UNIVERSITY
of
GLASGOW

Computing Science
Ph.D. Thesis

Subsequences and Supersequences of Strings

Campbell Bryce Fraser

Submitted for the degree of

Doctor of Philosophy

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

Stringology — the study of strings — is a branch of algorithmics which been the subject of mounting interest in recent years. Very recently, two books [M. Crochemore and W. Rytter, Text Algorithms, Oxford University Press, 1995] and [G. Stephen, String Searching Algorithms, World Scientific, 1994] have been published on the subject and at least two others are known to be in preparation. Problems on strings arise in information retrieval, version control, automatic spelling correction, and many other domains. However the greatest motivation for recent work in stringology has come from the field of molecular biology. String problems occur, for example, in genetic sequence construction, genetic sequence comparison, and phylogenetic tree construction. In this thesis we study a variety of string problems from a theoretical perspective. In particular, we focus on problems involving subsequences and supersequences of strings.

Many algorithms have been proposed, over the years, for the Longest Common Subsequence (LCS) problem on two strings. However, very little attention has been paid to to the problem for more than two strings, perhaps because the problem is **NP**-complete if the number of strings is not bounded [Maier, *Journal of the ACM*, 25:322-336, 1978]. In this thesis, we describe two algorithms for the LCS problem over three strings and show that both can be extended to solve the problem for any fixed number of strings. The first algorithm can be viewed as a "lazy" version of a basic dynamic programming strategy, and has time and space complexity $O(kn(n-l)^{k-1})$ where k is the number of strings, n is the length of each string, and l is the length of the LCS. The second algorithm employs a more sophisticated dynamic programming strategy called the "threshold" approach, and achieves time and space complexity $O(kl(n-l)^{k-1} + kzn)$ where z is the size of the alphabet.

Empirical evidence is presented to show that both algorithms improve significantly on the long-known basic dynamic programming approach, and on a previous algorithm proposed by Hsu and Du [*BIT*, 24:45-59, 1984], particularly when the LCS is relatively long.

Related to the LCS problem is the Shortest Common Supersequence (SCS) problem, also known to be **NP**-complete when the number of strings is not bounded [Maier, *Journal of the ACM*, 25:322-336, 1978]. When there are two strings, the LCS and SCS problems are dual so all the algorithms designed for the LCS problem on two strings apply also to the SCS problem on two strings. However when there are more than two strings the problems are no longer dual and, as for the LCS problem, very little attention has been paid to the general SCS problem. We describe two algorithms for the SCS problem over three strings and show how both can be extended to solve the problem for any fixed number of strings. Both algorithms are

derived from application of a threshold technique similar to that used successfully for the LCS problem. The first operates on matching symbols of the strings and achieves $O(2^t z n^{k-1} t)$ time and $O(n^{k-1} t)$ space complexity, where s is the length of the SCS and $t = kn - s$. The second algorithm operates on supersequences of increasing prefixes of the strings and has time and space complexity $O(kn(s-n)^{k-1})$.

Empirical evidence is presented to show that the first algorithm is better than basic dynamic programming for random strings when the alphabet is very large (and the SCS is consequently very long). When the SCS is short, the second algorithm performs significantly better than the basic dynamic programming algorithm.

Since the general LCS and SCS problems are **NP**-hard, exact solutions to large instances are not likely to be attainable. However the question arises as to how large the problem instances can become and remain solvable using realistic computational resources. For both the LCS and SCS problems, an attempt is made to characterise the range of problem instances solvable using realistic computational resources. To achieve this, a range of efficient algorithms employing two fundamentally different approaches have been implemented and experiments performed to discover how large the problem instances may become before the time or the memory space required to solve them becomes impractical. The first approach employs the familiar dynamic programming strategy. Throughout the literature, there appears to be an implicit assumption that algorithms based on dynamic programming are the best way to solve these two problems. To test this, the second approach utilises a natural branch-and-bound strategy and the extent to which this broadens the range of solvable problem instances is examined.

The expected LCS length over a given set of parameters (alphabet size, number of strings, and string lengths) is the average LCS length over all instances of the problem with that given set of parameters. The expected SCS length is defined similarly. A natural by-product of the experiments is a body of empirical data, more comprehensive than any available thus far, which allows estimation of the expected LCS and SCS for particular sets of problem parameters. This information is compared with theoretical results appearing in the literature and may be useful in guiding future theoretical work on estimating expected LCS and SCS lengths.

Given the **NP**-hard status of the LCS and SCS problems, it is natural to attempt to find polynomial-time approximation algorithms for them with good performance guarantees. The worst-case behaviour of a range of previously proposed, and new, approximation algorithms for the LCS and SCS problems is analysed.

It is shown that none of the approximation algorithms, under analysis, for the LCS problem has a performance guarantee better than $O(z)$ and $O(n)$. None of the approximation algorithms, under analysis, for the SCS problem has a performance guarantee better than $O(k)$ over an unbounded alphabet or $O(\log k)$ for a binary

alphabet.

Despite this worst-case behaviour, empirical evidence is presented to show that, in practice, some of the SCS approximation algorithms can be expected to give very good approximations.

The worst-case performance of the stronger SCS approximation algorithms, with respect to the size of the input, is also studied. It is shown that none has a performance guarantee better than $O((\frac{I}{\log_2 I})^{(\log_2 3-1)/2})$ where I is the size of the input.

As discussed above, we may have to resort to finding an approximate solution to an instance of the LCS (or SCS) problem because it is too big to solve using an exact algorithm. Having found an approximation γ to the LCS of a set P of strings, we could attemp to add symbols to γ, so that it remains a common subsequence of P, and continue to do so while it is possible. The question then arises as to how long we can typically expect γ to get before no more symbols can be added, for it to remain a common subsequence of P. Similarly, how short can we typically expect δ, an approximation to the SCS of P, to get while we remove symbols from δ so that it remains a common supersequence of P?

The Shortest Maximal Common Subsequence (SMCS) problem is shown to be **NP**-complete and a very strong negative result on the approximability of that problem, over an unbounded alphabet, is also established. The complexity of the Longest Minimal Common Supersequence (LMCS) problem is left open [1]. Algorithms are presented for the SMCS and LCMS problems over two strings (in which case, unlike the LCS and SCS problems, the problems are not dual), which can be extended to work for any fixed number of strings.

A consistent subsequence of a positive set P and a negative set N of strings is a string that is a subsequence of every string in P but of no string in N. A consistent supersequence is defined similarly. This thesis addresses, from the complexity point of view, existence and optimisation problems concerning consistent subsequences and supersequences.

Some consistent sequence optimisation problems are generalisations of previously studied sequence inclusion and sequence non-inclusion optimisation problems and thus inherit **NP**-hardness. Jiang and Li [*Theoretical Computer Science*, 119:363-371, 1992] showed that: (i) finding a consistent supersequence is **NP**-complete when $|P| \geq 2$ is bounded and N is unbounded, and (ii) finding a consistent supersequence is solvable in polynomial time when $|P|$ is unbounded and $|N| = 1$.

All existing results relating to consistent sequence problems are summarised in this thesis. Further, the following are shown to be **NP**-complete: (i) finding a con-

[1] Middendorf [Technical Report 300, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, 1994] has proved that both problems are **MAX SNP**-hard even over a binary alphabet.

sistent subsequence when $|P| \geq 2$ is bounded and N is unbounded, (ii) finding a consistent subsequence when $|P|$ is unbounded and $|N| = 1$, and (iii) finding a consistent supersequence when $|P|$ is unbounded and $|N| \geq 2$ is bounded. Polynomial time algorithms are given to find, when $|P|$ and $|N|$ are bounded: (i) a shortest or longest consistent subsequence, and (ii) a shortest consistent supersequence.

Analogous to the consistent sequence problems are consistent substring and superstring problems. A consistent substring of a positive set P and a negative set N of strings is a string that is a substring of every string in P but of no string in N. A consistent superstring is defined similarly. Polynomial-time algorithms are described for several consistent substring and superstring problems.

Declaration

This dissertation is submitted in accordance with the regulations for the degree of Doctor of Philosophy in the University of Glasgow. No part of it has been previously submitted by the author for a degree at any university and all results contained within are claimed as original, except where indicated in the preface.

Acknowledgements

This work could not have been completed without the first-class supervision of Rob Irving. He provided me with many pointers for research and his advice and guidance have been invaluable in broadening my understanding of Algorithmics. His constructive criticism of everything I have written has greatly improved the quality of the thesis.

Mike Paterson provided many worthwhile comments on the first complete draft of the thesis. I would like to thank Ron Poet and Arthur Allison (my supervisory committee), and Keith Van Risjbergen for their time and their advice.

Without my parents I would never have received the education to make this possible and thank-you to Elaine for tolerating my poverty during the last year.

I also wish to thank the Engineering and Physical Sciences Research Council[2] for financial support, in the form of a Research Studentship, during the period October 1991 to September 1994.

[2]Previously the Science and Engineering Research Council

Contents

1 Introduction and Background **1**

1.1 Introduction 1
1.2 Definitions and notations 1
1.3 The Longest Common Subsequence Problem 3
1.3.1 Complexity results 3
1.3.2 Exact algorithms for $k = 2$ strings 4
1.3.3 Exact algorithms for $k > 2$ strings 11
1.3.4 Approximation algorithms for the LCS 13
1.3.5 Expected LCS lengths 14
1.4 The Shortest Common Supersequence Problem 15
1.4.1 Complexity results 15
1.4.2 Exact algorithms for $k > 2$ strings 16
1.4.3 Approximation algorithms for the SCS 16
1.5 Problems related to LCS and SCS 17
1.5.1 Maximal subsequences and minimal supersequences 17
1.5.2 Negative subsequences and supersequences 17
1.5.3 Consistent subsequences and supersequences 18
1.5.4 Miscellaneous string comparison problems 20
1.6 Substring and Superstring Problems 20
1.6.1 The Longest Common Substring Problem 20
1.6.2 The Shortest Common Superstring Problem 21
1.6.3 Negative and consistent substrings and superstrings 22
1.7 The complexity of approximation 22

2 Exact Algorithms for the LCS problem **24**

2.1 Introduction 24
2.2 The "Lazy" Approach to Dynamic Programming 24
2.2.1 Recovering an LCS 28
2.2.2 Analysis 30
2.2.3 Extension to > 3 Strings 31
2.3 A Threshold Based Algorithm 31
2.3.1 Recovering an LCS 33
2.3.2 Analysis 33
2.3.3 Extension to > 3 Strings 36

2.4 Empirical results and conclusions 36

3 Exact Algorithms for the SCS problem 39
3.1 Introduction 39
3.2 An algorithm operating on matches 39
3.2.1 Recovering an SCS 42
3.2.2 Analysis 42
3.2.3 Extension to >3 Strings 44
3.3 An algorithm operating on supersequences 45
3.3.1 Recovering an SCS 46
3.3.2 Analysis 47
3.3.3 Extension to >3 Strings 48
3.4 Empirical results and conclusions 49

4 Exact solutions to large instances of the LCS and SCS problems 51
4.1 Introduction 51
4.2 Dynamic programming to solve LCS 52
4.2.1 Lazy dynamic programming using an Array 52
4.2.2 Lazy dynamic programming using a Trie 53
4.2.3 Analysis of Lazy DP solutions to the LCS 53
4.3 Branch-and-bound to solve LCS 54
4.3.1 Finding a good initial bound 54
4.3.2 Improving on the basic strategy 55
4.3.3 Analysis of Branch-and-Bound solutions to the LCS 57
4.4 Dynamic programming to solve SCS 57
4.4.1 Analysis of Lazy DP solutions to the SCS 58
4.5 Branch and bound to solve SCS 59
4.5.1 Finding a good initial bound 59
4.5.2 Improving on the basic strategy 59
4.5.3 Analysis of Branch-and-Bound solutions to the SCS 62
4.6 Solving LCS and SCS for two strings 63
4.7 The experiments 63
4.7.1 Behaviour of the branch and bound algorithms 65
4.7.2 Results for the LCS problem when $k > 2$ 65
4.7.3 Results for the SCS problem when $k > 2$ 69
4.7.4 Results for the LCS problem when $k = 2$ 71
4.8 The expected lengths of the LCS and the SCS of random strings . . . 72
4.8.1 The expected LCS length when $k > 2$ 72
4.8.2 The expected SCS length when $k > 2$ 72
4.8.3 The expected LCS length when $k = 2$ 72
4.9 Conclusions and future work 75

5 Approximation Algorithms for the LCS and SCS problems 78
5.1 Introduction 78
5.2 The Algorithm Long-Run 80
5.2.1 Worst-case behaviour of Long-Run 80
5.2.2 Bad examples for Long-Run 80
5.3 The Algorithm Best-Next 81
5.3.1 Worst-case behaviour of Best-Next 81
5.3.2 Bad examples for Best-Next 81
5.4 The Tournament Algorithm 82
5.4.1 Worst-case behaviour of the Tournament Algorithm 82
5.4.2 Bad examples for the Tournament Algorithm 85
5.4.3 Bad examples for the Tournament Algorithm on an alphabet of fixed size 87
5.5 The Majority-Merge Algorithm 87
5.5.1 Worst-case behaviour of the Majority-Merge Algorithm 88
5.5.2 Bad examples for the Majority-Merge Algorithm 88
5.5.3 Worst-case behaviour of the Majority-Merge Algorithm on an alphabet of fixed size 89
5.5.4 Bad examples for the Majority-Merge Algorithm on an alphabet of fixed size 90
5.5.5 "Algorithm M4" of Foulser, Li and Yang 92
5.6 The Algorithm Greedy1 92
5.6.1 Worst-case behaviour of Greedy1 92
5.6.2 Bad examples for Greedy1 on an alphabet of fixed size 94
5.7 The Centre of the Star Algorithm 97
5.7.1 Worst-case behaviour of the Centre of the Star Algorithm . . . 97
5.7.2 Bad examples for the Centre of the Star Algorithm on an alphabet of fixed size 98
5.8 The Minimum Spanning Tree Algorithm 99
5.8.1 Worst-case behaviour of the Minimum Spanning Tree Algorithm 99
5.8.2 Bad examples for the Minimum Spanning Tree Algorithm . . . 99
5.8.3 Bad examples for the Minimum Spanning Tree Algorithm on an alphabet of fixed size 100
5.9 The Algorithm Greedy2 101
5.9.1 Worst case behaviour of Greedy2 101
5.9.2 Bad examples for Greedy2 on an alphabet of fixed size 103
5.10 Summary of the approximation algorithms for the LCS and SCS problems 104
5.11 Empirical comparison of the approximation algorithms for the SCS problem 104
5.12 Bad examples with respect to the input size for the SCS approximation algorithms 107
5.13 Conclusions and open problems 110

6 Maximal subsequences and minimal supersequences 112
6.1 Introduction . . . 112
6.2 The SMCS problem for general k strings . . . 113
6.3 The SMCS and LMCS problems for $k = 2$ strings . . . 115
6.3.1 The SMCS Algorithm . . . 116
6.3.2 The LMCS Algorithm . . . 119
6.4 Conclusion and open problem . . . 122

7 Consistent subsequences and supersequences 123
7.1 Introduction . . . 123
7.2 Consistent Subsequence Problems . . . 125
7.2.1 Consistent subsequence when $|P|$ is bounded and $|N|$ is unbounded ($|P| \geq 2$) . . . 125
7.2.2 Consistent subsequence when $|P|$ is unbounded and $|N|$ is bounded ($|N| \geq 1$) . . . 126
7.2.3 Longest consistent subsequence when $|P| = 1$ and $|N|$ is unbounded . . . 127
7.2.4 Shortest or Longest consistent subsequence when $|P|$ and $|N|$ are bounded . . . 128
7.3 Consistent Supersequence Problems . . . 130
7.3.1 Consistent supersequence when $|P|$ is unbounded and $|N|$ is bounded ($|N| \geq 2$) . . . 130
7.3.2 Shortest consistent supersequence when $|P| = 1$ and $|N|$ is unbounded . . . 132
7.3.3 Shortest consistent supersequence when $|P|$ and $|N|$ are bounded 132
7.3.4 Longest consistent supersequence when $|P|$ is unbounded and $|N| = 1$ . . . 134
7.4 Open Problems . . . 135
7.5 Consistent String Problems . . . 136
7.5.1 Shortest/Longest consistent substring . . . 136
7.5.2 Consistent superstring when $|P|$ is unbounded and $|N| = 1$ . . . 137
7.5.3 Shortest consistent superstring when $|P| = 1$ and $|N|$ is unbounded . . . 139
7.5.4 Longest consistent superstring when $|P|$ is unbounded and $|N| = 1$ . . . 139
7.5.5 Open Problems on consistent strings . . . 139

Bibliography 141

List of Figures

1.1 An example of dynamic programming for the LCS of 2 strings 5

2.1 The lazy algorithm for the length of the LCS of 3 strings 29
2.2 Evaluation of $t(\Delta, 3p - x)$ for $\Delta = \langle i, j, k \rangle$ 29
2.3 Initialisation for the p^{th} iteration of the lazy algorithm 29
2.4 Recovering an LCS from the t table 30
2.5 The threshold algorithm for the length of the LCS of 3 strings 34
2.6 Recovering an LCS from the threshold table 34

3.1 Forward pass of MML_Thresh 43
3.2 Recovering an SCS in MML_Thresh 43
3.3 Forward pass of SCS_Thresh 47
3.4 Recovering an SCS in SCS_Thresh 48

4.1 The Lazy Dynamic Programming Algorithm for the LCS problem. . . 53
4.2 The basic Branch-and-Bound Algorithm for the LCS problem. 54
4.3 The procedure *Choose2.lcs*. 55
4.4 The function *Extendible2.lcs*. 56
4.5 The function *Extendible3.lcs*. 57
4.6 The Lazy Dynamic Programming Algorithm for the SCS problem. . . 58
4.7 The basic Branch-and-Bound Algorithm for the SCS problem. 60
4.8 The function *Choose2.scs*. 61
4.9 The function *Extends2.scs*. 62
4.10 The function *Extends3.scs*. 63
4.11 Expected lengths of the LCS of random strings. 73
4.12 Expected lengths of the SCS of random strings. 74
4.13 |Expected LCS|/n for $k = 2$. 75
4.14 Random instances of LCS solvable in 1 hour. 77
4.15 Random instances of SCS solvable in 1 hour. 77

List of Tables

1.1 Summary of LCS algorithms for $k = 2$ strings 12
1.2 Summary of LCS algorithms for $k \geq 2$ strings 13

2.1 CPU times in seconds when LCS is 90% of string length 37
2.2 CPU times in seconds when LCS is 50% of string length 37
2.3 CPU times in seconds when LCS is 10% of string length 37

3.1 CPU times in seconds for SCS algorithms on random strings 50
3.2 CPU times in seconds for SCS algorithms when $s \leq 11n/10$ 50

4.1 CPU times in second for LCS-DP on random strings. 66
4.2 CPU times in seconds for *LCS-BB-3* on random strings. 68
4.3 Instances of LCS with similar strings solvable by *LCS-BB-2* in 1 hour. 68
4.4 CPU times in seconds for *SCS-DP* on random strings. 70
4.5 CPU times in seconds for *SCS-BB-3* on random strings. 70
4.6 CPU times in seconds for *LCS2-THRESH* on random strings. 72

5.1 Worst-case behaviour of approximation algorithms for the LCS 104
5.2 Worst-case behaviour of approximation algorithms for the SCS 105
5.3 SCS approximations for random strings of length 100 106
5.4 SCS approximations for subsequences of a string of length 100 106

7.1 Consistent Sequence Existence Problems 124
7.2 Consistent Sequence Optimisation Problems 124
7.3 Consistent String Existence Problems 136
7.4 Consistent String Optimisation Problems 136

Glossary

Alphabet – A finite ordered list of symbols.

Consistent – A string is *consistent* for two sets P and N of strings if it is a subsequence (or supersequence/substring/superstring) of every string in P but of no string in N.

LCNS – Longest Common Non-supersequence.

LCS – Longest Common Subsequence.

LCSt – Longest Common Substring.

LMCS – Longest Minimal Common Supersequence.

Maximal – A subsequence (or substring) γ of a string α is *maximal* if no proper supersequence of γ is a subsequence (or substring) of α.

Minimal – A supersequence (or superstring) γ of a string α is *minimal* if no proper subsequence of γ is a supersequence (or superstring) of α.

Prefix – A *prefix* of a string α is any string obtainable by deleting zero or more symbols from the end of α.

Pseudo-code – A Pascal-like algorithm description language. The language adopts many of the reserved words of Pascal but, for brevity, uses indentation to indicate the scope of a compound statement, and omits statement separators.

SCNS – Shortest Common Non-subsequence.

SCS – Shortest Common Supersequence.

SCSt – Shortest Common Superstring.

SMCS – Shortest Maximal Common Subsequence.

String – A finite sequence of symbols of some alphabet.

Subsequence – A *subsequence* of a string α is any string obtainable by removing zero or more symbols from α.

Substring – A *substring* of a string α is any string obtainable by removing zero or more symbols from the start and zero or more symbols from the end of α.

Suffix – A *suffix* of a string α is any string obtainable by deleting zero or more symbols from the start of α.

Supersequence – A *supersequence* of a string α is any string obtainable by inserting zero or more symbols anywhere in α.

Superstring – A *superstring* of a string α is any string obtainable by prepending zero or more and appending zero or more symbols to α.

k – The number of strings.

l – The length of the LCS.

$lp(\alpha, \gamma)$ – The length of the longest prefix of α that is a subsequence of γ.

n – The length of the strings.

$next_\alpha(i, a)$ – The *next occurrence table* for a string α of length n is defined for $0 \leq i \leq n$ and all symbols $a \in \Sigma$ as follows;

$$next_\alpha(i, a) = \begin{cases} \min\{j : \alpha[j] = a, j > i\} & \text{if such a } j \text{ exists.} \\ n + 1 & \text{otherwise.} \end{cases}$$

s – The length of the SCS.

$sp(\alpha, \gamma)$ – The length of the shortest prefix of α that is a supersequence of γ.

z – The size of the alphabet.

Σ – The alphabet.

Preface

Statement of collaboration

The following describes the roles of student (Fraser) and supervisor (Irving) in work undertaken jointly.

Section 1.3.2

The evaluation strategy of the threshold algorithm to find the Longest Common Subsequence (LCS) of two strings was designed by Irving. The application of the divide and conquer technique to improve the space complexity of the algorithm was carried out by Fraser.

Sections 2.2, 2.3, and 2.4

The Lazy Algorithm for the LCS problem was developed simultaneously by Fraser and Irving. The Threshold Algorithm was developed by Irving. Sections 2.2 and 2.3 are adapted in notation and structure from text written by Irving. For the computational experiments, the Lazy and Threshold algorithms were implemented by Irving.

Sections 5.4, 5.6, and 5.9

The strategies to prove the upper bounds for the worst-case behaviour of the Tournament and Greedy approximation algorithms for the Shortest Common Supersequence problem were proposed by Irving. The sets of strings used to prove the lower bounds for the worst-case behaviour of the Tournament algorithm (on unbounded and fixed alphabets respectively) were proposed by Irving. Sections 5.4.1, 5.4.3, and 5.6.1 are adapted in notation and structure from text written by Irving. Section 5.9.1 was written jointly by Irving and Fraser, based on a strategy proposed by Irving.

Sections 6.3

The algorithms to find the Shortest Maximal Common Subsequence and the Longest Minimal Common Supersequence of two strings were developed by Irving. Sections 6.3.1 and 6.3.2 are adapted in notation and structure from text written by Irving.

Publications

R.W. Irving and C.B. Fraser. Two algorithms for the longest common subsequence of three (or more) strings. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching, 1992 [32].*
(This paper is based on Sections 2.2, 2.3, and 2.4.)

R.W. Irving and C.B. Fraser. On the worst-case behaviour of some approximation algorithms for the shortest common supersequence problem. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching, 1993 [33].*
(This paper is based on Sections 5.4 and 5.6.)

R.W. Irving and C.B. Fraser. Maximal common subsequences and minimal common supersequences. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching, 1994 [34].*
(This paper is based on Sections 6.2 and 6.3.)

Work submitted for publication

C.B. Fraser and R.W. Irving. Approximation algorithms for the shortest common supersequence. To appear in the *Nordic Journal of Computing*, 1995. (This paper is based on Sections 5.4, 5.5, 5.6, and 5.9.)

C.B. Fraser, R.W. Irving, and M. Middendorf. Maximal common subsequences and minimal common supersequences. Submitted to *Information and Computation*
(This paper is based on Sections 6.2 and 6.3 and related work by Middendorf.)

C.B. Fraser. Consistent subsequences and supersequences. Submitted to *Theoretical Computer Science.* Publication recommended subject to minor corrections.
(This paper is based on Sections 7.2 and 7.3.)

Overlaps with recently published work by other authors

Immediately after submission of the manuscript to *Theoretical Computer Science*, a copy of a just-published technical report of Middendorf [49] was obtained. The report contains some overlap with the work in Chapter 7. Specifically, using different problem transformations, Middendorf proves theorems equivalent to Theorems 7.2.1 and 7.3.1. Again using a different transformation, he proves a theorem similar to Theorem 7.2.2. His result is weaker in that it only applies if $|N| \geq 2$ (as opposed to $|N| \geq 1$) but stronger in that in applies even when the alphabet has size two (as opposed to unbounded size).

Shortly prior to submission of the thesis, a small overlap with a paper by Chin and Poon [12] was discovered. The paper contains a similar analysis of the worst-case behaviour of *Long-Run*, an approximation algorithm for the Longest Common Subsequence problem, to that in Chapter 5.

Chapter 1

Introduction and Background

1.1 Introduction

Stringology — the study of strings — is a branch of algorithmics which has been the subject of mounting interest in recent years. Very recently two books [14, 60] have been published on the subject and at least two others are known to be in preparation. Problems on strings arise in information retrieval, version control, automatic spelling correction, and many other domains. However the greatest motivation for recent work in stringology has come from the field of molecular biology. String problems occur, for example, in genetic sequence construction, genetic sequence comparison, and phylogenetic tree construction. In this thesis we study a variety of string problems from a theoretical perspective. In particular, we focus on problems involving subsequences and supersequences of strings.

1.2 Definitions and notations

An *alphabet* Σ is a finite ordered list of symbols. A *string* over Σ is a finite sequence of symbols from Σ. The i^{th} symbol of a string α is identified by $\alpha[i]$. The symbols from position i to position j $(i \leq j)$ are identified by $\alpha[i \ldots j]$. A *match* between k strings $\alpha_1, \alpha_2, \ldots, \alpha_k$ is an ordered tuple $(i_1, i_2, \ldots, i_k)$ such that $\alpha_1[i_1] = \alpha_2[i_2] = \cdots = \alpha_k[i_k]$.

Given a string α over an alphabet Σ, a *substring* of α is any string that can be obtained by deleting zero or more symbols from the start and deleting zero or more symbols from the end of α. A *subsequence* of α is any string that can be obtained by removing zero or more symbols from anywhere in α. A *supersequence* of α is any string that can be obtained by inserting zero or more symbols anywhere in α. The decision version of the *Longest Common Subsequence* (LCS) problem is to determine, for a finite set P of strings over an alphabet Σ, and an integer $t \in N$, whether there exists a string of length $\geq t$ over Σ which is a subsequence of every string in P.

The decision version of the *Shortest Common Supersequence* (SCS) problem is to determine, for a finite set P of strings over an alphabet Σ, and an integer $t \in N$, whether there exists a string of length $\leq t$ over Σ which is a supersequence of every string in P. In this thesis we will study the problems re-formulated in the natural way as optimisation problems i.e. given a finite set P of strings, find the length of an LCS (or an SCS) of P.

In general, neither the LCS nor the SCS of a set of strings is unique. In the case of two strings, the LCS and SCS problems are dual. A Shortest Common Supersequence of two strings is formed from a Longest Common Subsequence by inserting into it in appropriate places the unused symbols from each string. In general, an SCS formed from a particular LCS is not unique for that LCS. The following example serves to illustrate these points:

For the strings

$$\begin{aligned} \alpha &= abcd, \\ \beta &= dacb, \end{aligned}$$

the LCS's with their corresponding SCS's in parentheses are *ab (dacbcd)* and *ac (dabcbd, dabcdb)*.

If n and m are the lengths of the two strings and l and s are the lengths of the LCS and SCS respectively then it is clear that

$$s = n + m - l.$$

When there are more than two strings, there is no obvious duality between the LCS and SCS. A long LCS does not imply a short SCS and a short LCS does not imply a long SCS. This can be seen from the following example:

For the strings

$$\begin{array}{lll} \alpha_1 = abc & \beta_1 = abc & \gamma_1 = abzc \\ \alpha_2 = abd & \beta_2 = abd & \gamma_2 = bczd \\ \alpha_3 = acd & \beta_3 = abe & \gamma_3 = cdze \\ \alpha_4 = bcd & \beta_4 = abf & \gamma_4 = dezf. \end{array}$$

The unique SCS of the α's is *abcd* but the unique LCS is the empty string. Hence $s = n + 1$ and $l = 0$. An SCS of the β's is *abcdef* and the unique LCS is *ab*. Hence $s = 2n$ and $l = n - 1$. An SCS of the γ's is *abzczdzezf* and the unique LCS is z. Hence $s = 2n + 2$ and $l = 1$. Notice that the single character (z) common to all the strings appears in the SCS four times (once for each of the strings).

A generalisation of the LCS problem for two strings is the Minimum Edit Distance problem, also known as the String to String Correction problem. If we allow the operations *delete a symbol*, *insert a symbol*, and *substitute one symbol for*

another, on a string, and place a cost on each operation, then the Minimum Edit Distance problem is to find the minimum cost set of operations to convert one string into the other. The costs for each operation may not be uniform over the alphabet. When the cost of substitution is set prohibitively high (e.g. infinity) and insertion and deletion costs are uniform across the alphabet then the problem reduces to the LCS problem.

1.3 The Longest Common Subsequence Problem

1.3.1 Complexity results

The Longest Common Subsequence problem was shown to be **NP**-complete, even over a binary alphabet, by Maier [44]. The proof was by a transformation from the well-known Vertex Cover (or equivalently Independent Set) problem.

Jiang and Li [37] showed that there exists $\delta > 0$ such that if the LCS problem over an unbounded alphabet has a polynomial-time approximation algorithm with performance guarantee k^δ for k strings then **P=NP**.

Recently Bonizzoni, Duella, and Mauri [9] showed that the LCS problem is **MAX SNP**-hard, even over a binary alphabet. This means that no polynomial-time approximation scheme exists for the problem even in that special case unless **P=NP**. The problem class **MAX SNP**is discussed in Section 1.7.

Bounds for $k = 2$ strings

Wong and Chandra [70] showed that using a decision tree model of computation where only "equal/unequal" comparisons between symbols are permitted, $O(n^2)$ comparisons are necessary and sufficient to find the Minimum Edit Distance or the LCS of two strings of lengths n over an unbounded alphabet.

Independently Aho, Hirschberg, and Ullman [1] showed the same result for the LCS problem. They showed that the LCS problem over a binary alphabet can be solved with $2n + 1$ comparisons even if comparisons between symbols of the same string are forbidden. However when the alphabet size rises above two, they showed that n^2 comparisons are necessary and sufficient. Despite the positive result for the binary alphabet, no algorithm using only $2n + 1$ comparisons has been presented in the literature.

Hirschberg showed that $\Omega(n \log n)$ is a lower bound on the number of "less than/equal to/greater than" comparisons required to find the LCS of two strings of length n.

1.3.2 Exact algorithms for $k = 2$ strings

Throughout this section we will assume the algorithms operate on two strings α and β of lengths n and m respectively, and that $m \leq n$. The following definitions will be useful in the descriptions of the algorithms. A *prefix* of a string α is any string that can be obtained by deleting zero or more symbols from the end of α. The i^{th} prefix α^i of α is the string $\alpha[1 \ldots i]$. A *suffix* of a string α is any string that can be obtained by deleting zero or more symbols from the start of α.

The Minimum Edit Distance problem with arbitrary costs on the edit operations was introduced by Wagner and Fischer [68]. They described a Dynamic Programming (DP) Algorithm, to solve that problem and the LCS problem for two strings, which runs in $O(nm)$ time and space. If α is being converted to β and $D(i,j)$ contains the Minimum Edit Distance between α^i and β^j, then the algorithm uses the following recurrence relation:

$$\begin{aligned}
D(0,0) &= 0 \\
D(i,0) &= D(i-1,0) + Del(\alpha[i]) \\
D(0,j) &= D(0,j-1) + Ins(\beta[j]) \\
D(i,j) &= Min \begin{cases} D(i-1,j) + Del(\alpha[i]), \\ D(i,j-1) + Ins(\beta[j]), \\ D(i-1,j-1) + Sub(\alpha[i],\beta[j]) \end{cases}
\end{aligned}$$

where $Del(a)$ is the cost of deleting symbol a, $Ins(a)$ is the cost of inserting symbol a, and $Sub(a,b)$ is the cost of substituting symbol b for symbol a. The algorithm evaluates all $D(i,j)$ $(0 \leq i \leq n,\ 0 \leq j \leq m)$ and $D(n,m)$ contains the value of the Minimum Edit Distance between the two strings. A corresponding set of edit operations to convert α to β can be recovered by a trace through the distance table, requiring $O(n+m)$ time, starting at $D(n,m)$ and finishing at $D(0,0)$.

In a simplified recurrence relation for the LCS problem, $L(i,j)$ contains the length of the LCS of α^i and β^j for $0 \leq i \leq n$ and $0 \leq j \leq m$:

$$L(i,j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ L(i-1,j-1)+1, & \text{if } \alpha[i] = \beta[j] \\ \max(L(i-1,j), L(i,j-1)), & \text{otherwise.} \end{cases}$$

A possible strategy for a DP algorithm derived from the recurrence relation is to evaluate the L's in order of increasing i and increasing j. The process is illustrated in Figure 1.1.

An actual LCS can be recovered by a trace through the LCS table, requiring $O(n+m)$ time, which proceeds as follows. Starting at $L(n,m)$, scan left while the L value does not change, then scan up while the L value does not change. Select

		b	**a**	c	**d**	**a**
	0	0	0	0	0	0
a	0	0	*1*	1	1	1
c	0	0	1	1	1	1
d	0	0	1	1	*2*	2
b	0	1	1	1	2	2
a	0	1	2	2	2	*3*

LCS=**ada**

Figure 1.1: An example of dynamic programming for the LCS of 2 strings

the symbol $\sigma \in \Sigma$ represented by the current entry $L(i,j)$ - it is a consequence of the scans that $\alpha[i] = \beta[j] = \sigma$. Repeat the process starting at $L(i-1, j-1)$ until a symbol at an entry $L(i,j) = 1$ is selected. In Figure 1.1 the entries in italics show where symbols are selected by this process and the corresponding symbols appear in bold type. It is easy to see the time and space complexities of the algorithm are $O(nm)$.

A simple refinement DPL of DP can find the length of the LCS (or with a simple extension, the Minimum Edit Distance) but not an actual subsequence (or set of edit operations) in $O(nm)$ time and $O(n)$ space. When calculating the entries in row i ($L(i,0) \ldots L(i,m)$) of the table, only entries from rows i and $i-1$ are required. So when the value in entry $L(i-1, j-1)$ has been used in the evaluation of $L(i,j)$ it can be discarded and replaced with $L(i, j-1)$ - the value calculated just prior to $L(i,j)$. In this way, only one row of values is stored at any one time.

Hirschberg [25] suggested that the $O(nm)$ space complexity of DP would place a more severe restriction on the size of solvable problem instances than would the $O(nm)$ time complexity. He supported the claim with the following example. Consider two strings of length 1000. Assuming coefficients of 1 microsecond and 1 byte the basic DP algorithm would require 1 second and 1000 kilobytes to find the Minimum Edit Distance. To address this he provided an algorithm with $O(nm)$ time and $O(n)$ space complexity which returns the Minimum Edit Distance and a corresponding set of edit operations. The algorithm uses a divide and conquer strategy and applies DPL to compare successively smaller substrings of α with successively smaller substrings of β. The strategy is based on the following theorem, the proof of which is in [25]:

Theorem 1.3.1 *For all* i, $0 \leq i \leq n$,

$$D(n,m) = \min_{0 \leq j \leq m} (D(i,j) + D^*(i,j))$$

where D^* represents the Minimum Edit Distance between the suffixes $\alpha[i+1\ldots n]$ and $\beta[j+1\ldots m]$.

The algorithm DPL extended for the Minimum Edit Distance is applied to find $D(\lfloor n/2\rfloor, j)$ and $D^*(\lfloor n/2\rfloor, j)$. The optimum position j' to split β is calculated using Theorem 1.3.1 with $i = \lfloor n/2\rfloor$. This strategy is then applied recursively to $\alpha[1\ldots\lfloor n/2\rfloor]$ with $\beta[1\ldots j']$ and $\alpha[\lfloor n/2\rfloor+1\ldots n]$ with $\beta[j'+1\ldots m]$. Individual edit operations are identified when the substrings of α reach length 1.

Hirschberg [26] proposed two new algorithms aimed at speeding up computation of the LCS of two strings based on evaluation of the following structure. Define *the threshold* $\tau_{i,k}$ so that if $\tau_{i,k} = j$ then α^i and β^j have an LCS of length k but α^i and β^{j-1} have an LCS of length $k-1$. If α^i does not have a common subsequence as long as k with any prefix of β then then we define $\tau_{i,k} = m+1$. The τ's can be organised in a 2-dimensional table indexed by i and k and it is clear that the largest k for which $\tau_{n,k} \leq m$ is the length of the LCS.

Hirschberg's first algorithm for computing the table of τ values has time complexity $O(nl + n\log n)$ where l is the length of the LCS. This will be faster than DP when the LCS is short, and in the worst case is no worse than DP. The second algorithm has time complexity $O(l(m-l)\log n)$. When the LCS is long (probably the interesting cases in practical applications) this algorithm will be much faster than DP. However in the worst case it has complexity no better than $O(nm\log n)$. The space complexity of both algorithms is $O(nm)$.

Independently Hunt and Szymanski [30] developed the same threshold structure as Hirschberg and proposed a third strategy for the evaluation of the table. Their algorithm has time complexity $O(r\log m)$ where r is the number of matches between symbols of the two strings. In the worst case the number of matches can be nm giving $O(nm\log m)$ complexity for the algorithm. However when the number of matches is low, which is likely when the alphabet is very large, the algorithm will be very fast. For this reason the Hunt and Szymanski algorithm is used as the basis of the Unix *diff* command for file comparison. In this program, each line of a file is treated as a symbol and the LCS of two files is computed on this principle. The number of matches between two files of n lines will often be fewer than n since often no two lines in the one file are identical. The space complexity of the algorithm is $O(nm)$.

The algorithm of Masek and Paterson [45] to find the Minimum Edit Distance broke new ground in that it was the first (and so far the only) algorithm to achieve a worst-case time bound better than $O(n^2)$, for two strings of length n, in the worst case. Their algorithm uses the so-called "Four Russians" technique and achieves a worst-case time complexity of $O(n^2/\log n)$. The complexity is dependent upon a fixed size alphabet and all the edit distances being integer multiples of some

fixed real number. An often quoted result is that an extension of the algorithm has $O(n^2 \log\log n / \log n)$ complexity when the alphabet is unbounded. However no publication of that result could be found. Hirschberg [27] cites a private communication of the result from T.G. Szymanski.

The algorithm is based on computation of the same dynamic programming table as the Wagner and Fischer Algorithm. However, instead of calculating the entire $(n+1) \times (n+1)$ table, every possible table with dimensions $(i+1) \times (i+1)$, where $i = \theta(\log n)$, is calculated using the dynamic programming strategy. If the alphabet is fixed and i is set appropriately, the time required does not affect the algorithm's overall complexity, because of the limited number of possible tables with dimensions $(i+1) \times (i+1)$.

Each smaller table becomes a mapping from an initial row and column of edit distances, and one substring of each string, to a final row and column of edit distances. These smaller tables are then used to construct enough of the $(n+1) \times (n+1)$ table to find the Minimum Edit Distance and to construct a corresponding sequence of edit operations.

Despite the algorithm's good asymptotic behaviour, in practice the strings must have length greater than 200,000 before the algorithm outperforms even the basic Dynamic Programming Algorithm. This is due to a large constant factor in the complexity and a number of practical problems faced when implementing the algorithm

A number of algorithms in the literature are based on the basic Dynamic Programming Algorithm but attempt to improve upon it by suppressing unnecessary computations. The first was by Ukkonen [66] and it solves the more general Minimum Edit Distance problem.

Assuming the two strings have length n, the Distance Table is represented as a list of $2n+1$ diagonals numbered $(-n, 1-n, 2-n, \ldots, 0, 1,2, \ldots, n)$. Diagonal 0 contains the entries $D(0,0)$, $D(1,1)$, $\ldots$, $D(n,n)$. For $i < 0$, diagonal i contains the entries $D(abs(i),0)$, $D(abs(i)+1,1)$, $D(abs(i)+2,2)$, $\ldots$, $D(n, n-abs(i))$ where $abs(i)$ is the absolute value of i. For $i > 0$, diagonal i contains the entries $D(0,i)$, $D(1,i+1)$, $D(2,i+2)$, $\ldots$, $D(n-i,n)$. Evaluation proceeds along the entries of diagonal 0. When computing a particular position, paths from position 0 of diagonal 0 to that position are calculated as far as is necessary to be sure that they cannot contribute to its value. The terminating condition is based on the number of editing steps required for a path to affect the current position and the minimum of all the editing step costs. This facilitates "lazy evaluation" of the dynamic programming table. When $D(n,n)$ is evaluated the Minimum Edit Distance is known and a set of edit operations can be recovered by the same trace through the distance table as is used in Wagner and Fischer's algorithm. Ukkonen applies the strategy to two strings of differing length.

A new threshold algorithm

An algorithm with $O(nz + l(m-l))$ time and $O(nz)$ space complexity, where z is the size of the alphabet, was developed by Irving and Fraser. The algorithm employs the table of threshold values introduced by Hirschberg [26] and Hunt and Szymanski [30]. It uses an efficient strategy for evaluating the table to achieve the time bound and applies Hirschberg's [25] divide-and-conquer technique to achieve the space bound.

The algorithm computes the *next occurrence table* for α defined by

$$next_\alpha(i,a) = \begin{cases} \min\{j : \alpha[j] = a, j > i\}, & \text{if such a } j \text{ exists} \\ n+1, & \text{otherwise.} \end{cases}$$

It is easy to see that the next occurrence table will take $O(nz)$ time to evaluate and will require $O(nz)$ space.

The basis for the algorithm is in the following lemma.

Lemma 1.3.1 $\tau_{i,k} = \min(\tau_{i-1,k}, next_\alpha(\tau_{i-1,k-1}, \beta[i]))$

Proof
If β^{i-1} and α^j have a common subsequence of length k then so must β^i and α^j. If β^{i-1} and $\alpha^{j'}$, for some $j' < j$, have a common subsequence of length $k-1$ and $\beta[i] = \alpha[j]$ then β^i and α^j have a common subsequence of length k.

If $\tau_{i,k} = j$ then either β^{i-1} and α^j have an LCS of length k or $\beta[i] = \alpha[j]$ is the last element of an LCS of β^i and α^j. If the latter is true then β^{i-1} and $\alpha^{j'}$, for some $j' < j$ such that $next_\alpha(j', \beta[i]) = j$, must have a common subsequence of length $k-1$. □

We will first look at an $O(nz + l(m-l))$ time and space algorithm and then see how to reduce the space complexity to $O(nz)$. The algorithm begins evaluation of the τ's on the main diagonal of the threshold table - i.e. it uses Lemma 1.3.1 to evaluate $\tau_{1,1}, \tau_{2,2}, \ldots$ until $\tau_{i,i} = n+1$ indicating that there is no prefix of β which has an LCS of length i with α^i. Following this, the diagonal containing $\tau_{2,1}, \tau_{3,2}, \ldots$ is evaluated, again until the first τ equal to $n+1$ is reached. The process continues until $\tau_{n,l} < n+1$ is evaluated for some l. The length of the LCS will be this l and an actual LCS can be recovered by a fairly straightforward trace-back through the threshold table. Each diagonal has length at most l, and $m-l$ diagonals are evaluated. Every threshold value requires constant time to evaluate so the time complexity is $O(nz + l(m-l))$. By generating the diagonals dynamically the space requirement can be limited to $O(nz + l(m-l))$. The length of the LCS can easily be found using $O(nz)$ space if we waive the need to recover an actual LCS and calculate each diagonal from the previous diagonal in-situ. We will refer to this form of the algorithm as TL (Threshold Linear-space).

