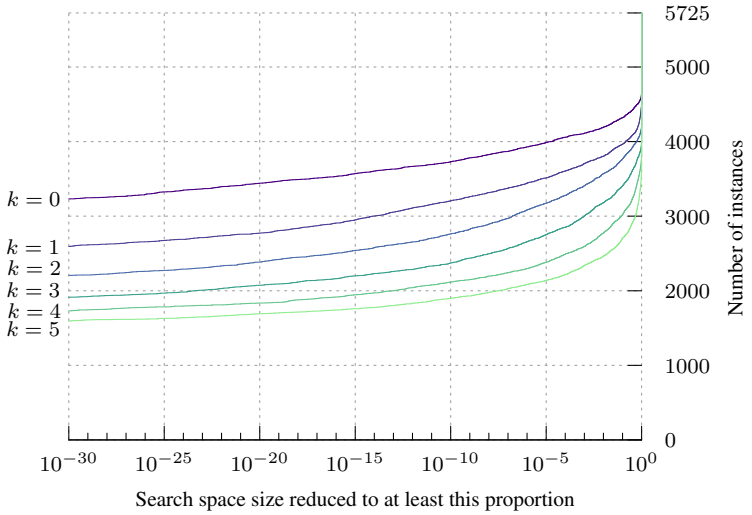


Figure 7.10: The amount of domain reduction achieved for the induced problem, with increasing k. The y value shows for how many instances we may reduce the initial search space to be at most the x proportion of the size the search space would be for the maximum common subgraph problem.

Corollary 7.1. Since both $\mathrm{S}(p)$ and $\mathrm{S}(t)$ are non-ascending, without loss of generality we can replace $S \setminus S_k$ from the definition of $S \preceq_k T$ with the subsequence consisting of all but the first k members of S.

Figure 7.10 demonstrates that Proposition 7.4 is effective in practice: we show the amount of domain reduction which can be achieved at the top of search by using invariants and a fixed k, compared to the search space size for the maximum common subgraph problem. We look at the product of the domain sizes, rather than the number of eliminations, as this better reflects the number of combinations remaining to be considered. The results on the non-induced version show a similar but slightly weaker trend.

Some other invariants do not translate. For example, another rule which can be effective on regular graphs involves counting the number of neighbours of a vertex which are present in a triangle (McKay and Piperno, 2014). Removing a single vertex can alter this count arbitrarily, so we cannot make use of this fact.

7.4.5 Filtering During Search Using Paths

As well as reasoning about degrees, we can also reason about paths. Let $\mathrm{paths}(p, q, n)$ be the number of paths of length n between the vertices p and q.

Proposition 7.5. Let $p, q \in \mathrm{V}(P)$ and $t, u \in \mathrm{V}(T)$ be pairs of pattern and target vertices respectively. If $p \mapsto_k t$ and $q \mapsto_k u$ then $\mathrm{paths}(p, q, 2) - k \leq \mathrm{paths}(t, u, 2)$.

Proof. Let P_2 be the set of all paths of length two between p and q,

$$P_2 = \{((p, x), (x, q)) : (p, x), (x, q) \in \mathrm{E}(P)\}.$$

As we are looking at paths of length 2, the intermediate vertices lie in both neighbour vertex sets of p and q. We can rewrite P_2 as

$$P_2 = \{((p, x), (x, q)) : x \in \mathrm{N}(p) \text{ and } x \in \mathrm{N}(q)\},$$

in other words the intermediate vertices lie in the intersection of the neighbour sets of p and q,

$$P_2 = \{((p, x), (x, q)) : x \in \mathrm{N}(p) \cap \mathrm{N}(q)\}.$$

Therefore $\mathrm{paths}(p, q, 2) = |\mathrm{N}(p) \cap \mathrm{N}(q)| \leq \deg(p)$.

If we remove up to k vertices from the neighbourhood of p, it will impact the intersection of the neighbourhoods of p and q, and by Proposition 7.3, as $p \,{}_k\!\mapsto t$,

$$|\mathrm{N}(p) \cap \mathrm{N}(q)| - k \leq \deg(p) - k \leq \deg(t).$$

As $|\mathrm{N}(p) \cap \mathrm{N}(q)| \leq \deg(p) \leq |\mathrm{N}(t) \cap \mathrm{N}(u)| \leq \deg(t)$, we have

$$\begin{aligned} |\mathrm{N}(p) \cap \mathrm{N}(q)| - k &\leq \deg(p) - k \\ &\leq |\mathrm{N}(t) \cap \mathrm{N}(u)| \\ &\leq \deg(t) \end{aligned}$$

which tells us

$$\mathrm{paths}(p, q, 2) - k \leq \mathrm{paths}(t, u, 2). \qquad \square$$

Corollary 7.2. Recall from Chapter 5 that for a graph G, we define $G^{n,\ell}$ to be the graph with vertex set $\mathrm{V}(G)$, and edges between vertices p and q if there are at least n simple paths of length exactly ℓ between p and q in G. Then any k-less subgraph isomorphism $P \,{}_k\!\rightarrowtail T$ induces a new k-less subgraph isomorphism $P^{n+k,2} \,{}_k\!\rightarrowtail T^{n,2}$ using the same vertex assignments.

Unlike in conventional subgraph isomorphism, we cannot extend this filtering to look at paths of length three: as the example in Figure 7.11 shows, removing a single vertex can delete arbitrarily many such paths. We *could* instead count paths of any length which are vertex disjoint, although calculating this appears to be too expensive to be beneficial in practice.

To allow for fast propagation, rather than calculating paths dynamically like Audemard, Lecoutre, et al. (2014), we follow the approach introduced in Chapter 5 and instead construct *supplemental graphs*, finding a mapping which is simultaneously a non-induced subgraph

Figure 7.11: The pattern graph on the left cannot be found in the target graph on the right, but if one vertex is removed, then a subgraph isomorphism exists.

isomorphism between every supplemental graph pair. We use paths of length 2, looking at whether there are at least 1, 2, and 3 in the target graph (and so whether there are at least $1 + k$ up to $3 + k$ in the pattern). We then investigate whether this leads to new constraints being generated.

By an *assignment*, we mean considering mapping a pattern vertex p to a target vertex t (and not $\perp$) which does not violate any loop constraints. An *assignment pair* is two assignments with distinct p and distinct t, which we say is permitted if it does not violate any adjacency constraint. We define the *permitted assignment pair ratio* to be the proportion of assignment pairs which are permitted. Given this, in Figure 7.12 we scatter plot the permitted assignment pair ratio with and without supplemental graphs. (Because of the large sizes of the domains, we randomly sample one million pairs rather than considering every pair. In some cases, we have nearly a thousand domains, each with nearly ten thousand values—a complete quadratic calculation involving even a trivial arithmetic operation on this would take many hours.)

For $k = 0$, we see many points above the $x - y$ diagonal, which shows that for many instances, a substantial number of new constraints are created at the top of search; on the other hand, there are also points on the diagonal, which shows that sometimes this technique provides no benefit. (Occasionally, points fall below the $x - y$ diagonal. This is because the use of neighbourhood degree sequence reasoning on supplemental graphs can also lead to increased domain filtering, which could in turn eliminate a higher proportion of forbidden than permitted assignment pairs.) For $k = 1$ and $k = 2$, the proportion of points above the diagonal diminishes, but we are still able to create new constraints for many instances. By the time $k = 3$, most of the benefit is disappearing—although sometimes we are still able to make a difference, and bear in mind that sometimes adding just one new constrained pair can vastly reduce the search space.

7.4.6 A New Algorithm

Algorithm 7.2 integrates these techniques into a full algorithm. This is derived from Algorithm 5.1, with additions to handle null values for the k-less case. The algorithm performs a constraint-based search. We have a set D containing a variable D_v for each vertex v in the pattern graph. Each variable has a domain containing one value for every vertex in the target graph. The algorithm is bit-parallel: each D_v is stored using bitset, and all graphs are stored

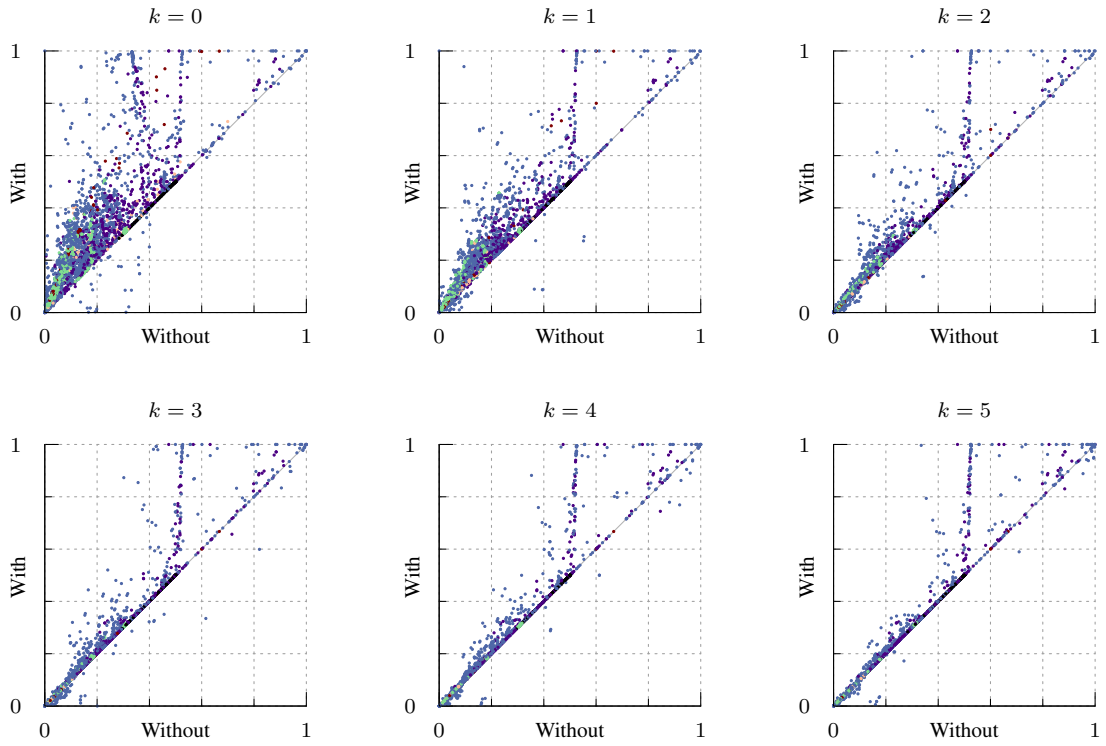


Figure 7.12: For the induced problem, the proportion of pairs of assignments from the filtered domains which are not permitted simultaneously, without path constraints on the x-axis and with path constraints on the y-axis, for increasing values of k. Point colours show instance families, as in Section 5.3.

as adjacency matrices.

In line 8 we use the reasoning from Section 7.4.4 to eliminate infeasible initial values from the domains. We do not use iterated label filtering (Zampelli, Deville, and Solnon, 2010) to recalculate neighbourhood degree sequences if any vertices are not present in any target domain after construction: preliminary experiments indicated that this happened very rarely on these instances.

To handle unmapped vertices, on line 9 we include k additional wildcard $\perp$ values in each domain (rather than a single value which may be used k times).

We begin by trying to infer domain deletions. The `propagate` function looks for domains which contain either only a single value, or only wildcards—we call such a domain *effectively-unit*. If such a domain D_v exists, we eliminate its value from every other domain, and then propagate adjacency: for each domain corresponding to a vertex adjacent to v, we eliminate any value from its domain which is not adjacent to v (treating wildcards as being adjacent to all vertices). If a domain wipeout occurs, we return failure.

We deal with the additional constraints discussed in Section 7.4.5 by constructing supplemental graphs, as in Chapter 5. This is done on line 4: the variable L contains a list of pattern / target pairs, and following corollary 7.2, we will search for a mapping which is simultaneously a subgraph isomorphism between every pair in this list. We also do degree-based reasoning

Algorithm 7.2: A bit-parallel algorithm for the k-less subgraph isomorphism problem.

```
1  klessSubgraphIsomorphism (Graph 𝒫, Graph 𝒯, Int k) → Bool
2  begin
3      if |V(𝒫)| + k > |V(𝒯)| then return false
4      L ← [(𝒫, 𝒯), (𝒫̄, 𝒯̄) only if we want induced,
           (𝒫^{1+k,2}, 𝒯^{1,2}), (𝒫^{2+k,2}, 𝒯^{2,2}), (𝒫^{3+k,2}, 𝒯^{3,2})]
5      foreach v ∈ V(𝒫) do
6          D_v ← V(𝒯)
7          foreach (P, T) ∈ L do
8              D_v ← {w ∈ D_v : v ∼_P v ⇒ w ∼_T w ∧ S_P(v) _k⪯ S_T(w)}
9          D_v ← D_v ∪ k distinct wildcard values
10     if propagate(L, D) then return search(L, {E ∈ D : |E| > 1}, k)
11     else return false

12 search (GraphPairs L, Domains D, Int k) → Bool
13 begin
14     if D = ∅ then return true
15     D_v ← the smallest domain in D
16     foreach v' ∈ D_v ordered by static degree in 𝒯 do
17         if v' is not the first wildcard tried then
               continue
18         D' ← clone(D)
19         D'_v ← {v'}
20         if propagate(L, D') then
21             if search(L, {E ∈ D' : |E| > 1}, k) then return true

22 propagate (GraphPairs L, Domains D) → Bool
23 begin
24     while true do
25         if no effectively-unit domains remain then
26             if not countingAllDifferent(D) then return false
27         if no effectively-unit domains remain then return true
28         D_v ← an effectively-unit domain from D
29         v' ← the single value in D_v, or an arbitrary wildcard value
30         foreach D_w ∈ D − D_v do
31             D_w ← D_w − v'
32             foreach (P, T) ∈ L do
33                 if v ∼_P w then
34                     D_w ← D_w ∩ (N_T(v') ∪ wildcards)
35             if D_w = ∅ then return false
```

using each of these graph pairs.

If no effectively-unit domains remain, we attempt stronger propagation for the all-different constraint. The `countingAllDifferent` function is the bit-parallel propagator used in Chapter 5, and is *not* the usual matching-based propagator (Régin, 1994) which guarantees generalised arc consistency. Because we use multiple wildcard values, we do not need to modify the algorithm to allow a single wildcard value to be used more than once. This propagation could create new effectively-unit domains; if so, we repeat the process.

If propagation is unable to prove unsatisfiability, we search. We pick the smallest domain (line 15) and try giving it each of its remaining values in turn. We use the value ordering heuristic from the original algorithm; wildcards are treated as having degree zero, in an attempt to maximise the expected number of solutions remaining during search (as in Chapter 6). We introduce a symmetry break (line 17) to try only a single wildcard value for each variable.

For the induced case, we make use of Proposition 7.1 (line 4). We considered using path reasoning on complement graphs, but this is expensive to calculate and provides little benefit in practice on these instances. We also do not strip isolated vertices as the original algorithm did, as this is not a valid simplification in the induced case.

7.4.7 Empirical Evaluation

We now evaluate Algorithm 7.2 and show that it is effective in practice, even on the larger subgraph isomorphism instances. In Figure 7.13 we give cumulative distributions for the induced and non-induced problems, with k ranging from 0 to 5 (we discuss the other lines in this plot in Section 7.4.8). The results are strong: with $k = 0$ we may solve nearly every instance, whilst even at $k = 5$ we can solve over $4,000$ instances in both variants.

But are we learning anything about the results—that is, are there instances for which $k = 0$ is unsatisfiable, but that are satisfiable for small k? For the problem families which do not consist entirely of satisfiable instances, we plot this in Figure 7.14. In the "phase" family, which consists of instances crafted to be extremely difficult to solve, we are not able to answer this question, and in the "scalefree" family we see no satisfiable instances with low but not zero k (this is simply due to there being many vertices with loops in some pattern graphs, but no loops at all in the targets). However, in several of the remaining families we can more than double the number of instances for which exact solutions are known, and gain upper bounds on many more. This is particularly interesting for the "images" family due to Damiand et al. (2011), where the size of the solution has a direct real-world interpretation in terms of closeness of image matching.

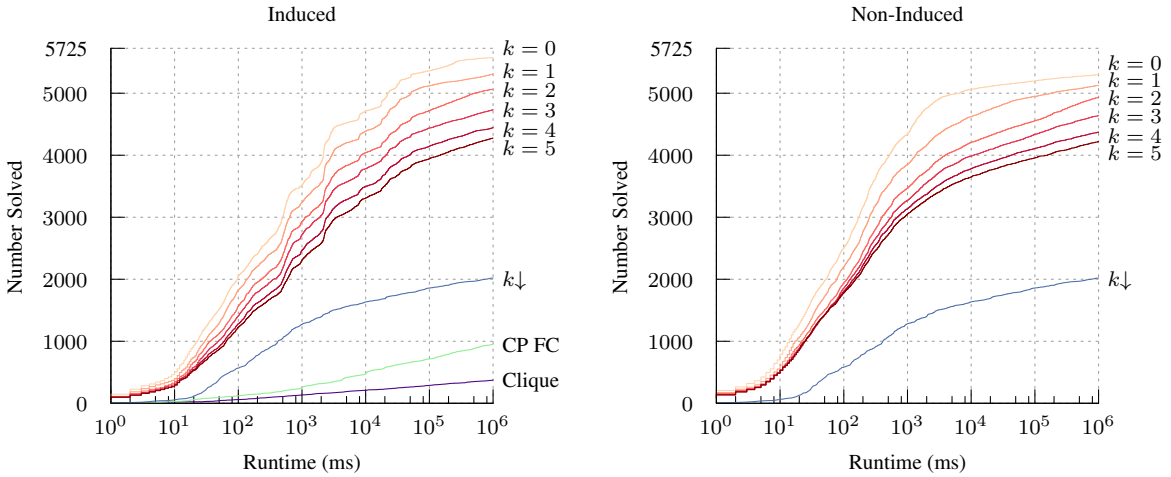


Figure 7.13: The cumulative number of instances solved over time, with different values of k. We also show the results of iteratively increasing k, and in the induced case, the performance of the two implementations discussed earlier in this chapter.

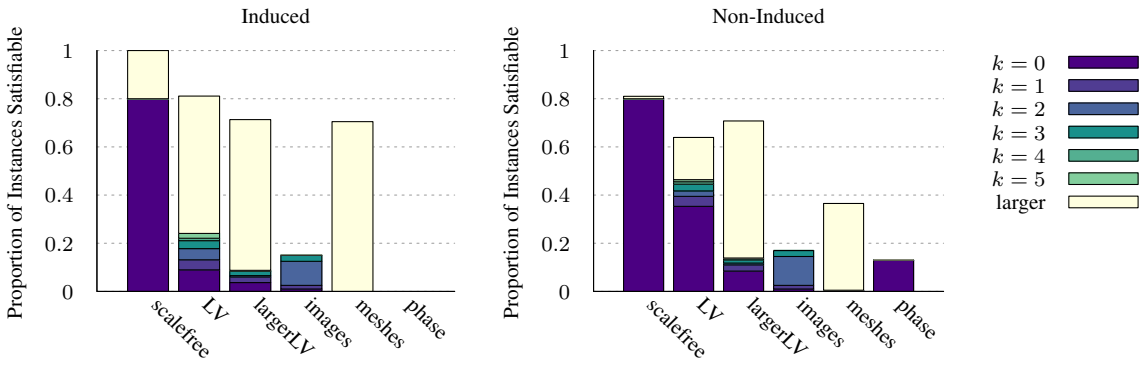


Figure 7.14: The proportion of instances, in different families, which become satisfiable for increasing values of k. The "larger" instances are those where we can prove unsatisfiability for $k = 5$, whilst the gap between the top of the bar and the top of the graph is the fraction of instances where a timeout was reached for at least one value of k.

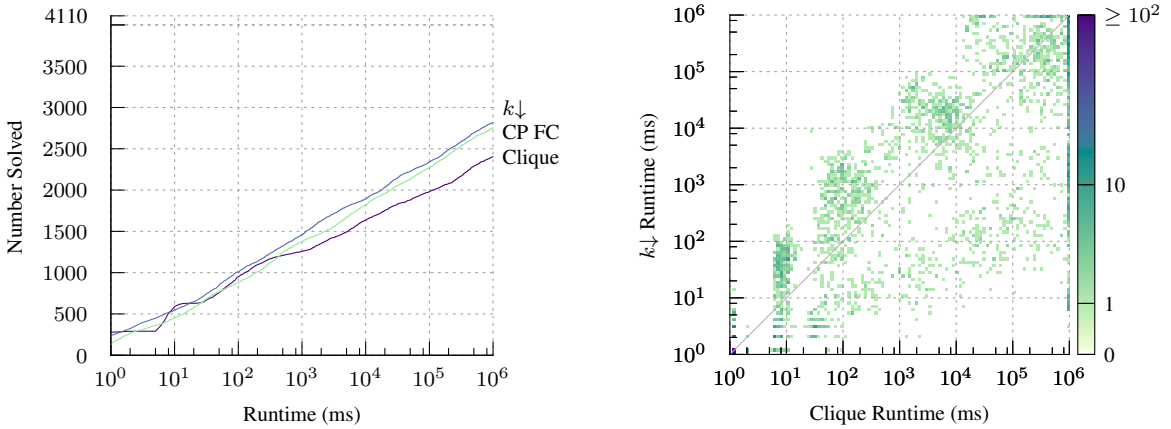


Figure 7.15: On the left, the cumulative number of instances solved over time, for the unlabelled maximum common subgraph instances. On the right, an instance by instance comparison with the clique approach.

