ON THE KNIGHT'S TOUR PROBLEM & ITS SOLUTION BY GRAPH-THEORETICAL AND OTHER METHODS

Being a thesis submitted in application for the degree of M.Sc. in Computing Science in the University of Glasgow

by

D.J.W. STONE

January, 1969

CONTENTS

CHAPTER 1

Introduction

1.1 The Statement of the Problem The knight's tour problem may be stated in various ways, according to the degree of difficulty it is desired should be overcome; the original intention in embarking on this work was that of enumerating all the possible knight's tours on a standard 8 x 8 chessboard. A little later, when the magnitude of the problem became fully apparent, sights were dropped somewhat to the more realistic objective of carrying out this enumeration on a 6 x 6 board (one hesitates to describe a 6 x 6 chessboard as realistic). It was hoped that as a result of this, certain general properties of the knight's tours considered as whole class of objects might emerge, and hence lead to faster, more subtle ways of finding tours on the larger board. But to envisage why this reductio of scale should be necessary in the first place we must define what we mean by a knight's tour, and describe the processes involved in finding tours.

1.2. Legal Knight's Moves

Definition (We represent the chessboard as in figure 1.1.)

A legal knight's move on a chessboard (of side ≥ 3 units) consists of increments $\Delta x, \Delta y$, in the x-y coordinates of the knight s.t.

$$|\Delta x| + |\Delta y| = 3,$$

where $\Delta x, \Delta y$, are integers, &

$$0 < |\Delta x| < 3$$
$$0 < |\Delta y| < 3,$$

and furthermore,

$$n-1 \geq x + \Delta x \geq 0$$
$$n-1 \geq y + \Delta y \geq 0,$$

the board being of dimensions $n \times n$.

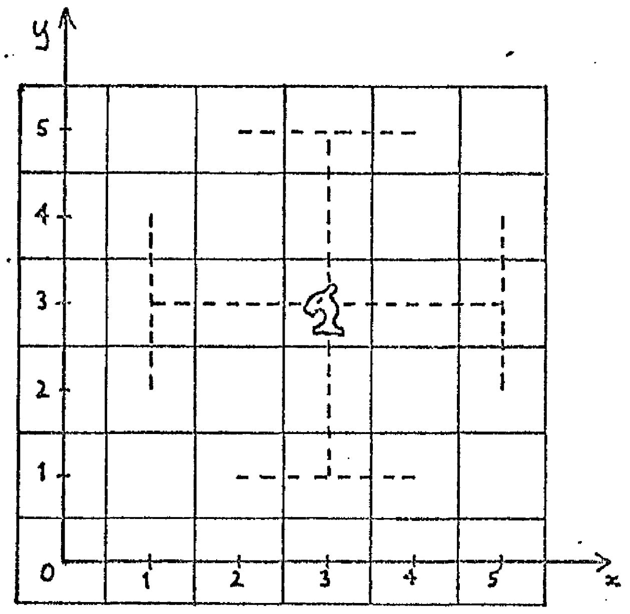


Fig. 1.1
6 x 6 chessboard

For an infinite chessboard, there are always 8 legal knight's moves that can be made (see figure 1.2.) corresponding to the increments:-

$$\Delta x = +1 \begin{cases} \Delta y = -2 \\ \Delta y = +2 \end{cases}$$
$$\Delta x = -1 \begin{cases} \Delta y = -2 \\ \Delta y = +2 \end{cases}$$
$$\Delta x = +2 \begin{cases} \Delta y = -1 \\ \Delta y = +1 \end{cases}$$
$$\Delta x = -2 \begin{cases} \Delta y = -1 \\ \Delta y = +1 \end{cases} ;$$

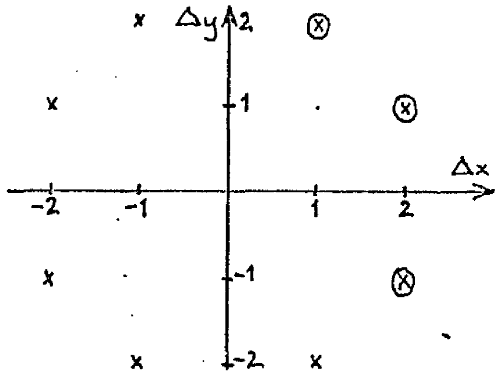


Figure 1.2.

Legal knight's moves

but in the more amenable finite cases, edge-effects creep in, and the number of legal moves/square may vary between 2 (e.g. square(0,0) in Figure 1.1.) and 8. The presence or otherwise of an edge effect for a square in any given position on a board of whatever (finite) size can easily be calculated, since we can always predict whether a move will take the knight 'overboard'.

(e.g.) If $y = 1$, $x = 0$, then for move to be legal,

$$0 \leq \Delta x < 3$$
$$-1 \leq \Delta y < 3,$$

which permits the 3 moves circled in Figure 1.2.

The full layout of the number of moves permitted for each square on a 6 x 6 board is as in Figure 1.3, and generalization to a 2n x 2n board (the reason for choosing an even-sided board will emerge later) shows that

No. of squares from which	2	legal knight's moves can be made	= 4
" " " " "	3	" " " " " " "	= 8
" " " " "	4	" " " " " " "	= 8n-12
" " " " "	6	" " " " " " "	= 8n-16
" " " " "	8	" " " " " " "	= $(2n-4)^2$

∴ Total no. of legal knight's moves on 2n x 2n board

$$8+24+32n-48+48n-96+32n^2 -128n+128 = 32n^2 -48n+16$$

$$= 16 . (2n-1).(n-1),$$

and the average number of legal knight's moves/square, $\bar{N}_{2n}$

is given by $\bar{N}_{2n} = 4/n^2.(2n-1).(n-1)$

The value of $\bar{N}$ is asymptotic to 8 (Figure 1.4), and for a 6 x 6 board (n = 3) is 40/9.

2	3	4	4	3	2
3	4	6	6	4	3
4	6	8	8	6	4
4	6	8	8	6	4
3	4	6	6	4	3
2	3	4	4	3	2

Figure 1.3

No. of legal knight's moves for each square on 6 x 6

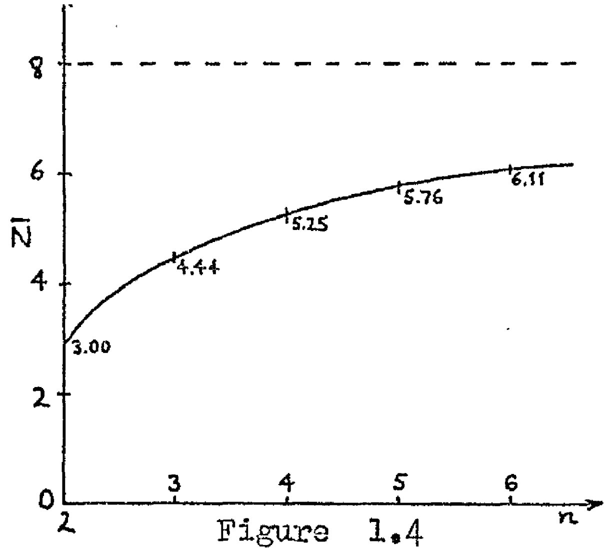


Figure 1.4

1.3 <u>The knight's Tour</u>

<u>Definition:</u> <u>A knight's tour</u> on a 2n x 2n chessboard is a sequence of $4n^2$ legal knight's moves (in no specified orientation) whereby each square is visited onee and only once, with the exception of the (arbitrary) starting and finishing squares, which must be the same.

1.4 <u>Graph and Tree Representations</u> We can represent the legal moves on the board by a linear graph, where the vertices denote squares, and an edge corresponds to a legal knight's move between the squares whose vertices it joins; such a graph (hereafter referred to as the <u>equivalent graph</u>) has been drawn for a 4 x 4 board in Figure 1.5. When the problem is presented in this form, the above definition is equivalent to:

<u>Definition:</u> <u>A knight's tour</u> on a 2n x 2n chessboard is a sequence of $4n^2$ moves such that the corresponding sequence of edges on the equivalent graph form a Hamiltonian cycle on that graph.

Note that this definition, following Berge's definition of a cycle rather than that of Saaty and Busacker, implies no orientation as was mentioned explicitly in the first variant. We define the tour this way deliberately, but arbitrarily; the question of whether a sequence of squares, and the sequence formed from it by traversing it in reverse order (which we clearly can do, since the sequence of squares is a closed loop) define two separate tours or only one is a moot one, and the latter choice has been made here.

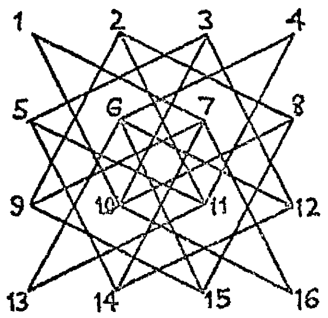


Figure 1.5
Equivalent graph for 4 x 4 board

Now that a knight's tour has been defined, it is apparent that, to find all possible tours, we must test all sequences of squares of length 36 to see if they comply with the conditions of legality and non-repetitiveness. It is axiomatic that if a string of squares does not satisfy these rules, then neither can any sequence formed by adding further squares to that string; so we can order this test by building up longer and longer strings, some of which may reach the length of 36 squares without violating the above criteria - these are the knight's tours we seek - while others may terminate after <36 moves due to violation of either condition. We are hence concerned with exhaustively searching a tree-structure and extracting from it only the branches of length 36.

Since, as we shall see, the number of branches rapidly becomes extremely large, with increasing board-size, it is of paramount importance to prune abortive branches as early as possible; in fact, time is so crucial when contemplating extension of a method to the full 8 x 8 board, that the method must stand or fall by its inherent speed.

1.5 <u>Size of Tree-Search</u> Figure 1.4 showed how the average number of legal moves for each square increases with increasing board-size; this might be deemed a relatively insignificant effect until we qualify it by the observation that the strings, both knight's tours and abortive attempts, increase in length as the number of squares on the board, i.e. the tree must be searched to a much greater depth on a larger board, while the number of choices at each level is also (slightly) greater. That is,

$$\text{Size of tree-search} \propto (\bar{N}_{2n})^{4n^2}$$

1.6 <u>Legal Tour Moves</u> It will be convenient to make the

<u>Definition:</u> A legal tour move (l.t.m.) is a legal knight's move to a square whose corresponding node in the search-tree has no predecessor the same as itself.

That is, for an l.k.m. to be an l.t.m., the non-repetitiveness criterion must be satisfied.

It will later be seen that the method used ensures 'built-in' legality, so that only l.k.m.'s are considered, and the tree-search is concentrated on the matter of finding l.t.m.'s. The generation of l.k.m.'s by inserting the definition into the program can easily be effected, but consumes too much time.

1.7 Boards of Different sizes In 1.3 the definition of a knight's tour was restricted to a board of even sides (i.e. 4 x 4, 6 x 6, 8 x 8 etc); the reason for this will now be explained, as it is relevant to the choice of a 6 x 6 board for testing reasons.

The result may be stated: 'No chessboard of odd dimensions (the board need not be square) has a knight's tour (i.e. Hamiltonian cycle) on it'. This follows as a corollary to Konig's theorem, which is stated as follows:-

Konig's Theorem A graph contains no elementary cycles of odd length if and only if it is bichromatic.

A cycle is elementary if each vertex appearing in it occurs only once, so that the knight's tour is an example of an elementary cycle.

Furthermore, a chessboard of odd dimensions has an odd number of squares, so that a Hamiltonian cycle on the graph of the board must be of odd length. Finally, we note that a l.k.m. as defined in 1.2 takes the knight from a black to a white square, if the board is coloured in the usual manner shown in Figure 1.1. Considering now the vertices of the equivalent graph to be coloured in the same fashion, we deduce that any two adjacent vertices of the graph are of opposite colours and the graph is bichromatic. Hence, Konig's Theorem shows there are no knight's tours on an odd-sided board.

We may add in passing that neither can the equivalent graph of such a board possess any factors; a factor must consist of one or more disjoint cycles, and in the latter case, since the sum of the lengths of the disjoint parts is odd, the length of at least one individual part must be odd, which contradicts the above result. We mention this here as factor-seeking is one of the ways and means discussed in Chapter 2.

It was stated at the outset that the enumeration of knight's tours on the 8 x 8 board seemed on preliminary investigation to be too large; the question which then arose was to what size the board should be reduced still to pose an adequate test of the method.

By the above considerations, 5 x 5 and 7 x 7 could be discounted as giving no knight's tours in any case (although they might prove useful for assessing the length in machine-time of the tree-search), and as will be shown in Chapter 3, the 4 x 4 board yields no tours either. The choice was thus resolved to searching on a 6 x 6 board.

1.7 Summary of Thesis The literature is not very forthcoming on the subject of computational algorithms for the total solution of the knight's tour problem with the exception of Duby's paper, and this paper, which uses a method remarkably similar to that of Chapter 3, was only discovered after most of the development work described there had been completed, and was referred to rather as a guide to expected speed than anything else; the published works mentioned in Appendix 3 have been sources of 'non-ideas' on the whole. Graphical nomenclature is taken from Berge (as is Fortet's method, 2.8) and tree-related terms from Scoins.

Chapter 2 is a review of the methods of solution which were considered, each method being surveyed as appropriate and discussed as to its feasibility. Chapter 3 details the method used from its early development to its final form, and includes a number of flow charts; Chapter 4 shows how this program was then used to analyse other tree-searching methods and also the knight's tours themselves, particularly with reference to classifying them by their symmetry properties. Chapter 5 comments, in conclusion, on the results obtained, and assesses the potential of methods not investigated in Chapter 2.

Flow charts have been included in the text where it has been felt that they could preclude the necessity for undue verbosity, but they are not presented in full detail with coding, this latter being included for reference in Appendix 1. 'One knight's tour looks very much like another', and certainly it would be pointless to include the output of the whole collection, but certain tours, strings, and other results are of special interest and have been gathered into Appendix 2. Appendix 3 contains the list of books and papers referred to within the text and during its preparation.

The reader who wishes to avoid <u>all</u> details of the programming could omit Chapter 3, but the nsxt chapter would induce back-references. Sections 3.4, 3.6, and 3.8 can be skipped without such consequences.

CHAPTER 2

A Review of Methods of Solution

2.1 Introduction There is a great diversity of methods by which this problem could conceivably be tackled: quite apart from the adoption of differing programming tricks within a fixed theoretical framework (as described at length in Chapter 3), there are theoretical variants aplenty, some not even based on like disciplines. Graph theory, and tree-searching, offer one approach we have already mentioned, but combinatorial considerations can be coupled with these to give feasible methods, too. The investigation of symmetry properties of knight's tours is certainly a means of reducing the burden of counting tours, but is more a property of knight's tours themselves, and is therefore not treated here but deferred till Chapter 4. Dynamic programming is a technique designed specifically to deal with multistage decision processes such as knight's tours, although serious difficulties seem to beset this method; and a simply combinatorial idea based on a heuristic transformation might be applied with some success.

Figure 2.1 indicates the ways and means considered:-

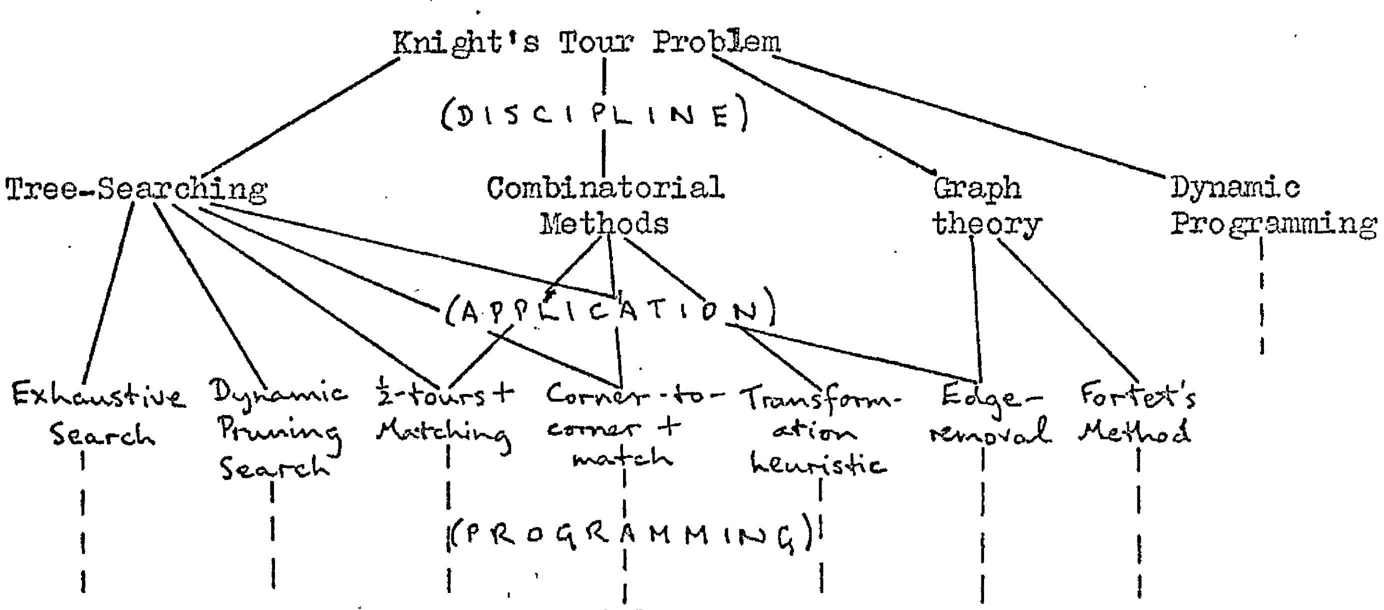


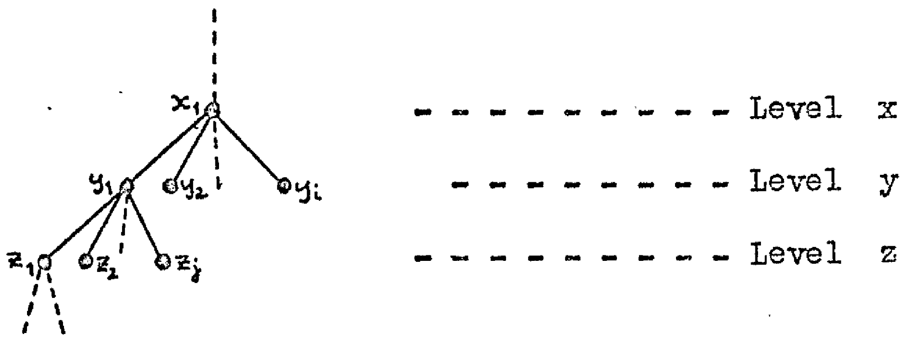
Figure 2.1

2.2 Exhaustive Tree-Search This is the method which has been mentioned in passing in 1.4, and which forms the bulk of the following discussion; its development as a program is described in detail in the next chapter: we only give the major features of the method here.

The basic scheme which is followed throughout is that at each stage of a knight's tour, there are a number of possible l.k.m's open to the knight, each of which must be tested in turn (some arbitrary order being adopted for this) to choose only the l.t.m's. A diagram will make the procedure adopted clear:-

Figure 2.2

Simple Tree-Search



Suppose the knight is moved from square x_1 to y_1 initially, and we want to choose the next move from the l.k.m's $z_1, \ldots\ldots, z_j$. These are scanned in some order (the obvious way: in ascending numerical order with $z_1 < z_2 < \ldots\ldots < z_j$) and the first l.t.m. encountered is chosen, i.e. if z_1 is not a l.t.m. from y_1, then z_2 is tested and so on. When a l.t.m. has been found, then we conduct a similar search for l.t.m's at the level below z. On the other hand, if none of the z's is a l.t.m., the knight cannot move from y_1, so we must go back up the tree to level x, and try the next move consequent to x_1, which is a l.t.m., y_3, say.

By this simple procedure, we can execute an exhaustive left-to-right, top-to-bottom, 'yoyo' tree-search, and print out our path down the tree every time a depth of 36 below the root is reached.

In 1.4, the convention was adopted of defining the tour in an undirected fashion; nevertheless, we introduce orientation considerations into the problem as they afford an easy means of extracting 'clockwise' tours which we have defined to be the same as 'anti-clockwise' ones.

This extraction is very simply performed. We observe that the corner squares are only doubly-connected so that there exists just one entrance and just one exit (see Figure 2.3) for such a square. If the tour is commenced at square X and leaves via Y, it must return to X through Z, and vice versa. We can make a complete tree-search for tours beginning X, Y, ... , and in so doing we automatically find all the tours ending . . . . . , Z, X; these, when traversed in the opposite direction give all the tours beginning X, Z, ... , and ending . . . . Y, X, and if we reflect each one of these in the board's main diagonal, we arrive back at a set of tours starting X, Y, ... and ending . . . . Z, X - obviously those we found at first. This set of transformations is shown in Figure 2.4.

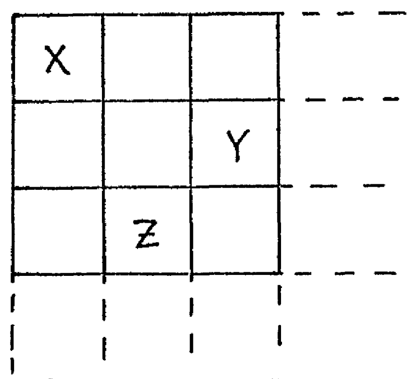


Figure 2.3 Corner Square Moves

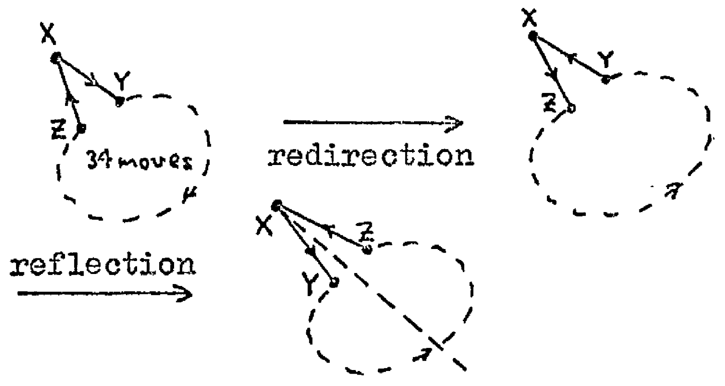


Figure 2.4 Diagonal Reflection and Redirection

Notice that no assumption of diagonal symmetry has been made here: the processes of redirection and reflection of a knight's tour assure finding a tour of the same orientation, but unless there is complete symmetry about the main diagonal in the original tour, the new tour will be a different one. We can see this by the following argument; let A be a knight's tour, and let ′ and * respectively represent the operations of redirection and reflection. Clearly we have

$$(A')' = A \quad - (i)$$
$$(A^*)^* = A \quad - (ii)$$

if

$$(A')^* = A$$
$$((A')^*)^* = A^*$$

out

$$((A')^*)^* = A', \text{ by (ii)}$$

$$\therefore \underline{A' = A^*}$$

We have tacitly assumed here the closure of the set of knight's tours under the two operations.

This result simply states that if a knight's tour undergoes the operations of redirection and reflection and the result is the same knight's tour, then the 35th square of the tour must be the mirror-image of the 2nd square, the 34th the image of the 3rd, and so on.

However, the original intention of restricting the tree-search to a particular direction of tour still holds good, for when we consider the knight's tours as a class, redirection and reflection will produce the same class, but its members will be ordered differently to the way theyappear in the tree-search.

So the undernoted convention is used: the board being numbered as in Figure 2.5, the knight starts at square 1, and always leaves the corner by square 9, re-enters it by square 14. This cuts the size of the search immediately by 50 per cent.

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

Figure 2.5 Board-Numbering

As we stated before, the chief task in attempting solution of the knight's tour problem is to reduce the tree-search as much as possible by pruning unproductive branches as early as possible. The question which then arises is how we recognise an unproductive branch; so far we have not bettered the stipulation of the problem itself, i.e. that if none of the legal successors to a node satisfies the nonrepetitivity criterion, then the subtree rooted at that node is unproductive. The most powerful additional criterion yet found will now be described: this is the basis of our method, and was found independently; it has previously appeared in an internal report by J.J. Duby (q.v.). Various weaker criteria were also investigated and will be discussed later.

2.2.1 Lookahead for Dead-ends In general, when the knight is situated at a square, it is faced with the option of making one of a number of moves (as many as 7); in certain circumstances, however, looking one move ahead reveals an untenable state of affairs which can only be resolved by one particular choice of move.

Although this method has been classified as a tree-search, a digression into graphical terminology will be helpful here.

Consider the knight positioned at vertex X (Figure 2.6), adjacent to vertices L, M, and N. If the knight moves to L then it has a further choice of 4 moves (dotted) subsequently; from M it has a choice of 3, and from N only one exit is open to it. Now assume we make the move XL. X will not now be visited again during the tour, and hence we have effectively removed the edges XM, and XN. The consequences of this are not serious for the vertex M which is still connected to 3 other vertices, but N is now only joined to P, so that there is no way of leaving N having once made the move PN at some later stage (note that we are obliged to visit N later in the tour).