Now we will see how to reduce the space complexity by applying a divide and conquer strategy to the evaluation of the threshold table. The algorithm evaluates the threshold table using algorithm TL above - the forward pass. In addition, using algorithm TL an equivalent table is evaluated for the strings α^* (the reverse of α) and β^* (the reverse of β) - the backward pass. During the forward pass, all $\tau_{\lceil m/2 \rceil,k}$ are retained in an array F. During the backward pass all $\tau^*_{\lceil m/2 \rceil,k}$ are retained in an array B, where $\tau^*_{i,k} = j$ means that $\beta[i+1 \ldots m]$ and $\alpha[j+1 \ldots n]$ have an LCS of length k but $\beta[i+1 \ldots m]$ and $\alpha[j+2 \ldots n]$ have an LCS of length $k-1$. The following is an analogue of Theorem 1.3.1 and can be proved in a similar way.

Theorem 1.3.2 *For all i, $0 \leq i \leq m$ and $0 \leq k, k' \leq m$,*

$$l = \max_{\tau_{i,k} < \tau^*_{i,k'}} (k + k'),$$

where l is the length of the LCS.

The string β is split into its prefix $\beta_p = \beta[1 \ldots \lceil m/2 \rceil]$ and its suffix $\beta_s = \beta[\lceil m/2 \rceil + 1 \ldots m]$. The position to split α into a prefix α_p and a suffix α_s so as to maximise the sum of the LCS's of α_p, β_p and α_s, β_s and hence the overall LCS is calculated using Theorem 1.3.2 and the arrays F and B. This strategy is applied recursively to α_p with β_p and to α_s with β_s until the substrings of β have length one. At this point, each substring of β will have a corresponding substring of α. The one-symbol substrings of β that appear in their corresponding substrings of α form the LCS.

Since the same number of table entries are evaluated in the forward passes as in the backward passes, we need only calculate the number of entries evaluated in the forward passes. At the first level of recursion, $l(m-l)$ entries are evaluated. If α_p and β_p have an LCS of length l_1 and α_s and β_s have an LCS of length l_2 then $l_1 + l_2 = l$ and at the second level of recursion, the number of entries evaluated is

$$\begin{aligned} & l_1(\frac{m}{2} - l_1) + l_2(\frac{m}{2} - l_2) \\ = \; & (l_1 + l_2)\frac{m}{2} - (l_1^2 + l_2^2) \\ = \; & l\frac{m}{2} - (l_1^2 + l_2^2). \end{aligned}$$

This is maximised for $l_1 = l_2 = \frac{l}{2}$ giving

$$\frac{l(m-l)}{2}.$$

Repeating this analysis at each level of recursion, the total number of entries eval-

uated comes out as

$$\begin{aligned} & l(m-l) + \frac{l(m-l)}{2} + \frac{l(m-l)}{4} + \frac{l(m-l)}{8} + \cdots \\ = \; & l(m-l) \sum_{x=0}^{\log_2 n} \frac{1}{2^x} \\ \leq \; & 2l(m-l). \end{aligned}$$

Hence the time complexity, including pre-computation of the next occurrence table for α, is $O(nz + l(m-l))$. The space complexity depends on the space required to evaluate a threshold table and compute the optimum position to split α, and on the depth of the recursion. Clearly $O(n)$ space is required to evaluate a threshold table and the recursion depth is logarithmic in the length of β. Therefore the pre-computation of the next occurrence table for α is the dominant factor for the whole algorithm and the space complexity is $O(nz)$. Apostolico and Guerra [6] showed how the next occurrence table can be reduced to size n rather than $z \times n$ at the expense of an extra $\log z$ factor in the time complexity.

A brief summary of the other published algorithms for the LCS of two strings follows. Nakatsu, Kambayashi, and Yajima [52], Hsu and Du [29], Apostolico and Guerra [6], Chin and Poon [13], and, very recently, Rick [57] described alternative strategies for evaluating the threshold table. One of the algorithms of Rick [57] is similar to the algorithm of Irving and Fraser just described. Independently but significantly later, Mukhopadhyay [50] developed the same algorithm as Hunt and Szymanski. Apostolico [4], Apostolico and Guerra [6], and Kuo and Cross [42] described improvements to the algorithm of Hunt and Szymanski. The algorithms by Apostolico and by Apostolico and Guerra depend on the number of *dominant matches* (also referred to as *minimal candidates*) instead of the number of matches. A dominant match (i, j) $(\alpha[i] = \beta[j])$ is a position such the LCS of α^i and β^j has length l but the LCS of α^{i-1} and β^j and the LCS of α^i and β^{j-1} both have length $l-1$. There are likely to be significantly fewer dominant matches than straightforward matches. Kumar and Rangan [41] and Apostolico, Brown, and Guerra [5] applied the divide and conquer technique to existing LCS algorithms to derive algorithms requiring space linear in the lengths of the strings. Myers [51], Hadlock [21], and Wu, Manber, Myers, and Miller [71] described alternative strategies for suppressing evaluation of unnecessary parts of the dynamic programming table. Allison and Dix [3] described an implementation of dynamic programming that achieves a speedup of the order of the word length in a computer, over basic DP. Allison [2] described an implementation of dynamic programming in a lazy functional language.

Summary of the algorithms for the LCS of $k = 2$ strings

Table 1.1 summarises, in chronological order, the characteristics of the published algorithms for finding the LCS of two strings. In the table, n and m ($m \leq n$) are the lengths of the strings, l is the length of the LCS, r is the number of matches between the two strings, d is the number of dominant matches between the two strings, c is the number of edit operations, and z is the number of symbols in the alphabet. If the entry in the author column is marked with a '*' then that algorithm solves the more general Minimum Edit Distance problem.

1.3.3 Exact algorithms for $k > 2$ strings

Throughout this section, we will assume for simplicity that the algorithms operate on k strings of length n. However, all the algorithms apply equally well to the case of varying length strings.

While there are very many available algorithms to find the LCS of two strings, very few have been proposed to find the LCS of three or more strings. Itoga [35] extended the basic Dynamic Programming Algorithm to find the LCS of an arbitrary number of strings. In his paper, Itoga presents an algorithm but here we show a recurrence relation for solving the problem with dynamic programming. If $L(i_1, i_2, \ldots, i_k)$ represents the length of the LCS between $\alpha_1^{i_1}$, $\alpha_2^{i_2}$, ..., and $\alpha_k^{i_k}$ then

$$
\begin{aligned}
L(0,0,\ldots,0) &= 0 \\
L(i_1,i_2,\ldots,i_k) &= \begin{cases} L(i_1-1,i_2-1,\ldots,i_k-1)+1, & \text{if } \forall j\ i_j > 0 \text{ and } \alpha_j[i_j] = \sigma \\ \max\limits_{1\leq j\leq k} \lambda_j, & \text{otherwise} \end{cases}
\end{aligned}
$$

where

$$
\lambda_j = \begin{cases} 0, & \text{if } i_j = 0 \\ L(i_1,i_2,\ldots,i_j-1,\ldots,i_k), & \text{otherwise} \end{cases}
$$

and σ is some arbitrary symbol of the alphabet.

The algorithm derived from the recurrence relation evaluates $O(n^k)$ entries each requiring $O(k)$ time so the time complexity is $O(kn^k)$ and the space complexity is $O(n^k)$.

Hsu and Du [28] described a strategy for finding the LCS applicable to two strings and generalised it for an arbitrary number of strings. Their algorithm is based on evaluation of what they call a *common subsequence tree* (CS-tree). The tree contains a root and one node representing each match between symbols in every string. A σ−match is an ordered k-tuple $(i_1, i_2, \ldots, i_k)$ such that $\alpha[i_1] = \alpha[i_2] = \cdots = \alpha_k[i_k] = \sigma$. We say match y between all the strings follows match x if, for every string, the position represented by match y follows the position represented by match x. In the CS-tree, for every $\sigma \in \Sigma$, there is an edge from the node representing match x to

Authors	**Year**	**Time**	**Space**
* Wagner and Fischer [68]	1974	$O(nm)$	$O(nm)$
* Hirschberg [25]	1975	$O(nm)$	$O(n)$
Hirschberg [26]	1977	$O(nl + n.\log n)$ $O(l(m-l)\log\ n)$	$O(nm)$ $O(nm)$
Hunt and Szymanski [30]	1977	$O(r\ \log\ n)$	$O(r)$
* Masek and Paterson [45]	1980	$O(n^2/\log\ n)$	$O(n^2)$
Mukhopadhyay [50]	1980	$O(r \log n)$	$O(r)$
Nakatsu, Kambayashi, and Yajima [52]	1982	$O(n(m-l))$	$O(nm)$
Hsu and Du [29]	1984	$O(ml\ \log(n/m) + ml)$ $O(ml\ \log(n/l) + ml)$	$O(nm)$ $O(nm)$
* Ukkonen [66]	1985	$O(mc)$	$O(c.\min(m,c))$
Apostolico [4]	1986	$O(m.\log n + d.\log(nm/d))$	$O(d)$
Myers [51]	1986	$O(n(m-l))$	$O(nm)$
Allison and Dix [3]	1986	$O(nm)$	$O(nm)$
Apostolico and Guerra [6]	1987	$O(ml.\log(\min(z,m,2n/m)))$ $O(m.\log n + d.\log(2nm/d))$	$O(d)$ $O(d)$
Kumar and Rangan [41]	1987	$O(n(m-l))$	$O(n)$
* Hadlock [21]	1988	$O(mc)$	$O(nm)$
Kuo and Cross [42]	1989	$O(r + n(l + \log n))$	$O(r)$
Chin and Poon [13]	1990	$O(nz + \min(dz, ml))$	$O(nz + d)$
Wu, Manber, Myers and Miller [71]	1990	$O(n(m-l))$	$O(nm)$
Irving and Fraser	1991	$O(nz + l(m-l))$	$O(nz)$
Apostolico, Brown, and Guerra [5]	1992	$O(n(m-l))$ $O(ml.\log(\min(z,n,2n/l)))$	$O(n)$ $O(n)$
Allison [2]	1992	$O(nm)$	$O(nm)$
Rick [57]	1994	$O(nz + \min(dz, ml))$ $O(nz + \min(l(n-l), ml)$	$O(nz + d)$ $O(nz + d)$

* indicates the algorithm solves the Minimum Edit Distance problem.

Table 1.1: Summary of LCS algorithms for $k = 2$ strings

Authors	Year	Time	Space
Itoga [35]	1981	$O(kn^k)$	$O(n^k)$
Hsu and Du [28]	1984	$O(kr)$	$O(r)$
Irving and Fraser [32]	1992	$O(kn(n-l)^{k-1})$ $O(l(n-l)^{k-1}))$	$O(kn(n-l)^{k-1})$ $O(l(n-l)^{k-1})$
Hakata and Imai [22]	1993	$O(knz + kzd(\log^{k-3} n + \log^{k-2} z)$	$O(d)$

Table 1.2: Summary of LCS algorithms for $k \geq 2$ strings

the node representing the first σ-match following match x. The length of the LCS is therefore equal to the length of the longest path from the root to a leaf node.

The algorithm builds and traverses the tree in an efficient way to achieve $O(kr)$ time complexity and $O(r)$ space complexity, where r is the number of matches between all the strings. In the worst case this is the same as dynamic programming. However when the number of matches is low, as is likely when the alphabet is large, the algorithm should require much less time and space than dynamic programming.

Chapter 2 of this thesis describes two previously published algorithms [32] which generalise two algorithms for the LCS of two strings to find the LCS of an arbitrary number of strings.

Hakata and Imai [22] described an algorithm suitable for finding the LCS of several strings over a small alphabet. Their algorithm is based on that of Chin and Poon [13] for the LCS of two strings and as such utilises the *threshold approach* described in Section 1.3.2. The time complexity of the algorithm is $O(knz + kzd(\log^{k-3} n + \log^{k-2} z)$ where z is the alphabet size and d is the number of dominant matches. The meaning of dominant matches for k strings generalises in a straightforward way from the meaning for two strings.

Summary of the algorithms for the LCS of $k > 2$ strings

Table 1.2 summarises, in chronological order, the characteristics of the published algorithms for finding the LCS of $k > 2$ strings. In the table, n is the length of the strings, l is the length of the LCS, r is the number of matches between all the strings, d is the number of dominant matches between all the strings, and z is the number of symbols in the alphabet.

1.3.4 Approximation algorithms for the LCS

Despite the negative results, cited in Section 1.3.1, on the approximability of the LCS problem there has been a little work on finding effective approximation algorithms for the LCS problem. Jiang and Li [37] proposed an algorithm they called "Long-Run".

For a set P of strings, the algorithm proceeds as follows. It finds the maximum i such that the string σ^i is a common subsequence of all the strings in P, for some $\sigma \in \Sigma$, and returns that string as the common subsequence. They analysed the average performance of the algorithm showing that on average it returns a subsequence of length $(l - O(l^{1/2+\epsilon}))$ for any $\epsilon > 0$ where l is the length of the LCS.

Chin and Poon [12] proposed essentially the same algorithm as Jiang and Li for approximating the LCS and appeared to be unaware of that previous work. They showed that in the worst case, the algorithm returns a subsequence of length l/z where l is the length of the LCS and z is the size of the alphabet. They provide a few heuristics for improving the performance of the algorithm in practice and show they all have the same worst-case performance. A general result regarding approximation algorithms is established: If an approximation algorithm exploits only global character frequency information then it might return a subsequence as short as l/z. Finally, they analyse the average performance of the algorithm over a binary alphabet.

Chapter 5 of this thesis has an analysis of the worst-case behaviour of "Long-Run" and of another natural approximation algorithm for the LCS.

1.3.5 Expected LCS lengths

The expected Longest Common Subsequence length over a given set of parameters (alphabet size (z), number of strings (k) and string lengths (n)), denoted here by $f(z, k, n)$ is the average LCS length over all instances of the problem with that given set of parameters. As n tends to infinity, the limiting value for the ratio of the expected length of the LCS to n is defined to be

$$F(z, k) = \lim_{n \to \infty} \frac{f(z, k, n)}{n}.$$

The existence of this limit was established by Deken [16]. Measuring the expected length of the LCS is worthwhile because it provides a frame of reference in which to assess the similarity of real strings under analysis.

Chvátal and Sankoff studied the expected LCS length for two strings. They calculated $f(2, 2, n)$ for all $n \leq 5$. For $2 \leq z \leq 15$ they derived upper and lower bounds for $F(z, 2)$. By experimentation, they estimated $f(z, 2, 100)$ for $2 \leq z \leq 15$, $f(2, 2, 1000)$, and $f(2, 2, 5000)$.

Deken [16] provided two new methods for deriving lower bounds for $F(z, 2)$, one which gives a tighter than previously known bound when $z = 2$ and one which gives a tighter than previously known bound when $3 \leq z \leq 15$. He showed that the rate of decrease of $F(z, k)$ with increasing z is no faster than $1/\sqrt{z}$.

Recently, Dančík and Paterson [15] provided a new method for calculating upper

bounds for $F(z,2)$ and derived a stronger upper bound for $F(2,2)$. They suggested that their method may be used to find a slightly tighter bound but that a new approach will be required to find the exact value.

The tightest theoretical bounds to date and estimates from experimental results of $F(z,2)$ for $2 \leq z \leq 16$ appear in Section 4.8.3 of this thesis. For $k > 2$ estimates from experimental results of $F(z,k)$ and, where experimentation was possible, $G(z,k)$, for $z = 2,4,8,16$ and $k = 3,4,\ldots,8,16,24$ are also given, in Sections 4.8.1 and 4.8.2 respectively.

1.4 The Shortest Common Supersequence Problem

1.4.1 Complexity results

The Shortest Common Supersequence problem was shown to be **NP**-complete even over an alphabet of size five by Maier [44]. The proof was by a transformation from the well known Vertex Cover problem. Räihä and Ukkonen [56] showed that **NP**-completeness applies even over a binary alphabet. Timkovsky[64] considered a set of restricted versions of the SCS problem – SCS(n,r) where all the strings have length $\leq n$ and no character from the alphabet appears more that r times throughout all the strings. He showed that SCS(2,3) and SCS(3,2) are **NP**-complete but that SCS(2,2) can be solved in polynomial time. Jiang and Li [37] showed that SCS(2,3) is **MAX SNP**-hard implying that it has no polynomial-time approximation scheme unless **P=NP**. Middendorf [48] proved that the SCS problem is **NP**-complete over a binary alphabet even if the strings all have the same length and all contain precisely three ones.

Jiang and Li [37] showed that over an unbounded alphabet the SCS problem does not have a polynomial-time approximation algorithm with a constant performance guarantee unless **P=NP**. Further, if over an unbounded alphabet the SCS problem has a polynomial-time approximation algorithm with performance guarantee $O(\log\log k)$ then **NP**is contained in DTIME($2^{polylog k}$), where k is the number of strings.

Recently, Bonizzoni, Duella, and Mauri [9] showed that the SCS problem is **MAX SNP**-hard even over a binary alphabet. This means that no polynomial-time approximation scheme exists for the problem even in that special case unless **P=NP**. The problem class **MAX SNP**is discussed in Section 1.7 of this chapter.

1.4.2 Exact algorithms for $k > 2$ strings

Since the LCS and SCS problems are dual when there are only two strings, there are many algorithms for finding the SCS of two strings. However, there is only one algorithm in the literature for finding the SCS of more than two strings. Itoga [35] and later Foulser, Li, and Yang [17] described essentially the same Dynamic Programming Algorithm to find the SCS of any fixed number of strings. Here we show a recurrence relation to solve the problem with dynamic programming. If $S(i_1, i_2, \ldots, i_k)$ represents the length of the SCS of $\alpha_1^{i_1}$, $\alpha_2^{i_2}$, $\ldots$, $\alpha_k^{i_k}$ then

$$\begin{aligned} S(0,0,\ldots,0) &= 0 \\ S(i_1, i_2, \ldots, \ldots, i_k) &= \min_{\lambda_{\sigma_j}} +1 \end{aligned}$$

where $\sigma_j \in \{\alpha_1[i_1], \alpha_2[i_2], \ldots, \alpha_k[i_k]\}$ and

$$\begin{aligned} \lambda_{\sigma_j} &= S(\mathcal{F}(i_1, \alpha_1[i_1], \sigma_j), \mathcal{F}(i_2, \alpha_2[i_2], \sigma_j), \ldots, \mathcal{F}(i_k, \alpha_k[i_k], \sigma_j)), \\ \mathcal{F}(h, \sigma_p, \sigma_q) &= \begin{cases} 0, & \text{if } h = 0 \\ h-1, & \text{if } h > 0 \text{ and } \sigma_p = \sigma_q \\ h, & \text{otherwise.} \end{cases} \end{aligned}$$

The number of λ's at $S(i_1, i_2, \ldots, i_k)$ is limited by the number of distinct symbols in the set $\{\alpha_1[i_1], \alpha_2[i_2], \ldots, \alpha_k[i_k]\}$ which is limited by the number of strings, so this leads to an algorithm with $O(kn^k)$ time and $O(n^k)$ space complexity.

Chapter 3 of this thesis describes two new algorithms for the SCS of an arbitrary number of strings.

1.4.3 Approximation algorithms for the SCS

Despite the negative results, cited in Section 1.4.1, on the approximability of the SCS problem there has been some work on finding effective approximation algorithms.

Timkovsky [64] proposed a tournament style approximation algorithm and asked for a characterisation of its worst-case performance. For a set P of k strings, the algorithm proceeds as follows. There are $\lceil \log_2 k \rceil$ rounds. In each round, the strings are paired up arbitrarily and an arbitrary SCS of every pair is found using any algorithm for the SCS of two strings, and that SCS is entered into the next round. Bradford and Jenkyns [11] showed that the Tournament Algorithm could not guarantee to find the SCS although Timkovsky had not suggested that it would do so.

Jiang and Li described a greedy algorithm called Majority-Merge. The algorithm proceeds as follows. Initialise the supersequence to the empty string. Let σ be the most common symbol of the leftmost symbols of the remaining strings. Ties are decided arbitrarily. Append σ to the supersequence and remove it from the front

of the strings of which it is a prefix. Repeat this process until all the strings are exhausted. They showed that on average Majority-Merge returns a supersequence of length $(s + O(s^{0.707}))$ where s is the length of the SCS.

Foulser, Li, and Yang [17] analysed four greedy approximation algorithms similar to Majority Merge (in fact one is precisely that algorithm) for the SCS problem. They provided both worst and average case analysis for the algorithms and some empirical results of their behaviour.

Chapter 5 of this thesis contains an analysis of the worst-case behaviour of the Tournament Algorithm, Majority-Merge, and four other approximation algorithms for the SCS problem over both unbounded and bounded alphabets. Some empirical results on the behaviour of the algorithms are also presented.

1.5 Problems related to LCS and SCS

1.5.1 Maximal subsequences and minimal supersequences

A common subsequence α of a set P of strings is *maximal* if no proper supersequence of α is also a common subsequence of P. A *Shortest Maximal Common Subsequence* (SMCS) of P is a maximal common subsequence of shortest possible length. Clearly, a maximal common subsequence of *longest* possible length is just a Longest Common Subsequence.

Analogously, a common supersequence α of a set P of strings is *minimal* if no proper subsequence of α is also a supersequence of P. A *Longest Minimal Common Supersequence* (LMCS) of P is a minimal common supersequence of longest possible length. Clearly, a minimal common supersequence of *shortest* possible length is just a Shortest Common Supersequence.

In Chapter 6 of this thesis the SMCS problem is shown to be **NP**-hard and a very strong negative result on the approximability of the SMCS, over an unbounded alphabet, is established. Polynomial-time algorithms are provided for both problems over any fixed number of strings.

Subsequently, Middendorf [49] has proved that even over a binary alphabet, finding an SMCS is **MAX SNP**-hard and finding an LMCS is **MAX SNP**-hard. This implies that (unless **P=NP**) the problems cannot be solved in polynomial time and they have no polynomial-time approximation scheme.

1.5.2 Negative subsequences and supersequences

Problems involving *negative strings*, known as *string non-inclusion* problems, were introduced by Timkovsky [64]. Given a set N of strings, a *common non-subsequence* of N is a string that is not a subsequence of any string in N. A *common non-*

supersequence of N is a string that is not a supersequence of any string in N. The decision version of the Shortest Common Non-subsequence (SCNS) problem is to determine, for a finite set N of strings over an alphabet Σ, and an integer $t \in N$, whether there exists a string of length $\leq t$ over Σ which is a subsequence of no string in N. The decision version of the Longest Common Non-supersequence (LCNS) problem is to determine, for a finite set N of strings over an alphabet Σ, and an integer $t \in N$, whether there exists a string of length $\geq t$ over Σ which is a supersequence of no string in N. Both problems can be re-formulated naturally as optimisation problems. There is also the problem of deciding whether *any* LCNS exists for a set of strings.

Rubinov and Timkovsky [58], and independently Middendorf [47], proved that the Shortest Common Non-subsequence problem is **NP**-complete. Middendorf's result holds even when the alphabet size is fixed at two. Rubinov and Timkovsky also showed that when either the number of strings or the lengths of the strings is bounded then the SCNS problem can be solved in polynomial time.

Rubinov and Timkovsky [58] showed that determining whether there exists a Longest Common Non-supersequence (of any length) over a fixed alphabet is trivially solvable in polynomial time. Specifically, an LCNS exists for N if and only if there is a string in N of the form σ^t for all $\sigma \in \Sigma$. However, they and independently Zhang [73] showed that the Longest Common Non-supersequence problem is **NP**-complete. Zhang's result applies even when the alphabet has size two and he also showed that over an unbounded alphabet, the problem is **MAX SNP**-hard. The result of Rubinov and Timkovsky applies when the alphabet is unbounded but the strings all have length two. They gave a polynomial-time algorithm for the case when the number of strings is bounded.

1.5.3 Consistent subsequences and supersequences

If we combine elements of the sequence-inclusion problems (LCS and SCS) and the sequence non-inclusion problems (SCNS and LCNS) then we obtain *consistent sequence* problems. Given two sets, P (Positive) and N (Negative), of strings, a *consistent subsequence* of P and N is a string that is a common subsequence of P and a common non-subsequence of N. A *consistent supersequence* is defined similarly.

There are two categories of problems, existence problems and optimisation problems. Given two sets, P and N, of strings, does there exist a consistent subsequence (supersequence)? If a consistent subsequence (supersequence) does exist, what is the length of the shortest or longest? There are therefore two existence problems and four optimisation problems. It is clear that an **NP**-completeness result for an existence problem implies **NP**-hardness for the two corresponding optimisation problems and further that no approximation is possible for them. Similarly, the

existence of a polynomial-time algorithm for an optimisation problem implies the corresponding existence problem is also in **P**. For the **NP**-complete problems, the following questions arise: do they become solvable in polynomial time if we bound $|P|$, or bound $|N|$, or bound both $|P|$ and $|N|$ (if the answer to the first two questions is no)?

It is clear that a number of the consistent optimisation problems are generalisations of previously studied sequence inclusion and sequence non-inclusion optimisation problems and thus some of the problems inherit **NP**-hardness. Four of the optimisation problems inherit **NP**-hardness in this way. When $|P|$ is unbounded, the Longest Consistent Subsequence problem is a generalisation of the Longest Common Subsequence problem. Similarly, when $|P|$ is unbounded, the Shortest Consistent Supersequence problem is a generalisation of the Shortest Common Supersequence problem. When $|N|$ is unbounded, the Shortest Consistent Subsequence problem is a generalisation of the Shortest Common Non-subsequence problem. Similarly when $|N|$ is unbounded, the Longest Consistent Supersequence problem is a generalisation of the Longest Common Non-supersequence problem.

The following results all apply over a binary alphabet. Jiang and Li [38] proved: (i) finding a consistent supersequence is **NP**-complete even when $|P| \geq 2$ is bounded ($|N|$ is unbounded) and (ii) a consistent supersequence can be found in polynomial time when $|P|$ is unbounded and $|N| = 1$. Zhang [73] proved: (i) the Shortest Consistent Subsequence problem is **MAX SNP**-hard even when $|P| = 2$ ($|N|$ is unbounded) and (ii) the Longest Consistent Supersequence problem is **MAX SNP**-hard even when $|P| = 1$ ($|N|$ is unbounded). Middendorf [49] strengthened Zhang's result (i) above to the case when $|P| = 1$. Further, he proved: (i) the Longest Consistent Subsequence problem is **MAX SNP**-hard even when $|P| = 1$ ($|N|$ is unbounded) and (ii) the Shortest Consistent Supersequence problem is **MAX SNP**-hard even when $|N| = 1$ ($|P|$ is unbounded). Of the existence problems Middendorf proved: (i) finding a consistent subsequence is **NP**-complete even when $|P| = 2$ ($|N|$ is unbounded) or $|N| = 2$ ($|P|$ is unbounded) and (ii) finding a consistent supersequence is **NP**-complete even when $|P| = 2$ ($|N|$ is unbounded) or $|N| = 2$ ($|P|$ is unbounded).

In Chapter 7 of this thesis the following existence problems are proved to be **NP**-complete: (i) Consistent Subsequence even when $|P| = 2$ ($|N|$ is unbounded) and the alphabet is binary, (ii) Consistent Subsequence even when $|N| = 1$ ($|P|$ is unbounded) and (iii) Consistent Supersequence even when $|N| = 2$ and the alphabet is binary. Polynomial-time algorithms are provided for the Longest (or Shortest) Consistent Subsequence and the Shortest Consistent Supersequence when both $|P|$ and $|N|$ are bounded. A table summarising the status of all variations of these problems is given.

1.5.4 Miscellaneous string comparison problems

Jacobson and Vo [36] generalised the LCS problem to the Heaviest Common Subsequence (HCS) problem. In this, matches have an associated weight dependent on the symbol involved and its position in the strings. The weight of a common subsequence is the sum of the weights of the matches from which it is formed. In the HCS problem, the common subsequence with the highest weight is sought. Jacobson and Vo provided an algorithm for the problem over two strings analogous to the algorithm of Hunt and Szymanski [30] for the LCS problem. For the case where the position of a match does not affect its weight, they provided an improvement to their algorithm analogous to the improvement to the Hunt and Szymanski algorithm due to Apostolico and Guerra [6].

Variations on the LCS and Minimum Edit Distance Problems include those studied by Hebrard [24], Lowrance and Wagner [43], and Tichy [63].

1.6 Substring and Superstring Problems

Analogous to the LCS and SCS problems are the Longest Common Substring and Shortest Common Superstring problems. Recall that, a *substring* of a string α is any string that can be obtained by deleting zero or more symbols from the start and deleting zero or more symbols from the end of α. A *superstring* of α is any string that can be obtained by prepending zero or more symbols and appending zero or more symbols to α. The Longest Common Substring (LCSt) problem is to determine, for a finite set P of strings over an alphabet Σ, and an integer $t \in N$, whether there exists a string of length $\geq t$ over Σ which is a substring of every string in P. The Shortest Common Superstring (SCSt) problem is to determine, for a finite set P of strings over an alphabet Σ, and an integer $t \in N$, whether there exists a string of length $\leq t$ over Σ which is a superstring of every string in P. The problems can be re-formulated in the natural way as optimisation problems.

1.6.1 The Longest Common Substring Problem

It has long been known that using a suffix tree, the LCSt of an arbitrary number of strings can be found in time proportional to the sum of the lengths of the strings.

Suffix trees

An important data structure in string comparison is the Suffix Tree. The structure was introduced by Weiner [69] along with a linear time construction algorithm. McCreight [46] described a linear time construction algorithm which requires significantly less space that Weiner's algorithm. Ukkonen [67] described a linear time

algorithm suitable for online construction of the suffix tree.

The main properties of a suffix tree are as follows. The suffix tree of a string is unique. To simplify the structure of the suffix tree T of a string α, a unique terminal symbol is appended to α prior to the construction of T. This ensures that no suffix of α is a prefix of any other suffix which in turn guarantees the following property. Every suffix of α is represented by a leaf node and every leaf node represents a suffix of α. Every edge of T is labelled with a non-empty substring of α. The concatenation of the the edge labels encountered in a path from the root to a leaf node form the suffix represented by that leaf node. Every node has at least two children and no two edge labels leading from a node start with the same symbol.

1.6.2 The Shortest Common Superstring Problem

The SCSt problem was shown to be **NP**-complete even over a binary alphabet by Gallant, Maier, and Storer [18]. They provided a polynomial-time algorithm for the case when all the strings have length ≤ 2. Blum, Jiang, Li, Tromp, and Yannakakis [8] proved that the problem is **MAX SNP**-hard.

Tarhio and Ukkonen [61] proposed a greedy approximation strategy (GREEDY) for the SCSt problem. The *overlap* between two strings is the maximum length suffix of either one which matches an equal length prefix of the other, in every position. The greedy algorithm repeatedly *merges* the two strings with the largest overlap by replacing them with their SCSt (e.g. if $\alpha = abac$ and $\beta = acba$ then they are replaced with $\gamma = abacba$) until only one string remains. The *compression* achieved by the algorithm is the sum of the overlaps of every pair of strings merged. They proved that GREEDY achieves at least half the compression of an SCSt. Note that this does not imply any constant bound on the length of the superstring returned by GREEDY with respect to the length of the SCSt. If we have k strings of length n and the SCSt has length xn for some real $x > 1$ then the compression in the SCSt is $kn - xn$. If GREEDY achieves $(kn - xn)/2$ compression then the sequence it returns has length $kn - (kn - xn)/2 = n(k + x)/2$. The ratio of the length of the approximation to the length of the SCSt increases with the limit $(k+1/2)$ as $x \to 1$. Turner [65] gave a much more concise and elegant proof that $GREEDY$ achieves at least half the optimal compression.

Blum et al. [8] proved that a variant of $GREEDY$ guarantees to return a superstring of length $\leq 3|\text{SCSt}|$. They conjectured that GREEDY actually guarantees to return a superstring of length $\leq 2|\text{SCSt}|$. Teng and Yao [62] improved the known bound by showing that a different variant of GREEDY guarantees to return a superstring of length $\leq 2.89|SCSt|$.

1.6.3 Negative and consistent substrings and superstrings

Given a set N of strings, a *common non-substring* of N is a string that is not a substring of any string in N. A *common non-superstring* of N is a string that is not a superstring of any string in N. We therefore have three natural problems. Given a set N of strings, (i) what is the length of the Shortest Common Non-substring (SCNSt) ? (ii) does there exist a Longest Common Non-superstring (LCNSt)? (iii) if an LCNSt does exist then what is its length?

Timkovsky [64] conjectured that if an LCNSt exists for a set N of strings then its length is bounded by the sum of the lengths of the strings in N. Rubinov and Timkovsky [58] proved this conjecture and gave polynomial time algorithms to solve both the existence and optimisation versions of the LCNSt problems and to solve the SCNSt problem.

Analogous to the consistent sequence problems, are the *consistent string* problems. Given two sets, P and N, of strings, a *consistent substring* of P and N is a string that is a common substring of P and a common non-substring of N. A *consistent superstring* is defined similarly.

There are two existence and four optimisation problems. Given two sets, P and N, of strings, does there exist a consistent substring (superstring)? If a consistent substring (superstring) does exist, what is the length of the shortest/longest?

The shortest consistent superstring problem is a generalisation of the shortest common superstring problem and thus inherits **NP**-completeness even over a binary alphabet. Jiang and Li [38] proved that finding a consistent superstring is **NP**-complete even if the number strings in P is bounded but greater than 1. They gave a polynomial time algorithm for the case where $|N| = 1$. Jiang and Timkovsky [39] gave polynomial time algorithms to find the Shortest or Longest Consistent Superstring when a Longest Common Non-superstring exists for N and to find the Shortest Consistent Superstring when the number of strings in P is bounded and every symbol of the alphabet appears at the end of some string in N.

In Section 7.5 of this thesis, polynomial-time algorithms are given to find the shortest or longest consistent substring, a consistent superstring when $|N| \leq 2$, and a shortest consistent superstring when $|P| = 1$. It is shown that when $|N| = 1$, there cannot exist a longest consistent superstring.

1.7 The complexity of approximation

In Chapter 5 we will examine some approximation algorithms for the LCS and SCS problems. It is therefore worthwhile taking a brief look at some recent developments on the approximability of **NP**-hard optimisation problems.

Papadimitriou and Yannakakis [53] gave a syntactic definition of a subset of

NP, called *strict* **NP** or **SNP**. They showed how to obtain the class **MAX SNP** containing the optimisation versions of the decision problems contained in **SNP**. They proved that any problem in **MAX SNP** can be approximated to within some constant factor in polynomial time, and showed that a number of **NP**-hard optimisation problems are members of **MAX SNP**. They defined a new kind of problem transformation called an *L-reduction.* In an L-reduction from problem Π_1 to problem Π_2 the approximability of Π_1 is preserved in Π_2 to within some constant factor. Hence if Π_2 can be approximated to within some constant factor then it follows that Π_1 can be approximated to within some constant factor. Conversely, if Π_1 cannot be approximated to within any constant factor then it follows that Π_2 cannot be approximated to within any constant factor. Under L-reductions, they showed that a number of optimisation problems are complete for **MAX SNP** e.g. Maximum Satisfiability and the Independent Set problem on bounded-degree graphs.

A significant development was the result of Arora, Lund, Motwani, Sudan, and Szegedy [7] that no polynomial-time approximation scheme exists for any **MAX SNP**-complete problem (unless **P**=**NP**). A thorough account of **MAX SNP** can be found in [54].

Chapter 2

Exact Algorithms for the LCS problem

2.1 Introduction

As was noted in Section 1.3.3, the Longest Common Subsequence problem for k strings of length n can be solved by dynamic programming in $O(kn^k)$ time and $O(n^k)$ space. The space complexity of dynamic programming can be improved to $O(n^{k-1})$ using the technique developed by Hirschberg[25]. See Section 1.3.3 for details of previous work. In this chapter we look at two alternative algorithms intended to provide, in particular circumstances, improvements on the time and space required by the dynamic programming algorithm. The two algorithms are presented in the context of finding the LCS of three strings $(\alpha_1, \alpha_2, \alpha_3)$. However, they can both be extended to work for any fixed number of strings and at the end of each section, this is explained. Throughout this chapter, l represents the length of the LCS.

2.2 The "Lazy" Approach to Dynamic Programming

Consider the problem of finding an LCS of 3 strings α_1, α_2 and α_3, each of length n. We shall assume equal length strings throughout, for simplicity, but all of our results can be extended in a straightforward way to cases where the strings are of different lengths. Denote by $L(\alpha_1, \alpha_2, \alpha_3)$ the length of an LCS of α_1, α_2 and α_3, and by $L(i, j, k)$ the length of an LCS of the i^{th} prefix $\alpha_1^i = \alpha_1[1 \ldots i]$ of α_1, the j^{th} prefix $\alpha_2^j = \alpha_2[1 \ldots j]$ of α_2 and the k^{th} prefix $\alpha_3^k = \alpha_3[1 \ldots k]$ of α_3. As in the classical case of two strings [68], a basic dynamic programming scheme, using $\Theta(n^3)$

time and space in this case, can be set up, based on the recurrence

$$L(i,j,k) = \begin{cases} L(i-1,j-1,k-1)+1 & \text{if } \alpha_1[i] = \alpha_2[j] = \alpha_3[k] \\ \max(L(i-1,j,k), L(i,j-1,k), L(i,j,k-1)) & \text{otherwise} \end{cases} \quad (2.1)$$

and the initial conditions

$$L(i,j,0) = L(i,0,k) = L(0,j,k) = 0 \quad \text{for all } i,\ j,\ k.$$

We now investigate how this dynamic programming approach can be speeded up by suppressing unnecessary evaluations. Our approach is similar to that employed in [51], [71] and [66] for the case of two strings, and we refer to our algorithm as the "lazy" approach to dynamic programming.

We first look at the problem from a slightly different point of view. Define the *difference factor* $D = D(\alpha_1, \alpha_2, \alpha_3)$ of the three strings to be the smallest total number of elements that need be deleted from the strings in order to leave three identical strings. Clearly D is a multiple of 3 if the strings are of equal lengths, since the same number of elements must be deleted from each string.

Lemma 2.2.1 $D(\alpha_1, \alpha_2, \alpha_3) = 3(n - L(\alpha_1, \alpha_2, \alpha_3))$.

Proof
The identical strings that remain when D elements are deleted from α_1, α_2, α_3 form an LCS of α_1, α_2 and α_3. Hence the deleted symbols, together with the 3 copies of the LCS constitute the $3n$ symbols of the original strings. □

It follows that evaluation of $D(\alpha_1, \alpha_2, \alpha_3)$ gives the value of $L(\alpha_1, \alpha_2, \alpha_3)$, and that identification of a minimum set of deletions reveals an LCS. This is the approach that we shall now follow.

Let $D(i,j,k)$ be the difference factor of the prefixes α_1^i, α_2^j and α_3^k. Then the recurrence corresponding to 2.1 above is

$$D(i,j,k) = \begin{cases} D(i-1,j-1,k-1) & \text{if } \alpha_1[i] = \alpha_2[j] = \alpha_3[k] \\ 1 + \min(D(i-1,j,k), D(i,j-1,k), D(i,j,k-1)) & \text{otherwise} \end{cases} \quad (2.2)$$

subject to

$$D(i,j,0) = i+j,\ D(i,0,k) = i+k,\ D(0,j,k) = j+k \quad \text{for all } i,\ j,\ k.$$

Clearly, in the light of this equation, cells (i,j,k) such that $\alpha_1[i] = \alpha_2[j] = \alpha_3[k]$ have a special significance — we refer to such a cell as *a match position.*

We now consider the task of evaluating $D(n,n,n)$ while calculating as few as possible of the other D values. We refer to the 3-dimensional table $D(i,j,k)$

$(1 \leq i, j, k \leq n)$ as the *D-table* and we say that the cell (i, j, k) has *D-value* $D(i, j, k)$. Note that we will include coordinate values of -1 and 0 when implementing the table to facilitate appropriate initialisation in our algorithm.

By a *diagonal* of the D-table we mean an ordered sequence of cells with coordinates $(i + l, j + l, k + l)$ where $\min(i, j, k) = -1$ and $l = 0, \ldots, n - \max(i, j, k)$. The diagonal containing cell (i, j, k) will be denoted by $\langle i, j, k \rangle$. We say that a given cell (i, j, k) occupies *position* $i + j + k$ on its diagonal. Note that the positions on diagonal $\langle i, j, k \rangle$ increase by 3 from cell to cell, and are all $\equiv i + j + k \pmod 3$.

We also define

$$\ll i, j, k \gg = \{\langle i, j, k \rangle, \langle i, k, j \rangle, \langle j, i, k \rangle, \langle j, k, i \rangle, \langle k, i, j \rangle, \langle k, j, i \rangle\} \ ,$$

noting that this set of diagonals is of size 6, 3 or 1 depending whether the number of distinct values in the set $\{i, j, k\}$ is 3, 2 or 1. Any such set of diagonals has a *canonical* representation as $\ll i, j, 0 \gg$ with $i \geq j \geq 0$.

The following lemma is immediate:

Lemma 2.2.2 *If* $\langle x, y, z \rangle \in \ll i, j, 0 \gg$, *with* $x, y, z \geq 0$ *and* $i \geq j \geq 0$, *then* $D(x, y, z) \geq i + j$.

The *neighbours* of diagonal $\langle i, j, k \rangle$ are the 3 diagonals $\langle i-1, j, k \rangle$, $\langle i, j-1, k \rangle$ and $\langle i, j, k-1 \rangle$ and the *neighbourhood* of the set $\ll i, j, k \gg$ consists of all the diagonals that neighbour at least one diagonal in that set. (Note that the neighbour relation is not a symmetric one — indeed it is antisymmetric, in that if diagonal $\langle i', j', k' \rangle$ is a neighbour of diagonal $\langle i, j, k \rangle$ then $\langle i, j, k \rangle$ is not a neighbour of $\langle i', j', k' \rangle$. However, the following lemma may be easily verified.

Lemma 2.2.3 *If diagonal* Δ' *is a neighbour of diagonal* Δ, *then there is a diagonal* Δ'' *such that* Δ'' *is a neighbour of* Δ' *and* Δ *is a neighbour of* Δ''.

The *neighbours* of a cell (i, j, k), likewise, are defined to be the cells $(i - 1, j, k)$, $(i, j - 1, k)$ and $(i, j, k - 1)$. Obviously, the cell in position p in a given diagonal has, as its neighbours, the cells in position $p - 1$ in each of the neighbouring diagonals.

For the cell in position p in a given diagonal Δ, let us denote the D-value of that cell by $D(p, \Delta)$. The following technical lemmas may be established easily from the definitions of difference factor, neighbour and position.

Lemma 2.2.4 *Let diagonal* Δ' *be a neighbour of diagonal* Δ. *Then*

$$D(p, \Delta) \leq 1 + D(p - 1, \Delta') \ .$$

Lemma 2.2.5 *The values in any diagonal of the D-table form a non-decreasing sequence, with the difference between successive elements in that sequence being 0 or 3.*

Description of the "Lazy" Algorithm

The objective of the p^{th} iteration of our so-called "lazy" algorithm is the determination of the last (i.e., highest numbered) position in the main diagonal (diagonal $\langle 0,0,0 \rangle$) of the D-table occupied by the value $3p$. But this will be preceded by the determination of the last position in each neighbouring diagonal occupied by $3p-1$, which in turn will be preceded by the determination of the last position in each neighbouring diagonal of these occupied by $3p-2$, and so on. As soon as we discover that the last position in the main diagonal occupied by $3p$ is position $3n$ (corresponding to cell (n,n,n)), we have established that $D(\alpha_1,\alpha_2,\alpha_3) = 3p$, and therefore, by Lemma 2.2.1, that $L(\alpha_1,\alpha_2,\alpha_3) = n - D(\alpha_1,\alpha_2,\alpha_3)/3 = n-p$.

In general, when we come to find the last position in diagonal $\langle i,j,k \rangle$ occupied by the value r, we will already know the last position in each neighbouring diagonal occupied by $r-1$. If we denote by $t(\Delta,r)$ the last position in diagonal Δ occupied by the value r, then we can state the fundamental lemma that underpins the lazy algorithm.