7.4.8 Solving From the Top Down

What would happen if we used this approach to solve the maximum common (induced) subgraph problem? We could simply start at $k = 0$, and increase k until a solution is found. This would be tackling the problem in the opposite direction from the previous two approaches, which work by attempting to construct larger and larger solutions. The remaining lines in Figure 7.13 show this approach: "$k\downarrow$" is this algorithm, whilst "FC" and "clique" are the two algorithms discussed earlier in this chapter. Recalling the disclaimer in Section 7.4.3 regarding instances not fitting in the 64GBytes of RAM we have available, we see that this approach is able to close over twice as many of these instances as the previous state of the art. (The same conclusion holds even if every instance which is satisfiable with $k = 0$ is removed from the dataset.)

What if we use the instances designed for the maximum common subgraph problem? In Figure 7.15 we again compare these approaches, but returning to the undirected, unlabelled maximum common subgraph instances. In these instances the number of vertices in the pattern and target graphs is the same, which is not ideal for this algorithm (although the invariants are still effective in many cases). Nonetheless, this is by a small margin the single strongest solver so far. Interestingly, this algorithm often has complementary performance to the clique approach, which suggests that there is scope for an algorithm which runs both an upper bound and a constructive lower bound approach simultaneously or in parallel, stopping when the two bounds meet.

7.5 A Splitting Algorithm

We return now to the basic forward-checking constraint programming algorithm. McCreesh, Prosser, and Trimble (2017) make the following observation: at any stage in the search, the domains of any two variables are either identical or disjoint (excluding $\perp$, which is either present in every domain or in no domain). This allows the traditional set-based domain store to be replaced with flat arrays, one containing a permutation of all the values, and the other mapping each variable to a contiguous subset of this permutation—this is similar to data structures used in partition backtracking for graph isomorphism (López-Presa and Anta, 2009; McKay and Piperno, 2014) and the Bron and Kerbosch (1973) clique enumeration algorithm. Doing so allows for fast propagation (each subset is simply split in two following an assignment), allows the matching-based bound calculation to be replaced with a simple linear-time counting operation, and opens up opportunities for better variable ordering heuristics which can make use of dual viewpoint (Geelen, 1992) information for no cost.

The intricacies of this algorithm are described in McCreesh, Prosser, and Trimble (2017), and are not a contribution of this thesis. Here we simply present computational results in the left-hand column of Figure 7.16 which show that this algorithm is consistently at least an order

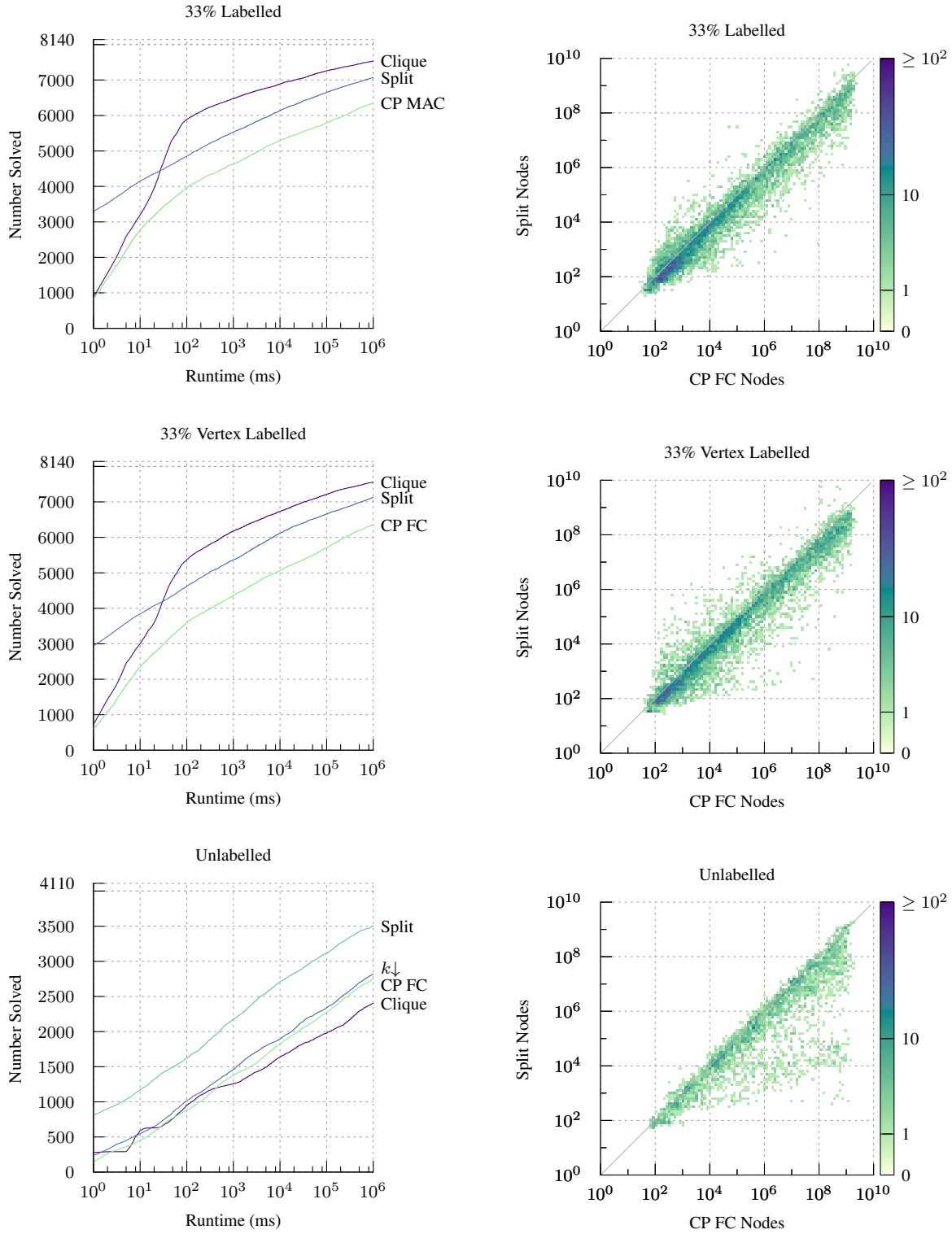


Figure 7.16: On the left, cumulative number of maximum common subgraph instances solved in under a certain time. On the right, search space size comparisons between the splitting and conventional forward checking algorithms.

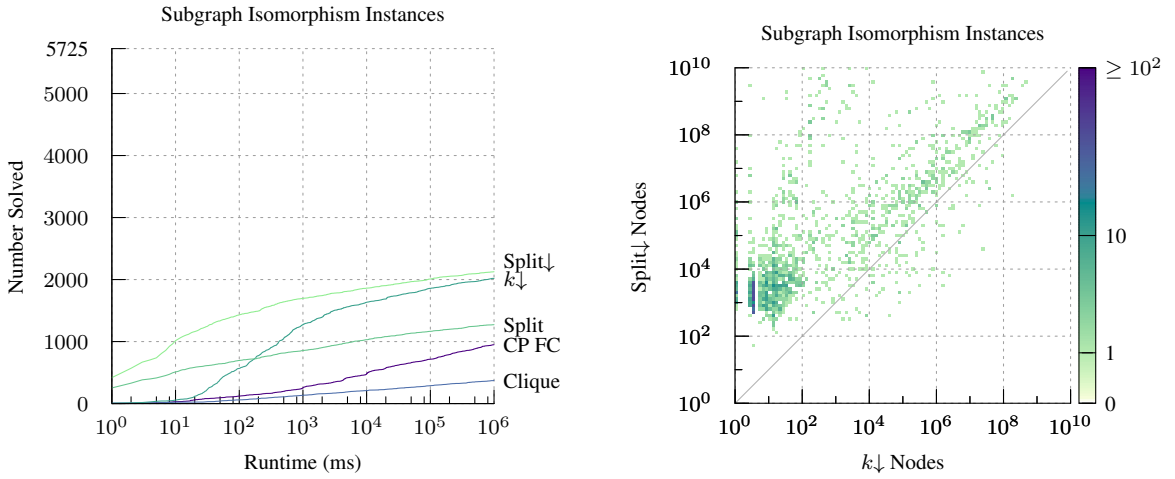


Figure 7.17: On the left, cumulative number of instances solved in under a certain time. On the right, search space size comparisons between the splitting and $k\downarrow$ algorithms.

of magnitude faster than the conventional constraint programming model, whilst performing similar amounts of work to the forward-checking algorithm (shown in the right-hand column; the slight reduction in work is due to the dual viewpoint value-ordering heuristics).

The algorithm is similarly strong on the large subgraph isomorphism instances, when adapted to use a top-down iteration method similar to that of $k\downarrow$. We show these results in Figure 7.17. However, on these instances, the heatmap shows a more interesting picture: although this approach is often fastest, it does not benefit from the additional constraints and variable filtering available to the $k\downarrow$ algorithm, it often does much more work, and it is sometimes much worse on an instance by instance basis.

Finally, there are many kinds of constraint which this approach cannot handle at all, since they would violate the disjoint domains assumption. For example, we could not incorporate the additional constraints and filtering available to the $k\downarrow$ algorithm. Nor could we adapt this technique for the non-induced k-less problem. However, it *can* be adapted to find maximum common connected subgraphs to great effect, as shown in Figure 7.18: again, it vastly outperforms the conventional constraint programming implementations.

7.6 Parallel Search

The results so far suggest there are three promising but very different approaches for maximum common (connected) subgraph problems: depending upon the orders of the graphs, and whether they have labels, we may wish to use clique-based models (Sections 7.2 and 7.3.3), k-less subgraph isomorphism (Section 7.4), or the splitting algorithm (Section 7.5). We now investigate whether these approaches can be parallelised. We should perhaps not expect to be successful—Minot, Ndiaye, and Solnon (2015) suggest that decomposition for "embarrasingly parallel search" style parallelism (Malapert, Régin, and Rezgui, 2016) is very difficult for

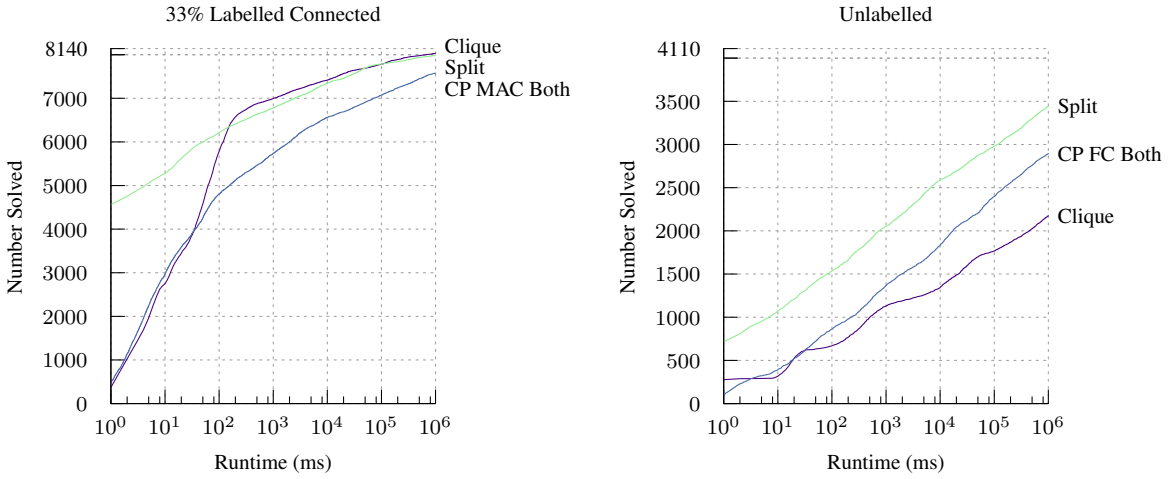


Figure 7.18: Cumulative numbers of maximum common connected subgraph instances solved in under a certain time, for labelled and unlabelled instances.

these problems.

For the clique model, we use the method described in Chapter 3. This technique applies identically in the connected case (Algorithm 7.1). Preliminary experiments suggested that, as for some of the maximum k-clique instances in Chapter 4, a splitting depth of 3 would lead to work balance problems, so we use a depth of 5. As before, we explicitly steal from highest in the search tree first, to introduce diversity into the search.

For the k-less algorithm, we follow the same process as in Chapter 5, using parallelism to help solve each value of k in turn (rather than exploring different values of k in parallel). Recall that the key idea is to proceed with execution as normal, but to use threads to pre-compute future iterations of the value assignment **foreach** loop (line 16 in Algorithm 7.2), and to terminate search globally when a solution is found. Also recall that helper threads can themselves be helped (which is necessary for work balance), and that parallelism is again tailored to offset weak early branching choices. We also parallelise the neighbourhood degree sequence and supplemental graph initialisation steps, using routine parallel loops.

For the splitting algorithm, things are not so simple: although it is a branch and bound algorithm with a strong constraint programming flavour, it makes heavy use of in-place, backtrackable data structures, and does not copy state between recursive calls. This is important from a performance perspective, particularly when one of the graphs is large—unconditionally copying data leads to more than a factor of ten slowdown for some instances. We therefore adopt a slightly different approach for parallelising the value-assignment for loop: before the loop, we make a single copy of inplace data structures. The active thread then proceeds as normal. To allow for parallelism, whenever a helper thread starts pre-computing future items in the **foreach** loop, it makes its own copies of the copies, and replays the entire **foreach** loop but skipping any recursive calls which have already been made. This somewhat reduces the overhead penalties compared to unconditional copying; to further reduce the costs,

we limit parallelism to a depth of 5 in the search tree, and use a non-copying search below that. Otherwise, we follow the same approach as before: for the regular algorithm, we parallelise the search and share an incumbent, and for the descending algorithm we use parallelism to accelerate search rather than to consider different decision problems simultaneously.

7.6.1 Experimental Results

Cumulative performance plots comparing each parallel algorithm to its sequential version are given in Figure 7.19, using 32 threads on 16 core hyper-threaded Intel Xeon E5-2640 v2 systems. The results are consistent: in each case the parallel algorithm is clearly and substantially better than its sequential version, except when the sequential runtime is below 100 milliseconds.

Figure 7.20 gives instance-by-instance comparisons for a representative subset of these algorithm and problem combinations. For the clique algorithm on unlabelled graphs (top left), the results show fairly consistent speedups for harder instances: there is only a single instance whose sequential runtime is over two seconds where we obtain a speedup of less than ten. We also see a significant number of superlinear speedups. Referring back to the discussion in Chapter 3, this is because there are relatively long proofs of optimality for most of the harder instances, but strong solutions are usually found quickly (and where they are not, the additional diversity helps). The results are similarly positive on labelled graphs (centre left), with speedups of greater than ten being the norm for harder instances, although there is more variety in the results.

For $k\downarrow$ on the unlabelled maximum common subgraph instances (top centre), the typical speedup is between two and ten, and occasionally there are speedups of between ten and one hundred. A few instances show small slowdowns: remember that when running 32 threads on a 16 core, hyper-threaded system, each individual thread runs somewhat slower than it would if only a single thread were running—in other words, these slight slowdowns are due to hardware effects, not changes to the search tree.

On the larger subgraph isomorphism instances (bottom left), we see a concentration of speedups of around ten. We also see strongly superlinear speedups in some cases, with eight instances finishing in under one second which timed out sequentially at one thousand seconds. Examining these instances closely shows that in each case, they are satisfiable with either $k = 0$ or very low values of k, and that the superlinear speedups are due to the parallel search introducing early diversity, offsetting an incorrect initial value-ordering heuristic choice. We also see more instances with slight absolute slowdowns: these are instances where the additional threads contribute no helpful work to the solution, and where the slowdown due to not having 32 times as many resources when using 32 threads is particularly pronounced (memory contention is much more of a problem for these larger graphs). Although not ideal, in aggregate these results are rare and are more than offset by the superlinear speedups for

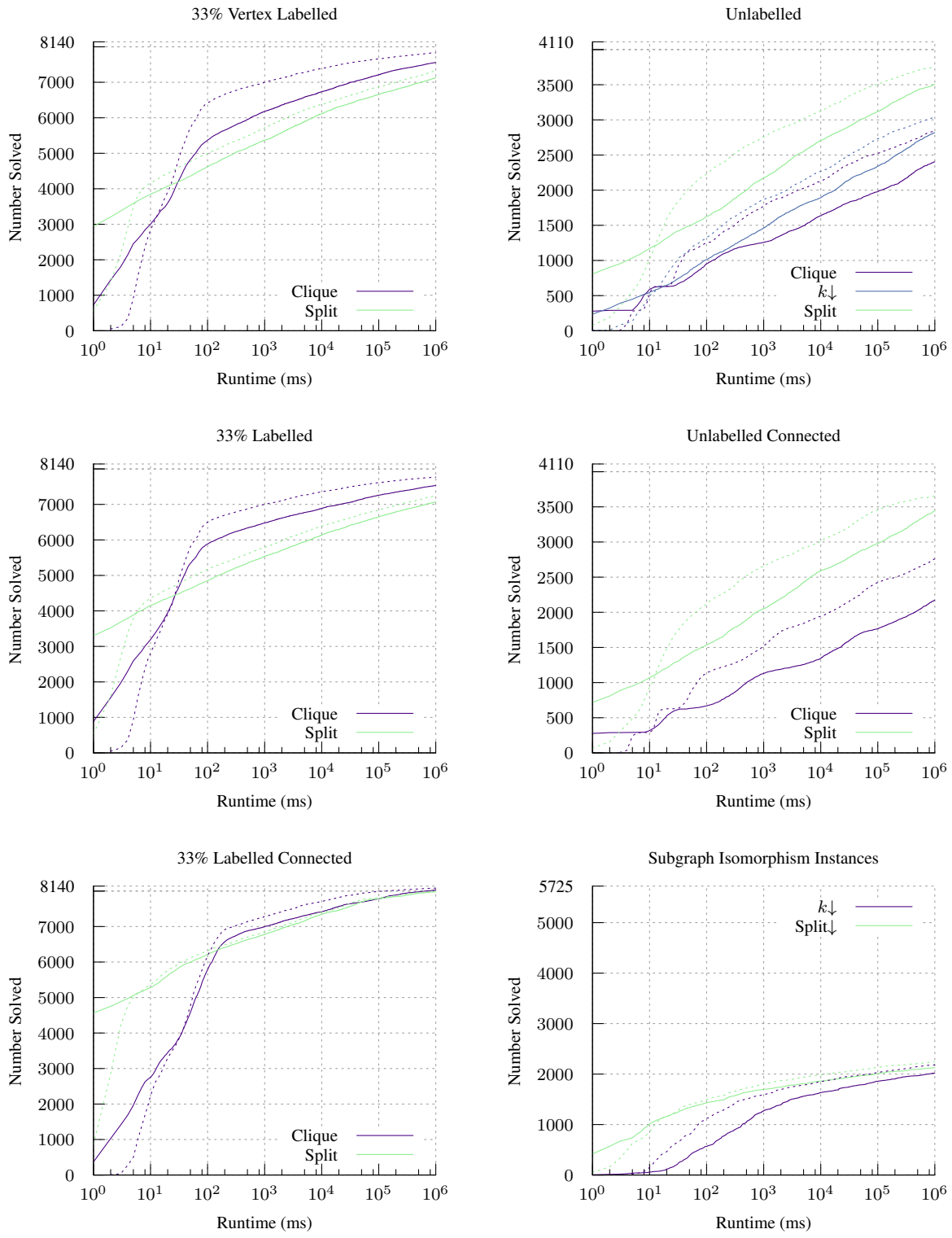


Figure 7.19: The cumulative number of instances solved in under a certain time, comparing sequential (solid lines) and parallel (dotted lines) algorithms, for six different families of maximum common (connected) subgraph problems. All experiments use 32 threads on a 16 core, hyper-threaded system.