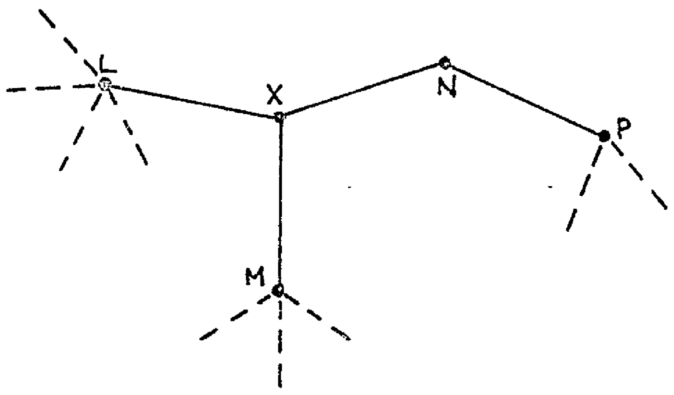


Figure 2.6
Lookahead

This condition has arisen because we have ignored a fundamental property of knight's tours, viz. each vertex of the equivalent graph of a knight's tour has a degree of 2. By moving to L, we have allowed the degree of vertex N to become 1, thus precluding any possibility of reaching a tour; similarly if we move to M. The knight at X has no option but to go to N, and the lookahead technique which recognises situations of this type can provide an extremely powerful tool for pruning the search-tree. Its effect on the tree is shown in Figure 2.7, and it is plain that the earlier in the search that the lookahead condition is fulfilled the larger the subtree will be pruned.

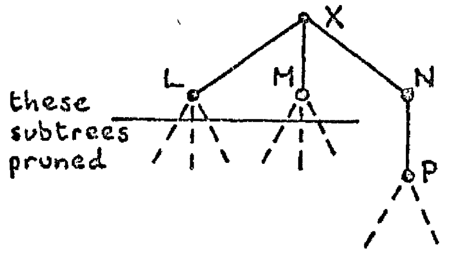


Figure 2.7 Lookahead Pruning

For convenience in describing the program in the next chapter, two items of nomenclature will be introduced here.

By a GOTO move, we will mean a legal tour move which is forced by the presence of the above condition, i.e. one of the possible subsequent moves is to a doubly-connected square.

By a TEST move we will mean any other legal move down the search-tree.

Referring again to Figure 2.7, there are a number of other benefits arising from the use of the criterion.

In the normal way, had only L and M been pendant from X, then after the subtree of L had been searched for tours, the search would have gone up to X, down to M,and through all of M's subtree. In the presence of N, however, the subtrees of L and M are pruned completely, and when N's subtree has been exhaustively tried the level of X can be ignored, and an immediate rise to the level of X's predecessor made before moving down again. This process reduces to absurdity the tree-search on the 4 x 4 board (Figure 1.5), for after the initial move $1 \longrightarrow 7$, the choice of square 9, 14, or 16 is determined by $7 \longrightarrow 16$ being a GOTO move. The $16 \longrightarrow 10$ is forced, and then $10 \longrightarrow 1$ is also a GOTO move, so that if no dead ends are to be left, the knight must return to square 1 in only 4 moves: it is thus impossible to get a knight's tour on a 4 x 4 board. A more complex situation (but nevertheless one which does occur in practice) is when more than one of L, M, and N is doubly-connected, i.e. there are 2 or more GOTO moves from the square presently occupied; in this event (Figure 2.8) choice of either M or N leads to the other one being a 'dead-end', and the choice of L leaves both M and N attainable from only one vertex (P and Q respectively). There is no way of moving down the tree here whilst still retaining the possibility of a knight's tour, so we must backtrack to the level above X once more before resuming the downward search.

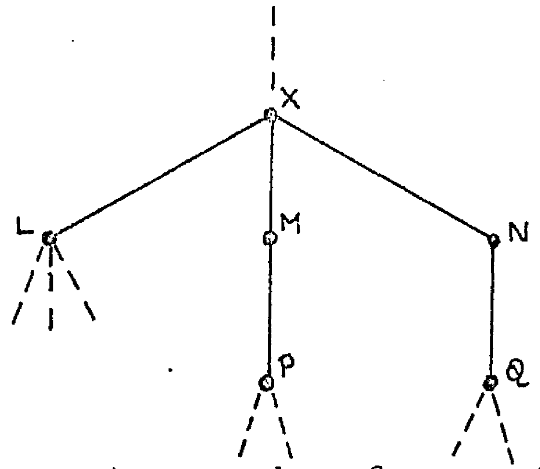


Figure 2.8

Double GOTO

GOTO moves can occur at any number of successive levels in the tree, and particularly in the deeper levels it is common to find 6 or 7 consecutive forced moves due to thinning of the graph by the edge-removal technique mentioned at the beginning of the subsection. This of course implies that backtracking up several levels without need to check downward branches will subsequently occur, and therefore a fast backtrack on GOTO moves should be incorporated in the program.

We close this subsection with a summary in table form of the action required at nodes in the tree with various numbers of GOTO moves, both when moving up and when moving down the search-tree.

No. of GOTO moves	Down	Up
0	Arbitrary choice of TEST move	Try new TEST move down; if all already tried, up one level
1	Forced choice of GOTO move; ignore TEST moves	Go straight up one level
>1	Up one level	-

Table 2.1

2.2.2 Short Loops The concept of the short loop as a criterion for pruning branches is due solely to the choice of the corner square as the knight's starting point. We have seen that corner square 1 is connected only to squares 9 and 14, the former being used at the start of the tour: the very simple short loop criterion states that square 14 must not be reached before move 35. Otherwise one of two unacceptable alternatives may occur:-

(a) if the move $14 \longrightarrow 1$ is made, the cycle from square 1 back to square 1 has been accomplished in <36 moves, so not every square on the board has been visited.

(b) if the move $14 \longrightarrow 1$ is not made, the knight cannot get back to square 1 subsequently without visiting square 14 a second time.

Both these violate the rules of the knight's tour.

Square 14 may appear either as a TEST move or as a GOTO move. In the former instance, application of the criterion is equivalent to loss of just one branch (and its consequent subtree), but in the latter both criteria can be combined: if N is square 14 in Figure 2.7, then choice of L or M cannot produce any tours, as before, but now neither can choosing N for the reasons just stated. Hence, the combination of a GOTO and a short loop move at a particular level of the search give sufficient reason to prune the branch at that level, and go one up.

2.2.3 Extensions of the Short Loop Principle Since we have shown the necessity of leaving square 14 unused until the 35th move, plainly we may extend this idea back and conclude that not all of squares 3, 10, 22, 25 and 27 may be used before the 34th move and even that not all of 2, 6, 7, 9, 11, 16, 18, 19, 21, 23, 26, 30, 31, 33, 35 can occur before the 33rd. The likelihood of this ever in fact happening must be regarded as extremely small, and in any case the time consumed in applying the test will certainly not balance up the time saved. The criterion grows weaker the further up the tree we try to apply it, as the number of squares all of which must not have been visited increases whilst the number of moves allotted is decreasing; it rapidly becomes a near certainty that not all the squares stipulated will have ~~to be~~ been visited.

A similar approach is to build the tour from both ends which, rather than testing whether all of a set of squares has been used before a certain stage, tests to see when one of that set remains and chooses that square as an end move. For example in conducting the ordinary tree-search we have:

0th square is 1 , $\therefore$ 36th square = 1

1st square is 9 , $\therefore$ 35th square = 14

After several moves, squares 3, 10, 25 and 27 have been used.

$\therefore$ 34th square = 22.

This suffers from the same weakness as the other, however, and cannot be considered a viable method. A much more potent 'two-way' construction will be described in a later section of this chapter.

2.3 <u>Dynamic Pruning Tree-Search</u> The elements of a string of squares in a tour may in certain cases be permuted (see 2.6). Again we take as axiomatic the statement 'if the string of squares A, starting from the root of the tree, is unproductive of knight's tours, then so will be any string P, where $P \epsilon p(A)$, $p(A)$ being the class of permutations of A, also starting at the root, which are sequences of l.t.m's.

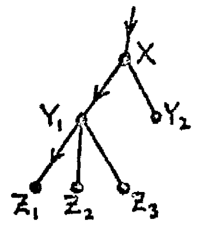
The procedure would consist of storing a table of unproductive strings and comparing the string of moves being generated with this table. If at any stage a match is made (in the sense that the elements of the unproductive string and the current string are the same, though differently ordered) then the current branch is abandoned. A new entry in the table is made every time the current branch ceases its downward trend; the elements, starting right at the root of the branch with square 1, are listed. The next step is to go up one level, then down one again, trying the next consequent move, (and entering that entire branch in the table) and so on until all the consequents at that level have been tried, whereupon searching for productive routes goes up to the next level, and the entries just made are merged, the last element being obliterated (Figure 2.9). The idea is to reduce the length of entries in the table as much as possible since

(a) the shorter a string, the more significant it is (see below);

(b) storage space is an important consideration particularly if the method is to be extended to an 8 x 8 board.

By (a) we mean that if, for example, a string of 19 moves proves to be unproductive, then, if some permutation of these moves later arises, we can abort the branch at this point (i.e. effectively we prune the subtree below the 20th node); how much bigger, therefore, the subtree we prune if a permutation of an unproductive 7-move string appears.

On the other hand, the number of permutations (which might be termed the usefulness) on a string increases with increasing length of string.



Downward progress from Z_1 impeded by failure to comply with nonrepetitivity criterion

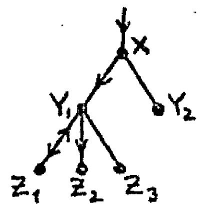
(a)

PREVIOUS ENTRIES

- - - - - - - -

1 9 . . . . . . X Y_1 Z_1 Table entry made

(b)

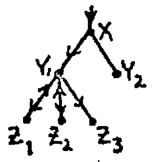


Z_2 tried - both it and subtree unproductive

(c)

1 9 . . . . . . X Y_1 Z_1 Table entry

1 9 . . . . . . X Y_1 Z_2



(d)

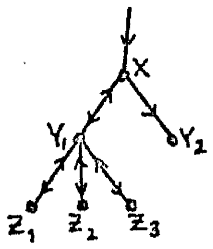
Similarly for Z_3.

State of table:-

- - - - - - - - - -

1 9 . . . . . . X Y_1 Z_1
1 9 . . . . . . X Y_1 Z_2
1 9 . . . . . . X Y_1 Z_3

(e)



No more possibilities at level Z, so up to level X, then try Y_2.

(f)

PREVIOUS ENTRIES

- - - - - - - - - -

1 9 . . . . X Y_1 ~~$\left.\begin{matrix}Z_1\\Z_2\\Z_3\end{matrix}\right\}$~~

PREVIOUS ENTRIES

- - - - - - - - -

1 9 . . . . X Y_1

1 9 . . . . X Y_2

Table housekeeping and new entry

(g)

Figure 2.9 Dynamic pruning tree-search table housekeeping

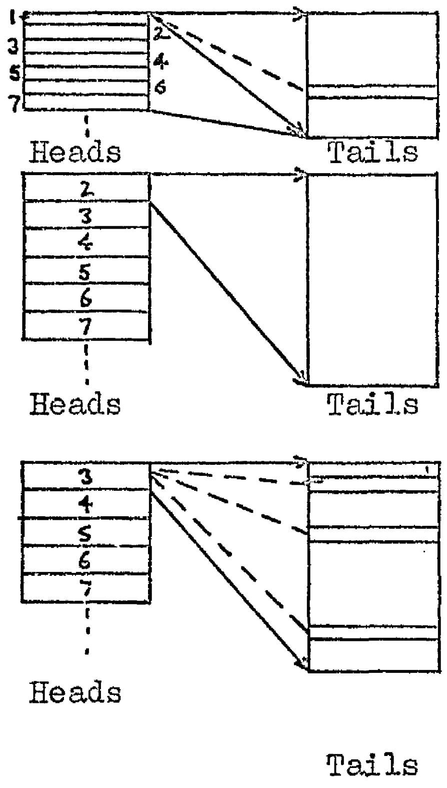
Merely from consideration of the heavy, continuous amount of house-keeping and the accompanying storage problem, however, this approach apparently has serious drawbacks; other criteria may provide more drastic means of pruning.

2.4 <u>Listing Half-tours and Matching</u> Let us next consider the problem of growing two trees, each 18 moves long and matching branches of these to form knight's tours. The required condition is as follows:-

Suppose the operations *, $'$ to retain their previous significance and let S_i, S_j be any strings of 18 l.t.m's starting 1->9. Then $S_i(S_j^*)'$ is a knight's tour if $S_i \cap S_j^* = \{1\}$ (considering S_i, S_j^* as sets of squares). By juxtaposing S_i and $(S_j^*)'$ we mean the sequence S_i followed by the sequence $(S_j^*)'$.

Now this approach, too, runs into complications because it must not be assumed that for every S_i there exists a unique S_j so that $\frac{1}{2}$-tours can be cancelled in pairs off a list which is continually decreasing in length. We frequently find in the exhaustive top-down tree-search that branches terminate at some depth greater than 18, yet short of a full tour; the first 18 moves of such a branch may have no corresponding tail-end to make a complete knight's tour. On the other hand, it is quite feasible that an S_j exists for a half-tour S_i , and that a permutation of the squares of S_j of the type mentioned in the last paragraph and described in more detail in 2.6 exists so that there may be a class of knight's tours $S_i\,(p(S_j^*))'$ all starting with the same half-tour, S_i.

The matching might be effected in a number of ways, the easiest conceptually being that of having a list of 'heads' and a list of 'tails', and each 'head' is compared with all the 'tails' for a match. After this process is complete, only the 'head' can be removed, since any 'tails' matched with it may also match another 'head'. An attempt is then made to match the next head and so on right down the list of 'heads' (see Figure 2.10).



(a) 1st 'Head' matches 1'tail'

(b) 1st 'Head' is removed, and 2nd matches no tails

(c) 2nd 'Head' removed, and 3rd matches 3 tails, etc.

Figure 2.10 Matching of Half-tours

The easiest way to compare two strings for common squares is to represent the squares present in each in a 36 [redacted]-bit boolean vector and perform the logical operation 'and' on them: The result can readily be checked to ensure that 1 is the only common square. But for output purposes this is insufficient, and we also require a binary representation (for compactness) of the actual ordering of the squares in the half-tour; for the 6 x 6 problem, 18 x 6 bits = 3 48-bit words is adequate. A complete specification of all 18-move strings from square 1 necessitates either storing the mirror-images or forming them at match-time. In the case of the boolean vectors, this is accomplished by using the logical operation $\neg$, while the binary representation of the move-ordering requires a symmetrical numbering of the board (Figure 2.11), and subtraction of the specification from a word of the form $(5353535353535353)_8$ with suitable modification to allow for diagonal elements. Thus each half-tour and its reflection needs 8 words to specify it completely.

Figure 2.11 Symmetric board-numbering

1	7	8	9	10	11
36	2	12	13	14	15
35	31	3	16	17	18
34	30	27	4	19	20
33	29	26	24	5	21
32	28	25	23	22	6

Whilst the tree-search for half-tours is significantly curtailed, the additional procedure of matching half-tours in quite involved, and the viability or otherwise of the method really depends on the number of half-tours produced and the consequent magnitude (both from the time and space points of view) of the matching.

2.5 Corner-to-Corner Strings We have already made use of the particular property of the corner squares that each has only one entrance and exit. Their usefulness in this respect suggests another way of dividing knight's tours to use the symmetry of the board to a greater extent: we construct the tour of 4 strings each of which joins one corner to another.

Now every tour passes through each corner once, and it is readily apparent that these corners must be traversed in one or other of the ways shown in Figure 2.12. The discussion of the symmetry of these types of tours will appear in Chapter 4, but at present we are merely interested in building up a tour from corner-to-corner strings, and this we see is possible either by combining 4 strings which go from one corner to an adjacent corner, or by using 2 strings of this type, and 2 which go from one corner to the diagonally opposite corner. If the corners are numbered $1 \longrightarrow 4$ clockwise from top left, the tour may be either of the forms

(i) $1 \to 2 \to 3 \to 4$
(ii) $1 \to 4 \to 3 \to 2$
(iii) $1 \to 3 \to 2 \to 4$
(iv) $1 \to 3 \to 4 \to 2$

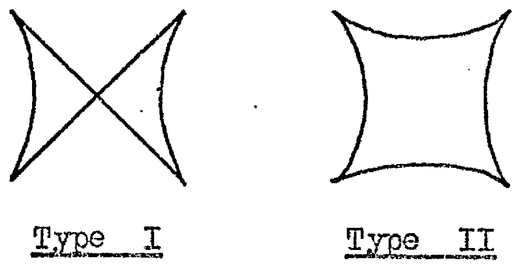


Figure 2.12 Use of corner-to-corner strings to build up knight's tours

We therefore need strings joining each corner to every other and these can be obtained from 1→2, 1→4, & 1→3 strings by the operations of reflection and rotation.

Typically, the operation of building up a knight's tour might consist of referencing a list of 1→2 strings and manipulating 4 of these until a set be found having no non-corner squares in common and also satisfying the condition that the sum of the lengths of the strings be 36 moves. Alternatively, the manipulation could be carried out earlier to generate 2→3, 3→1, etc. and these stored with the 1→2, 1→3, and 1→4 strings to be accessed when needed.

A hierarchy of vectors might be built up as in Figure 2.13 with the boolean vectors of the last section divided and sub-divided by initial corner and final corner, and by length, and with each boolean vector pointing to (several) orderings of the squares within it listed in binary representation vectors as in 2.4.

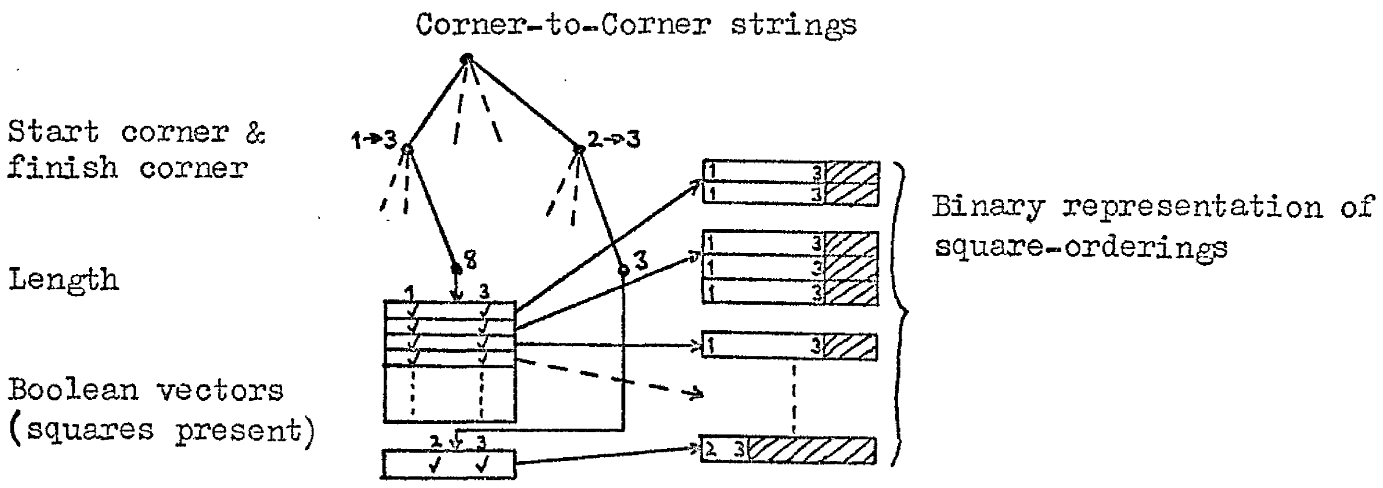

Corner-to-Corner strings
Start corner & finish corner
Length
Boolean vectors (squares present)
1→3
2→3
8
3
1
3
2
3
Binary representation of square-orderings

Figure 2.13 Hierarchy of vectors in Corner-to-Corner Matching Problem

When 4 suitable vectors have been found by testing to the boolean vector level of the tree, a number of tours may be generated by combining the various orderings of these vectors. The obvious difficulty with the method is the gigantic organisational problem of the matching. Unless the number of strings to be matched is relatively small, the additional variable introduced since 2.4 - length of string - threatens to make the whole scheme totally unmanageable.

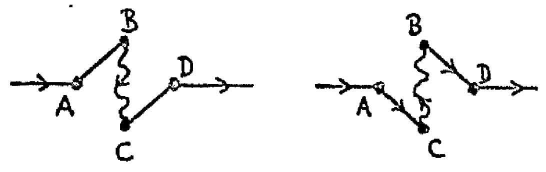
2.6 Transformations of Knight's Tours A knight's tour can be envisaged as just a string of numbers, each representing a square visited, and as such it might be subject to the operation of permutation of its elements. Clearly, if we generate all permutations of the numbers, we will generate all the knight's tours (and a great many 36-number strings which are not tours besides). The question arises as to whether we can devise some operation which, when applied to a given knight's tour, generates a new knight's tour, application to which gives a new knight's tour, and repeated application of which generates all the tours on the board.

2.6.1. Quasifactors and Reversals One operation which generates a new tour from a given one is that of reversing a string within a tour under special conditions which we will now specify.

If the squares A and B are separated by a knight's move, we write $A \mathcal{R} B$.

If a knight's tour contains 2 pairs of consecutive squares (the pairs not themselves consecutive) $\{A,B\}$ and $\{C,D\}$ and $A\mathcal{R}C$; $B\mathcal{R}D$, then we say that the tour has a quasifactor of degree 4, or a 4-quasifactor.

Our given tour is of the form . . . . AB . . . CD . . . . . , so that also $A\mathcal{R}B$ and $C\mathcal{R}D$. Thus we have the situation of Figure 2.14 (a) of a loop of moves between B and C which can be traversed in either direction according as the pairs $\{A,B\}$ and $\{C,D\}$ or $\{A,C\}$ and $\{B,D\}$ are connected up. The resulting new knight's tour is . . . . AC . . . . . BD . . . . . ; This is obtained by simply reversing the single string B. . . C, and so we refer to it as a first-order reversal. There is only one type of 1st order reversal, that already specified, and it occurs when the tour has a 4-quasifactor (the quasifactor itself is the graph showing the relationship $\mathcal{R}$ on the set $\{A, B, C, D\}$ - Figure 2.14 (b)).



(a)

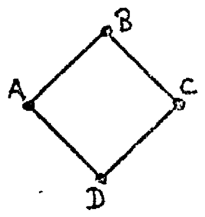


Figure 2.14
Quasifactor of degree 4 (b)

These concepts can be extended to more complex situations; consider the tour . . . AB . . . CD . . . . . EF . . . . . : if a tour has three pairs of consecutive moves (N.B. this implies $\mathcal{R}$ between the members of a pair) a, b, c, and there exist three relationships :-

$$\left.\begin{array}{l}\alpha' \mathcal{R} \beta \\ \beta' \mathcal{R} \gamma \\ \gamma' \mathcal{R} \alpha\end{array}\right\} \alpha,\alpha' \in a;\ \beta,\beta' \in b;\ \gamma,\gamma' \in c$$

then we say that the tour contains a quasifactor of degree 6 (6-quasifactor).
Figure 2.15 shows the relationship.

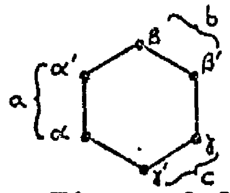


Figure 2.15
6-quasifactor

Let us return to the . . . AB . . . CD . . . EF . . . representation, and see what possible string-permutations arise from this development, remembering that not all permutations of B, C, D and E are feasible due to the linking of B and C and D and E by unspecified chains of moves. The permissible variations are :-

(a) . . . . . . AB . . . . . . CE . . . . . . DF . . . . . .

(b) . . . . . . AC . . . . . . BD . . . . . . EF . . . . . .

(c) . . . . . . AC . . . . . . BE . . . . . . DF . . . . . .

(d) . . . . . . AD . . . . . . EB . . . . . . CF . . . . . .

(e) . . . . . . AD . . . . . . EC . . . . . . BF . . . . . .

(f) . . . . . . AE . . . . . . DB . . . . . . CF . . . . . .

(g) . . . . . . AE . . . . . . DC . . . . . . BF . . . . . .

On studying the $\mathcal{R}$-relations these transformations require, (a), (b), and (g) are found to be degenerate cases where the tour has only a 4-quasifactor.

The relations for the other 4 cases are :-

(c) $A\mathcal{R}C$, $D\mathcal{R}F$, $B\mathcal{R}E$;

(d) $A\mathcal{R}D$, $C\mathcal{R}F$, $B\mathcal{R}E$;

(e) $A\mathcal{R}D$, $C\mathcal{R}E$, $B\mathcal{R}F$;

(f) $B\mathcal{R}D$, $C\mathcal{R}F$, $A\mathcal{R}E$;

We now consider these transformations in terms of reversals, first showing how (f) might be viewed (for clarity) as two consecutive reversals. The steps are :-

$$\begin{tikzcd}[column sep=small]
1.\ \cdots\cdots & AB \arrow[r, bend right, leftrightarrow] & \cdots\cdots\ CD & \cdots\cdots\ EF\ \cdots\cdots \\
2.\ \cdots\cdots & AC \arrow[rr, bend right, leftrightarrow] & \cdots\cdots\ BD & \cdots\cdots\ EF\ \cdots\cdots \\
3.\ \cdots\cdots & AE & \cdots\cdots\ DB & \cdots\cdots\ CF\ \cdots\cdots
\end{tikzcd}$$

Notice, however, that we do not know that $A\mathcal{R}C$ or $B\mathcal{R}D$, so that the string at stage 2 is <u>not</u> a knight's tour, although the final result is. It is therefore more in keeping with our original intention to consider the reversals as simultaneous. Transformations (c) and (e) are similar to (f), and if we denote by [ ] the result of reversing a string, (c), (e), and (f) are obtained from . . . . AB . . . . CD . . . . EF . . . . respectively by :-

(c) $\alpha[\beta][\gamma]\delta$
(e) $\alpha[\beta[\gamma]]\delta$
(f) $\alpha[[\beta]\gamma]\delta$

where α, β, γ, and δ represent respectively the strings . . . . A, B . . . . C, D . . . . E, and F . . . .