Lemma 2.2.6 *If $x = \max t(\Delta', r-1)$, where the maximum is taken over the neighbours Δ' of diagonal Δ, then*

$$t(\Delta,r) = x + 1 + 3s \ ,$$

where s is the largest integer such that positions $x+4, x+7, \ldots, x+3s+1$ on diagonal Δ are match positions. (These positions constitute a "snake" in the terminology of Myers [51] - albeit a very erect snake.)

Proof
It follows from Lemma 2.2.4 that $D(x+1,\Delta) \leq r$, and clearly if its value is $< r$ then it is $\leq r-3$. But in this latter case, by Lemmas 2.2.3 and 2.2.4, $D(x+3,\Delta') \leq r-1$ for Δ' a neighbour of Δ, contradicting the maximality of x. Hence $D(x+1,\Delta) = r$. Further, if $D(y,\Delta) = r$ for any $y > x+1$ then, since no neighbouring diagonal has a value less than r beyond position x, according to the basic dynamic programming formula (2.2) for D, it must follow that y is a match position in that diagonal. □

We define the *level* of a diagonal Δ to be the smallest m such that there is a sequence $\Delta_0 = \langle 0,0,0 \rangle, \Delta_1, \ldots, \Delta_m = \Delta$ with Δ_s a neighbour of Δ_{s-1} for each s, $s = 1, \ldots, m$. It may easily be verified that the level of $\Delta = \langle i,j,k \rangle$ is $f_{ijk} = 3\max(i,j,k) - (i+j+k)$, and therefore that the level of diagonal $\Delta = \langle i,j,0 \rangle$ $(i \geq j)$ is $2i-j$. This can be expressed differently, as in the following lemma.

Lemma 2.2.7 *The diagonals at level x are those in the sets $\ll x-y, x-2y, 0 \gg$ for $y = 0, 1, \ldots, \lfloor \frac{x}{2} \rfloor$.*

The next lemma is an easy consequence of Lemmas 2.2.2 and 2.2.7.

Lemma 2.2.8 *The smallest D-value in any cell of a diagonal at level x is $\frac{x}{2}$ if x is even, and $\frac{x+3}{2}$ if x is odd.*

During the p^{th} iteration of our algorithm, we will determine, in decreasing order of x, and for each relevant diagonal Δ at level x, the value of $t(\Delta, 3p-x)$. Since, by Lemma 2.2.8, no diagonal at level $> 2p$ can contain a D-value $\leq p$, we need consider, during this p^{th} iteration, only diagonals at levels $\leq 2p$. Furthermore, by Lemma 2.2.2, the only diagonals at level x that can include a D-value as small as $3p-x$ are those diagonals in the family $\ll x-y, x-2y, 0 \gg$ with $(x-y)+(x-2y) \leq 3p-x$, i.e., with $y \geq x-p$.

Hence, to summarise, the p^{th} iteration involves the evaluation, for $x = 2p, 2p-1, \ldots, 0$, of $t(\Delta, 3p-x)$ for $\Delta \in \ll x-y, x-2y, 0 \gg$ with $\max(0, x-p) \leq y \leq \lfloor \frac{x}{2} \rfloor$.

Lemma 2.2.9 *If the t values are calculated in the order specified above, then, for arbitrary Δ, r, the values of $t(\Delta', r-1)$, for each neighbour Δ' of Δ, will be available when we come to evaluate $t(\Delta, r)$.*

Proof
Consider, without loss of generality, $\Delta = \langle i, j, k \rangle$ with $i \geq j \geq k$. If the t values are calculated according to the scheme described, then $t(\langle i, j, k, \rangle, r)$ is evaluated during iteration $p = \frac{r+2i-j-k}{3}$. Furthermore, $t(\langle i, j-1, k \rangle, r)$ and $t(\langle i, j, k-1 \rangle, r)$ are also evaluated during iteration p, and $t(\langle i-1, j, k \rangle, r)$ is evaluated during iteration $p-1$ or p according as $i > j$ or $i = j$. Since, during iteration p, any $t(., r-1)$ is evaluated before any $t(., r)$, it follows that $t(\Delta', r-1)$ is evaluated before $t(\langle i, j, k \rangle, r)$ for every neighbour Δ' of $\langle i, j, k \rangle$. □

The algorithm is shown in pseudo-code in Figure 2.1. The evaluation of the value $t(\Delta, 3p-x)$ is carried out using Lemma 2.2.6, following a snake, as in Figure 2.2 for diagonal $\langle i, j, k \rangle$.

The relevant initialisation for the p^{th} iteration involves those diagonals not considered in the $(p-1)^{th}$ iteration, and these turn out to be the diagonals in the set $\ll p, r, 0 \gg$ for $p \geq r \geq 0$. We initialise t for such a diagonal so that the (imaginary) last occurrence of the value $p+r-3$ is in position $p+r-3$ — see Figure 2.3.

2.2.1 Recovering an LCS

Recovering an LCS involves a (conceptual) trace-back through the n-cube from cell (n, n, n) to cell $(0, 0, 0)$ under the control of the t values. At each step, we are in a cell whose D-value became known during the execution of the algorithm. We step back one position in the current diagonal if that cell is in a snake, and otherwise step into a neighbouring cell whose D-value (which can be discovered from the t array) is exactly one smaller. In the former case, we will have found one more character

```
LCS_Lazy
p:=-1
repeat
   p:=p+1
   Initialise t values for the p^th iteration
   for x:=2p downto 0 do
      for y:=max(0, x-p) to ⌊x/2⌋ do
         for Δ ∈≪ x-y, x-2y, 0 ≫ do
            evaluate t(Δ, 3p − x)
until t(⟨0,0,0⟩, 3p) = n
```

Figure 2.1: The lazy algorithm for the length of the LCS of 3 strings

```
u:=1 + maxt(Δ',3p-x-1) over neighbours Δ' of ⟨ i, j, k ⟩
v:= (u - i - j - k) div 3
while (α1[i+v+1] = α2[j+v+1] = α3[k+v+1]) do {Match position}
   v:=v+1 {Assuming sentinels}
t(⟨ i, j, k ⟩, 3p-x) := i + j + k + 3v
```

Figure 2.2: Evaluation of $t(\Delta, 3p - x)$ for $\Delta = \langle i, j, k \rangle$

```
for r:=0 to p do
   for ⟨i,j,k⟩ ∈≪ p, r, 0 ≫ do
      t(⟨i,j,k⟩, p+r−3) := p+r−3
```

Figure 2.3: Initialisation for the p^{th} iteration of the lazy algorithm

```
LCS_Lazy: Recover the LCS
{ Throughout, Δ represents diagonal ⟨i,j,k⟩ }
i,j,k:=n
d:=3p
m:=n - p
repeat
   if α1[i]=α2[j]=α3[k] then
      LCS[m]:= α1[i]
      m:=m-1
      decrement each of i,j,k
   else
      find neighbour Δ' of Δ such that t(Δ', d-1) = i + j + k - 1
      d:=d-1
      decrement i, j or k according as Δ' differs from Δ
         in its 1st, 2nd or 3rd coordinate
until i=0
```

Figure 2.4: Recovering an LCS from the t table

in the LCS, and as this backward trace unfolds, the LCS will be revealed in reverse order.

The algorithm is shown in pseudo-code in Figure 2.4.

2.2.2 Analysis

We now consider the time and space requirements of the algorithm. As far as time is concerned, it is not hard to see that the worst-case complexity is dependent on the total number of elements of the t table that are evaluated. In iteration p, the diagonals for which a t value is calculated for the first time are precisely the diagonals in the sets $\ll p, r, 0 \gg$ for $r = 0, \ldots, p$. There are $6p$ such diagonals in total, since $| \ll p, r, 0 \gg | = 3$ for $r = 0$ or $r = p$, and $| \ll p, r, 0 \gg | = 6$ for $1 \leq r \leq p-1$. The number of iterations is the value of p for which $D(\alpha_1, \alpha_2, \alpha_3) = 3p$, and by Lemma 2.2.1, this is exactly equal to $n - l$, where $l = L(\alpha_1, \alpha_2, \alpha_3)$.

So, the total number of diagonals involved during the execution of the algorithm is

$$\sum_{p=0}^{n-l} 6p = 3(n-l)(n-l+1) \ .$$

Furthermore, the $6p$ diagonals that are started at the p^{th} iteration each contains at most $n - p$ entries in the n-cube, so the total number of t elements evaluated cannot

exceed

$$\sum_{p=0}^{n-l} 6p(n-p) = (n-l)(n-l+1)(n+2l-1). \tag{2.3}$$

As a consequence, the worst-case complexity of the algorithm is $O(n(n-l)^2)$, showing that we can expect it to be much faster than the naive dynamic programming algorithm in cases where the LCS has length close to n.

Diagonal $\langle i, j, k \rangle$ may be uniquely identified by the ordered pair $(i-j, j-k)$ (say), and such a representation leads in an obvious way to an array based implementation of the algorithm using $O(n^3)$ space. But appropriate use of dynamic linked structures enables just one node to be created and used for each t value calculated, and so by Equation 2.3, this yields an implementation that uses both $O(n(n-l)^2)$ time and space in the worst case. A little more care is required, in that case, in recovering the LCS from the linked lists of t values.

As with standard algorithms for the LCS of two strings, the space requirement can be reduced to $O((n-l)^2)$ if only the length of the LCS is required — in that case, only the most recent t value in each diagonal need be retained.

2.2.3 Extension to > 3 Strings

The lazy algorithm may be extended in a natural way to find the LCS of a set of k strings for any $k \geq 3$. Such an extended version can be implemented to use $O(kn(n-l)^{k-1})$ time and space in the worst case, the factor of k arising from the need to find the smallest of k values in the innermost loop.

2.3 A Threshold Based Algorithm

We extend the evaluation strategy of the threshold algorithm of Irving and Fraser, described in Section 1.3.2, to find the LCS of three strings. In the case of 3 strings α_1, α_2 and α_3 of length n, we define the *threshold set* $\tau_{i,m}$ to be the set of ordered pairs (j,k) such that α_1^i, α_2^j and α_3^k have a common subsequence of length m, but neither α_1^i, α_2^j and α_3^{k-1} nor α_1^i, α_2^{j-1} and α_3^k have such a common subsequence.

For example, for the sequences

$$\alpha_1 = abacbcabbcac \quad \alpha_2 = bbcabcbaabcb \quad \alpha_3 = cabcacbbcaba$$

we have $\tau_{5,2} = \{(2,7),(3,4),(5,3)\}$, corresponding to the subsequences *bb*, *bc*, and *ab* or *cb* respectively.

If (x,y), (x',y') are distinct ordered pairs of non-negative integers, we say that $(x,y) \leq (x',y')$ if $x \leq x'$ and $y \leq y'$. Let S be a set of ordered pairs of non-negative integers. The set S is therefore a partial order. The set $\hat{S}$ is the set of all minimal

elements of S. The set $\hat{S}$ is therefore an antichain of the partial order. The set $\hat{S}$ can be found from S by successively removing from S any pair that is greater than another pair in S. If the pairs in S are arranged in a list so that (x, y) precedes (x', y') in the list whenever $x \leq x'$, then a similarly ordered list representing $\hat{S}$ can be obtained by a single scan of the original list, comparing successive neighbours on the list and deleting any pair found to be greater than its immediate predecessor. So deriving $\hat{S}$ from S can be done in time linear in the size of S.

It is immediate from the definitions that $\nexists (x, y), (x', y') \in \tau_{i,m}$ such that $(x, y) \leq (x', y')$ i.e. the $\tau_{i,m}$ sets are antichains in the partial order. The length of a longest common subsequence of α_1, α_2 and α_3 is the largest value of m for which $\tau_{n,m}$ is non-empty. Hence, evaluation of the threshold sets $\tau_{i,m}$ in some appropriate order will enable us to determine the length of the LCS, and by suitable back tracing, to find an actual LCS of the 3 strings.

We shall now describe an algorithm for the LCS of 3 strings based on the evaluation of the threshold sets $\tau_{i,m}$ in diagonal order — i.e., in general we evaluate $\tau_{i,m}$ immediately after $\tau_{i-1,m-1}$. To explain the algorithm we need some additional terminology.

For any character α in the string alphabet, and any position i $(0 \leq i \leq n)$, we define the *next-occurrence* table $next_{\alpha_2}$ for string α_2 by

$$next_{\alpha_2}(\sigma, i) = \begin{cases} \min j : j > i \text{ and } \alpha_2[j] = \sigma & \text{if such a } j \text{ exists} \\ \infty & \text{otherwise} \end{cases}$$

So, as a special case, $next_{\alpha_2}(\sigma, 0)$ is the position of the first occurrence in string α_2 of character σ. The next-occurrence table $next_{\alpha_3}$ for string α_3 is defined analogously.

For any positions i and j in strings α_1 and α_2 respectively, we define $b_{i,j}$ to be the first position after j in α_2 that is occupied by character $\alpha_1[i]$, i.e.,

$$b_{i,j} = next_{\alpha_2}[\alpha_1[i], j] \ .$$

Similarly, we define

$$c_{i,j} = next_{\alpha_3}[\alpha_1[i], j] \ .$$

The basis of our algorithm is the following Lemma.

Lemma 2.3.1 $\tau_{i,m} = \hat{S}$ *where* $S = \tau_{i-1,m} \cup \bigcup_{(j,k) \in \tau_{i-1,m-1}} (b_{i,j}, c_{i,k})$.

Proof
Let $(j, k) \in \tau_{i,m}$, so that α_1^i, α_2^j and α_3^k have a common subsequence of length m but neither α_1^i, α_2^{j-1}, α_3^k nor α_1^i, α_2^j, α_3^{k-1} have such a common subsequence.
If α_1^{i-1}, α_2^j, α_3^k have a common subsequence of length m, then $(j, k) \in \tau_{i,m-1}$. Otherwise any common subsequence of α_1^i, α_2^j, α_3^k of length m contains $\alpha_1[i] = \alpha_2[j] =$

$\alpha_3[k]$ as its last element. So there is some $(j', k') \in \tau_{i-1,m-1}$ such that $b_{i,j'} = j$, $c_{i,k'} = k$. As a consequence, any element of $\tau_{i,m}$ is in the set $\sigma_{i,m}$ defined in the lemma, and since $\tau_{i,m}$ is an antichain, it is included in $\hat{\sigma}_{i,m}$.

On the other hand, any member of $\tau_{i-1,m}$ and any pair $(b_{i,j'}, c_{i,k'})$ for $(j', k') \in \tau_{i-1,m}$ is bound to be greater than a member of $\tau_{i,m}$, so that there are no elements in $\hat{\sigma}_{i,m}$ that are not in $\tau_{i,m}$. □

Our algorithm begins the evaluation of the sets $\tau_{i,m}$ on the main diagonal — i.e., it uses the above lemma to evaluate $\tau_{1,1}$, $\tau_{2,2}, \ldots$ until $\tau_{i,i} = \emptyset$, this last condition indicating that the i^{th} prefix of α_1 is not a common subsequence of α_1, α_2 and α_3. Next comes the evaluation of $\tau_{2,1}$, $\tau_{3,2}, \ldots$, again continuing until the first empty set is reached.

The process continues until $\tau_{n,m}$ is evaluated for some m. At that point the algorithm terminates, for no $\tau_{n,m'}$ can be non-empty for any $m' > m$. If $\tau_{n,m} \neq \emptyset$ then m is the length of the LCS of α_1, α_2 and α_3, otherwise its length is $m - 1$, since $\tau_{n-1,m-1}$ must have been non-empty. To enable the recurrence scheme to work correctly, at the beginning of the i^{th} iteration $\tau_{i,0}$ is initialised to $(0, 0)$, and whenever an unevaluated set $\tau_{i,m}$ is required (in the diagonal above the current one) it is initialised to the empty set.

The algorithm is shown in pseudo-code in Figure 2.5.

2.3.1 Recovering an LCS

If the algorithm terminates with $\tau_{n,m}$ non-empty then we start the trace-back at an arbitrary pair (j, k) in $\tau_{n,m}$, otherwise at an arbitrary pair (j, k) in $\tau_{n-1,m-1}$. In either case, $\alpha_2[j] = \alpha_3[k]$ is the last character of our LCS. At each stage, when we have reached $\tau_{i,m}$ say, we repeatedly decrement i until the current pair (j, k) is not a member of $\tau_{i-1,m}$. This indicates that, when constructing the τ table, we had $\alpha_1[i] = \alpha_2[j] = \alpha_3[k]$, and (j, k) must have arisen from an element (j', k') in $\tau_{i-1,m-1}$ that is less than it — so we locate such a pair, and record $\alpha_1[i]$ as an element in the LCS. We then decrement i and m, and the pair (j', k') becomes the new (j, k). The algorithm is shown in pseudo-code in Figure Figure 2.6.

2.3.2 Analysis

As described, the algorithm requires the pre-computation of the $next_{\alpha_2}$ and $next_{\alpha_3}$ tables, each requiring $\Theta(zn)$ steps, where z is the alphabet size.

The crux of the main part of the algorithm is the evaluation of $\tau_{i,m}$ from $\tau_{i-1,m-1}$ and $\tau_{i-1,m}$. This can be achieved in $O(|\tau_{i-1,m}| + |\tau_{i-1,m-1}|)$ time by maintaining each set as a linked list with (j, k) preceding (j', k') in the list if and only if $j < j'$. For then the lists may easily be merged in that time bound to form a list representing

```
LCS_Thresh; Build the array of τ sets
   max:=0 {length of LCS found so far}
   d:=-1 { d is the current diagonal }
   repeat
      d:=d+1
      i:=d { i is the row number in the τ table }
      τ_{i,0}:={(0, 0)}
      m:=0 { m is the column number in the τ table }
      repeat
         i:=i + 1
         m:=m+1
         if m > max then
            τ_{i-1,m}:=∅ { further initialisation }
         S:=τ_{i-1,m} ∪ ⋃_{(j,k)∈τ_{i-1,m-1}} (b_{i,j}, c_{i,k})
         τ_{i,m}:=Ŝ
      until (τ_{i,m} = ∅) or (i=n)
      if m-1>max then
         max:=m-1
   until i=n
   if τ_{n,m} = ∅ then l:=m-1
   else l:=m
   end
```

Figure 2.5: The threshold algorithm for the length of the LCS of 3 strings

```
   LCS_Thresh; Recover the LCS
   choose (j,k) ∈ τ_{n,l}
   i:=n
   r:=l
   LCS:=the empty string
   while r>0 do
      while (j,k) ∈ τ_{i-1,r} do
         i:=i-1
      LCS:=α1[i] ++ LCS  {α1[i] = α2[j] = α3[k] in LCS}
{ The symbol '++' denotes concatenation.}
      i:=i-1
      r:=r-1
      choose (j',k') ∈ τ_{i,r} such that (j',k') ≤ (j,k)
      (j,k):=(j',k')
   end
```

Figure 2.6: Recovering an LCS from the threshold table

$\sigma_{i,m}$, and as observed earlier, the list representing $\tau_{i,m} = \hat{\sigma}_{i,m}$ may be generated by a further single traversal of that list, deleting zero or more elements in the process.

It follows that the total number of operations involved in the forward pass of the algorithm is bounded by a constant times the number of pairs summed over all the sets $\tau_{i,m}$ evaluated. The backward pass involves tracing a single path through the τ table, examining a subset of the entries generated during the forward pass, and hence the overall complexity is dominated by the forward pass.

A trivial bound on the number of pairs in $\tau_{i,m}$ is $n - m + 1$, since the first component of each pair must be unique and in the range $m, \ldots, n$. Hence, since the number of sets $\tau_{i,m}$ evaluated is bounded by $l(n - l + 1)$ — at most l sets in each of $n - l + 1$ diagonals — this leads to a trivial worst-case complexity bound of $O(l(n-l)(n-\frac{l}{2}))$. In fact, it seems quite unlikely that all, or even many, of the sets $\tau_{i,m}$ can be large simultaneously, and we might hope for a worst-case bound of, say, $O(l(n-l)^2)$. But it seems to be quite difficult to establish such a bound for the algorithm as it stands. To realise this bound, we adapt the algorithm to a rather different form, with the aid of a trick also used by Apostolico et al. [5] in a slightly different context.

The crucial observation is that, if a pair (j, k) in set $\tau_{i,m}$ is to be part of an LCS of length l, then we must have $0 \leq j - m \leq n - l$ and $0 \leq k - m \leq n - l$, since there cannot be more than $n - l$ elements in either string α_2 or string α_3 that are not part of the LCS. So, if we knew the value of l in advance, we could immediately discard from each set $\tau_{i,m}$ any pair (j, k) for which $j > n - l + m$ or $k > n - l + m$. Hence each set would have effective size $\leq n - l + 1$, and since the number of sets is $O(l(n-l))$, this would lead to an algorithm with $O(l(n-l)^2)$ worst-case complexity.

Of course, the problem with this scheme is that we do not know the value of l in advance — it is precisely this value that our algorithm is seeking to determine. However, suppose we parameterise the forward pass of our algorithm with a bound p, meaning that we are going to test the hypothesis that the length of the LCS is $\geq n - p$. In that case, we need evaluate only $p + 1$ diagonals (at most), each of length $\leq l$, and in each set $\tau_{i,m}$ we need retain at most $p+1$ pairs. So this algorithm requires $O(lp^2)$ time in the worst case.

Suppose that we now apply the parameterised algorithm successively with $p = 0, 1, 2, 4, \ldots$ until it returns a "yes" answer. At that point we will know the length of the LCS, and we will have enough of the threshold table to reconstruct a particular LCS. As far as time complexity is concerned, suppose that $2^{t-1} < n - l \leq 2^t$. Then the algorithm will be invoked $t + 1$ times, with final parameter 2^t. So the overall worst-case complexity of the resulting LCS algorithm is $O(l \sum_{i=0}^{t} (2^i)^2) = O(l 2^{2t+2}) = O(l(n-l)^2)$, as claimed.

As it turns out, the above worst-case analysis of the threshold algorithm does not depend on the fact that the sets $\tau_{i,m}$ are antichains, merely on the fact that if

$(j,k),(j',k') \in \tau_{i,m}$ then $j \neq j'$ and $k \neq k'$ (though the maintenance of the sets as antichains, by the method described earlier, will undoubtedly speed up the algorithm in practice.)

As described above, the space complexity of the algorithm is also $O(l(n-l)^2 + zn)$, the first term arising from the bound on the number of elements summed over all the $\tau_{i,m}$ sets evaluated, and the second term from the requirements of the next-occurrence tables. The first term can be reduced somewhat by changing the implementation so that, for each pair (j,k) that belongs to some $\tau_{i,m}$, a node is created with a pointer to its "predecessor". In each column of the τ table, a linked list is maintained of the pairs in the most recent position in that column. By this means, each pair generated uses only a single unit of space, so the overall space complexity is big oh of the number of (different) pairs generated in each column, summed over the columns. An LCS can be re-constructed by following the predecessor pointers.

A Time-Space Tradeoff

As described by Apostolico and Guerra [6], the next-occurrence tables can be reduced to size $1 \times n$ rather than of size $z \times n$, for an alphabet of size z, at the cost of an extra $\log z$ factor in the time complexity — each lookup in the one-dimensional next-occurrence table takes $\log z$ time rather than constant time. In fact, for most problem instances over small or medium sized alphabets, the size of the next-occurrence tables is likely to be less significant than the space needed by the threshold table.

2.3.3 Extension to > 3 Strings

The idea of the threshold algorithm can be extended to find an LCS of k strings for any $k \geq 4$, using sets of $(k-1)$-tuples rather than sets of pairs. In the general case, it is less clear how the sets $\tau_{i,m}$ can be efficiently maintained as antichains, but as observed above for the case of three strings, the worst-case complexity argument does not require this property. In general, the extended version of the algorithm has worst-case time and space complexity that is $O(kl(n-l)^{k-1} + kzn)$, with the same time-space trade-off option available as before.

2.4 Empirical results and conclusions

To obtain empirical evidence as to the relative merits of the various algorithms, we implemented, initially for 3 strings, the basic dynamic programming algorithm (DP), the lazy algorithm (Lazy), the diagonal threshold algorithm (Thresh), and the algorithm of Hsu and Du (HD) [28]. In the case of the lazy algorithm, we used

Algorithm		n = 100	200	400	800	1600	3200	6400
DP		2.5	19.2	-	-	-	-	-
Lazy		0.05	0.18	1.5	11.9	-	-	-
Thresh	$z = 4$	0.03	0.1	0.4	2.3	21.7	136.2	-
	$z = 8$	0.03	0.1	0.3	1.7	8.8	54.8	-
	$z = 16$	0.02	0.1	0.3	1.3	7.1	33.6	212.5
HD	$z = 4$	0.9	10.5	-	-	-	-	-
	$z = 8$	0.4	5.0	73.4	-	-	-	-
	$z = 16$	0.2	2.8	36.7	-	-	-	-

Table 2.1: CPU times in seconds when LCS is 90% of string length

Algorithm		n = 100	200	400	800	1600	3200
DP		2.3	18.9	-	-	-	-
Lazy		3.0	23.1	-	-	-	-
Thresh	$z = 4$	0.2	1.5	11.7	107.8	-	-
	$z = 8$	0.1	0.7	5.5	48.3	-	-
	$z = 16$	0.05	0.4	2.5	18.8	184.0	-
HD	$z = 4$	0.8	9.7	-	-	-	-
	$z = 8$	0.3	4.5	59.6	-	-	-
	$z = 16$	0.2	2.3	31.8	-	-	-

Table 2.2: CPU times in seconds when LCS is 50% of string length

Algorithm		n = 100	200	400	800	1600	3200
DP		2.4	18.0	-	-	-	-
Lazy		17.0	-	-	-	-	-
Thresh	$z = 4$	0.02	0.4	2.6	17.1	142.2	-
	$z = 8$	0.02	0.3	2.3	15.3	128.5	-
	$z = 16$	0.02	0.2	1.5	10.6	117.2	-
HD	$z = 4$	0.06	0.5	-	-	-	-
	$z = 8$	0.06	0.4	9.7	-	-	-
	$z = 16$	0.06	0.3	5.0	-	-	-

Table 2.3: CPU times in seconds when LCS is 10% of string length

a dynamic linked structure, as discussed earlier, to reduce the space requirement. We implemented both the straightforward threshold algorithm of Figures 2.5 and 2.6, and the version with the guaranteed $O(l(n-l)^2 + zn)$ complexity, and found that the former was consistently between 1.5 and 2 times faster in practice — so we have included the figures for the faster version. The Hsu-Du algorithm was implemented using both a 3-dimensional array and a dynamic structure to store match nodes. There was little difference between the two in terms of time, but the latter version allowed larger problem instances to be solved, so we quote the results for that version.

The algorithms were coded in Pascal, compiled and run under the optimised Sun Pascal compiler on a Sun 4/25 with 8 megabytes of memory. We used alphabet sizes of 4, 8 and 16, string lengths of 100, 200, 400, ... (for simplicity, we took all 3 strings to be of the same length), and we generated sets of strings with an LCS of respectively 90%, 50% and 10% of the string length in each case. In all cases, times were averaged over 3 sets of strings.

The results of the experiments are shown in the tables, where the CPU times are given in seconds. Times for DP and Lazy were essentially independent of alphabet size, so only one set of figures is included for each of these algorithms. In every case, we ran the algorithms for strings of length 100, and repeatedly doubled the string length until the program failed for lack of memory, as indicated by "-" in the table.

Conclusions.

As far as comparisons between the various algorithms are concerned, the threshold algorithm appears to be the best across the whole range of problem instances generated. The lazy algorithm is competitive when the LCS is close to the string length, and the Hsu-Du algorithm only when the LCS is very short.

However, the clearest conclusion that can be drawn from the empirical results is that, at least using the known algorithms, the size of LCS problem instances with three strings that can be solved in practice is constrained by space requirements rather than by time requirements. Preliminary experiments with versions of the threshold and Hsu-Du algorithms for more than three strings confirm the marked superiority of the former, and show that the dominance of memory constraint is likely to be even more pronounced for larger numbers of strings. So there is clearly a need for algorithms that use less space. One obvious approach worthy of further investigation is the application of the space-saving divide-and-conquer technique [25], and other possible time-space trade-offs, particularly, in view of the evidence, to the threshold algorithm.

Chapter 3

Exact Algorithms for the SCS problem

3.1 Introduction

As was noted in Section 1.4.2, the Shortest Common Supersequence problem for k strings of length n can be solved by dynamic programming in $O(kn^k)$ time and $O(n^k)$ space. The space complexity of dynamic programming can be improved to $O(n^{k-1})$ using the technique developed by Hirschberg[25]. In this chapter we look at two alternative algorithms intended to provide, in particular circumstances, improvements on the time and space required by the dynamic programming algorithm. The two algorithms are presented in the context of finding the SCS of three strings $(\alpha_1, \alpha_2, \alpha_3)$. However, they can both be extended to work for any fixed number of strings and at the end of each section, this is explained. Throughout this chapter, s represents the length of the SCS.

3.2 An algorithm operating on matches

The first algorithm is called "MML_Thresh" (for a reason which will become clear shortly), and is an analogy of the Threshold Algorithm of Chapter 2 for the Longest Common Subsequence problem in that it operates on the matches between symbols of increasing prefixes of the three strings.

In a fixed embedding of the three strings in an SCS, each position of the SCS corresponds to either a single symbol from one of the strings, a match between symbols in a pair of the strings or a match between symbols in all three of the strings. It is a trivial observation that the longest possible SCS consists of the three strings concatenated (or interleaved) together, and this occurs when there are no elements common to any two of the strings. Positions in an SCS are assigned values 0,1, and 2, depending on whether they correspond to symbols in 1, 2, or 3 strings respectively

in the fixed embedding. Thus the value assigned to a position represents the saving achieved, by the symbol in that position, over the longest possible SCS. Positions with a value of 1 represent a match between two of the strings and positions with a value of 2 represent a match between all three strings. The matches represented by these positions form a list of matches over α_1, α_2, and α_3. Defining m to be the sum of the values of all the matches in a match list, the maximum match list (MML) is the list of matches that maximises m. Defining t to be the size of the MML, it is clear that $s = 3n - t$. A supersequence can be formed from a list of matches by forming a string from the symbols of the matches respectively and inserting the unmatched symbols from the three strings appropriately.

We define the threshold set $\lambda_{i,m}$ to be the set of pairs (j,k), ordered by increasing j, such that the i^{th} prefix $\alpha_1^i = \alpha_1[1..i]$ of α_1, the j^{th} prefix of α_2 and the k^{th} prefix of α_3 have an MML of size m but neither α_1^i, α_2^{j-1}, and α_3^k nor α_1^i, α_2^j, and α_3^{k-1} have an MML of size as large as m.

For example for the strings

$$\alpha_1 = abca \qquad \alpha_2 = abcb \qquad \alpha_3 = cbac$$

the set $\lambda_{1,2} = \{(1,3),(2,2),(3,1)\}$, corresponding respectively to the alignments

	c	b	a		a	c	b		a	b	c
α_1 :	$-$	$-$	a		a	$-$	$-$		a	$-$	$-$
α_2 :	$-$	$-$	a		a	$-$	b		a	b	c
α_3 :	c	b	a		$-$	c	b		$-$	$-$	c
	0	0	2		1	0	1		1	0	1

where the numbers show the values assigned to the positions of the supersequences.

Recall the following definitions given in Section 2.3. If $(x,y),(x',y')$ are distinct ordered pairs of non-negative integers, we say that $(x,y) \leq (x',y')$ if $x \leq x'$ and $y \leq y'$. Let S be a set of ordered pairs of non-negative integers. The set S is therefore a partial order. The set $\hat{S}$ is the set of all minimal elements of S and is an antichain of the partial order. Section 2.3 shows how $\hat{S}$ can be derived from S in time linear in the size of S.

It is immediate from the definitions that there are no two pairs $(x,y),(x',y')$ in the set $\pi_{i,l}$ such that $(x,y) \leq (x',y')$. In other words, the $\lambda_{i,m}$ sets are antichains of a partial order. The largest m for which $\lambda_{n,m}$ is not empty gives the size t of the MML of $\alpha_1, \alpha_2, \alpha_3$, from which the length of the SCS can be calculated. The table of threshold sets will have the following structure:

		m					
		0	1	2	3	...	t
	0	$\{(0,0)\}$	$\lambda_{0,1}$	$\lambda_{0,2}$	$\lambda_{0,3}$	...	$\lambda_{0,t}$
	1	$\{(0,0)\}$	$\lambda_{1,1}$	$\lambda_{1,2}$	$\lambda_{1,3}$	...	$\lambda_{1,t}$
i	2	$\{(0,0)\}$	$\lambda_{2,1}$	$\lambda_{2,2}$	$\lambda_{2,3}$	...	$\lambda_{2,t}$
	:	:	:	:	:	:	:
	n	$\{(0,0)\}$	$\lambda_{n,1}$	$\lambda_{n,2}$	$\lambda_{n,3}$	...	$\lambda_{n,t}$

We will set up a scheme in which the evaluation of set $\lambda_{i,m}$ requires the sets $\lambda_{i-1,m-2}$, $\lambda_{i-1,m-1}$, $\lambda_{i-1,m}$ and $\lambda_{i,m-1}$ and no other sets. This leads to calculation of the sets row by row (increasing i) from left to right (increasing m). Some further definitions will be helpful in establishing the scheme.

$$next_\alpha(i,\sigma) = \begin{cases} \min\{j : \alpha[j] = \sigma, j > i\} & \text{if such a } j \text{ exists} \\ \infty & \text{otherwise.} \end{cases}$$

The function *next* was used in the threshold algorithm for the LCS problem in Section 2.3 of Chapter 2.

$\text{match2}((j,k),\sigma) = \{(next_{\alpha_2}(j,\sigma), next_{\alpha_3}(k,\sigma))\}$
$\text{MATCH2}(\lambda,\sigma) = \hat{S}$ where $S = \bigcup_{(j,k)\in\lambda} \text{match2}((j,k),\sigma)$

The function *MATCH2* will be used to generate matches between α_1 and both α_2 and α_3.

$\text{match1}((j,k),\sigma) = \{(j, next_{\alpha_3}(k,\sigma)), (next_{\alpha_2}(j,\sigma), k)\}$
$\text{MATCH1}(\lambda,\sigma) = \hat{S}$ where $S = \bigcup_{(j,k)\in\lambda} \text{match1}((j,k),\sigma)$

The function *MATCH1* will be used to generate matches between α_1 and each of α_2 and α_3 independently.

$\text{OTHER2}(\lambda) = \hat{S}$ where $S = \bigcup_{(j,k)\in\lambda} \bigcup_{\sigma\in\Sigma} \text{match2}((j,k),\sigma)$

The function *OTHER2* will be used to generate matches between α_2 and α_3 but not α_1. It operates in time $O(z|\lambda|)$ since each pair in λ will generate at most z new pairs.

The crux of the scheme is in Lemma 3.2.1 and the algorithm is shown in pseudo-code in Figure 3.1.

Lemma 3.2.1 $\lambda_{i,m} = \hat{S}$ *where*

$$S = MATCH2(\lambda_{i-1,m-2}, \alpha_1[i]) \cup MATCH1(\lambda_{i-1,m-1}, \alpha_1[i]) \cup \lambda_{i-1,m} \cup OTHER2(\lambda_{i,m-1})$$

Proof
There are two stages to this proof. The first stage shows how a pair in one of the sets

$\lambda_{i-1,m-2}, \lambda_{i-1,m-1}, \lambda_{i-1,m}$, and $\lambda_{i,m-1}$ may generate a pair in $\lambda_{i,m}$. The second stage shows how every pair in $\lambda_{i,m}$ must have a 'parent' in one of $\lambda_{i-1,m-2}, \lambda_{i-1,m-1}, \lambda_{i-1,m}$, or $\lambda_{i,m-1}$.

If pair $(j,k) \in \lambda_{i-1,m-2}$, and $next_{\alpha_2}(j, \alpha_1[i]) < \infty$ and $next_{\alpha_3}(k, \alpha_1[i]) < \infty$ then there is a match list with size m ending at $\alpha_1[i] = \alpha_2[next_{\alpha_2}(j, \alpha_1[i])] = \alpha_3[next_{\alpha_3}(k, \alpha_1[i])]$. If pair $(j,k) \in \lambda_{i-1,m-1}$, and $next_{\alpha_2}(j, \alpha_1[i]) < \infty$ then there is a match list with size m ending at $\alpha_1[i] = \alpha_2[next_{\alpha_2}(j, \alpha_1[i])]$. If pair $(j,k) \in \lambda_{i-1,m-1}$, and $next_{\alpha_3}(k, \alpha_1[i]) < \infty$ then there is a match list with size m ending at $\alpha_1[i] = \alpha_3[next_{\alpha_3}(k, \alpha_1[i])]$. If $\alpha_1^{i-1}, \alpha_2^j$, and α_3^k have an MML of size m then clearly α_1^i, α_2^j, and α_3^k have a match set of size m. If pair $(j,k) \in \lambda_{i,m-1}$, and for symbol σ, $next_{\alpha_2}(j, \sigma) < \infty$ and $next_{\alpha_3}(k, \sigma) < \infty$ then there is a match list with size m ending at $\alpha_2[next_{\alpha_2}(j, \sigma)] = \alpha_3[next_{\alpha_3}(k, \sigma)]$.

A pair $(j,k) \in \lambda_{i,m}$ represents a list of matches, the last of which is a match between symbols of α_1 and one of the other strings, or of α_1 and both of the other strings, or of α_2 and α_3. If the last match is between $\alpha_1[i]$ and symbols in both the other strings then the match list of size $m-2$ with that last match removed must be represented by a tuple $(j',k') \in \lambda_{i-1,m-2}$ such that $j' < j$ and $k' < k$. If the last match is between $\alpha_1[i]$ and a symbol of α_2 (or α_3) then the match list of size $m-1$ with that last match removed must be represented by a tuple (j',k) (or (j,k')) $\in \lambda_{i-1,m-1}$ such that $j' < j$ (or $k' < k$). If α_1^i, α_2^j, and α_3^k have an MML of size m and the last match does not involve α_1^i then clearly $\alpha_1^{i-1}, \alpha_2^j$, and α_3^k have an MML of size m. If the last match is between symbols of α_2 and α_3 then the match list of size $m-1$ with that last match removed must be represented by a tuple $(j',k') \in \lambda_{i,m-1}$ such that $j' < j$ and $k' < k$. □

3.2.1 Recovering an SCS

If appropriate extra information is stored, recovering the SCS involves a straightforward trace back through the threshold table. When a pair is created, it is linked via a pointer to its parent, i.e. the pair from which it was created. Where there is more than one parent, any one may be chosen. In this way, the matches in the MML will form a list, in reverse order, of the matches in an SCS, starting at any match in the set $\lambda_{n,t}$. Following this list back to $\lambda_{0,0}$ will trace the matches used in an MML. The elements from each string not included in any of the matches can be inserted at their appropriate places between the matches of the MML. The algorithm is shown in pseudo-code in Figure 3.2.

3.2.2 Analysis

The algorithm evaluates n rows each with at most $t+1$ sets. The only other factor is the time required to evaluate a particular set of pairs. The time required

MML_Thresh: Build the table of λ's
for row:=0 **to** n **do**
  col:=0;
  repeat
    col:=col+1
    $L_1, L_2, L_3, L_4 := \phi$
    if (row>0) **and** (col>1) **then**
      L_1:=MATCH2($\lambda_{row-1,col-2}, \alpha_1$[row])
    if (row>0) **and** (col>0) **then**
      L_2:=MATCH1($\lambda_{row-1,col-1}, \alpha_1$[row])
    if (row>0) **then**
      $L_3 := \lambda_{row-1,col}$
    if (col>0) **then**
      L_4:=OTHER2($\lambda_{row,col-1}$)
    S:=$L_1 \cup L_2 \cup L_3 \cup L_4 \cup \{(0,0)\}$
    $\lambda_{row,col} := \hat{S}$
  until $\lambda_{row,col} = \phi$
t:=col-1
end

Figure 3.1: Forward pass of MML_Thresh

MML_Thresh: Recover the SCS
i := n
p := Any pair in $\lambda_{n,t}$
SCS:=The empty string
while p<>(0,0) **do**
  p′:=Parent of p ($\in \lambda_{i',m}$)
{*Add the unmatched symbols of* α_1, α_2 *and* α_3}
  SCS:=α_3[p.second+1 ...k-1] $+\!\!+$ SCS
  SCS:=α_2[p.first+1 ...k-1] $+\!\!+$ SCS
  SCS:=α_1[i′+2 ...i] $+\!\!+$ SCS
{*Add the character of the current match.*}
  if p.first > p′.first **then**
    SCS:=α_2[j] $+\!\!+$ SCS
  else {*p.second* > *p′.second*}
    SCS:=α_3[k] $+\!\!+$ SCS
  p := p′
  i := i′
end

Figure 3.2: Recovering an SCS in MML_Thresh

to generate a set is proportional to the sum of the times required to evaluate the functions $MATCH1$, $MATCH2$, and $OTHER2$. The dominant factor there is the time to evaluate $OTHER2$ which is $O(n|\lambda|)$ where $|\lambda|$ is the maximum size of a set. An immediate bound on the number of pairs in a set is $n+1$. This gives a running time of $O(zn^2t)$. Since the space complexity depends directly on the number of pairs generated, it is $O(n^2t)$. When there are very few matches between the strings we can expect MML_Thresh to be better than Dynamic Programming.

This analysis does not depend on the λ sets being antichains although the maintenance of them as such will undoubtedly speed the algorithm up in practice.

3.2.3 Extension to >3 Strings

We can extend MML_Thresh to find the size of the MML and hence the length of the SCS of k strings $(\alpha_1, \alpha_2, \ldots, \alpha_k)$ for any fixed $k > 3$. However it is easy to see that the rather inelegant resulting algorithm is very inefficient for even modest values of k. Pairs are replaced by $(k-1)$-tuples. The evaluation of $\lambda_{i,m}$ requires the $2k-2$ previously evaluated sets: $\lambda_{i-1,m-k+1}, \lambda_{i-1,m-k+2}, \ldots, \lambda_{i-1,m}$ and $\lambda_{i,m-k+2}, \lambda_{i,m-k+3}, \ldots, \lambda_{i,m-1}$. The functions "match1", "MATCH1", "match2", and "MATCH2" are extended to "match" and "MATCH" which take an additional parameter a in the range $0 \ldots k-1$. The function "OTHER2" is extended to "OTHERS" which takes an additional parameter b in the range $2 \ldots k-1$. The new functions behave as follows,

$$\begin{aligned}
\text{match}(a,p,\sigma) &= \{ \text{ The } \tbinom{k-1}{a} \text{ tuples such that } next(\alpha,\sigma) \text{ is applied to} \\
&\quad \text{precisely } a \text{ components of } p.\} \\
\text{MATCH}(a,\lambda,\sigma) &= \hat{S} \text{ where } S = \textstyle\bigcup_{p\in\lambda} \text{match}(a,p,\sigma) \\
\\
\text{OTHERS}(b,\lambda) &= \hat{S} \text{ where } S = \textstyle\bigcup_{p\in\lambda} \bigcup_{\sigma\in\Sigma} \text{match}(b,p,\sigma)
\end{aligned}$$

where p is a tuple, σ is a character, and α represents any one of $\alpha_1 \ldots \alpha_k$. Evaluation of sets is based on the following lemma.

Lemma 3.2.2

$$\lambda_{i,m} = \hat{S} \text{ where } S = \bigcup_{a=0}^{k-1} MATCH(a, \lambda_{i-1,l-a}, \alpha_1[i]) \bigcup \bigcup_{b=2}^{k-1} OTHERS(b, \lambda_{i,l-b+1}).$$

Entries in the table of λ sets are calculated in the same order as for $k=3$. The time complexity of the algorithm is $O(2^k z n^{k-1} t)$ in the worst case. The 2^k factor arising from the execution of $MATCH(a)$ with a in the range $0 \ldots k-1$ and $OTHERS(b)$ with b in the range $2 \ldots k-1$. The space complexity is $O(n^{k-1}t)$ arising from the number of tuples stored in the array.

3.3 An algorithm operating on supersequences

The second algorithm is called "SCS_Thresh". It is dual to the Threshold Algorithm of Chapter 2 for the Longest Common Subsequence Problem in that it operates on the lengths of the SCS's (as opposed to LCS's) of increasing prefixes of the three strings.