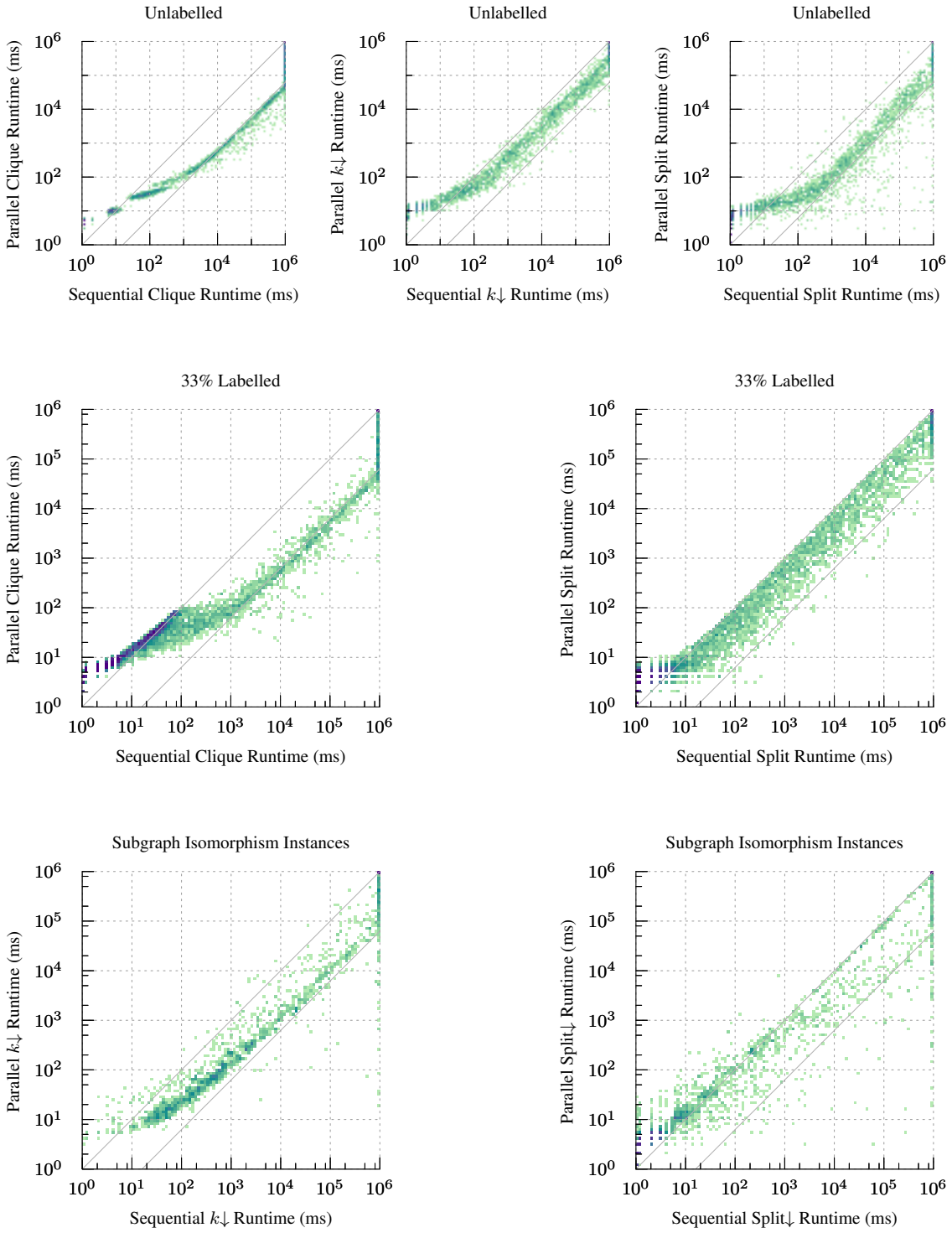


Figure 7.20: The benefits of parallelism, for different sets of problem instances and for different algorithms. All experiments use 32 threads on a 16 core, hyper-threaded system; points below the top diagonal line indicate a speedup, and the lower diagonal line is a sixteen times speedup.

other instances.

What about the splitting algorithm? On unlabelled instances (top right), our speedups are typically between those obtained from the clique algorithm and those from $k\downarrow$, being mainly a little below ten times. However, in some cases we see absolute slowdowns of up to four times: in these cases, parallelism does little, and is not enough to offset the costs we paid when switching from an in-place data structure to a data structure which requires copying. Interestingly, we also see many more superlinear speedups for this algorithm, which suggests that its value ordering heuristics are particularly poor early-on in search. (This last point is not particularly surprising: compared to $k\downarrow$, the splitting algorithm does not try to reduce domains at all at the top of search. The clique encoding, meanwhile, is able to capture even richer information about variable-value relationships. The splitting algorithm is fast, but comparatively stupid.)

Similar trends occur with the subgraph isomorphism instances (bottom right), and the absolute slowdowns go as high as a factor of ten—this is on instances with particularly large target graphs, where copying is most expensive, and where there is a high branching factor at the root of the search tree. Again, superlinear speedups are common.

On labelled instances (centre right), the parallel performance is less erratic. Although we no longer see absolute slowdowns (excluding on very easy instances), we also see fewer superlinear speedups, with most speedups being between one and ten. In many of these instances, parallelism does very little: restricting splitting to a depth of five in these cases gives work balance problems, but does help us avoid the substantial overheads that come with deeper splitting. This is because the search trees for these instances tend to have a very low branching factor—in constraint programming terms, the labels make each domain contain only a few values.

Figures 7.19 and 7.20 show that for all three algorithms, parallel search is beneficial, and also (reasonably) risk-free: although in a few cases the constant factor slowdowns due to hardware limitations and having to make changes to data structures are unpleasant, we do not encounter any exponential slowdowns due to search tree changes. But what about reproducibility and scalability? Further experiments in Figure 7.21 establish that both of these properties also hold. In the left-hand column, we plot each algorithm run against itself, using 32 threads in both cases. For the clique and $k\downarrow$ approaches, every instance is on the central diagonal line, showing an extremely high level of reproducibility. For the splitting algorithm, there is a little more variability, particularly for easier instances: inspecting the results suggests that this is not down to different amounts of search work being performed, but rather whether or not helper threads end up triggering additional copying.

In the right-hand column of Figure 7.21, we plot the effects of moving from 8 threads (on the same system) to 32 threads. Points below the diagonal line indicate an improvement. For the clique approach, increasing the number of threads is unequivocally beneficial. For

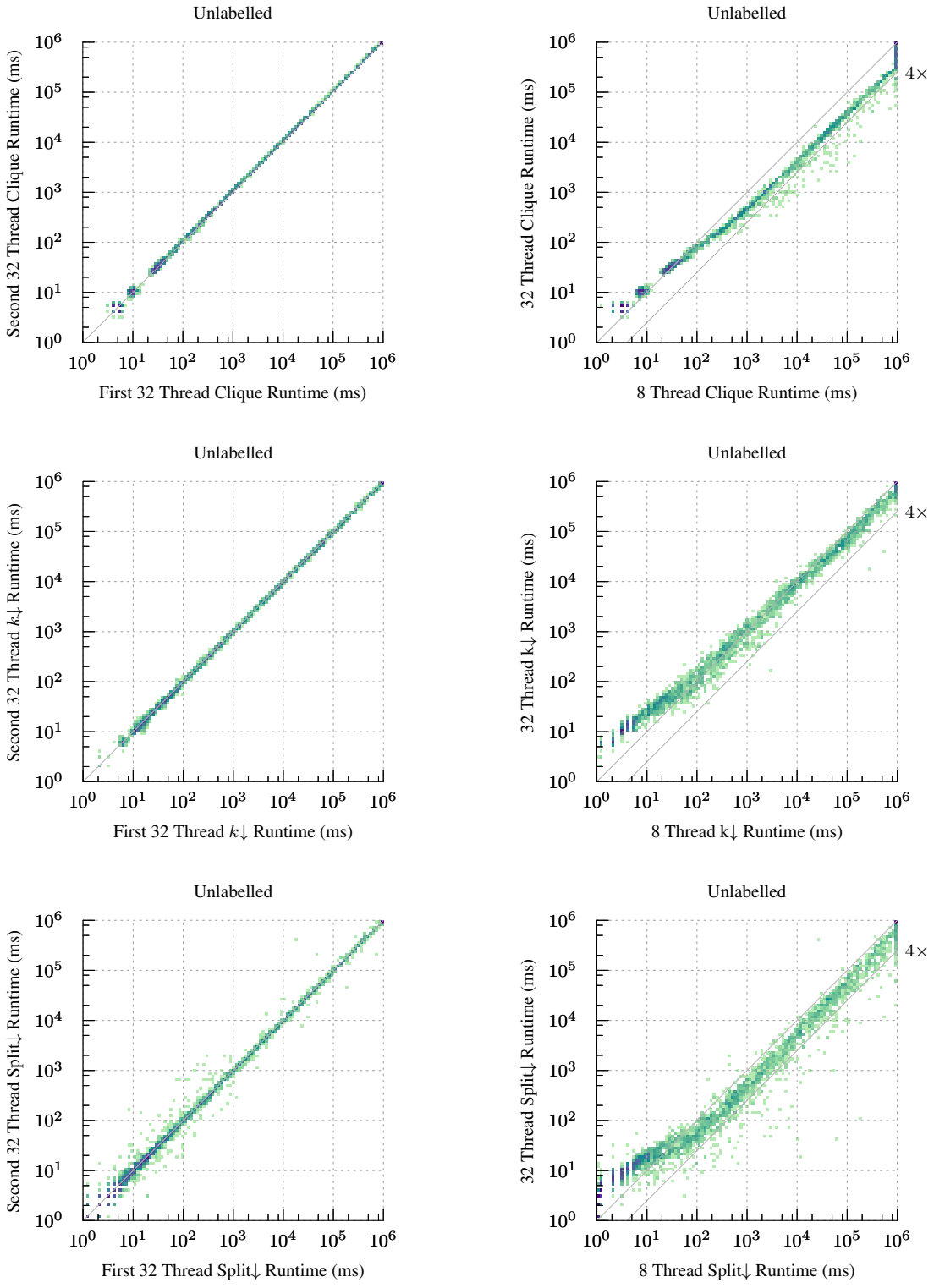


Figure 7.21: Parallel search is reproducible and scalable, as well as risk-free and beneficial. In the left-hand column, we compare each of the three parallel algorithms against a repeated run with the same parameters, on the unconnected, unlabelled instances (points on the diagonal line show reproducibility of runtimes). In the right-hand column, we compare going from 8 threads to 32 threads (points below the diagonal line are an improvement). All experiments are on a 16 core, hyper-threaded system.

the $k\downarrow$ algorithm, the benefits are slim, and sometimes performance worsens slightly due to memory contention (not work done) but we do not see any exponential slowdowns. Finally, for the splitting algorithm, the results are usually strictly better, although in a few cases the overheads increase further giving a slowdown; we also see an increase in the occurrence of strongly superlinear speedups.

It is likely that the parallel results for the splitting algorithm can be improved further. In the labelled case, we are often not obtaining a good work balance. However, deeper splitting is expensive, particularly with larger graphs. One option to consider would be recomputing parts of the search tree to avoid the costs of potentially allowing a copy in the future. This would be extremely unpleasant from an implementation perspective, but may be necessary to obtain more even results. However, even without this change, the parallel algorithm is clearly preferable to the sequential one.

7.7 Conclusion

We looked at two established and two new ways of solving the maximum common subgraph problem. Contradicting earlier claims in the literature, we have seen that a modern clique algorithm can perform competitively for maximum common subgraph problems, particularly when vertex or edge labels are involved. There may be futher scope for tailoring clique algorithms for these kinds of problem instance—for example, the first vertex selected has unusually strong filtering power in these instances, so it may be worth treating it specially (Suters et al., 2005). Better initial vertex orderings, and a bound functions which is aware that it is working on an association graph, may also be beneficial.

A deeper understanding of the colour bound's behaviour would be beneficial, since many of these instances appear to have unusually long proofs of optimality. The clique bound is potentially stronger than the soft all-different bound, since it can capture (the lack of) relationships between values in different domains. However, it can also severely misbehave. Not only does it sometimes produce colourings using more colours than there are variables in the constraint programming model, but it can produce colourings which remain too large at every level of branching (as an example, consider a greedy colouring on the association graph of a graph with no edges, and a graph with one edge).

However, we also saw that the association graph encoding was much too large to be practical for many instances which we would like to solve; conventional constraint programming models also struggle at this size. We therefore discussed two new approaches to the problem.

By considering a restricted variation of the problem, we have been able to go some of the way towards tackling the maximum common subgraph problem on much larger graphs than have previously been possible. We saw that some invariants from the subgraph isomorphism problem can still be used, albeit in a weakened form, with considerable effect when a small

number of vertices can be removed. We have by no means cracked the problem completely—these results are still far from where we would like them to be for use in real-world applications, and many instances remain open. However, for many of the open instances we are at least able to get upper bounds on the result for the first time.

We also had a first look at a promising forthcoming approach, which emulates the constraint programming algorithm, but using in-place data structures which exploit special structure present in domains for much faster propagation and reduced memory usage. Such an approach can explore the search space very quickly, making it strong in aggregate, but in individual cases it is often beaten by the k-less approach. The experimental results suggest there is work to be done in combining these two approaches, starting with the k-less algorithm, and switching to the splitting algorithm (or the clique algorithm) when k-less is no longer beneficial.

We do not have a clear picture of what makes maximum common subgraph instances hard: having a relatively large solution certainly seems to help, as do vertex labels, but other factors appear not to be useful in predicting runtimes. Our answers for this question are less definitive than in previous chapters, which raises an interesting question: are maximum common subgraph problems genuinely different in their difficulties, or are existing algorithms still unnecessarily weak on certain classes of instance?

We also looked at the maximum common connected subgraph problem. We introduced a novel clique-like algorithm, seeing how to adapt the combined bound and branching rule to this new setting; this clique-like algorithm performed particularly well on labelled instances. For constraint programming models, we looked at using a branching rule for maintaining connectedness during search, rather than simply as an ordering heuristic. This is unconventional and does not cleanly fit into the abstractions used by toolkits. However, we saw that combining conventional filtering and the special branching rule was beneficial.

There are other variants of the problem. We have yet to investigate partial (i.e. non-induced) or weighted graphs. Nor do we know a good way to find strongly connected common subgraphs—this would make the branching approach impossible, and filtering would be much more complicated. More generally, both the clique encoding and the splitting algorithm have limitations on the kinds of side constraints which they could support, whereas the conventional constraint programming and k-less algorithms do not. For example, restrictions on which vertices may or may not be removed, similar to the approach of Zampelli, Deville, and Dupont (2005), may require reverting to a weaker starting algorithm.

Our main goal of the chapter, however, was to introduce parallelism. This was a success: not only is the parallel algorithm clearly better overall than the best sequential algorithm in each case, but it is almost always better on an instance-by-instance basis (except for very easy instances with short sequential runtimes). This is despite overheads from changes to the algorithm design, and hardware complications like hyper-threading and limited memory

bandwidth. Furthermore, we saw that our parallel runtimes are reproducible, that increasing the number of threads does not introduce an exponential slowdown, and that we can use the interaction between work splitting and value ordering heuristics to our advantage. This is particularly significant, since simpler techniques like embarassingly parallel search (which does not provide these basic performance guarantees, let alone give a controlled interaction with value-ordering heuristics) behave poorly on these problems.

Chapter 8

Conclusion

This thesis has improved the state of the art in exact, practical algorithms for three families of hard subgraph problem: maximum clique, subgraph isomorphism, and maximum common subgraph. In each case, the results are then further improved by the use of parallel search to exploit the multiple cores present in every modern processor.

However, the most interesting parts of this work are not the faster algorithms we introduced. Rather, by combining careful thought and detailed experiments, we have achieved a deeper understanding of *why* these sequential and parallel algorithms behave the way they do, and this understanding has implications for problems beyond those considered in this thesis. We therefore conclude by reflecting upon the relationships between empirical hardness, heuristics, and parallelism, and considering the broader implications of our results.

8.1 Are Hard Subgraph Problems Hard?

Despite all these problems being NP-hard, for clique and subgraph isomorphism we were able to work with graphs with thousands of vertices. In an attempt to explain why we were so successful, we looked in detail at the behaviour of random instances for these problems. By generating instances close to a satisfiable / unsatisfiable phase transition, we were able to create small, really hard instances, and reassure ourselves that these algorithms could exhibit exponential behaviour. For clique and non-induced subgraph isomorphisms, we saw behaviour similar to that of other NP-complete problems. However, in Chapter 6 we saw that for induced subgraph isomorphisms, constrainedness and "closeness to a phase transition" gave very different predictions, with constrainedness matching empirical observations.

For common subgraph problems it is less clear whether similar behaviour occurs: although having many labels and a large solution makes instances reasonably easy, in general every approach struggles even with relatively small input graphs.

The primary goal of these algorithms is not to solve random instances, although as we discussed in Chapter 1 and demonstrated in Chapters 2 and 6, experiments on large families

of random instances can be helpful as a way of improving algorithm design. Hence we also experimented with real-world and other commonly-used sets of problem instances, which we argued were of varying levels of quality. For the maximum clique problem, the standard DIMACS instances exhibit a range of interesting properties, but there are not very many of them, which can lead to overfitting in algorithm design. The most widely-used suite for subgraph isomorphism is much larger, but uses synthetic data which is supposed to mimic real-world properties. We saw in Chapter 6 that use of this dataset can lead to extremely misleading conclusions. This is unfortunate: both clique and subgraph isomorphism are used in many papers and practical applications for a huge range of problems, but problem instances are rarely published. We therefore view Solnon's (2016) growing collection of subgraph isomorphism instances as particularly valuable, and would encourage others to contribute to it.

8.2 Lessons from Constraint Programming

Constraint programming toolkits have not appeared in this thesis (although they were used for preliminary experiments during development, as a sanity check on more complex algorithms). Despite this, almost every algorithm we discussed was either explicitly influenced by constraint programming, or has been better understood by reinterpreting it using language from constraint programming.

The first algorithm we discussed was for the maximum clique problem, in Chapter 2. This algorithm performs a combination of (very simple) inference and backtracking search, but its combined branching and bounding rule does not fit into the traditional constraint programming paradigm. This rule was also not understood, even by its inventors: our experiments demonstrated that the claimed reasons for its success were entirely false. By rephrasing this rule in terms of variables and ordering heuristics, we saw that it was a surrogate for the well-known "smallest domain first" heuristic (albeit with a different set of variables at each level of search). Although exploiting this knowledge only allowed us to obtain at-best marginal performance gains, the *understanding* is still useful, and should help inform future improvements to this family of algorithms.

The subgraph isomorphism algorithm we introduced in Chapter 5 is more obviously a constraint programming algorithm: it features domains, explicit variable- and value-ordering heuristics, and propagators. The domains are stored using bitsets, and the adjacency constraints are propagated using bit-parallel operations rather than sequentially. The choice of a bit-parallel all-different propagator rather than achieving generalised arc consistency is not standard, but is justified experimentally. There is also no constraint propagation queue, since a fixed ordering is possible. Together, these design choices meant that using a dedicated algorithm rather than a toolkit is not simply a matter of a small performance boost.

For the maximum common subgraph problem from Chapter 7, the CP and $k\downarrow$ algorithms are similarly inspired by constraint programming. The splitting algorithm, meanwhile, carries out the same inference and search, but using different data structures for domain stores, and unusual propagation algorithms. Similarly, the restricted branching rule we used to maintain connectedness does not fit cleanly into the way propagation and heuristics are usually separated in constraint programming toolkits, although constraint programming terminology makes the rule relatively easy to explain (at least compared to the version appearing in the clique-inspired algorithm).

Finally, the use of constraint programming as an aid to enlightenment was not limited to helping us understand and improve algorithms. It also helped us to identify problems in the design of larger systems: the second half of Chapter 6 highlights the need for constraints research to inform the redesign of graph databases, despite constraints technology not being used directly in these systems.

8.3 Perspectives on Parallel Search

Each of the algorithms in this thesis comes in a thread-parallel version, and in each case the parallel version is clearly the better choice. By this, we do not just mean that the parallel version is fastest. Instead, we mean that parallelism is not a risky choice: we can unequivocally state that enabling or increasing parallelism is not going to make things exponentially worse. Nor does opting for parallelism introduce new complications with reproducibility—although possibly less important for regular end users, for developers and scientists this factor can be critical.

Indeed, we believe that without these guarantees, parallel search cannot reasonably become the default for an optimisation toolkit. We therefore encourage researchers to think about these properties, and to see whether they hold both in theory and in practice when proposing new parallelisation mechanisms. We do not claim that doing so will be easy—indeed, it may be that such guarantees cannot be provided for many modern sequential algorithms. In Chapter 4 we saw that multi-objective optimisation could theoretically cause problems in this respect (although we did not witness them in practice, and in this case enforcing injectivity on the objective function would suffice to work around the problem, albeit with a potential loss of superlinear speedups). Weighted heuristics cause similar theoretical difficulties, and detailed experiments would be needed to establish whether these problems are common in practice. And unfortunately, it appears that learning solvers might be fundamentally incompatible with any kind of parallelism guarantee.

In our experiments, superlinear speedups were reasonably common. In Chapter 3 we justified why this is not unexpected: although we have value-ordering heuristics which are quite good most of the time for these graph problems, they ultimately use degree information,

and so are particularly weak at the top of search. One could certainly argue that these algorithms should therefore use something other than plain backtracking search, although discrepancy searches introduce considerable overheads. Instead, we have seen that using parallelism to introduce diversity tends to be successful in finding strong solutions faster, without adding cost to optimality proofs. Critical to this success is controlling the interaction between work splitting and value-ordering heuristics: given the extensive research into ordering heuristics in constraint programming, it is not surprising that systems which ignore these effects are limited to enumeration problems and certain optimisation problems with a high solution density.