In each of these, the operation [ ] appears twice, so we call them second-order reversals.(d) is a third-order reversal :-

$$\alpha[[\beta][\gamma]]\delta$$

The juxtaposition of the symbols representing the strings is taken to mean the concatenation of the strings themselves. Now it is quite feasible to extend the scheme to higher degrees of quasifactors and orders of reversal, but it is dubious whether the inclusion of still more complex systems of R- relations would be of much value (the length of the tour obviously sets a limit on this) and it would rapidly become very costly in time.

More important is the deliberation of what scheme should be superimposed on these transformations and at this point we might envisage a purely heuristic approach. A fairly naive heuristic was, in fact, adopted just to give some idea of the speed and success at tour-finding of the basic method. Two strategems were inherent in the program :-

1. To take the 'least significant' transformation.

2. To take the simplest transformation.

By (1), we mean the reversal(s) nearest the end of the given tour was chosen, the idea being to restrict changes, as far as possible to the lower branches of the search-tree and so find knight's tours on neighbouring branches. We can think of this transformation method as a way of missing out completely all the abortive branches in the search-tree; our first ploy ensures a 'vernier control' on tour selection (see Figure 2.16), tending to transform tour α to tour β rather than tour γ .

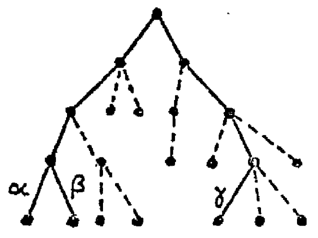


Figure 2.16
Reversal Strategy

The second strategem means that if the given tour has both a 4 - and a 6-quasifactor, then the former will be chosen if significance of the strings indicates no preference.

The heuristic was limited to second-order reversals and carried a first-in first-out store of the four most recent tours to insure against short-term recurrences. Proneness to long-term cyclical transformations is a grave difficulty of the method as used. Conversely, it is a fundamental advantage of the method that the tour-search is easily segmented, i.e. it can be stopped and restarted with only a minimal amount of temporary storage between runs ('freezing'). Only the last knight's tour found need be retained, whereas tree-searching methods demand some way of referencing the part of the tree so far not searched and this can involve a large area of core being frozen and later 'revived' (q.v. sub).

We give below the flow charts for the entire transformation procedure, for first-order reversals, and for second-order reversals. (Figures 2.17, 2.18, and 2.19). A FORTRAN version of this method and an ALGOL tree-search (2.2) were approximately of the same speed, but the heuristic program entered a cyclic transformation group of 3 tours after finding 17 tours, and self-aborted.

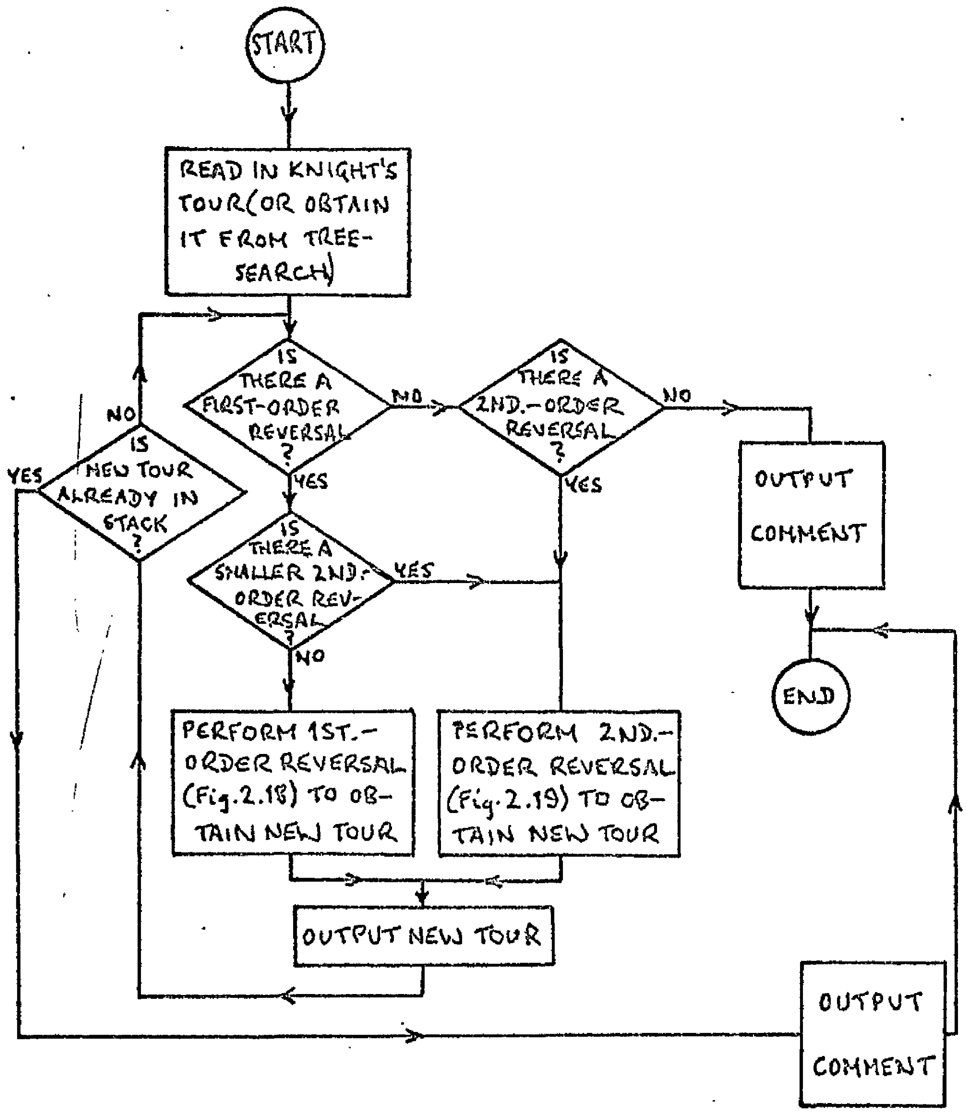
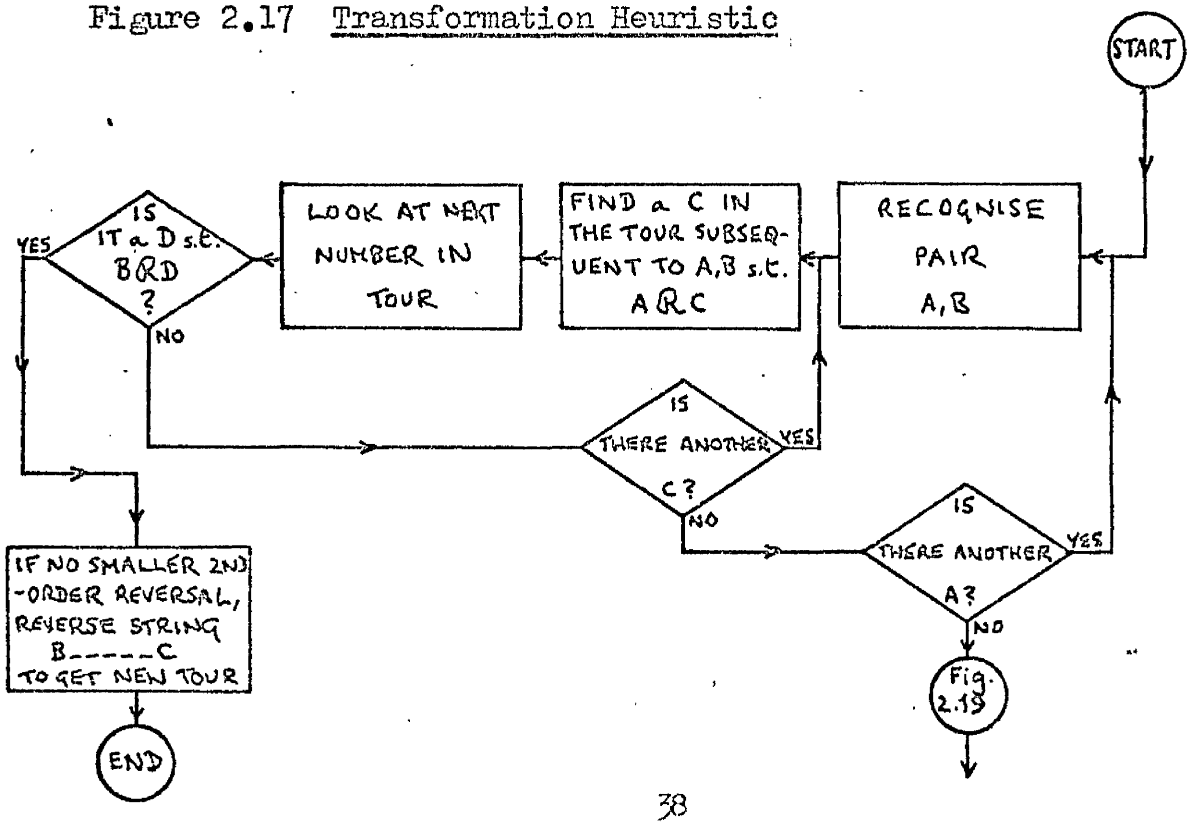

START
READ IN KNIGHT'S TOUR (OR OBTAIN IT FROM TREE-SEARCH)
IS THERE A FIRST-ORDER REVERSAL ?
NO
IS THERE A 2ND.-ORDER REVERSAL ?
NO
OUTPUT COMMENT
END
YES
IS THERE A SMALLER 2ND-ORDER REVERSAL ?
YES
NO
PERFORM 1ST.-ORDER REVERSAL (Fig.2.18) TO OBTAIN NEW TOUR
PERFORM 2ND.-ORDER REVERSAL (Fig.2.19) TO OBTAIN NEW TOUR
OUTPUT NEW TOUR
IS NEW TOUR ALREADY IN STACK ?
YES
NO
OUTPUT COMMENT

Figure 2.18 (below)
1st-order Reversal on tour . . . AB . . CD . . .

Figure 2.17 Transformation Heuristic


START
RECOGNISE PAIR A,B
FIND a C IN THE TOUR SUBSEQUENT TO A,B s.t. A R C
LOOK AT NEXT NUMBER IN TOUR
IS IT a D s.t. BRD ?
YES
NO
IS THERE ANOTHER C ?
YES
NO
IS THERE ANOTHER A ?
YES
NO
Fig. 2.19
IF NO SMALLER 2ND-ORDER REVERSAL, REVERSE STRING B-----C TO GET NEW TOUR
END

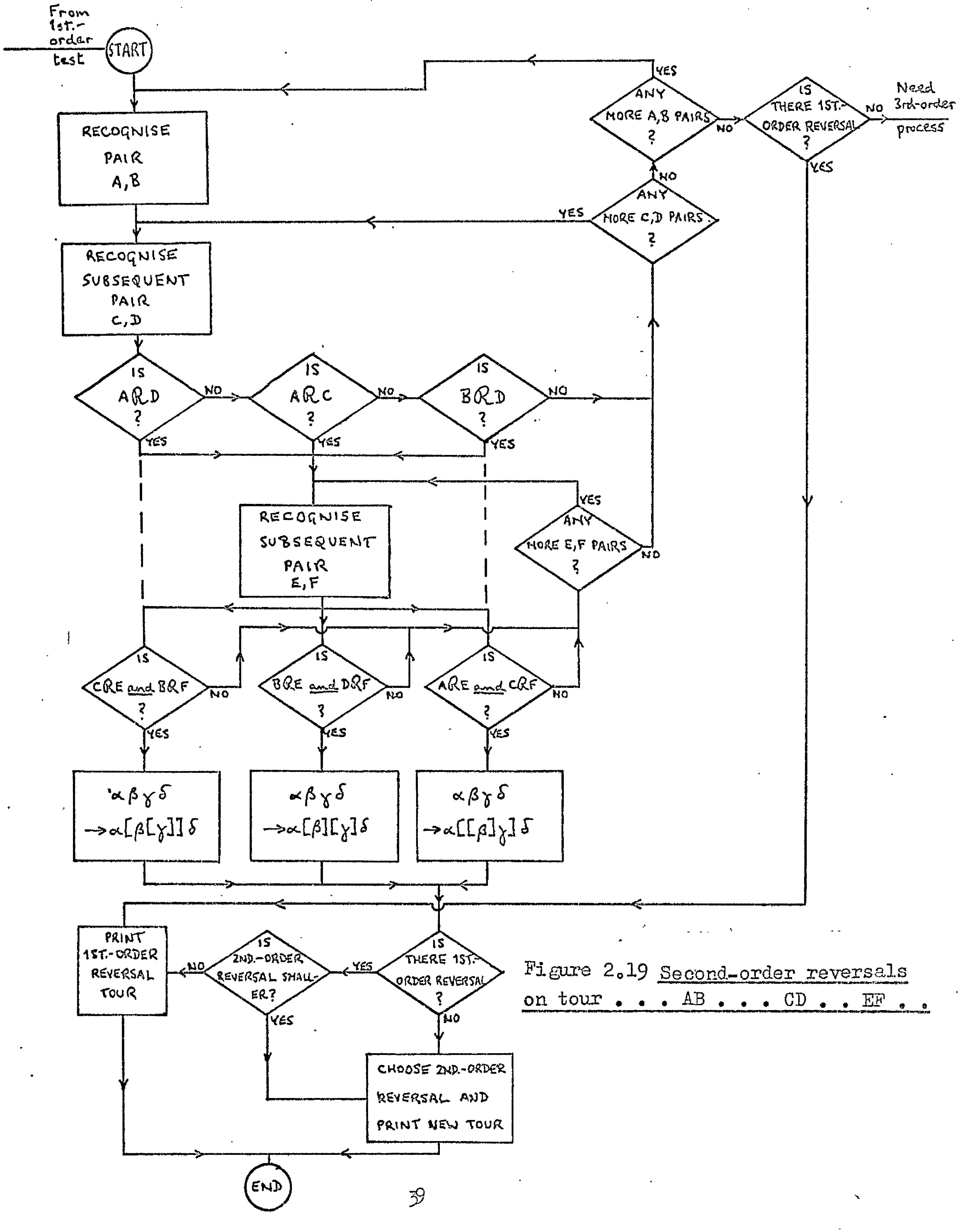


Figure 2.19 Second-order reversals on tour . . . AB . . . CD . . EF . .

2.7 Edge-Removal This is, in a sense, an amalgam of the methods already described in 2.2 and 2.4, in that it includes a tree-search which can absorb all the optimising features of 2.2 and also constructs its solutions both forwards and backwards, but it is essentially different in that it is a factor-finding method.

While the methods of tree-search specifically designed to find knight's tours check for the previous occurrence of each chosen square, the edge-removal method is solely concerned with reducing the degree of each vertex of the equivalent graph to 2, which is the condition for a factor, though not necessarily a Hamiltonian cycle; the graph in Figure 2.20, for instance has a factor, but no Hamiltonian cycle (We use the term factor rather than semi-factor since although we have defined the tour to be un-orientated, it will be recalled that in practice, use was made of the symmetry of the graph, and the end-result is directed, even if only inferentially).

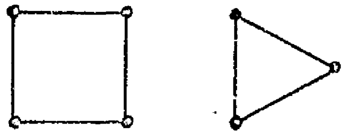


Figure 2.20 Graph with factor

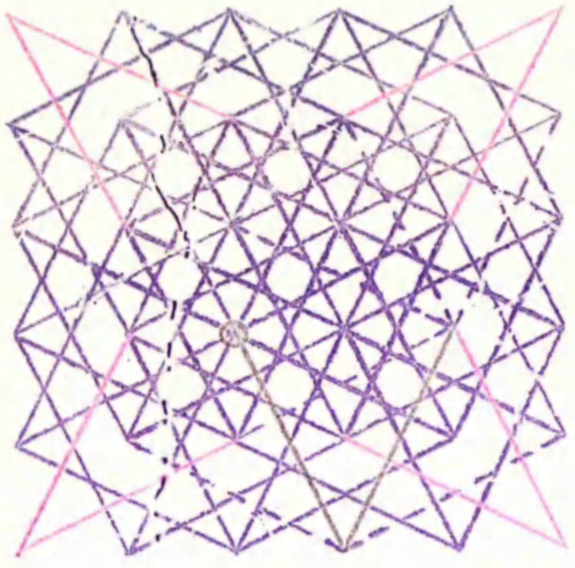
The method consists of choosing squares in much the same way as in 2.2, but a record is kept of all edges which must be traversed in one direction or the other, and chains of these are built up as succeeding choices of squares are made. Once a square has been visited, all edges incident at that square's vertex on the graph, other than the entrance and exit edges, can be erased.

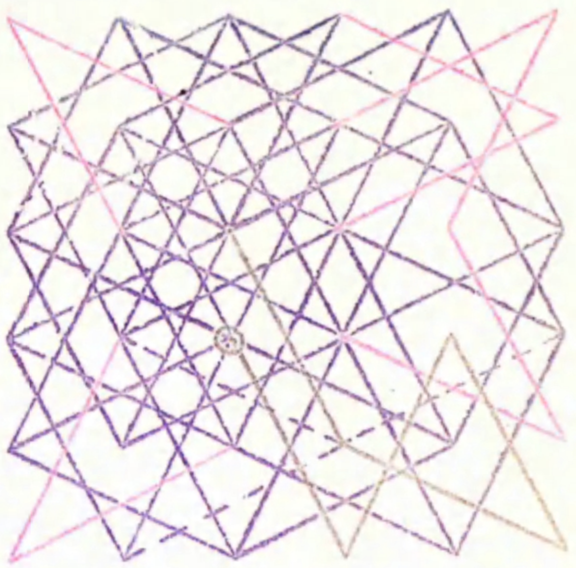
This may render some other vertex doubly-connected, and if this vertex is only one edge removed from the end of a chain, its entrance and exit edges (as they now must be) can be annexed to the chain. Further possibilities are the joining of two or more consecutive chains, and chains building up backwards from the starting square, this being somewhat similar to the process of 2.4. It is found that after some 6 - 8 moves have been chosen, the act of removing edges has built up chains of forced moves to the extent that all moves are in fact constrained; at this juncture, either

(a) a factor (or knight's tour, possibly) has been found

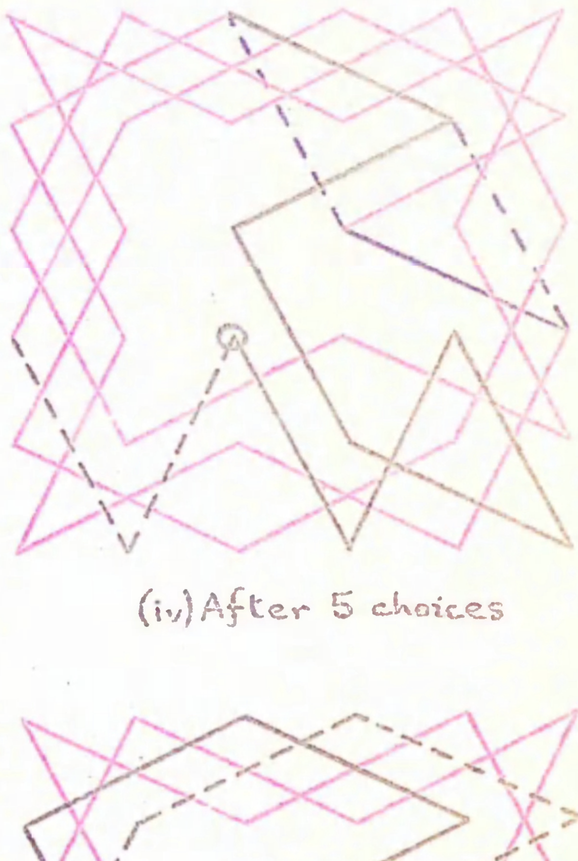
or (b) we have the equivalent of a double GOTO (3 chains joined at a point) or failure to observe a GOTO (a dead-end).

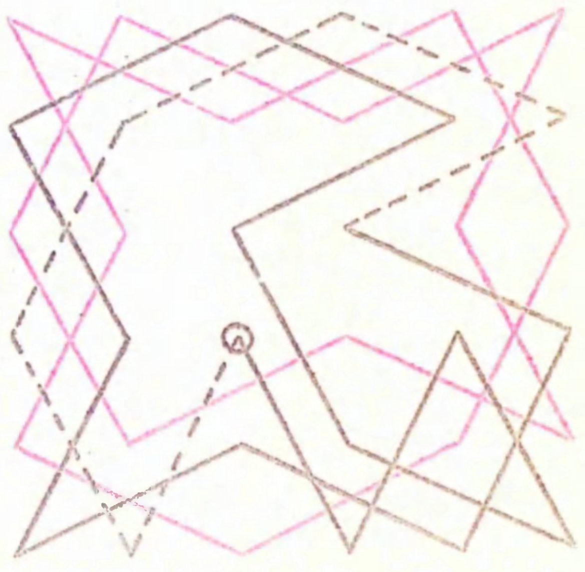
An example of the procedure is shown in Figure 2.21, this particular branch of the search-tree requiring only 5 deliberate choices before the 3 disjoint cycles are found. The starting square is ringed. In each diagram only the deletions of immediate consequence from the move just chosen are indicated (by dotted lines); the next diagram also omits any second, third- etc. order deletions, and includes chains (crossed lines) used in the path taken (solid lines).


(i) After 2 choices


(ii) After 3 choices

(iii) After 4 choices


(iv) After 5 choices


(v) A solution with 3 disjoint factors

—— Optional unused l.k.m's
—— Forced unused l.k.m's
- - - 1st.-order edge-removals for last move made
—— Forward soln.
- - - Backward soln.
Starting square ringed

Figure 2.21

It will be seen that after the choice of 4 moves, a 2-element chain of forced moves already extends back from the starting square; after 5 moves have been picked this has risen to 6 moves in length. The existence of disjoint cycles becomes apparent at this level, however, so the move denoted by the last solid line in (d) must be rejected, and an alternative tried.

This example shows that, typically, only a very curtailed tree-search is necessary with the edge-removal method, but it also demonstrates (chiefly by the large number of deletions between (b) and (c)) the very complicated housekeeping necessary for a method where one move can have such a profound effect on the state of the graph.

2.8 Fortet's Boolean Algebraic Method This method is described, with an example, in Berge, so we will not describe it again here. It is pertinent to remark, however, that the size of the application in which we are interested is a good deal larger than that of the example quoted, and it is found that not all the connection veriables, X_j^i, can be expressed in terms of one independent auxiliary variable (W_j^i, Z_j^i, or Y_j^i for 4x4 problem); in fact, for the 4x4 board 8 independent variables must be specified, and according to the different choices for the values of these variables, we obtain the 256 factors on the 4x4 board's equivalent graph. These are:-

1x4membered circuit	6x2-membered circuits	64
2x4 " " circuits	4x2 " " " "	96
3x4 " "	2x2 " " " "	64
4x4 " "		16
	8x2 " " " "	16
		256

As there are no knight's tours amongst these, it seems plain that the number of factors to be sorted in the 6x6 case will be very large. The main difficulty in implementing Fortet's method seems to be the choice of value of the variable which will most simplify the system of equations, an operation which the human eye finds none too difficult from long experience, but which demands 'machine intelligence' nevertheless, notoriously a tricky and often inefficient thing to program.

2.9 Dynamic Programming This method has been successfully applied by Dantzig et al, Bellman, and others to Hamiltonian circuit problems of the travelling-salesman type, where, however, there is a weighting of the edges according to the distance along the edge (the problem being, briefly to find the Hamiltonian path or circuit of shortest length passing through a specified set of cities). In the problem considered here we are hampered by a system where with few exceptions (such as GOTO moves) all the edges are equally weighted;

it is conceivable that a probabilistic weighting system could be formulated which would find all the knight's tours, but it has rather been the aim in this work to confine ourselves to deterministic methods. The lack of a deterministic return function which could be evaluated after the choice of each square leads us to reject dynamic programming as a method of approach.

2.9.1 <u>The Warnsdorff Algorithm</u> This algorithm was devised as long ago as 1823; it does not pretend to infallibility, but is a useful method of finding a Hamiltonian path (N.B. not circuit) on a graph, and may be stated as follows:-

The l.t.m. (next to be made) is selected which connects with the smallest number of subsequent l.t.m's, provided this number is non-zero. If two l.t.m's connect with an equal number of further moves, the tie may be broken arbitrarily.

Pohl (q.v.) describes a method for breaking ties which has as its basis the maximisation of the number of connections towards the end of the tour. These algorithms in effect weight the alternative moves, or branches of the search-tree, but in a probabilistic sense as mentioned in the last paragraph. The maximisation of connections at the stage immediately following (by minimising edge-removals at the present level) is not a sine qua non of knight's tours, and while Pohl's generalised algorithm may be quite adequate to find a Hamiltonian path on a graph, many tours do, in fact, contravene the rule, and this could not therefore be deemed a satisfactory 'optimal policy' of a dynamic programming technique.

CHAPTER 3

The Exhaustive Tree-Search with One-Level Lookahead

3.1 Development All the early programs in the development of the method were written in KDF9 ALGOL. The comparison of a working 6 x 6 ALGOL version with the rates of tour-finding mentioned in Duby led to the rethinking of program structure and also conversion to FORTRAN within the Egdon system, with considerable portions of the program (the tree-search particularly) being converted to UCA3 assembly language; even this did not remove entirely the considerable discrepancy between FORTRAN/UCA3 on the KDF9 and FAP on the 7094.

The very earliest attempt was to solve the 8 x 8 problem running an ALGOL program through the Whetstone (fast translation, inefficient object code) compiler. This produced copious diagnostic output designed to chart the progress of the tree-search, including the chosen square, the tree-level (invaluable when backtracking several levels), and the state of certain variables. The chief achievement of this program was that it led to a working tree-search: it also exposed the folly of attempting to solve the 8 x 8 problem with such puny weapons! That speed was of the essence was plain: all subsequent ALGOL programs were compiled on the (slow translation, efficient object code) Kidsgrove compiler using the POST magnetic-tape based operating system: a 4 x 4 tree-search of this type took 2 mins. 50 secs. to conclude (correctly) that no knight's tours existed.