We define the threshold set $\pi_{i,l}$ to be the set of pairs (j,k), ordered by increasing j, such that α_1^i, α_2^j and α_3^k have a shortest common supersequence of length l but neither $\alpha_1^i, \alpha_2^{j+1}$ and α_3^k nor α_1^i, α_2^j and α_3^{k+1} have such a common supersequence.

For example, for the strings

$$\alpha_1 = bcab \qquad \alpha_2 = cbca \qquad \alpha_3 = abbc$$

the set $\pi_{2,4} = \{(1,4), (3,2), (4,1)\}$, corresponding to the supersequences $abbc, acbc$ and $cbca$ respectively.

If (x,y) and (x',y') are distinct ordered pairs of non-negative integers, we say that $(x,y) \geq (x',y')$ if $x \geq x'$ and $y \geq y'$. Let S be a set of ordered pairs of non-negative integers. The set S is therefore a partial order. The set $\check{S}$, is the set of all maximal elements of S. Like $\hat{S}$, $\check{S}$ is an antichain of the partial order.

It is immediate from the definitions that there are no $(x,y),(x',y') \in \pi_{i,l}$ such that $(x,y) \geq (x',y')$, i.e., the $\pi_{i,l}$ sets are antichains. Any set containing the pair (n,n) will contain no other pairs, and therefore the length s of the shortest common supersequence is the smallest l for which $\pi_{n,l} = \{(n,n)\}$. The table of threshold sets $\pi_{i,l}$ has the following structure:

		l									
		0	1	2	3	...	n	...	$s-2$	$s-1$	s
	0	$\{(0,0)\}$	$\pi_{0,1}$	$\pi_{0,2}$	$\pi_{0,3}$	...	$\pi_{0,n}$	...	-	-	-
	1	ϕ	$\pi_{1,1}$	$\pi_{1,2}$	$\pi_{1,3}$	...	$\pi_{1,n}$	...	-	-	-
i	2	ϕ	ϕ	$\pi_{2,2}$	$\pi_{2,3}$	...	$\pi_{2,n}$	...	-	-	-
	:	:	:	:	:	:	:	:	:	:	:
	n	ϕ	ϕ	ϕ	ϕ	...	$\pi_{n,n}$	...	$\pi_{n,s-2}$	$\pi_{n,s-1}$	$\{(n,n)\}$

So we require a method to evaluate the threshold sets in some appropriate order leading to the required $\pi_{n,l}$ containing only the pair (n,n). We will set up a scheme in which the evaluation of $\pi_{i,l}$ requires the sets $\pi_{i,l-1}$ and $\pi_{i-1,l-1}$. Hence the evaluation will be in diagonal order, starting with the main diagonal $\pi_{0,0}$, $\pi_{1,1}$, $\pi_{2,2}$, ..., $\pi_{n,n}$ and then the adjacent diagonal $\pi_{0,1}$, $\pi_{1,2}$, $\pi_{2,3}$, ..., $\pi_{n,n+1}$ and so on until a set $\pi_{n,l}$ is evaluated which contains only the pair (n,n). Some further definitions will be helpful in establishing the scheme.

For a pair of non-negative integers (j,k), a character σ and a set of pairs of non-negative integers π, we define the functions

$$
\begin{aligned}
\text{extend}((j,k),\sigma) &= \{(j+1,k+1)\} && \textbf{if } \alpha_2[j+1] = \alpha_3[k+1] = \sigma \\
&= \{(j+1,k)\} && \textbf{if } \alpha_2[j+1] = \sigma \textbf{ and } \alpha_3[k+1] <> \sigma \\
&= \{(j,k+1)\} && \textbf{if } \alpha_2[j+1] \neq \sigma \textbf{ and } \alpha_3[k+1] = \sigma \\
&= \{(j,k)\} && \textbf{if } \alpha_2[j+1] \neq \sigma \textbf{ and } \alpha_3[k+1] <> \sigma \\
\text{EXTEND}(\pi,\sigma) &= \bigcup_{(j,k)\in\pi} \text{extend}((j,k),\sigma) &&
\end{aligned}
$$

$$
\begin{aligned}
\text{fork}((j,k)) &= \{(j+1,k+1)\} && \textbf{if } \alpha_2[j+1] = \alpha_3[k+1] \\
&= \{(j+1,k),(j,k+1)\} && \textbf{otherwise} \\
\text{FORK}(\pi) &= \bigcup_{(j,k)\in\pi} \text{fork}((j,k)). &&
\end{aligned}
$$

The crux of the scheme is in Lemma 3.3.1 and the algorithm is shown in pseudo-code in Figure 3.3.

Lemma 3.3.1 *$\pi_{i,l} = \check{S}$ where $S = EXTEND(\pi_{i-1,l-1}, \alpha_1[i]) \cup FORK(\pi_{i,l-1})$*

Proof

There are two stages to this proof. The first stage shows how a pair in one of the sets $\pi_{i-1,l-1}$ and $\pi_{i,l-1}$ may generate a pair in $\pi_{i,l}$. The second stage shows how every pair in $\pi_{i,l}$ must have a parent in either $\pi_{i-1,l-1}$ or $\pi_{i,l-1}$.

Every pair $(j,k) \in \pi_{i-1,l-1}$ represents a supersequence of $\alpha_1^{i-1}, \alpha_2^j$, and α_3^k of length $l-1$. If we append $\alpha_1[i]$ to that supersequence we will get a supersequence of $\alpha_1^i, \alpha_2^{j'}$, and $\alpha_3^{k'}$ of length l such that $j' = j+1$ if $\alpha_2[j+1] = \alpha_1[i]$, otherwise $j' = j$, and $k' = k+1$ if $\alpha_3[k+1] = \alpha_1[i]$, otherwise $k' = k$. Every pair $(j,k) \in \pi_{i,l-1}$ represents a supersequence of α_1^i, α_2^j and α_3^k of length $l-1$. If we append $\alpha_2[j+1]$ to that supersequence we will get a supersequence of $\alpha_1^i, \alpha_2^{j+1}$, and $\alpha_3^{k'}$ of length l such that $k' = k+1$ if $\alpha_3[k+1] = \alpha_2[j+1]$, otherwise $k' = k$. Every pair $(j,k) \in \pi_{i,l-1}$ represents a supersequence of α_1^i, α_2^j and α_3^k of length $l-1$. If we append $\alpha_3[k+1]$ to that supersequence we will get a supersequence of $\alpha_1^i, \alpha_2^{j'}$, and α_3^{k+1} of length l such that $j' = j+1$ if $\alpha_2[j+1] = \alpha_3[k+1]$, otherwise $j' = j$.

A pair $(j,k) \in \pi_{i,l}$ represents a supersequence γ of length l of α_1^i, α_2^j, and α_3^k. If $\gamma[l] = \alpha_1[i]$ then the supersequence with that last symbol removed must be represented by a pair $(j',k') \in \pi_{i-1,l-1}$. If $\alpha_2[j] = \gamma[l]$ then $j' = j-1$ otherwise $j' = j$. If $\alpha_3[k] = \gamma[l]$ then $k' = k-1$ otherwise $k' = k$. If $\gamma[l] \neq \alpha_1[i]$ then the supersequence with that last symbol removed must be represented by a pair $(j',k') \in \pi_{i,l-1}$. If $\alpha_2[j] = \gamma[l]$ then $j' = j-1$ otherwise $j' = j$. If $\alpha_3[k] = \gamma[l]$ then $k' = k-1$ otherwise $k' = k$. □

3.3.1 Recovering an SCS

When each pair is created, it is linked via a pointer to its parent. Hence the final pair (n,n) will be at the head of a list of $(s+1)$ pairs, ending with the pair $(0,0) \in \pi_{n,s}$. To recover an SCS this list is followed and for each pair except the

```
SCS_Thresh: Build the table of π's
r:=-1
repeat
   r:=r+1
   for d:=0 to n do
      L1,L2:=φ
      if d>0 then
         L1:=EXTEND(π_{d-1,r+d-1}, α1[d])
      if r>0 then
         L2:=FORK(π_{d,r+d-1})
      S:=L1∪ L2 ∪{(0,0)}
      π_{d,r+d}:=Š
until π_{d,r+d}={(n,n)}
s:=r+d
end
```

Figure 3.3: Forward pass of SCS_Thresh

last, the corresponding symbol is prepended to the SCS. The algorithm is shown in pseudo-code in Figure 3.4

3.3.2 Analysis

The table of threshold sets shows that each diagonal calculated has $(n+1)$ sets and that there are $(s-n+1)$ diagonals to evaluate. The time taken to evaluate the set $\pi_{i,l}$ is proportional to $|\pi_{i-1,l-1}| + |\pi_{i,l-1}|$ so the time to process each set is limited by the maximum size of a set. A slight modification to the running of the algorithm can ensure a bound of $(s-n)$ pairs in each set. This is the same 'trick' used to achieve a performance guarantee for the Threshold Algorithm for the LCS Problem in Chapter 2. We parameterise the algorithm with parameter p so that we are testing the hypothesis that $s \leq n+p$. That means we must evaluate at most $p+1$ diagonals with at most $p+1$ pairs in each set. This algorithm will therefore have time complexity $O(np^2)$. If we successively call the algorithm with parameter $p = 0, 1, 2, 4, 8, \ldots$ then when the answer "yes" is returned, we will know the length of the SCS and have enough information to reconstruct a particular SCS. Suppose that

$$n + 2^{u-1} < s \leq n + 2^u$$

i.e.

$$2^{u-1} < s - n \leq 2^u.$$

```
SCS_Thresh: Recover the SCS
p:=(n,n) ∈ π_{n,s}
i:=n
SCS:=The empty string
while p<>(0,0) do
   p':=Parent of p (∈ π_{i',l})
   if p.first > p'.first then
      SCS:=α_2[p.first] # SCS
   else
      if p.second > p'.second then
         SCS:=α_3[p.second] # SCS
      else
         SCS:=α_1[i] # SCS
   p:=p'
   i:=i'
end
```

Figure 3.4: Recovering an SCS in SCS_Thresh

The algorithm will be invoked $u+1$ times with the final parameter $p = 2^u$. The time complexity of the resulting algorithm is $O(n\Sigma_{i=0}^{2^u}(2^i)^2) = O(n2^{2u+2}) = O(n(s-n)^2)$. Since the space complexity depends directly on the number of pairs generated throughout all the sets, the space complexity is also $O(n(s-n)^2)$. We can therefore expect SCS_Thresh to be good when the SCS is short.

As for the analysis of MML_Thresh, the analysis for SCS_Thresh does not depend on the π sets being antichains although the maintenance of them as such will undoubtedly speed the algorithm up in practice.

3.3.3 Extension to >3 Strings

We can extend SCS_Thresh to find the SCS of k strings for any fixed $k > 3$. In this case, pairs are replaced by $(k-1)$-tuples. As for $k = 3$, $\pi_{i,l} = \check{S}$ where $S = \text{EXTEND}(\pi_{i-1,l-1}, \alpha_1[i]) \cup \text{FORK}(\pi_{i,l-1})$. The function extend$(p, \sigma)$ on tuple p and symbol σ expands easily so that in the new tuple generated, the i^{th} component will be one greater than c, the i^{th} component of p, if and only if $\alpha_i[c+1] = \sigma$, otherwise it will have the same value (c). As before the function EXTEND(π, σ) on set π applies extend(p, σ) to each tuple $p \in \pi$. Extending the function fork(p) to work in linear time is less straightforward. The tuple p generates between 1 and $(k-1)$ new tuples, depending on the number of groups of strings with distinct, matching characters in the positions following their corresponding components in p. This is best explained with an example.

For the strings

$$\alpha_1 = \mathbf{a}bcbc \qquad \alpha_2 = \mathbf{c}abab \qquad \alpha_3 = a\mathbf{c}bac$$

$$\alpha_4 = \mathbf{c}acab \qquad \alpha_5 = c\mathbf{b}abc \qquad \alpha_6 = cabcb$$

where $p = (i_1, i_2, i_3, i_4, i_5, i_6) = (1,1,2,1,2,0)$, the function fork($p$) generates three new tuples – (1,**2**,2,**2**,**3**,0), (**2**,1,**3**,1,2,0), and (1,1,2,1,2,**1**). The first tuple arises because $\alpha_2[i_2 + 1] = \alpha_4[i_4 + 1] = \alpha_5[i_5 + 1] =$'$a$'. The second tuple arises because $\alpha_1[i_1 + 1] = \alpha_3[i_3 + 1] =$'$b$'. The third tuple arises because α_6 alone has 'c' in position $i_6 + 1$. One pass of the components of p is required to generate this list of tuples. For the i^{th} component $p.i = c$, the character $\alpha_i[c]$ is checked to see if it has been seen already in this pass. If not, then a new tuple associated with $\alpha_i[c]$ is created. The new tuple p' is a copy of p except that $p'.i = c + 1$. If $\alpha_i[c]$ has been seen before then the i^{th} component of the tuple associated with that character is incremented.

Using these expansions, the threshold table is evaluated in the same order as before, until the tuple $(n, n, n, ..)$ is reached in the bottom row. By parameterising the algorithm as for three strings, the resulting worst case time and space complexity is $O(kn(s-n)^{k-1})$. When the shortest common supersequence is short, this algorithm can be expected to be much better than basic Dynamic Programming.

3.4 Empirical results and conclusions

To compare the performances of the two new algorithms to that of Dynamic Programming we implemented, for three strings, the basic Dynamic Programming Algorithm, algorithm MML_Thresh, and algorithm SCS_Thresh. As for the threshold algorithm for the LCS problem, the straightforward version of SCS_Thresh was used (and not the version with the parameterisation trick to ensure the worst-case time complexity). They were coded in Pascal and compiled with the Sun Pascal Compiler version 1.0 using the "-O" option to provide the widest range of optimisations. They were run on a SPARCstation-10 with sixteen megabytes of memory.

Alphabet sizes of 4, 8, 16, and, where appropriate, 128 and 256 were used. String lengths of 100, 200, 400, ... were used. Two sets of experiments were performed. The first used completely random strings. That is, every symbol was selected uniformly at random from the alphabet. The second set used strings with an SCS of length less than or equal to $11n/10$. They were generated by taking random subsequences, with length n, of a string of length $11n/10$. Similar strings were chosen because it is when strings are similar that their relationship is likely to be interesting in practical applications. The CPU times in seconds for the running times of the algorithms are shown in Tables 3.1 and 3.2. The times for the DP algorithm were independent of

Algorithm		n = 100	200	400	800
DP		1.7	13	-	-
MMS_Thresh	$z = 4$	13	-	-	-
	$z = 8$	8.7	74	-	-
	$z = 16$	6	46	-	-
	$z = 128$	0.9	8.8	80	-
	$z = 256$	0.4	4.4	42	-
SCS_Thresh	$z = 4$	2.8	22	-	-
	$z = 8$	6.5	50	-	-
	$z = 16$	11	-	-	-

Table 3.1: CPU times in seconds for SCS algorithms on random strings

Algorithm		n = 100	200	400	800	1600
DP		1.6	13	-	-	-
MMS_Thresh	$z = 4$	14	-	-	-	-
	$z = 8$	12	93	-	-	-
	$z = 16$	10	86	-	-	-
SCS_Thresh	$z = 4$	0.1	0.6	3.7	32	-
	$z = 8$	0.1	0.5	3.5	28	-
	$z = 16$	0.1	0.4	3.0	23	-

Table 3.2: CPU times in seconds for SCS algorithms when $s \leq 11n/10$

the size of the alphabet so only one set of figures for it is included in each of the tables. As the lengths of the strings increased, all the algorithms eventually failed due to a lack of memory as indicated by "-" in the tables.

For random strings, it appears that the two algorithm do not provide any improvement over the basic dynamic programming. However when the length of the SCS is known to be close to n, SCS_Thresh shows a big improvement over the Dynamic Programming Algorithm.

As was the case for the LCS problem in Chapter 2, the clearest conclusion that can be drawn from the experimental results is that the size of instances of the SCS problem which can be solved in practice using these algorithms is constrained by space requirements and not time requirements.

Chapter 4

Exact solutions to large instances of the LCS and SCS problems

4.1 Introduction

The primary aim in this chapter is to identify the range of instances of the Longest Common Subsequence and Shortest Common Supersequence Problems that can be solved in practice using realistic computational resources. To achieve this, a range of efficient algorithms employing two fundamentally different approaches have been implemented and experiments performed to discover how large the problem instances may become before the time or the memory space required to solve them becomes impractical.

Throughout the literature, there appears to be an implicit assumption that algorithms based on dynamic programming are the best way to solve these two problems. The second aim of this chapter is to investigate the applicability of algorithms based on a natural branch-and-bound strategy and find out whether they can extend the range of problem instances that we can expect to be able to solve in practice.

The two strings case has been treated separately because the two problems become dual in this case and because it has received so much attention in the literature. For more than two strings, all programs implemented solve instances with an arbitrary, specifiable number of strings. The results for any particular number of strings could be improved a little by tailoring the algorithms specifically for that case.

The critical parameters of a problem instance are (i) the number of strings k, (ii) the size of the alphabet used z, and (iii) the lengths of the strings n. In all cases, strings of equal length n have been used because studying this case is sufficient to gain the information sought. For the rest of the chapter, P represents the set of strings and Σ represents the alphabet. The strings are labelled $\alpha_1, \alpha_2, \ldots, \alpha_k$.

A third purpose of this chapter is to provide a substantial body of empirical data to enable accurate estimation of the expected length of the LCS or SCS of

an arbitrary number of random strings of any particular length over an alphabet of any particular size. The expected LCS or SCS length over a given set of parameters (alphabet size (z), number of strings (k) and string lengths (n)), denoted by $f(z,k,n)$ or $g(z,k,n)$ respectively, is the average LCS or SCS length over all instances of the problem with that given set of parameters. As n tends to infinity, the limiting values for the ratio of the lengths of the LCS and SCS to n are defined to be

$$F(z,k) = \lim_{n\to\infty} \frac{f(z,k,n)}{n}$$

$$G(z,k) = \lim_{n\to\infty} \frac{g(z,k,n)}{n}.$$

See Section 1.3.5 for details of previous work.

4.2 Dynamic programming to solve LCS

As we know, Dynamic Programming was the first method of solution proposed for the LCS problem for two or more strings [25, 35, 68] and Dynamic Programming based strategies are the first approach employed in our experiments. Two different implementations of a *Lazy Dynamic Programming Algorithm*, derived directly from the recurrence relation given in Section 1.3.3 of Chapter 1, have been used. This is not the same as the lazy algorithm of Chapter 2. Calculation starts at the final entry of the dynamic programming table i.e. $L(n,n,\ldots,n)$. Evaluation of this entry requires the value of either 1 or k other entries in the table, each of which require the value of 1 or k other entries etc. Thus a recursive process evaluates a subset of the entries of the dynamic programming table. The algorithm is shown in pseudo-code in Figure 4.1.

4.2.1 Lazy dynamic programming using an Array

The first implementation of the Lazy Algorithm uses a one-dimensional array to represent the k-dimensional table required for the algorithm. This is done by representing the multi-dimensional table in a way similar to how a program in machine language represents a multi-dimensional array in "flat" memory. Assuming all the k dimensions have length m, indexed $(0 \ldots m-1)$, the single dimensional array has length m^k (the product of the lengths of dimensions) and is indexed $(0 \ldots m^k - 1)$. The multi-dimensional table entry $(i_1, i_2, \ldots, i_k)$ is stored in the single-dimensional array entry $((i_1 \times m^{k-1}) + (i_2 \times m^{k-2}) + \cdots + (i_{k-1} \times m^1) + i_k)$.

```
Solve LCS with Lazy Dynamic Programming
Output LCS(n,n,...,n)
end

function LCS(i_1, i_2, ..., i_k)
if L[i_1, i_2, ..., i_k] has not been calculated then
   if (i_1, i_2, ..., i_k > 0) then
      if α_1[i_1] = α_2[i_2] = ··· = α_k[i_k] then
         L[i_1, i_2, ..., i_k] :=LCS(i_1 - 1, i_2 - 1, ..., i_k - 1)+1
      else
         L[i_1, i_2, ..., i_k] :=max(LCS(i_1 - 1, i_2, ..., i_k),
                                     LCS(i_1, i_2 - 1, ..., i_k),
                                     ...
                                     LCS(i_1, i_2, ..., i_k - 1))
   else
      L[i_1, i_2, ..., i_k] :=0
return L[i_1, i_2, ..., i_k]
end
```

Figure 4.1: The Lazy Dynamic Programming Algorithm for the LCS problem.

4.2.2 Lazy dynamic programming using a Trie

If the dynamic programming table is very sparse then the array representation will be very inefficient with respect to space usage. This is likely to happen when the LCS is very long and the alphabet size is large. In this case, it would be ideal if memory space is used only for those entries that are calculated. This is possible using a trie [59] to represent the dynamic programming table.

For every entry in the DP array calculated, there is a path from the root to a leaf node where the value is stored. The steps in the path represent the indexes into the DP array. Physically, each node of the trie contains three elements, a *key*, a pointer to a child node and a pointer to a sibling node. In branch nodes, the key stores the value of the index into the DP array and in the leaf nodes, the key stores the DP table entry and the two pointers are null.

4.2.3 Analysis of Lazy DP solutions to the LCS

The worst-case requirement of the Lazy Dynamic Programming Algorithm is clearly $O(kn^k)$ for time and $O(n^k)$ for space. The array based implementation requires the initialisation of the entire array. In practice this initialisation takes very little time because it is such a simple task. The trie-based implementation requires no significant initialisation. It only uses space for the dynamic programming table

```
Solve LCS by Branch-and-Bound
γ, δ:= the empty string
for i:=1 to k do F[i,0]:=0
Choose.lcs(1)
Output δ
end

procedure Choose.lcs(length)
for every σ ∈ Σ do
   if Extendible.lcs(σ,length) then
      Append σ to γ
      if length > |δ| then δ:=γ
      Choose.lcs(length+1)
      Remove σ from the end of γ
end

function Extendible.lcs(σ,length)
for i:=1..k do
   F[i,length]:=next occurrence of σ after F[i,length-1] in α_i
return (length+min_{1≤i≤k}(n-F[i,length])>|δ|)
end
```

Figure 4.2: The basic Branch-and-Bound Algorithm for the LCS problem.

entries it evaluates but requires more space for each entry.

4.3 Branch-and-bound to solve LCS

The second approach employed to solve the LCS problem is a very simple branch-and-bound strategy. Strings are generated lexicographically. Each string is tested to see if it is a common subsequence of P. If a string γ is a common subsequence of P and the bounding conditions are satisfied then all one character extensions of γ are tested. For each string α_i in P, the length of the shortest prefix of α_i which is a supersequence of γ is stored in a 2-dimensional array F, indexed $[1 \ldots k, 0 \ldots n]$, in entry $F[i, |\gamma|]$. A simple bounding condition is that $|\gamma| + \min_{1 \leq i \leq k} (n - F[i, |\gamma|])$ be greater than the length of the longest common subsequence found so far. The algorithm is shown in pseudo-code in Figure 4.2.

4.3.1 Finding a good initial bound

The execution of branch-and-bound algorithms can usually be speeded up greatly if a good initial bound is found. For this it is natural to use a fast approximation

```
procedure Choose2.lcs(length)
tot:=0
for every σ ∈ Σ do
   tot:=tot+min_{1≤i≤k}(occurrences of σ in α_i after position F[i,length-1])
if length-1+tot>|δ| then
   for every σ ∈ Σ do
      if Extendible.lcs(σ,length) then
         Append σ to γ
         if length > |δ| then δ:=γ
         Choose2.lcs(length+1)
         Remove σ from the end of γ
end
```

Figure 4.3: The procedure *Choose2.lcs*.

algorithm. The approximation algorithm Best-Next from Chapter 5 was employed for this purpose because, in practice, it was found to produce a very good approximation.

4.3.2 Improving on the basic strategy

Using character frequency analysis

The first method applied to speed up the Branch-and-Bound Algorithm used character frequencies to calculate a tighter bound. Some simple preprocessing of the strings in P is carried out. For every α_i, the number of occurrences of each symbol in the alphabet after every position in α_i is calculated and retained in an array. During execution of the algorithm, before extending a common subsequence γ, the sum t of the minimum numbers of each symbol occurring in the unused suffixes of the α's is calculated. That is, the sum for all $\sigma \in \Sigma$ of

$$\min_{1 \leq i \leq k} (\text{the number of occurrences of } \sigma \text{ in } \alpha_i \text{ after position } F[i, |\gamma|]).$$

The sum t is then an upper bound on the length of the LCS of the unused suffixes of the α's. The bounding condition is that $|\gamma| + t$ be greater than the length of the longest common subsequence found so far. This generates a bound which is independent from the bound already used in the function *Extendible.lcs*. Pseudocode for the procedure *Choose2.lcs* which is an alternative to *Choose.lcs* is shown in Figure 4.3.

```
function Extendible2.lcs(σ,length)
for i:=1 to k do
   F[i,length]:=next occurrence of σ after F[i,length-1] in α_i
q:=max_{1≤i≤k}(n-F[i,length])
return ((length+min_{i=1...k}(n-F[i,length])>|δ|) and (length+U[q]>|δ|))
end

Calculate U
for i:=1 to n do U[i]:=i;
for every pair of strings α_i and α_j do
   build L, the LCS DP array of the reverse of α_i and the reverse of α_j
   for x:=1 to n do
      U[x]:=min(U[x], L[x,x])
end
```

Figure 4.4: The function *Extendible2.lcs*.

Using all pairwise LCS lengths

A strong bounding condition can be achieved using pairwise comparison of the strings in P. For $1 \leq q \leq n$, the array $U[q]$ represents an upper bound on the LCS of the k suffixes $\alpha_1[n-q+1 \ldots n], \alpha_2[n-q+1 \ldots n], \ldots, \alpha_k[n-q+1 \ldots n]$. The array U is calculated as follows. For every pair of strings, α_i and α_j, the entire dynamic programming table for the LCS of the reverse of α_i and the reverse of α_j is calculated. Thus the length of the LCS of every suffix of α_i and every suffix of α_j is *temporarily* stored. For $1 \leq q \leq n$, $U[q]$ is set to the minimum of the LCS's of the suffixes $\alpha_i[n-q+1 \ldots n]$ and $\alpha_j[n-q+1 \ldots n]$ over all α_i and α_j.

After the algorithm generates the common subsequence γ it finds the length q of the longest[1] suffix, of a string α_i, not required for α_i to be a supersequence of γ. The bounding condition is that $|\gamma| + U[q]$ be greater than the length of the longest common subsequence found so far. Pseudo-code for the function *Extendible2.lcs* which is an alternative to *Extendible.lcs* is shown in Figure 4.4.

Using all pairwise LCS DP arrays

A stronger bound is obtained if the dynamic programming table for the LCS of every pair of strings, α_i and α_j, in P is calculated and retained. Each array is calculated using the reverse of α_i and the reverse of α_j. Thus the length of the LCS of any suffix of α_i and any suffix of α_j can be found in constant time. A very tight bounding condition is established as follows. When the common subsequence γ is generated,

[1] In fact, the second longest could be used but the difference this makes is not significant.

```
Extendible3.lcs(σ,length)
for i:=1 to k do
   F[i,length]:=next occurrence of σ after F[i,length-1] in α_i
p:=min_{1≤i≤k}(n-F[i,length])
q:=min_{1≤i≤k}(n-F[i,length]) i<>p
return ((length+min_{1≤i≤k}(n-F[i,length])>|δ|) and
        (length+LCS(p,F[p,length]+1,q,F[q,length]+1)>|δ|))
end

where LCS(p,x,q,y) = the length of the LCS of the suffixes
                     α_p[n − x + 1...n] and α_q[n − y + 1...n].
```

Figure 4.5: The function *Extendible3.lcs*.

the two strings α_i and α_j, such that $F[i]$ and $F[j]$ are the highest two values in the array F, are found. The bounding condition is that the length of γ plus the length of the LCS of $\alpha_i[F[i]\ldots n]$ and $\alpha_j[F[j]\ldots n]$ be greater than the length of the longest common subsequence found so far. Pseudo-code for the function *Extendible3.lcs* which is an alternative to *Extendible.lcs* is shown in Figure 4.5.

4.3.3 Analysis of Branch-and-Bound solutions to the LCS

It is very difficult to say anything concrete about the behaviour of branch-and-bound algorithms in practice. However, since the branch-and-bound algorithm generates strings of length $\leq l$ spending $O(k)$ on each, and there are $O(z^l)$ strings of length $\leq l$, the worst-case time complexity is $O(kz^l)$. The space complexity is $O(knz)$ to store the next occurrence tables. In practice, the heuristics employed will hopefully make branch-and-bound run much faster than the worst case.

Using all the pairwise LCS DP arrays imposes a

$$\Theta\left(\binom{k}{2}n^2\right)$$

space requirement. For high k, this will be very significant. Later in this chapter the behaviour of the branch-and-bound algorithms for both LCS and SCS are discussed in more detail.

4.4 Dynamic programming to solve SCS

As for the LCS problem, two implementations of a Lazy Dynamic Programming Algorithm for the SCS problem have been used. The algorithm is derived directly

```
Solve SCS with Lazy Dynamic Programming
Output SCS(n, n, ..., n)
end

function SCS(i_1, i_2, ..., i_k)
if S[i_1, i_2, ..., i_k] has not been calculated then
    if (i_1 = i_2 = ··· = i_k = 0) then
        S[i_1, i_2, ..., i_k]:=0
    else
        low:=n × k
        for every σ ∈ {α_1[i_1], α_2[i_2], ..., α_k[i_k]} do
            low:=min(low,SCS(F(i_1, σ), F(i_2, σ), ..., F(i_k, σ))
        S[i_1, i_2, ..., i_k]:=low
return S[i_1, i_2, ..., i_k]
end

function F(i_h, σ)
if i_h = 0 then
    return 0
else
    if α_h[i_h] = σ then
        return i_h − 1
    else
        return i_h
end
```

Figure 4.6: The Lazy Dynamic Programming Algorithm for the SCS problem.

from the recurrence relation given in Section 1.4.2. Calculation starts at the final entry of the dynamic programming table i.e. $S(n, n, \ldots, n)$. Evaluation of this entry requires the value of between 1 and k other entries, each of which requires the value of between 1 and k other entries, etc. Thus a recursive process evaluates a subset of the entries of the dynamic programming table. The algorithm is shown in pseudo-code in Figure 4.6. As for the LCS problem, two implementations of the Lazy Dynamic Programming algorithm were employed. They used an array and a trie respectively to represent the dynamic programming table in precisely the same way as for the LCS problem.

4.4.1 Analysis of Lazy DP solutions to the SCS

The analysis is precisely the same as that for the Lazy Dynamic Programming Algorithm for the LCS problem. Both implementations require $O(kn^k)$ time and

$O(n^k)$ space in the worst case.

4.5 Branch and bound to solve SCS

A branch-and-bound strategy, similar to that employed for the LCS problem, was the second approach used to solve the SCS problem. The basic algorithm proceeds as follows. Strings are generated lexicographically. For the current string γ of length m, a two-dimensional array F indexed $[1 \ldots k, 0 \ldots n]$ stores in $F[i, m]$, for $1 \leq i \leq k$, the length of the longest prefix of α_i which is a subsequence of γ. So γ is a supersequence of the prefixes $\alpha_i[1 \ldots F[i, m]]$, for $1 \leq i \leq k$. The string γ', a one character extension of γ, is clearly a supersequence of all $\alpha_i[1 \ldots F[i, m]]$ and may be a supersequence of some $\alpha_i[1 \ldots F[i, m] + 1]$, for $1 \leq i \leq k$. If for any i, γ' is a supersequence of $\alpha_i[1 \ldots F[i, m] + 1]$ and the bounding conditions are met, then all one character extensions of γ' are tested. A simple bounding condition is that $m + 1 + \max_{1 \leq i \leq k} (n - F[i, m + 1])$ be less than the length of the shortest common supersequence found so far. The algorithm is shown in pseudo-code in Figure 4.7.

4.5.1 Finding a good initial bound

To find a good lower initial bound on the length of the SCS, two of the approximation algorithms from Chapter 5 were employed. The Majority Merge algorithm and the Greedy2 algorithm were chosen because when testing the approximation algorithms, almost invariably one of those two produced the best approximation. Both algorithms are executed and the lower approximation chosen as the initial upper bound for the SCS length. See Chapter 5 for descriptions of the two algorithms.

4.5.2 Improving on the basic strategy

The methods used to speed up the Branch-and-Bound Algorithm for the SCS problem are all analogies of the methods used to speed up the Branch-and-Bound Algorithm for the LCS problem.

Using character frequency analysis

Precisely the same preprocessing of the strings is carried out as for the LCS problem. During execution of the algorithm, before extending a common supersequence γ, the sum t of the maximum numbers of each symbol occurring in the unused suffixes of the α's is calculated. That is, the sum for all $\sigma \in \Sigma$ of

$$\max_{1 \leq i \leq k} (\text{the number of occurrences of } \sigma \text{ in } \alpha_i \text{ after position } F[i, |\gamma|]).$$

```
Solve SCS by Branch-and-Bound
γ:= the empty string
δ:= Σ^n (δ therefore has length kn)
for i:=1 to k do F[i,0]:=0
Choose.scs(1)
Output δ
end

procedure Choose.scs(length)
for every σ ∈ P do
   if Extends.scs(σ,length) then
      Append σ to γ
      if F[1...k,length]=n then δ:=γ
      else Choose.scs(length+1)
      Remove σ from the end of γ
end

function Extends.scs(σ,length)
xtends:=FALSE
for i:=1 to k do
   if α_i[F[i,length-1]+1]=σ then
      F[i,length]:=F[i,length-1]+1
      xtends:=TRUE
   else
      F[i,length]:=F[i,length-1]
return (xtends and (length+max_{1≤i≤k}(n-F[i,length])<|δ|))
end
```

Figure 4.7: The basic Branch-and-Bound Algorithm for the SCS problem.

```
procedure Choose2.scs(length)
tot:=0
for every σ ∈ Σ do
   tot:=tot+max_{1≤i≤k}(occurrences of σ in α_i after position F[i,length-1])
if length-1+tot<|δ| then
   for every σ ∈ P do
      if Extends.scs(σ,length) then
         Append σ to γ
         if F[1...k,length]=n then δ:=γ
         else Choose.scs(length+1)
         Remove σ from the end of γ
end
```

Figure 4.8: The function *Choose2.scs*.

The sum t is then a lower bound on the length of the SCS of the unused suffixes of the α's. The bounding condition is that $|\gamma| + t$ be less than the length of the shortest common supersequence found so far. Pseudo-code for the procedure *Choose2.scs* which is an alternative to *Choose.scs* is shown in Figure 4.8.

Using all pairwise SCS lengths

For $1 \leq q \leq n$, the array $L[q]$ represents a lower bound on the SCS of the k suffixes $\alpha_1[n-q+1 \ldots n], \alpha_2[n-q+1 \ldots n], \ldots, \alpha_k[n-q+1 \ldots n]$. The array L is calculated as follows. For every pair of strings, α_i and α_j, the entire dynamic programming table for the SCS of the reverse of α_i and the reverse of α_j is calculated. Thus the length of the SCS of every suffix of α_i and every suffix of α_j is *temporarily* stored. For $1 \leq q \leq n$, $L[q]$ is set to the maximum of the SCS's of the suffixes $\alpha_i[n-q+1 \ldots n]$ and $\alpha_j[n-q+1 \ldots n]$.

After the algorithm generates the common supersequence γ it finds the length q of the shortest[2] suffix, of a string α_i, not required for α_i to be a supersequence of γ. The bounding condition is that $|\gamma| + L[q]$ be less than the length of the shortest common supersequence found so far. Pseudo-code for the function *Extends2.scs* which is an alternative to *Extends.scs* is shown in Figure 4.9.

Using all pairwise SCS DP arrays

For every pair of strings, α_i and α_j, the SCS DP table of the reverse of α_i and the reverse of α_j is calculated and retained. Thus the length of the SCS of any suffix of α_i and any suffix of α_j can be found in constant time. A very tight bounding condition

[2]In fact, the second shortest could be used but the difference this makes is not significant.

```
function Extends2.scs(σ,length)
xtends:=FALSE
for i:=1 to k do
   if α_i[F[i,length-1]+1]=σ then
      F[i,length]:=F[i,length-1]+1
      xtends:=TRUE
   else
      F[i,length]:=F[i,length-1]
p:=min_{1≤i≤k}(n-F[i,length])
return (xtends and (length+max_{1≤i≤k}(n-F[i,length])<|δ|)
                and (length+L[p]<|δ|))
end

Calculate L
for i:=1 to n do L[i]:=i
for every pair of strings α_i and α_j do
   build S, the SCS DP array of the reverse of α_i and the reverse of α_j
   for x:=1 to n do
      L[x]:=max(L[x],S[x,x]) end
```

Figure 4.9: The function *Extends2.scs*.

is established as follows. When the partial supersequence γ is generated, the two strings, α_i and α_j, with the shortest maximal length prefixes that are subsequences of γ are found. The bounding condition is that the length of γ plus the SCS of the uncovered suffixes of α_i and α_j be less than the length of the shortest common supersequence found so far. Pseudo-code for the function *Extdends3.scs* which is an alternative to *Extends.scs* is shown in Figure 4.10.

4.5.3 Analysis of Branch-and-Bound solutions to the SCS

Since the branch-and-bound algorithm generates strings of length $\leq s$ spending $O(k)$ on each, and there are $O(z^s)$ strings of length $\leq s$, the worst-case time complexity is $O(kz^s)$. The space complexity is $O(kn)$ to store the strings themselves. In practice, the heuristics employed will hopefully make branch-and-bound run much faster than the worst case.

Using all the pairwise SCS DP arrays imposes a

$$\Theta\left(\binom{k}{2}n^2\right)$$

space requirement. For high k, this will be very significant. Later in this chapter the

```
function Extends3.scs(σ,length)
xtends:=FALSE
for i=1 to k do
   if α_i[F[i,length-1]+1]=σ then
      F[i,length]:=F[i,length-1]+1
      xtends:=TRUE
   else
      F[i,length]:=F[i,length-1]
p:=max_{1≤i≤k}(n-F[i,length])
q:=max_{1≤i≤k}(n-F[i,length]) i<>p
return (xtends and (length+max_{i=1...k}(n-F[i,length])<|δ|)
               and (length+SCS( p,F[p,length]+1,q,F[q,length]+1)<|δ|))
end

where SCS(p,x,q,y) = the length of the SCS of the suffixes
                     α_p[x...n] and α_q[y...n]
```

Figure 4.10: The function *Extends3.scs*.

behaviour of the branch-and-bound algorithms for both LCS and SCS are discussed in more detail.

4.6 Solving LCS and SCS for two strings

As was discussed in Chapter 1, when there are only two strings, the LCS and SCS problems become dual and this special case has received a great deal of attention in the literature. The threshold algorithm described in Section 1.3.2 was used because it combines excellent time and space performance in a very straightforward algorithm. Recently Rick [57] published a similar algorithm (but with quadratic space complexity) and showed empirically that for most values of l ($0 \leq l \leq n$) it is faster than four previously published algorithms (Apostolico and Guerra [6], Nakatsu et el. [52], Chin and Poon [13], and Wu et al. [71]).

4.7 The experiments

The objectives of the experiments were:

1. Identify, for both problems (LCS and SCS), the range of problem instances solvable in practice using realistic computational resources.

2. Assess the applicability of branch and bound algorithms to solving these problems.

3. Generate a large body of empirical results to allow accurate estimation of the expected lengths of LCS and SCS for random strings over a wide range of string numbers, alphabet sizes and string lengths.

All the programs were implemented in Pascal and compiled with the Sun Pascal Compiler version 1.0 using the "-O" option to provide the widest range of optimisations. The experiments were carried out on a SPARCstation-10 workstation with sixteen megabytes of RAM.

Two sets of experiments were carried out. The first set used completely random strings. That is, every symbol was selected uniformly at random from the alphabet. The second set used strings known to have a relatively long LCS or a relatively short SCS. Similar strings were used because it is when strings are similar that their relationship is likely to be interesting in practical applications. For the experiments on the LCS problem, the strings had an LCS of length at least $9n/10$. They were created by first generating a random string with length $9n/10$. Then k copies of that string were made, each with $n/10$ additional random characters inserted in random positions. The LCS for those k strings of length n therefore has length at least $9n/10$. For the experiments on the SCS problem, the strings had an SCS of length at most $11n/10$. They were created by first generating a random string with length $11n/10$. Then k copies of that string were made, each with $n/10$ characters deleted from random positions. The SCS for those k strings of length n therefore has length at most $11n/10$.

To measure the effect of the number of strings, k was taken over the values $2, 3, 4, 5, 6, 7, 8, 16, 24$.

To measure the effect of the alphabet size for $k > 2$, an exponential scale was used - alphabet sizes of $2, 4, 8, 16$, because preliminary testing suggested that, with all other factors being equal, a significant change in the alphabet size was required to show a significant change in either of the measured quantities (time and LCS/SCS length). For $k = 2$, all alphabet sizes from two to sixteen were studied to find accurate estimations of $F(2, z)$ which can be compared to the upper and lower bounds found in the literature.

For every k and z pair, the length of the strings tested rose in a linear scale until the problem instances could not be solved in one hour of CPU time or due to a lack of memory. For the various programs and for different values of k, different scales were used, depending on how quickly the time required to solve problems rose with n.

The experiments are grouped into families where each k and z pair represents one family of experiments. For each family, as n increases we can expect to see a pattern in the behaviour of each program. Except where otherwise stated, each individual experiment within a family was only run once. That is, for each k, z and n triple, only one problem instance was generated and solved. So for each family,

a set experiments was performed, with one parameter changing slightly from one experiment to the next.

For the array-based Lazy Dynamic Programming Algorithms, an upper bound of 2^{24} was imposed on the number of cells used by the multi-dimensional DP table to represent a reasonable limit on the amount of RAM available to the machine.

4.7.1 Behaviour of the branch and bound algorithms

For studying the results of the experiments, some definitions and observations about the behaviour of the branch and bound algorithms are useful. A branch and bound algorithm can be viewed as searching a tree where a correct solution lies at some leaf node. The *breadth* of the tree is the maximum number of children of each node. The *depth* of the tree is the length of the longest path from the root node to a leaf node. The *density* of the tree is a measure of the proportion of the tree actually visited during execution of the algorithm. This will depend mainly on how tight an initial bound is found and on the strength of the bounding conditions used.

The time required to solve a problem with a branch and bound algorithm depends on the number of nodes of the tree visited and on the time spent at each node. The number of nodes visited will depend on the above three factors - breadth, depth, and density. In the context of the branch and bound algorithms for the LCS and SCS problems, proposed here, the following facts are trivial;

1. The breadth is equal to the size of the alphabet.
2. The depth is equal to the length of the LCS or SCS respectively.
3. The time spent at each node is proportional to k since each new symbol generated must be checked against k strings.

4.7.2 Results for the LCS problem when $k > 2$

Experimental results for only one program based on dynamic programming are presented. In all cases, the array-based implementation of the Lazy Dynamic Programming Algorithm was superior to the trie-based implementation. The array-based implementation was consistently significantly faster. All instances that could not be solved by the array-based implementation due to the amount of memory required also could not be solved by the trie-based implementation for the same reason. From here on, the array-based implementation will be referred to as *LCS-DP*.

The experimental results for two programs based on branch-and-bound are presented. In all cases of the algorithms based on branch-and-bound, the basic algorithm with the function *Extendible3.lcs* was the fastest. No further increase in speed was observed when, additionally, the procedure *Choose2.lcs* replaced *Choose.lcs*.

	n									
k	10	20	30	40	50		100	150	200	250
3					1.5		12	45	159	345
4	0.1	2.4	13	42	136		-	-	-	-
5	2.0	62	-	-	-		-	-	-	-
6	26	-	-	-	-		-	-	-	-
7	-	-	-	-	-		-	-	-	-

$z = 4$

Table 4.1: CPU times in second for LCS-DP on random strings.