8.4 Implementing Parallelism

> But if you want your application to benefit from the continued exponential throughput advances in new processors, it will need to be a well-written concurrent (usually multithreaded) application. And that's easier said than done, because not all problems are inherently parallelizable and because concurrent programming is hard.
>
> Sutter (2005)

Our experience from this thesis has been that implementing parallelism properly is hard. It requires deep and intrusive coupling with the algorithm, and can increase the amount of code needed by as much as an order of magnitude. This coupling has thus-far prevented us from separating parallel search mechanisms into a library: we found, for example, that each of the algorithms in Chapter 7 required subtle but important differences in how parallel search was implemented due to differing assumptions in the underlying sequential algorithms.

Now that we have spent time identifying the aspects which are most important for success, we hope that parallelism and programming languages researchers will be able to do more to help simplify this process. We found the high-level mechanisms offered by Intel Cilk Plus to be very convenient as a starting point for some algorithms, but not others: these subtle and important differences can mean the difference between Cilk being trivial to introduce or effectively impossible. The use of Cilk can also require introducing overheads and speculative copying for potential parallelism which may never be exploited. The costs of doing so are variable, and are both algorithm- and instance-dependent, ranging from negligible (in the clique case for small graphs) to over a factor of four (for large instances in the splitting maximum common subgraph case). Cilk's work stealing strategy is also hard-coded, and is not aware of value-ordering heuristics—although current implementations which steal early tend to give good diversity by a happy coincidence.

Cilk is limited to multi-core systems, and is not designed for distributed parallelism. Our experience is that high-level distributed systems like MPI are designed primarily for

parallel loops and similar computations which occur when solving differential equations, and that they struggle with highly irregular task parallelism. We continue to dream of a simple, non-intrusive task parallelism system which works at any scale, from trivial instances on a multi-core desktop up to supercomputers, which can solve the unnecessary copying problem, and which offers programmer control over the work distribution strategy.

8.5 Future Directions

We have seen the benefits that constraint programming and artificial intelligence technologies can bring to subgraph solving. We conclude this conclusion with a brief discussion of three pieces of work which we intend to carry out next, which if successful, could contribute knowledge back to the constraint programming community.

The unexpected success of the clique encoding for the maximum common subgraph problem in Chapter 7 suggests that it may be worth revisiting microstructure as a practical way of solving some other constraint problems. Because the objective function was, in effect, a 0/1-weighted scalar product over assignments, we could reformulate the maximum common subgraph problem as a maximum clique problem, rather than as a sequence of clique decision problems. We have not discussed the maximum weight clique problem in this thesis, but supporting weights would relax the 0/1 restriction and give more flexibility in this direction. It is also likely possible to tailor clique algorithms further for this application, by exploiting the special structure of microstructure graphs—for example, understanding which vertices correspond to which variables could improve the colour bound.

However, microstructure encodings can be very memory-intensive, and cannot easily be used for constraints that do not have a straightforward binary decomposition. In Chapters 4 and 7 we looked at integrating additional constraints into an augmented clique algorithm instead. Although effective in these two cases, the interaction with the bound was non-trivial, and it is not clear when else such an approach might be fruitful.

We also intend to continue our attempts to integrate learning into a subgraph solver. As well as the potential for direct performance benefits, this would pave the way for *subgraph modulo theories* problems. A hybrid system involving a learning subgraph algorithm communicating with a constraint programming or mixed integer solver could deliver the strengths of both approaches, combining the flexibility of a general-purpose solver with the performance and scalability of dedicated subgraph algorithms. Hybrid solving could also broaden the appeal of a microstructure solver, allowing a clique algorithm to handle a subset of variables and constraints whilst leaving the remainder of the problem to conventional propagators.

Finally, Chapters 2 and 6 suggest that we still do not fully understand variable- and value-ordering heuristics, particularly for optimisation problems. We believe that more research into the empirical hardness of subgraph problems would be helpful. For example,

we have started to repeat the experiments in Chapter 6 using k-regular and scale-free random models. These models are ideal for experimental work, since they have a suitable number of parameters to vary—not too many to be computationally infeasible or impossible to visualise, but enough to uncover much richer behaviour than is seen in single-parameter phase transition experiments. Preliminary experiments with weighted maximum clique algorithms, where both densities and weight distributions can be varied, suggest a similar story: there is a complex interaction between the parameters, and this affects how heuristics should be designed (should we branch based upon weight or degree?), but existing research cannot fully explain what we see. Knowing how to design tailored maximum weight clique heuristics could have broader implications: just as induced subgraph isomorphism can be viewed as solving two non-induced subgraph isomorphism problems simultaneously, many real-world constraint programming problems resemble a mix of two or more NP-hard problems.

References

Abello, James, Mauricio G. C. Resende, and Sandra Sudarsky (2002). "Massive Quasi-Clique Detection". In: *LATIN 2002: Theoretical Informatics, 5th Latin American Symposium, Cancun, Mexico, April 3-6, 2002, Proceedings*. Ed. by Sergio Rajsbaum. Vol. 2286. Lecture Notes in Computer Science. Springer, pp. 598–612. ISBN: 3-540-43400-3. DOI: 10.1007/3-540-45995-2_51 (cit. on p. 123).

Achlioptas, Dimitris, Michael S. O. Molloy, Lefteris M. Kirousis, Yannis C. Stamatiou, Evangelos Kranakis, and Danny Krizanc (2001). "Random Constraint Satisfaction: A More Accurate Picture". In: *Constraints* 6.4, pp. 329–344. DOI: 10.1023/A:1011402324562 (cit. on p. 157).

Aho, Alfred V., David S. Johnson, Richard M. Karp, S. Rao Kosaraju, and Catherine C. McGeoch (1996). *Theory of computing: Goals and directions*. Special Report of the National Science Foundation of the USA (cit. on p. 13).

Aisch, Gregor (2013). *Mastering Multi-hued Color Scales with Chroma.js*. URL: https://www.vis4.net/blog/posts/mastering-multi-hued-color-scales/ (visited on March 31, 2017) (cit. on p. 16).

Akiba, Takuya and Yoichi Iwata (2016). "Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover". In: *Theor. Comput. Sci.* 609, pp. 211–225. DOI: 10.1016/j.tcs.2015.09.023 (cit. on pp. 18, 60).

Akutsu, Tatsuya and Takeyuki Tamura (2013). "A Polynomial-Time Algorithm for Computing the Maximum Common Connected Edge Subgraph of Outerplanar Graphs of Bounded Degree". In: *Algorithms* 6.1, pp. 119–135. DOI: 10.3390/a6010119 (cit. on p. 179).

Anton, Călin and Lane Olson (2009). "Generating Satisfiable SAT Instances Using Random Subgraph Isomorphism". In: *Advances in Artificial Intelligence*. Ed. by Yong Gao and Nathalie Japkowicz. Vol. 5549. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 16–26. ISBN: 978-3-642-01817-6. DOI: 10.1007/978-3-642-01818-3_5 (cit. on p. 145).

Araujo Tavares, Wladimir (2016). "Algoritmos exatos para problema da clique maxima ponderada". Portugese. PhD thesis. Universidade Federal do Cearà (cit. on p. 59).

Archibald, Blair, Ciaran McCreesh, Patrick Maier, Rob Stewart, and Phil Trinder (2017). "Replicable Parallel Branch and Bound Search". In: *CoRR* abs/1703.05647 (cit. on p. 90).

Audemard, Gilles, Christophe Lecoutre, Mouny Samy Modeliar, Gilles Goncalves, and Daniel Cosmin Porumbel (2014). "Scoring-Based Neighborhood Dominance for the Subgraph Isomorphism Problem". In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. Ed. by Barry O'Sullivan. Vol. 8656. Lecture Notes in Computer Science. Springer, pp. 125–141. ISBN: 978-3-319-10427-0. DOI: 10.1007/978-3-319-10428-7_12 (cit. on pp. 125, 133, 134, 192).

Audemard, Gilles and Laurent Simon (2014). *The Glucose SAT Solver* (cit. on p. 154).

Bacchus, Fahiem and Adam J. Grove (1995). "On the Forward Checking Algorithm". In: *Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings*. Ed. by Ugo Montanari and Francesca Rossi. Vol. 976. Lecture Notes in Computer Science. Springer, pp. 292–308. ISBN: 3-540-60299-2. DOI: 10.1007/3-540-60299-2_18 (cit. on p. 169).

Backofen, Rolf and Sebastian Will (2002). "Excluding Symmetries in Constraint-Based Search". In: *Constraints* 7.3-4, pp. 333–349. DOI: 10.1023/A:1020533821509 (cit. on p. 118).

Bader, David A., William E. Hart, and Cynthia A. Phillips (2005). "Parallel Algorithm Design for Branch and Bound". In: *Tutorials on Emerging Methodologies and Applications in Operations Research*. Ed. by HJ. G. Vol. 76. International Series in Operations Research & Management Science. New York, NY, USA: Springer New York. Chap. 5, pp. 1–44. ISBN: 978-0-387-22826-6. DOI: 10.1007/0-387-22827-6_5 (cit. on p. 66).

Bahiense, Laura, Gordana Manic, Breno Piva, and Cid C. de Souza (2012). "The maximum common edge subgraph problem: A polyhedral investigation". In: *Discrete Applied Mathematics* 160.18, pp. 2523–2541. DOI: 10.1016/j.dam.2012.01.026 (cit. on p. 174).

Balas, Egon and William Niehaus (1993). "Finding large cliques in arbitrary graphs by bipartite matching". In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Ed. by David S. Johnson and Michael A. Trick. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 29–52 (cit. on p. 30).

Balas, Egon and Jue Xue (1996). "Weighted and Unweighted Maximum Clique Algorithms with Upper Bounds from Fractional Coloring". In: *Algorithmica* 15.5, pp. 397–412. DOI: 10.1007/BF01955041 (cit. on p. 57).

Balas, Egon and Chang Sung Yu (1986). "Finding a Maximum Clique in an Arbitrary Graph". In: *SIAM J. Comput.* 15.4, pp. 1054–1068. DOI: 10.1137/0215075 (cit. on p. 175).

Balasundaram, Balabhaskar and Sergiy Butenko (2006). "Graph Domination, Coloring and Cliques in Telecommunications". In: *Handbook of Optimization in Telecommunications*. Ed. by Mauricio G. C. Resende and Panos M. Pardalos. Springer, pp. 865–890. ISBN: 978-0-387-30662-9. DOI: 10.1007/978-0-387-30165-5_30 (cit. on p. 45).

Balasundaram, Balabhaskar, Sergiy Butenko, and Svyatoslav Trukhanov (2005). "Novel Approaches for Analyzing Biological Networks". In: *J. Comb. Optim.* 10.1, pp. 23–39. DOI: 10.1007/s10878-005-1857-x (cit. on p. 96).

Batagelj, Vladimir and Andrej Mrvar (2006). *Pajek datasets*. URL: http://vlado.fmf.uni-lj.si/pub/networks/data/ (visited on March 31, 2017) (cit. on pp. 99, 114).

Batsyn, Mikhail, Boris Goldengorin, Evgeny Maslov, and Panos M. Pardalos (2014). "Improvements to MCS algorithm for the maximum clique problem". In: *J. Comb. Optim.* 27.2, pp. 397–416. DOI: 10.1007/s10878-012-9592-6. (Cit. on pp. 43, 57–59, 90, 119).

Battiti, Roberto and Franco Mascia (2007). "An Algorithm Portfolio for the Sub-graph Isomorphism Problem". In: *Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics, International Workshop, SLS 2007, Brussels, Belgium, September 6-8, 2007, Proceedings*. Ed. by Thomas Stützle, Mauro Birattari, and Holger H. Hoos. Vol. 4638. Lecture Notes in Computer Science. Springer, pp. 106–120. ISBN: 978-3-540-74445-0. DOI: 10.1007/978-3-540-74446-7_8 (cit. on pp. 145, 168).

Bergman, David, André A. Ciré, Ashish Sabharwal, Horst Samulowitz, Vijay A. Saraswat, and Willem Jan van Hoeve (2014). "Parallel Combinatorial Optimization with Decision Diagrams". In: *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*. Ed. by Helmut Simonis. Vol. 8451. Lecture Notes in Computer Science. Springer, pp. 351–367. ISBN: 978-3-319-07045-2. DOI: 10.1007/978-3-319-07046-9_25 (cit. on pp. 59, 66).

Berman, Piotr and Andrzej Pelc (1990). "Distributed probabilistic fault diagnosis for multiprocessor systems". In: *Proceedings of the 20th International Symposium on Fault-Tolerant Computing, FTCS 1990, Newcastle Upon Tyne, UK, 26-28 June, 1990*. IEEE Computer Society, pp. 340–346. ISBN: 0-8186-2051-X. DOI: 10.1109/FTCS.1990.89383. (Cit. on p. 43).

Bessière, Christian and Jean-Charles Régin (1996). "MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems". In: *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*. Ed. by Eugene C. Freuder. Vol. 1118. Lecture Notes in Computer Science. Springer, pp. 61–75. DOI: 10.1007/3-540-61551-2_66 (cit. on p. 127).

Bessiere, Christian, Bruno Zanuttini, and Cèsar Fernàndez (2004). "Measuring Search Trees". In: *Workshop on Modelling and Solving Problems with Constraints - ECAI'2004*. Ed. by Hnich B. and Walsh T. Workshop held in conjunction with the 16th European Conference on Artificial Intelligence (ECAI 2004). Valencia, Spain: Hnich B., pp. 31–40 (cit. on pp. 14, 34, 37).

Binstock, Andrew (2008). "Interview with Donald Knuth". In: *InformIT* (cit. on p. 19).

Blindell, Gabriel Hjort, Roberto Castañeda Lozano, Mats Carlsson, and Christian Schulte (2015). "Modeling Universal Instruction Selection". In: *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*. Ed. by Gilles Pesant. Vol. 9255. Lecture Notes in Computer Science. Springer, pp. 609–626. ISBN: 978-3-319-23218-8. DOI: 10.1007/978-3-319-23219-5_42 (cit. on p. 125).

Bomze, Immanuel M., Marco Budinich, Panos M. Pardalos, and Marcello Pelillo (1999). "The Maximum Clique Problem". In: *Handbook of Combinatorial Optimization: Supplement Volume A*. Ed. by Ding-Zhu Du and Panos M. Pardalos. Boston, MA: Springer US, pp. 1–74. ISBN: 978-1-4757-3023-4. DOI: 10.1007/978-1-4757-3023-4_1 (cit. on pp. 43, 45).

Bonnici, Vincenzo, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro (2013). "A subgraph isomorphism algorithm and its application to biochemical data". In: *BMC Bioinformatics* 14.7, S13. ISSN: 1471-2105. DOI: 10.1186/1471-2105-14-S7-S13 (cit. on p. 125).

Borland, D. and R. M. Taylor II (2007). "Rainbow Color Map (Still) Considered Harmful". In: *IEEE Computer Graphics and Applications* 27.2, pp. 14–17. ISSN: 0272-1716. DOI: 10.1109/MCG.2007.323435 (cit. on p. 16).

Bourjolly, Jean-Marie, Gilbert Laporte, and Gilles Pesant (2000). "Heuristics for finding k-clubs in an undirected graph". In: *Computers & OR* 27.6, pp. 559–569. DOI: 10.1016/S0305-0548(99)00047-7 (cit. on p. 96).

Bourjolly, Jean-Marie, Gilbert Laporte, and Gilles Pesant (2002). "An exact algorithm for the maximum k-club problem in an undirected graph". In: *European Journal of Operational Research* 138.1, pp. 21–28. DOI: 10.1016/S0377-2217(01)00133-3 (cit. on p. 96).

Brélaz, Daniel (1979). "New Methods to Color Vertices of a Graph". In: *Commun. ACM* 22.4, pp. 251–256. DOI: 10.1145/359094.359101. (Cit. on p. 56).

Brockington, Mark and Joseph C. Culberson (1993). "Camouflaging independent sets in quasi-random graphs". In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Ed. by David S. Johnson and Michael A. Trick. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 75–88. (Cit. on pp. 40, 62).

Bron, Coenraad and Joep Kerbosch (1973). “Finding All Cliques of an Undirected Graph (Algorithm 457)”. In: *Commun. ACM* 16.9, pp. 575–576 (cit. on pp. 60, 172, 198).

Brown, Kenneth. N., Patrick Prosser, J. Christopher Beck, and Christine Wei Wu (2005). “Exploring the use of constraint programming for enforcing connectivity during graph generation”. In: *Proceedings IJCAI Workshop on Modelling and Solving Problems with Constraints, Edinburgh, Scotland,* pp. 26–31 (cit. on p. 180).

Bruschi, Danilo, Lorenzo Martignoni, and Mattia Monga (2006). “Detecting Self-mutating Malware Using Control-Flow Graph Matching”. In: *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006, Proceedings*. Ed. by Roland Büschkes and Pavel Laskov. Vol. 4064. Lecture Notes in Computer Science. Springer, pp. 129–143. ISBN: 3-540-36014-X. DOI: 10.1007/11790754_8 (cit. on p. 125).

Bunke, Horst (1997). “On a relation between graph edit distance and maximum common subgraph”. In: *Pattern Recognition Letters* 18.8, pp. 689–694. DOI: 10.1016/S0167-8655(97)00060-3 (cit. on p. 171).

Bunke, Horst, Pasquale Foggia, C. Guidobaldi, Carlo Sansone, and Mario Vento (2002). “A Comparison of Algorithms for Maximum Common Subgraph on Randomly Connected Graphs”. In: *Structural, Syntactic, and Statistical Pattern Recognition, Joint IAPR International Workshops SSPR 2002 and SPR 2002, Windsor, Ontario, Canada, August 6-9, 2002, Proceedings*. Ed. by Terry Caelli, Adnan Amin, Robert P. W. Duin, Mohamed S. Kamel, and Dick de Ridder. Vol. 2396. Lecture Notes in Computer Science. Springer, pp. 123–132. ISBN: 3-540-44011-9. DOI: 10.1007/3-540-70659-3_12 (cit. on p. 177).

Butenko, Sergiy and Wilbert E. Wilhelm (2006). “Clique-detection models in computational biochemistry and genomics”. In: *European Journal of Operational Research* 173.1, pp. 1–17. DOI: 10.1016/j.ejor.2005.05.026 (cit. on p. 45).

Caniou, Yves, Philippe Codognet, Daniel Diaz, and Salvador Abreu (2011). “Experiments in Parallel Constraint-Based Local Search”. In: *Evolutionary Computation in Combinatorial Optimization - 11th European Conference, EvoCOP 2011, Torino, Italy, April 27-29, 2011. Proceedings*. Ed. by Peter Merz and Jin-Kao Hao. Vol. 6622. Lecture Notes in Computer Science. Springer, pp. 96–107. ISBN: 978-3-642-20363-3. DOI: 10.1007/978-3-642-20364-0_9 (cit. on p. 82).

Cao, Ning, Zhenyu Yang, Cong Wang, Kui Ren, and Wenjing Lou (2011). “Privacy-Preserving Query over Encrypted Graph-Structured Data in Cloud Computing”. In: *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*. IEEE Computer Society, pp. 393–402. ISBN: 978-0-7695-4364-2. DOI: 10.1109/ICDCS.2011.84 (cit. on p. 163).

Carletti, Vincenzo (2016). “Exact and Inexact Methods for Graph Similarity in Structural Pattern Recognition”. PhD thesis. Université de Caen; Universita degli studi di Salerno (cit. on pp. 126, 169).

Carletti, Vincenzo, Pasquale Foggia, and Mario Vento (2015). “VF2 Plus: An Improved version of VF2 for Biological Graphs”. In: *Graph-Based Representations in Pattern Recognition - 10th IAPR-TC-15 International Workshop, GbRPR 2015, Beijing, China, May 13-15, 2015. Proceedings*. Ed. by Cheng-Lin Liu, Bin Luo, Walter G. Kropatsch, and Jian Cheng. Vol. 9069. Lecture Notes in Computer Science. Springer, pp. 168–177. ISBN: 978-3-319-18223-0. DOI: 10.1007/978-3-319-18224-7_17 (cit. on pp. 126, 169).

Carrabs, Francesco, Raffaele Cerulli, and Paolo Dell’Olmo (2014). “A Mathematical Programming Approach for the Maximum Labeled Clique Problem”. In: *Procedia - Social and Behavioral Sciences* 108.0, pp. 69–78. ISSN: 1877-0428. DOI: 10.1016/j.sbspro.2013.12.821 (cit. on pp. 3, 108, 109, 111, 112, 114, 123, 124).

Carraghan, Randy and Panos M. Pardalos (1990). “An exact algorithm for the maximum clique problem”. In: *Operations Research Letters* 9.6, pp. 375–382. ISSN: 0167-6377. DOI: 10.1016/0167-6377(90)90057-C. (Cit. on p. 29).

Carvalho, Filipa D. and Maria Teresa Almeida (2016). “The triangle k-club problem”. In: *Journal of Combinatorial Optimization*, pp. 1–33. ISSN: 1573-2886. DOI: 10.1007/s10878-016-0009-9 (cit. on p. 96).

Chang, Maw-Shang, Ling-Ju Hung, Chih-Ren Lin, and Ping-Chen Su (2013). “Finding large k-clubs in undirected graphs”. In: *Computing* 95.9, pp. 739–758. DOI: 10.1007/s00607-012-0263-3 (cit. on pp. 96, 97, 103, 104, 106).