At this stage, the one-level lookahead was introduced, and immediately showed its worth, taking zero time for the 4 x 4 board (shown to be trivial in 2.2.1), and 3 seconds in the abortive search on a 5 x 5 (see 1.7). Extension to a 6 x 6 board demonstrated that the lookahead saved several hours of machine-time merely by spotting a GOTO move at level 8! The program found 112 knight's tours in a 60 minute run, and an estimate of 50 hours was put on the complete search.

What has been described so far might be termed 'preliminary skirmishing': at this point the method was established, and later development was at the programming, rather than the theoretical level, i.e. at the third level of the tree in Figure 2.1. There are, basically, three types of programs, which we call first-, second-, and third-generation. Their characteristics are laid out in Table 3.1.

Generation	Tree-Search	Driven by	Written In
1	Recursive	Static 2-D condensed connection matrix & dynamic stack of degree vectors	ALGOL & FORTAN/UCA3
2	Iterative	As 1	ALGOL only
3	Iterative	Dynamic 3-D stack of boolean connection matrices	ALGOL & FORTRAN/UCA3

Table 3.1

The programs themselves will be found in Appendix 1. Our purpose in this chapter is to flowchart the main features of each program, describe fully their characteristics and show the advantages of each program over the preceding generation.

3.2 First-Generation Programs The initial factor which influenced the decision to write the tree-search in recursive form was its repetitive nature; the basic structure which is maintained throughout the search is remarkably simple and is shown in Figure 3.1. The stack of vectors, 'new-deg' contains at each level in the tree the degrees of all the squares. These two features will now be discussed more fully.

3.2.1 Recursive Structure The diagram in Figure 3.1 shows only the very simplest interpretation of the algorithm for choosing a square. In practice, the housekeeping subsequent to the choice was not done in two separate pieces but in one integrated procedure, 'stepcount'; this procedure actually incremented the tree-level counter, 'reccount', when normal downward progress was being made in the tree. The 'back 1 move' directive was maintained within the move-choosing procedure, 'nextstep', and the two procedures called one another recursively. The following points about backtracking should perhaps be clarified; the notation NX_n, ST_{n+1}, etc. mean ''next-step' at level n', ''stepcount' at level n+1' and so on; '<-' means 'returns control to','->' means 'calls'.

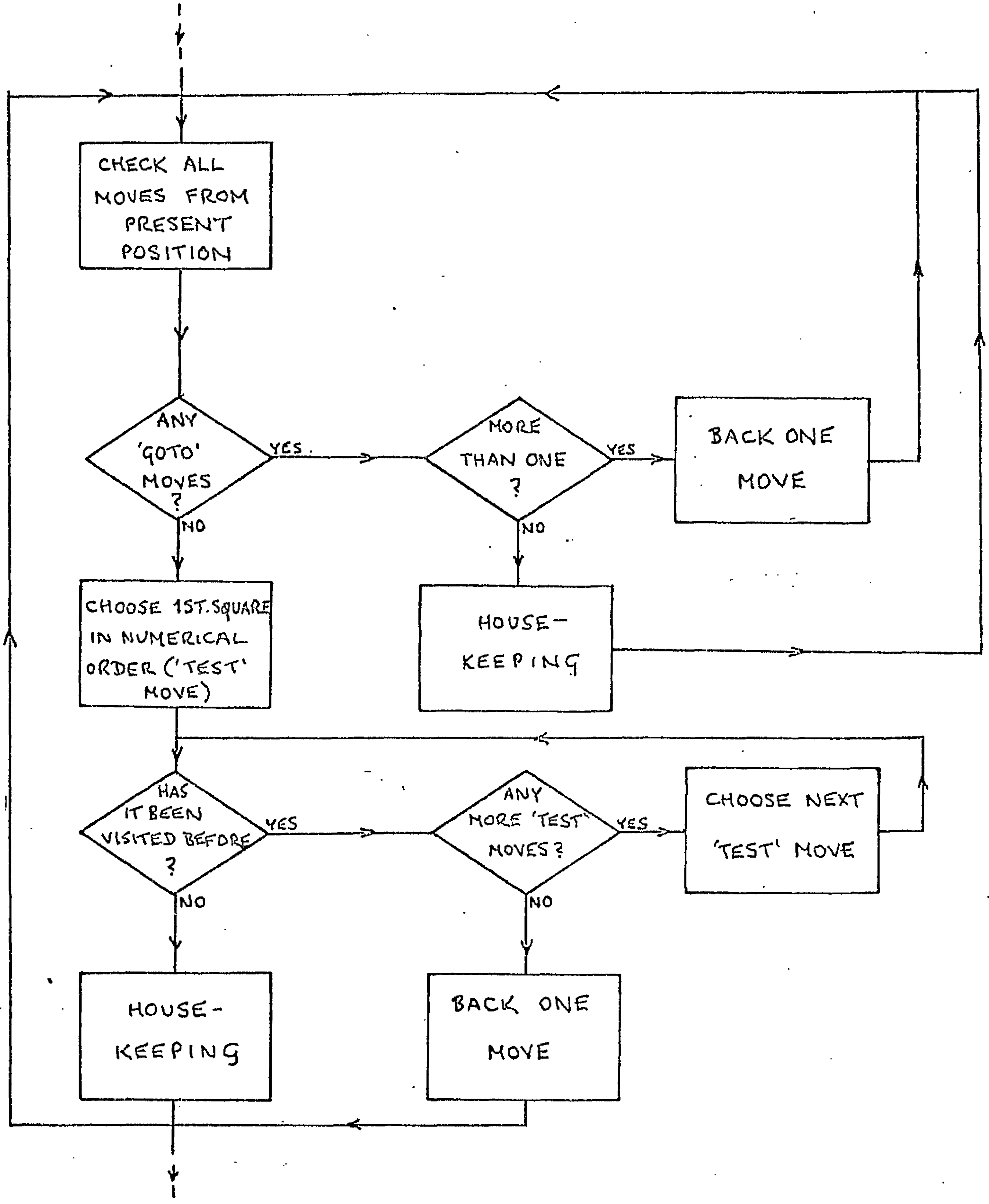


Figure 3.1

Basic Tree-Searching Loop

(a) After the downward sequence

$$\ldots \; NX_{n-1} \longrightarrow ST_{n-1} \longrightarrow NX_n \xrightarrow{GOTO} ST_n \longrightarrow NX_{n+1} \qquad - \text{①}$$

if there is no l.t.m. at level n+1, the backtrack sequence is

$$NX_{n+1} \longleftarrow ST_n \longleftarrow NX_n \longleftarrow ST_{n-1} \longleftarrow NX_{n-1} \qquad - \text{②}$$

and another square is chosen, the progression (1) being reasserted.

(b) After the downward sequence

$$\ldots \; NX_{n-1} \longrightarrow ST_{n-1} \longrightarrow NX_n \xrightarrow{TEST} ST_n \longrightarrow NX_{n+1} \qquad - \text{③}$$

is halted, the backtrack may follow the form

$$NX_{n+1} \longleftarrow ST_n \longleftarrow NX_n \ldots \qquad - \text{④}$$

and another TEST move tried, or if no TEST moves remain, a sequence analogous to (2) is pursued. These backtracks are shown dotted in Figure 3.2, and are in accordance with 2.2.1.

(c) The sequence when a double GOTO is encountered is

$$\ldots \; NX_{n-1} \longrightarrow ST_{n-1} \longrightarrow NX_n \xleftarrow{D.G.} ST_{n-1} \longleftarrow NX_{n-1} \ldots \qquad - \text{⑤}$$

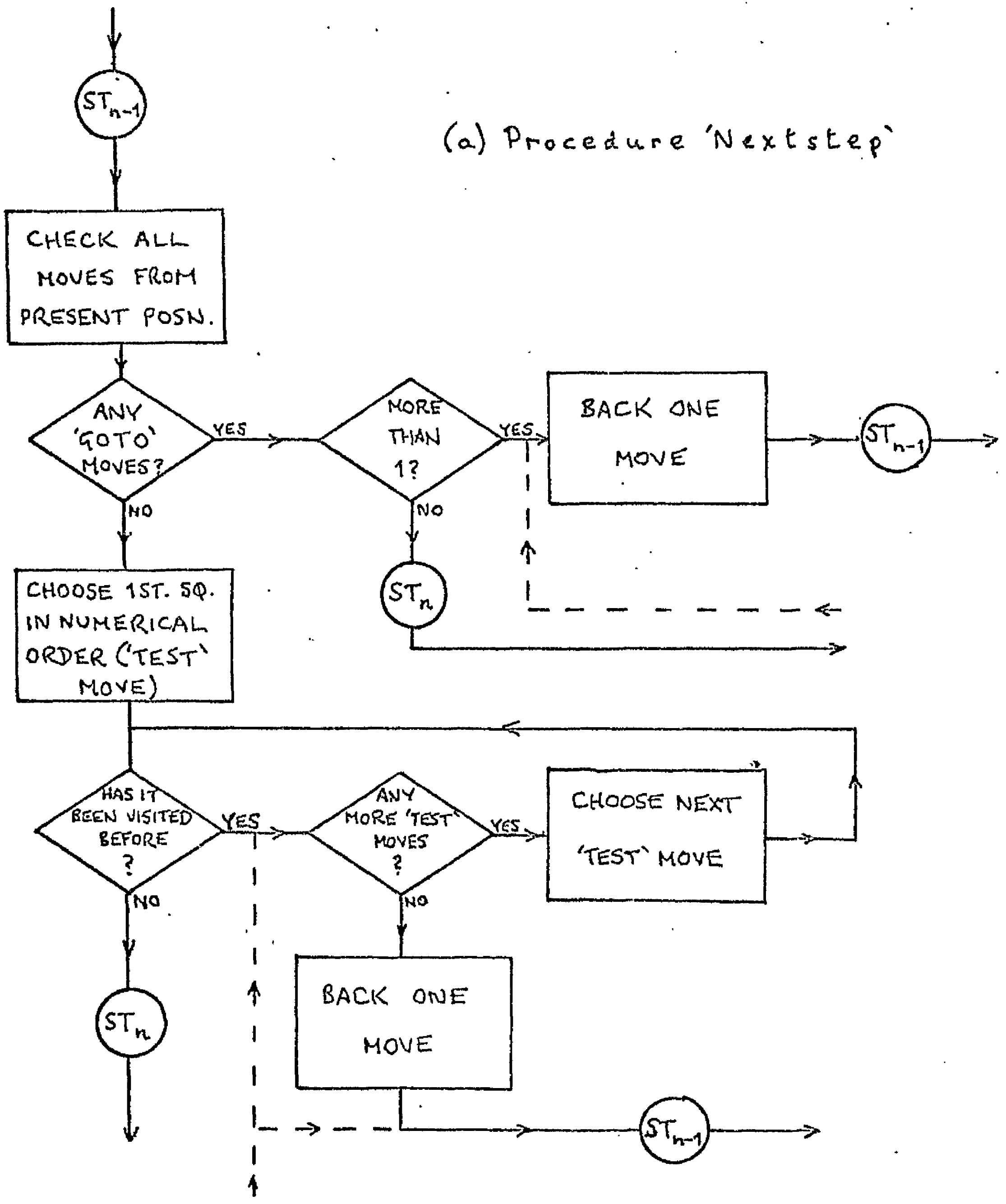
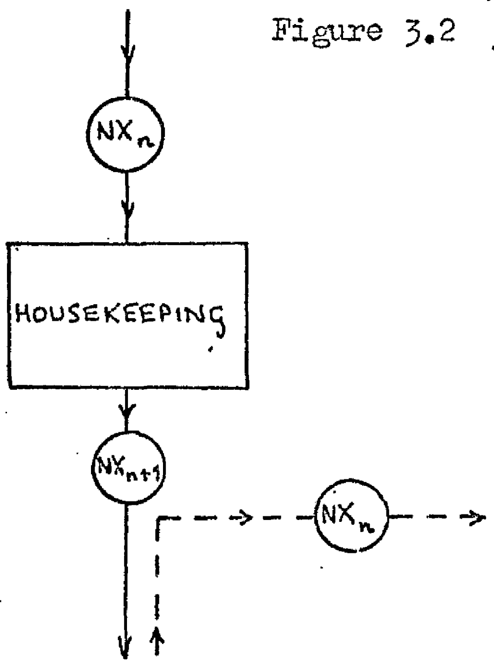


Figure 3.2 Interrelationship of 'Nextstep' and 'Stepcount'



3.2.2 <u>The Degree Vectors</u> These appear in the program as 'origdeg' ('ORIGinal DEGree') and 'newdeg', the former being merely a list of degrees of the various vertices in the equivalent graph before any moves have been made. Its elements have the values given by comparing Figures 1.5 and 2.11 (the symmetric board-numbering was used in the early stages - see 3.2.6), these being read in at run-time. The array 'newdeg' was designed to contain the degree of each vertex at each level in the tree, and was based on a kind of dynamic stack principle; in practice, the method of upkeep was as follows. Consider the sequence of moves:

$$\begin{array}{llll} \text{square} & A \xrightarrow{TEST} & B \xrightarrow{GOTO} & C \\ \text{level} & n-1 & n & n+1 \end{array}$$

The depth of recursion counter, 'reccount', is used as a pointer to index 'newdeg', as each move is made a new set of degrees, corresponding to the edge-removal idea of 2.2.1, is calculated and entered in the area of the array specified by the pointer (Figure 3.3; ND_n means 'new degrees for level n').

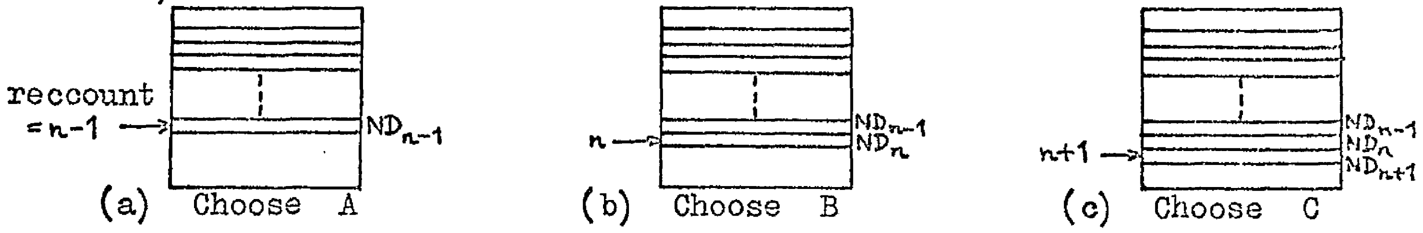


Figure 3.3 <u>Updating 'Newdeg' in Down-tree Search</u>

If C is a dead-end, the considerations of 3.2.1 show that we climb two levels in the tree before starting a downward seek again, viz,

$$\begin{array}{lllllll} \text{square} & A & \xrightarrow{TEST} D & \xrightarrow{TEST} E & \xrightarrow{TEST} F & \ldots \\ \text{level} & n-1 & n & n+1 & n+2 & \end{array}$$

The now unwanted versions of ND_n and ND_{n+1} corresponding to the fruitless choices of B and C are not erased: they are simply overwritten by the new ND_n and ND_{n+1} when D, E, etc. are picked. When backtracking, the 'reccount' pointer has gone back to the level of ND_{n-1} which is the last version of 'newdeg' it is still desired to retain; on resuming the downward search, overwriting occurs automatically, with no call for erasure of the unwanted vectors. Extending our notation slightly, we represent by $ND_n(X)$ the vector 'newdeg' at level n corresponding to the choice of square X at that level. Then the procedure just described will be seen to be equivalent to a true pushdown store, whose contents in the example cited are as in Figure 3.4.

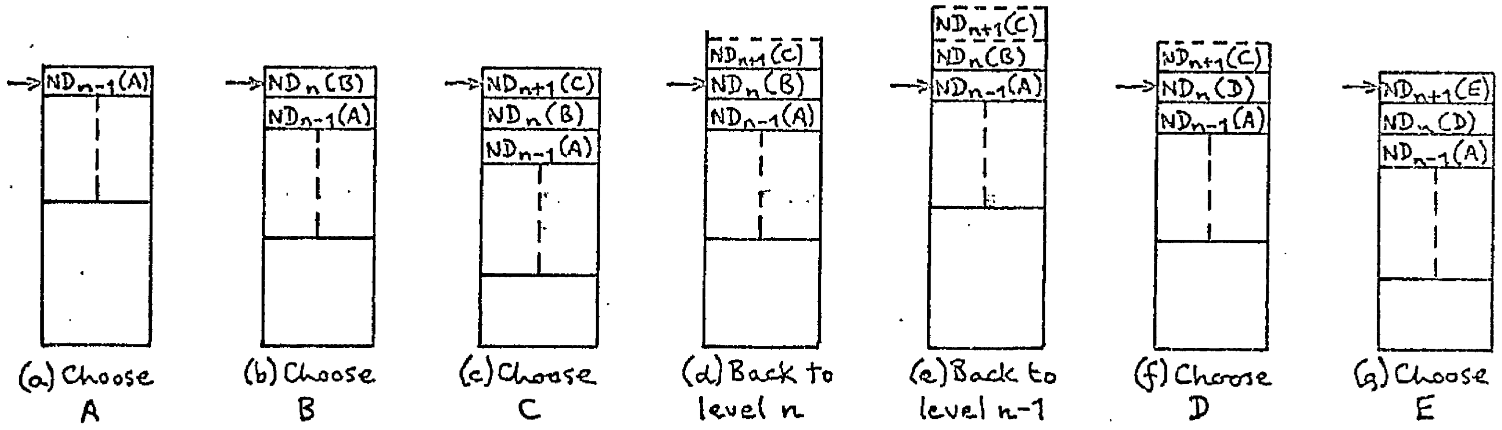


Figure 3.4 Pushdown Equivalent of 'Newdeg' Stack

3.2.3 <u>The Condensed Connection Matrix</u> This is the array 'XX' in the program. To define it, we use the observation of 1.2 that no square has a degree greater than 8 (irrespective of the board-size).

Definition The condensed connection matrix XX, associated with the equivalent graph of the knight's moves on a 2n x 2n chessboard, is a $4n^2 \times 8$ matrix whose elements

$(xx)_{ij} = k$ if i, & k are connected vertices of the graph

subject to $0 < i \leq 4n^2$, $1 \leq j \leq 8$, and $(xx)_{ij} < (xx)_{i,j+1}$ excepting that if i is connected to p (< 8) vertices, then

$$(xx)_{ij} = 0, \quad (p+1 \leq j \leq 8)$$

The form of this matrix for the 6 x 6 board numbered according to Figure 2.11 is shown in Figure 3.5.

This is the most compact way of storing the original state of the graph, without having recourse to assembly language, and was read in at run-time, thus (with 'origdeg') completing the data for first and second generation programs in ALGOL. 'Origdeg' was in fact used as an indexing vector when reading in 'XX' (see program).

It should be stressed that 'XX' contains a complete specification of every l.k.m. on the 6 x 6 board, and both first and second- generations largely relied on 'XX' for the choice of an l.t.m., referring each time to 'newdeg' to check if any of these were a GOTO move. It is this sense that we say in Table 3.1 that these programs are driven by 'XX' and 'newdeg'.

3.2.4 Exclusion of Short Loops This pruning aid, described in 2.2.2, was incorporated into procedure 'nextstep'. The two separate exits to 'stepcount' necessitated two tests for short loops which were located as shown in Figure 3.6. In the GOTO case, the order of testing for double GOTO and short loop is immaterial; it will be noticed here that the occurrence of a short loop and GOTO move together does indeed cause the search to go up one level as remarked in 2.2.2.

3.2.5 The Solution Vector Called in first-generation programs 'storepath', this 36-element vector is a pushdown list in the strict sense which contains the l.t.m's chosen at any stage in the tree-search. There is no overwriting of abortive l.t.m's by newly-chosen ones: as soon as a move has been fully investigated it is replaced with a zero and the pointer ('reccount') set one back; as a simple example, we show the change of state of 'storepath' during the tree-search shown in Figure 2.9.

9	16	27	34	0	0	0	0
7	9	14	19	24	29	34	36
12	14	18	21	22	25	29	31
16	18	25	27	0	0	0	0
19	24	0	0	0	0	0	0
3	13	35	0	0	0	0	0
14	16	31	36	0	0	0	0
2	3	15	17	0	0	0	0
12	16	18	0	0	0	0	0
13	17	0	0	0	0	0	0
1	4	10	17	30	35	0	0
7	11	18	19	27	31	0	0
3	4	8	20	0	0	0	0
9	16	19	0	0	0	0	0
2	5	8	10	15	20	26	30
9	11	12	21	24	27	0	0
4	5	10	13	0	0	0	0
3	6	13	15	23	26	0	0
14	16	22	24	0	0	0	0
4	17	23	0	0	0	0	0
4	20	26	0	0	0	0	0
19	21	27	29	0	0	0	0
3	6	17	20	28	30	0	0
4	5	30	33	0	0	0	0
16	19	22	31	32	34	0	0
2	5	13	17	23	28	33	35
24	27	34	0	0	0	0	0
3	4	23	35	0	0	0	0
12	16	24	25	32	36	0	0
1	4	8	13	26	33	0	0
26	30	0	0	0	0	0	0
25	27	31	0	0	0	0	0
2	3	26	28	0	0	0	0
7	12	27	29	0	0	0	0
3	8	30	0	0	0	0	0

Figure 3.5 Condensed connection matrix for symmetrically numbered 6 x 6 board

1 9 . . . . X Y_1 0 0

1 9 . . . . X Y_1 Z_1 0 . . . 0

1 9 . . . . X Y_1 0 0

1 9 . . . . X Y_1 Z_2 0 . . . 0

1 9 . . . . X Y_1 0 0

1 9 . . . . X Y_1 Z_3 0 . . . 0

1 9 . . . . X Y_1 0 0

1 9 . . . . X 0 0

1 9 . . . . X Y_2 0 0

The reason this method was adopted in the early programs was largely for clarity of discrimination, but since the'reccount' pointer is always kept on the last acceptable move (Y_1 , Z_1 , Y_1 , Z_2 , Y_1 , Z_3 , Y_1, X, Y_2 successively above), these valid moves can be indexed with another pointer whose upper bound is set at the present value of 'reocount', and there is no risk of accessing abortive moves. This technique is used when seeking a TEST move, as all the (significant) entries in 'storepath' must then be scanned to check that the square under consideration has not been visited before (in the program, the second pointer, 'k' is incremented by 1 from 0 to 'reccount').

When 'reccount' reaches 35 and the corresponding entry in 'storepath' has been made, the knight's tour is complete and can be printed out in order of squares used simply by the output of the solution vector. The housekeeping for 'storepath'is carried out by procedure 'stepcount', and the last action of this subroutine is to test the level counter 'reccount' to see whether transfer of control should be made once more to 'nextstep', or whether 35 moves (the 36th being a formality) have been and it is required to output the solution. Adding this information to Figure 3.2(b) gives us Figure 3.7; PR stands for print routine.

3.2.6 <u>Output Routine</u> The knight's tours are printed out by the process and in the form described briefly above in the procedure 'printroute'. In the earlier versions of the program, symmetry about the board's main diagonal was taken into account by using the symmetric board-numbering of Figure 2.11 and a simple arithmetic rule, viz.

```
if square no. ≤ 6  print it in both unreflected and reflected tours
"     "     "  > 6  print it in the unreflected tour,
                    print (43-it) in the reflected tour.
```

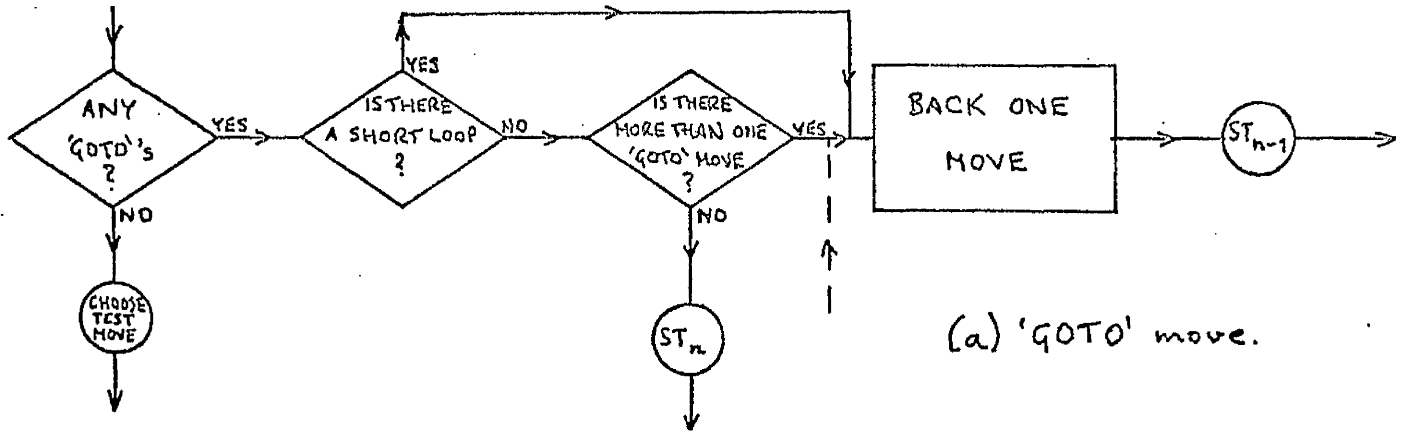
This gave 2 tours each time the solution vector was filled, but later, due to the disorder of the reflected tours (see 2.2) only the direct one was printed.

One other inclusion in 'printroute' was the call of procedure 'time', a short code procedure designed to hand back the run-time to 'printroute', and this could then be printed with each tour. This feature was found particularly useful when undertaking dissected step-timing, to be described later.