From here on, the basic algorithm using *Extendible3* will be referred to as *LCS-BB-3*. For the instances involving random strings, *LCS-BB-3* failed to solve instances as the length of the strings increased because the time required was too great. However, for the instances where the LCS had length at least $9n/10$, much higher values for n were achieved before the solution time became a problem. Consequently, the amount of memory required to store the LCS DP array for all pairs of strings, used in *Extendible3.lcs*, was the limiting factor. Our second program *LCS-BB-2* is an implementation of the basic algorithm with the function *Extendible2.lcs*. For most cases, it was able to solve instances with larger n than *LCS-BB-3*, when the LCS was long, so results for it are presented. Again, no further increase in speed was observed when, additionally, the procedure *Choose2.lcs* replaced *Choose.lcs*.

Results for the LCS of random strings

The CPU times in seconds for the program *LCS-DP* running on random strings over an alphabet of size 4 are shown in Table 4.1. For every k and n pair, the time required to solve an instance with those parameter values rose very slowly with increasing alphabet size. Hence for alphabet sizes of 2,8 and 16, very similar sets of timings were observed. It is clear from the sizes of problem unsolvable due to memory requirements, and from the time required to solve the solvable instances, that memory requirements are the limiting factor in the random instances of the LCS problem solvable by dynamic programming based algorithms.

The CPU times in seconds for the program *LCS-BB-3* running on random strings are shown in Table 4.2. It appears that as the alphabet size increases, the effect of increasing the number of strings changes. When the alphabet size is two, increasing k results in the problems requiring more time to be solved and there is a reduction in the maximum value of n for which the problem can be solved within 1 hour of CPU time. When the alphabet size is sixteen however, increasing k appears to make the problems easier - less time is required to solve them and there is an increase in the maximum value of n for which the problem can be solved within 1 hour of CPU time. There appears to be a gradual change from the former behaviour to the latter as the alphabet size increases from two to sixteen. The explanation for

this behaviour can be found by looking at the actual lengths of the LCS's. As the number of strings increases, there is a decrease in l, the observed length of the LCS. The rate at which l decreases appears not to depend heavily upon the alphabet size, as illustrated by the following table which shows the observed value for l when $n = 70$ and its decrease over $k = 3$ and $k = 24$;

	$k = 3$	$k = 24$	Decrease
$z = 2$	48	36	12
$z = 4$	34	18	16
$z = 8$	24	9	15
$z = 16$	14	4	10

The observed $|LCS|$ for random strings when $n = 70$.

When $k = 3$, the value of l decreases very sharply with increasing alphabet size. Consequently the decrease of l *in proportion to its original value* increases with the alphabet size. As was observed in Section 4.7.1, increasing k increases the work done at each node. However it also reduces the length of the LCS i.e. the depth of the search tree, therefore reducing the number of nodes visited. Explanation for the behaviour described above is therefore as follows. For increasing k, when $z = 2$ the increase in work at each node outweighs the drop in the depth of the search tree. As the alphabet size increases, the drop in the depth of the search tree caused by increasing k starts to outweigh the increase in work required at each node caused by increasing k.

It is clear that, for random problem instances, the limiting factor on the size of solvable instances for branch and bound based algorithms is the time required to solve them.

Results for the LCS of similar strings

For problem instances where the length of the LCS was known to be long, the *LCS-DP* required less time than it did to solve the equivalent instances of random strings. However, it was unable to solve larger instances because the amount of memory required for the array to represent the DP table was unchanged.

As was stated at the beginning of the section, the program *LCS-BB-2* was able to solve larger problem instances than *LCS-BB-3* when the LCS was known to be very long. As the length of the strings increased, the behaviour of *LCS-BB-2* became less predictable as large variances in the running times were recorded. The explanation for this lay in the lengths of the common subsequence found by the approximation algorithm BestNext. When BestNext found a strong initial bound, *LCS-BB-2* completed fairly quickly and when BestNext found a weak initial bound, *LCS-BB-2* required a relatively long time to complete. Table 4.3 shows, for alphabet

	n					
k	80	90	100	110	120	130
3	1.2	5.1	5.5	80	236	485
4	8.8	56	52	2779	-	-
5	41	329	511	1990	-	-
6	62	388	1739	-	-	-
7	58	1668	-	-	-	-
8	96	2096	-	-	-	-
:						
16	369	-	-	-	-	-
:						
24	865	-	-	-	-	-

$z = 2$

	n					
k	60	70	80	90	100	110
3	0.1	3	4.9	19	440	126
4	1.5	23	52	337	2201	-
5	3.7	30	25	3125	-	-
6	17	49	117	2418	-	-
7	18	149	229	-	-	-
8	15	95	265	-	-	-
:						
16	8.8	141	688	-	-	-
:						
24	8.1	143	642	-	-	-

$z = 4$

	n					
k	60	70	80	90	100	110
3	0.2	3.5	1.8	52	909	2201
4	0.6	7.4	34	403	1321	-
5	1.4	5.9	230	294	2305	-
6	0.7	3	380	177	1015	-
7	0.8	2.8	300	94	870	-
8	0.8	4.5	42	132	924	-
:						
16	0.7	1.8	20	60	451	-
:						
24	1.5	2.4	14	32	165	2181

$z = 8$

	n					
k	110	120	130	140	150	160
3	56	140	1794	-	-	-
4	131	879	-	-	-	-
5	213	502	2936	-	-	-
6	88	433	2173	-	-	-
7	43	269	2408	-	-	-
8	43	261	1475	-	-	-
:						
16	5.7	19	116	304	1148	-
:						
24	5.3	11	31	31	252	1463

$z = 16$

Table 4.2: CPU times in seconds for *LCS-BB-3* on random strings.

z	n
2	250
4	400
8	600
16	900

Table 4.3: Instances of LCS with similar strings solvable by *LCS-BB-2* in 1 hour.

sizes of 2,4,8, and 16, how long the strings could be, typically, before *LCS-BB-2* was unable to solve the problem instances within 1 hour of CPU time for the recorded results. The number of strings did not appear to be a significant factor in this. The figures for n are to the nearest 50.

As the alphabet size increases, the length of strings for which the problem can be solved increases. This is because, as the alphabet size increases, the number of subsequences with length close to that of the longest decreases. The density of the search tree is therefore reduced and this outweighs the increase in the breadth.

It is clear that for very similar strings significantly longer instances can be solved than for random instances and that time is still the limiting factor on how large the problem instances can become and still be solved by realistic resources.

4.7.3 Results for the SCS problem when $k > 2$

As for the LCS problem, the array-based implementation of the lazy dynamic programming algorithm was consistently superior to the trie-based implementation. The results for the array-based implementation, labelled *SCS-DP*, are presented.

Unlike for the LCS problem, results for only one program based on branch-and-bound are presented. The basic algorithm with the function *Extends.scs3* was consistently the fastest and was able to solve all instances solvable by the other variations.

Results for the SCS of random strings

The CPU times in seconds for the program *SCS-DP* running on random strings are shown in Table 4.4. It is clear that as the alphabet size increases, the time required increases very quickly. There are two reasons for this. Examination of the observed SCS lengths reveals, unsurprisingly, that as the alphabet size increases, the lengths of the SCS's, and hence the proportion of the DP table required to calculate the value of the final cell, increase. Close examination of the algorithm (Figure 4.6) reveals that the increase in the alphabet size is itself likely to increase the proportion of the cells in the DP table that are required to calculate the value of the final cell. This is because, for any particular cell, the number of cells required to calculate its value is equal to the size of the set of symbols from the 'current' positions in the strings. This set is likely to become bigger as the alphabet size increases. As for the LCS problem, it is clear that memory requirements are the limiting factor on the size of random problem instances solvable by dynamic programming based algorithms.

The CPU times for the program *SCS-BB-3* running on random strings are shown in Table 4.5. As for the Lazy Dynamic Programming Algorithm, as the alphabet size increases, the time required increases very quickly. The explanation is quite straightforward. As noted in Section 4.7.1, the alphabet size is equivalent to the breadth of the search tree. As noted above, as the alphabet size increases, the lengths of the SCS's increases. Hence the depth of the search tree increases. With both the breadth and depth of the search tree increasing with alphabet size, there is a very sharp increase in the number of nodes in the search tree and a corresponding increase in the time required to solve the problem instances. As for the LCS problem, time appears to be the limiting factor on the sizes of random problem instances solvable by branch and bound based algorithms.

Results for the SCS of similar strings

For problem instances where the length of the SCS was known to be short, *SCS-DP* required less time than it did to solve equivalent instances of random strings. As for *LCS-DP* however, it was unable to solve larger instances than for random strings

	n										
k	10	20	30	40	50	60		100	150	200	250
3					0.3			2.1	8.4	17	35
4	0	0.1	1.0	2.0	6.5	13		-	-	-	-
5	0.1	1.6	-	-	-	-		-	-	-	-
6	0.3	-	-	-	-	-		-	-	-	-
7	-	-	-	-	-	-		-	-	-	-

$z = 2$

	n										
k	10	20	30	40	50	60		100	150	200	250
3					0.9			6.9	24	60	119
4	0.1	0.8	4.2	13	32	71		-	-	-	-
5	0.5	12	-	-	-	-		-	-	-	-
6	2.2	-	-	-	-	-		-	-	-	-
7	-	-	-	-	-	-		-	-	-	-

$z = 4$

	n										
k	10	20	30	40	50	60		100	150	200	250
3					2.1			16	55	137	262
4	0.2	2.7	12	41	113	216		-	-	-	-
5	1.9	58	-	-	-	-		-	-	-	-
6	16	-	-	-	-	-		-	-	-	-
7	-	-	-	-	-	-		-	-	-	-

$z = 8$

	n										
k	10	20	30	40	50	60		100	150	200	250
3					4.2			34	113	264	527
4	0.4	6.6	33	97	250	474		-	-	-	-
5	4.3	146	-	-	-	-		-	-	-	-
6	37	-	-	-	-	-		-	-	-	-
7	-	-	-	-	-	-		-	-	-	-

$z = 16$

Table 4.4: CPU times in seconds for *SCS-DP* on random strings.

	n													
k	10	20	30	40	50	60	70		10	20	30		10		10
3	0	0	0	0.2	25	2.9	474		0	0.2	16		0		0.5
4	0	0	2.6	4.2	427	657	-		0.1	4.6	-		1.4		2133
5	0	0.6	5.1	20	-	-	-		0.5	-	-		310		-
6	0	0.1	7.6	86	-	-	-		0.8	-	-		2537		-
7	0	0.8	33	287	-	-	-		2.5	-	-		-		-
8	0	0.3	47	650	-	-	-		13.7	-	-		-		-
:										-	-		-		-
16	0.1	4.1	723	-	-	-	-		-	-	-		-		-
:										-	-		-		-
24	0.2	8.4	-	-	-	-	-		-	-	-		-		-
				$z = 2$						$z = 4$			$z = 8$		$z = 16$

Table 4.5: CPU times in seconds for *SCS-BB-3* on random strings.

because the amount of memory required for the array to represent the DP table was unchanged.

Similarly when the length of the SCS was known to be short, *SCS-BB-3* required less time than it did to solve the equivalent instances of random strings. Consequently *much larger* instances could be solved within 1 hour of CPU time since memory requirement were still not a problem. The following table shows how long the strings could be, typically, before *SCS-BB-3* was unable to solve the problem instances within 1 hour of CPU time. The values for n are to the nearest 50.

	k								
z	3	4	5	6	7	8		16		24
2	250	250	250	250	250	250		250		200
4	350	350	350	350	350	350		250		200
8	450	400	350	350	350	350		250		200
16	550	450	450	350	350	350		250		200

Instances of SCS with similar strings solvable by *SCS-BB-3* in 1 hour.

As the alphabet size increases, the length of strings solvable within 1 hour of CPU time increases by an amount depending on k. As k increases, the increase in n is reduced. The explanation is as follows. As z increases, the number of supersequences of length almost as short as the shortest, and therefore the density of the search tree, decreases. This effect is more pronounced for small k because increasing k also has this effect. So when k is large, increasing the alphabet size has little or no effect on the density of the search tree.

4.7.4 Results for the LCS problem when $k = 2$

The implementation of the threshold algorithm for the LCS of 2 strings is referred to as *LCS2-THRESH*.

The CPU times in seconds for *LCS2-THRESH* running on random strings are shown in Table 4.6. The length of the longest common subsequence of two random strings of length $100,000$ over alphabet sizes $2, 4, 8, 16$ were found within one hour of CPU time.

A similar set of experiments were run where the length of the LCS was known to be long (i.e. 9n/10). As for when $k > 2$, less time was required to solve equivalent instances of the problem. In fact the time required to solve any instance with k strings of length n decreased with the alphabet size. The reason for this is that as the alphabet size increases but the length of the LCS remains constant, the number of common subsequences decreases making it "easier" for the algorithm to find the longest.

	n									
z	10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000	100,000
2	23	91	204	369	575	832	1125	1461	1857	2282
4	33	133	296	522	817	1183	1607	2108	2750	3308
8	35	141	317	563	887	1286	1748	2256	2861	3543
16	34	136	308	545	852	1223	1669	2170	2716	3406

Table 4.6: CPU times in seconds for *LCS2-THRESH* on random strings.

4.8 The expected lengths of the LCS and the SCS of random strings

4.8.1 The expected LCS length when $k > 2$

Recall that $f(z,k,n)$ represents the average LCS length over all instances of the problem for k strings of length n over an alphabet of size z. Estimations of

$$F(z,k) = \lim_{n\to\infty} \frac{f(z,k,n)}{n}$$

for $k = 3,4,5,6,7,8,16,24$ and $z = 2,4,8,16$ were made using the LCS lengths from the experiments carried out. For each k and z pair, a graph of $(|LCS|/n)$ was drawn and the limiting value estimated from the graph. Figure 4.11 shows a graph of the estimations augmented with those for $k = 2$ (which follow shortly).

4.8.2 The expected SCS length when $k > 2$

Recall that $g(z,k,n)$ represents the average SCS length over all instances of the problem for k strings of length n over an alphabet of size z. Estimations of

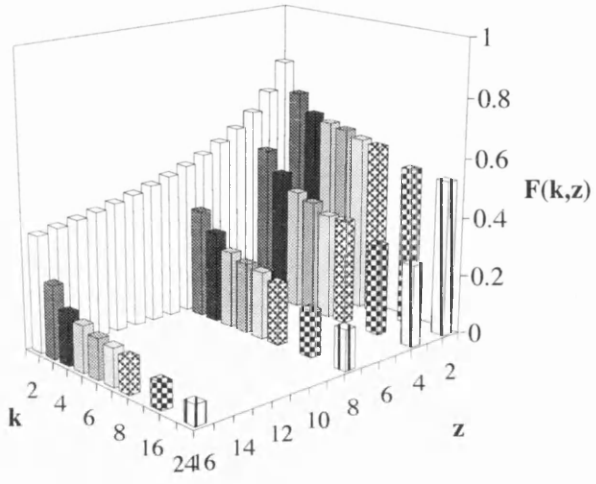
$$G(z,k) = \lim_{n\to\infty} \frac{g(z,k,n)}{n}$$

for $k = 3,4,5,6,7,8,16,24$ and $z = 2,4,8,16$ were made using the SCS lengths from the experiments carried out. For each k and z pair, where experimental results were available, a graph of $(|SCS|/n)$ was drawn and the limiting value estimated from the graph. Figure 4.12 shows a graph of the estimations augmented with those for $k = 2$ expressed in terms of $G(z,2)$.

4.8.3 The expected LCS length when $k = 2$

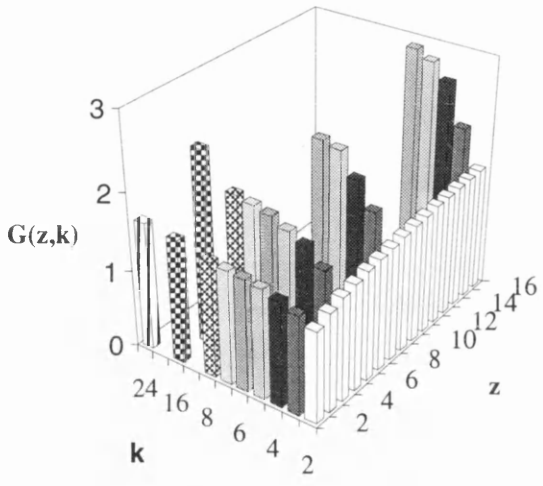
Estimations of

$$F(z,2) = \lim_{n\to\infty} \frac{f(z,2,n)}{n}$$



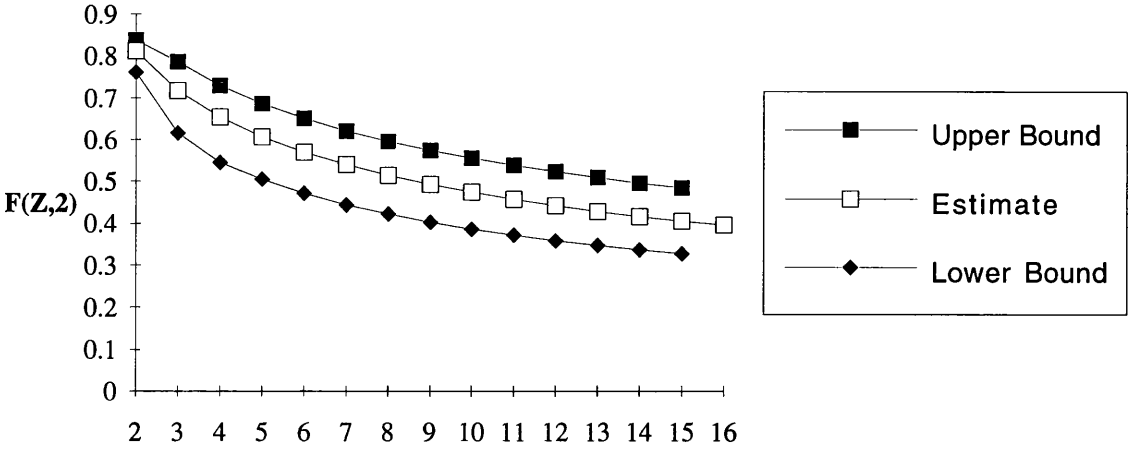
	k										
z	2	3	4	5	6	7	8		16		24
2	0.81	0.71	0.65	0.63	0.62	0.6	0.59		0.54		0.53
4	0.65	0.53	0.46	0.41	0.39	0.36	0.36		0.31		0.28
8	0.52	0.37	0.31	0.26	0.24	0.23	0.21		0.16		0.14
16	0.40	0.25	0.18	0.16	0.14	0.13	0.12		0.09		0.07

Figure 4.11: Expected lengths of the LCS of random strings.



	k										
z	2	3	4	5	6	7	8		16		24
2	1.19	1.29	1.37	1.43	1.46	1.48	1.5		1.6		1.65
4	1.35	1.59	1.8	1.9	2	2.05	2.1		2.5		-
8	1.48	1.84	2.16	2.45	2.5	-	-		-		-
16	1.60	2.08	2.6	2.8	2.9	-	-		-		-

Figure 4.12: Expected lengths of the SCS of random strings.



z	2	3	4	5	6	7	8	9
Upper Bound	0.8376	0.7864	0.7297	0.6861	0.6509	0.6209	0.5967	0.5750
Estimate	0.8123	0.7171	0.6542	0.6068	0.5702	0.5399	0.5151	0.4931
Lower Bound	0.7615	0.6153	0.5454	0.5061	0.4716	0.4450	0.4223	0.4032

z	10	11	12	13	14	15	16	-
Upper Bound	0.5559	0.5389	0.5236	0.5097	0.4970	0.4853	-	-
Estimate	0.4744	0.4573	0.4428	0.4291	0.4171	0.4062	0.3960	-
Lower Bound	0.3865	0.3719	0.3589	0.3473	0.3368	0.3273	-	-

Figure 4.13: |Expected LCS|/n for $k = 2$.

for $2 \leq z \leq 16$ were made by running, for each alphabet size, three random instances where the strings had length 100,000. For each alphabet size, the average value for the $(|LCS|/n)$ was computed and used as an estimation of $F(2, z)$. A table of the results and a graph comparing them to the upper and lower bounds found in the literature is shown in Figure 4.13.

4.9 Conclusions and future work

Figure 4.14 shows, for $k \geq 3$, the range of LCS problem instances solvable in 1 hour using the Dynamic Programming and Branch and Bound algorithms employed in these experiments. In the graph, for only and all the instances where $k = 3$, the largest values of n are solvable by the Lazy Dynamic Programming Algorithm. For only and all the instances where $k \geq 4$, the largest values of n are solvable by a branch-and-bound algorithm. The range of problems solvable by DP based strategies can however be extended. Adapting the space saving technique developed by Hirschberg [25] to any particular DP based algorithm reduces the $O(n^k)$ space requirement to $O(n^{k-1})$. Algorithms which improve on the basic Lazy Dynamic

Programming, such as those in Chapter 2, can also increase the maximum string length solvable for any particular k, but it is clear from the evidence presented in Chapter 2 that for $k \geq 5$ they will not compete with the Branch and Bound Algorithms. For the LCS problem then, algorithms based on branch and bound greatly extend the range of solvable problem instances.

Figure 4.15 shows, for $k \geq 3$, the range of SCS problem instances solvable in 1 hour using the Dynamic Programming and Branch and Bound algorithms employed in these experiments. In the graph, for only and all the instances where $k = 3$, the largest values of n are solvable by the Lazy Dynamic Programming Algorithm. For only and all the instances where $k \geq 4$, the largest values of n are solvable by a branch-and-bound algorithm. As for the LCS problem, the range of problem instances solvable by DP based algorithms can be extended by using Hirschberg's space saving technique. The ranges could be further extended by using algorithms which improve on the on basic Lazy Dynamic Programming, such as those in Chapter 3 but from the evidence in Chapter 3 it is clear that for a binary alphabet and $k \geq 4$ they will not compete with Branch and Bound Algorithms on *random strings.* For the SCS problem then, algorithms based on branch and bound extend the range of solvable problem instances but not yet by a great deal.

Estimations of the limiting value for the expected LCS and SCS lengths over the ranges $k = 3, \ldots, 8, 16, 24$, $z = 2, 4, 8, 16$ and $k = 2$, $z = 2, \ldots, 16$ have been found. The estimations for $F(z, 2)$ consistently lie close to half way between the upper and lower bounds provided in the literature.

One possible improvement to the Branch and Bound Algorithms for the LCS problem involves a a simple refinement of its use of the next occurrence table. At any node in the search tree, if

$$next_{\alpha_i}(j_i, \sigma) < next_{\alpha_i}(j_i, \varsigma) \qquad \forall i, 1 \leq i \leq k$$

for symbols σ and ς and positions j_i then ς is said to be *dominated* by σ. When the algorithm chooses a new symbol to append to the growing subsequence, it ignores any dominated symbols to prevent it from following some "blind alleys" in the search tree.

No doubt a variety of alternative branch and bound strategies for the SCS problem are possible. One particular alternative involves starting with the string χ consisting of the concatenation of n copies of all the symbols in Σ in lexicographic order. The string χ is then a supersequence of all the strings in P. Subsequences of χ are removed in lexicographic order as long as the resultant string remains a supersequence of all the strings in P. Preliminary testing with an implementation of this algorithm shows that it will at least allow finding the SCS of very many short strings for small alphabets ($k > 60$, $n < 20$, $z \leq 4$).

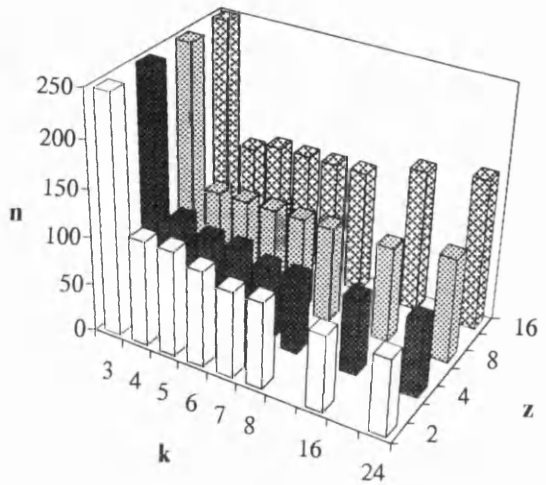


Figure 4.14: Random instances of LCS solvable in 1 hour.

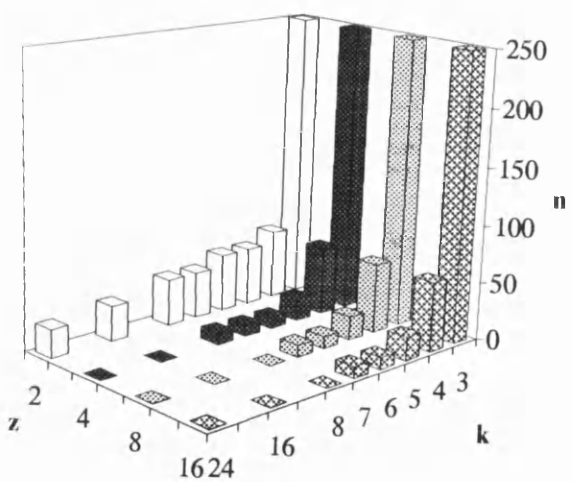


Figure 4.15: Random instances of SCS solvable in 1 hour.

Chapter 5

Approximation Algorithms for the LCS and SCS problems

5.1 Introduction

Given the **NP**-hard status of the Longest Common Subsequence and Shortest Common Supersequence problems, it is natural to attempt to find approximation algorithms for them with good performance guarantees. However Jiang and Li [37] showed that $\exists \delta > 0$ such that if the LCS problem has a polynomial time approximation algorithm with performance guarantee k^{δ} (where k is the number of strings) then **P**=**NP**. They showed that over an unbounded alphabet there can be no polynomial time approximation algorithm for SCS with a constant worst-case performance guarantee (unless **P**=**NP**). In this chapter we analyse the worst-case behaviour over bounded and unbounded alphabets for some approximation algorithms for the LCS and SCS problems. For the LCS problem, the analysis is either with respect to z, the alphabet size, or with respect to n, the length of the strings. This is because the worst-case behaviour of the algorithms under study turns out to depend directly on one or the other of these parameters. For the SCS problem, the analysis is almost entirely with respect to k, the number of strings. This is because it is the number of strings which make the problem hard to solve. In the final section, the worst-case of some of the SCS approximation algorithms with respect to the input size is considered.

To analyse the worst-case performance of the approximation algorithms for the LCS problem, we find upper and lower bounds for the ratio

$$\frac{\textit{Optimal length}}{\textit{Approximation length}}$$

in the worst case. In this way, the performance is measured as a positive real not less than 1 such that, the smaller the ratio, the better the approximation and a value

of 1 indicates exact solution. To achieve the same principle for the approximation algorithms for the SCS problem, we swap the arguments of the ratio thus;

$$\frac{\textit{Approximation length}}{\textit{Optimal length}}.$$

The first algorithm for the LCS problem was proposed by Jiang and Li [37]. They called it Long-Run and provided an analysis of its behaviour on average. The second algorithm uses a natural greedy strategy to build a common subsequence from left to right and is called Best-Next.

For the SCS problem, Timkovsky [64] proposed a tournament style approximation algorithm, and asked for a characterisation of its worst-case behaviour. Jiang and Li [37] described an algorithm called the Majority-Merge Algorithm and provided an analysis of its performance on average. An obvious approximation algorithm, named Greedy1, involves a greedy strategy, in which we start with the first string in the set, and repeatedly find an SCS of the current common supersequence and the next string in the original set. For a similar problem in multiple sequence comparison, Gusfield [20] described an algorithm named the Centre of the Star Algorithm, based on aligning one string optimally with all the other strings, and showed it had a constant worst-case performance guarantee for that problem. Our fourth algorithm adapts this strategy to the SCS problem. A direct improvement of this is considered where a minimum spanning tree of optimal alignments is used, instead of a star structure. This is the Minimum Spanning Tree (MST) Algorithm. Finally, a direct improvement of MST is considered where a minimum spanning tree is built, but after each edge is added to the spanning tree and hence to the alignment, the weights of all unused edges are re-calculated. This algorithm can also be viewed as a greedy algorithm since at each step it does what is locally best. Hence it is named Greedy2.

In this chapter, the aim is to characterise the worst case behaviour of these algorithms. The analysis assumes that all the strings are of equal length but can be easily extended to the case of varying string lengths.

Precise characterisation is given for both the LCS approximation algorithms. It is shown that the worst-case of Long-Run depends on the alphabet size and the worst-case of Best-Next depends on the length of the strings.

For the SCS approximation algorithms, it is shown that none has a worst-case performance guarantee better than $O(k)$ for an unbounded alphabet, and that none has a constant performance guarantee for a binary alphabet. This analysis can be contrasted with a trivial approximation algorithm which achieves a worst-case ratio of z for an alphabet of size z. If the alphabet is $\{a_1, a_2, \ldots, a_z\}$ and the strings have length n, the trivial algorithm returns the supersequence $(a_1 a_2 \ldots a_z)^n$.

All of the algorithms under study have stages in their execution where a choice

must be made between two or more equally valid options e.g. between two SCS's of a pair of strings. We will assume that when this occurs, the choice will be made arbitrarily. When describing a bad case for a particular algorithm, the behaviour described therefore represents a *feasible* execution of the algorithm but not necessarily the only possible execution.

When studying the behaviour of an algorithm over an alphabet of size two, the alphabet $\Sigma = \{a, b\}$ is used.

Approximation Algorithms for the LCS problem

5.2 The Algorithm Long-Run

The algorithm Long-Run proceeds as follows. It finds the maximum i such that the string σ^i is a common subsequence of all the strings in P, for some $\sigma \in \Sigma$ and returns that string as the common subsequence. This can be achieved in $O(kn + kz)$ time with one pass of every string.

5.2.1 Worst-case behaviour of Long-Run

Theorem 5.2.1 *For a given set of k strings over an alphabet of size z, denote by l the length of an LCS and by r the length of a common subsequence returned by Long-Run. Then*

$$\frac{l}{r} \leq z.$$

Proof
Assume, without loss of generality, that Long-Run returns the string a^x where $(1 \leq x \leq n)$. Any LCS contains $\geq x$ a's and less than or equal to x occurrences of each other symbol in the alphabet. The length of the LCS is therefore less than or equal to zx and the theorem is proved. □

5.2.2 Bad examples for Long-Run

Theorem 5.2.2 *For any $k \geq 2$, there is a set of k strings, over an alphabet of size z, for which*

$$\frac{l}{r} = z$$

where l is the length of an LCS and r the length of a common subsequence returned by Long-Run.

Proof
Consider the case where all the strings have length z and contain all the symbols of the alphabet in lexicographic order. The LCS clearly has length z but Long-Run will return a sequence of length 1. □

5.3 The Algorithm Best-Next

The algorithm Best-Next proceeds as follows. A common subsequence is constructed from left to right. Initialise the subsequence to the empty string. Let σ be the symbol which maximises the length of the shortest suffix of a string starting immediately after the first occurrence of σ in that string, over all the strings in P. Append σ to the subsequence and remove the shortest prefix containing σ from each string. Repeat this process until the strings have no common symbol remaining. With suitable preprocessing of the strings this can be achieved in $O(knz)$ time.

5.3.1 Worst-case behaviour of Best-Next

Theorem 5.3.1 *For a given set of k strings of length n, denote by l the length of an LCS, and by b the length of a common subsequence returned by Best-Next. Then*

$$\frac{l}{b} \leq \frac{n}{2}.$$

Proof
If $l \geq 1$ then Best-Next will return a sequence of length at least 1. Therefore for the ratio l/b to be greater than $n/2$, l must be greater than $n/2$. The first symbol of a common subsequence of length greater than $n/2$ must appear at or before position $\lfloor n/2 \rfloor$ and the last symbol of that sequence must appear at or after position $\lfloor n/2 \rfloor + 1$ in each string. In that case, the positions of those two symbols ensure that Best-Next will return a sequence of length at least 2 thus guaranteeing $l/b \leq n/2$. □

5.3.2 Bad examples for Best-Next

Theorem 5.3.2 *For any $k \geq 3$, there is a set of k strings over an alphabet of size 2 for which*

$$\frac{l}{b} = \frac{n}{2}$$

where l is the length of an LCS and b the length of a common subsequence returned by Best-Next.

Proof
Consider $k \geq 3$ strings of length n (n even) over the alphabet $\{0, 1\}$ defined as follows

$$\begin{aligned} \alpha_1 &= 0^{n/2}1^{n/2} \\ \alpha_2 &= 1^{n/2}0^{n/2} \\ \alpha_i &= 10^{n-1} \qquad (3 \leq i \leq k). \end{aligned}$$

The LCS of the strings is $0^{n/2}$ and has length $n/2$. A feasible string returned by Best-Next is the one character string 1. Hence $l/b = n/2$. □

Approximation Algorithms for the SCS problem

5.4 The Tournament Algorithm

The Tournament Algorithm applied to $k = 2q$ strings $\alpha_1^0, \ldots, \alpha_k^0$ of length n proceeds as follows. The tournament has $p = \lceil \log_2 k \rceil$ 'rounds'. In the r^{th} round there are $\lfloor \mathcal{R}(k) = (k + 2^{r-1} - 1)/2^r \rfloor$ 'matches' leaving $\lceil k/2^r \rceil$ strings remaining. The i^{th} match generates an arbitrary shortest common supersequence α_i^r of the two strings α_{2i-1}^{r-1} and α_{2i}^{r-1}, for $i = 1, \ldots, \mathcal{R}(k)$. If there are an odd number of strings in round r then the final string $\alpha_{\lceil k/2^{r-1} \rceil}^{r-1}$ receives a 'bye' and forms $\alpha_{\lceil k/2^r \rceil}^r$.

It follows that the string α_i^r $(i < \lceil (k/2^r) \rceil)$ is a common supersequence of the 2^r strings $\alpha_{(i-1)2^r+1}^0, \ldots, \alpha_{i2^r}^0$, that α_i^r $(i = \lceil (k/2^r) \rceil)$ is a supersequence of the $k - (i-1)2^r$ strings $\alpha_{(i-1)2^r+1}^0, \ldots, \alpha_k^0$ and in particular that α_1^p is a common supersequence of all the original $2q$ strings.

It is easy to verify that, using an $O(nm)$ algorithm for the SCS of 2 strings, the Tournament Algorithm for k strings of length n has $O(k^2n^2)$ time complexity.

5.4.1 Worst-case behaviour of the Tournament Algorithm

Theorem 5.4.1 *For a given set of $k = 2^p$ strings, denote by s the length of an SCS, and by t the length of a common supersequence returned by the Tournament Algorithm. Then*

$$\frac{t}{s} \leq \frac{3k+2}{8}.$$

Proof

Consider an SCS α, of length s say, of the strings $\alpha_1^0, \ldots, \alpha_k^0$, together with a particular embedding of each of the k individual strings in α. For $i = 1, 2, \ldots, k$, let x_i denote the number of positions in α that correspond to symbols in exactly i of the individual strings in the chosen embedding. Then

$$\sum_{i=1}^{k} x_i = s \tag{5.1}$$

and

$$\sum_{i=1}^{k} i x_i = kn. \tag{5.2}$$

Also, it is easy to see that

$$t = kn - \sum_{r=1}^{p} \sum_{i=1}^{2^{p-r}} l_{2i-1,2i}^{r-1} \tag{5.3}$$

where $l^{r-1}_{2i-1,2i}$ is the length of the longest common subsequence of the strings α^{r-1}_{2i-1} and α^{r-1}_{2i}.

Let $m_{i,j}$ $(1 \leq i < j \leq k)$ be the number of positions in α that correspond to symbols in both α^0_i and α^0_j for the chosen embeddings.

Since α^{r-1}_{2i-1} is a supersequence of the strings $\alpha^0_{(i-1)2^r+1}, \ldots, \alpha^0_{(i-1)2^r+2^{r-1}}$, and α^{r-1}_{2i} is a supersequence of the strings $\alpha^0_{(i-1)2^r+2^{r-1}+1}, \ldots, \alpha^0_{i2^r}$, it follows that

$$l^{r-1}_{2i-1,2i} \geq \max_{\substack{1 \leq u \leq 2^{r-1} \\ 1 \leq v \leq 2^{r-1}}} m_{(i-1)2^r+u,(i-1)2^r+2^{r-1}+v}.$$

and so

$$l^{r-1}_{2i-1,2i} \geq \frac{1}{2^{2r-2}} \sum_{u=1}^{2^{r-1}} \sum_{v=1}^{2^{r-1}} m_{(i-1)2^r+u,(i-1)2^r+2^{r-1}+v}. \tag{5.4}$$

Combining (5.3) and (5.4), we get

$$t \leq kn - X \tag{5.5}$$

where

$$X = \sum_{r=1}^{p} \frac{1}{2^{2r-2}} \sum_{i=1}^{2^{p-r}} \sum_{u=1}^{2^{r-1}} \sum_{v=1}^{2^{r-1}} m_{(i-1)2^r+u,(i-1)2^r+2^{r-1}+v}. \tag{5.6}$$

Manipulation of (5.6) gives

$$X = \frac{1}{2^{2p-2}} \sum_{1 \leq u < v \leq 2^p} m_{u,v} + \sum_{r=1}^{p-1} \frac{3}{2^{2p-2r}} \sum_{i=0}^{2^r-1} \sum_{i2^{p-r}+1 \leq u < v \leq (i+1)2^{p-r}} m_{u,v}. \tag{5.7}$$

This may be seen by noting that, in both (5.6) and (5.7) the coefficient of $m_{u,v}$ is $1/2^{2x}$, where x is such that 2^x divides some number in the closed interval $[u, v-1]$ but 2^{x+1} does not.

We now seek a lower bound, in terms of $x_1, \ldots, x_{2^p}$, for the two expressions involving $\sum m_{u,v}$ in (5.7). The first is easy, namely

$$\sum_{1 \leq u < v \leq 2^p} m_{u,v} = \sum_{j=2}^{2^p} \binom{j}{2} x_j. \tag{5.8}$$

This is because x_j counts the number of positions in α that represent symbols in exactly j of the strings for the chosen embeddings. For any particular choice of exactly j strings, there are $\binom{j}{2}$ choices of $m_{u,v}$ that contribute.

The lower bound for

$$\sum_{i=0}^{2^r-1} \sum_{i2^{p-r}+1 \leq u < v \leq (i+1)2^{p-r}} m_{u,v} \tag{5.9}$$

in terms of the x_j is less obvious. Recall that x_j counts the number of positions in the SCS α that correspond to symbols in exactly j of the original strings. So we ask, for any particular j strings, what is the smallest number of terms in (5.9) that count the positions in α at which just these j strings are represented? This smallest number will arise when the indexes of the j strings are as evenly distributed as possible in the 2^r intervals $[i2^{p-r}+1,(i+1)2^{p-r}]$ $(i=0,\ldots,2^r-1)$, and will be precisely

$$\sum_{w=0}^{2^r-1}\binom{y_w}{2} \tag{5.10}$$

where y_w is the number of indexes in the wth interval.

But in a typical case of even distribution, there will be $\lfloor \frac{j}{2^r} \rfloor$ indices in the first interval, $\lfloor \frac{j+1}{2^r} \rfloor$ indices in the second interval, ..., and $\lfloor \frac{j+2^r-1}{2^r} \rfloor$ indices in the 2^rth interval, and these values of y_w, together with (5.10), establish

$$\sum_{i=0}^{2^r-1}\sum_{i2^{p-r}+1\le u<v\le(i+1)2^{p-r}} m_{u,v} \ge \sum_{j=1}^{2^p}\sum_{w=0}^{2^r-1}\binom{\lfloor \frac{j+w}{2^r} \rfloor}{2} x_j. \tag{5.11}$$

From (5.7), (5.8), (5.9) and (5.11) we obtain

$$X \ge \sum_{i=1}^{2^p}\left\{\frac{1}{2^{2p-2}}\binom{i}{2}+\sum_{r=1}^{p-1}\frac{3}{2^{2p-2r}}\sum_{w=0}^{2^r-1}\binom{\lfloor \frac{i+w}{2^r} \rfloor}{2}\right\}x_i. \tag{5.12}$$

So, combining (5.1), (5.2), (5.3), (5.5) and (5.12)

$$\frac{t}{s} \le \frac{\sum_{i=1}^{k} c_i x_i}{\sum_{i=1}^{k} x_i}. \tag{5.13}$$

where

$$c_i = i - \frac{1}{2^{2p-2}}\binom{i}{2} - \sum_{r=1}^{p-1}\frac{3}{2^{2p-2r}}\sum_{w=0}^{2^r-1}\binom{\lfloor \frac{i+w}{2^r} \rfloor}{2}. \tag{5.14}$$

It is immediate from (5.13) that

$$\frac{t}{s} \le \max_{1\le i\le k} c_i. \tag{5.15}$$

It is not hard to show that $c_i < c_{i+1}$ if $i \le 2^{p-1}-1$, and $c_i > c_{i+1}$ if $i \ge 2^{p-1}$, so that the maximum value of c_i occurs when $i = 2^{p-1}(=\frac{k}{2})$. Substituting in (5.14) and simplifying gives $\frac{t}{s} \le \frac{3k+2}{8}$, and the proof of the theorem is complete. □

5.4.2 Bad examples for the Tournament Algorithm

We now show that the Tournament Algorithm cannot guarantee to return a supersequence within a factor better than $O(k)$ of the optimal.

Theorem 5.4.2 *For any even $k \geq 2$, there is a set of k strings for which*

$$\frac{t}{s} \geq \frac{k+2}{4}$$

where s is the length of an SCS and t the length of a common supersequence found by applying the Tournament Algorithm.

Proof
We give an example with $k = 2q$ strings of length $n = 2^{k/2-1} = 2^{q-1}$ over the alphabet $\Sigma^q = \{0, 1, \ldots, 2^q - 1\}$. Let α^q be the string of length 2^q comprising the symbols of Σ^q in natural order.

The set P_q of strings is defined as follows. For $i = 1 \ldots q$ let $\alpha^{0,q}_{2i-1}$ be the subsequence of α^q of length 2^{q-1} containing all and only those symbols whose i^{th} least significant bit is 1 and let $\alpha^{0,q}_{2i}$ be the subsequence of α^q of length 2^{q-1} containing all and only those symbols whose i^{th} least significant bit is 0.

Every $\alpha^{0,q}$ is a subsequence of α^q and every symbol of α^q appears in $k/2$ of the $\alpha^{0,q}$'s so α^q of length 2^q is a supersequence of all the strings in P_q.

In the first round of the tournament, $\alpha^{0,q}_{2i-1}$ is matched with $\alpha^{0,q}_{2i}$ for $i = 1 \ldots k/2$ and a feasible outcome is $\alpha^{1,q}_i$ = the permutation of α^q obtained from α^q by flipping the i^{th} least significant bit.

Claim 5.4.1 *Executing the Tournament Algorithm on the above set of strings can return a supersequence of length*

$$t = f(k) = 2^{k/2-2}(k+2).$$

The proof of Claim 5.4.1 is by induction. The base case is when $k = 2$. The two strings are $\alpha^{0,1}_1 = 1$ and $\alpha^{0,1}_2 = 0$. The Tournament Algorithm will return the supersequence $\alpha^1_{1,1} = 10$ which has length 2 thus establishing the base case.

We will see that if the claim is true for $k-2$ strings then it is true for k strings. To study the behaviour of the algorithm beyond the first round, some more definitions will be helpful. Define $\Sigma^{q+} = \{2^q, 2^q + 1, \ldots, 2^{q+1} - 1\}$ and let α^{q+} be the string of length 2^q comprising the symbols of Σ^{q+} in natural order. Let $\alpha^r_{i,q+} = \alpha^r_{i,q}$ where every symbol a is replaced by the symbol $(a + 2^q)$.

For P_q, it is easy to verify that for $i = 1 \ldots q - 1$,

$$\alpha^{1,q}_i = \alpha^{1,q-1}_i \# \alpha^{1,(q-1)+}_i,$$

and

$$\alpha_q^{1,q} = \alpha^{(q-1)+} + \alpha^{q-1}$$

where $+$ denotes concatenation.