Cheeseman, Peter, Bob Kanefsky, and William M. Taylor (1991). “Where the Really Hard Problems Are”. In: *Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991*. Ed. by John Mylopoulos and Raymond Reiter. Morgan Kaufmann, pp. 331–340. ISBN: 1-55860-160-0 (cit. on pp. 35, 146).

Cheng, James, Yiping Ke, Wilfred Ng, and An Lu (2007). “Fg-index: towards verification-free query processing on graph databases”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. Ed. by Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou. ACM, pp. 857–872. ISBN: 978-1-59593-686-8. DOI: 10.1145/1247480.1247574 (cit. on p. 163).

Chu, Geoffrey, Christian Schulte, and Peter J. Stuckey (2009). “Confidence-Based Work Stealing in Parallel Constraint Programming”. In: *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*. Ed. by Ian P. Gent. Vol. 5732. Lecture Notes in Computer Science. Springer, pp. 226–241. ISBN: 978-3-642-04243-0. DOI: 10.1007/978-3-642-04244-7_20 (cit. on pp. 83, 93).

Clausen, Jens (1997). "Parallel Branch and Bound — Principles and Personal Experiences". In: *Parallel Computing in Optimization*. Ed. by Athanasios Migdalas, Panos M. Pardalos, and Sverre Storøy. Boston, MA: Springer US, pp. 239–267. ISBN: 978-1-4613-3400-2. DOI: 10.1007/978-1-4613-3400-2_7 (cit. on p. 81).

Clearwater, Scott H., Bernardo A. Huberman, and Tad Hogg (1991). "Cooperative Solution of Constraint Satisfaction Problems". In: *Science* 254.5035, pp. 1181–1183. DOI: 10.1126/science.254.5035.1181 (cit. on p. 91).

Coffman, Thayne, Seth Greenblatt, and Sherry Marcus (2004). "Graph-based technologies for intelligence analysis". In: *Commun. ACM* 47.3, pp. 45–47. DOI: 10.1145/971617.971643 (cit. on p. 125).

Cohen, David A., Martin C. Cooper, Páidí Creed, Dániel Marx, and András Z. Salamon (2012). "The Tractability of CSP Classes Defined by Forbidden Patterns". In: *J. Artif. Intell. Res. (JAIR)* 45, pp. 47–78. DOI: 10.1613/jair.3651 (cit. on p. 12).

Cohen, David A., Peter G. Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith (2006). "Symmetry Definitions for Constraint Satisfaction Problems". In: *Constraints* 11.2-3, pp. 115–137. DOI: 10.1007/s10601-006-8059-8 (cit. on p. 12).

Cohen, David A., Christopher Jefferson, and Karen E. Petrie (2016). "A Theoretical Framework for Constraint Propagator Triggering". In: *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016, Tarrytown, NY, USA, July 6-8, 2016*. Ed. by Jorge A. Baier and Adi Botea. AAAI Press, pp. 19–27. ISBN: 978-1-57735-769-8 (cit. on p. 10).

Combier, Camille, Guillaume Damiand, and Christine Solnon (2013). "Map Edit Distance vs. Graph Edit Distance for Matching Images". In: *Graph-Based Representations in Pattern Recognition - 9th IAPR-TC-15 International Workshop, GbRPR 2013, Vienna, Austria, May 15-17, 2013. Proceedings*. Ed. by Walter G. Kropatsch, Nicole M. Artner, Yll Haxhimusa, and Xiaoyi Jiang. Vol. 7877. Lecture Notes in Computer Science. Springer, pp. 152–161. ISBN: 978-3-642-38220-8. DOI: 10.1007/978-3-642-38221-5_16 (cit. on p. 171).

Conte, Donatello, Pasquale Foggia, Carlo Sansone, and Mario Vento (2004). "Thirty Years Of Graph Matching In Pattern Recognition". In: *IJPRAI* 18.3, pp. 265–298. DOI: 10.1142/S0218001404003228 (cit. on p. 125).

Conte, Donatello, Pasquale Foggia, and Mario Vento (2007). "Challenging Complexity of Maximum Common Subgraph Detection Algorithms: A Performance Analysis of Three Algorithms on a Wide Database of Graphs". In: *J. Graph Algorithms Appl.* 11.1, pp. 99–143 (cit. on p. 177).

Cook, Diane J. and Lawrence B. Holder (1994). “Substructure Discovery Using Minimum Description Length and Background Knowledge”. In: *J. Artif. Intell. Res. (JAIR)* 1, pp. 231–255. DOI: 10.1613/jair.43 (cit. on p. 171).

Cook, Stephen A. and David G. Mitchell (1996). “Finding hard instances of the satisfiability problem: A survey”. In: *Satisfiability Problem: Theory and Applications, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, March 11-13, 1996*. Ed. by Ding-Zhu Du, Jun Gu, and Panos M. Pardalos. Vol. 35. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 1–18 (cit. on p. 11).

Cooper, Martin C., Peter G. Jeavons, and András Z. Salamon (2010). “Generalizing constraint satisfaction on trees: Hybrid tractability and variable elimination”. In: *Artif. Intell.* 174.9-10, pp. 570–584. DOI: 10.1016/j.artint.2010.03.002 (cit. on p. 12).

Cordella, Luigi P., Pasquale Foggia, Carlo Sansone, and Mario Vento (2004). “A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 26.10, pp. 1367–1372. DOI: 10.1109/TPAMI.2004.75 (cit. on pp. 26, 126, 133, 134, 138, 147, 165).

Corrêa, Ricardo C., Philippe Michelon, Bertrand Le Cun, Thierry Mautor, and Diego Delle Donne (2014). “A Bit-Parallel Russian Dolls Search for a Maximum Cardinality Clique in a Graph”. In: *CoRR* abs/1407.1209. (Cit. on pp. 43, 59).

Crainic, Teodor Gabriel, Bertrand Le Cun, and Catherine Roucairol (2006). “Parallel Branch-and-Bound Algorithms”. In: *Parallel Combinatorial Optimization*. Hoboken, NJ, USA: John Wiley & Sons, Inc., pp. 1–28. ISBN: 9780470053928. DOI: 10.1002/9780470053928.ch1 (cit. on p. 66).

Crawford, James M., Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy (1996). “Symmetry-Breaking Predicates for Search Problems”. In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR’96), Cambridge, Massachusetts, USA, November 5-8, 1996*. Ed. by Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro. Morgan Kaufmann, pp. 148–159. ISBN: 1-55860-421-9 (cit. on p. 118).

Dalla Preda, Mila and Vanessa Vidali (2017). “Abstract Similarity Analysis”. In: *Electronic Notes in Theoretical Computer Science* 331. Proceedings of the Sixth Workshop on Numerical and Symbolic Abstract Domains (NSAD 2016), pp. 87–99. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2017.02.006 (cit. on p. 125).

Damiand, Guillaume, Christine Solnon, Colin de la Higuera, Jean-Christophe Janodet, and Émilie Samuel (2011). “Polynomial algorithms for subisomorphism of nD open combinatorial maps”. In: *Computer Vision and Image Understanding* 115.7, pp. 996–1010. DOI: 10.1016/j.cviu.2010.12.013 (cit. on pp. 125, 134, 196).

Daylight Chemical Information Systems, Inc. (2011). *Daylight Theory Manual, version 4.9*. http://www.daylight.com/dayhtml/doc/theory/theory.thor.html (cit. on p. 161).

De Santo, Massimo, Pasquale Foggia, Carlo Sansone, and Mario Vento (2003). "A large database of graphs and its use for benchmarking graph isomorphism algorithms". In: *Pattern Recognition Letters* 24.8, pp. 1067–1079. DOI: 10.1016/S0167-8655(02)00253-2 (cit. on pp. 145, 177).

De Bruin, Arie, Gerard A. P. Kindervater, and Harry W. J. M. Trienekens (1995). "Asynchronous Parallel Branch and Bound and Anomalies". In: *Parallel Algorithms for Irregularly Structured Problems, Second International Workshop, IRREGULAR '95, Lyon, France, September 4-6, 1995, Proceedings*. Ed. by Afonso Ferreira and José D. P. Rolim. Vol. 980. Lecture Notes in Computer Science. Springer, pp. 363–377. ISBN: 3-540-60321-2. DOI: 10.1007/3-540-60321-2_29 (cit. on pp. 67, 80).

Debroni, Jennifer, John D. Eblen, Michael A. Langston, Wendy Myrvold, Peter W. Shor, and Dinesh Weerapurage (2011). "A complete resolution of the Keller maximum clique problem". In: *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*. Ed. by Dana Randall. SIAM, pp. 129–135. ISBN: 978-0-89871-993-2. DOI: 10.1137/1.9781611973082.11 (cit. on pp. 42, 44, 66).

Delavallade, Thomas, Simon Fossier, Claire Laudy, and Gaëlle Lortal (2016). "On the Challenges of Using Social Media for Crisis Management". In: *Fusion Methodologies in Crisis Management: Higher Level Fusion and Decision Making*. Ed. by Galina Rogova and Peter Scott. Cham: Springer International Publishing, pp. 137–175. ISBN: 978-3-319-22527-2. DOI: 10.1007/978-3-319-22527-2_8 (cit. on p. 171).

Dent, Michael J. (1996). "Minimal Forward Checking–a Lazy Constraint Satisfaction Search Algorithm: Experimental And Theoretical Results". PhD thesis. The University of Western Ontario (cit. on p. 169).

Dent, Michael J. and Robert E. Mercer (1994). "Minimal Forward Checking". In: *Sixth International Conference on Tools with Artificial Intelligence, ICTAI '94, New Orleans, Louisiana, USA, November 6-9, 1994*. IEEE Computer Society, pp. 432–438. ISBN: 0-8186-6785-0. DOI: 10.1109/TAI.1994.346460 (cit. on p. 169).

Depolli, Matjaz, Janez Konc, Kati Rozman, Roman Trobec, and Dusanka Janezic (2013). "Exact Parallel Maximum Clique Algorithm for General and Protein Graphs". In: *Journal of Chemical Information and Modeling* 53.9, pp. 2217–2228. DOI: 10.1021/ci4002525 (cit. on pp. 44, 64, 66–70, 73, 78, 90, 92, 105).

Djoko, Surnjani, Diane J. Cook, and Lawrence B. Holder (1997). "An Empirical Study of Domain Knowledge and Its Benefits to Substructure Discovery". In: *IEEE Trans. Knowl. Data Eng.* 9.4, pp. 575–586. DOI: 10.1109/69.617051 (cit. on p. 171).

Dooms, Grégoire, Yves Deville, and Pierre Dupont (2005). "CP(Graph): Introducing a Graph Computation Domain in Constraint Programming". In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*. Ed. by Peter van Beek. Vol. 3709. Lecture Notes in Computer Science. Springer, pp. 211–225. ISBN: 3-540-29238-1. DOI: 10.1007/11564751_18 (cit. on p. 180).

Droschinsky, Andre, Nils Kriege, and Petra Mutzel (2016). "Faster Algorithms for the Maximum Common Subtree Isomorphism Problem". In: *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*. Ed. by Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier. Vol. 58. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 33:1–33:14. ISBN: 978-3-95977-016-3. DOI: 10.4230/LIPIcs.MFCS.2016.33 (cit. on pp. 174, 179).

Droschinsky, Andre, Nils Kriege, and Petra Mutzel (2017). "Finding Largest Common Substructures of Molecules in Quadratic Time". In: *SOFSEM 2017: Theory and Practice of Computer Science - 43rd International Conference on Current Trends in Theory and Practice of Computer Science, Limerick, Ireland, January 16-20, 2017, Proceedings*. Ed. by Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria. Vol. 10139. Lecture Notes in Computer Science. Springer, pp. 309–321. ISBN: 978-3-319-51962-3. DOI: 10.1007/978-3-319-51963-0_24 (cit. on p. 174).

Durand, Paul J, Rohit Pasari, Johnnie W Baker, and Chun-che Tsai (1999). "An efficient algorithm for similarity analysis of molecules". In: *Internet Journal of Chemistry* 2.17, pp. 1–16 (cit. on p. 175).

Eblen, John D., Charles A. Phillips, Gary L. Rogers, and Michael A. Langston (2012). "The maximum clique enumeration problem: algorithms, applications, and implementations". In: *BMC Bioinformatics* 13.S-10, S5. DOI: 10.1186/1471-2105-13-S10-S5 (cit. on pp. 45, 60).

Ehrlich, Hans-Christian and Matthias Rarey (2011). "Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review". In: *Wiley Interdisciplinary Reviews: Computational Molecular Science* 1.1, pp. 68–79. ISSN: 1759-0884. DOI: 10.1002/wcms.5 (cit. on pp. 4, 171, 179).

Englert, Péter and Péter Kovács (2015). "Efficient Heuristics for Maximum Common Substructure Search". In: *Journal of Chemical Information and Modeling* 55.5, pp. 941–955. DOI: 10.1021/acs.jcim.5b00036 (cit. on p. 174).

Eppstein, David and Darren Strash (2011). "Listing All Maximal Cliques in Large Sparse Real-World Graphs". In: *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*. Ed. by Panos M. Pardalos and Steffen Rebennack. Vol. 6630. Lecture Notes in Computer Science. Springer,

pp. 364–375. ISBN: 978-3-642-20661-0. DOI: 10.1007/978-3-642-20662-7_31. (Cit. on p. 55).

Fahle, Torsten (2002). "Simple and Fast: Improving a Branch-And-Bound Algorithm for Maximum Clique". In: *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*. Ed. by Rolf H. Möhring and Rajeev Raman. Vol. 2461. Lecture Notes in Computer Science. Springer, pp. 485–498. ISBN: 3-540-44180-8. DOI: 10.1007/3-540-45749-6_44. (Cit. on p. 30).

Fang, Meng, Jie Yin, Xingquan Zhu, and Chengqi Zhang (2015). "TrGraph: Cross-Network Transfer Learning via Common Signature Subgraphs". In: *IEEE Trans. Knowl. Data Eng.* 27.9, pp. 2536–2549. DOI: 10.1109/TKDE.2015.2413789 (cit. on p. 171).

Fernández, Mirtha-Lina and Gabriel Valiente (2001). "A graph distance metric combining maximum common subgraph and minimum common supergraph". In: *Pattern Recognition Letters* 22.6/7, pp. 753–758 (cit. on p. 171).

Fischetti, Matteo, Michele Monaci, and Domenico Salvagnin (2014). "Self-splitting of Workload in Parallel Computation". In: *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*. Ed. by Helmut Simonis. Vol. 8451. Lecture Notes in Computer Science. Springer, pp. 394–404. ISBN: 978-3-319-07045-2. DOI: 10.1007/978-3-319-07046-9_28 (cit. on p. 83).

Foggia, Pasquale, Gennaro Percannella, and Mario Vento (2014). "Graph Matching and Learning in Pattern Recognition in the Last 10 Years". In: *IJPRAI* 28.1. DOI: 10.1142/S0218001414500013 (cit. on p. 125).

Frenkel, Karen A. (1986). "Complexity and Parallel Processing: An Interview with Richard Karp". In: *Commun. ACM* 29.2, pp. 112–117. ISSN: 0001-0782. DOI: 10.1145/5657.6326 (cit. on p. 19).

Freuder, Eugene C. (1982). "A Sufficient Condition for Backtrack-Free Search". In: *J. ACM* 29.1, pp. 24–32. DOI: 10.1145/322290.322292. (Cit. on p. 55).

Frieze, Alan M. (1990). "On the independence number of random graphs". In: *Discrete Mathematics* 81.2, pp. 171–175. DOI: 10.1016/0012-365X(90)90149-C (cit. on p. 61).

Fukagawa, Daiji, Takeyuki Tamura, Atsuhiro Takasu, Etsuji Tomita, and Tatsuya Akutsu (2011). "A clique-based method for the edit distance between unordered trees and its application to analysis of glycan structures". In: *BMC Bioinformatics* 12.S-1, S13. DOI: 10.1186/1471-2105-12-S1-S13 (cit. on p. 45).

Gao, Debin, Michael K. Reiter, and Dawn Xiaodong Song (2008). "BinHunt: Automatically Finding Semantic Differences in Binary Programs". In: *Information and Communications Security, 10th International Conference, ICICS 2008, Birmingham, UK, October 20-22, 2008, Proceedings*. Ed. by Liqun Chen, Mark Dermot Ryan, and Guilin Wang. Vol. 5308.

Lecture Notes in Computer Science. Springer, pp. 238–255. ISBN: 978-3-540-88624-2. DOI: 10.1007/978-3-540-88625-9_16 (cit. on p. 171).

Garey, M. R. and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman. ISBN: 0-7167-1044-7 (cit. on pp. 5, 115).

Gay, Steven, François Fages, Thierry Martinez, Sylvain Soliman, and Christine Solnon (2014). "On the subgraph epimorphism problem". In: *Discrete Applied Mathematics* 162, pp. 214–228. DOI: 10.1016/j.dam.2013.08.008 (cit. on pp. 143, 171).

Gebser, Martin, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider (2011). "Potassco: The Potsdam Answer Set Solving Collection". In: *AI Commun.* 24.2, pp. 107–124. DOI: 10.3233/AIC-2011-0491 (cit. on p. 154).

Geelen, Pieter Andreas (1992). "Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems". In: *ECAI*, pp. 31–35 (cit. on p. 198).

Gendreau, Michel, Patrick Soriano, and Louis Salvail (1993). "Solving the maximum clique problem using a tabu search approach". In: *Annals OR* 41.4, pp. 385–403. DOI: 10.1007/BF02023002 (cit. on p. 40).

Gent, Ian P. (1998). "Heuristic Solution of Open Bin Packing Problems". In: *J. Heuristics* 3.4, pp. 299–304. DOI: 10.1023/A:1009678411503 (cit. on p. 169).

Gent, Ian P., Warwick Harvey, and Tom Kelsey (2002). "Groups and Constraints: Symmetry Breaking during Search". In: *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*. Ed. by Pascal Van Hentenryck. Vol. 2470. Lecture Notes in Computer Science. Springer, pp. 415–430. ISBN: 3-540-44120-4 (cit. on p. 118).

Gent, Ian P., Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh (1996). "An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem". In: *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*. Ed. by Eugene C. Freuder. Vol. 1118. Lecture Notes in Computer Science. Springer, pp. 179–193. DOI: 10.1007/3-540-61551-2_74 (cit. on p. 151).

Gent, Ian P., Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh (2001). "Random Constraint Satisfaction: Flaws and Structure". In: *Constraints* 6.4, pp. 345–372. DOI: 10.1023/A:1011454308633 (cit. on p. 157).

Gent, Ian P., Ewan MacIntyre, Patrick Prosser, and Toby Walsh (1996). "The Constrainedness of Search". In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1*. Ed. by William J. Clancey and Daniel S.

Weld. AAAI Press / The MIT Press, pp. 246–252. ISBN: 0-262-51091-X (cit. on pp. 146, 156).

Gent, Ian P., Ian Miguel, and Peter Nightingale (2008). "Generalised arc consistency for the AllDifferent constraint: An empirical survey". In: *Artif. Intell.* 172.18, pp. 1973–2000. DOI: 10.1016/j.artint.2008.10.006 (cit. on p. 131).

Gent, Ian P., Karen E. Petrie, and Jean-François Puget (2006). "Symmetry in Constraint Programming". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, pp. 329–376. ISBN: 978-0-444-52726-4. DOI: 10.1016/S1574-6526(06)80014-3 (cit. on p. 118).

Gent, Ian P. and Toby Walsh (1994). "Easy Problems are Sometimes Hard". In: *Artif. Intell.* 70.1-2, pp. 335–345. DOI: 10.1016/0004-3702(94)90109-0 (cit. on p. 16).

Gent, Ian P. and Toby Walsh (1995). "Phase transitions from real computational problems". In: *Proceedings of the 8th International Symposium on Artificial Intelligence*, pp. 356–364 (cit. on p. 17).

Giugno, Rosalba, Vincenzo Bonnici, Nicola Bombieri, Alfredo Pulvirenti, Alfredo Ferro, and Dennis Shasha (2013). "GRAPES: A Software for Parallel Searching on Biological Graphs Targeting Multi-Core Architectures". In: *PLOS ONE* 8.10, pp. 1–11. DOI: 10.1371/journal.pone.0076911 (cit. on p. 165).

Goldberg, Mark K. and Reid D. Rivenburgh (1993). "Constructing cliques using restricted backtracking". In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Ed. by David S. Johnson and Michael A. Trick. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 89–102 (cit. on p. 30).

Gomes, Carla P. and Bart Selman (2001). "Algorithm portfolios". In: *Artif. Intell.* 126.1-2, pp. 43–62. DOI: 10.1016/S0004-3702(00)00081-3 (cit. on p. 168).

Gomes, Carla P., Bart Selman, and Henry A. Kautz (1998). "Boosting Combinatorial Search Through Randomization". In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*. Ed. by Jack Mostow and Chuck Rich. AAAI Press / The MIT Press, pp. 431–437. ISBN: 0-262-51098-7 (cit. on p. 91).

Grömping, Ulrike (2014). "R Package FrF2 for Creating and Analyzing Fractional Factorial 2-Level Designs". In: *Journal of Statistical Software* 56.1, pp. 1–56. ISSN: 1548-7660. DOI: 10.18637/jss.v056.i01 (cit. on p. 160).