Finally, the identifier 'ccount' was used for the running total of knight's tours, this being updated by one each time 'printroute' was entered; at the completion of the tree-search, therefore, this counter could be printed out as the total number of Hamiltonian circuits on the graph of the 6 x 6 board.

3.2.7 Updating the Degree Vectors This is the chief housekeeping task of procedure 'stepcount' (the others being 'reccount' and 'storepath' updates and checking for solution), and we now give a little more detail of its implementation.

The first important note we make is that this activity is strictly confined to downward 'seeks' in the tree, according to the organization of 'newdeg' outlined in 3.2.2; further, for this purpose, no distinction is made between GOTO and TEST moves. One simple example will thus be adequate. Reference to Figure 3.5 shows that square 7 is connected to 3, 15, and 35. On the choice of square 7 at level n, therefore, 'stepcount' performs the following operations :



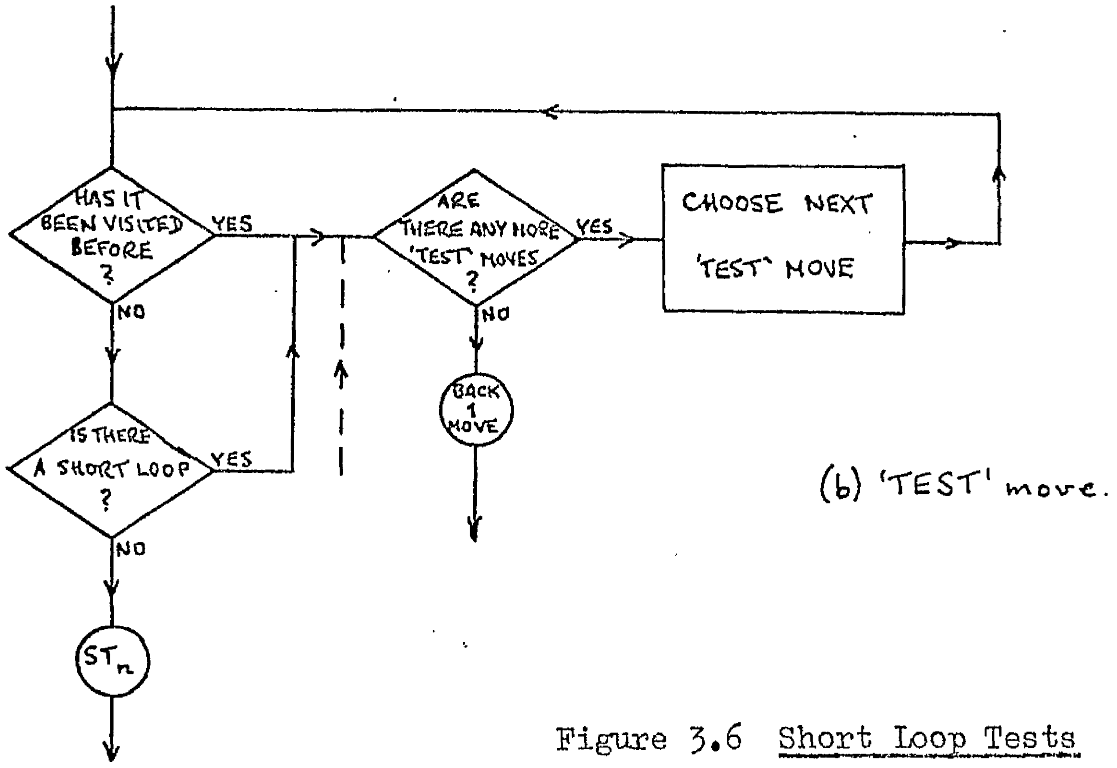


Figure 3.6 Short Loop Tests (detail of Figure 3.2)

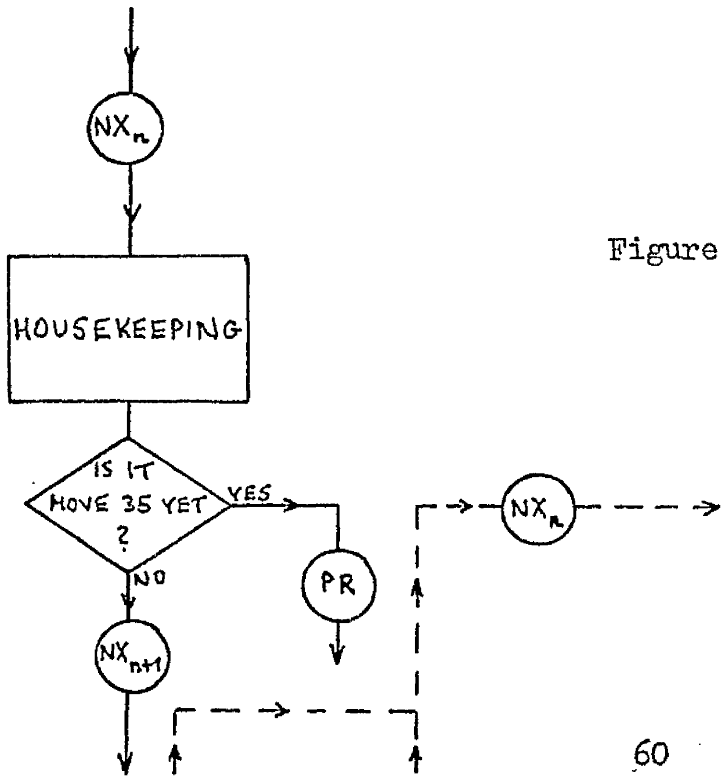


Figure 3.7 Solution exit to Procedure 'Stepcount'

(a) 'newdeg'$[n,7]$ becomes 0

(b) 'newdeg'$[n,3]$ $\begin{cases}\text{becomes, if non-zero, 'newdeg' } [n-1,3] -1. \\ \text{if zero, stays the same.}\end{cases}$

Similarly for 'newdeg' $[n, 15]$ and 'newdeg' $[n, 35]$

(c) For all other k, $1 \leq k \leq 36$, 'newdeg' $[n, k]$ = 'newdeg' $[n-1, k]$.

Effectively, this has removed all the edges incident at square 7 thus reducing its degree to 0, and the degrees of all squares formerly joined to it by one; other squares are unaffected. This is slightly different from what we have designated edge-removal methods which leave used squares with a degree of 2, but only insofar as we can visualize here the edges of the tour being rubbed out as soon as they are determined and the vertices entered in 'storepath' (a function discharged by 'stepcount' immediately afterwards).

This quite lengthy section has described the main features of first-generation programs and how they were used in the implementation of the one-level lookahead tree-search. As should be obvious, we have been primarily discussing the ALGOL version of the program, and indeed most of the work done with the early programs was in ALGOL. The FORTRAN/UCA3 variant was, to a great extent, a development program, as by the time of its completion 2nd-generation ALGOL programs were operational, and the possibility of the 3rd-generation concept of a dynamic conndction matrix stack was under consideration.

Suffice it to say that within the bounds of possibility, the FORTRAN/UCA3 program was a straight translation from ALGOL, using exactly the same methods applied to exactly the same theory; what will concern us later (3.5) is the routine structure of this program, and its modification to produce third-generation programs.

Before mentioning applications of the first-generation programs, we remark that many of the features already explained carry over into the second-generation programs and a few even into the third.

3.3 Applications of 1st-Generation Programs

The first use of the early programs was optimization of the tree-search, clearly the most vital part of the procedure; this was achieved by seeding the program liberally with calls of the procedure 'time' (q.v. super) to give a breakdown of the comparative time consumption by various parts of the program.
The most notable result to emerge was the length of time spent in 'newdeg' updating, and this was consequently programmed more efficiently. While the sizes of the results were not particularly significant due to 'time' itself taking time, the comparison was useful in highlighting slower parts of the tree-search and it also raised the suspicion as to the efficiency of a recursive tree-search.

Other results were obtained through checks being made on 'newdeg' after each move, and by keeping counts of 'reccount' and the number of GOTO moves.

It was thought possible that testing for GOTO moves might be unprofitable at certain levels in the tree (especially high up), and it was determined to find the percentage of moves at each level which were GOTO's. This was effected by keeping two vectors of counters, 'step' and 'gotostep', the latter set being adjusted at each passage through the first (GOTO move) exit to 'nextstep', and the former at both exits. The relevant counters of the two sets were indexed as usual by 'reccount', so that if, for example, a TEST move were made at level 17 then 'step' [17] would be incremented by 1, or if a GOTO move were made on move 22 both 'step' [22] and 'gotostep' [22] would increase. Before close-down, the program printed out the percentages : 'gotostep'[i] x 100/'step'[i] , the results of a sample of about 10% of the entire tree-search being as shown in Table 3.2. We can see that (as expected) the high percentages are concentrated in the lower branches of the search-tree where there are few squares to choose, anyway, but although the figures are only around 33% for the first 10 moves, these are the GOTO moves really worth recognising, since this implies the pruning of huge subtrees (see 2.2.1).

Move	%	Move	%	Move	%	Move	%	Move	%	Move	%
1	0	7	33	13	38	19	37	25	62	31	98
2	0	8	34	14	36	20	47	26	36	32	100
3	14	9	34	15	41	21	43	27	53	33	100
4	31	10	36	16	36	22	44	28	66	34	100
5	34	11	33	17	38	23	53	29	49	35	100
6	31	12	37	18	42	24	47	30	65		

Table 3.2 Approximate Percentages of GOTO moves at each tree-level.

The other application of these programs was to count the number of remaining edges on the equivalent graph at each level, and hence to work out the average number remaining at each stage; further the maximum and minimum number per level was recorded. Taking the degree of a typical square, i, at level n to be δ_i^n, we have the number of remaining edges is

$$E_n = \tfrac{1}{2}\sum_i \delta_i^n ,$$

so that the average number of remaining edges at level n, over j_n downward seeks through the level is

$$\bar{E}_n = 1/(2j_n).\sum_{j_n}\sum_i \delta_i^n ,$$

δ_i^n being a function of the previous moves. A cumulative total of E_n's was kept for each n in a vector 'sumnewdeg', and after an arbitrary j_n seeks (j_n being obtained from 'step'), the $\bar{E}_n$ could be obtained by division. The maxima and minima

$$\begin{aligned} H_n &= \max_{j_n}. \left(\tfrac{1}{2}\sum_i \delta_i^n\right) \\ L_n &= \min_{j_n}. \left(\tfrac{1}{2}\sum_i \delta_i^n\right) \end{aligned}$$

were also stored in vectors, and the requisite members compared against the current E_n for every move.

Interest was focused initially on the L_n in order to see if the values were ever such as to imply less than one remaining edge for each remaining move, an obviously unsatisfactory situation.

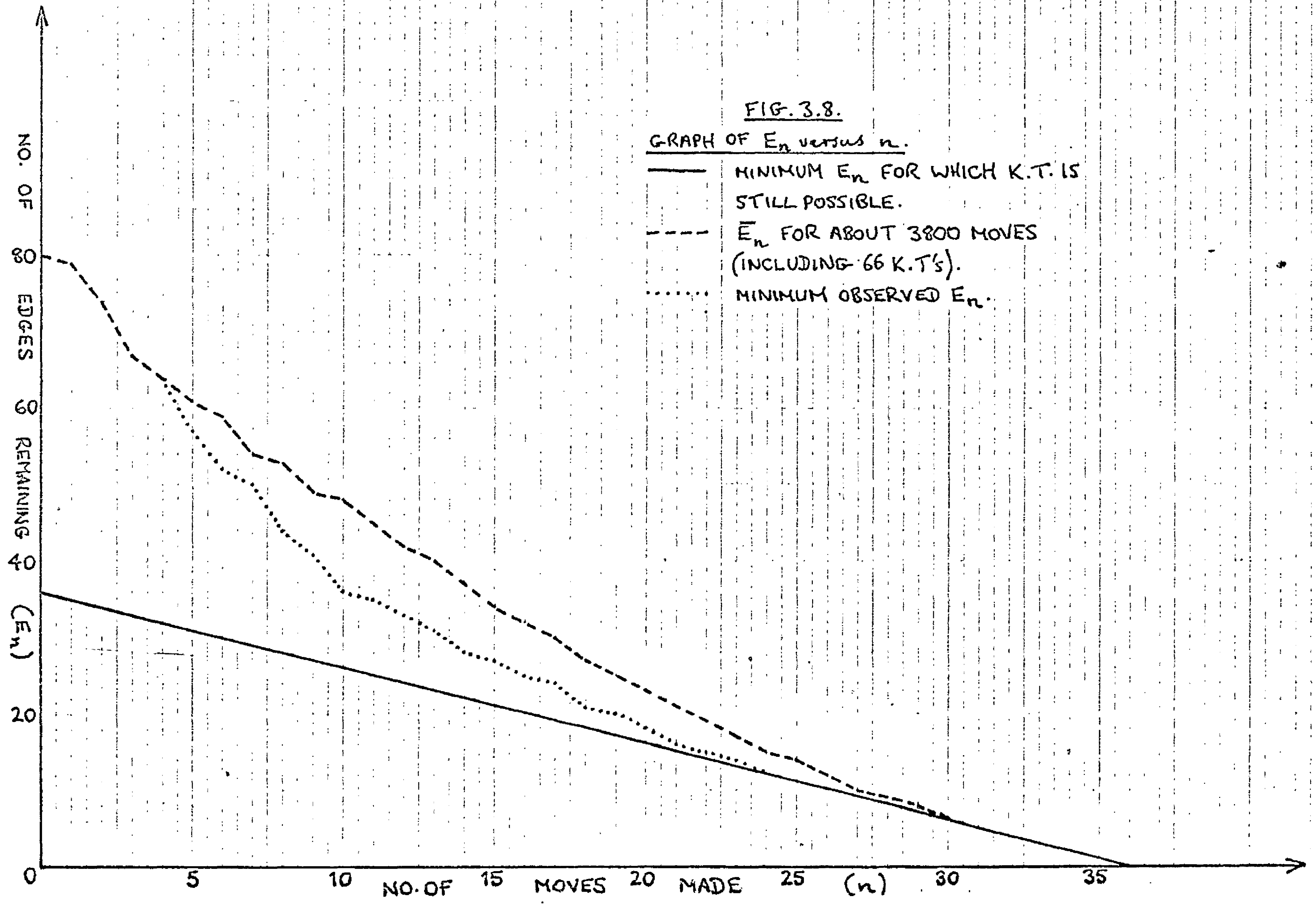


FIG. 3.8.

GRAPH OF E_n versus n.

Figure 3.8 shows, however, that this does not occur from which we may conclude that the GOTO and nonrepetitivity criteria are stronger than that just explained. It is, in fact, the GOTO lookahead which precludes all possibility of the number of edges remaining/move being less than one, for if this were to occur, it would imply an unvisited isolated vertex, and this we have rendered impossible by the GOTO criterion (2.2.1).

The other interesting observation we can make from Figure 3.8 is that $\frac{d^2\bar{E}_n}{dn^2}$ is negative for the small sample taken (about 3800 moves), which suggests that edges are removed rapidly at first, but more slowly in later moves. We mention this as it provides a direct contrast with the policy of the Warnsdorff algorithm which attempts to remove as few edges as possible in the early moves; many actual tours depart considerably from this strategy, which would tend to produce $\frac{d^2\bar{E}_n}{dn^2} > 0$.

3.4. Second-Generation Programs We have mentioned that first-generation step-timing programs seemed to indicate a considerable amount of time being spent on stack maintenance due to the recursive structure employed, and tests on a simple loop structure using control transfers were inaugurated. While these programs might not be considered sufficiently different from the preceding ones to justify a separate 'generation', there is within this small change the nub of the thinking which led to third-generation runs. These programs could not, therefore, properly be designated 1 (a) or pre-3; they are a crucial turning point in the development.

A feature of ALGOL is that variables used locally to a recursive procedure are duplicated at each entry to the procedure, so that the use of 'j' at all levels as a marker to indicate the branches not yet tried leads to no ambiguity so far as a recursive technique is concerned. Each level has a 'j' implicitly subscripted, i.e. local to itself, thus ensuring that when a backtrack is made, the top version of 'j' is lost and that corresponding to the level of the tree nearer the root again becomes available with its previous value intact; thus the 'j's can be thought of as own variables, the scope of which are determined by the value of 'reccount'. The necessity for retaining the counter values is stressed by Figure 3.9. Writing subscripts explicitly, to choose from level r a TEST move to level (r+1), j_r is set to zero, and the squares tried in turn. If the first is chosen, $j_r = 1$. The process is repeated at level (r+2). If none of these squares is chosen, j_{r+1} having gone from 0 to 3, is thrown away, and we backtrack one level.

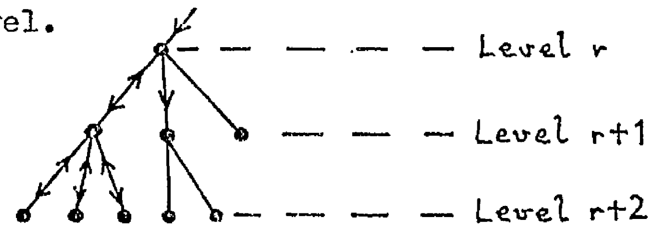


Figure 3.9 Level branch counters

At this juncture, it is vital to know which branch from level r we have just ascended; this information is given by j_r. The tree-search now continues by putting $j_r = 2$, and resuming downward searching.

Having emphasised the importance of having a unique branch counter at every level, we must now heed the consequences of removing the aforesaid implicit subscripts imposed by recursive usage. When we use our so-called iterative method, the instructions for choice of a TEST move are at the same block-level whatever the value of 'reccount'; indeed 'reccount' is the only distinguishing attribute of a level in the iterative tree-search (see Appendix 1), and thus, since 'j's for different levels are not protected by block structure, they must be explicitly subscripted in the program, using the ubiquitous 'reccount' for an index marker; this vector in 2nd-generation programs is called 'jj'.

A similar argument applies to 'd' which represents the last square chosen. In first-generation programs, 'd' is 'owned' by recursive procedure 'nextstep', so a unique 'd' exists at any time for all the levels up to the one currently occupied; for a second-generation program, however, the last version only of 'd' can be accessed, all previous ones having been overwritten. The effect of this backtracking is simply as follows (implicit subscripts) :

At level r, first-generation has d_1 , d_2 , ... d_{r-2} , d_{r-1} , d_r ; second-generation only has d_r . Backtrack two levels: first-generation loses d_{r-1} , d_r and 'd' now refers now refers to d_{r-2} as required; but for for 2nd-generation, 'd' still means d_r which is the square after the one currently occupied .

Hence, in an iterative environment, 'd's must also be effectively subscripted; this was done, not literally by declaring an array of 'd's, but by re-assigning a value to 'd' from the list of completed moves in 'storepath'.

All other major characteristics of the first-generation program were retained, particularly the 'driving forces' of 'XX' and 'newdeg', but the merging of procedures 'nextstep' and 'stepcount' into the main stream of the program also produced the need for another vector called 'link'. It will be recalled that in Figure 3.2 two exits to 'stepcount' from 'nextstep' were possible, those corresponding to a choice of a GOTO and a TEST move; we have already shown the call for retention of counters after choosing a TEST move in the inevitable event of backtracks: it is just as vital to ensure return to the right point in the coding for square-choice when backtracking. In 1st-generation programs, the use of recursive procedures overcame any difficulty here, but the simple 'top-to-bottom' ALGOL of the 2nd-generation required explicit markers to be set after each choice of move indicating a TEST move or a GOTO; these markers were stored in the boolean array 'link' and referenced by 'reccount'.

To preserve similarity between the generations, the coding equivalent to procedures 'nextstep' and 'stepcount' was labelled NXST: and STCT: in generation 2; the structure of that section of these programs corresponding to Figure 3.2 is shown in Figure 3.10.

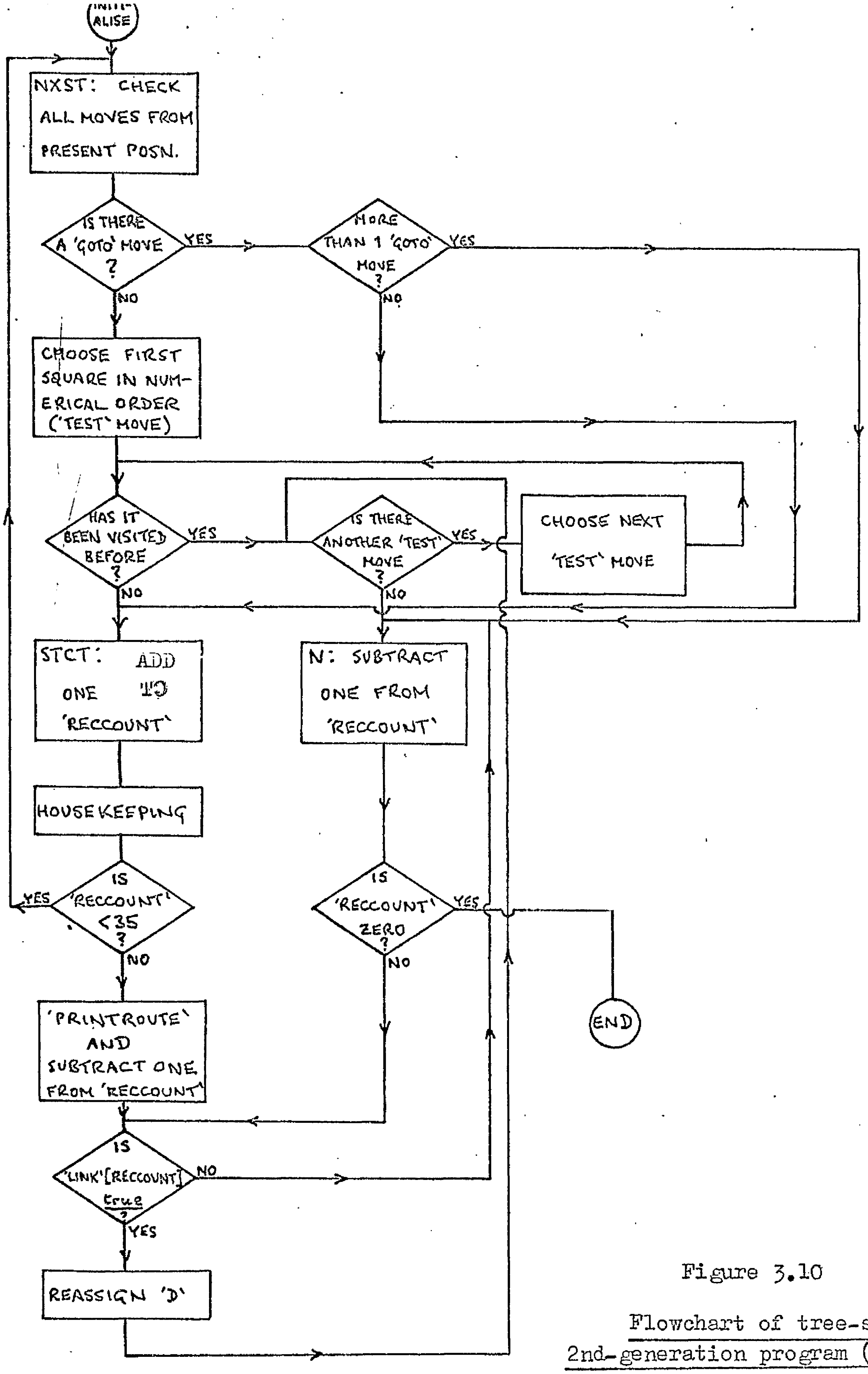

INITI-ALISE
NXST: CHECK ALL MOVES FROM PRESENT POSN.
IS THERE A 'GOTO' MOVE ?
YES
NO
MORE THAN 1 'GOTO' MOVE ?
YES
NO
CHOOSE FIRST SQUARE IN NUMERICAL ORDER ('TEST' MOVE)
HAS IT BEEN VISITED BEFORE ?
YES
NO
IS THERE ANOTHER 'TEST' MOVE ?
YES
NO
CHOOSE NEXT 'TEST' MOVE
STCT: ADD ONE TO 'RECCOUNT'
N: SUBTRACT ONE FROM 'RECCOUNT'
HOUSEKEEPING
IS 'RECCOUNT' <35 ?
YES
NO
IS 'RECCOUNT' ZERO ?
YES
NO
END
'PRINTROUTE' AND SUBTRACT ONE FROM 'RECCOUNT'
IS 'LINK'[RECCOUNT] true ?
NO
YES
REASSIGN 'D'

Figure 3.10

Flowchart of tree-search in 2nd-generation program (iterative)

(short loop test omitted)

Notice particularly that a simple and quick loop for GOTO backtracks is provided between the discrimination on 'link' ['reccount'] and label N as suggested by 2.2.1 ('link' ['reccount'] is set <u>true</u> or <u>false</u> according as the ('reccount')th. move is TEST or GOTO). We have observed that replacing a recursive structure by an iterative one has occasioned the introduction of two new vectors 'jj' and 'link' and extra housekeeping; this might be expected to nullify to a great extent the advantage of cutting out repeated procedure calls, and in fact it was found that an improvement in speed of only 6% resulted. The door to the third-generation programs had been opened, though, but before describing this, we must first make mention of an important modification made in late generation 2.

3.4.1 <u>Graphic Output</u> This title was coined for the knight's tours printed out in board - rather than strip-form; examples appear in Appendix 2. The board is assumed numbered asymmetrically as in Figure 2.5, and the numbers actually displayed are the number of squares visited (including the presently occupied one); this has the attraction of demonstrating the sequence of knight's moves as they are followed on a chessboard, and is therefore much clearer. It does, however, require an extra vector 'board' and additional housekeeping, but it is not used solely for output.

We find that there are basically two things we are interested in at any stage of the tour:-

(a) has a particular square been visited ?