In subsequent rounds of the tournament, $\alpha_q^{1,q}$ will be given a 'bye' until the number of strings in round r, say, is even. Until then the other strings will behave as two independent, isomorphic sets of strings, each equivalent to the case where $k = 2q - 2$. When, in round r, the number of strings is even an arbitrary SCS of

$$\alpha_{\lceil k/2^{r-1} \rceil - 1}^{r-1,q} = \alpha_{\lceil k/2^{r-1} \rceil}^{r-1,q-1} + \alpha_{\lceil k/2^{r-1} \rceil}^{r-1,(q-1)+}$$

and

$$\alpha_{\lceil k/2^{r-1} \rceil}^{r-1,q} = \alpha^{(q-1)+} + \alpha^{q-1}$$

will form the string

$$\alpha_{\lceil k/2^r \rceil}^{r,q}.$$

It is easy to see that the string

$$\alpha^{(q-1)+} + \mathrm{SCS}(\alpha^{q-1}, \alpha_{\lceil k/2^r \rceil}^{r-1,q-1}) + \alpha_{\lceil k/2^r \rceil}^{r,(q-1)+} = \alpha^{(q-1)+} + \beta_{\lceil k/2^r \rceil}^{r,q-1} + \alpha_{\lceil k/2^r \rceil}^{r,(q-1)+}$$

where $\mathrm{SCS}(\gamma, \delta)$ represents an arbitrary SCS of strings γ and δ, and $\beta_{\lceil dk/2^r \rceil}^{r,q-1}$ represents an arbitrary supersequence of $\alpha_{\lceil k/2^r \rceil}^{r,q-1}$, is a feasible result for $\alpha_{\lceil k/2^r \rceil}^{r,q}$.

For rounds $u = (r+1) \ldots p$ of the tournament, the string

$$\alpha_{\lceil k/2^r \rceil}^{r,q} = \alpha^{(q-1)+} + \beta_{\lceil k/2^r \rceil}^{r,q-1} + \alpha_{\lceil k/2^r \rceil}^{r,(q-1)+}$$

will either receive a 'bye' to form $\alpha_{\lceil k/2^u \rceil}^{u,q}$ or be matched with the string

$$\alpha_{\lceil k/2^{u-1} \rceil - 1}^{u-1q} = \alpha_{\lceil (k-2)/2^{u-1} \rceil - 1}^{u-1,q-1} + \alpha_{\lceil (k-2)/2^{u-1} \rceil - 1}^{u-1,(q-1)+}.$$

A feasible SCS generated from this match is

$$\alpha^{(q-1)+} + SCS(\beta_{\lceil k/2^{u-1} \rceil}^{u-1,q-1}, \alpha_{\lceil k/2^{u-1} \rceil - 1}^{u-1,q-1}) + SCS(\alpha_{\lceil k/2^{u-1} \rceil}^{u-1,(q-1)+}, \alpha_{\lceil k/2^{u-1} \rceil - 1}^{u-1,(q-1)+})$$

since

$$|\mathrm{LCS}(\alpha^{(q-1)+} + \alpha_{\lceil k/2^{u-1} \rceil}^{u-1,(q-1)+}, \alpha_{\lceil k/2^{u-1} \rceil - 1}^{u-1,(q-1)+})| \leq$$

$$|\mathrm{LCS}(\beta_{\lceil k/2^{u-1} \rceil}^{u-1,q-1}, \alpha_{\lceil k/2^{u-1} \rceil - 1}^{u-1,q-1})| + |\mathrm{LCS}(\alpha_{\lceil k/2^{u-1} \rceil}^{u-1,(q-1)+}, \alpha_{\lceil k/2^{u-1} \rceil - 1}^{u-1,(q-1)+})|$$

where $\mathrm{LCS}(\gamma, \delta)$ represents an arbitrary LCS of the strings γ and δ.

Hence the length $t = f(k)$ of the final string

$$\alpha_1^{p,q} = \alpha^{(q-1)+} + \beta_1^{p,q-1} + \alpha_1^{p,(q-1)+}$$

will be $\geq |\alpha^{q-1}| + 2f(k-2) = 2^{k/2-1} + 2f(k-2)$. Solving the recurrence relation where the base case is $f(2) = 2$ gives

$$t \geq (k+2)2^{k/2-2}$$

proving Claim 5.4.1.

Since we know that $s = 2^{k/2}$, the theorem is proved. □

5.4.3 Bad examples for the Tournament Algorithm on an alphabet of fixed size

Theorem 5.4.3 *For k an arbitrary power of 2, there is a set of k strings over an alphabet of size 2 for which*

$$\frac{t}{s} \geq \frac{1}{2}(\log_2 k + 1)$$

where s is the length of an SCS and t the length of a common supersequence found by applying the Tournament Algorithm.

Proof

Consider the instance involving $k = 2^p$ strings all of length 1 over the alphabet $\{a, b\}$ defined by $\alpha^0_{2i-1} = a$, $\alpha^0_{2i} = b$ for $1 \leq i \leq 2^{p-1}$. It is easy to see that a feasible application of the Tournament Algorithm leads to the following:

$$\alpha^{2j}_{2i-1} = (ab)^j a$$

$$\alpha^{2j-1}_{2i-1} = (ab)^j$$

$$\alpha^{2j}_{2i} = (ba)^j b$$

$$\alpha^{2j-1}_{2i} = (ba)^j.$$

Hence, after the rth round of the tournament, all of the strings have length $r+1$, and, in particular, the common supersequence generated by the algorithm has length $p + 1 = \log_2 k + 1$. But a shortest common supersequence has length 2, and the proof of the theorem is complete. □

5.5 The Majority-Merge Algorithm

The Majority-Merge Algorithm applied to k strings of length n proceeds as follows. Initialise the supersequence to the empty string. Let σ be the most common character of the leftmost characters of the remaining strings. Ties are decided arbitrarily. Append σ to the supersequence and remove it from the front of the strings of which

it is a prefix. Repeat this process until all the strings are exhausted. In a naive implementation, each step requires at most time proportional to the number of strings. Since at least one string becomes shorter at each merge, the number of merges is limited by kn, the total number of characters in all the strings. The worst-case time complexity is therefore $O(k^2n)$. Careful implementation making use of a heap can give a time complexity of $O(kn \log z)$ for an alphabet of size z. For an unbounded alphabet this is then $O(kn \log(kn))$ in the worst case.

5.5.1 Worst-case behaviour of the Majority-Merge Algorithm

Theorem 5.5.1 *For a given set of k strings, denote by s the length of an SCS, and by m the length of a common supersequence returned by the Majority-Merge Algorithm. Then*

$$\frac{m}{s} \leq k.$$

Proof
The length of the SCS is at least n so it suffices to show that the length of the supersequence returned by the algorithm is not greater than kn. At each merge, one character is added to the supersequence and at least one character is removed from the set of strings. Since there are a total of kn characters in the strings, the supersequence returned cannot have length greater than kn. □

5.5.2 Bad examples for the Majority-Merge Algorithm

We now show that for a particular set of strings, the algorithm will simply concatenate the strings.

Theorem 5.5.2 *There is a set of k strings for which*

$$\frac{m}{s} \geq \frac{kn}{n+k-1}$$

where s is the length of an SCS and m the length of a common supersequence found by applying the Majority-Merge Algorithm.

Proof
Consider the set of k strings, $\alpha_1, \ldots, \alpha_k$, of length n over the alphabet $\{a_1, a_2, \ldots, a_k\}$ of size k, with $\alpha_i = a_i(a_1)^{n-1}$ for $1 \leq i \leq k$. In the first merge, there is one occurrence of every character in the alphabet so only one can be selected. We take it that the first character of α_1 is selected. This leads to a similar situation with no majority character and this repeats until α_1 has been appended to the supersequence without

using any of the other strings. Again a similar situation arises and α_2 is appended without using any of the other strings. This continues to arise so that the strings are appended individually to the supersequence in any order, after the first, giving a supersequence of length kn. It is clear that an SCS of $\alpha_1, \ldots, \alpha_k$ is $a_1 a_2 \ldots a_k a_1^{n-1}$ which has length $n + k - 1$. □

By increasing the value of n it is possible to provide a set of strings for which the Majority-Merge Algorithm will return a supersequence with length arbitrarily close to a factor of k of the optimal. The lower bound can therefore be brought arbitrarily close to the upper bound and this completes characterisation of the worst-case behaviour of the Majority-Merge Algorithm on an unbounded alphabet.

5.5.3 Worst-case behaviour of the Majority-Merge Algorithm on an alphabet of fixed size

Theorem 5.5.3 *For a given set of k strings over a binary alphabet,*

$$\frac{m}{s} \leq (\log_2 k + 1)$$

where s is the length of an SCS and m the length of a supersequence returned by the Majority-Merge Algorithm.

Proof
The length of the SCS is at least n so it suffices to show that the length of the supersequence returned by the algorithm is not greater than $(\log_2 k + 1)n$. The total number of characters to be merged is kn. Let c_i denote the number of characters remaining and l_i denote the number of non-empty strings prior to step i of the algorithm. Clearly $c_1 = kn$ and $l_1 = k$. The number of characters under consideration at step i is clearly equal to l_i. Since there are only 2 possibilities for each character, the number of characters merged at step i is equal to at least $\lceil l_i/2 \rceil$.

Over the n steps $i, \ldots, i + n - 1$ of the algorithm, the number of characters considered from each string, including repeated consideration of the same character, is at least equal to the string's length since no string has more than n characters and a character from every un-exhausted string is considered in each of the n steps. Thus the total number of characters considered for merging is at least equal to the number of characters remaining prior to step i, i.e.

$$\sum_{j=1}^{n} l_{i+j-1} \geq c_i.$$

As stated above, the number of characters merged at each step is at least half that under consideration, so over the n steps $i, \ldots, i + n - 1$ the total number of characters merged will be at least $\frac{1}{2}c_i$. So after any arbitrary n steps, the number

of characters remaining will be at most half that before the n steps. It follows easily then that after $(\log_2 k)n$ steps, kn characters will be reduced to at most n characters, which will require at most a further n steps to be merged. Therefore at most $\log_2 k + 1$ groups of n steps will be executed before all the characters are merged. Since one character is added to the supersequence for each step, the length of the supersequence cannot be greater than $(\log_2 k + 1)n$. □

5.5.4 Bad examples for the Majority-Merge Algorithm on an alphabet of fixed size

In Section 5.5.2 bad examples for the Majority-Merge Algorithm were constructed using an alphabet of unbounded size. Here we show that when the alphabet size is limited to 2, the algorithm still cannot guarantee to find a common supersequence within a constant factor of the optimal.

Theorem 5.5.4 *For k an arbitrary power of 2, there is a set of k strings over an alphabet of size 2 for which*

$$\frac{m}{s} \geq \frac{(\log_2 k + 1)n - \frac{1}{2}\log_2^2 k + \frac{1}{2}\log_2 k}{n + \log_2 k}$$

where s is the length of an SCS and m the length of a supersequence returned by the Majority-Merge Algorithm.

Proof
Consider the following instance involving $k = 2^p$ strings of length $n \geq p$ over the alphabet $\{a, b\}$. For $1 \leq i \leq k$, $\alpha_i = b^{x_i}a^{n-x_i}$ where $x_i = \log_2 k - \lceil \log_2(k - i + 1) \rceil$. It may be verified that that the first n steps will merge the first $k/2$ strings, which are all equal to a^n, into n elements in the supersequence, if the ties are resolved appropriately. Following this, one 'column' of $b's$ prefixing all the un-exhausted strings will be merged. This will give rise to a situation analogous to that at the start but where now the strings have length $n - 1$. Now if l is the number of strings remaining and c the length of these strings, the first $l/2$ strings will be merged into c elements in the supersequence and then one column of $b's$ prefixing the now $l/2$ remaining un-exhausted strings will be merged into one element in the supersequence. This process will repeat until one string remains un-exhausted and this will be merged into $n - \log_2 k$ elements. Thus the supersequence will contain $\sum_{x=0}^{\log_2 k}(n - x)$ a's and $\log_2 k$ b's. Hence the length of the supersequence generated will be

$$m = \sum_{x=0}^{\log_2 k}(n - x) + \log_2 k$$

$$
\begin{aligned}
&= \sum_{x=0}^{\log_2 k} n - \sum_{x=0}^{\log_2 k} x + \log_2 k \\
&= (\log_2 k + 1)n - \frac{1}{2}(\log_2^2 k) + \frac{1}{2}\log_2 k.
\end{aligned}
$$

An SCS for the given strings is clearly $b^{\log_2 k}a^n$, of length $n + \log_2 k$, and the proof is complete. □

By increasing the value of n it is possible to provide a set of strings for which the Majority-Merge Algorithm will return a supersequence with length arbitrarily close to a factor of $\log_2 k$ of the optimal. The lower bound can therefore be brought arbitrarily close to the upper bound and this completes characterisation of the worst-case behaviour of the Majority-Merge Algorithm on an alphabet of size 2.

We can generalise the strategy in the proof of the above theorem to alphabets of size greater then 2. For the alphabet $\{a_1, \ldots, a_z\}$ of size z and $k \geq z$ (the structure is clearest when k is a power of z), let

$$
\alpha_i = a_z^{\mathcal{C}(z,k,i)} a_{z-1}^{\mathcal{B}(z,z-1,k,i)} a_{z-2}^{\mathcal{B}(z,z-2,k,i)} \ldots a_1^{\mathcal{A}(z,k,i,n)}, \qquad \text{for } 1 \leq i \leq k.
$$

where

$$
\begin{aligned}
\mathcal{C}(z,k,i) &= 0 \qquad \text{if } (i \leq \lceil k(z-1)/z \rceil) \\
&= 1 + \mathcal{C}(z, k - \lceil k(z-1)/z \rceil, i - \lceil k(z-1)/z \rceil) \qquad \text{otherwise} \\
\mathcal{B}(z,j,k,i) &= 0 \quad \text{if } (i \leq \lceil k(j-1)/z \rceil) \text{ or } (\lceil kj/z \rceil < i \leq \lceil k(z-1)/z \rceil) \\
&= 1 \quad \text{if } (\lceil k(j-1)/z \rceil < i \leq \lceil kj/z \rceil) \\
&= \mathcal{B}(z,j,k - \lceil k(z-1)/z \rceil, i - \lceil k(z-1)/z \rceil) \qquad \text{otherwise} \\
\mathcal{A}(z,k,i,n) &= n \qquad \text{if } (i \leq \lceil k/z \rceil) \\
&= (n-1) \qquad \text{if } (\lceil k/z \rceil < i \leq \lceil k(z-1)/z \rceil) \\
&= \mathcal{A}(z, k - \lceil k(z-1)/z \rceil, i - \lceil k(z-1)/z \rceil, n-1) \quad \text{otherwise.}
\end{aligned}
$$

The length of the supersequence generated by an application of Majority-Merge, analogous to the above, is

$$
(z-1)(n \log_z k - \frac{\log_z k(\log_z k - 1)}{2}) + n
$$

and the length of the SCS is $n + \log_z k + (z-2)$. As n tends to infinity, the error in the approximation tends to $O((z-1)\log_z k)$. When $k = z$, the above strings reduce to the bad case in Section 5.5.2 for an unbounded alphabet.

Calling it "Algorithm M3", Foulser, Li and Yang [17] also analysed the worst-case behaviour of the Majority-Merge algorithm for a binary alphabet, focussed on the special case of n strings of length n. For that case they used a different method

to show that Majority-Merge guarantees to return a sequence of length $O(n \log n)$. They used essentially the bad example above to prove that, in that special case, it can return a sequence of length $\Omega(n \log n)$.

5.5.5 "Algorithm M4" of Foulser, Li and Yang

Foulser, Li and Yang [17] analysed an algorithm which they called "Algorithm M4", a hybrid of the Majority-Merge algorithm and the trivial algorithm described in Section 5.1. Algorithm M4 proceeds as follows. Let Ψ be the set of leftmost symbols of the remaining strings. Merge every symbol in Ψ, in order of decreasing frequency in the leftmost positions, removing the symbol from the leftmost position of each string where it appears and recalculating the frequencies of the symbols in Ψ after each merge. When every symbol in Ψ has been merged, form a new Ψ from the current leftmost symbols. Repeat this process until all the strings are exhausted. They showed that for a fixed alphabet size, M4 has the same worst-case guarantee as the trivial algorithm and that it has the same average case behaviour as the Majority-Merge algorithm. It is easy to show that in the worst case, Algorithm M4 behaves as badly as the trivial algorithm for a fixed alphabet. For an unbounded alphabet it is easy to show that, as for the Majority-Merge algorithm, no worst-case guarantee beyond the trivial factor of k is possible. They showed empirically that "Algorithm M4" has similar performance to Majority-Merge in practice.

5.6 The Algorithm Greedy1

The algorithm Greedy1 applied to k strings $\alpha_1, \ldots, \alpha_k$ of length n proceeds as follows. There are $k-1$ steps. Define $\beta_1 = \alpha_1$. In the ith step, an arbitrary SCS of the strings α_{i+1} and β_i is found and labelled β_{i+1}. Hence β_i is a common supersequence of the strings $\alpha_1, \ldots, \alpha_i$, and in particular β_k is a supersequence of all k strings.

It is easy to verify that, using a standard $O(nm)$ algorithm for the SCS of two strings, Greedy1 for k strings of length n has $O(k^2n^2)$ time complexity.

5.6.1 Worst-case behaviour of Greedy1

Theorem 5.6.1 *For a given set of k strings, denote by s the length of an SCS, and by g_1 the length of a common supersequence returned by Greedy1. Then*

$$\frac{g_1}{s} \leq \frac{k-1}{e} + 1.$$

Proof
As before, consider an SCS α of the strings $\alpha_1, \ldots, \alpha_k$, together with a particular

embedding of each of the k individual strings in α. For $i = 1, 2, \ldots, k$, let x_i denote the number of positions in α that correspond to symbols in exactly i of the individual strings in the chosen embedding. Then equations (5.1) and (5.2) hold as before.

Also, it is easy to see that

$$g_1 = kn - \sum_{r=1}^{k-1} l_r \tag{5.16}$$

where l_r is the length of a longest common subsequence of the strings α_{r+1} and β_r.

Let m_{ij} $(1 \leq i < j \leq k)$ be the number of positions in α that correspond to symbols in both α_i and α_j for the chosen embeddings.

Since β_r is a supersequence of the strings $\alpha_1, \ldots, \alpha_r$, it follows that

$$l_r \geq \max_{1 \leq i \leq r} m_{i,r+1}$$

and so,

$$l_r \geq \frac{1}{r} \sum_{i=1}^{r} m_{i,r+1}. \tag{5.17}$$

Combining (5.16) and (5.17) we get

$$g_1 \leq kn - X \tag{5.18}$$

where

$$X = \sum_{r=1}^{k-1} \frac{1}{r} \sum_{i=1}^{r} m_{i,r+1}. \tag{5.19}$$

We now seek a lower bound, in terms of $x_1, \ldots, x_k$, for the expression on the right hand side of (5.19). Recall that x_j counts the number of positions in the SCS α that correspond to symbols in exactly j of the original strings. So we ask, for any choice of j strings, how small the sum of coefficients of terms in (5.19) can be, terms that count the positions in α at which just these j strings are represented. This smallest sum will clearly arise for the particular j strings $\alpha_{k-j+1}, \alpha_{k-j+2}, \ldots, \alpha_k$, and will have value

$$\sum_{l=k-j+1}^{k-1} \frac{l-k+j}{l} = j - 1 - \sum_{l=k-j+1}^{k-1} \frac{k-j}{l}.$$

So

$$X \geq \sum_{j=1}^{k} \left(j - 1 - \sum_{l=k-j+1}^{k-1} \frac{k-j}{l} \right) x_j. \tag{5.20}$$

Combining (5.1), (5.2), (5.18) and (5.20) gives

$$\frac{g_1}{s} \leq \frac{\sum_{i=1}^{k} (1 + \sum_{l=k-i+1}^{k-1} \frac{k-i}{l}) x_i}{\sum_{i=1}^{k} x_i}. \tag{5.21}$$

It is immediate from (5.21) that

$$\begin{aligned}\frac{g_1}{s} &\leq 1 + \max_{1\leq i\leq k} \sum_{l=k-i+1}^{k-1} \frac{k-i}{l} \\ &= 1 + \max_{1\leq j\leq k-1} j \sum_{l=j+1}^{k-1} \frac{1}{l} \\ &= 1 + \max_{1\leq j\leq k-1} j(H_{k-1} - H_j) \\ &\leq 1 + \max_{1\leq j\leq k-1} j(\log(k-1) - \log j).\end{aligned}$$

By calculus, the maximum value of $x(K - \log x)$ occurs when $\log x = K - 1$, i.e., when $x = e^{K-1} = \frac{k-1}{e}$ $(e = 2.718\ldots)$. Hence

$$\frac{g_1}{s} \leq 1 + \frac{k-1}{e}(\log(k-1) - \log(k-1) + 1) = \frac{k+e-1}{e}.$$

This completes the proof of the theorem. □

5.6.2 Bad examples for Greedy1 on an alphabet of fixed size

Algorithm Greedy1 differs from the Tournament in that it appears to behave as badly for a bounded alphabet as for an unbounded alphabet. In fact, in the worst case, the ratio of the greedy solution to the optimal solution is at least a constant times k, for any alphabet.

Theorem 5.6.2 *For any $k \geq 2$, there is a set of k strings over an alphabet of size 2 for which*

$$\frac{g_1}{s} \geq \frac{k+3}{8}$$

where s is the length of an SCS and g_1 the length of a common supersequence found by applying Greedy1.

Consider the set of k strings of length $n = 2^k$ over the alphabet $\{a, b\}$ defined by $\alpha_i = (b^{2^{k-i}} a^{2^{k-i}})^{2^{i-1}}$, for $1 \leq i \leq k$.

Lemma 5.6.1 *For $1 \leq i < j \leq k$*
(i) $|LCS(\alpha_i, \alpha_j)| = 2^{k-1}(1 + 2^{i-j})$.
(ii) $|LCS(\alpha_i, \alpha_j)| \leq \frac{3n}{4}$.

Proof
(i) A particular common subsequence, of length $2^{k-1}(1+2^{i-j})$, of α_i and α_j consists of all of the 2^{k-1} a's in each string and as many b's as possible. The number of b's that can be included is equal to the number of blocks of b's in α_i (namely 2^{i-1})

multiplied by the size of each block of b's in α_j (namely 2^{k-j}). To see that this is an LCS, note that α_i consists of 2^i blocks of identical symbols, each block of length 2^{k-i}. Suppose that an LCS of α_i and α_j has x_m symbols from block m ($m = 1, \ldots, 2^i$). To obtain x_m identical symbols from α_j we must exclude at least $x_m - 2^{k-j}$ copies of the other symbol. So

$$\sum_{m=1}^{2^i} (x_m + x_m - 2^{k-j}) \leq 2^k,$$

so that

$$2|LCS| - 2^{k+i-j} \leq 2^k,$$

from which the result follows at once.

(ii) Follows from (i) and the fact that $n = 2^k$. □

Proof

(**of Theorem 5.6.2**). The particular bad case occurs for the set of strings described above. We show by induction that at stage i ($1 \leq i \leq k-1$) of the algorithm, if Greedy1 merges an accumulating supersequence β_i with α_{i+1} to form β_{i+1}, and $\beta_1 = \alpha_1$ then a possible value for β_i for $1 \leq i \leq k$ is

$$\delta_{i,k} = b^{2^{k-1}} a^{2^{k-i}} \bigoplus_{j=1}^{2^{i-1}-1} (b^{\mathcal{F}(i,j)} a^{2^{k-i}})$$

where

$$\mathcal{F}(i,j) = 2^{k-i} \max\{2^p : 2^p | j\}$$

and $\bigoplus$ denotes concatenation.

The base case is $\beta_1 = \alpha_1 = b^{2^{k-1}} a^{2^{k-1}} = \delta_{1,k}$. Now we show that if $\delta_{i,k}$ gives a possible value for β_i, $i \geq 1$ then $\delta_{i+1,k}$ gives a possible value for β_{i+1}. So we find an SCS of β_i and α_{i+1} to get β_{i+1} as follows:

$$\beta_i = b^{2^{k-1}} a^{2^{k-i}} \bigoplus_{j=1}^{2^{i-1}-1} b^{\mathcal{F}(i,j)} a^{2^{k-i}};$$

$$\alpha_{i+1} = (b^{2^{k-(i+1)}} a^{2^{k-(i+1)}})^{2^{(i+1)-1}}.$$

We claim that an LCS of β_i and α_{i+1} is $\gamma_i = (b^{2^{k-i-1}} a^{2^{k-i}})^{2^{i-1}}$. Firstly, γ_i is a subsequence of β_i since they have an equal number (namely 2^{i-1}) of blocks of the form $b^x a^y$ for some x, y, and in each block, β_i has at least as many (namely 2^{k-i}) b's and an equal number (namely 2^{k-i}) of a's. Also, γ_i is a subsequence of α_{i+1} since,

expressed differently,

$$\alpha_{i+1} = (b^{2^{k-i-1}} a^{2^{k-i-1}} b^{2^{k-i-1}} a^{2^{k-i-1}})^{2^{i-1}}$$

which is clearly a supersequence of $(b^{2^{k-i-1}} a^{2^{k-i-1}} a^{2^{k-i-1}})^{2^{i-1}} = \gamma_i$. By precisely the same argument as was used to prove Lemma 5.6.1, where β_i and α_{i+1} replace α_i and α_j respectively, γ_i can be shown to be an LCS of β_i and α_{i+1}.

We can now build the unique SCS of β_i and α_{i+1} constructible from γ_i. All the a's of β_i and α_{i+1} have been matched and each block of a's in β_i has a block of length 2^{k-i-1} of b's inserted half way along it. Hence

$$\begin{aligned} SCS(\beta_i, \alpha_{i+1}) &= b^{2^{k-1}} a^{2^{k-i-1}} b^{2^{k-i-1}} a^{2^{k-i-1}} \bigoplus_{j=1}^{2^i-2} b^{\mathcal{F}(i,j)/2} a^{2^{k-i-1}} \\ &= b^{2^{k-1}} a^{2^{k-i-1}} \bigoplus_{j=1}^{2^i-1} b^{\mathcal{F}(i+1,j)} a^{2^{k-i-1}} \\ &= \beta_{i+1}. \end{aligned}$$

So the supersequence generated by the algorithm is

$$\beta_k = b^{2^{k-1}} a \bigoplus_{j=1}^{2^{k-1}-1} b^{\mathcal{F}(k,j)} a.$$

There are $2^{k-1} = n/2$ b's in the prefix and $2^{k-1} = n/2$ individual a's. The number of b's in the remaining part of the string is

$$\begin{aligned} \sum_{j=1}^{2^{k-1}-1} \max\{2^p : 2^p | j\} &= \sum_{l=1}^{k-1} 2^{k-1-l} 2^{l-1} \\ &= (k-1)2^{k-2} \\ &= \frac{(k-1)n}{4} \qquad \text{since } n = 2^k. \end{aligned}$$

The total length of the supersequence is therefore $n + (k-1)(n/4) = n(k+3)/4$. The string $(ba)^n$ is a common supersequence, so the length of the shortest common supersequence is $\leq 2n$, and Greedy cannot guarantee to be within a factor better than $(n(k+3)/4)/2n = (k+3)/8$ of the optimal. This completes the proof of the theorem. □

We can generalise the strategy in the proof of the above theorem to alphabets of size greater than 2. For an alphabet $\{a_1, ..., a_z\}$ of size z, let $n = z^k$, and

$$\alpha_i = (\bigoplus_{j=0}^{z-1} a_{z-j}^{z^{k-i}})^{z^{i-1}}, \quad \text{for } 1 \leq i \leq k.$$

The length of the supersequence generated by an analogous application of Greedy1 is

$$n + (k-1)\frac{(z-1)^2}{z^2}n$$

and the length of the shortest common supersequence cannot be greater than nz. Therefore the error in the approximation is $\geq 1/z + (k-1)(z-1)^2/z^3$, which is maximised for $z = 3$, when $k > 8$, leading to a lower bound of $(4k+5)/27$.

5.7 The Centre of the Star Algorithm

The Centre of the Star Algorithm applied to a set of k strings proceeds as follows. Find all the pairwise SCS lengths between the strings and represent this information as a weighted graph G in which the strings are vertices and the weight of an edge connecting α_i and α_j is $s_{i,j} - n$ where $s_{i,j}$ is the length of the SCS of α_i and α_j. Find the string α_c such that the *star graph* centred on α_c and including all the strings of G has minimum weight. A star graph is a graph in which one vertex is connected to every other vertex but no two other vertices are connected. Form a common supersequence by independently aligning every string with α_c. This will always be possible because a star graph is acyclic. The length of the supersequence generated is n plus the sum of the weights of the edges in the star graph. Using an $O(n^2)$ algorithm for the SCS of two strings of length n, this algorithm has $O(k^2n^2)$ time complexity.

5.7.1 Worst-case behaviour of the Centre of the Star Algorithm

Theorem 5.7.1 *For a given set of k strings, denote by s the length of an SCS, and by c the length of a common supersequence returned by the Centre of the Star Algorithm. Then*

$$\frac{c}{s} \leq \frac{k}{2}.$$

Proof

We can, without loss of generality, assume that α_k is the centre string. Denote the sum of its SCS values by $m = \sum_{i=1}^{k-1} |SCS(\alpha_k, \alpha_i)|$. The length c of the supersequence returned will be equal to n (for α_k) plus the number of extra characters added by

each other string in the alignment. Denote by $\bar{s} = m/(k-1)$, the average SCS of α_k and α_i for $i = 1, \ldots, (k-1)$. The average number of extra characters added by $\alpha_1, \ldots, \alpha_{k-1}$ is then $\bar{s} - n$. Therefore $c = n + (\bar{s} - n).(k-1)$. Clearly s is at least equal to the average SCS value, i.e. $s \geq \bar{s}$. We then have

$$\begin{aligned} \frac{c}{s} &\leq \frac{(m/(k-1) - n).(k-1) + n}{m/(k-1)} \\ &= \frac{m(k-1) - (k-1)^2 n + (k-1)n}{m} \\ &= (k-1) - \frac{(k-1)^2 n}{m} + \frac{(k-1)n}{m}. \end{aligned} \tag{5.22}$$

Clearly $(k-1)n \leq m \leq 2(k-1)n$ therefore $\frac{1}{2(k-1)} \leq \frac{n}{m} \leq \frac{1}{(k-1)}$. So using this range to maximise (5.22) we get,

$$\begin{aligned} \frac{c}{s} &\leq (k-1) - \frac{(k-1)}{2} + \frac{1}{2} \\ &= \frac{k}{2}.\square \end{aligned}$$

5.7.2 Bad examples for the Centre of the Star Algorithm on an alphabet of fixed size

Theorem 5.7.2 *For k an arbitrary even number, there is a set of k strings, over an alphabet of size 2, for which*

$$\frac{c}{s} \geq (k+2)/4$$

where s is the length of an SCS and c the length of a common supersequence found by applying the Centre of the Star Algorithm.

Proof

Consider the following set of k strings, $\alpha_1, \ldots, \alpha_k$, of length 1. For $1 \leq i \leq \lfloor \frac{k}{2} \rfloor$, $\alpha_i = a$ and for $\lfloor \frac{k}{2} \rfloor + 1 \leq i \leq k$, $\alpha_i = b$. For the case of k even, each string could clearly be chosen as the centre string. The supersequence generated will have length one plus the number of characters unmatched with other strings. The strings identical to the centre string will add nothing to the supersequence and the $(k/2)$ strings which do not match the centre string will each add one element to the supersequence. Thus the length of the supersequence will be $(k/2 + 1)$. The SCS is clearly ab which has length 2. Therefore the error ratio is in general $(k/2 + 1)/2 = (k+2)/4$. $\square$

5.8 The Minimum Spanning Tree Algorithm

The Minimum Spanning Tree Algorithm proceeds as follows. Build a weighted graph G as for the Centre of the Star algorithm. Now find a minimum weight spanning tree of G and form a supersequence by aligning each string optimally with its neighbours in the spanning tree. This will always be possible because a minimum spanning tree is guaranteed to have no cycles. The supersequence will have length n plus the sum of the weights of the edges in the spanning tree. The graph can be constructed in $O(k^2n^2)$ time and a minimum spanning tree can be found with Prim's algorithm [55, 59] in $O(k^2)$ time.

5.8.1 Worst-case behaviour of the Minimum Spanning Tree Algorithm

This algorithm is a refinement of the Centre of the Star Algorithm since a star is a spanning tree. It therefore guarantees to return a supersequence of length not greater than that returned by the Centre of the Star Algorithm. This gives us the following theorem.

Theorem 5.8.1 *For a given set of k strings, denote by s the length of an SCS, and by q the length of a common supersequence returned by the Minimum Spanning Tree Algorithm. Then*

$$\frac{q}{s} \leq \frac{k}{2}.$$

5.8.2 Bad examples for the Minimum Spanning Tree Algorithm

Theorem 5.8.2 *For k an arbitrary even number, there is a set of k strings for which*

$$\frac{q}{s} \geq \frac{(k+1)}{4}$$

where s is the length of an SCS and q the length of a common supersequence found by applying the Minimum Spanning Tree Algorithm.

Proof

Recall the bad example, for an unbounded alphabet, for the Tournament algorithm. It comprises $k = 2q$ strings of length $n = 2^{q-1}$ over the alphabet $\Sigma = \{0, 1, \ldots, 2^q - 1\}$. Let α be the string of length 2^q comprising the symbols of Σ in natural order. The set P of strings is defined as follows. For $i = 1, \ldots, q$ let α_{2i-1} be the subsequence of α containing all and only those symbols whose i^{th} least significant bit is 1 and let α_{2i} be the subsequence of α containing all and only those symbols whose i^{th} least significant bit is 0. Each α_i is of length $n = 2^{q-1}$.

Every α_i $(1 \leq i \leq k)$ is a subsequence of α and every symbol of α appears in $k/2$ of the α_i's, so α of length $2n$ is an SCS of all the strings in P.

It is immediate that the length of the longest common subsequence of any two strings in P is not more than $n/2$. Every edge in the spanning tree will therefore have weight no less than $n/2$. From the relationship between the spanning tree and the length of the supersequence returned we get

$$\begin{aligned} q &\geq n + \frac{n}{2}(k-1) \\ &= \frac{n(k+1)}{2}. \end{aligned}$$

Given that $s = 2n$, we conclude that

$$\frac{q}{s} \geq \frac{(k+1)}{4}.\square$$

5.8.3 Bad examples for the Minimum Spanning Tree Algorithm on an alphabet of fixed size

Theorem 5.8.3 *For any $k \geq 2$, there is a set of k strings, over an alphabet of size 2 for which*

$$\frac{q}{s} \geq \frac{(k+3)}{8}$$

where s is the length of an SCS and q the length of a common supersequence found by applying the Minimum Spanning Tree Algorithm.

Proof

Recall the bad example for Greedy1. It comprises k strings of length 2^k over the alphabet $\{a, b\}$, defined by $\alpha_i = (b^{2^{k-i}} a^{2^{k-i}})^{2^{i-1}}$ $(i = 1, \ldots, k)$. From Lemma 5.6.1 we know that the length of the longest common subsequence of any two strings in P is not more than $3n/4$. Every edge in the spanning tree will therefore have weight no less that $n/4$. Therefore

$$\begin{aligned} q &\geq n + \frac{n}{4}(k-1) \\ &= \frac{n(k+3)}{4}. \end{aligned}$$

Given that $\alpha = (ba)^n$ of length $2n$ is a supersequence of the strings in P, we have

$$\frac{q}{s} \geq \frac{(k+3)}{8}.\square$$

5.9 The Algorithm Greedy2

Algorithm Greedy2 applied to k strings, $\alpha_1, \ldots, \alpha_k$, of length n proceeds as follows. Let $\beta_j^1 = \alpha_j$ for $1 \leq j \leq k$. There are $k-1$ steps. In the i^{th} step, an arbitrary SCS of two strings, β_p^e and β_q^f, $p < q$, with the longest LCS is found and labelled β_p^{e+f} (the subscript then represents the smallest numbered α which is guaranteed to be a subsequence of this β). The string β_p^{e+f} replaces β_p^e and β_q^f in the set and we say that β_p^e and β_q^f are *merged*. It follows that β_p^e is a supersequence of e of the α's and in particular that β_1^k is a supersequence of all the α's. It is not hard to verify that using an $O(nm)$ algorithm for the SCS of two strings, Greedy2 for k strings of length n has $O(k^2n^2 \log k)$ time complexity.

5.9.1 Worst case behaviour of Greedy2

The *compression*, c, achieved by an approximation algorithm is a measure of the degree of matching that exists in the alignment it returns and is defined by

$$c = kn - a \tag{5.23}$$

where a is the length of the string returned by the approximation algorithm.

Theorem 5.9.1 *For a given set of k strings, denote by s the length of an SCS, and by g_2 the length of a common supersequence returned by Greedy2. Then*

$$\frac{g_2}{s} \leq \frac{(k+3)}{4}.$$

Proof
Suppose that we knew the length of the LCS of α_j and each of the other α's, and that these $k-1$ numbers were written in descending order. Let v_j^i denote the i^{th} element in this list.

At step i of Greedy2, the compression achieved, c_i, is the length of the LCS of the two strings merged and it is clear that $c = \sum_{i=1}^{k-1} c_i$. In order to obtain a lower bound for the level of compression achieved by Greedy2, the *credit* for the compression at each merge is distributed amongst the strings involved according to the following strategy. When merging β_p^x and β_q^y at step i,

$$\begin{aligned} \beta_p^x \text{ is credited with } \frac{y}{(x+y)}c_i; \\ \beta_q^y \text{ is credited with } \frac{x}{(x+y)}c_i. \end{aligned} \tag{5.24}$$

The compression credited to β_p^x is divided equally among the x α's that we know to be subsequences of β_p^x. A subsequence, α_j, of β_p^x, is therefore credited with $\frac{y}{x(x+y)}c_i$.

Since c_i is the greatest level of compression that can be achieved at step i, and α_j has been merged with a total of $x-1$ other α's, it follows that α_j is credited with $\geq \frac{y}{x(x+y)} v_j^x$ compression.

Suppose α_j is involved in r merges altogether, and that, in the pth of these merges, α_j is a subsequence of some β^{x_p} which is merged with some β^{y_p}. Then $x_1 = 1$, $x_p = x_{p-1} + y_{p-1}$ for $p = 2, \ldots, r$, and $x_r + y_r = k$. The total compression credited to α_j is at least

$$\sum_{p=1}^{r} \frac{y_p}{x_p(x_p + y_p)} v_j^{x_p}.$$

It can be proved by induction that

$$\sum_{p=1}^{s} \frac{y_p}{x_p(x_p + y_p)} v_j^{x_p} \geq \frac{1}{x_s + y_s} \sum_{i=1}^{x_s+y_s-1} v_j^i,$$

for $s = 1, \ldots, r$, and in the case $s = r$ this shows that the compression credited to α_j is

$$\begin{aligned} &\geq \frac{1}{k} \sum_{i=1}^{k-1} v_j^i \\ &= \frac{1}{k} \sum_{1 \leq i,j \leq k,\ i<>j} l_{i,j} \end{aligned}$$

where $l_{i,j}$ represents the length of the LCS of α_i and α_j. The total compression achieved is then

$$\begin{aligned} c &\geq \frac{2}{k} \sum_{1 \leq i < j \leq k} l_{i,j} \\ &\geq \frac{2}{k} \sum_{1 \leq i < j \leq k} m_{i,j} \end{aligned} \tag{5.25}$$

where $m_{i,j}$ represents the number of positions in the SCS represented by symbols from both α_i and α_j. Combining (5.2), (5.23) and (5.25) we get

$$\begin{aligned} g_2 &\leq \sum_{i=1}^{k} i x_i - \frac{2}{k} \sum_{1 \leq i < j \leq k} m_{i,j} \\ &= \sum_{i=1}^{k} (i - \frac{2}{k} \binom{i}{2}) x_i \end{aligned}$$

where the equality holds by the same reasoning as in (5.8), and so

$$g_2 \leq \sum_{i=1}^{k}(i - \frac{i(i-1)}{k})x_i \tag{5.26}$$

where x_i represents the number of positions in an SCS which correspond to symbols in exactly i of the individual strings. Combining (5.26) with (5.1) we get

$$\frac{g_2}{s} \leq \max_i(i - \frac{i(i-1)}{k}).$$

The maximum occurs for

$$i = \frac{k+1}{2}$$

giving

$$\frac{g_2}{s} \leq \frac{k+3}{4}.\square$$

5.9.2 Bad examples for Greedy2 on an alphabet of fixed size

Recall the bad example for Greedy1 in Section 5.6.2. The set of strings used represents a potentially bad case for Greedy2 with precisely the same behaviour as for Greedy1. It is required to show that the behaviour described in Section 5.6.2 represents feasible behaviour of Greedy2 on that set of strings. Recall the definitions of α_i, β_i and γ_i for $(1 \leq i \leq k)$. For $1 \leq i \leq k-1$,

$$\begin{aligned} |\gamma_i| &= 2^{i-1}(2^{k-i-1} + 2^{k-i})) \\ &= 2^{k-2} + 2^{k-1} \\ &= \frac{n}{4} + \frac{n}{2} \qquad \text{since } n = 2^k \\ &= \frac{3n}{4}. \end{aligned}$$

Therefore by Lemma 5.6.1, γ_i is as long as the LCS of any pair of α's in the original set. By an argument very similar to that which was used to prove Lemma 5.6.1, the LCS of β_i and α_j $(j > i+1)$ can be shown to have length not greater than $|\gamma_i|$. Therefore at step i, merging β_i with α_{i+1} is a feasible step for Greedy2.

This gives a result for Greedy2 for a binary alphabet equivalent to the result for Greedy1. Generalising the strategy as for Greedy1 is also effective for Greedy2 to give an equivalent result for Greedy2 over an unbounded alphabet.

LCS Approximation Algorithm	Lower Bound	Upper Bound
Long-Run	z	z
Best-Next for a binary alphabet	n/2	n/2

Table 5.1: Worst-case behaviour of approximation algorithms for the LCS

5.10 Summary of the approximation algorithms for the LCS and SCS problems

The status of the analysis of the worst-case behaviour of the approximation algorithms for the LCS and SCS problems are shown in Tables 5.1 and 5.2 respectively.

5.11 Empirical comparison of the approximation algorithms for the SCS problem

The algorithms for approximating the SCS have been tested to find how close an approximation to the optimal they provide in practice, in particular how varying the three important parameters - the number of strings (k), the length of the strings (n), and the size of the alphabet (z) affects their performance. Two sets of experiments over different conditions were carried out. The first set was over groups of random strings (of length 100). Where possible, rough estimates of the length of the SCS, calculated from the experimental results obtained for Chapter 4, are provided. The second set of experiments used random subsequences (of length 90) of a supersequence (of length 100). In this case it is possible to give an upper bound on the length of the SCS and accurately assess how close the algorithms come to the optimal. In both cases, the results are averaged over four distinct sets of strings. The results are shown in Tables 5.3 and 5.4 respectively. Because Greedy2 subsumes the Centre of the Star and Minimum Spanning Tree algorithms and performs significantly better than them in practice, results for those two algorithms are not presented.

SCS Approximation Algorithm	**Lower Bound**	**Upper Bound**
Tournament		
for unbounded alphabet	$(k+2)/4$	$(3k+2)/8$
for binary alphabet	$(\log_2 k+1)/2$	$(3k+2)/8$
Majority-Merge		
for unbounded alphabet	k	k
for binary alphabet	$\log_2 k$	$\log_2 k$
Greedy1		
for unbounded alphabet	$(4k+5)/27$	$((k-1)/e)+1$
for binary alphabet	$(k+3)/8$	$((k-1)/e)+1$
Centre of the Star		
for unbounded alphabet	$(k+2)/4$	$k/2$
for binary alphabet	$(k+2)/4$	$k/2$
Min. Span. Tree		
for unbounded alphabet	$(k+1)/4$	$k/2$
for binary alphabet	$(k+3)/8$	$k/2$
Greedy2		
for unbounded alphabet	$(4k+5)/27$	$(k+3)/4$
for binary alphabet	$(k+3)/8$	$(k+3)/4$

Table 5.2: Worst-case behaviour of approximation algorithms for the SCS

$n = 100$		Random Strings				
z	k	Tournament	Majority-Merge	Greedy1	Greedy2	Estimate
2	4	145	147	142	142	134
	8	171	154	158	155	141
	12	185	155	167	162	144
	16	200	159	170	167	146
4	4	186	204	184	184	175
	8	248	231	230	227	200
	12	294	243	255	250	-
	16	330	252	272	268	-
8	4	225	284	222	222	212
	8	337	355	313	309	-
	12	422	396	368	361	-
	16	497	412	408	403	-
16	4	265	348	262	261	249
	8	427	510	400	398	-
	12	557	577	496	489	-
	16	675	652	574	560	-

Table 5.3: SCS approximations for random strings of length 100

$n = 90$		$s \leq 100$			
z	k	Tournament	Majority-Merge	Greedy1	Greedy2
2	4	102	128	101	100
	8	108	125	103	104
	12	112	128	103	103
	16	112	126	103	103
4	4	102	143	101	100
	8	104	142	101	101
	12	110	102	101	101
	16	109	100	101	102
8	4	102	161	101	101
	8	105	122	101	101
	12	110	100	101	100
	16	108	100	101	101
16	4	102	154	101	101
	8	105	122	101	101
	12	107	100	101	101
	16	109	100	101	101

Table 5.4: SCS approximations for subsequences of a string of length 100

5.12 Bad examples with respect to the input size for the SCS approximation algorithms

The preceding sections analysed the performance of various approximation algorithms for the SCS problem, with respect to the number of strings in the problem instances. However, when studying other optimisation problems, it is common to assess the performance of approximation algorithms with respect to the size of the input rather than with respect to one of the input parameters.