Hamadi, Youssef, Saïd Jabbour, and Lakhdar Sais (2009). "ManySAT: a Parallel SAT Solver". In: *JSAT* 6.4, pp. 245–262 (cit. on p. 91).

Hamadi, Youssef and Christoph M. Wintersteiger (2012). “Seven Challenges in Parallel SAT Solving”. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* Ed. by Jörg Hoffmann and Bart Selman. AAAI Press (cit. on p. 23).

Han, Jiawei, Hong Cheng, Dong Xin, and Xifeng Yan (2007). “Frequent pattern mining: current status and future directions”. In: *Data Min. Knowl. Discov.* 15.1, pp. 55–86. DOI: 10.1007/s10618-006-0059-1 (cit. on p. 163).

Haralick, Robert M. and Gordon L. Elliott (1980). “Increasing Tree Search Efficiency for Constraint Satisfaction Problems”. In: *Artif. Intell.* 14.3, pp. 263–313. DOI: 10.1016/0004-3702(80)90051-X (cit. on pp. 10, 48, 168).

Harary, Frank and Ian C. Ross (1957). “A Procedure for Clique Detection Using the Group Matrix”. In: *Sociometry* 20.3, pp. 205–215. ISSN: 00380431. (Cit. on p. 29).

Hartung, Sepp, Christian Komusiewicz, and André Nichterlein (2015). “Parameterized Algorithmics and Computational Experiments for Finding 2-Clubs”. In: *J. Graph Algorithms Appl.* 19.1, pp. 155–190. DOI: 10.7155/jgaa.00352 (cit. on p. 96).

Harvey, William D. and Matthew L. Ginsberg (1995). “Limited Discrepancy Search”. In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*. Morgan Kaufmann, pp. 607–615 (cit. on pp. 73, 76, 90, 92).

He, Yongning, Feng Lin, Paul R. Chipman, Carol M. Bator, Timothy S. Baker, Menachem Shoham, Richard J. Kuhn, M. Edward Medof, and Michael G. Rossmann (2002). “Structure of decay-accelerating factor bound to echovirus 7: A virus-receptor complex”. In: *Proc Natl Acad Sci U S A* 99.16. 12119400[pmid], pp. 10325–10329. ISSN: 0027-8424. DOI: 10.1073/pnas.152161599 (cit. on p. 165).

Hoffmann, Ruth, Ciaran McCreesh, and Craig Reilly (2017). “Between Subgraph Isomorphism and Maximum Common Subgraph”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.* Ed. by Satinder P. Singh and Shaul Markovitch. AAAI Press, pp. 3907–3914 (cit. on pp. 27, 173).

Hong, Liang, Lei Zou, Xiang Lian, and Philip S. Yu (2015). “Subgraph Matching with Set Similarity in a Large Graph Database”. In: *IEEE Trans. Knowl. Data Eng.* 27.9, pp. 2507–2521. DOI: 10.1109/TKDE.2015.2391125 (cit. on p. 164).

Hooker, John N. (1995). “Testing heuristics: We have it all wrong”. In: *J. Heuristics* 1.1, pp. 33–42. DOI: 10.1007/BF02430364 (cit. on p. 92).

Hromkovič, Juraj (2004). *Algorithmics for Hard Problems - Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics, Second Edition*. Texts in

Theoretical Computer Science. An EATCS Series. Springer. ISBN: 978-3-642-07909-2. DOI: 10.1007/978-3-662-05269-3 (cit. on p. 5).

Janert, Philipp K. (2009). *Gnuplot in Action: Understanding Data with Graphs*. Greenwich, CT, USA: Manning Publications Co. (cit. on p. 16).

Jégou, Philippe (1993). "Decomposition of Domains Based on the Micro-Structure of Finite Constraint-Satisfaction Problems". In: *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993*. Ed. by Richard Fikes and Wendy G. Lehnert. AAAI Press / The MIT Press, pp. 731–736. ISBN: 0-262-51071-5 (cit. on pp. 12, 176).

Jiang, Haoliang, Haixun Wang, Philip S. Yu, and Shuigeng Zhou (2007). "GString: A Novel Approach for Efficient Search in Graph Databases". In: *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*. Ed. by Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis. IEEE Computer Society, pp. 566–575. ISBN: 1-4244-0802-4. DOI: 10.1109/ICDE.2007.367902 (cit. on p. 162).

Jiang, Hua, Chu-Min Li, and Felip Manyà (2016). "Combining Efficient Preprocessing and Incremental MaxSAT Reasoning for MaxClique in Large Graphs". In: *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS) 2016*. Ed. by Gal A. Kaminka, Maria Fox, Paolo Bouquet, Eyke Hüllermeier, Virginia Dignum, Frank Dignum, and Frank van Harmelen. Vol. 285. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 939–947. ISBN: 978-1-61499-671-2. DOI: 10.3233/978-1-61499-672-9-939 (cit. on p. 60).

Johnson, David S. and Michael A. Trick (1993). "Introduction to the Second DIMACS Challenge: Cliques, coloring, and satisfiability". In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Ed. by David S. Johnson and Michael A. Trick. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 1–10. (Cit. on pp. 30, 40).

Kasif, Simon (1990). "On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks". In: *Artif. Intell.* 45.3, pp. 275–286. DOI: 10.1016/0004-3702(90)90009-O (cit. on p. 22).

Katsarou, Foteini, Nikos Ntarmos, and Peter Triantafillou (2015). "Performance and Scalability of Indexed Subgraph Query Processing Methods". In: *PVLDB* 8.12, pp. 1566–1577 (cit. on pp. 163, 165).

Katsarou, Foteini, Nikos Ntarmos, and Peter Triantafillou (2017). "Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms". In: *Proceedings of the*

20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017. Ed. by Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß. OpenProceedings.org, pp. 25–36. ISBN: 978-3-89318-073-8. DOI: 10.5441/002/edbt.2017.04 (cit. on pp. 160, 167–169).

Kessel, Philippe Van and Claude-Guy Quimper (2012). "Filtering Algorithms Based on the Word-RAM Model". In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* Ed. by Jörg Hoffmann and Bart Selman. AAAI Press (cit. on p. 132).

Knuth, Donald E. (1994). "The Sandwich Theorem". In: *Electr. J. Comb.* 1 (cit. on p. 57).

Koch, Ina (2001). "Enumerating all connected maximal common subgraphs in two graphs". In: *Theor. Comput. Sci.* 250.1-2, pp. 1–30. DOI: 10.1016/S0304-3975(00)00286-3 (cit. on pp. 177, 179, 184).

Konc, Janez and Dušanka Janežič (2007a). "A Branch and Bound Algorithm for Matching Protein Structures". In: *Adaptive and Natural Computing Algorithms, 8th International Conference, ICANNGA 2007, Warsaw, Poland, April 11-14, 2007, Proceedings, Part II*. Ed. by Bartlomiej Beliczynski, Andrzej Dzielinski, Marcin Iwanowski, and Bernardete Ribeiro. Vol. 4432. Lecture Notes in Computer Science. Springer, pp. 399–406. ISBN: 978-3-540-71590-0. DOI: 10.1007/978-3-540-71629-7_45. (Cit. on p. 45).

Konc, Janez and Dušanka Janežič (2007b). "An improved branch and bound algorithm for the maximum clique problem". In: *MATCH Commun. Math. Comput. Chem.* 58.3, pp. 569–590. (Cit. on p. 58).

Korf, Richard E. (1996). "Improved Limited Discrepancy Search". In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1.* Ed. by William J. Clancey and Daniel S. Weld. AAAI Press / The MIT Press, pp. 286–291. ISBN: 0-262-51091-X (cit. on p. 76).

Korf, Richard E. (2014). "How Do You Know Your Search Algorithm and Code Are Correct?" In: *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014.* Ed. by Stefan Edelkamp and Roman Barták. AAAI Press. ISBN: 978-1-57735-676-9 (cit. on p. 15).

Kotthoff, Lars, Ciaran McCreesh, and Christine Solnon (2016). "Portfolios of Subgraph Isomorphism Algorithms". In: *Learning and Intelligent Optimization - 10th International Conference, LION 10, Ischia, Italy, May 29 - June 1, 2016, Revised Selected Papers*. Ed. by Paola Festa, Meinolf Sellmann, and Joaquin Vanschoren. Vol. 10079. Lecture Notes in Computer Science. Springer, pp. 107–122. ISBN: 978-3-319-50348-6. DOI: 10.1007/978-3-319-50349-3_8 (cit. on pp. 26, 125, 127, 130, 133, 134, 137, 143, 168, 169).

Kriege, Nils (2015). "Comparing graphs". PhD thesis. Technische Universität Dortmund (cit. on pp. 4, 171).

Krissinel, Evgeny B. and Kim Henrick (2004). "Common subgraph isomorphism detection by backtracking search". In: *Softw., Pract. Exper.* 34.6, pp. 591–607. DOI: 10.1002/spe.588 (cit. on p. 174).

Kubale, Marek and Boguslaw Jackowski (1985). "A Generalized Implicit Enumeration Algorithm for Graph Coloring". In: *Commun. ACM* 28.4, pp. 412–418. DOI: 10.1145/3341.3350. (Cit. on p. 56).

Lai, Ten-Hwang and Sartaj Sahni (1984). "Anomalies in Parallel Branch-and-Bound Algorithms". In: *Commun. ACM* 27.6, pp. 594–602. DOI: 10.1145/358080.358103 (cit. on pp. 24, 67, 92).

Langer, Akhil, Ramprasad Venkataraman, Udatta S. Palekar, and Laxmikant V. Kalé (2013). "Parallel branch-and-bound for two-stage stochastic integer optimization". In: *20th Annual International Conference on High Performance Computing, HiPC 2013, Bengaluru (Bangalore), Karnataka, India, December 18-21, 2013*. IEEE Computer Society, pp. 266–275. ISBN: 978-1-4799-0730-4. DOI: 10.1109/HiPC.2013.6799130 (cit. on p. 84).

Larrosa, Javier and Pedro Meseguer (1998). "Partial Lazy Forward Checking for MAX-CSP". In: *ECAI*, pp. 229–233 (cit. on p. 169).

Larrosa, Javier and Gabriel Valiente (2002). "Constraint Satisfaction Algorithms for Graph Pattern Matching". In: *Mathematical Structures in Computer Science* 12.4, pp. 403–422. DOI: 10.1017/S0960129501003577 (cit. on pp. 125, 133, 138).

Lee, Jinsoo, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee (2012). "An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases". In: *PVLDB* 6.2, pp. 133–144 (cit. on p. 165).

Leroy, Rudi, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens (2014). "Work Stealing Strategies For Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System". In: *Proceedings of the 2014 PPOPP International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2014, Orlando, Florida, USA, February 15, 2014*. Ed. by Pavan Balaji, Minyi Guo, and Zhiyi Huang. ACM, p. 111. ISBN: 978-1-4503-2657-5. DOI: 10.1145/2560683.2560694 (cit. on p. 83).

Levi, Giorgio (1973). "A note on the derivation of maximal common subgraphs of two directed or undirected graphs". In: *CALCOLO* 9.4, pp. 341–352. ISSN: 1126-5434. DOI: 10.1007/BF02575586 (cit. on pp. 20, 154, 175).

Lewandowski, Gary and Anne Condon (1993). "Experiments with parallel graph coloring heuristics and applications of graph coloring". In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Ed. by David S. Johnson and Michael A. Trick. Vol. 26. DIMACS Series in Discrete

Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 309–334 (cit. on p. 91).

Leyton-Brown, Kevin, Holger H. Hoos, Frank Hutter, and Lin Xu (2014). "Understanding the empirical hardness of *NP*-complete problems". In: *Commun. ACM* 57.5, pp. 98–107. DOI: 10.1145/2594413.2594424 (cit. on p. 148).

Li, Chu-Min, Zhiwen Fang, and Ke Xu (2013). "Combining MaxSAT Reasoning and Incremental Upper Bound for the Maximum Clique Problem". In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*. IEEE Computer Society, pp. 939–946. ISBN: 978-1-4799-2971-9. DOI: 10.1109/ICTAI.2013.143 (cit. on pp. 44, 57).

Li, Chu-Min, Hua Jiang, and Felip Manyà (2017). "On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem". In: *Computers & Operations Research* 84, pp. 1–15. ISSN: 0305-0548. DOI: http://dx.doi.org/10.1016/j.cor.2017.02.017 (cit. on p. 57).

Li, Chu-Min, Hua Jiang, and Ruchu Xu (2015). "Incremental MaxSAT Reasoning to Reduce Branches in a Branch-and-Bound Algorithm for MaxClique". In: *Learning and Intelligent Optimization - 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers*. Ed. by Clarisse Dhaenens, Laetitia Jourdan, and Marie-Eléonore Marmion. Vol. 8994. Lecture Notes in Computer Science. Springer, pp. 268–274. ISBN: 978-3-319-19083-9. DOI: 10.1007/978-3-319-19084-6_26 (cit. on p. 57).

Li, Chu-Min and Zhe Quan (2010a). "An Efficient Branch-and-Bound Algorithm Based on MaxSAT for the Maximum Clique Problem". In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. Ed. by Maria Fox and David Poole. AAAI Press (cit. on p. 57).

Li, Chu-Min and Zhe Quan (2010b). "Combining Graph Structure Exploitation and Propositional Reasoning for the Maximum Clique Problem". In: *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*. IEEE Computer Society, pp. 344–351. ISBN: 978-0-7695-4263-8. DOI: 10.1109/ICTAI.2010.57 (cit. on p. 57).

Li, Guo-Jie and Benjamin W. Wah (1984). "How to Cope With Anomalies in Parallel Approximate Branch-and-Bound Algorithms". In: *Proceedings of the National Conference on Artificial Intelligence. Austin, TX, August 6-10, 1984*. Ed. by Ronald J. Brachman. AAAI Press, pp. 212–215. ISBN: 0-262-51053-7 (cit. on p. 67).

Li, Guo-Jie and Benjamin W. Wah (1986). "Coping with Anomalies in Parallel Branch-and-Bound Algorithms". In: *IEEE Trans. Computers* 35.6, pp. 568–573. DOI: 10.1109/TC.1986.5009434 (cit. on p. 67).

Lobachev, Oleg (2012). "Parallel Computation Skeletons with Premature Termination Property". In: *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*. Ed. by Tom Schrijvers and Peter Thiemann. Vol. 7294. Lecture Notes in Computer Science. Springer, pp. 197–212. ISBN: 978-3-642-29821-9. DOI: 10.1007/978-3-642-29822-6_17 (cit. on p. 144).

López-Presa, José Luis and Antonio Fernández Anta (2009). "Fast Algorithm for Graph Isomorphism Testing". In: *Experimental Algorithms, 8th International Symposium, SEA 2009, Dortmund, Germany, June 4-6, 2009. Proceedings*. Ed. by Jan Vahrenhold. Vol. 5526. Lecture Notes in Computer Science. Springer, pp. 221–232. ISBN: 978-3-642-02010-0. DOI: 10.1007/978-3-642-02011-7_21 (cit. on p. 198).

Lu, Si Wei, Ying Ren, and Ching Y. Suen (1991). "Hierarchical attributed graph representation and recognition of handwritten chinese characters". In: *Pattern Recognition* 24.7, pp. 617–632. DOI: 10.1016/0031-3203(91)90029-5 (cit. on p. 171).

Luce, R. Duncan (1950). "Connectivity and generalized cliques in sociometric group structure". In: *Psychometrika* 15.2, pp. 169–190. ISSN: 0033-3123. DOI: 10.1007/BF02289199 (cit. on pp. 3, 96).

Luo, Chen, Xin Wang, Chun Su, and Zhonghua Ni (2017). "A Fixture Design Retrieving Method Based on Constrained Maximum Common Subgraph". In: *IEEE Transactions on Automation Science and Engineering* PP.99, pp. 1–13. ISSN: 1545-5955. DOI: 10.1109/TASE.2017.2674961 (cit. on pp. 171, 179).

Mackworth, Alan K. (1977). "Consistency in Networks of Relations". In: *Artif. Intell.* 8.1, pp. 99–118. DOI: 10.1016/0004-3702(77)90007-8 (cit. on p. 8).

Mahdavi Pajouh, Foad and Balabhaskar Balasundaram (2012). "On inclusionwise maximal and maximum cardinality k-clubs in graphs". In: *Discrete Optimization* 9.2, pp. 84–97. DOI: 10.1016/j.disopt.2012.02.002 (cit. on p. 96).

Malapert, Arnaud, Jean-Charles Régin, and Mohamed Rezgui (2016). "Embarrassingly Parallel Search in Constraint Programming". In: *J. Artif. Intell. Res. (JAIR)* 57, pp. 421–464. DOI: 10.1613/jair.5247 (cit. on pp. 70, 82, 105, 143, 200).

Malitsky, Yuri (2014). *Instance-Specific Algorithm Configuration*. Springer. ISBN: 978-3-319-11229-9. DOI: 10.1007/978-3-319-11230-5 (cit. on p. 130).

Mannino, Carlo and Antonio Sassano (1993). "Edge projection and the maximum cardinality stable set problem". In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Ed. by David S. Johnson and Michael A. Trick. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 205–220 (cit. on p. 30).

Mannino, Carlo and Antonio Sassano (1995). "Solving hard set covering problems". In: *Oper. Res. Lett.* 18.1, pp. 1–5. DOI: 10.1016/0167-6377(95)00034-H (cit. on p. 44).

Maslov, Evgeny, Mikhail Batsyn, and Panos M. Pardalos (2014). “Speeding up branch and bound algorithms for solving the maximum clique problem”. In: *J. Global Optimization* 59.1, pp. 1–21. DOI: 10.1007/s10898-013-0075-9. (Cit. on pp. 43, 58, 90).

Matula, David W. and Leland L. Beck (1983). “Smallest-Last Ordering and clustering and Graph Coloring Algorithms”. In: *J. ACM* 30.3, pp. 417–427. DOI: 10.1145/2402.322385. (Cit. on p. 55).

McCool, Michael, James Reinders, and Arch Robison (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. (cit. on pp. 21, 89).

McCreesh, Ciaran, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon (2016). “Clique and Constraint Models for Maximum Common (Connected) Subgraph Problems”. In: *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*. Ed. by Michel Rueher. Vol. 9892. Lecture Notes in Computer Science. Springer, pp. 350–368. ISBN: 978-3-319-44952-4. DOI: 10.1007/978-3-319-44953-1_23 (cit. on pp. 26, 27, 173).

McCreesh, Ciaran and Patrick Prosser (2013). “Multi-Threading a State-of-the-Art Maximum Clique Algorithm”. In: *Algorithms* 6.4, pp. 618–635. DOI: 10.3390/a6040618 (cit. on pp. 40, 41, 43, 44, 64, 66, 68, 69, 71, 84, 92).

McCreesh, Ciaran and Patrick Prosser (2014a). “An Exact Branch and Bound Algorithm with Symmetry Breaking for the Maximum Balanced Induced Biclique Problem”. In: *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*. Ed. by Helmut Simonis. Vol. 8451. Lecture Notes in Computer Science. Springer, pp. 226–234. ISBN: 978-3-319-07045-2. DOI: 10.1007/978-3-319-07046-9_16 (cit. on pp. 26, 95).

McCreesh, Ciaran and Patrick Prosser (2014b). “Reducing the Branching in a Branch and Bound Algorithm for the Maximum Clique Problem”. In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. Ed. by Barry O’Sullivan. Vol. 8656. Lecture Notes in Computer Science. Springer, pp. 549–563. ISBN: 978-3-319-10427-0. DOI: 10.1007/978-3-319-10428-7_40 (cit. on pp. 25, 29).

McCreesh, Ciaran and Patrick Prosser (2015a). “A Parallel, Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs”. In: *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*. Ed. by Gilles Pesant. Vol. 9255. Lecture Notes in Computer Science. Springer, pp. 295–312. ISBN: 978-3-319-23218-8. DOI: 10.1007/978-3-319-23219-5_21 (cit. on pp. 26, 127, 143, 147).

McCreesh, Ciaran and Patrick Prosser (2015b). "A parallel branch and bound algorithm for the maximum labelled clique problem". In: *Optimization Letters* 9.5, pp. 949–960. DOI: 10.1007/s11590-014-0837-4 (cit. on pp. 26, 95, 133, 134).

McCreesh, Ciaran and Patrick Prosser (2015c). "The Shape of the Search Tree for the Maximum Clique Problem and the Implications for Parallel Branch and Bound". In: *TOPC* 2.1, 8:1–8:27. DOI: 10.1145/2742359 (cit. on pp. 25, 64).

McCreesh, Ciaran and Patrick Prosser (2016). "Finding Maximum k-Cliques Faster Using Lazy Global Domination". In: *Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016, Tarrytown, NY, USA, July 6-8, 2016*. Ed. by Jorge A. Baier and Adi Botea. AAAI Press, pp. 72–80. ISBN: 978-1-57735-769-8 (cit. on pp. 26, 29, 95).

McCreesh, Ciaran, Patrick Prosser, and James Trimble (2016). "Heuristics and Really Hard Instances for Subgraph Isomorphism Problems". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. Ed. by Subbarao Kambhampati. IJCAI/AAAI Press, pp. 631–638. ISBN: 978-1-57735-770-4 (cit. on pp. 26, 146).