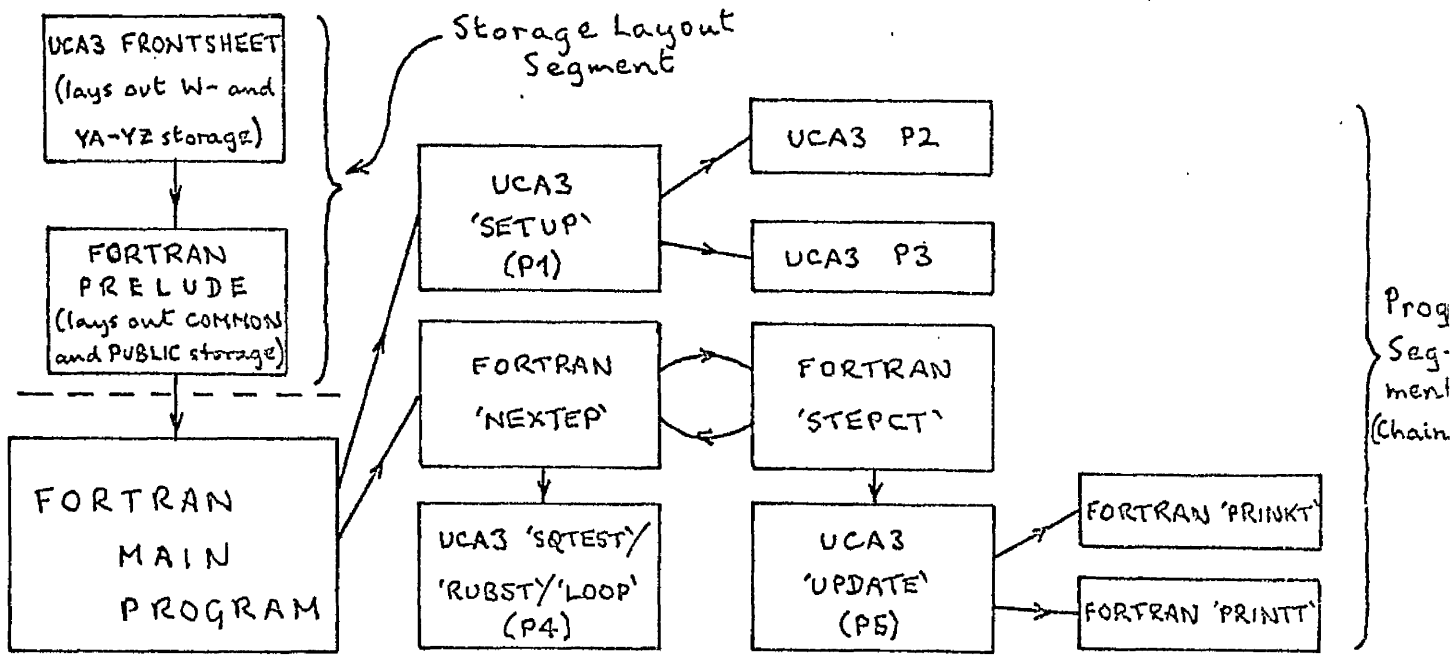
(b) which was the last square visited ?

Question (b) can very readily be answered by looking at the last (significant) element of 'storepath', but (a) would need a search of all its significant elements. We can therefore find a use when seeking a TEST move for a vector which contains in position i the integer p if square i was the pth. to be visited and zero if the square is so far untouched. This is exactly what 'board' contains; and when a tour is found and all 36 elements are filled, the vector can be printed out in 6-element strips to assume the appearance of a board.

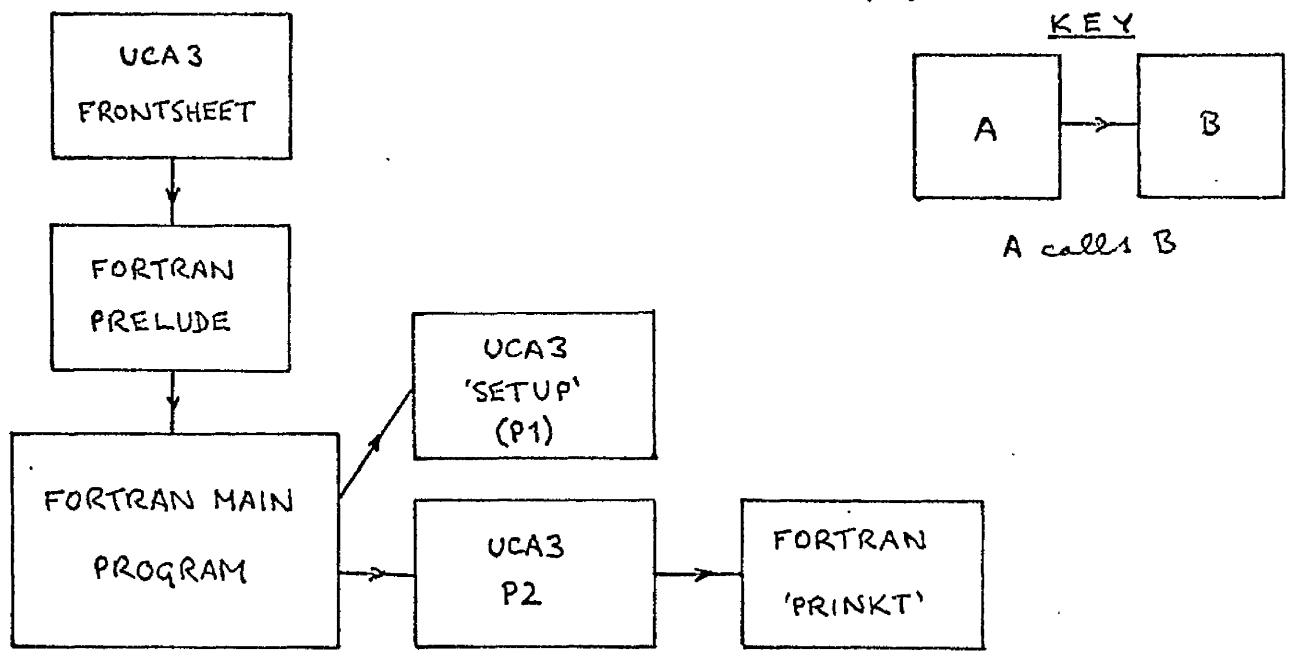
3.5 <u>Third-Generation Programs</u> The development of generation 3 happened in two stages, the first attributable to generation 1, and the second to generation 2. The first stage was the realization that if faster tour-searches were to be achieved, the searching would have to be as far as possible in assembly language, which to exclude the problems of recursive Usercode or UCA3,- meant ALGOL/Usercode or FORTAN/UCA3. We chose the latter in view of the availability of the much vaunted Egtran FORTRAN compiler, and also because the FORTRAN routine structure, extended in Egdon to UCA3 routines, offered a neat compartmentalization of languages (and purposes).

The second stage came with the foundation of iterative methods: these enabled a gigantic simplification of the program structure as shown in Figure 3.11. In first-generation Egdon programs, the ALGOL procedures 'nextstep' and 'stepcount' were written as recursive FORTRAN subroutines 'NEXTEP' and 'STEPCT' calling one another; this, together with 'reccount' updating, was their only purpose, however: they were 'shell' routines. Each included a call of a (non-recursive) UCA3 routine containing a large quantity of coding and carrying out the functions of testing squares and housekeeping. For example, after choosing a square, control would be returned from P4 to NEXTEP which would then call STEPCT which would call P5 which would effect the housekeeping ; iteration permitted the compaction of P4 and P5 into one piece of UCA3 coding without any need for FORTRAN as an intermediary.

Few features of the original EGDON programs have been retained, the structure having been greatly simplified, and the techniques being the same as those of 1st-generation ALGOL programs, anyway. We will omit, therefore, discussion of the use of side-entries in P4 and V-store preservation buffers, and merely include a specimen program in Appendix 1 for interest; perhaps it is worth remarking that the initialization routines SETUP, P2, and P3, differ little from the preliminary statements of the 1st-generation ALGOL programs, and indeed throughout the development of the lookahead tree-search these overtures have consisted of the operations of presetting counters, zeroing arrays, and setting up the zero level of the driving dynamic array, whether it be degree vector or connection matrix (see below), consequent to the choice of square 1 as starting position.



(a) 1st-Generation recursive FORTRAN/UCA3 program structure



(b) 3rd-Generation iterative structure

Figure 3.11

Figure 3.11(b) shows the streamlined Egdon program of the third-generation with initialization conducted by SETUP and all tree-searching by P2; with a controlling main routine in FORTRAN and knight's tour output (in graphic form) also in FORTRAN. We have shown how 2nd-generation ideas led to the structure being recast in these terms; now we must discuss in some depth the new programming tools utilised in generation 3, and briefly describe the Egdon programming system where this has special relevance to further events.

3.6. The Egdon System This is a comprehensive system for compiling and running programs on the KDF9, using a disc unit as random-access backing store (~ 4M 48-bit words of storage). The disc is divided into two equal areas of ~2M words, one of which contains all the system's files - system programs, library routines, user's programs, and data-blocks - and the other is available to the user as work-space for his program (i.e. temporary storage only).

All peripheral operations are controlled by the Director, a master program which, in the simple Egdon 2 system, is permanently in core, but which in the Egdon/COTAN (on-line) system is composed of segments some of which may be written out to or called in from disc from time to time.

Programming facilities consist of a FORTRAN (II+) compiler and a UCA3 assembler, and a program may consist of any combination of FORTRAN or UCA3 main routine and an unspecified number of FORTRAN subroutines (or function subprograms) or UCA3 P- or L-routines.

3.6.1 Intercommunication between FORTRAN and UCA3 In the Egdon system, UCA3 and FORTAN routines can call each other with complete freedom, but a UCA3 routine which calls or is called by a FORTRAN routine must observe certain conventions. The link (the address to which the called routine returns control when it has finished) is left in the top cell of the subroutine jump nesting-store (SJNS) by the calling routine. If the called routine in turn calls other routines, the link is preserved in VØ to prevent the possibility of SJNS overflow. For the purposes of describing transmission of arguments, the Egdon programming manual introduces the term 'value'; where the 'value' of an argument is on entry depends on whether the called routine is a function subprogram or a subroutine - only the latter need concern us, and in this event, the address of the 'value' is left in the top cell of the nesting-store; what the 'value' is depends on whether the argument is a variable or an array name: in the latter case, the 'value' is the address of the first element of the array, in the former, the 'value' is merely the current numerical value of the variable. Thus a FORTRAN routine transmitting a vector J as an argument to a UCA3 routine leaves in the top cell of the nest (N1) the address of J[1].

One of the features of communication between routines in Egdon is that in some circumstances the 'value' is planted in own variables of the called routine; in the case of a called UCA3 routine, this means V-stores may be overwritten, and care has to be taken to declare a sufficient number for this purpose.

Q1 and Q2 have also to be preserved in all routines, but any other Q-store can be used as convenient, though it cannot be guaranteed that they will be preserved by routines called by UCA3 routines.

3.6.2 Storage Layout As indicated in Figure 3.11 (a), the layout of certain storage areas is effected by preliminary segments of coding, prior to entering the program segment. The reservation of UCA3 storage of W and YA-YZ stores has to be carried out first by a separate assembler called the Frontsheet assembler; and FORTRAN (non-local) common and public variables and arrays are then laid out at the top end of core by a prelude segment which arranges this by allocating storage space down the core (in successively lower addresses) and then filling the space up the core. A map of this, together with explanations of the uses of the common list and the common array area and the correspondences between these and the Z-stores will be found in the Egdon manual (11). We give below (Table 3.3) a map of the common and public storage areas for 3rd-generation programs, but a detailed explanation will not be included.

PRELUDE NAME	FORTRAN (CHAIN SEGMENT) NAME	UCA3 NAME	
BA(INTAR1)	BA(KEEPP)	Z1	Common List
BA(INTAR2)	BA(BOARD)	Z2	
BA(INTAR3)	BA(ORGD)	Z3	
KTTOT	KTTOT	Z4	Public list
-	KEEPP(36)	Z32	Common Array Area
-	KEEPP(1)	Z67	
-	BOARD(36)	Z68	
-	BOARD(1)	Z103	
-	ORGD(36)	Z104	
-	ORGD(1)	Z139	

Table 3.3 Storage Map of High-Address end of Core

(BA = 'base address')

The decision whether or not to place an array or variable in common or public storage was largely resolved by consideration of whether the quantities in question were to be accessed both by FORTRAN and UCA3 routines, when the Z-storage correspondences made addressing a simple matter. The connection matrix stack (q.v.) was accessed solely by UCA3 routines, in accordance with 3.5, and hence was placed in YA-stores, while BOARD was used in the FORTRAN output routine, and also in P2 (Figure 3.11(b)) so necessitating a common declaration.

3.7 The Dynamic Connection Matrix In the first- and second-generation programs, the combination of 'XX' and 'newdeg' meant that at any level in the tree-search there were known the legal knight's moves from the square occupied and the number of l.t.m's from that square; there were no means of obtaining promptly those l.k.m's which were also l.t.m's - all had to be tested in the normal way (3.2). This was obviously wasting time by taking no account of the decreasing number of l.t.m's and the concept of a dynamic stack of connection matrices was initiated to overcome this.

Basically, the idea is to update the connection matrix itself so that a version exists for each level of the tree above that occupied; we use the nomenclature XX_i for the connection matrix for level i, and extend first the idea of a condensed connection matrix (3.2.3) to dynamic stacks.

Suppose XX_i is an 8 x 36mx. (i.e. the transpose of that described in 3.2.3) with elements defined as before so that this matrix specifies all the moves that can be made from every square, taking into account those already visited. To form XX_{i+1} :-

(a) Choose a square j,

(b) The column corresponding to j is set to zero, i.e.

$$XX_{i+1}[k,j] = 0, \quad 1 \leq k \leq 8$$

(c) All references to j in other columns are removed and each column nested up to leave no zero's in the middle.

This procedure leaves no trace of j at levels $(i+1)$ and greater, i.e. we have 'disconnected' the vertex from the equivalent graph; note, though, that just as we did for 'newdeg' in 3.2.2, we 'throw away' connection matrices when backtracking, so that if we revoke the choice of square j, XX_{i+1} will be lost and in XX_i j will be an 'available' square again. Figure 3.12 shows an example of updating a connection matrix in this way, which is analogous to the transition from Figure 3.4(a) to Figure 3.4(b).

The important feature of this method is that there does exist one copy of the whole connection matrix associated with each move kept in KEEPP ('storepath' of generation 1 and 2). That a method which does not exhibit this technique can be implemented will now be shown.

$$\begin{array}{cccccccccccccccc}
0 & 7 & 5 & 8 & 3 & 10 & 2 & 4 & 5 & 2 & 2 & 8 & 1 & 2 & 5 & 3 \\
0 & 10 & 7 & 13 & 9 & 13 & 3 & 12 & 11 & 6 & 9 & 10 & 4 & 3 & 8 & 6 \\
0 & 11 & 14 & 0 & 15 & 16 & 0 & 15 & 13 & 12 & 15 & 16 & 6 & 0 & 11 & 12 \\
0 & 14 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0
\end{array} \quad \xrightarrow[\text{square 8}]{\text{choose}}$$

(a) XX_i

$$\begin{array}{cccccccccccccccc}
0 & 7 & 5 & 13 & 3 & 10 & 2 & 0 & 5 & 2 & 2 & 10 & 1 & 2 & 5 & 3 \\
0 & 10 & 7 & 0 & 9 & 13 & 3 & 0 & 11 & 6 & 9 & 16 & 4 & 3 & 11 & 6 \\
0 & 11 & 14 & 0 & 15 & 16 & 0 & 0 & 13 & 12 & 15 & 0 & 6 & 0 & 0 & 12 \\
0 & 14 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0
\end{array}$$

(b) XX_{i+1}

Figure 3.12 Dynamic Stack Updating Technique

3.7.1 Pushdown Vector Method This method is more economical in appearance, but we shall see that it in fact compares unfavourably with the above. The scheme is to consider each column of the connection mx. the top element of a pushdown list of vectors; when the move is chosen, not the whole matrix is updated, but only those columns which are to be altered (it will be seen in Figure 3.12 that many columns are unchanged between (a) and (b)). Altered versions of columns are filed in the top cell of the appropriate pushdown list, everything below being nested down one level. The depth of each pushdown list will be determined by the maximum number of states that a column can take, for a board of 6 squares a side this being 8. The exact sequence followed is:-

(i) Downward Move

(a) Choose move to square j.

(b) Zero in $XX_1(k,j)$, $1 \leq k \leq 8$. The former XX_1 is written into XX_2, XX_2 into XX_3, and so on.

(c) The squares connected to j are obtained from the new XX_2.

(d) Suppose these are $a_1, \ldots\ldots a_p$; then $XX_m(k, a_q) \rightarrow XX_{m+1}(k, a_q)$ for $1 \leq m \leq 7$, $1 \leq k \leq 8$, $1 \leq q \leq p$. $XX_1(k, a_q)$ is given by (for every q): if $0 \leq XX_2(k, a_q) < j$, then $XX_1(k, a_q) = XX_2(k, a_q)$; if $XX_2(k, a_q) \geq j$, then $XX_1(k, a_q) = XX_2(k+1, a_q)$.

(ii) Backtrack

(a) Find previous square, k, from KEEPP.

(b) Pop-up this list in the array by overwriting $XX_\ell(m, k)$ with $XX_{\ell+1}(m, k)$, $1 \leq \ell \leq 7$, $1 \leq m \leq 8$. The vector $XX_8(m, k)$ is filled with zeroes.

(c) $XX_1(m, k)$ now contains the numbers $a_1 \ldots\ldots a_p$ of those squares which were connected to j, so we can pop-up the lists $XX(m, a_q)$, $1 \leq q \leq p$, in the manner of (b).

To imagine the process described above working in practice, consider the matrix of Figure 3.12(a) to be XX_1 before choosing square 8, and assume this to be the first square chosen; by convention, then, the matrices XX_2 , XX_3 , ... , XX_8 are zero. Figure 3.12(b) shows XX_1 after 8 is chosen, and XX_2 consists of 8,13, 0,0 in column 4; 4, 12, 15, 0 in column 8; 8, 10, 16, 0 in column 12; 5, 8, 11, 0 in column 15, with zeros elsewhere, and XX_3 , ... , XX_8 still untouched.

The feature of the method is that the current state of the connection matrix is always held in the same addresses, so that addressing for testing moves is extremely easy; updating, being piecemeal, is kept to a minimum. The disadvantages are that there exists only one complete connection matrix at any time - that for the current level - and this implies updating both when moving down and backtracking; updating is also more complicated than for the former method.

3.7.2 <u>Bit-Matrix Representation</u> The dynamic stack method requires $4n^2$ levels in the stack for a 2n x 2n board, which means $128n^4$ words for the whole array. If n = 3 (6 x 6), the storage required is some 10K of core, but for n = 4, 32K are needed, and thus the available core-space of KDF9 would be exceeded.

An alternative representation is to store the array in effectively boolean form with the position in the computer 'word' the significant attribute; for the 6 x 6 board, the arrangement we use is as follows:- XD is declared as a 36 x 36 array, one dimension giving 36 levels (or layers), one dimension 36 connectors, and 36 bit-positions in each word (D∅-35) giving the 36 connectees. Then

$$XD\ (i,j),\ D_k \begin{cases} = 1 \text{ if square } j \text{ is connected to square} \\ \quad (k+1) \text{ at level } i \text{ and} \\ = \emptyset \text{ otherwise } (1 \leq i\,,\ j \leq 36;\ 0 \leq k \leq 35). \end{cases}$$

This definition is specially designed to take advantage of the instruction set of UCA3 and reduces considerably the problem of addressing a particular word in the array. For an 8 x 8 board, this advantage would not be applicable since there are 64 possible connectees, and this needs 2 KDF9 words which are best kept in consecutive addresses; the consecutive addressing instruction of UCA3 could not then be used (as it was in the 6 x 6 case) for fast construction of level i+1 from level i in the stack.

To describe the transformation to be executed at update-time in the dynamic stack case we must emphasise the following nomenclature which is crucial to an understanding of the process :-

'Word' has its usual significance, i.e. 1 48-bit KDF9 word. By 'the array' we mean XD, i.e. 36 words x 36 words 2-dimensional array in the ordinary sense. By 'matrix' we mean a 2-dimensional array in the unusual sense that rows are indexed by specifying particular words; but columns are indexed by specifying bits within these words; each of these 36 words x 36 bits matrices is a complete connection matrix.

An illustrative analogy is that of a square mail-sorting frame, sunk into a wall so that we see it as a two-dimensional entity, but whose pigeon-holes have depth; furthermore some particular significance is attached to the distance from the front of the pigeon-hole that a letter is filed. Returning to the updating procedure on the connection matrix in the dynamic stack case, we choose a square (k+1), at level i, say, then, following precedent, we want to remove all mention of (k+1) from the (i+1)th. connection matrix. All squares connected to (k+1) have 1's in the relevant bits of XD (i,k+1) so this word must be zeroed (by logical 'and' with a zero word); and all squares to which (k+1) is connected (note the distinction) have a 1 in bit D_k of the relevant words of row i of XD, i.e. XD (i, a_q) so we must perform a logical 'and' between these rows of bits and a word which is all 1's apart from D_k. In practice, since no other words in this row of the array have a 1 in D_k , we can perform the same operation on them, so that we actually program the transformation as :-

$$\mathrm{XD}\ (i)\ \underline{\text{and}}\ \begin{array}{cccccc|c} & & & D_k & & & \\ \hline & & & 0 & & & \\ & 1 & \cdot & 0 & 1 & & \\ & & & 0 & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & \text{row } (k+1) \\ & 1 & & 0 & 1 & & \\ & & & 0 & & & \end{array} \longrightarrow \mathrm{XD}(i+1),$$

(XD (i) being shorthand for XD (i,j), $1 \leq j \leq 36$). The operation is carried out word by word, i.e.

XD (i,1)	<u>and</u>	1 . . . . 101 1	XD (i+1,1)
XD (i,2)	<u>and</u>	1 . . . . 101 1	XD (i+1,2)
- - - - -	- - - -	- - - - - - - - - - -	- - - -
XD (i,k+1)	<u>and</u>	0 0	XD (i+1, k+1)
XD (i,k+2)	<u>and</u>	1 . . . . 101 . . . . 1, etc.	

Note that D36-48 of every word in the original connection matrix is set to zero so that the '<u>and</u>' operation has no effect on them. The pushdown vector method can similarly be applied to a similar boolean matrix, but rather than vectors of words being pushdown lists, here vectors of bits i.e. individual words are the stacks, the method being analogous to the above except that XD (i) is updated piecemeal as before and thus only 8 levels ($1 \leq i \leq 8$) are needed. We see that the 'bit-by-bit' representation forms a compact method of storing all the 'driving' mechanism required for the tree-search - for a 6 x 6 board 1296 words are used: an 8 x 8 would need ~~8092~~ 8192. Comparative tests of the pushdown vector (288 and 1024 respectively) and dynamic stack techniques showed the latter to be faster, and this was the updating scheme thereafter adopted in generation 3 programs.

The author feels that some of the concepts of this section are hard enough to visualize, let alone explain, and therefore an ALGOL version of of the tree-search with specimen results has been included in Appendix 1 in the hope that this may at least clarify any obscure points raised in the text. The program uses the full 3-dimensional array (called 'XX') of 3.7, but adds the refinement that when square j is chosen at level i, row j of XX_i is zeroed, and then the square's number in the order of moves (i.e. (i+1)) is entered in the first element of the row. This conserves space by using part of 'XX' in place of 'board', and the complete tour can be printed in exactly the fashion of 3.4.1.

3.8 <u>FORTRAN Identifiers</u> The limitation of a length of 6 characters on FORTRAN identifiers meant alterations were necessary to generation 2 programs which only had the generous Kidsgrove ALGOL requirements to meet. The significance of the initial character in determining in FORTRAN integer and non-integer variable types has caused one or two radical changes, but largely, ALGOL identifiers have merely been contracted to within the 6 character limit.

ORGD replaces 'origdeg' but rather than containing the degrees of all the squares before even the choice of square 1 as its predecessor did, ORGD has the 'bit-by-bit' connection matrix for the initial state, these 36 rows of 36 bits each being read in in octal form in the main routine. KEEPP is the exact equivalent of 'storepath', KTTOT of 'ccount' (3.2.6), and BOARD of 'board'; the storage limit of these arrays is shown in Table 3.3. Subroutine PRINKT corresponds closely with the ALGOL procedure 'printroute'. P2 combines the functions of 'nextstep' and 'stepcount', and the initialization described in comment at the head of P1 was explained in 3.5.

3.9 Speed of 3rd-Generation Programs The type of output produced by 3rd-generation programs is shown in Appendix 2, the actual process of printing tours proving time-consuming and slowing the rate of tour-discovery to around 4 tours/second. A version of the program intended only to enumerate the tours found 9862 (in agreement with Duby's results) at an average rate of very nearly 8 tours/second. (run-time of 20minutes 42 seconds).

Duby (4) claims a program speed some 10 times faster than this (partly or wholly attributable to the 7094/KDF9 machine-speed factor), and still finds the 8 x 8 search an discouraging proposition, maintaining that there may be over 10^6 knight's tours on a standard chessboard. In view of the statement that the 75,000 tours found all had the first 35 moves in common, it seems in order to suggest a total more of the scale of 10^{26}! Whatever the final total might be, our very crude estimate of the size of tree-search indicates that the 8 x 8 search will be much larger, probably, without the aid of better pruning algorithms, too large ever to be solved in a single run of a program; one might hope, however, that a combination of faster machines and a segmented tree-search such as that outlined in Chapter 4 will eventually lead to a solution of the knight's tour problem on the standard chessboard. Some of the methods of Chapter 2 could very well, as we indicate in the Conclusion, prove more fruitful in payment for greater attention than it has been possible to accord them in the duration of this work.