Here we briefly analyse bad cases for the SCS approximation algorithms with respect to the input size. It is shown that none of the algorithms has a performance guarantee with respect to the input size better than $O((\frac{I}{\log_2 I})^{(\log_2 3-1)/2})$ where I is the size of the input.

The Tournament Algorithm

The bad cases for the Tournament Algorithm given in Sections 5.4.2 and 5.4.3 do not elicit poor performance for the algorithms with respect to the size of the input. We describe an alternative set of bad cases which does.

Consider a set of $k = 4^p$ strings $\alpha_0^p, \alpha_1^p, \ldots, \alpha_{k-1}^p$ of length 1 over the alphabet $\Sigma = \{0, 1, \ldots, 2^p - 1\}$ of size 2^p. The strings are defined recursively by

$$\begin{aligned} \alpha_i^p &= \mathcal{F}(i,p) \qquad (0 \le i < 4^p) \\ \mathcal{F}(i,p) &= \begin{cases} i \bmod 2, & \text{if } p = 1 \\ 2^{p-1} \times (\lfloor i/4^{p-1} \rfloor \bmod 2) + \mathcal{F}(i \bmod 4^{p-1}, p-1), & \text{otherwise} \end{cases} \end{aligned}$$

For example, when $p = 2$, the sixteen strings are

$\alpha_0^2 = 0$	$\alpha_4^2 = 2$	$\alpha_8^2 = 0$	$\alpha_{12}^2 = 2$
$\alpha_1^2 = 1$	$\alpha_5^2 = 3$	$\alpha_9^2 = 1$	$\alpha_{13}^2 = 3$
$\alpha_2^2 = 0$	$\alpha_6^2 = 2$	$\alpha_{10}^2 = 0$	$\alpha_{14}^2 = 2$
$\alpha_3^2 = 1$	$\alpha_7^2 = 3$	$\alpha_{11}^2 = 1$	$\alpha_{15}^2 = 3$

It is not hard to verify that a feasible application of the Tournament Algorithm, pairing α_0^p with α_1^p, α_2^p with α_3^p etc, leads, after two rounds of the tournament, to the analogous set of strings for $k = 4^{p-1}$ where symbol j $(0 \le j \le 2^{p-1} - 1)$ is replaced by the concatenation of the three symbols $2j$, $(2j + 1)$, and $2j$, respectively.

In the above example, after two rounds of the tournament the four remaining strings could be;

$$\alpha_{0'}^2 = 010 \qquad \alpha_{1'}^2 = 232 \qquad \alpha_{2'}^2 = 010, \qquad \alpha_{3'}^2 = 232$$

where the number of dashes indicates the number of pairs of rounds of the tournament the string has been through. After a further two rounds, the one remaining

string could be (ignoring the white space);

$$\alpha_{0''}^2 = 010\ 232\ 010.$$

In the whole tournament, there are p pairs of rounds which divide the number of strings by four and multiply the lengths of the strings by three. The length of the supersequence returned by the Tournament Algorithm is therefore 3^p. The unique SCS is just the string containing the symbols of the alphabet in lexicographic order and has length 2^p. The size of the input is $kn \log_2 z = 4^p \log_2 2^p = 4^p p$. We therefore have

$$\frac{t}{s} \geq \left(\frac{3}{2}\right)^p > \left(\frac{I}{\log_2 I}\right)^c \text{ where } c = \frac{\log_2 3 - 1}{2}$$

where s is the length of SCS, t is the length of a supersequence returned by the Tournament Algorithm, and I is the size of the input.

The Majority-Merge Algorithm

Recall the bad case for the Majority-Merge Algorithm over an unbounded alphabet in Section 5.5.2. It comprises k strings of length n over the alphabet $\Sigma = \{a_1, a_2, \ldots, a_k\}$, defined by $\alpha_i = a_i(a_1)^{n-1}$ for $1 \leq i \leq k$. If we fix $n = k$ then for the behaviour of the algorithm described in Section 5.5.2 we get

$$\frac{m}{s} \geq \frac{kn}{n+k-1} = \frac{n^2}{2n-1} > \frac{n}{2} \tag{5.27}$$

where s is the length of an SCS and m is the length of a common supersequence found by applying the Majority-Merge Algorithm. If I is the input size then

$$\begin{aligned} I &= n^2 \log_2 n, \\ \frac{I}{2\log_2 I} &= \frac{n^2 \log_2 n}{4(\log_2 n + \frac{1}{2}\log_2 \log_2 n)}, \\ \frac{I}{2\log_2 I} &< \left(\frac{n}{2}\right)^2. \end{aligned} \tag{5.28}$$

Combining 5.27 and 5.28 we get

$$\frac{m}{s} > \left(\frac{I}{2\log_2 I}\right)^{1/2}.$$

Algorithms Greedy1 and Greedy2

The bad case for Greedy1 and Greedy2 given in Section 5.6.2 does not elicit poor performance for the algorithms with respect to the size of the input. We describe an

alternative set of bad cases which does. In what follows, the name "Greedy" refers to both Greedy1 and Greedy2.

Consider a set of $k = 2^n$ strings $\alpha_0^n, \alpha_1^n, \ldots, \alpha_{k-1}^n$ of length n ($n \geq 1$) over the alphabet $\Sigma = \{0, 1, 2, \ldots, n\}$ of size $n+1$. The strings are defined recursively by

$$\begin{aligned} \alpha_i^n &= \gamma_i^n + 0^{n-|\gamma_i^n|}, \\ \gamma_i^p &= \begin{cases} \textit{the empty string}, & \text{if } i = 0 \\ \gamma_i^{p-1}, & \text{if } 1 \leq i < 2^{p-1}, \\ \gamma_{i-2^{p-1}}^{p-1} + p, & \text{if } 2^{p-1} \leq i < 2^p. \end{cases} \end{aligned}$$

For example, when $n = 3$ the eight strings are

$$\begin{aligned} \alpha_0^3 &= 000 \\ \alpha_1^3 &= 100 \\ \alpha_2^3 &= 200 \\ \alpha_3^3 &= 120 \\ \alpha_4^3 &= 300 \\ \alpha_5^3 &= 130 \\ \alpha_6^3 &= 230 \\ \alpha_7^3 &= 123 \end{aligned}$$

It is clear that no two of the strings $\alpha_0^n, \ldots, \alpha_{k-1}^n$ are identical so we get the following lemma.

Lemma 5.12.1 $|LCS(\alpha_i^n, \alpha_j^n)| \leq n-1$ *for* $0 \leq i < j \leq 2^n - 1$

It is not difficult to verify that if Greedy merges an accumulating supersequence β_i^n with α_{i+1}^n to form β_{i+1}^n, and $\beta_0^n = \alpha_0^n$, then a possible value for β_i^n ($1 \leq i \leq k$) is

$$\delta_i^n = \alpha_i^n[1] + \beta_{i-1}^n$$

and the length of the LCS of β_i^n and α_{i+1}^n is $n-1$.

For example, when $n = 3$ the eight strings and the accumulating supersequence are

$$\begin{aligned} \alpha_0^3 &= 000 & \beta_0^3 &= 000 \\ \alpha_1^3 &= 100 & \beta_1^3 &= 1000 \\ \alpha_2^3 &= 200 & \beta_2^3 &= 21000 \\ \alpha_3^3 &= 120 & \beta_3^3 &= 121000 \\ \alpha_4^3 &= 300 & \beta_4^3 &= 3121000 \\ \alpha_5^3 &= 130 & \beta_5^3 &= 13121000 \\ \alpha_6^3 &= 230 & \beta_6^3 &= 213121000 \\ \alpha_7^3 &= 123 & \beta_7^3 &= 1213121000 \end{aligned}$$

It is clear that the above represents feasible behaviour for Greedy1. It follows from Lemma 5.12.1 and the fact that $|LCS(\beta_i^n, \alpha_{i+1}^n)| = n-1$ that it is also feasible

behaviour for Greedy2.

The length of the supersequence returned by Greedy is therefore $n + 2^n - 1$. The length of an SCS (e.g. $12 \ldots n0^n$) is clearly $2n$. The size of the input is $kn \log_2 z = 2^n n \log_2(n+1)$. We therefore have

$$\frac{g}{s} \geq \frac{n + 2^n - 1}{2n} \geq \frac{I}{2(\log_2 I)^{2+\epsilon}} \text{ for any } \epsilon > 0$$

where s is the length of an SCS, g the length of a supersequence returned by either Greedy Algorithm, and I is size of the input.

5.13 Conclusions and open problems

An exact characterisation of the worst-case performance of both Long-Run and Best-Next over bounded and unbounded alphabets have been found. It is shown that the worst-case behaviour of Long-Run depends on the alphabet size and the worst-case behaviour of Best-Next depends on the length of the strings.

Close bounds for the worst-case performance of the Tournament, Greedy1, Centre of the Star, Minimum Spanning Tree and Greedy2 algorithms, on an unbounded alphabet, have been found. A precise characterisation of the worst-case behaviour for bounded and unbounded alphabets for the Majority-Merge Algorithm has been found. All the algorithms have been shown to have bad performance on bounded alphabets and to have worst-case behaviour no better than $\Omega(k)$ for an unbounded alphabet.

From the empirical results, it is clear that the Majority-Merge clearly suffers the least by increasing the number of strings but suffers the most by an increase in the alphabet size. The Tournament, Greedy1 and Greedy2 algorithms have similar performance but the Greedy algorithms and in particular Greedy2 appears to consistently return the shorter supersequence. When the strings have a very short SCS, the Tournament, Greedy1, Greedy2 and Majority-Merge algorithms are all capable of returning good approximations. The performance of the Majority-Merge Algorithm varies dramatically depending on the number of strings because as the number of strings increases, the probability of Majority-Merge choosing the "correct" first character increases. In fact, except in unlikely circumstances, the probability of Majority-Merge choosing the "correct" character at any step of the algorithm increases.

The following problems remain open:

1) Is there a polynomial-time approximation algorithm for the LCS with worst-case performance guarantee better than each of $O(z)$, $O(k)$, and $O(n)$ where no restriction is placed on the value of any parameter?

2) As for the LCS, is there a polynomial-time approximation algorithm for the SCS

with worst-case performance rising more slowly than all the three main problem parameters (z, k, n) where no restriction is placed on the value of any parameter? 3) Is there a polynomial-time approximation algorithm for the SCS with a worst-case performance guarantee better than $O((\frac{I}{\log_2 I})^{(\log_2 3-1)/2})$ where I is the input size?

In connection with 1), Jiang and Li [37] showed that $\exists \delta > 0$ such that if there is an approximation algorithm for the LCS problem (with arbitrarily long strings) with worst-case performance k^δ then **P=NP**, where k is the number of strings. Bonizzoni, Duella, and Mauri [9] proved that the LCS problem on a bounded alphabet is **MAX SNP**-hard implying that a polynomial time approximation scheme is not possible for the LCS on a bounded alphabet (unless **P=NP**). In connection with 2), Jiang and Li [37] showed that $\exists \delta > 0$ such that if there is an approximation algorithm for the SCS problem with worst-case performance $O(\log^\delta k)$ then **NP** is contained in DTIME($2^{polylog\ k}$). Bonizzoni et al. [9] proved that the SCS problem on a bounded alphabet is **MAX SNP**-hard.

Chapter 6

Maximal subsequences and minimal supersequences

6.1 Introduction

As discussed in Chapters 4 and 5, we may have to resort to finding an approximate solution to an instance of the LCS (or SCS) problem because it is too big to solve using an exact algorithm. Having found an approximation γ to the LCS of a set P of strings, we could attempt to add symbols to γ, so that it remains a common subsequence of P, and continue to do so while it is possible. The question then arises as to how long we can typically expect γ to get before no more symbols can be added, for it to remain a common subsequence of P. Similarly, how short can we typically expect δ, an approximation to the SCS of P, to get while we remove symbols from δ so that it remains a common supersequence of P?

Recall the following definitions from Chapter 1. A common subsequence α of a set P of strings is *maximal* if no proper supersequence of α is also a common subsequence of P. A *Shortest Maximal Common Subsequence* (SMCS) of P is a maximal common subsequence of shortest possible length. Analogously, a common supersequence α of a set P of strings is *minimal* if no proper subsequence of α is also a supersequence of P. A *Longest Minimal Common Supersequence* (LMCS) of P is a minimal common supersequence of longest possible length.

For example, for the strings

$$\begin{aligned} \alpha_1 &= abcd \\ \alpha_2 &= bacd \\ \alpha_3 &= bcad \\ \alpha_4 &= bcda \end{aligned}$$

the SMCS is the string a of length 1 but the LCS is the string bcd of length 3.

The LMCS is the string *bcdabcd* of length 7 although the string *abcda* is the SCS of length 5.

In this chapter, we study the Shortest Maximal Common Subsequence problem and and the Longest Minimal Common Supersequence problem from the complexity point of view.

We show that the SMCS problem is **NP**-hard when the number k of strings becomes a problem parameter. Furthermore, we prove a strong negative result regarding the likely existence of good polynomial-time approximation algorithms for the SMCS problem in the case of general k.

We also show that, like the LCS and SCS problems, both of these new problems can be solved in polynomial time by dynamic programming for $k = 2$ (and, by extending the algorithms, for any fixed value of k). However, the dynamic programming algorithms are less straightforward than those for the LCS and SCS problems, and have time and space complexities $O(m^2n)$ and $O(mn^2)$ respectively for strings of lengths m and n ($m \leq n$). Note that the existence of polynomial-time algorithms for the SMCS and LMCS problems in the case of two strings is by no means obvious. Consider the problem of finding a maximum cardinality matching in a bipartite graph. This problem is well known to be solvable in polynomial time, whereas the problem of finding a smallest maximal bipartite matching is **NP**-hard [72].

6.2 The SMCS problem for general k strings

It is well-known that the problem of finding an LCS of k strings is **NP**-hard, even in a number of special cases [44, 56, 64]. Recently, Jiang and Li [37] described a polynomial-time transformation from the Maximum Clique problem for graphs to the LCS problem with the property that the strings constructed have a common subsequence of length r if and only if the original graph has a clique of size r. If the given graph has k vertices then the derived LCS instance comprises $2k$ strings. Now, it has recently been established by Arora et al [7] that, if **P**$\neq$ **NP**, then there cannot exist a polynomial-time approximation algorithm for the Maximum Clique problem with a performance guarantee of k^δ, for some $\delta > 0$. Hence, Jiang and Li were able to conclude that, unless **P**= **NP**, there cannot exist a polynomial-time approximation algorithm for LCS with a performance guarantee of k^δ, for some $\delta > 0$.

As we shall see in Theorem 6.2.1, the transformation, from Independent Set, given by Maier [44] to prove the **NP**-completeness for LCS also serves as a transformation from the Minimum Independent Dominating Set problem to SMCS. The former problem is also **NP**-hard [19], and was shown by Irving [31] not to have a polynomial-time approximation algorithm with a constant performance guarantee (if **P**$\neq$ **NP**). Halldórsson [23] has recently strengthened this result to show that,

if **P**$\neq$ **NP**, then, for no $\delta < 1$, can there exist a polynomial-time approximation algorithm with performance guarantee k^{δ}. Maier's transformation has the property that the strings constructed have an LCS of length r if and only if the original graph has an independent set of size r. If the given graph has k *edges* then the derived LCS instance has $k+1$ strings. It will therefore follow from the transformation, not only that SMCS is **NP**-hard, but also that this same strongly negative approximability result applies to the SMCS problem.

Theorem 6.2.1 *(i) The SMCS problem is* **NP***-hard.*
(ii) If **P**$\neq$ **NP**, *then, for no* $\delta < 1$, *can there exist a polynomial-time approximation algorithm for SMCS on k strings with performance guarantee* k^{δ}.

Proof
(i) Let $G = (V, E)$, t, with $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$, be an arbitrary instance of (the decision version of) the Minimum Independent Dominating Set problem. We construct an instance of SMCS as follows. Include in the set S of strings the string $\alpha_0 = v_1 v_2 \ldots v_n$. For each edge $e_i = \{v_p, v_q\}$ $(p < q)$ include in the set S of strings the string α_i defined by

$$\alpha_i = v_1 v_2 \ldots v_{p-1} v_{p+1} \ldots v_n v_1 v_2 \ldots v_{q-1} v_{q+1} \ldots v_n.$$

We claim that G has an independent dominating set of size t if and only if S has a maximal common subsequence of length t.

To prove this claim, we must show that (a) if G has an independent dominating set U of size t then S has a maximal common subsequence of length t; (b) if S has a maximal common subsequence of length t then G has an independent dominating set of size t.

To prove (a), assume that $U = \{v_{u_1}, v_{u_2}, \ldots, v_{u_t}\}$ is an independent dominating set of size t, where $1 \leq u_1 < u_2 < \ldots < u_t \leq n$.

It can easily be checked that the string $\alpha = v_{u_1} v_{u_2} \ldots v_{u_t}$ is a common subsequence of S. If some supersequence, α', of α, is a common subsequence of S then observe for a contradiction, that $\exists$ v_p in α' but not in α, which is connected to $v_q \in U$, in G by edge $e_j = \{v_p, v_q\}$ since U is dominating. Assuming $p < q$ then the string $\alpha_j = v_1 v_2 \ldots v_{p-1} v_{p+1} \ldots v_n v_1 v_2 \ldots v_{q-1} v_{q+1} \ldots v_n$. For α' to be a subsequence of α_0, v_p must precede v_q in α'. But this prevents α' from being a subsequence of α_j. A similar contradiction is obtained if $p > q$ is assumed.

To prove (b), assume $\alpha = v_{u_1} v_{u_2} \ldots v_{u_t}$, of length t, is a maximal common subsequence of the strings in S. The first observation is that if v_{u_p} and v_{u_q} are two symbols in α and $p < q$ then $u_p < u_q$. For otherwise α could not be a subsequence of α_0. The elements of α must form an independent set, U, of size t, in G. To see this, observe for a contradiction that if two elements, v_{u_p} and v_{u_q} $(p < q)$, of α are connected in G by edge $e_j = \{v_{u_p}, v_{u_q}\}$ then the string $v_{u_p} v_{u_q}$, a subsequence of α,

would not be a subsequence of α_j and hence α would not be a subsequence of α_j. If U is not maximal then $\exists\ U'$, an independent set of size $t' > t$, and $U \subset U'$. Observe for a contradiction that this would imply $\exists v_j \in U'$ and $v_j \notin U$. Then it is easy to see that the string $\alpha' = v_{u_1} \ldots v_{u_p} v_j v_{u_{p+1}} \ldots v_{u_t}$, where $u_p < j < u_{p+1}$, a supersequence of α, would be a common subsequence of all the strings in S, contradicting the maximality of α. This concludes the proof of part (i).

The proof of part (ii) follows from the observation that the reduction is linear [53], and therefore preserves the approximability of the Minimum Independent Dominating Set, and from the result of Halldórsson [23] on the approximability of that problem. □

6.3 The SMCS and LMCS problems for $k = 2$ strings

As is discussed in Section 1.1, when restricted to the case of just two strings α and β of lengths m and n respectively, the LCS and SCS problems are easily solvable in $O(mn)$ time by dynamic programming. Indeed, in this case, the problems are dual, in that $s = m + n - l$, where l and s are the lengths of an LCS and an SCS respectively, and an SCS can easily be formed from an LCS.

The following example illustrates the fact that there is no obvious corresponding duality between the SMCS and LMCS in the case of two strings. For the strings

$$\begin{aligned} \alpha &= abc, \\ \beta &= dab, \end{aligned}$$

the only maximal common subsequence is *ab* of length 2, while *dabc*, *abcdab*, *abdcab*, and *abdacb* are the minimal common supersequences, the latter three being the longest.

It is true, however, that if γ is a maximal common subsequence of length r of α and β, then forming an alignment of α and β in which the elements of γ are matched reveals a minimal common supersequence of α and β of length $m + n - r$. Hence, if l' is the length of an SMCS, and s' the length of an LMCS, it follows that $s' \geq n + m - l'$.

Hence the question arises as to whether either or both of the SMCS and LMCS problems can be solved in polynomial time, by dynamic programming or otherwise.

In the following two sections we describe polynomial-time algorithms to determine the length of an SMCS and an LMCS of two strings, respectively. In fact these algorithms will determine the lengths of all maximal common subsequences and all minimal common supersequences respectively. They will also allow the construction

of an SMCS, and indeed of all the maximal common subsequences (respectively an LMCS, and all minimal common supersequences) of the two strings (although to construct all maximal common subsequences or minimal common supersequences would require exponential time).

The algorithms use a dynamic programming approach based on a table that relates α^i and β^j, for $i = 1,\ldots,m$, $j = 1,\ldots,n$, where m,n are the lengths of α,β respectively. However, as we shall see, for each i,j we must retain rather more information than merely the lengths of the maximal common subsequences, or of the minimal common supersequences, of α^i and β^j.

6.3.1 The SMCS Algorithm

Given a string α and a subsequence γ of α, we define

$$sp(\alpha,\gamma) = \text{length of the shortest prefix of } \alpha \text{ that is a supersequence of } \gamma.$$

Given strings α,β of lengths m and n respectively, we define the set S_{ij}, for each $i = 1,\ldots,m$, $j = 1,\ldots,n$, by

$S_{ij} = \{(r,(x,y)) : \alpha^i$ and β^j have a maximal common subsequence γ of length r, and $sp(\alpha,\gamma) = x, sp(\beta,\gamma) = y\}$

with

$$S_{00} = \{(0,(0,0))\}.$$

For string α of length m, position i and symbol a, we define

$$next_\alpha(i,a) = \begin{cases} \min\{k : \alpha[k] = a, k > i\} & \text{if such a } k \text{ exists} \\ m+1 & \text{otherwise.} \end{cases}$$

If α is a string and a a symbol of the alphabet, we denote by $\alpha + a$ the string obtained by appending a to α. Likewise, if the last character of α is a, we denote by $\alpha - a$ the string obtained by deleting the final a from α.

The algorithm for SMCS is based on a dynamic programming scheme for the sets S_{ij} defined above. So evaluation of S_{mn} reveals the length of the SMCS, but also finds the lengths of all maximal common subsequences of α and β (indeed of all maximal common subsequences of all pairs of prefixes of α and β). Furthermore, by applying suitable tracebacks through the array of S_{ij} values, we can recover an SMCS and all maximal common subsequences.

The basis of the dynamic programming scheme is contained in the following theorem:

Theorem 6.3.1 *(i) If* $\alpha[i] = \beta[j] = a$ *then*

$$S_{ij} = \{(r, (next_\alpha(x,a), next_\beta(y,a))) : (r-1,(x,y)) \in S_{i-1,j-1}\}.$$

(ii) If $\alpha[i] \neq \beta[j]$ *then*

$$S_{ij} = \{(r,(x,j) \in S_{i-1,j}\} \cup \{(r,(i,y)) \in S_{i,j-1}\} \cup (S_{i-1,j} \cap S_{i,j-1}).$$

Proof

(i) Suppose $(r-1,(x,y)) \in S_{i-1,j-1}$ and that γ' is a maximal common subsequence of α^{i-1} and β^{j-1} of length $r-1$ with $sp(\alpha,\gamma') = x$ and $sp(\beta,\gamma') = y$. Then it is immediate that $\gamma = \gamma' + a$ is a maximal common subsequence of α^i and β^j, and that $sp(\alpha,\gamma) = next_\alpha(x,a)$, $sp(\beta,\gamma) = next_\beta(y,a)$.

On the other hand, suppose that γ is a maximal common subsequence of length r of α^i and β^j. Then the last symbol of γ is a, and $\gamma' = \gamma - a$ is certainly a common subsequence of α^{i-1} and β^{j-1}. If it were not maximal, then some supersequence δ of γ' would be a common subsequence of α^{i-1} and β^{j-1}, and therefore $\delta + a$, a supersequence of γ, would be a common subsequence of α^i and β^j, contradicting the maximality of γ. So $(r-1,(sp(\alpha,\gamma'),sp(\beta,\gamma')) \in S_{i-1,j-1}$ and $(r,(sp(\alpha,\gamma),sp(\beta,\gamma)) \in S_{ij}$ with $sp(\alpha,\gamma) = next_\alpha(sp(\alpha,\gamma'),a)$ and $sp(\beta,\gamma) = next_\beta(sp(\beta,\gamma'),a)$.

(ii) Suppose $(r,(x,j)) \in S_{i-1,j}$, and that γ is a maximal common subsequence of α^{i-1} and β^j of length r with $sp(\alpha,\gamma) = x$, $sp(\beta,\gamma) = j$. Then γ is a common subsequence of α^i and β^j, and must be maximal since $\gamma + \alpha[i]$ cannot be a subsequence of β^j. A similar argument holds for $(r,(i,y)) \in S_{i,j-1}$. So $\{(r,(x,j) \in S_{i-1,j}\} \cup \{(r,(i,y)) \in S_{i,j-1}\} \subseteq S_{ij}$.

Further, if $(r,(x,y)) \in S_{i-1,j} \cap S_{i,j-1}$, then there is a string γ with $sp(\alpha,\gamma) = x < i$, $sp(\beta,\gamma) = y < j$, of length r, which is a maximal common subsequence of α^i and β^{j-1}, and of α^{i-1} and β^j. So γ must also be a maximal common subsequence of α^i and β^j. For any supersequence of γ that is a subsequence of α^i and β^j must either be a subsequence of α^i and β^{j-1}, or of α^{i-1} and β^j.

On the other hand, suppose that γ is a maximal common subsequence of length r of α^i and β^j.

case (iia) $sp(\alpha,\gamma) = i$. Then γ is a maximal common subsequence of α^i and β^{j-1}, and so $(r,(i,y)) \in S_{i,j-1}$ for some y.

case (iib) $sp(\beta,\gamma) = j$. Then γ is a maximal common subsequence of α^{i-1} and β^j, and so $(r,(x,j)) \in S_{i-1,j}$ for some x.

case (iic) $sp(\alpha,\gamma) < i, sp(\beta,\gamma) < j$. Then γ is both a maximal common subsequence of α^{i-1} and β^j, and of α^i and β^{j-1}. So $(r,(sp(\alpha,\gamma),sp(\beta,\gamma))) \in S_{i-1,j} \cap S_{i,j-1}$.

This completes the proof of the theorem. □

Recovering a Shortest Maximal Common Subsequence

The recovery of a particular Shortest Maximal Common Subsequence involves a standard type of traceback through the dynamic programming table from cell (m, n), during which the sequence is constructed in reverse order. To facilitate this traceback, each entry in position (i, j) in the table (for all i, j) should have associated with it, during the application of the dynamic programming scheme, one or more pointers indicating which particular element(s) in cells $(i-1, j)$, $(i, j-1)$ or $(i-1, j-1)$ led to the inclusion of that element in cell (i, j). For example, if $\alpha[i] = \beta[j] = a$, and $(r-1, (x, y)) \in S_{i-1,j-1}$ then $(r, (next_\alpha(x, a), next_\beta(y, a))$ is placed in cell (i, j) with a pointer to the element $(r-1, (x, y))$ in cell $(i-1, j-1)$.

With these pointers, any path from an element $(r, (x, y))$ in cell (m, n) to the element in cell $(0, 0)$ represents a maximal common subsequence of α and β of length r, namely the reversed sequence of matching symbols from the two strings corresponding to cells from which the path takes a diagonal step.

Analysis of the SMCS Algorithm

The number of cells in the dynamic programming table is essentially mn, so that if we could show that the number of entries in each cell was bounded by, say, $\min(m, n)$, and that the total amount of computation was bounded by a constant times the total number of table entries, then we would have a cubic time worst-case bound for the complexity of the algorithm. However, this turns out not to be the case, as the following example shows.

Consider two strings of length $n = p(p+1)/2 + q$ over an alphabet $\Sigma = \{a_1, \ldots, a_n\}$, defined as follows

$$\alpha = \alpha_1 + \alpha_2 + \cdots + \alpha_p + a_{p(p+1)/2+1}, \ldots, a_n$$

$$\beta = \alpha_p + \alpha_{p-1} + \cdots + \alpha_1 + a_n, \ldots, a_{p(p+1)/2+1}$$

where $\alpha_1 = a_1$, $\alpha_2 = a_2 a_3$, $\ldots$, $\alpha_p = a_{(p-1)p/2+1} \ldots a_{p(p+1)/2}$, and $+$ denotes concatenation.

It is not hard to see that position (n, n) in the dynamic programming table contains the pq entries $(r, (x, y))$ for $r = 2, \ldots, p+1$, $x = p(p+1)/2 + 1, \ldots, n$, $y = n + 1 + p(p+1)/2 - x$. With $q = \Theta(p^2)$, this gives $\Theta(n^{3/2})$ entries in the (n, n)th cell.

However, suppose that we wish to find only the length of an SMCS (and to construct such a sequence by traceback through the table). Then, if any particular cell in the table contains more than one entry $(r, (x, y))$ with the same (x, y) component, we may discard all but the one with the smallest r value. For if a maximal common

subsequence γ has a prefix γ' such that $sp(\alpha,\gamma') = x$ and $sp(\beta,\gamma') = y$, then to make γ as short as possible, γ' must be chosen as short as possible.

Also, if the entries $(r,(x,y))$ in the (i,j)th cell are listed in increasing order of x, then they must clearly also be in decreasing order of y, and therefore, since $x \leq i$, $y \leq j$, the number of such entries with distinct (x,y) components cannot exceed $\min(i,j)$. Further, it is easy to see that by processing the lists of cell entries in this fixed order, the amount of work done in computing the contents of cell (i,j) is, in case (i) bounded by a constant times the number of entries in cell $(i-1,j-1)$, and in case (ii) bounded by a constant times the sum of the numbers of entries in cells $(i-1,j)$ and $(i,j-1)$. (In case (i), this assumes precomputation of the tables of *next* values, which can easily be achieved in $O(n|\Sigma|)$ time for a string of length n, where Σ is the alphabet.)

In conclusion, the length of an SMCS can be established by a suitably amended version of the above dynamic programming scheme in $O(m^2n)$ time in the worst case, for strings of lengths m and n $(m \leq n)$. Furthermore, such a subsequence can also be constructed from the dynamic programming table without increasing that overall time bound. But it remains open whether the lengths of all maximal common subsequences can be established within that time bound. A trivial bound of $O(m^3n)$ applies in that case, since the number of entries in each cell is certainly bounded by m^2.

6.3.2 The LMCS Algorithm

The LMCS algorithm is not dissimilar in spirit to the SMCS algorithm, and there is a certain duality involving the terms in which the algorithm is expressed.

Given strings α and γ, we define

$$lp(\alpha,\gamma) = \text{length of the longest prefix of } \alpha \text{ that is a subsequence of } \gamma.$$

Given strings α,β of lengths m and n respectively, we define the set T_{ij}, for each $i = 0,\ldots,m$, $j = 0,\ldots,n$, by

$T_{ij} = \{(r,(x,y)) :$ there exists a minimal common supersequence γ of α^i and β^j, of length r, such that $lp(\alpha,\gamma) = x, lp(\beta,\gamma) = y\}$.

Finally, for string α, position i and symbol a, we define

$$f_\alpha(i,a) = \begin{cases} i+1 & \text{if } \alpha[i+1] = a \\ i & \text{otherwise.} \end{cases}$$

The algorithm for LMCS is based on a dynamic programming scheme for the sets T_{ij} defined above. So evaluation of T_{mn} not only reveals the length of the LMCS, but also finds the lengths of all minimal common supersequences of α and β (indeed of all

minimal common supersequences of all pairs of prefixes of α and β). Furthermore, by applying suitable tracebacks through the array of T_{ij} values, we can recover an LMCS and all minimal common supersequences.

The zero'th row and column of the T_{ij} table can be evaluated trivially, as follows:

$$T_{i0} = \{(i, (i, lp(\beta, \alpha^i)))\} \quad (1 \leq i \leq m)$$

and

$$T_{0j} = \{(i, (lp(\alpha, \beta^j), j))\} \quad (1 \leq j \leq n)$$

with

$$T_{00} = \{(0, (0, 0))\}.$$

The basis of the dynamic programming scheme is contained in the following theorem:

Theorem 6.3.2 *(i) If* $\alpha[i] = \beta[j] = a$ *then*

$$T_{ij} = \{(r, (f_\alpha(x, a), f_\beta(y, a)) : (r-1, (x, y)) \in T_{i-1,j-1}\}$$

(ii) If $\alpha[i] = a \neq b = \beta[j]$ *then*

$$\begin{aligned} T_{ij} \;=\; & \{(r, (f_\alpha(x, b), j)) : (r-1, (x, j-1)) \in T_{i,j-1}\} \\ & \cup \{(r, (i, f_\beta(y, a)) : (r-1, (i-1, y)) \in T_{i-1,j}\}. \end{aligned}$$

Proof

(i) Suppose $(r-1, (x, y)) \in T_{i-1,j-1}$ and that γ' is a minimal common supersequence of α^{i-1} and β^{j-1} of length $r-1$ with $lp(\alpha, \gamma') = x$ and $lp(\beta, \gamma') = y$. Then it is immediate that $\gamma = \gamma' + a$ is a minimal common supersequence, of length r, of α^i and β^j, and that $lp(\alpha, \gamma) = f_\alpha(x, a)$ and $lp(\beta, \gamma) = f_\beta(y, a)$.

On the other hand, suppose that γ is a minimal common supersequence of length r of α^i and β^j. Then $\gamma[r] = a$, and $\gamma' = \gamma - a$ is certainly a common supersequence of α^{i-1} and β^{j-1}. If γ' were not minimal, then some subsequence δ of γ' would be a common supersequence of α^{i-1} and β^{j-1}, and therefore $\delta + a$, a subsequence of γ, would be a common supersequence of α^i and β^j, contradicting the minimality of γ. So $(r-1, (x, y)) \in T_{i-1,j-1}$ with $x = lp(\alpha, \gamma')$, $y = lp(\beta, \gamma')$ and $lp(\alpha, \gamma) = f_\alpha(x, a)$, $lp(\beta, \gamma) = f_\beta(y, a)$.

(ii) Suppose $(r-1, (i-1, y)) \in T_{i-1,j}$, and that γ' is a minimal common supersequence of α^{i-1} and β^j of length $r-1$ with $lp(\alpha, \gamma') = i-1$, $lp(\beta, \gamma') = y$. (The argument is similar in the case $(r-1, (x, j-1)) \in T_{i,j-1}$.) Then $\gamma = \gamma' + a$ is a common supersequence of α^i and β^j with $lp(\alpha, \gamma) = i$ and $lp(\beta, \gamma) = f_\beta(y, a)$. Further, γ must be minimal. For suppose that a subsequence δ of γ is a common supersequence

of α^i and β^j. If δ were a subsequence of γ', then γ' would not be a minimal common supersequence of α^{i-1} and β^j. So $\delta = \delta' + a$, where δ' is a subsequence of γ'. So δ' cannot be a common supersequence of α^{i-1} and β^j. If it is not a supersequence of α^{i-1} then $\delta' + a$ cannot be a supersequence of α^i — a contradiction. If it is not a supersequence of β^j then, since $\delta' + a$ is a supersequence of β^j, we must have $\beta[j] = a$ — a contradiction.

On the other hand, suppose that γ is a minimal common supersequence of length r of α^i and β^j. Then $\gamma[r] = a$ or b.
case (iia) $\gamma[r] = a$. It is immediate that $lp(\alpha, \gamma) = i$, for otherwise $\gamma - a$ would be a common supersequence of α^i and β^j. So $\gamma' = \gamma - a$ is a minimal common supersequence of α^{i-1} and β^j with $lp(\alpha, \gamma') = i - 1$ and $lp(\beta, \gamma') = y$ for some y such that $lp(\beta, \gamma) = f_\beta(y, a)$.
case (iib) $\gamma[r] = b$. A similar argument shows that $\gamma' = \gamma - b$ is a minimal common supersequence of α^i and β^{j-1} with $lp(\beta, \gamma') = j - 1$ and $lp(\alpha, \gamma') = x$ for some x such that $lp(\alpha, \gamma) = f_\alpha(x, b)$.

This completes the proof of the theorem. □

Recovering a Longest Minimal Common Supersequence

As in the case of an SMCS, the recovery of a particular Longest Minimal Common Supersequence involves a traceback through the dynamic programming table from cell (m, n) to cell $(0, 0)$, during which the sequence is constructed in reverse order. To facilitate the traceback, each entry in position (i, j) in the table (for all i, j) should have associated with it, during the application of the dynamic programming algorithm, one or more pointers indicating which particular element(s) in cells $(i - 1, j)$, $(i, j - 1)$ or $(i - 1, j - 1)$ led to the inclusion of that element in cell (i, j). For example, if $\alpha[i] = a = \beta[j]$ and $(r - 1, (x, y)) \in T_{i-1,j-1}$, then $(r, (f_\alpha(x, a), f_\beta(y, a)))$ is placed in cell (i, j) with a pointer to the element $(r - 1, (x, y))$ in cell $(i - 1, j - 1)$.

With these pointers, any path from an element $(r, (x, y))$ in cell (m, n) to the element in cell $(0, 0)$ represents a minimal common supersequence of α and β of length r, namely the reversed sequence of symbols found by recording $\alpha[i]$ for a vertical or diagonal step from cell (i, j) and $\beta[j]$ for a horizontal step from cell (i, j).

Analysis of the LMCS Algorithm

As in the case of the SMCS algorithm, we can establish a cubic time bound for the restricted version of the LMCS algorithm that is designed to find the length of an LMCS, and to construct such a common supersequence from the dynamic programming table. The trick again is the observation that, for this purpose, whenever $(r, (x, y))$ elements in the same cell have the same (x, y) component, only one need

be retained, namely that with the largest r value. For if a minimal common supersequence γ has a prefix γ' such that $lp(\alpha, \gamma') = x$ and $lp(\beta, \gamma') = y$, then to make γ as long as possible, γ' should be chosen as long as possible.

By this means we can restrict the number of elements in the (i,j)th cell to at most $i + j$, recalling that each such entry $(r, (x, y))$ has either $x = i$ or $y = j$. This leads to a worst-case time bound of $O(mn^2)$ for this version of the algorithm. Again, it is not clear whether the lengths of all minimal common supersequences can be found in time better than $O(mn^3)$ in the worst case, this arising from the obvious upper bound of $(m + n)^2$ on the number of elements in each cell of the table.

6.4 Conclusion and open problem

We have shown that, in the case of two strings, a Shortest Maximal Common Subsequence and a Longest Minimal Common Supersequence can be found in polynomial time by dynamic programming. In fact, both algorithms can be extended in a fairly straightforward way to find the SMCS or the LMCS of any fixed number of strings. However, for general k, we have shown that finding an SMCS of k strings is **NP**-hard, and further, that, unless **P**= **NP**, over an unbounded alphabet, the length of an SMCS cannot be approximated, in polynomial time, within a factor of k^δ for any $\delta < 1$. We conjecture that over an unbounded alphabet, the length of the LMCS is as hard to approximate as the length of the SMCS. [1]

[1] Middendorf [49] has shown that both problems are **MAX SNP**-hard, over a binary alphabet.

Chapter 7

Consistent subsequences and supersequences

7.1 Introduction

Recall the following definitions from Section 1.5.3. Given two sets, P (*Positive*) and N (*Negative*), of strings, a *consistent* subsequence (supersequence) of P and N is a string that is a common subsequence (supersequence) of P and a common non-subsequence (non-supersequence) of N. In this chapter, we study consistent sequence problems from a complexity point of view. There are two categories of problems, existence problems and optimisation problems. Given two sets, P and N, of strings, does there exist a consistent subsequence (supersequence)? If a consistent subsequence (supersequence) does exist, what is the length of the shortest/longest? There are therefore two existence problems and four optimisation problems.

It is clear that an **NP**-completeness result for an existence problem implies **NP**-hardness for the two corresponding optimisation problems. Similarly, the existence of a polynomial-time algorithm for an optimisation problem implies the corresponding existence problem is also in **P**. For the **NP**-complete problems, the following questions arise: do they become solvable in polynomial time if we bound $|P|$, or bound $|N|$, or bound both $|P|$ and $|N|$ (if the answer to the first two questions is no)? In this chapter, when a problem is characterised as being in **P**, the most efficient solution is not sought, the aim is merely to show that it is solvable in polynomial time. See Section 1.5.3 for details of previous work.

The known complexities of the existence and optimisation problems are summarised in Tables 7.1 and 7.2 respectively, together with the source for each characterisation. The characterisation is displayed above the source, where NPC/NPH means the problem is **NP**-complete/**NP**-hard, P means the problem can be solved in polynomial time, $\nexists$ means no such sequence exists for any instance of the problem and "Open" means the complexity of the problem remains open. A table reference

		1	2	3	4	5	6
	Existence Problem	$\|P\|, \|N\|$ Unbounded	$\|P\| \geq 2$ Bounded	$\|P\| = 1$	$\|N\| \geq 2$ Bounded	$\|N\| = 1$	$\|P\|, \|N\|$ Bounded
A	Consistent Subsequence	NPC Th. 7.2.1		P (D3)→	NPC Th. 7.2.2		P (D6)→
B	Consistent Supersequence	NPC [38]		P (E3)→	NPC Th. 7.3.1	P [38]	P (E6)→

Table 7.1: Consistent Sequence Existence Problems

		1	2	3	4	5	6
	Optimisation Problem	$\|P\|, \|N\|$ Unbounded	$\|P\| \geq 2$ Bounded	$\|P\| = 1$	$\|N\| \geq 2$ Bounded	$\|N\| = 1$	$\|P\|, \|N\|$ Bounded
C	Shortest Con. Subsequence	NPH [47, 58]→			NPH (A5)→		P Th. 7.2.3
D	Longest Con. Subsequence	NPH [44]→	NPH (A2)→	P §7.2.3	NPH [44]→		P Th. 7.2.4
E	Shortest Con. Supersequence	NPH [44]→	NPH (B2)→	P §7.3.2	NPH [44]→		P Th. 7.3.2
F	Longest Con. Supersequence	NPH [58]→			NPH (B4)→	$\not\exists$ §7.3.4	Open

Table 7.2: Consistent Sequence Optimisation Problems

of the form $(Xi)\rightarrow$ means the entry follows immediately from the table entry in row X and column i. A table reference of the form $[n]$ means the entry is proved in reference $[n]$. A table reference of the form $[n]\rightarrow$ means the entry follows immediately from a result proved in reference $[n]$. A table reference of the form Th.n means the entry is proved in Theorem n of this chapter. Finally, a table reference of the form §n means the entry is quite straightforward and is explained in Section n of this chapter.

One additional piece of notation is used throughout this chapter. The string α^i (or a^i) represents i copies of the string α (or the symbol a) concatenated.

7.2 Consistent Subsequence Problems

7.2.1 Consistent subsequence when $|P|$ is bounded and $|N|$ is unbounded ($|P| \geq 2$)

See table entries A1 and A2.

Theorem 7.2.1 *Determining whether there exists a consistent subsequence is* **NP**-*complete even when* $|P| = 2$.

Proof
Given an instance of the 3-Satisfiability problem (3-SAT), well-known to be **NP**-complete [19], we construct an instance of the Consistent Subsequence problem, over the alphabet $\Sigma = \{\#, 1, 0\}$, as follows.