McCreesh, Ciaran, Patrick Prosser, and James Trimble (2017). "A Partitioning Algorithm for Maximum Common Subgraph Problems". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, 19-25 August 2017*. To appear (cit. on pp. 173, 198).

McGeoch, Catherine C. (2012). *A Guide to Experimental Algorithmics*. Cambridge University Press. ISBN: 978-0-521-17301-8 (cit. on pp. 14, 17).

McGregor, James J. (1979). "Relational consistency algorithms and their application in finding subgraph and graph isomorphisms". In: *Inf. Sci.* 19.3, pp. 229–250. DOI: 10.1016/0020-0255(79)90023-9 (cit. on p. 125).

McGregor, James J. (1982). "Backtrack Search Algorithms and the Maximal Common Subgraph Problem". In: *Softw., Pract. Exper.* 12.1, pp. 23–34. DOI: 10.1002/spe.4380120103 (cit. on pp. 125, 174).

McKay, Brendan D. and Adolfo Piperno (2014). "Practical graph isomorphism, II". In: *J. Symb. Comput.* 60, pp. 94–112. DOI: 10.1016/j.jsc.2013.09.003 (cit. on pp. 18, 143, 191, 198).

Menouer, Tarek, Mohamed Rezgui, Bertrand Le Cun, and Jean-Charles Régin (2016). "Mixing Static and Dynamic Partitioning to Parallelize a Constraint Programming Solver". In: *International Journal of Parallel Programming* 44.3, pp. 486–505. DOI: 10.1007/s10766-015-0356-7 (cit. on p. 70).

Mercure, Hélène, Jean-Marie Bourjolly, Paul Gill, and Gilbert Laporte (1993). "An exact quadratic 0-1 algorithm for the stable set problem". In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13,*

1993. Ed. by David S. Johnson and Michael A. Trick. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 53–74 (cit. on p. 30).

Meyerhenke, Henning (2011). *Clustering Instances*. URL: http://www.cc.gatech.edu/dimacs10/archive/clustering.shtml (visited on March 31, 2017) (cit. on p. 104).

Michel, Laurent, Andrew See, and Pascal Van Hentenryck (2009). "Transparent Parallelization of Constraint Programming". In: *INFORMS Journal on Computing* 21.3, pp. 363–382. DOI: 10.1287/ijoc.1080.0313 (cit. on p. 90).

Minot, Mael, Samba Ndojh Ndiaye, and Christine Solnon (2015). "A Comparison of Decomposition Methods for the Maximum Common Subgraph Problem". In: *27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2015, Vietri sul Mare, Italy, November 9-11, 2015*. IEEE, pp. 461–468. ISBN: 978-1-5090-0163-7. DOI: 10.1109/ICTAI.2015.75 (cit. on pp. 172, 200).

Mitchell, David G., Bart Selman, and Hector J. Levesque (1992). "Hard and Easy Distributions of SAT Problems". In: *Proceedings of the 10th National Conference on Artificial Intelligence. San Jose, CA, July 12-16, 1992*. Ed. by William R. Swartout. AAAI Press / The MIT Press, pp. 459–465. ISBN: 0-262-51063-4 (cit. on pp. 35, 146).

Moisan, Thierry, Jonathan Gaudreault, and Claude-Guy Quimper (2013). "Parallel Discrepancy-Based Search". In: *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*. Ed. by Christian Schulte. Vol. 8124. Lecture Notes in Computer Science. Springer, pp. 30–46. ISBN: 978-3-642-40626-3. DOI: 10.1007/978-3-642-40627-0_6 (cit. on pp. 76, 90).

Mokken, Robert J. (1979). "Cliques, clubs and clans". In: *Quality and Quantity* 13.2, pp. 161–173. ISSN: 0033-5177. DOI: 10.1007/BF00139635 (cit. on p. 96).

Murray, Alastair Colin (2012). "Customising compilers for customisable processors". PhD thesis. The University of Edinburgh (cit. on pp. 125, 160).

Murray, Alastair Colin and Björn Franke (2012). "Compiling for automatically generated instruction set extensions". In: *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*. Ed. by Carol Eidt, Anne M. Holler, Uma Srinivasan, and Saman P. Amarasinghe. ACM, pp. 13–22. ISBN: 978-1-4503-1206-6. DOI: 10.1145/2259016.2259019 (cit. on p. 125).

Mycielski, Jan (1955). "Sur le coloriage des graphes". French. In: *Colloq. Math.* Vol. 3. 161-162, p. 9 (cit. on p. 11).

Ndiaye, Samba Ndojh and Christine Solnon (2011). "CP Models for Maximum Common Subgraph Problems". In: *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Pro-*

ceedings. Ed. by Jimmy Ho-Man Lee. Vol. 6876. Lecture Notes in Computer Science. Springer, pp. 637–644. ISBN: 978-3-642-23785-0. DOI: 10.1007/978-3-642-23786-7_48 (cit. on pp. 173, 174, 177).

Nguyen, T. and Yves Deville (1998). "A Distributed Arc-Consistency Algorithm". In: *Sci. Comput. Program.* 30.1-2, pp. 227–250. DOI: 10.1016/S0167-6423(97)00012-9 (cit. on p. 21).

Nikolaev, Alexey, Mikhail Batsyn, and Pablo San Segundo (2015). "Reusing the Same Coloring in the Child Nodes of the Search Tree for the Maximum Clique Problem". In: *Learning and Intelligent Optimization - 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers*. Ed. by Clarisse Dhaenens, Laetitia Jourdan, and Marie-Eléonore Marmion. Vol. 8994. Lecture Notes in Computer Science. Springer, pp. 275–280. ISBN: 978-3-319-19083-9. DOI: 10.1007/978-3-319-19084-6_27. (Cit. on pp. 58, 123).

Okubo, Yoshiaki and Makoto Haraguchi (2006). "Finding Conceptual Document Clusters with Improved Top-N Formal Concept Search". In: *2006 IEEE / WIC / ACM International Conference on Web Intelligence (WI) 2006, 18-22 December 2006, Hong Kong, China*. IEEE Computer Society, pp. 347–351. ISBN: 0-7695-2747-7. DOI: 10.1109/WI.2006.81 (cit. on p. 45).

Östergård, Patric R. J. (2002). "A fast algorithm for the maximum clique problem". In: *Discrete Applied Mathematics* 120.1-3, pp. 197–207. DOI: 10.1016/S0166-218X(01)00290-6. (Cit. on pp. 30, 59).

Otten, Lars and Rina Dechter (2017). "AND/OR Branch-and-Bound on a Computational Grid". In: *J. Artif. Intell. Res. (JAIR)* 59. To appear (cit. on p. 72).

Pardalos, Panos M., Jonas Rappe, and Mauricio G. C. Resende (1998). "An Exact Parallel Algorithm for the Maximum Clique Problem". In: *High Performance Algorithms and Software in Nonlinear Optimization*. Ed. by Renato De Leone, Almerico Murli, Panos M. Pardalos, and Gerardo Toraldo. Boston, MA: Springer US, pp. 279–300. ISBN: 978-1-4613-3279-4. DOI: 10.1007/978-1-4613-3279-4_18 (cit. on p. 66).

Pardalos, Panos M. and Gregory P. Rodgers (1992). "A branch and bound algorithm for the maximum clique problem". In: *Computers & OR* 19.5, pp. 363–375. DOI: 10.1016/0305-0548(92)90067-F. (Cit. on p. 29).

Pardalos, Panos M. and Jue Xue (1994). "The maximum clique problem". In: *J. Global Optimization* 4.3, pp. 301–328. DOI: 10.1007/BF01098364 (cit. on p. 30).

Park, Young Hee, Douglas S. Reeves, and Mark Stamp (2013). "Deriving common malware behavior through graph clustering". In: *Computers & Security* 39, pp. 419–430. DOI: 10.1016/j.cose.2013.09.006 (cit. on p. 171).

Pataki, Gábor, Egon Balas, Sabastian Ceria, and Gerard Cornuejols (1993). "Polyhedral methods for the maximum clique problem". In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Ed. by David S. Johnson and Michael A. Trick. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 11–28 (cit. on p. 30).

Pattabiraman, Bharath, Md. Mostofa Ali Patwary, Assefaw Hadish Gebremedhin, Wei-keng Liao, and Alok N. Choudhary (2013). "Fast Algorithms for the Maximum Clique Problem on Massive Sparse Graphs". In: *Algorithms and Models for the Web Graph - 10th International Workshop, WAW 2013, Cambridge, MA, USA, December 14-15, 2013, Proceedings*. Ed. by Anthony Bonato, Michael Mitzenmacher, and Pawel Pralat. Vol. 8305. Lecture Notes in Computer Science. Springer, pp. 156–169. ISBN: 978-3-319-03535-2. DOI: 10.1007/978-3-319-03536-9_13 (cit. on p. 60).

Pattabiraman, Bharath, Md. Mostofa Ali Patwary, Assefaw Hadish Gebremedhin, Wei-keng Liao, and Alok N. Choudhary (2015). "Fast Algorithms for the Maximum Clique Problem on Massive Graphs with Applications to Overlapping Community Detection". In: *Internet Mathematics* 11.4-5, pp. 421–448. DOI: 10.1080/15427951.2014.986778 (cit. on p. 60).

Peng, Peng, Lei Zou, Lei Chen, Xuemin Lin, and Dongyan Zhao (2016). "Answering subgraph queries over massive disk resident graphs". In: *World Wide Web* 19.3, pp. 417–448. DOI: 10.1007/s11280-014-0322-0 (cit. on p. 164).

Petit, Thierry, Jean-Charles Régin, and Christian Bessière (2001). "Specific Filtering Algorithms for Over-Constrained Problems". In: *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*. Ed. by Toby Walsh. Vol. 2239. Lecture Notes in Computer Science. Springer, pp. 451–463. ISBN: 3-540-42863-1. DOI: 10.1007/3-540-45578-7_31 (cit. on p. 175).

Picker, Marten (2015). "Algorithms and Experiments for Finding Robust 2-Clubs". MA thesis. Technische Universität Berlin (cit. on p. 96).

Piperno, Adolfo (2008). "Search Space Contraction in Canonical Labeling of Graphs (Preliminary Version)". In: *CoRR* abs/0804.4881 (cit. on pp. 13, 18).

Piva, Breno and Cid Carvalho de Souza (2012). "Polyhedral study of the maximum common induced subgraph problem". In: *Annals OR* 199.1, pp. 77–102. DOI: 10.1007/s10479-011-1019-8 (cit. on p. 174).

Poldner, Michael and Herbert Kuchen (2008). "Algorithmic Skeletons for Branch and Bound". In: *Software and Data Technologies: First International Conference, ICSOFT 2006, Setúbal, Portugal, September 11-14, 2006, Revised Selected Papers*. Ed. by Joaquim Filipe, Boris Shishkov, and Markus Helfert. Berlin, Heidelberg: Springer Berlin Heidelberg,

pp. 204–219. ISBN: 978-3-540-70621-2. DOI: 10.1007/978-3-540-70621-2_17 (cit. on pp. 81, 84).

Prosser, Patrick (2012). "Exact Algorithms for Maximum Clique: A Computational Study". In: *Algorithms* 5.4, pp. 545–587. DOI: 10.3390/a5040545. (Cit. on pp. 30, 33–35, 55–57).

Prosser, Patrick and Chris Unsworth (2006). "A Connectivity Constraint Using Bridges". In: *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS) 2006), Proceedings*. Ed. by Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso. Vol. 141. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 707–708. ISBN: 1-58603-642-4 (cit. on p. 180).

Prosser, Patrick and Chris Unsworth (2011). "Limited discrepancy search revisited". In: *ACM Journal of Experimental Algorithmics* 16. DOI: 10.1145/1963190.2019581 (cit. on p. 76).

Puget, Jean-Francois (1998). "A Fast Algorithm for the Bound Consistency of alldiff Constraints". In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*. Ed. by Jack Mostow and Chuck Rich. AAAI Press / The MIT Press, pp. 359–366. ISBN: 0-262-51098-7 (cit. on p. 131).

Pullan, Wayne, Franco Mascia, and Mauro Brunato (2011). "Cooperating local search for the maximum clique problem". In: *J. Heuristics* 17.2, pp. 181–199. DOI: 10.1007/s10732-010-9131-5 (cit. on p. 91).

Quimper, Claude-Guy and Toby Walsh (2005). "The All Different and Global Cardinality Constraints on Set, Multiset and Tuple Variables". In: *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2005, Uppsala, Sweden, June 20-22, 2005, Revised Selected and Invited Papers*. Ed. by Brahim Hnich, Mats Carlsson, François Fages, and Francesca Rossi. Vol. 3978. Lecture Notes in Computer Science. Springer, pp. 1–13. ISBN: 3-540-34215-X. DOI: 10.1007/11754602_1 (cit. on p. 131).

Raymond, John W. and Peter Willett (2002). "Maximum common subgraph isomorphism algorithms for the matching of chemical structures". In: *Journal of Computer-Aided Molecular Design* 16.7, pp. 521–533. DOI: 10.1023/A:1021271615909 (cit. on pp. 171, 175, 179).

Rayward-Smith, Victor J., S. A. Rush, and Geoff P. McKeown (1993). "Efficiency considerations in the implementation of parallel branch-and-bound". In: *Annals OR* 43.2, pp. 123–145. DOI: 10.1007/BF02024489 (cit. on p. 83).

Redmond, Ursula and Pádraig Cunningham (2013). "Temporal subgraph isomorphism". In: *Advances in Social Networks Analysis and Mining 2013, ASONAM '13, Niagara, ON, Canada - August 25 - 29, 2013*. Ed. by Jon G. Rokne and Christos Faloutsos. ACM,

pp. 1451–1452. ISBN: 978-1-4503-2240-9. DOI: 10.1145/2492517.2492586 (cit. on p. 142).

Redmond, Ursula and Pádraig Cunningham (2016). "Subgraph Isomorphism in Temporal Networks". In: *CoRR* abs/1605.02174 (cit. on p. 142).

Régin, Jean-Charles (1994). "A Filtering Algorithm for Constraints of Difference in CSPs". In: *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*. Ed. by Barbara Hayes-Roth and Richard E. Korf. AAAI Press / The MIT Press, pp. 362–367. ISBN: 0-262-61102-3 (cit. on pp. 8, 131, 163, 165, 196).

Régin, Jean-Charles (1995). "Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique". French. PhD thesis. Université Montpellier 2 (cit. on p. 125).

Régin, Jean-Charles (2003). "Using Constraint Programming to Solve the Maximum Clique Problem". In: *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*. Ed. by Francesca Rossi. Vol. 2833. Lecture Notes in Computer Science. Springer, pp. 634–648. ISBN: 3-540-20202-1. DOI: 10.1007/978-3-540-45193-8_43. (Cit. on p. 30).

Régin, Jean-Charles, Mohamed Rezgui, and Arnaud Malapert (2013). "Embarrassingly Parallel Search". In: *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*. Ed. by Christian Schulte. Vol. 8124. Lecture Notes in Computer Science. Springer, pp. 596–610. ISBN: 978-3-642-40626-3. DOI: 10.1007/978-3-642-40627-0_45 (cit. on pp. 70, 82, 85).

Régin, Jean-Charles, Mohamed Rezgui, and Arnaud Malapert (2014). "Improvement of the Embarrassingly Parallel Search for Data Centers". In: *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. Ed. by Barry O'Sullivan. Vol. 8656. Lecture Notes in Computer Science. Springer, pp. 622–635. ISBN: 978-3-319-10427-0. DOI: 10.1007/978-3-319-10428-7_45 (cit. on p. 70).

Regula, Gergely and Béla Lantos (2013). "Formation Control of Quadrotor Helicopters with Guaranteed Collision Avoidance via Safe Path". In: *Electrical Engineering and Computer Science* 56.4, pp. 113–124 (cit. on p. 45).

Rossi, Francesca, Peter van Beek, and Toby Walsh, eds. (2006). *Handbook of Constraint Programming*. Vol. 2. Foundations of Artificial Intelligence. Elsevier. ISBN: 978-0-444-52726-4 (cit. on p. 7).

Rossi, Ryan A., David F. Gleich, Assefaw Hadish Gebremedhin, and Md. Mostofa Ali Patwary (2013). "A Fast Parallel Maximum Clique Algorithm for Large Sparse Graphs and Temporal Strong Components". In: *CoRR* abs/1302.6256 (cit. on p. 66).

Sabin, Daniel and Eugene C. Freuder (1994). "Contradicting Conventional Wisdom in Constraint Satisfaction". In: *Principles and Practice of Constraint Programming, Second International Workshop, PPCP'94, Rosario, Orcas Island, Washington, USA, May 2-4, 1994, Proceedings*. Ed. by Alan Borning. Vol. 874. Lecture Notes in Computer Science. Springer, pp. 10–20. ISBN: 3-540-58601-6. DOI: 10.1007/3-540-58601-6_86 (cit. on pp. 8, 175).

San Segundo, Pablo (2012). "A new DSATUR-based algorithm for exact vertex coloring". In: *Computers & OR* 39.7, pp. 1724–1733. DOI: 10.1016/j.cor.2011.10.008. (Cit. on p. 56).

San Segundo, Pablo, Jorge Artieda, Mikhail Batsyn, and Panos M. Pardalos (2017). "An enhanced bitstring encoding for exact maximum clique search in sparse graphs". In: *Optimization Methods and Software* 32.2, pp. 312–335. DOI: 10.1080/10556788.2017.1281924 (cit. on p. 60).

San Segundo, Pablo, Jorge Artieda, Rafael León, and Cristóbal Tapia (2016). "An Enhanced Infra-Chromatic Bound for the Maximum Clique Problem". In: *Machine Learning, Optimization, and Big Data - Second International Workshop, MOD 2016, Volterra, Italy, August 26-29, 2016, Revised Selected Papers*. Ed. by Panos M. Pardalos, Piero Conca, Giovanni Giuffrida, and Giuseppe Nicosia. Vol. 10122. Lecture Notes in Computer Science. Springer, pp. 306–316. ISBN: 978-3-319-51468-0. DOI: 10.1007/978-3-319-51469-7_26 (cit. on p. 57).

San Segundo, Pablo, Alvaro Lopez, Jorge Artieda, and Panos M. Pardalos (2017). "A parallel maximum clique algorithm for large and massive sparse graphs". In: *Optimization Letters* 11.2, pp. 343–358. DOI: 10.1007/s11590-016-1019-3 (cit. on p. 60).

San Segundo, Pablo, Alvaro Lopez, and Mikhail Batsyn (2014). "Initial Sorting of Vertices in the Maximum Clique Problem Reviewed". In: *Learning and Intelligent Optimization - 8th International Conference, Lion 8, Gainesville, FL, USA, February 16-21, 2014. Revised Selected Papers*. Ed. by Panos M. Pardalos, Mauricio G. C. Resende, Chrysafis Vogiatzis, and Jose L. Walteros. Vol. 8426. Lecture Notes in Computer Science. Springer, pp. 111–120. ISBN: 978-3-319-09583-7. DOI: 10.1007/978-3-319-09584-4_12 (cit. on p. 55).

San Segundo, Pablo, Alvaro Lopez, Mikhail Batsyn, Alexey Nikolaev, and Panos M. Pardalos (2016). "Improved initial vertex ordering for exact maximum clique search". In: *Appl. Intell.* 45.3, pp. 868–880. DOI: 10.1007/s10489-016-0796-9 (cit. on p. 55).

San Segundo, Pablo, Alvaro Lopez, and Panos M. Pardalos (2016). "A new exact maximum clique algorithm for large and massive sparse graphs". In: *Computers & OR* 66, pp. 81–94. DOI: 10.1016/j.cor.2015.07.013 (cit. on pp. 60, 105).

San Segundo, Pablo, Fernando Matía, Diego Rodríguez-Losada, and Miguel Hernando (2013). "An improved bit parallel exact maximum clique algorithm". In: *Optimization Letters* 7.3, pp. 467–479. DOI: 10.1007/s11590-011-0431-y. (Cit. on pp. 30, 33, 57).

San Segundo, Pablo, Alexey Nikolaev, and Mikhail Batsyn (2015). "Infra-chromatic bound for exact maximum clique search". In: *Computers & OR* 64, pp. 293–303. DOI: 10.1016/j.cor.2015.06.009 (cit. on p. 57).

San Segundo, Pablo, Alexey Nikolaev, Mikhail Batsyn, and Panos M. Pardalos (2016). "Improved Infra-Chromatic Bound for Exact Maximum Clique Search". In: *Informatica, Lith. Acad. Sci.* 27.2, pp. 463–487 (cit. on pp. 44, 57).

San Segundo, Pablo, Diego Rodríguez-Losada, Ramón Galán, Fernando Matía, and Agustín Jiménez (2007). "Exploiting CPU Bit Parallel Operations to Improve Efficiency in Search". In: *ICTAI (1)*. IEEE Computer Society, pp. 53–59 (cit. on pp. 20, 132).