In summary, we have traced in this chapter the detailed development of the implementation of the one-level lookahead tree-search, beginning with the recursive ALGOL programs driven by a static condensed connection matrix and dynamic stack of degree vectors and printing solutions in strip-format; we finished by describing the heavily assembly-language-oriented FORTRAN/UCA3 programs with graphic output and dynamic connection matrix stack in bit form activating the tree-search. In the course of this development the speed of the search was improved by a factor of about 150 (the improvement due to the introduction of the one-level lookahead in the first place is not readily calculable but must be much greater - 3.1).

CHAPTER 4

Extensions & Results

4.1 Introduction At the end of the last chapter, we stated the result which was the primary target of this work, i.e. the number of Hamiltonian cycles or knight's tours on a 6 x 6 chessboard is 9862. This is a result which brings with it the unyielding intractibility that the author has found to characterize this problem throughout: the smallest set of solutions for any size of board contains 9862 members. The discovery of significant properties amongst a set of such magnitude immediately raised a pattern recognition problem of vast dimensions: how do we classify the tours? How do other tree-searching methods stand comparison with the one we have used? We give an account in this chapter of methods utilising variations of the basic generation 3 program already described (3.7-8) which try to give some idea of the answers to these questions.

4.2 Classification by Corner-Order We investigated in 2.5 the feasibility of matching 4 strings each of which included as initial and terminal square one of the 4 corners of the board. Now a relatively minor alteration was made to the basic tree-searching program to cause output at the first corner reached as well as reversing the level pointer in this event; the average depth of search was considerably reduced, and corner-to-corner strings found some 4 times faster than knight's tours. The total number of strings found was 4471, enough to make the prospect of the matching process very unattractive. The study of the symmetry of tours can be approached from the angle of these corner-to-corner strings, however, and this in turn gives a broad classification of the tours.

4.2.1 We make the following

Definitions A C-square is a corner-square, and an N-square is square separated by 1 l.k.m. from a C-square. Thus for the asymmetric board-numbering of Figure 2.5., the C-squares are 1,6,31,36 (called hereafter corners 1,2,4, and 3 respectively) and the N-squares are 9,10, 14, 17, 20, 23, 27 and 28. We say that each C-square has 2 N-squares associated with it, and that relative to another C-square one of these is the inferior and one the superior N-square.

If a, b are 2 C-squares then the superior N-square associated with a relative to b, (s(a,b)), is that N-square associated with a which lies on the opposite side of the board diagonal through from b. Similarly the inferior N-square, (i(a,b)), lies on the same side of the diagonal as a. Figure 2.5 shows that the superior N-squares and inferior N-squares for each pair of corners are as in Table 4.1; note that this concept does not apply to diagonally opposite C-squares.

Now let us suppose that the 3 C-squares a_1, a_2, & a_3 are visited in that order: we concentrate on the corner a_2. If a_1 and a_2, and a_2 and a_3 are not diagonally opposite corners, then if the order of traversing a_2 is s (a_2, a_1), a_2, s (a_2, a_3) we say the knight's tour has a <u>loop</u> at a_2, and if the order is $i(a_2, a_1)$, a_2, $i(a_2, a_3)$ we say it has a <u>cusp</u>. These terms are illustrated in Figure 4.1, where we show a cusp at a1 and a loop at a3.

That is, if a C-square a, bounded by 2 strings of l.t.m's joining it to 2 (distinct) diagonally opposite C-squares is traversed via 2 superior N-squares, then there is a loop at a; if it is traversed via 2 inferior N-squares there is a cusp at a. If a is joined to two C-squares not diagonally opposite, then a is diagonally opposite to one of these squares, so the respective requirements for a loop and a cusp are 1 superior N-square, and 1 inferior N-square.

A <u>diagonal C-string</u> is a string of l.t.m's joining two diagonally opposite C-squares; an <u>adjacent C-string</u> is any other string of l.t.m's joining 2 C-squares.

Table 4.1

(a and b are corner nos. not square nos.)

a	b	s(a,b)	i(a,b)
1	2	14	9
1	4	9	14
2	1	17	10
2	3	10	17
3	2	28	23
3	4	23	28
4	1	27	20
4	3	20	27

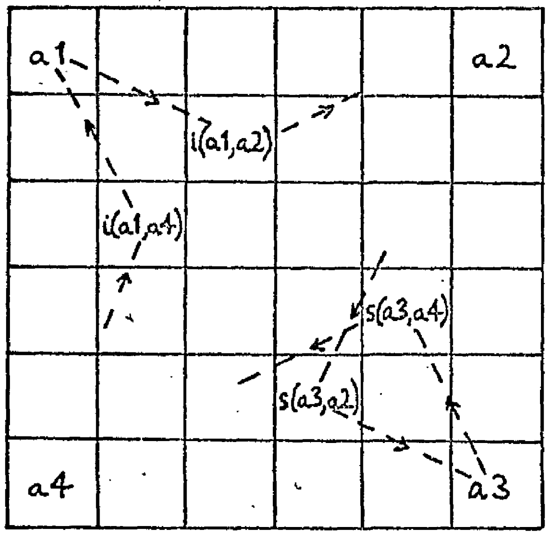


Figure 4.1 Cusps & Loops

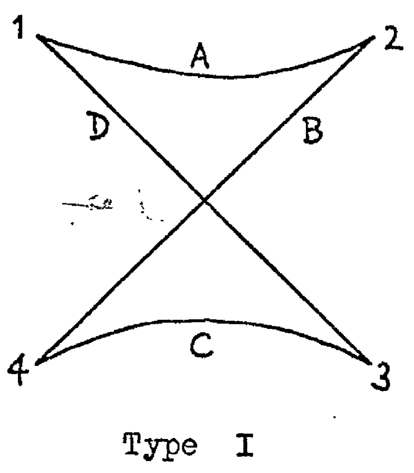
In 2.5 we discussed two orders of tours traversing corners and we can now define them specifically using the nomenclature we have just introduced.

Definition A Type I knight's tour is one which consists of two diagonal C-strings and two adjacent C-strings.

A Type II knight's tour is one which consists of four adjacent C-strings.

These two varieties of tours are shown schematically in Figure 4.2.; it should be remembered that long interwoven strings of moves separate the C-squares, and these diagrams merely indicate which corner is joined to which, without even indicating whether loops or cusps exist.

4.2.2 Further Classification We can now make use of the varying ways of traversing a particular corner to subdivide the two classes so far introduced: all the possibilities are shown in Figure 4.3.



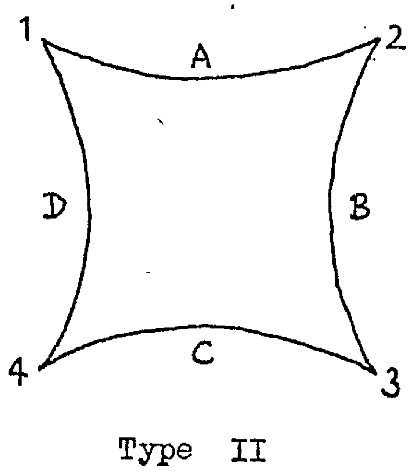


Figure 4.2 Knight's Tour Classification

We have not yet introduced any considerations of symmetry into the tours themselves - this subject is dealt with in the next section. It is evident however, on inspecting the diagrams of Figure 4.3 that several configurations have an inherent symmetry attached: why only some of these are realised in practice is our next topic.

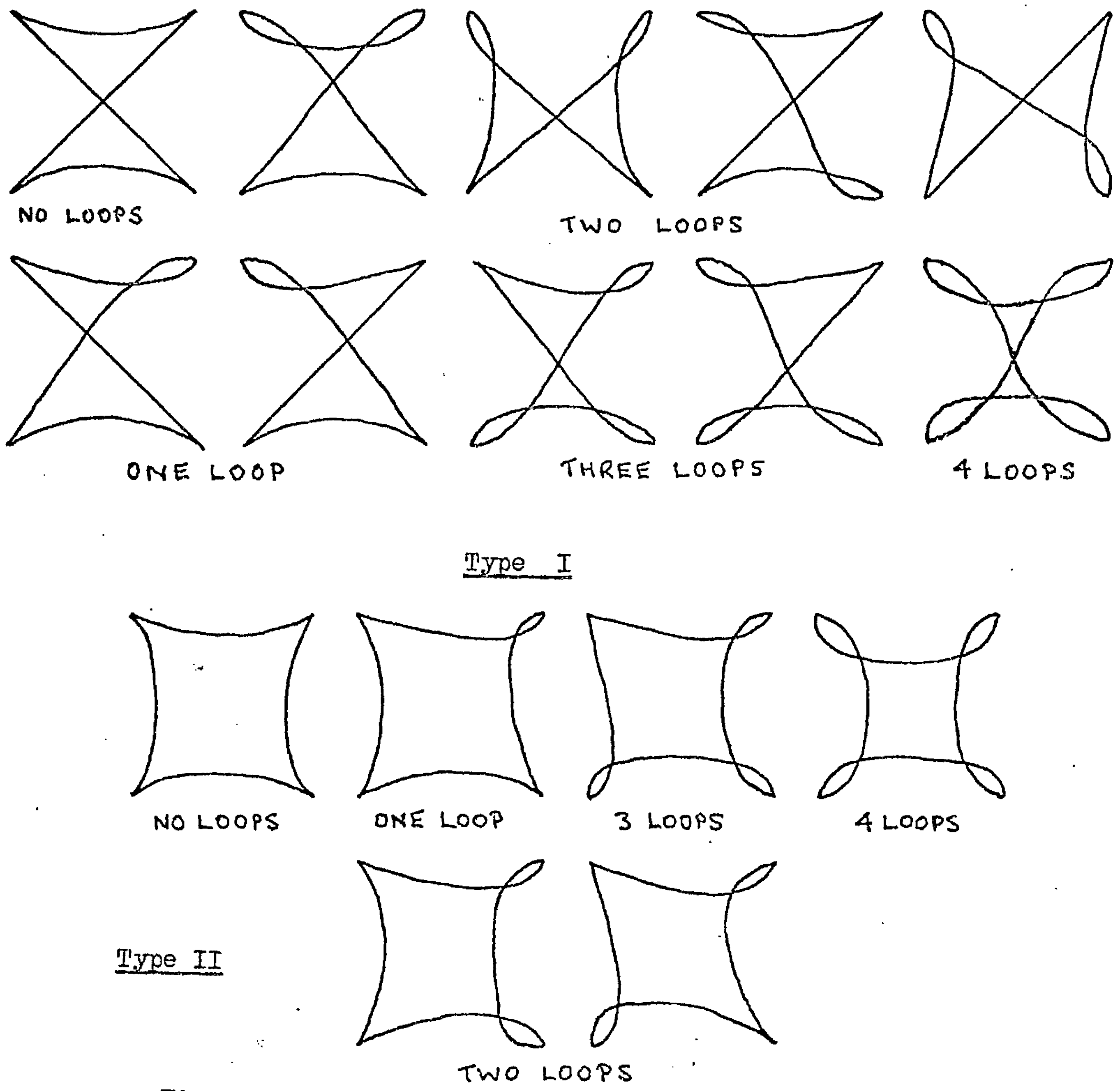


Figure 4.3 Subclassification by Loops

4.3 Symmetry of Knight's Tours The symmetry properties we are going to discuss are a property of the knight's tour's relationship with the chessboard, and, as is usual with this kind of discussion, there are two viewpoints we may take. We shall imagine the tour to be a fixed framework of moves, the numbers of the squares associated with these moves being determined by the orientation of the chessboard relative to this move-framework. Alternatively, we could have specified the board as being fixed and the framework reorientable. When we refer in the succeeding paragraphs to equality ('= ') between strings, we are conceptually divorcing the patterns of moves from the board, i.e. dissociating them from any numbering system, and superimposing them one on another. Two strings indistinguishable under these circumstances satisfy the equivalence relation '= '. In the study of symmetries we are interested in 8 orientations of the board as follows:-

(O) its initial state

(α) rotation about an axis through one pair of diagonally opposite C-squares

(β) rotation about an axis at 90° to this in the plane of the board

(γ) rotation about an axis through the mid-points of one pair of opposite sides

(δ) rotation about an axis at 90° to (γ) in the plane of the board

(ε) rotation through 90° about an axis perpendicular to the board

(η) rotation through 180° about an axis perpendicular to the board

(θ) rotation through 270° about an axis perpendicular to the board.

This set of transformations of a knight's tour forms a group under the operation 'f' representing 'followed by'; the zero element is 0, and each transformation, apart from ε & θ, is its own inverse; the transformations 0, ε, η and θ form a subgroup.

Given a knight's tour at random, it may on application of the 8 transformations yield 8 distinct knight's tours; we say that such a tour is non-symmetric. Some tours yield only 4 distinct transformations and some 2: we call these 2-and 4-symmetric respectively. There now follow some results on the symmetry of knight's tours; we use the symbol * to denote the operation of reflection in a diagonal as in 2.2., and A, B, C, D are corner-to-corner strings. By reflection symmetry we mean any of the transformations $\alpha, \beta, \gamma, \delta$ produce the same effect as 0; by rotational symmetry we mean that one or more of $\varepsilon, \eta, \theta$ produces the same effect as 0. The board is assumed even-sided.

Lemma 4.1 No corner-to-corner string between adjacent corners of the board is symmetric about the mid-point of the side of the board joining the two C-squares.

Proof Consider, for example, corners 1 and 2 with x-coordinates 0 and 5. If a move is made from 1 with $\Delta x = \Delta x_1$, then the mirror image move from 2 must be made also with $\Delta x = -\Delta x_1$. The corner squares were originally 5 squares apart, and the separation of the two ends of the strings being constructed from the C-squares is now $5-2\Delta x_1$.

After n moves have been made from 1, and the corresponding reflected moves made from 2, the separation is

$$5 - 2\sum_{i=1}^{n} \Delta x_i ,$$

which is always odd. Since the separation is never zero, the strings can never meet - hence the result.

Lemma 4.2 Any string of l.k.m's joining adjacent C-squares has an odd number of moves, and any string joining diagonally opposite C-squares an even number of moves.

Proof Suppose ~~the converse of the first part to be true~~ that, in fact, opposite parity to the above holds. Let the number of moves in the string with

$\Delta x = +1$ be n_{+1} ; $\Delta y = +1$ be m_{+1}

$\Delta x = -1$ be n_{-1} ; $\Delta y = -1$ be m_{-1}

$\Delta x = +2$ be n_{+2} ; $\Delta y = +2$ be m_{+2}

$\Delta x = -2$ be n_{-2} ; $\Delta y = -2$ be m_{-2}

Consider the first part of the lemma:—

Let the total number of moves be 2p, and the board -size be $2q \times 2q$.

Then obviously

$$n_{+1} + n_{-1} + n_{+2} + n_{-2} = m_{+1} + m_{-1} + m_{+2} + m_{-2} = 2p \qquad (1)$$

Also we have $|\Delta x| = 2q - 1$, $|\Delta y| = 0$ (or vice versa) for an adjacent C-string.

Now

$$\left.\begin{aligned} |\Delta y| = 0 &= m_{+1} - m_{-1} + 2m_{+2} - 2m_{-2} \\ |\Delta x| = 2q - 1 &= n_{+1} - n_{-1} + 2n_{+2} - 2n_{-2} \end{aligned}\right\} - (2)$$

and substituting into these equations from (1), we get

$$2m_{+1} + 3m_{+2} - m_{-2} = 2p$$

$$2n_{+1} + 3n_{+2} - n_{-2} = 2(p+q) - 1$$

From these results we deduce that

$3m_{+2} - m_{-2}$ is even, $\therefore$ m_{+2}, m_{-2} have the same parity.

Similarly, $3n_{+2} - n_{-2}$ is odd, $\therefore$ n_{+2} and n_{-2} have opposite parity.

But $n_{+1} + n_{-1} = m_{+2} + m_{-2}$

$n_{+2} + n_{-2} = m_{+1} + m_{-1}$

So n_{+1} and n_{-1} must have the same parity, and m_{+1} and m_{-1} must have opposite parity. Therefore

$n_{+1} + n_{-1} + n_{+2} + n_{-2}$ consists of 3 odds + 1 even or 3 evens + 1 odd, either of which gives an odd result, in contradiction with (1); similarly for the m's.

By an analogous argument, we can show that the supposition of the converse of the second part of the lemma leads to a contradiction also so the lemma is proved.

Theorem 4.3 There are no 8-symmetric knight's tours.

Proof Clearly Type I are inherently not completely symmetric: an 8-symmetric knight's tour would have to be Type II, consisting of 4 identical adjacent C-strings to satisfy rotational symmetry; the C-strings would also need to be symmetrical about the mid-point to comply with reflection symmetry requirements.(i.e. $A=B=C=D$; $A^*=A$ in Figure 4.2). But we have shown this to be impossible in Lemma 4.1. Hence there are no 8-symmetric knight's tours.

Theorem 4.4 No Type I knight's tour has rotational symmetry.

Proof Suppose a Type I tour does have rotational symmetry, then it is plain from Figure 4.2 that this must be a 2-fold symmetry with $A=C$, $B=D$. Therefore $A+B$ must be 18 moves, yet Lemma 4.2 shows that $A+B$ is inescapably odd, so that even 2-fold rotational symmetry of Type I tours is impossible.

Theorem 4.5 No Type I knight's tour has reflection symmetry.

i.e. All Type I tours are non-symmetric.

Lemma 4.1 shows that no symmetrics can arise out of the transformations γ and δ above. Consideration of Figure 4.2 also shows that α and β applied to the general Type II tour produces distinct knight's tours unless we have either A=B*, C=D* or A=D*, B=C*.

Theorem 4.6 No Type II knight's tour has reflection symmetry.

Proof In the foregoing remarks we have shown the sufficiency of proving the tour in Figure 4.4 cannot exist. Consider the board shown in 4.5., and let us try to construct the knight's tour shown in the second diagram of Figure 4.4. Now the squares X and Y must be used once and only once each. If X is in string A and is reached in x moves, and corner 2 is reached in n moves, then X* must be reached in n+(n-x)=2n-x moves; but X*=X, so the square X is used twice. Whether X or Y is used in A or B it must also occur in A* or B*, or vice versa, so these two squares are either used twice or not at all, and thus this sequence of strings is not a knight's tour. An identical argument exists for the left diagram of Figure 4.4, relating to the use of squares W and Z. Hence we have the desired result.

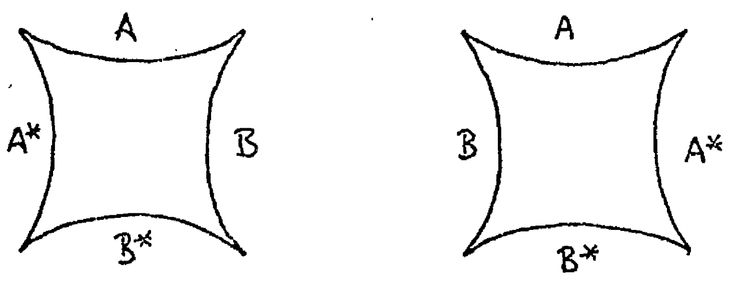


Figure 4.4 Hypothetical Reflection-Symmetric Type II Tour

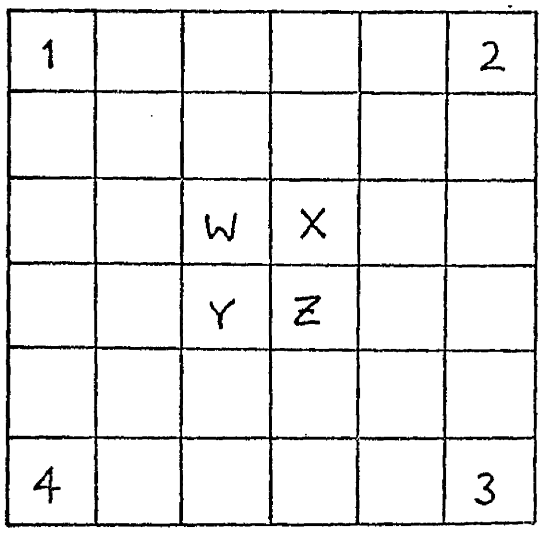


Figure 4.5

Note that we have also eliminated the particular case of Figure 4.4 where $A = B$. Thus we conclude from the above theorems that only Type II tours show any symmetry, and this is at most 4-fold symmetry of the rotational type.

These theoretical considerations lead us to expect that the total number of Type I tours, n_I, will be divisible by 8 and the number of Type II's, n_{II}, divisible (only possibly) by 2. In fact the former number is only divisible by 4, since our earlier freedom of orientation leads us here into ambiguity: the transformations $0, \alpha, \ldots, \theta$ generate from each knight's tour 8 tours no one of which may be generated from any other tour (unless it be one of the other 7) since the transformations form a group, and consequently we should be able to choose $\frac{1}{8}n_I$ tours which will generate all the n_I Type I tours (it is clear that Type I tours generate Type I, and Type II generate Type II, no cross-generation being feasible). We will now prove, however, that the transformations described always generate 4 clockwise and 4 anti-clockwise tours. Since we have denied the separate identity of tours starting $1 \longrightarrow 14$, it is apparent that $\frac{1}{4}n_I$ tours must be chosen from which to generate all Type I tours.

<u>Theorem 4.7</u> The eight transformations on a knight's tour always produce 4 clockwise and 4 anti-clockwise tours.

<u>Proof</u> Figure 4.6 shows the results of applying the operation 'f' to pairs of transformations. From this we see that if, for the purposes of the argument, we term 0 a zero rotation, then any reflection transformation can be accomplished by a rotation transformation followed by the specific reflection α. We have

$$\begin{aligned} \alpha &= 0 \, f \, \alpha \\ \beta &= \eta \, f \, \alpha \\ \gamma &= \theta \, f \, \alpha \\ \& \quad \delta &= \varepsilon \, f \, \alpha \end{aligned}$$

The effect of α is to reverse the sense of the tour, so that if the untransformed tour 0 began $1 \longrightarrow 9$, α is a tour beginning $1 \longrightarrow 14$ (and thus unacceptable). Whatever the orientations of the tours obtained by the single transformations ε, η, & θ, those got by the double transformations δ, β, γ respectively will be opposite. So there are equal numbers of clockwise and anti-clockwise circuits: 4 of each.

f	0	α	β	γ	δ	ε	η	θ
0	0	α	β	γ	δ	ε	η	θ
α	α	0	η	ε	θ	γ	β	δ
β	β	η	0	θ	ε	δ	α	γ
γ	γ	δ	ε	0	η	β	δ	α
δ	δ	ε	θ	η	0	α	β	γ
ε	ε	δ	γ	α	β	η	θ	0
η	η	β	α	δ	γ	θ	0	ε
θ	θ	γ	δ	β	α	0	ε	η

Table 4.6 Table of Effect of Successive Reflections and Rotations on knight's tours

$\frac{1}{4}n_I$ is thus an integer, and the number of Type I tours is divisible by 4. A minor alteration to the tree-searching routine of the basic generation 3 program was made to count Type I tours, this consisting of a check on the order of traversing C-squares; Type II tours always pass through exactly one C-square before reaching corner 3, whereas this square is always the 1st or 3rd. C-square on a Type I itinerary. The result obtained for the Type I and II counts (these two figures were achieved from two separate runs, not by subtracting either from 9862) were :-

Type I	3752	tours
Type II	6110	tours

We see our supposition that n_{II} might only be divisible by 2 is in fact correct, but in the light of Theorem 4.7., it is worth reflecting that the reason for this is not the reaction of 4-fold symmetry on groups of 8 transformations. The fact that we do not admit 4 out of these 8 might cause us to revise our ideas and expect rather n_{II} to be odd, but we have also to bear in mind the essential asymmetry of each adjacent C-string in the 4-symmetric Type II tours. For these tours (shown in Figure 4.7) there is an isomorph comprising four-mirror-image strings so that 4-symmetric (and 2-symmetric) tours naturally occur in pairs and n_{II} is thus even.

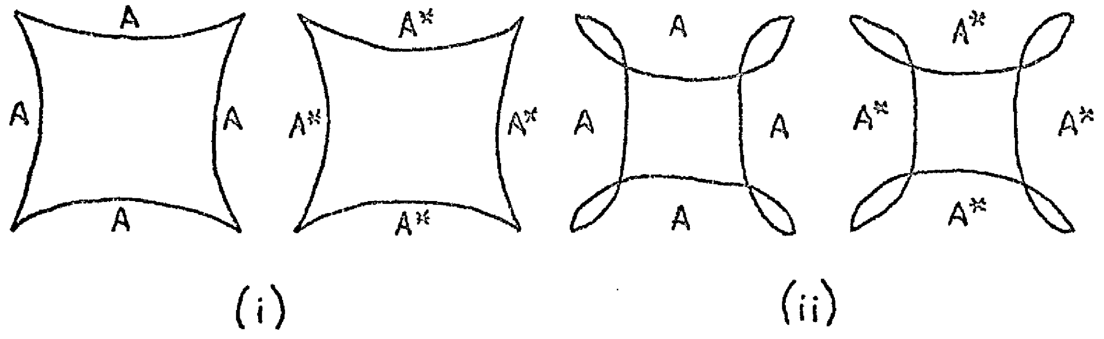


Figure 4.7 The 2 Structures of 4-Symmetric Type II Tours and their Isomorphs

Further refinement to the tree-searching routine enabled the actual number of 2- and 4-symmetric knight's tours to be determined, and representative examples of the various loop/cusp combinations permissible are displayed in Appendix 2.