Set P contains the two strings

$$\begin{aligned} \alpha_1 &= (\#10)^{3c}, \\ \alpha_2 &= (\#01)^{3c}, \end{aligned}$$

where c is the number of clauses in the instance of 3-SAT.

Set N is the union of four subsets, N_1, N_2, N_3, N_4. Set N_1 contains the two strings

$$\begin{aligned} \beta_1 &= 10(\#1010)^{3c-1}, \\ \beta_2 &= (\#\#10)^{3c-1}\#. \end{aligned}$$

The string β_1 prevents any consistent subsequence from having fewer than $3c$ #'s. The string β_2 prevents any consistent subsequence from having fewer than a total of $3c$ 0's and 1's.

Every consistent subsequence of the sets P and N_1 has the form $(\#\{1 \text{ or } 0\})^{3c}$. Such a string represents an assignment of truth values to the literals, in the order in which they appear, in the instance of 3-SAT; 1 represents a *true* assignment and 0 represents a *false* assignment. It remains to ensure that every consistent subsequence of P and N will assign matching literals with matching logical values (N_2), assign opposite literals with opposite logical values (N_3) and satisfy every clause i.e. assign *true* to at least one literal of every clause (N_4).

Set N_2 contains two strings for every pair of matching literals in the instance of 3-SAT. If clause p, literal q matches clause s, literal t, $(1 \leq p < s \leq c,\ 1 \leq q, t \leq 3)$ then N_2 contains the two strings

$$\begin{aligned} \gamma_{pqst} &= (\#10)^{3(p-1)+(q-1)}\#1(\#10)^{(3-q)+3(s-p-1)+(t-1)}\#0(\#10)^{(3-t)+3(c-s)}, \\ \gamma'_{pqst} &= (\#10)^{3(p-1)+(q-1)}\#0(\#10)^{(3-q)+3(s-p-1)+(t-1)}\#1(\#10)^{(3-t)+3(c-s)}. \end{aligned}$$

The string γ_{pqst} prevents the first literal being true while the second is false and γ'_{pqst} prevents the first literal being false while the second is true.

Set N_3 contains two strings for every pair of opposite literals in the instance of 3-SAT. If clause p, literal q is the negation of clause s, literal t, $(1 \leq p < s \leq c,\ 1 \leq q, t \leq 3)$ then N_3 contains the two strings

$$\begin{aligned} \delta_{pqst} &= (\#10)^{3(p-1)+(q-1)}\#1(\#10)^{(3-q)+3(s-p-1)+(t-1)}\#1(\#10)^{(3-t)+3(c-s)}, \\ \delta'_{pqst} &= (\#10)^{3(p-1)+(q-1)}\#0(\#10)^{(3-q)+3(s-p-1)+(t-1)}\#0(\#10)^{(3-t)+3(c-s)}. \end{aligned}$$

The string δ_{pqst} prevents both literals being true and δ'_{pqst} prevents both literals being false.

Set N_4 contains one string for each clause in the instance of 3-SAT, namely, corresponding to clause i, the string

$$\theta_i = (\#10)^{3(i-1)}(\#0)^3(\#10)^{3(c-i)} \qquad (1 \leq i \leq c).$$

The string θ_i prevents clause i from having no *true* literals.

As should now be clear, there exists a satisfying assignment to the variables in the instance of 3-SAT if and only if there exists a consistent subsequence of the strings in the derived instance of Consistent Subsequence. □

The transformation can be modified to work on a binary alphabet by replacing every # with 001 in the construction of P and N.

7.2.2 Consistent subsequence when $|P|$ is unbounded and $|N|$ is bounded ($|N| \geq 1$)

See table entries A4 and A5.

Theorem 7.2.2 *Determining whether there exists a consistent subsequence is* **NP***-complete even when* $|N| = 1$.

Proof

Given an instance of the Independent Set problem, well-known to be **NP**-complete [19], on the graph $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_p\}$, $E = \{e_1, e_2, \ldots, e_q\}$, and t is the target size for an independent set, we construct an instance C of the Consistent Subsequence problem as follows.

The alphabet Σ is the set of vertices of G, i.e. $\{v_1, v_2, \ldots, v_p\}$. Set P contains $q + 1$ strings. The first string in P is

$$\alpha_0 = v_1 v_2 \ldots v_p.$$

This ensures that every consistent subsequence will be a string of vertices with increasing subscript. For each edge $e_i = \{v_x, v_y\}$ $(x < y)$, P contains the string

$$\alpha_i = v_1 v_2 \ldots v_{x-1} v_{x+1} \ldots v_p v_1 v_2 \ldots v_{y-1} v_{y+1} \ldots v_p.$$

The string α_i is a supersequence of every subsequence of α_0 that does not contain both v_x and v_y. It is not a supersequence of the string $\delta = v_x v_y$ or of any supersequence of δ. Hence no common subsequence of α_0 and α_i and, therefore, no consistent subsequence of P and N can contain both v_x and v_y.

Set N contains the single string

$$\beta = (v_p v_{p-1} \ldots v_2 v_1)^{t-1}.$$

The string β is a supersequence of every string over Σ of length less than t and thus prevents a consistent subsequence from having length less than t. However β is not a supersequence of any string containing $\geq t$ vertices in order of increasing subscript.

To prove the theorem, we must prove two claims; (i) if G has an independent set of size t then C has a consistent subsequence (of length t) and (ii) if C has a consistent subsequence then G has an independent set of size t.

Proof of (i). Let $U = \{v_{i_1}, v_{i_2}, \ldots, v_{i_t}\}$ with $1 \leq i_1 < i_2 < \ldots < i_t \leq p$, be an independent set of G of size t. It is clear that the string $\gamma = v_{i_1} v_{i_2} \ldots v_{i_t}$ represents a common subsequence, of length t, of P. The string γ will not be a subsequence of β because γ has t vertices in order of increasing subscript. Hence γ is a consistent subsequence of C.

Proof of (ii). Let $\gamma = v_{i_1} v_{i_2} \ldots v_{i_u}$ be a consistent subsequence of C. It is immediate that $u \geq t$ since γ is a non-subsequence of β. The set $U = \{v_{i_1}, v_{i_2}, \ldots, v_{i_u}\}$ represents an independent set, of size u, of G. Since every consistent subsequence must be a list of vertices ordered by increasing subscript, for every pair of vertices, v_x and v_y $(x < y)$ in γ there can be no edge e_i connecting them in G. For otherwise the string

$$\alpha_i = v_1 v_2 \ldots v_{x-1} v_{x+1} \ldots v_p v_1 v_2 \ldots v_{y-1} v_{y+1} \ldots v_p \in P$$

would prevent v_x and v_y from being in a common subsequence of P. □

7.2.3 Longest consistent subsequence when $|P| = 1$ and $|N|$ is unbounded

See table entry D3.

Let α be the single positive string. If there exists a consistent subsequence then α must be a consistent subsequence since, if any subsequence of α is a common non-

subsequence of N, then α itself must be a common non-subsequence of N. Therefore, if α is a common non-subsequence of N, which can be checked in polynomial time, then α is the longest consistent subsequence of P and N, otherwise there is no consistent subsequence.

7.2.4 Shortest or Longest consistent subsequence when $|P|$ and $|N|$ are bounded

See table entries C6 and D6.

In this section, we describe a polynomial-time algorithm to find both the shortest and the longest consistent subsequence when both $|P|$ and $|N|$ are bounded. The algorithm is explained in terms of $|P| = |N| = 2$ but can be easily extended to work for any fixed $|P|$ and $|N|$. The positive strings are labelled α_1 and α_2 and the negative strings β_1 and β_2.

The algorithm uses a dynamic programming approach that is a generalisation and an extension of that used in Section 6.3.1 to find the *Shortest Maximal Common Subsequence* of a fixed number of (positive) strings. It is based on a table that relates the ith prefix $\alpha_1^i = \alpha_1[1 \ldots i]$ of α_1 and the jth prefix $\alpha_2^j = \alpha_2[1 \ldots j]$ of α_2.

Given the positive strings α_1,α_2 and the negative strings β_1,β_2 of lengths m, n, p, q respectively ($m \leq n$), we define the set S_{ij} for each $i = 0, \ldots, m$, $j = 0, \ldots, n$, by

$S_{ij} = \{(r, (w, x, y, z)) : \alpha_1^i$ and α_2^j have a common subsequence γ of length r, ending at $\gamma[r] = \alpha_1[w] = \alpha_2[x]$, and $sp(\beta_1, \gamma) = y, sp(\beta_2, \gamma) = z\}$ $(1 \leq i \leq m)$ $(1 \leq j \leq n)$ and

$$\begin{aligned} S_{i0} &= \{(0, (0,0,0,0))\} \qquad (0 \leq i \leq m), \\ S_{0j} &= \{(0, (0,0,0,0))\} \qquad (1 \leq j \leq n). \end{aligned}$$

For a set S of tuples and symbol a, we define

$$\begin{aligned} extend(S, a) = \{&(r, (next_{\alpha_1}(w, a), next_{\alpha_2}(x, a), \\ &next_{\beta_1}(y, a), next_{\beta_2}(z, a)) : (r-1, (w, x, y, z)) \in S\}. \end{aligned}$$

The algorithm is based on a dynamic programming scheme for the sets S_{ij} defined above. Evaluation of S_{mn} reveals the lengths of the shortest, the longest and indeed, all consistent subsequences. A tuple $(r, (w, x, y, z)) \in S_{mn}$ represents a consistent subsequence if and only if $y = p+1$ and $z = q+1$. The lowest (highest) r from such a tuple is the length of the shortest (longest) consistent subsequence. Furthermore, by applying suitable tracebacks through the array, a shortest, a longest, and all consistent subsequences can be found.

The basis of the dynamic programming scheme is contained in the following lemma.

Lemma 7.2.1

$$\begin{aligned} (i)\ \textit{If } \alpha_1[i] &= \alpha_2[j] = a \textit{ then} \\ S_{ij} &= S_{i-1,j} \cup S_{i,j-1} \cup \textit{extend}(S_{i-1,j-1}, a). \\ (ii)\ \textit{If } \alpha_1[i] &\neq \alpha_2[j] \textit{ then} \\ S_{ij} &= S_{i-1,j} \cup S_{i,j-1}. \end{aligned}$$

Proof
(i) It is straightforward that every common subsequence of α_1^i and α_2^{j-1} and every common subsequence of α_1^{i-1} and α_2^j is a common subsequence of α_1^i and α_2^j. It is also straightforward that every common subsequence of α_1^{i-1} and α_2^{j-1}, with the symbol a appended, will be a common subsequence of α_1^i and α_2^j. If a string γ is a common subsequence of α_1^i and α_2^j and γ ends in the symbol a then $\gamma - a$ must be a common subsequence of α_1^{i-1} and α_2^{j-1}. If a string γ is a common subsequence of α_1^i and α_2^j and γ does not end in symbol a then clearly γ is a common subsequence of α_1^i and α_2^{j-1} or γ is a common subsequence of α_1^{i-1} and α_2^j. With regard to β_1 and equivalently β_2, the behaviour of *next* ensures their conditions are always satisfied. It is clear that $sp(\beta_1, \gamma) = g$ if and only if $sp(\beta_1, \gamma + a) = next_{\beta_1}(g, a)$. (ii) This is simply case (i) restricted. □

Analysis of the consistent subsequence algorithm

The number of cells in the dynamic programming array is $O(mn)$ and the number of tuples in S_{ij} is bounded by $O(ij.pq.\min(i,j))$. If the tuples $(r,(w,x,y,z))$ in S_{ij} are stored in order of increasing w, then by increasing x and so on, then the amount of work done in computing the contents of S_{ij} is, in case (i) bounded by a constant times $|S_{i-1,j}| + |S_{i-1,j-1}| + |S_{i,j-1}|$, and in case (ii) bounded by a constant times $|S_{i-1,j}| + |S_{i,j-1}|$. The lengths of both a shortest and a longest consistent subsequences can therefore be returned in time bounded by $O(m^3n^2pq)$.

This gives us the following two theorems.

Theorem 7.2.3 *The length of a shortest consistent subsequence, when $|P|$ and $|N|$ are bounded, can be found in polynomial time.*

Theorem 7.2.4 *The length of a longest consistent subsequence, when $|P|$ and $|N|$ are bounded, can be found in polynomial time.*

Recovering a shortest or longest consistent subsequence

Recovering a shortest (longest) consistent subsequence involves a traceback through the dynamic programming table, starting at S_{mn} and ending at S_{00}, during which the sequence is constructed in reverse order. The subsequence δ is initially the empty string. A tuple, $(r,(w,x,p+1,q+1))$, in S_{mn} giving rise to a shortest (longest) consistent subsequence is chosen and $\alpha_1[w](=\alpha_2[x])$ is prepended to δ. The second step of the traceback occurs at S_{wx} where the tuple $(r,(w,x,p+1,q+1))$ must have been created. Any tuple from the set $S_{w-1,x-1}$ which could have given rise to $(r,(w,x,p+1,q+1))$ in S_{wx} is chosen and the process repeated until S_{00} is reached.

7.3 Consistent Supersequence Problems

7.3.1 Consistent supersequence when $|P|$ is unbounded and $|N|$ is bounded ($|N| \geq 2$)

See table entry B4.

Theorem 7.3.1 *Determining whether there exists a consistent supersequence is* **NP***-complete even when* $|N| = 2$.

Proof
Given an instance of the Vertex Cover problem, well-known to be **NP**-complete [19], on the graph G with vertex set $\{v_1, v_2, \ldots, v_p\}$ and edge set $\{e_1, e_2, \ldots, e_q\}$, and with t the target size for a vertex cover, we construct an instance C of the Consistent Supersequence problem, over the alphabet $\Sigma = \{0,1\}$, as follows. The transformation is a straightforward extension of that used by Räihä and Ukkonen [56] to prove the SCS problem **NP**-complete over a binary alphabet. In our notation we make the following substitutions; $\bar{N} \to V$, $\bar{E} \to E$, $s(sink) \to *$, $t \to p$, $r \to q$, and $k \to t$.

For $c = \max(p,q)$, the following strings are useful in the construction. The first group relates to the vertices of G;

$$\begin{aligned}
V &= (1^{7c})^{p+1}, \\
V_i &= (1^{7c})^i 0 (1^{7c})^{p+1-i} \qquad (1 \leq i \leq p), \\
V_{i_1,i_2,\ldots,i_u} &= (1^{7c})^{i_1} 0 (1^{7c})^{i_2-i_1} 0 \ldots (1^{7c})^{i_u-i_{u-1}} 0 (1^{7c})^{p+1-i_u} \\
&\qquad (1 \leq i_1 < \ldots < i_u \leq p), \\
V_* &= (1^{7c}0)^p (1^{7c}).
\end{aligned}$$

The second group of strings is analogous to the first and relates to the edges of G;

$$E = (0^{7c})^{q+1},$$

$$\begin{aligned} E_j &= (0^{7c})^j 11 (0^{7c})^{q+1-j} \qquad (1 \le j \le q), \\ E_{j_1,j_2,\ldots,j_v} &= (0^{7c})^{j_1} 11 (0^{7c})^{j_2-j_1} 11 \ldots (0^{7c})^{j_v-j_{v-1}} 11 (0^{7c})^{q+1-j_v}, \\ & \qquad (1 \le j_1 < \ldots < j_v \le q), \\ E_* &= (0^{7c}11)^q (0^{7c}). \end{aligned}$$

Hence V_i is a subsequence of $V_{i_1,i_2,\ldots,i_u}$ if and only if i is contained in the list $i_1, i_2, \ldots, i_u$, and V and V_i are subsequences of V_* for all i. The analogous relationships are true for E, E_j, $E_{j_1,j_2,\ldots,j_v}$, and E_*. Set P contains $q+1$ strings. The first is

$$T = EV_*E_*VE_*V_*E.$$

For each edge, $e_j = \{v_y, v_z\}$ $1 \le j \le q$, P contains the string

$$\alpha_j = E_j V_y V_z E_j.$$

Set N contains the two strings

$$\begin{aligned} \beta_1 &= 0^{7c(4q+4)+2p+t+1}, \\ \beta_2 &= 1^{7c(3p+3)+6q+1}. \end{aligned}$$

The following two claims are proved in [56].

Claim 7.3.1 *If G has a vertex cover of size t then P has a common supersequence with $7c(4q+4)+2p+t$ 0's and $7c(3p+3)+6q$ 1's.*

Claim 7.3.2 *If P has a common supersequence of length $7c(4q+3p+7)+6q+2p+t$ then G has a vertex cover of size t.*

The following example will make the transformation clearer. For vertex set $\{v_1, v_2, v_3\}$, edge set $\{\{v_1, v_2\}, \{v_2, v_3\}\}$, and $t = 1$ the target size for a vertex cover, vertex v_2 alone is a suitable vertex cover. The SCS γ of the derived set P together with the strings of P embedded in γ are as shown;

$$\begin{aligned} \gamma &= E_1V_*E_*V_2E_*V_*E_2 \\ T &= E\ V_*E_*V\ E_*V_*E \\ \alpha_1 &= E_1V_1 \quad V_2E_1 \\ \alpha_2 &= \qquad E_2V_2 \quad V_3E_2. \end{aligned}$$

To prove the theorem, we must prove two assertions; (i) if G has a vertex cover of size t then C has a consistent supersequence and (ii) if C has a consistent supersequence then G has a vertex cover of size t.

Proof of (i). If G has a vertex cover of size t then, by Claim 7.3.1, P has a common supersequence with $7c(4q+4)+2p+t$ 0's and $7c(3p+3)+6q$ 1's. Such a string has too few 0's to be a supersequence of β_1 and too few 1's to be a supersequence of β_2 and is therefore a consistent supersequence of C.

Proof of (ii). A consistent supersequence of C must have fewer than $7c(4q+4)+2p+t+1$ 0's (or it would be a supersequence of β_1) and fewer than $7c(3p+3)+6q+1$ 1's (or it would be a supersequence of β_2). It must therefore have $7c(4q+3p+7)+6q+2p+t$ or fewer characters in total which, by Claim 7.3.2, implies that G has a vertex cover of size t.

This completes the proof of the theorem. □

7.3.2 Shortest consistent supersequence when $|P| = 1$ and $|N|$ is unbounded

See table entry E3.

Let α be the single positive string. If there exists a consistent supersequence then α must be a consistent supersequence since, if any supersequence of α is a common non-supersequence of N, then α itself must be a common non-supersequence of N. Therefore, if α is a common non-supersequence of N, which can be checked in polynomial time, then α is the shortest consistent supersequence of P and N, otherwise there is no consistent supersequence.

7.3.3 Shortest consistent supersequence when $|P|$ and $|N|$ are bounded

See table entry E6.

In this section, we describe a polynomial-time algorithm to find the shortest consistent supersequence when both $|P|$ and $|N|$ are bounded. As in Section 7.2.4, the algorithm is explained in terms of $|P| = |N| = 2$ but can be easily extended to work for any fixed $|P|$ and $|N|$. The algorithm is a straightforward extension of that used in Section 6.3.2 to find the *Longest Minimal Common Supersequence* of a fixed number of (positive) strings. The positive strings are labelled α_1 and α_2 and the negative strings are labelled β_1 and β_2.

Given the positive strings α_1,α_2 and the negative strings β_1,β_2 of lengths m, n, p, q respectively ($m \leq n$), we define the set T_{ij} for each $i = 0, \ldots, m$, $j = 0, \ldots, n$, by

$T_{ij} = \{(r, (w, x, y, z)) :$ there exists a minimal common supersequence γ of α_1^i and α_2^j, of length r, such that $lp(\alpha_1, \gamma) = w, lp(\alpha_2, \gamma) = x, lp(\beta_1, \gamma) = y, lp(\beta_2, \gamma) = z\}$

$(1 \leq i \leq m)$ $(1 \leq j \leq n)$ and

$$\begin{aligned} T_{00} &= \{(0,(0,0,0,0))\}, \\ T_{i0} &= \{(i,(i,lp(\alpha_2,\alpha_1^i),lp(\beta_1,\alpha_1^i),lp(\beta_2,\alpha_1^i)))\} \quad (1 \leq i \leq m), \\ T_{0j} &= \{(j,(lp(\alpha_1,\alpha_2^j),j,lp(\beta_1,\alpha_2^j),lp(\beta_2,\alpha_2^j)))\} \quad (1 \leq j \leq n). \end{aligned}$$

For string α of length m, position i and symbol a, we define

$$f_\alpha(i,a) = \begin{cases} i+1 & \text{if } \alpha[i+1] = a \\ i & \text{otherwise.} \end{cases}$$

For a set, S, of tuples and symbol a, we define

$$\begin{aligned} extend'(S,a) = \{(r,(&f_{\alpha_1}(w,a),f_{\alpha_2}(x,a), \\ &f_{\beta_1}(y,a),f_{\beta_2}(z,a)) : (r-1,(w,x,y,z)) \in S\}. \end{aligned}$$

The algorithm is based on a dynamic programming scheme for the sets T_{ij} defined above. So evaluation of T_{mn} reveals the length of a shortest consistent supersequence. A tuple $(r,(m,n,y,z)) \in T_{mn}$ represents a consistent supersequence if and only if $y < p$ and $z < q$. The lowest r from such a tuple is the length of a shortest consistent supersequence. Furthermore, by applying suitable tracebacks in the array, a shortest consistent supersequence and indeed all minimal consistent supersequences can be found.

The basis of the dynamic programming scheme is contained in the following lemma.

Lemma 7.3.1

$$\begin{aligned} (i)\ \textit{If } \alpha_1[i] &= \alpha_2[j] = a \textit{ then} \\ T_{i,j} &= extend'(T_{i-1,j-1},a). \\ (ii)\ \textit{If } \alpha_1[i] &= a \neq b = \alpha_2[j] \textit{ then} \\ T_{i,j} &= \{(r,(i,f_{\alpha_2}(x,a),f_{\beta_1}(y,a),f_{\beta_2}(z,a))) \\ &\qquad : (r-1,(i-1,x,y,z)) \in T_{i-1,j}\} \\ &\quad \cup \{(r,(f_{\alpha_1}(w,b),j,f_{\beta_1}(y,b),f_{\beta_2}(z,b))) \\ &\qquad : (r-1,(w,j-1,y,z)) \in T_{i,j-1}\}. \end{aligned}$$

Proof
See Section 6.3.2. □

Analysis of the consistent supersequence algorithm

The number of cells in the dynamic programming array is $(m+1)(n+1)$ and the number of tuples in each cell is bounded by $O(ij.pq.\min(i,j))$. The term $\min(i,j)$ comes from the range of possible values for r; $\max(i,j) \leq r \leq i+j$. If the tuples $(r,(w,x,y,z))$ in T_{ij} are stored in order of increasing w, then by increasing x and so on, then the amount of work done in computing the contents of T_{ij} is, in case (i) bounded by a constant times $|T_{i-1,j-1}|$, and in case (ii) bounded by a constant times $|T_{i-1,j}| + |T_{i,j-1}|$. The length of a shortest consistent supersequence can therefore be returned in time bounded by $O(m^3n^2pq)$ $(m \leq n)$.

This gives us the following theorem.

Theorem 7.3.2 *The length of a shortest consistent supersequence, when $|P|$ and $|N|$ are bounded, can be found in polynomial time.*

Recovering a shortest consistent supersequence

Recovering a shortest consistent supersequence involves a traceback through the dynamic programming table, starting at T_{mn} and ending at T_{00}, during which the sequence is constructed in reverse order. To facilitate the traceback, when a tuple t in T_{ij} is created, it should have a pointer or pointers associated with it, indicating which tuple(s) in $T_{i-1,j}$,T_{ij} or $T_{i,j-1}$ led to the creation of t. The supersequence δ is initially the empty string. A path is then followed from the appropriate tuple in T_{mn} to T_{00}. When a pointer from a tuple $(r,(w,x,y,z))$ is followed to a tuple $(r',(w-1,x',y',z'))$ then the symbol $\alpha[w]$ is prepended to δ and when a pointer from a tuple $(r,(w,x,y,z))$ is followed to a tuple $(r',(w',x-1,y',z'))$ then the symbol $\beta[x]$ is prepended to δ.

7.3.4 Longest consistent supersequence when $|P|$ is unbounded and $|N| = 1$

See table entry F5.

When $|N| = 1$, it is clear that the alphabet size, $|\Sigma|$, is larger than $|N|$. Whenever this is the case, there does not exist a longest consistent supersequence. This is because there must be at least one symbol $a \in \Sigma$ that is not the last character of any string in N. Therefore, any consistent supersequence could have an arbitrary number of a's appended and remain a common non-supersequence of the strings in N.

7.4 Open Problems

Finding a longest consistent supersequence when both $|P|$ and $|N|$ are bounded remains open. The algorithm in Section 7.3.3, to find a shortest consistent supersequence when $|P|$ and $|N|$ are bounded, can find the longest *minimal* consistent supersequence in polynomial time. Timkovsky showed, in a private communication, how to find, in polynomial time, a longest consistent supersequence when $|P|$ and $|N|$ are bounded and there exists a longest non-supersequence of N, which can be tested in polynomial time [58]. However there are instances where no longest non-supersequence exists for N but a longest consistent supersequence exists for P and N as the following example shows;

$$\begin{aligned} P &= \{aba\} \\ N &= \{bb, aab, baa\}. \end{aligned}$$

Assuming the alphabet $\Sigma = \{a, b\}$, the longest consistent supersequence for P and N is aba but the string a^x is a non-supersequence of N for all integers x.

The complexity of determining whether there exists a consistent subsequence, and finding the length of a shortest consistent subsequence, when $|P|$ is unbounded and $|N| = 1$ over an alphabet of fixed size, remain open. All the other **NP**-complete consistent sequence problems remain **NP**-complete when the alphabet size is fixed at 2 as shown by the following **NP**-completeness results which apply to a binary alphabet. The entries in Tables 7.1 and 7.2 to which they apply, directly or by implication, are given in parenthesis: Maier [44] – finding the length of the longest common subsequence (D1, D4-D5); Räihä and Ukkonen [56] – finding the length of the shortest common supersequence (E1, E4-E5); Middendorf [47] – finding the length of the shortest common non-subsequence (C1-C3); L. Zhang [73] – finding the length of the longest common non-supersequence (F1-F3); Jiang and Li [38] – finding a consistent supersequence when $|P| \geq 2$ is bounded and $|N|$ is unbounded (B1-B2, E2); Theorem 7.2.1 – finding a consistent subsequence when $|P| \geq 2$ is bounded and $|N|$ is unbounded (A1-A2, D2); Theorem 7.3.1 – finding a consistent supersequence when $|N| \geq 2$ is bounded (B4, F4).

		1	2	3	4	5	6
	Existence Problem	$\|P\|, \|N\|$ Unbounded	$\|P\| \geq 2$ Bounded	$\|P\| = 1$	$\|N\| \geq 2$ Bounded	$\|N\| = 1$	$\|P\|, \|N\|$ Bounded
G	Consistent Substring	P (I1)→					
H	Consistent Superstring	NPC [38]	P [38]		Open	P §7.5.2	P [38]

Table 7.3: Consistent String Existence Problems

		1	2	3	4	5	6
	Optimisation Problem	$\|P\|, \|N\|$ Unbounded	$\|P\| \geq 2$ Bounded	$\|P\| = 1$	$\|N\| \geq 2$ Bounded	$\|N\| = 1$	$\|P\|, \|N\|$ Bounded
I	Shortest/Longest Con. Substring	P §7.5.1					
J	Shortest Con. Superstring	NPH [18]→	Open	P §7.5.3	NPH [18]→		Open
K	Longest Con. Superstring	NPH (H1)→	Open	Open	Open	$\not\exists$ §7.5.4	Open

Table 7.4: Consistent String Optimisation Problems

7.5 Consistent String Problems

Recall the following definitions from Section 1.6.3 Given two sets, P and N, of strings, a *consistent substring* of P and N is a string that is a common substring of P and a common non-substring of N. A *consistent superstring* is defined similarly.

There are two existence and four optimisation problems. Given two sets, P and N, of strings, does there exist a consistent substring (superstring)? If a consistent substring (superstring) does exist, what is the length of the shortest/longest? As for the consistent sequence problems, when a problem is characterised as being in **P**, the most efficient solution is not sought, the aim is merely to show that it is solvable in polynomial time. See Section 1.6.3 for details of previous work. The known complexities of the problems are summarised in Tables 7.3 and 7.4, together with the source for each characterisation. The tables use the same format as the tables for the consistent sequence problems.

7.5.1 Shortest/Longest consistent substring

See table entries I1-I6.

All the problems on substrings can be solved in polynomial time using a suffix tree [46, 67, 69]. The suffix tree of a string is unique. To simplify the structure of the suffix tree T of a string α, a unique terminal symbol is appended to α prior to the construction of T. This ensures that no suffix of α is a prefix of any other suffix which in turn guarantees the following property. Every suffix of α is represented by a leaf node and every leaf node represents a suffix of α. Every edge of T is labelled with a non-empty substring of α. The concatenation of the the edge labels encountered in a path from the root to a leaf node form the suffix represented by that leaf node. Every node has at least two children and no two edge labels leading from a node start with the same symbol.

Having built the suffix tree T for α_1, T can be pruned using strings $\alpha_2, \ldots, \alpha_k$ so that there is a one-to-one correspondence between the common substrings of $\alpha_1, \ldots, \alpha_k$ that cannot be extended to the right, and the leaf nodes of T. Moreover, the string represented by the path from the root to a leaf node is the common substring corresponding to that leaf node. This pruning can be performed in time proportional to the combined lengths of $\alpha_2, \ldots, \alpha_k$.

To find the longest consistent substring of two sets, P and N, of strings, build a suffix tree T and prune as above using the strings in P. Now traverse T testing every common substring of P represented in T to see if it is a common non-substring of N. The longest such string is the longest consistent substring and the shortest such string is the shortest consistent substring. Since every string has only $O(n^2)$ substrings (where n is the string length), and checking whether a string is a non-substring of another can be achieved in linear time [10, 40], the whole process can be carried out in polynomial time.

It follows that the existence problem is also in P. It is not true that a consistent substring always exists as the following example shows;

$$\begin{aligned} P &= \{aabb, bbaa\} \\ N &= \{aaa, bbb\}. \end{aligned}$$

7.5.2 Consistent superstring when $|P|$ is unbounded and $|N| = 1$

See table entry H5.

Let $P = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$, $N = \{\beta\}$ and $|\beta| = n$. If β is a substring of any string in P then clearly no consistent superstring exists. If some symbol $x \in \Sigma$ does not appear in either the first or last position of β then a consistent superstring is

$$\gamma = \alpha_1 \# \delta_1 \# \alpha_2 \# \delta_1 \# \cdots \# \delta_1 \# \alpha_k,$$

where $\delta_1 = x^n$.

If every symbol of Σ appears in either the first or the last position of β then the alphabet must binary, and we can assume that the first symbol of β is 0 and the last symbol is 1. If β has the structure $0 \ldots 1 \ldots 0 \ldots 1$ (where the dots represent arbitrary strings of zero or more symbols) then a consistent superstring is

$$\gamma = \alpha_1 \# \delta_2 \# \alpha_2 \# \delta_2 \# \cdots \# \delta_2 \# \alpha_k,$$

where $\delta_2 = 0^n 1^n$.

If β has the structure $0 \ldots 1$ but not $0 \ldots 1 \ldots 0 \ldots 1$ then it must be of the form $0^x 1^y$ for some $x, y \geq 1$. If both x and y are greater than 2 then

$$\forall \alpha_i \in P \text{ such that } \alpha_i \text{ ends with 0, let } \alpha_i' = \alpha_i \# 101,$$

$$\forall \alpha_i \in P \text{ such that } \alpha_i \text{ ends with 1, let } \alpha_i' = \alpha_i \# 01.$$

A consistent superstring is

$$\gamma = \alpha_1' \# \alpha_2' \cdots \# \alpha_k'.$$

If $\beta = 0^x 1$ (i.e. $x \geq 2, y = 1$) then for $i = 1, \ldots, k$, let s_i be the length of the longest suffix of α_i consisting purely of the symbol 0. Any α_i for which $s_i \geq x$ cannot precede the symbol 1 in a consistent superstring. Therefore if there exists more than one α_i containing the symbol 1 and for which $s_i \geq x$ then no consistent superstring exists. Otherwise a consistent superstring can be formed as follows.

$$\begin{aligned} \text{Let } \alpha_i' &= \alpha_i \# 1 \qquad \forall \alpha_i \in P \text{ such that } s_i < x, \\ \gamma_1 &= \text{the concatenation of all the } \alpha_i', \\ \alpha_j &= \text{the only string containing 1 and for which } s_j \geq x, \text{ if it exists.} \end{aligned}$$

Then a consistent superstring is

$$\gamma = \gamma_1 \# \alpha_j \# 0^m \text{ where } m = \max_{1 \leq i \leq k, s_i \geq x} |\alpha_i|.$$

If $\beta = 01^y$ (i.e. $x = 1, y \geq 2$) then the existence of a consistent superstring can be checked in a way similar to the case when $\beta = 0^x 1$ (i.e. $x \geq 2, y = 1$).

Finally, if $\beta = 01$ (i.e. $x = y = 1$) then, for a consistent superstring to exist, all $\alpha_i \in P$ must have the form $1^v 0^w$ for $v, w \geq 0$, otherwise β would be a substring of some $\alpha_i \in P$. Therefore a consistent superstring is

$$\gamma = 1^m 0^m \text{ where } m = \max_{1 \leq i \leq k} |\alpha_i|.$$

7.5.3 Shortest consistent superstring when $|P| = 1$ and $|N|$ is unbounded

See table entry J3.

Let α be the string in P. If α is a common non-superstring of the strings in N then it is the shortest consistent superstring of P and N. If α is a superstring of any string in N then clearly no consistent superstring exists. This is easily checked in polynomial time.

7.5.4 Longest consistent superstring when $|P|$ is unbounded and $|N| = 1$

See table entry K5.

When $|N| = 1$, it is clear that the alphabet size is larger than $|N|$. When this is the case, there does not exist a longest consistent superstring. This is because there must be at least one symbol $x \in \Sigma$ that is not the last character of any string in N. Therefore, any consistent superstring could have an arbitrary number of x's appended and remain a common non-superstring of N.

7.5.5 Open Problems on consistent strings

The following problems remain open:

1) Can the existence of a consistent superstring be checked in polynomial time when the number of negative strings is bounded above but greater than 1? (Table entry H4)

2) Is there a polynomial time algorithm to find a shortest consistent superstring when the number of positive strings is bounded above but greater than 1? (J2)

3) If the answer to 2) is "no" then, is there a polynomial time algorithm to find a shortest consistent superstring when the numbers of positive and negative strings are bounded above ? (J6)

4) Is there a polynomial time algorithm to find a longest consistent superstring when the number of positive strings is bounded above ? (K2,K3)

5) Is there a polynomial time algorithm to find a longest consistent superstring when the number of negative strings is bounded above? (K4)

6) If the answers to 4) and 5) are "no" then, is there a polynomial time algorithm to find a longest consistent superstring when the numbers of positive and negative strings are bounded above? (K6)

In connection with 1), the algorithm in Section 7.5.2 used to check the existence of a consistent superstring (table entry H5) when there is only one negative string

can be extended to the case of two negative strings. The algorithm uses a case-by-case analysis of the structures of the two negative strings. We conjecture that the existence of a consistent superstring can be checked in polynomial time for any fixed number of negative strings but a more easily generalised method is required to prove this.

In connection with 2) and 3), the shortest *minimal* consistent superstring can be found in polynomial time when the number of positive strings is bounded. This is because there are a polynomial number of minimal common superstrings of P, each of which can be checked in polynomial time to see if it is a common non-superstring of N. In this context, minimal requires that every symbol in the superstring be necessary for it to be a superstring of the strings in P. Further, Jiang and Timkovsky [39] showed how to find, in polynomial time, a shortest consistent superstring when the number of positive strings is bounded above and every symbol of the alphabet appears at the end of some negative string.

Jiang and Timkovsky [39] showed how to find, in polynomial time, a shortest or a longest consistent superstring when there exists a common non-superstring for the set of negative stings, even when the sizes of both sets are unbounded. Rubinov and Timkovsky [58] showed that the existence of a common non-superstring can be checked in polynomial time.

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] L. Allison. Lazy dynamic-programming can be eager. *Information Processing Letters*, 43:207–121, 1992.

[3] L. Allison and T.I. Dix. A bit-string longest common subsequence algorithm. *Information Processing Letters*, 23:305–310, 1986.

[4] A. Apostolico. Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings. *Information Processing Letters*, 23:63–69, 1986.

[5] A. Apostolico, S. Browne, and C. Guerra. Fast linear-space computations of longest common subsequences. *Theoretical Computer Science*, 92:3–17, 1992.

[6] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[7] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and intractability of approximation problems. In *33rd FOCS*, pages 14–23, 1992.

[8] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *Journal of the A.C.M.*, 41:630–647, 1994.

[9] P Bonizzoni, M Duella, and G Mauri. Approximation complexity of longest common subsequence and shortest common supersequence over fixed alphabet. Technical Report 117/94, Università degli Studi di Milano, Dipartmento di Scienze dell'Informazione, 1994.

[10] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the A.C.M.*, 20:762–772, 1977.

[11] J.H. Bradford and T.A. Jenkyns. On the inadequacy of tournament algorithms for the n-SCS problem. *Information Processing Letters*, 38:169–171, 1991.

[12] F. Chin and C.K. Poon. Performance analysis of some heuristics for computing longest common subsequences. *Algorithmica*, 12:293–311, 1994.

[13] F.Y.L. Chin and C.K. Poon. A fast algorithm for computing longest common subsequences of small alphabet size. *Journal of Information Processing*, 13:463–459, 1990.

[14] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1995.

[15] V. Dančík and M. Paterson. Upper bounds for the expected length of a longest common subsequence of two binary sequences. *Random Structures and Algorithms*, 6, 1995. To appear.

[16] J. G. Deken. Some limit results for longest common subsequences. *Discrete Mathematics*, 26:17–31, 1979.

[17] D.E. Foulser, M. Li, and Q. Yang. Theory and algorithms for plan merging. *Artificial Intelligence*, 57:143–181, 1992.

[18] J. Gallant, D. Maier, and J.A. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20:50–58, 1980.

[19] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freedman, San Francisco, CA, 1979.

[20] D. Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. Technical report, Computer Science Division, University of California, 1991.

[21] F. Hadlock. Minimum detour methods for string or sequence comparison. *Congressus Numerantium*, 61:263–274, 1988.

[22] K. Hakata and H. Imai. The longest common subsequence problem for small alphabet size between many strings. In *Proceedings of the 3rd Annual Symposium on Algorithms and Computation*, volume 650 of *Lecture Notes in Computing Science.* Springer-Verlag, 1992.

[23] M.M. Halldorsson. Approximating the minimum maximal independence number. Technical report, Japan Advanced Institute of Science and Technology, 1993.

[24] J Hebrard. An algorithm for distinguishing efficiently bit-strings by their subsequences. *Theoretical Computer Science*, 82:35–49, 1991.

[25] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the A.C.M.*, 18:341–343, 1975.

[26] D.S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the A.C.M.*, 24:664–675, 1977.

[27] D.S. Hirschberg. Recent results on the complexity of common subsequence problems. In *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, chapter 12, pages 325–330. Addison-Wesley, 1983.

[28] W.J. Hsu and M.W. Du. Computing a longest common subsequence for a set of strings. *BIT*, 24:45–59, 1984.

[29] W.J. Hsu and M.W. Du. New algorithms for the LCS problem. *Journal of Computer and System Sciences*, 29:133–152, 1984.

[30] J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the A.C.M.*, 20:350–353, 1977.

[31] R.W. Irving. On approximating the minimum independent dominating set. *Information Processing Letters*, 37:197–200, 1991.

[32] R.W. Irving and C.B. Fraser. Two algorithms for the longest common subsequence of 3 (or more) strings. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 214–229. Springer-Verlag, 1992.

[33] R.W. Irving and C.B. Fraser. On the worst case behaviour of some approximation algorithms for the shortest common supersequence. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, volume 684 of *Lecture Notes in Computer Science*, pages 63–73. Springer-Verlag, 1993.

[34] R.W. Irving and C.B. Fraser. Maximal common subsequences and minimal common supersequences. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, volume 807 of *Lecture Notes in Computer Science*, pages 173–183. Springer-Verlag, 1994.

[35] S.Y. Itoga. The string merging problem. *BIT*, 21:20–30, 1981.

[36] G. Jacobson and K.P. Vo. Heaviest increasing/common subsequence problems. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 52–65. Springer-Verlag, 1992.

[37] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. Submitted to SIAM J. Comp., 1992.

[38] T. Jiang and M. Li. On the complexity of learning strings and sequences. *Theoretical Computer Science*, 119:363–371, 1992.

[39] T. Jiang and V. G. Timkovsky. Shortest consistent superstrings computable in polynomial time. *Theoretical Computer Science*, 143(1):113–122, 1995. To appear.

[40] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[41] S. K. Kumar and C. P. Rangan. A linear space algorithm for the LCS problem. *Acta Informatica*, 24:353–362, 1987.

[42] S. Kuo and G. R. Cross. An improved algorithm to find the length of the longest common subsequence of two strings. *Sigir Forum*, 23:89–99, 1989.

[43] R. Lowrance and R.A. Wagner. An extension of the string-to-string correction problem. *Journal of the A.C.M.*, 22:177–183, 1975.

[44] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the A.C.M.*, 25:322–336, 1978.

[45] W.J. Masek and M.S. Paterson. A faster algorithm for computing string editing distances. *J. Comput. System Sci.*, 20:18–31, 1980.

[46] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the A.C.M.*, 23:262–272, 1976.

[47] M. Middendorf. The shortest common nonsubsequence problem is NP-complete. *Theoretical Computer Science*, 108:365–369, 1993.

[48] M. Middendorf. More on the complexity of common superstring and supersequence problems. *Theoretical Computer Science*, 125:205–228, 1994.

[49] M. Middendorf. On finding minimal, maximal and consistent sequences over a binary alphabet. *Theoretical Computer Science*, 145(327):317–327, 1995. To appear.

[50] A. Mukhopadhyay. A fast algorithm for the longest common subsequence problem. *Information Sciences*, 20:69–82, 1980.

[51] E.W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[52] N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982.

[53] C.H. Papadimitriou and M. Yannakakis. Optimisation, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[54] P. H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[55] R. C. Prim. Shortest connection networks and some generalisations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[56] K.J. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16:187–198, 1981.

[57] C. Rick. New algorithms for the longest common subsequence problem. Technical report, Institut Für Informatik Der Universität Bonn, 1994.

[58] A. R. Rubinov and V.G. Timkovsky. String non-inclusion optimisation problems. Submitted to SIAM J. on Discrete Math., 1992.

[59] R. Sedgewick. *Algorithms.* Addison-Wesley, second edition, 1988.

[60] G. A. Stephen. *String Searching Algorithms.* World Scientific, 1994.

[61] J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57:131–145, 1988.

[62] S. Teng and F. Yao. Approximating shortest superstrings. In 34^{th} *FOCS*, 1993. To appear in *Theoretical Computer Science.*

[63] W.F. Tichy. The string-to-string correction problem with block moves. *ACM transactions on computer systems*, 2:309–321, 1984.

[64] V.G. Timkovsky. Complexity of common subsequence and supersequence problems and related problems. *Kibernetika*, 5:1–13, 1989. English translation in Cybernetics 25:565-580,1990.

[65] J.S. Turner. Approximation algorithms for the shortest common superstring problem. *Information and Computation*, 83:1–20, 1989.

[66] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[67] E. Ukkonen. On-line construction of suffix trees. Technical Report A-1993-1, University of Helsinki, 1993.

[68] R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the A.C.M.*, 21:168–173, 1974.

[69] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[70] C.K. Wong and A.K. Chandra. Bounds on the string editing problem. *Journal of the A.C.M.*, 23:13–16, 1976.

[71] S. Wu, U. Manber, G. Myers, and W. Miller. An O(NP) sequence comparison algorithm. *Information Processing Letters*, 35:317–323, 1990.

[72] M Yannakakis and F Gavril. Edge dominating sets in graphs. *SIAM Journal for Applied Mathematics*, 38(3):364–372, 1980.

[73] L. Zhang. On the approximation of longest common non-supersequences and shortest common non-subsequences. *Theoretical Computer Science*, 143(2):353–362, 1995. To appear.