San Segundo, Pablo, Diego Rodríguez-Losada, and Agustín Jiménez (2011). "An exact bit-parallel algorithm for the maximum clique problem". In: *Computers & OR* 38.2, pp. 571–581. DOI: 10.1016/j.cor.2010.07.019. (Cit. on pp. 30, 33).

San Segundo, Pablo, Diego Rodríguez-Losada, Fernando Matía, and Ramón Galán (2010). "Fast exact feature based data correspondence search with an efficient bit-parallel MCP solver". In: *Appl. Intell.* 32.3, pp. 311–329. DOI: 10.1007/s10489-008-0147-6 (cit. on p. 45).

San Segundo, Pablo and Cristóbal Tapia (2014). "Relaxed approximate coloring in exact maximum clique search". In: *Computers & OR* 44, pp. 185–192. DOI: 10.1016/j.cor.2013.10.018 (cit. on p. 58).

San Segundo, Pablo, Cristóbal Tapia, and Alvaro Lopez (2013). "Watching Subgraphs to Improve Efficiency in Maximum Clique Search". In: *Contemporary Challenges and Solutions in Applied Artificial Intelligence*. Ed. by Moonis Ali, Tibor Bosse, Koen V. Hindriks, Mark Hoogendoorn, Catholijn M. Jonker, and Jan Treur. Heidelberg: Springer International Publishing, pp. 115–122. ISBN: 978-3-319-00651-2. DOI: 10.1007/978-3-319-00651-2_16. (Cit. on p. 57).

Sanchis, Laura A. (1992). "Test Case Construction for the Vertex Cover Problem". In: *Computational Support for Discrete Mathematics, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, March 12-14, 1992*. Ed. by Nathaniel Dean and Gregory E. Shannon. Vol. 15. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 315–326. (Cit. on p. 43).

Sanchis, Laura A. (1995). "Generating Hard and Diverse Test Sets for NP-hard Graph Problems". In: *Discrete Applied Mathematics* 58.1, pp. 35–66. DOI: 10.1016/0166-218X(93)E0140-T. (Cit. on p. 43).

Sanders, Peter (1995). "Better Algorithms for Parallel Backtracking". In: *Parallel Algorithms for Irregularly Structured Problems, Second International Workshop, IRREGULAR '95, Lyon, France, September 4-6, 1995, Proceedings*. Ed. by Afonso Ferreira and José D. P. Rolim. Vol. 980. Lecture Notes in Computer Science. Springer, pp. 333–347. ISBN: 3-540-60321-2. DOI: 10.1007/3-540-60321-2_27 (cit. on p. 72).

Schulte, Christian and Peter J. Stuckey (2008). "Efficient constraint propagation engines". In: *ACM Trans. Program. Lang. Syst.* 31.1, 2:1–2:43. DOI: 10.1145/1452044.1452046 (cit. on pp. 10, 131).

Schulte, Christian and Guido Tack (2009). "Weakly Monotonic Propagators". In: *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*. Ed. by Ian P. Gent. Vol. 5732. Lecture Notes in Computer Science. Springer, pp. 723–730. ISBN: 978-3-642-04243-0. DOI: 10.1007/978-3-642-04244-7_56 (cit. on pp. 10, 131).

Schulte, Christian, Guido Tack, and Mikael Z Lagerkvist (2016). *Modeling and programming with Gecode*. Version 5.0.0. (cit. on p. 81).

Sevegnani, Michele and Muffy Calder (2015). "Bigraphs with sharing". In: *Theor. Comput. Sci.* 577, pp. 43–73. DOI: 10.1016/j.tcs.2015.02.011 (cit. on p. 125).

Sewell, Edward C. (1993). "A. improved algorithm for exact graph coloring". In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Ed. by David S. Johnson and Michael A. Trick. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 359–376. (Cit. on p. 56).

Shahinpour, Shahram and Sergiy Butenko (2013). "Distance-Based Clique Relaxations in Networks: s-Clique and s-Club". In: *Models, Algorithms, and Technologies for Network Analysis: Proceedings of the Second International Conference on Network Analysis*. Ed. by Boris I. Goldengorin, Valery A. Kalyagin, and Panos M. Pardalos. New York, NY: Springer New York, pp. 149–174. ISBN: 978-1-4614-8588-9. DOI: 10.1007/978-1-4614-8588-9_10 (cit. on p. 96).

Shang, Haichuan, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu (2008). "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism". In: *PVLDB* 1.1, pp. 364–375 (cit. on p. 165).

Sharad, Kumar and George Danezis (2013). "De-anonymizing d4d datasets." In: *Workshop on Hot Topics in Privacy Enhancing Technologies, Bloomington, Indiana, USA*. (Cit. on p. 171).

Sharmin, Sadia (2014). “Practical Aspects of the Graph Parameter Boolean-width”. PhD thesis. The University of Bergen (cit. on pp. 18, 44).

Shasha, Dennis E., Jason Tsong-Li Wang, and Rosalba Giugno (2002). “Algorithmics and Applications of Tree and Graph Searching”. In: *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*. Ed. by Lucian Popa, Serge Abiteboul, and Phokion G. Kolaitis. ACM, pp. 39–52. ISBN: 1-58113-507-6. DOI: 10.1145/543613.543620 (cit. on pp. 161, 171).

Smith, Barbara M. (1994). “The Phase Transition and the Mushy Region in Constraint Satisfaction Problems”. In: *ECAI*, pp. 100–104 (cit. on p. 37).

Smith, Barbara M. and Martin E. Dyer (1996). “Locating the Phase Transition in Binary Constraint Satisfaction Problems”. In: *Artif. Intell.* 81.1-2, pp. 155–181. DOI: 10.1016/0004-3702(95)00052-6 (cit. on pp. 37, 151).

Smith, Barbara M. and Stuart A. Grant (1997). “Modelling Exceptionally Hard Constraint Satisfaction Problems”. In: *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*. Ed. by Gert Smolka. Vol. 1330. Lecture Notes in Computer Science. Springer, pp. 182–195. DOI: 10.1007/BFb0017439 (cit. on p. 148).

Solnon, Christine (2010). “AllDifferent-based filtering for subgraph isomorphism”. In: *Artif. Intell.* 174.12-13, pp. 850–864. DOI: 10.1016/j.artint.2010.05.002 (cit. on pp. 125, 133, 134, 147).

Solnon, Christine (2016). *Benchmarks for the Subgraph Isomorphism Problem*. URL: http://liris.cnrs.fr/csolnon/SIP.html (visited on March 31, 2017) (cit. on pp. 133, 212).

Solnon, Christine, Guillaume Damiand, Colin de la Higuera, and Jean-Christophe Janodet (2015). “On the complexity of submap isomorphism and maximum common submap problems”. In: *Pattern Recognition* 48.2, pp. 302–316. DOI: 10.1016/j.patcog.2014.05.019 (cit. on pp. 125, 134).

Soriano, Patrick and Michel Gendreau (1993). “Tabu search algorithms for the maximum clique problem”. In: *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*. Ed. by David S. Johnson and Michael A. Trick. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, pp. 221–244 (cit. on p. 40).

Strash, Darren (2016). “On the Power of Simple Reductions for the Maximum Independent Set Problem”. In: *Computing and Combinatorics - 22nd International Conference, COCOON 2016, Ho Chi Minh City, Vietnam, August 2-4, 2016, Proceedings*. Ed. by Thang N. Dinh and My T. Thai. Vol. 9797. Lecture Notes in Computer Science. Springer, pp. 345–356. ISBN: 978-3-319-42633-4. DOI: 10.1007/978-3-319-42634-1_28 (cit. on pp. 18, 60).

Stroustrup, Bjarne (2012). “Software Development for Infrastructure”. In: *IEEE Computer* 45.1, pp. 47–58. DOI: 10.1109/MC.2011.353 (cit. on p. 21).

Stuckey, Peter J. and Carleton Coffrin (2016). *Modeling Discrete Optimization*. Coursera. (MOOC) (cit. on p. 7).

Suters, W. Henry, Faisal N. Abu-Khzam, Yun Zhang, Christopher T. Symons, Nagiza F. Samatova, and Michael A. Langston (2005). “A New Approach and Faster Exact Methods for the Maximum Common Subgraph Problem”. In: *Computing and Combinatorics, 11th Annual International Conference, COCOON 2005, Kunming, China, August 16-29, 2005, Proceedings*. Ed. by Lusheng Wang. Vol. 3595. Lecture Notes in Computer Science. Springer, pp. 717–727. ISBN: 3-540-28061-8. DOI: 10.1007/11533719_73 (cit. on p. 207).

Sutter, Herb (2005). “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”. In: *Dr. Dobb’s Journal* 30.3 (cit. on pp. 19, 214).

Sutter, Herb and James R. Larus (2005). “Software and the concurrency revolution”. In: *ACM Queue* 3.7, pp. 54–62. DOI: 10.1145/1095408.1095421 (cit. on p. 19).

Tack, Guido (2009). “Constraint Propagation – Models, Techniques, Implementation”. Doctoral Dissertation. Saarland University (cit. on p. 10).

Tarhio, Jorma, Jan Holub, and Emanuele Giaquinta (2016). “Technology Beats Algorithms (in Exact String Matching)”. In: *CoRR* abs/1612.01506 (cit. on p. 21).

Tian, Yuanyuan and Jignesh M. Patel (2008). “TALE: A Tool for Approximate Large Graph Matching”. In: *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*. Ed. by Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen. IEEE Computer Society, pp. 963–972. ISBN: 978-1-4244-1836-7. DOI: 10.1109/ICDE.2008.4497505 (cit. on p. 164).

Tomita, Etsuji (2017). “Efficient Algorithms for Finding Maximum and Maximal Cliques and Their Applications”. In: *WALCOM: Algorithms and Computation: 11th International Conference and Workshops, WALCOM 2017, Hsinchu, Taiwan, March 29–31, 2017, Proceedings*. Ed. by Sheung-Hung Poon, Md. Saidur Rahman, and Hsu-Chun Yen. Cham: Springer International Publishing, pp. 3–15. ISBN: 978-3-319-53925-6. DOI: 10.1007/978-3-319-53925-6_1 (cit. on p. 30).

Tomita, Etsuji and Toshikatsu Kameda (2007). “An Efficient Branch-and-bound Algorithm for Finding a Maximum Clique with Computational Experiments”. In: *J. Global Optimization* 37.1, pp. 95–111. DOI: 10.1007/s10898-006-9039-7. (Cit. on pp. 30, 45, 47, 51, 56).

Tomita, Etsuji and Tomokazu Seki (2003). “An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique”. In: *Discrete Mathematics and Theoretical Computer Science, 4th International Conference, DMTCS 2003, Dijon, France, July 7-12, 2003. Proceedings*. Ed. by Cristian Calude, Michael J. Dinneen, and Vincent Vajnovszki. Vol. 2731. Lecture

Notes in Computer Science. Springer, pp. 278–289. ISBN: 3-540-40505-4. DOI: 10.1007/3-540-45066-1_22. (Cit. on p. 30).

Tomita, Etsuji, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki (2010). "A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique". In: *WALCOM: Algorithms and Computation, 4th International Workshop, WALCOM 2010, Dhaka, Bangladesh, February 10-12, 2010. Proceedings*. Ed. by Md. Saidur Rahman and Satoshi Fujita. Vol. 5942. Lecture Notes in Computer Science. Springer, pp. 191–203. ISBN: 978-3-642-11439-7. DOI: 10.1007/978-3-642-11440-3_18. (Cit. on pp. 30, 56).

Tomita, Etsuji, Kohei Yoshida, Takuro Hatta, Atsuki Nagao, Hiro Ito, and Mitsuo Wakatsuki (2016). "A Much Faster Branch-and-Bound Algorithm for Finding a Maximum Clique". In: *Frontiers in Algorithmics, 10th International Workshop, FAW 2016, Qingdao, China, June 30- July 2, 2016, Proceedings*. Ed. by Daming Zhu and Sergey Bereg. Vol. 9711. Lecture Notes in Computer Science. Springer, pp. 215–226. ISBN: 978-3-319-39816-7. DOI: 10.1007/978-3-319-39817-4_21. (Cit. on pp. 30, 43, 56, 58, 90).

Trienekens, Harry W. J. M. (1990). "Parallel Branch and Bound Algorithms". PhD thesis. Erasmus University Rotterdam (cit. on pp. 24, 67, 73).

Ullmann, Julian R. (1976). "An Algorithm for Subgraph Isomorphism". In: *J. ACM* 23.1, pp. 31–42. DOI: 10.1145/321921.321925 (cit. on pp. 125, 132, 163, 165).

Ullmann, Julian R. (2010). "Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism". In: *ACM Journal of Experimental Algorithmics* 15. DOI: 10.1145/1671970.1921702 (cit. on p. 132).

Vasilyeva, Elena, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner (2016). "Answering "Why Empty?" and "Why So Many?" queries in graph databases". In: *J. Comput. Syst. Sci.* 82.1, pp. 3–22. DOI: 10.1016/j.jcss.2015.06.007 (cit. on p. 171).

Vehlow, Corinna, Henning Stehr, Matthias Winkelmann, José M. Duarte, Lars Petzold, Juliane Dinse, and Michael Lappe (2011). "CMView: Interactive contact map visualization and analysis". In: *Bioinformatics* 27.11, p. 1573. DOI: 10.1093/bioinformatics/btr163 (cit. on p. 165).

Verfaillie, Gérard, Michel Lemaître, and Thomas Schiex (1996). "Russian Doll Search for Solving Constraint Optimization Problems". In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1.* Ed. by William J. Clancey and Daniel S. Weld. AAAI Press / The MIT Press, pp. 181–187. ISBN: 0-262-51091-X (cit. on p. 59).

Vismara, Philippe and Benoît Valery (2008). "Finding Maximum Common Connected Subgraphs Using Clique Detection or Constraint Satisfaction Algorithms". In: *Modelling,*

Computation and Optimization in Information Systems and Management Sciences, Second International Conference, MCO 2008, Metz, France - Luxembourg, September 8-10, 2008. Proceedings. Ed. by Le Thi Hoai An, Pascal Bouvry, and Pham Dinh Tao. Vol. 14. Communications in Computer and Information Science. Springer, pp. 358–368. ISBN: 978-3-540-87476-8. DOI: 10.1007/978-3-540-87477-5_39 (cit. on pp. 172, 174, 176, 177, 179, 180, 184).

Walsh, Toby (1997). "Depth-bounded Discrepancy Search". In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*. Morgan Kaufmann, pp. 1388–1395 (cit. on p. 76).

Walsh, Toby (1998). "The Constrainedness Knife-Edge". In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*. Ed. by Jack Mostow and Chuck Rich. AAAI Press / The MIT Press, pp. 406–411. ISBN: 0-262-51098-7 (cit. on pp. 51, 153).

Walsh, Toby (2015). *How can you stop killer robots*. Talk at TEDxBerlin (cit. on p. 45).

Walshaw, Chris (2016). *The Graph Partitioning Archive*. URL: http://chriswalshaw.co.uk/partition/ (visited on March 31, 2017) (cit. on p. 104).

Wang, Guoren, Bin Wang, Xiaochun Yang, and Ge Yu (2012). "Efficiently Indexing Large Sparse Graphs for Similarity Search". In: *IEEE Trans. Knowl. Data Eng.* 24.3, pp. 440–451. DOI: 10.1109/TKDE.2010.28 (cit. on p. 163).

Wang, Jing, Nikos Ntarmos, and Peter Triantafillou (2016). "Indexing Query Graphs to Speedup Graph Query Processing". In: *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*. Ed. by Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis. OpenProceedings.org, pp. 41–52. ISBN: 978-3-89318-070-7. DOI: 10.5441/002/edbt.2016.07 (cit. on p. 164).

Williams, Anthony (2012). *C++ concurrency in action: practical multithreading*. Shelter Island, NY: Manning Publ. (cit. on pp. 21, 22).

Wood, David R. (1997). "An algorithm for finding a maximum clique in a graph". In: *Oper. Res. Lett.* 21.5, pp. 211–217. DOI: 10.1016/S0167-6377(97)00054-0 (cit. on pp. 30, 57).

Wotzlaw, Andreas (2014). "On Solving the Maximum k-club Problem". In: *CoRR* abs/1403.5111 (cit. on pp. 96, 103, 104).

Wu, Qinghua and Jin-Kao Hao (2015). "A review on algorithms for maximum clique problems". In: *European Journal of Operational Research* 242.3, pp. 693–709. DOI: 10.1016/j.ejor.2014.09.064 (cit. on pp. 2, 45, 60).

Xiang, Jingen, Cong Guo, and Ashraf Aboulnaga (2013). "Scalable maximum clique computation using MapReduce". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. Ed. by Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou. IEEE Computer Society, pp. 74–85. ISBN: 978-1-4673-4909-3. DOI: 10.1109/ICDE.2013.6544815. (Cit. on pp. 66, 68, 70, 72, 73, 90).

Xu, Ke (2014). *BHOSLIB: Benchmarks with Hidden Optimum Solutions for Graph Problems*. URL: http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm (visited on March 31, 2017) (cit. on p. 44).

Xu, Ke, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre (2005). "A Simple Model to Generate Hard Satisfiable Instances". In: *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*. Ed. by Leslie Pack Kaelbling and Alessandro Saffiotti. Professional Book Center, pp. 337–342. ISBN: 0938075934 (cit. on p. 44).

Xu, Ke and Wei Li (2006). "Many hard examples in exact phase transitions". In: *Theor. Comput. Sci.* 355.3, pp. 291–302. DOI: 10.1016/j.tcs.2006.01.001 (cit. on p. 44).

Yan, B. and S. Gregory (2009). "Detecting communities in networks by merging cliques". In: *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*. Vol. 1, pp. 832–836. DOI: 10.1109/ICICISYS.2009.5358036 (cit. on p. 45).

Yan, Xifeng, Philip S. Yu, and Jiawei Han (2004). "Graph Indexing: A Frequent Structure-based Approach". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. Ed. by Gerhard Weikum, Arnd Christian König, and Stefan Deßloch. ACM, pp. 335–346. ISBN: 1-58113-859-8. DOI: 10.1145/1007568.1007607 (cit. on p. 161).

Yan, Xifeng, Philip S. Yu, and Jiawei Han (2005). "Graph indexing based on discriminative frequent structure analysis". In: *ACM Trans. Database Syst.* 30.4, pp. 960–993. DOI: 10.1145/1114244.1114248 (cit. on p. 162).

Yuan, Dayu and Prasenjit Mitra (2013). "Lindex: a lattice-based index for graph databases". In: *VLDB J.* 22.2, pp. 229–252. DOI: 10.1007/s00778-012-0284-8 (cit. on p. 163).

Yuan, Dayu, Prasenjit Mitra, and C. Lee Giles (2013). "Mining and Indexing Graphs for Supergraph Search". In: *PVLDB* 6.10, pp. 829–840 (cit. on p. 164).

Yuan, Dayu, Prasenjit Mitra, Huiwen Yu, and C. Lee Giles (2015). "Updating Graph Indices with a One-Pass Algorithm". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, pp. 1903–1916. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2746482 (cit. on p. 164).

Zampelli, Stéphane, Yves Deville, and Pierre Dupont (2005). "Approximate Constrained Subgraph Matching". In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*. Ed. by Peter van Beek. Vol. 3709. Lecture Notes in Computer Science. Springer, pp. 832–836. ISBN: 3-540-29238-1. DOI: 10.1007/11564751_74 (cit. on pp. 187, 208).

Zampelli, Stéphane, Yves Deville, and Christine Solnon (2010). "Solving subgraph isomorphism problems with constraint programming". In: *Constraints* 15.3, pp. 327–353. DOI: 10.1007/s10601-009-9074-3 (cit. on pp. 125, 128, 130, 133, 145, 194).

Zhang, Shijie, Meng Hu, and Jiong Yang (2007). "TreePi: A Novel Graph Indexing Method". In: *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*. Ed. by Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis. IEEE Computer Society, pp. 966–975. ISBN: 1-4244-0802-4. DOI: 10.1109/ICDE.2007.368955 (cit. on p. 163).

Zhang, Shijie, Shirong Li, and Jiong Yang (2009). "GADDI: distance index based subgraph matching in biological networks". In: *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*. Ed. by Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold. Vol. 360. ACM International Conference Proceeding Series. ACM, pp. 192–203. ISBN: 978-1-60558-422-5. DOI: 10.1145/1516360.1516384 (cit. on p. 163).

Zhao, Peixiang, Jeffrey Xu Yu, and Philip S. Yu (2007). "Graph Indexing: Tree + Delta >= Graph". In: *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. Ed. by Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold. ACM, pp. 938–949. ISBN: 978-1-59593-649-3 (cit. on p. 162).

Zheng, Weiguo, Xiang Lian, Lei Zou, Liang Hong, and Dongyan Zhao (2016). "Online Subgraph Skyline Analysis over Knowledge Graphs". In: *IEEE Trans. Knowl. Data Eng.* 28.7, pp. 1805–1819. DOI: 10.1109/TKDE.2016.2530063 (cit. on p. 164).

Zuckerman, David (2006). "Linear degree extractors and the inapproximability of max clique and chromatic number". In: *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*. Ed. by Jon M. Kleinberg. ACM, pp. 681–690. ISBN: 1-59593-134-1. DOI: 10.1145/1132516.1132612 (cit. on p. 18).