Figure 4.8 shows a 2-symmetric arrangement which has 2 loops and no counterpart amongst the 4-symmetric tours. The search for symmetric tours was programmed particularly efficiently by making use of the 'skew-symmetry' (i.e. the 'equality' in the sense of 4.3 of the first and second strings to the third and fourth strings) inherent in them. A test was made after 18 moves to see if any square waspresent with its skew-image (in the centre of the board); if so, the string found was rejected and the level counter reduced by 1; if not, a complete symmetric tour could be formed by constructing a second half-tour of skew-images of the squares already used. The asymmetric board-numbering of Figure 2.5 made this easy to program, a rough translation of the UCA3 actually employed being:-

```
 for i: = 2 step 1 until 35 do begin
if board (i)= 0 or board (i)>19 then goto I else
if board (37-i) ≠ 0 then goto uptree else board (37-i):= 18+
                                                      board(i)
I:           ;
end;
printroute (board);, etc.
```

It was found that the total number of 2-symmetric Type II tours was 68, and the number of 4-symmetric tours was 10.

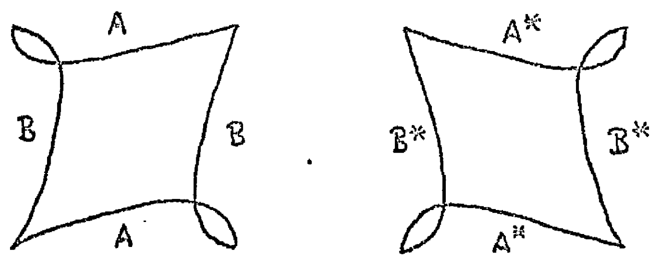


Figure 4.8 2-Loop 2-Symmetric Tour with Isomorph

4.4 Results Appertaining to Other Tree-Search Methods Back at the beginning of 4.2 we condemned the corner-to-corner tree-search with matching because the size of the matching (and formation of reflected strings) appeared too great. It is perhaps worthwhile to show how these corner-to-corner strings are distributed according to length. (Table 4.2) so that an idea of the number of combinations of 4-strings whose lengths total 36 moves may be formed; this exposition was deferred pending the proof of Lemma 4.2.

The number of half-tours could easily be found, also by a simple adjustment to the ordinary generation 3 tree-search routine calling only for curtailment after 18 moves rather than 36 moves. A trial run showed that the frequency of half-tours (unlike that of C-strings) was greater than whole tours, so that the overall saving of time on the tree-search phase was unlikely to be much better than 25%. The remaining 5 minutes or so of machine time (see 3.9) would hardly be adequate to invert and match in excess of 10,000 half-tours, and we conclude that tree-segmentation of this type does not compare favourably with a straightforward tree-search through $4n^2$ (for a 2n x 2n board) stratification levels.

Adjacent C-strings		Diagonal C-strings	
Length	Number	Length	Number
1	0	2	0
3	2	4	0
5	8	6	13
7	40	8	44
9	120	10	107
11	256	12	188
13	474	14	289
15	624	16	388
17	516	18	380
19	324	20	218
21	220	22	46
23	140	24	2
25	60	26	0
27	12	28	0
29	0	30	0
31	0	32	0
33	0	34	0
35	0	36	0

Table 4.2 Variation of Frequency of C-strings with Length

4.5 Time-Segmentation of the One-Level Lookahead In 2.6.1. and 3.9 we introduced the idea of time-segmentation of the tree-search, and briefly discussed the amount of intermediate storage that might be needed. We now probe this approach in more detail, particularly with reference to the implementation effected by the author using the COTAN on-line system in conjunction with the Egdon operating system. The program was, in essence, a modified generation 3 program, developed itself using the on-line facility, and is included in Appendix 1.

4.5.1 The COTAN 2 On-Line System COTAN 2 is a multi-access on-line system for a KDF9 computer connected to a small PDP8 machine which acts as a multiplexor for a number of teletype terminals. The system provides the user with a set of commands which enable him to manipulate and edit files, and initiate standard Egdon jobs in the remote (short runs, high priority) or background (long runs, standard priority) streams. Facilities also exist within the Egdon system for writing areas of core to the on-line file section of the disc.

Programs and data for the on-line system are stored permanently on the disc, and the user may amend, compile and run any of his program files by means of the commands at his disposal. A file consists of two consecutive blocks in the block substitution area of the disc, the first of which contains system housekeeping information, and the second being the user's data: this may be accessed by using the appropriate command together with the block-name and 'key' of the file and facilities similarly exist for preserving altered files and deleting unwanted ones. Loading of a new file may be done either by direct input at the teletype console or (useful for large programs) by supplying the two blocks on cards for a disc update run. Once the file on the disc is loaded, it can be altered or merely executed as a program (after development) as often as required.

4.5.2. <u>Modifications to Generation 3 Programs</u> The underlying strategy used in the time-segmentation program was to split the tree-search into a number of (not necessarily equal) time-slices each of which would be compatible with program time-limits imposed by the operating requirements of the installation. For a 6 x 6 board, this segmentation was artificial since the entire tree-search could be completed within the specified time, but the program nevertheless showed the feasibility of extending the method to an 8 x 8 board. Initially, data were read in as for an ordinary run, but after the time elapsed had approached a preset time-limit to within a specific tolerance, all intermediate information, i.e. connection matrix stack, solution vectors, branch counters, was written into an on-line disc file especially created for the purpose. Other information to be retained in a separate file were the knight's tours and run-time running totals so that these counts could be reinitiated on resumption of the tree-search. In order to restart after 'freeze', one item in the DATA section of the program file (the value of the variable MODE) had to be altered, using a suitable on-line command, before injecting the job once more into the background stream; this change caused the action of the program to be to obtain its data from the two disc-files rather than from the DATA section, after which it continued with the tree-search at the point (in time) of 'freeze'. The time-check was carried out just after printing each solution to standardise the procedure of resumption by ensuring that the level in the tree-search would always be 36 on such an occasion. Time-checking was again used in the continuation segment to determine whether the (new) intermediate data would need writing back to disc before another 'freeze'. If so, another continuation segment could later carry on the tree-search; otherwise the tree-search was terminated in the ordinary way after which the normal end-of-job condition of a standard generation 3 program ensued. Any number of continuation segments might be needed, but only the first one necessitated changing MODE; Figure 4.8 shows how time-segmentation requires two modes: (i) initial, where data is stored in the program file, and (ii) continuation (including terminal), where data is stored in another on-line file.

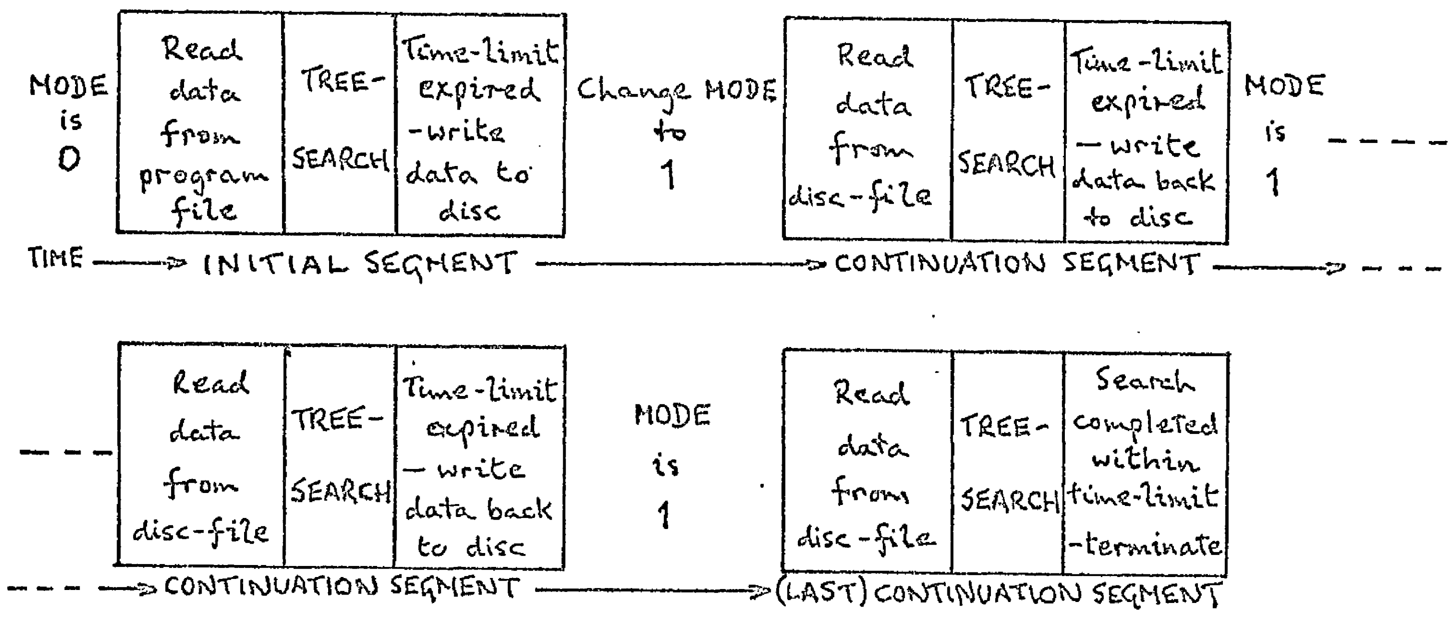

MODE is 0
Read data from program file
TREE-SEARCH
Time-limit expired -write data to disc
Change MODE to 1
Read data from disc-file
TREE-SEARCH
Time-limit expired -write data back to disc
MODE is 1
TIME → INITIAL SEGMENT → CONTINUATION SEGMENT →
Read data from disc-file
TREE-SEARCH
Time-limit expired -write data back to disc
MODE is 1
Read data from disc-file
TREE-SEARCH
Search completed within time-limit -terminate
→ CONTINUATION SEGMENT → (LAST) CONTINUATION SEGMENT

Figure 4.8 Tree-Search Time-Segmentation

The purpose of MODE having been described, the next new variable to be introduced is RUNRUN which is the running-total of the program run-time. This was saved between segments, and on resumption was added to the current segment run-time to give the 'time so far' printed out with each knight's tour. When a continuation segment tree-search was halted prior to writing information to a disc-file, the value of RUNRUN was updated by adding to its current value the total run-time for the present segment. Finally, TIMELT is the elapsed-time time-limit which is also supplied to the system through a COTAN command at run-time; the program must conform to this time-limit and write data to disc before the system throws it off - hence the need for the program to have a copy of the time-limit.

KTTOT, RUNRUN, and MODE are all public variables since other FORTRAN routines use them; TIMELT was declared as a common variable so that it should be accessible to the tree-search routine P2 via Z-addresses.

Minor improvements in the coding of P2 are discernible, but the chief alteration was in the layout of the arrays ORGD, BOARD, XD, KEEPP, and JJ. ORGD was omitted completely, and a suitably adjusted initial connection matrix read directly into level 1 of the XD stack. Instead of keeping XD in YA-stores and JJ in YB-stores, JJ, XD, BOARD and KEEPP were all declared as common arrays so that they would occupy a continuous series of blocks of Z-stores and might therefore be transferred to and from disc in a single transfer. At 'freeze', the repositories for various quantities were as follows:-

Arrays JJ, XD, BOARD, KEEPP	in	File DATA02/DS
Variables KTTOT, RUNRUN	in	File DATA03/DS

The fact that TIMELT was not preserved between segments means that a variable time-limit could be applied.

The new routine P3 contains the read from/write to disc-file instructions. The job 'opens' the on-line file by making an exit to the Director program (OUT 130) which returns to the program the size of the file and the address of its first sector.

The routine then sets up transfers in 40-word sectors (to avoid watching for the end of a track) and lets Director initiate these transfers. The main (direct) entry is for reading from DATA02/DS and DATA03/DS into core: the side entry is for writing from core out to the disc-files.

The interface between P2 and P3 is organised as shown in Figure 4.9. Note that P3 is always side-entered from P2: reading into core from disc is only initiated by main routine which explains the FORTRAN entry-name RESUME of P3; note also that P2 is always side-entered from P3: the main entry to P2 is only used in an initial segment when P1 calls P2.

In the event of only one segment being necessary, termination is via P1 and main routine which calls EXIT in the normal fashion. Termination of a continuation segment is ordinarily by OUT 100 in P3 (normal UCA3 end-of-job command), but if the continuation segment in question happens to be the final one, return is made from P2 to P3 and thence to main routine for library subroutine EXIT to terminate the job.

In view of the comparatively slow output routine PRINKT, it was thought that better results might be obtained by incorporating the peripheral transfer instructions explicitly within P2. The resulting block of coding was some 30 48-bit words long (due to producing output in format) and was substituted in place of the JSF1 instruction.

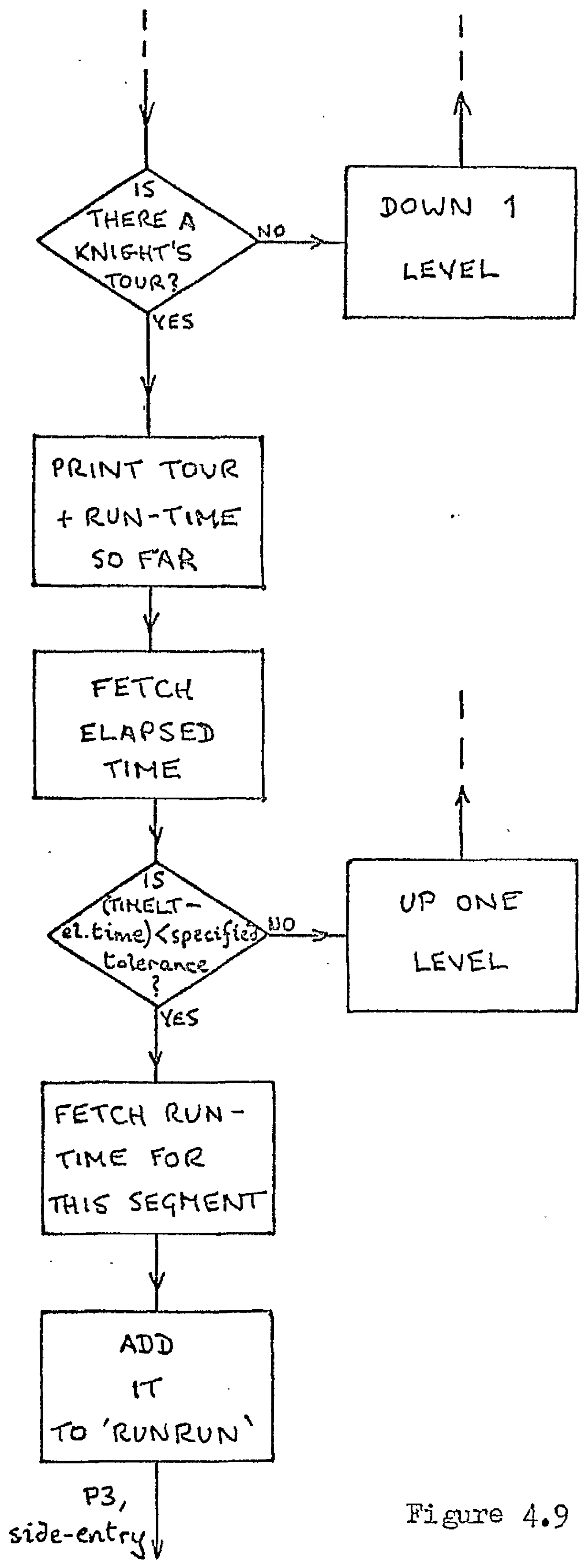

IS THERE A KNIGHT'S TOUR?
NO
DOWN 1 LEVEL
YES
PRINT TOUR + RUN-TIME SO FAR
FETCH ELAPSED TIME
IS (TIMELT - el.time) < specified tolerance ?
NO
UP ONE LEVEL
YES
FETCH RUN-TIME FOR THIS SEGMENT
ADD IT TO 'RUNRUN'
P3, side-entry

Figure 4.9 <u>Testing Prior to writing data to disc</u>

Two 24-word areas of W-stores were used for the double-buffered output, and 36 YA-stores for intermediate storage of the elements of BOARD after conversion to character form. The use of double-buffered output produces speeds of printing in excess of 5 knight's tours per second.

In this chapter, we have discussed the use of modified and extended versions of the basic 3rd-generation tree-searching program for assessing some of the other tree-search methods of Chapter 2 and also how the tree-search with one-level lookahead may be 'frozen' to permit time-segmentation and hence make searches too large for a single run a viable proposition. The major topic of the chapter was the application of the knight's tour enumerator to a novel classification of the set of tours; a number of results about the symmetry of tours in connection with this classification were proved.

CHAPTER 5

Conclusion

5.1 Appraisal of Methods In a number of instances, the methods described in Chapter 2 were judged on their merits at the time, while evaluation of other techniques received mention in Chapters 3 and 4. Our purpose in the present section is to correlate all the previous findings, and try to conclude from them the best avenues into which future research might be directed.

The one-level lookahead tree-search procedure developed in Chapter 3 is clearly at the limit of its applicability with a 6 x 6 board and even the time-segmentation of Chapter 4 is really only avoiding the issue: for an 8 x 8 board an exhaustive tree-search is impracticable unless better pruning criteria are to hand. The size of the solution set for the 6 x 6 board makes the discovery of properties of tours which might be used as criteria (sine qua non) a difficult one. The symmetry properties discussed in 4.3 give us a classification of a kind, but this does not redeem the method of 2.5 from the extremely unwieldly matching problem, and the matching of half-tours does not inspire much optimism in this as a viable method either. Dynamic programming suffers from the lack of a deterministic weighting function to resolve the optimal policy problem (2.9) and provision of such a function would seem to imply again a greater understanding of knight's tours' properties. This general dearth of properties is the obstacle which must be surmounted before any of the above techniques can be considered applicable other than to restricted problems. Other questions which are also relevant here are: does the presence of knight's tours on a board of dimensions 2n x 2n imply tours on boards 2(n+i) x (2n+j), where i, j are positive integers? And does there in fact exist some formula whence the number of tours on a board of given dimensions may be calculated?

Two methods, however, have been described which require little or no further knowledge of tours, and which might prove reasonably successful: the methods of heuristic transformations and edge-removal.

5.2 <u>Future Development of Techniques</u> The method of 2.6 was only implemented for one reversal-finding rule - in fact a particularly simple one. There are obvious possibilities for extensions directed towards escape from cyclic situations (which would presumably necessitate larger 'memories' then the four-tour FIFO store experimented with) by maintaining a list of occurrences of choices of transformations. We may surmise naively that if a cycle of tours is found, new tours may be generated by returning to the last tour for which a choice of transformations existed (obtained from the list) and making a different choice.

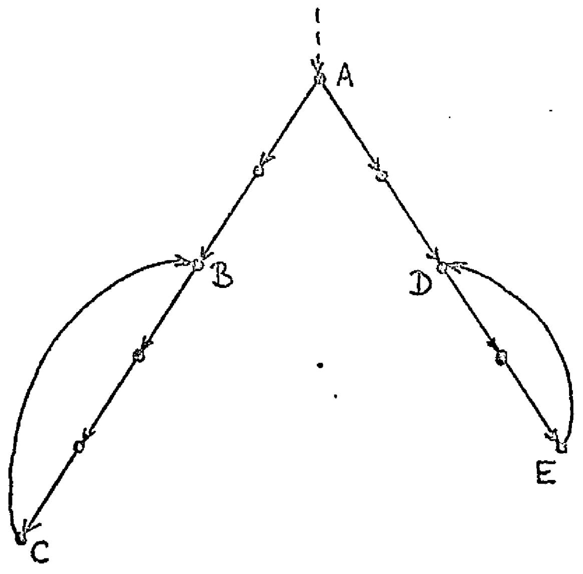


Figure 5.1 Cycles of Heuristic Transformations

Figure 5.1 shows a choice being made at A and a cycle of tours B⟶C⟶B; on noting the cycle, the choice list is invoked, and the right-hand branch from A is tried (A is now removed from the list as all paths from it have been tested). This too leads to a cycle D⟶E⟶D, so the choice list must be consulted again to find the choice that occurred before A (i.e. higher up the tree).

Different ways of indexing elements of a tour inevitably basically change the transformation, and the stratagem of 2.6 of choosing the 'least significant' transformation is only one of a great number of alternatives: for instance, one might determine to select the shortest transformation (in the sense of transforming the shortest string of knight's tour elements) whatever position in the tour it occupied. Finally, as we pointed out, there is nothing to prevent the inclusion of third - and even higher-order reversals.

The edge-removal method of 2.7 is clearly worthy of further investigation since it drastically reduces tree-searching. The chief difficulty to be overcome is that of organising the very complex house-keeping attached to a method where one move generates tremendous alterations to the connection matrix as well as extending the chains of forced moves. A feasible approach might use the same basic tree-searching algorithm and employ the boolean connection matrix stack in bit-form described in 3.7. It is clear, however, that, since at anytime we wish to know not only if an edge is present or not but also if the corresponding move is forced, a purely binary representation of the matrix is an unsatisfactory information storage medium; a solution to this problem would be to maintain two stacks, one exactly as in 3.7, and the other containing merely details of the chains of forced moves. It would be necessary to store for each square in a chain its two neighbours in the chain; a square at the end of a chain would have one neighbour only - the space thus left might be used for a marker (-1, say) since end-of-chain squares should be quickly detectable to aid joining two chains. Thus the chain of squares 23, 36, 28 might be stored as :

row 23	-1, 36
row 28	-1, 36
row 36	23, 28

By keeping a dynamic stack of these chain records, we retain the advantage of this technique of being able to 'discard' a level of the stack if at any time a move is rejected, later overwriting it as occasion demands. The depth of the stack necessary is a matter for some conjecture, but this could be determined by experiment; 10 levels would probably be ample. When a move is made, cross-references must be made between the two stacks to effect the consequent updating. The type of cross-reference involved would consist of removing edges in the current connection matrix (on stack A, say) and checking to see if any vertex had newly become doubly-connected; the two squares joined to this vertex would then be referenced in stack B (the chain lists)'s top cell to determine if either (or both) were at the end of an existing chain. If so, stack B would have to be updated to include a new member of the chain and stack A altered to allow for edge-removals at the join of the two chains as shown diagrammatically in Figure 5.2.

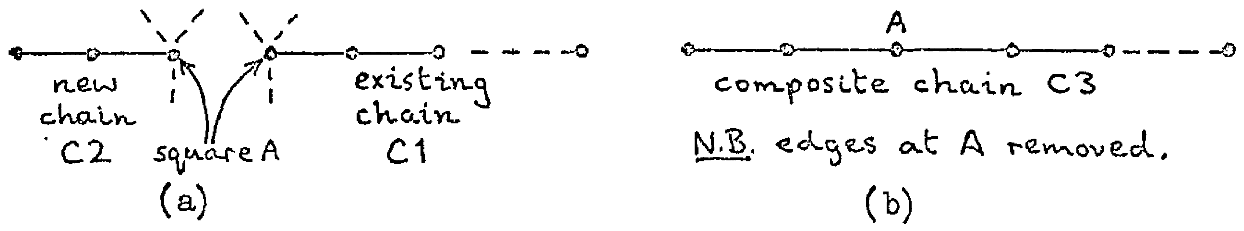


Figure 5.2 Coalescence of Chains

The difficulty of ordering the maintenance of the stacks might best be resolved by making distance (from the mode just moved to) the determining factor, matrix entries for the nearest nodes being carried out first, chain updates for these nodes following, then consequent edge-removals for nodes 2 edges distant from the move-node, then more chain-concatenation, and so on until every chain square (barring chain-ends) is doubly-connected and no doubly-connected square is not a chain-square.

If the updating procedure produces no inconsistencies (such as trying to chain squares to the middle of an existing chain) then the move is to be considered viable, and the tree-search progresses to the next lowest level, otherwise it goes up one level.

Once a solution had been found, it would be necessary to ensure that it was a knight's tour, and not merely a factor. The actual test employed would depend largely on the storage format of the solution, but would basically rest on ensuring that no square was used twice or not at all in the solution vector before printing it out.

APPENDIX 1

6 programs have been included as described below. The ALGOL programs are intended to specify completely, together with the description of Chapter 3, the method used and its development. The FORTRAN/UCA3 programs are included more for completeness' sake, as they are not annotated sufficiently to be easily understood.

Program	Type
DD174AM--	1st. generation(recursive) ALGOL.
DD174AS--	2nd. generation(iterative) ALGOL with graphic output.
DD174AV--	3rd. generation(iterative) ALGOL with 3-D condensed connection matrix stack.
COMMONSTOURS	1st. generation(recursive) FORTRAN/UCA3.
NONRECURSKT	3rd. generation FORTRAN/UCA3.
SEGSEARCH	Time-segmented 3rd. generation.

```
DD174AM00KP4+30100951RST;
begin libraryA0,A6,A12;
integer f,f1,f2,h,i,x,ccount,reccount;real klokin;
integer array XX[1:8,1:36],storepath,origdeg[0:35],newdeg[0:35,
1:36];
procedure nextstep;
begin integer a,j,k,d;
d:=storepath[reccount];
forj:=ifreccount=1then2else1step1until18do
begin ifXX[j,d]≠0thena:=XX[j,d]else gotoH;
ifnewdeg[reccount,a]=1then
begin ifreccount<34andXX[1,a]=1then gotoN;
fork:=j+1step1until18do
ifXX[k,d]=0then gotoGelse ifnewdeg[reccount,XX[k,d]]=1then goto
G: write text(70,[GOTO*SQUARE]);
write(70,f1,a);
stepcount(a);
newline(70,1);
gotoN;
end;
end;
H: forj:=1step1until18do
begin a:=XX[j,d];
write text(70,[TEST*SQUARE]);
write(70,f1,a);
ifa≠0then
begin fork:=0step1untilreccountdo ifstorepath[k]=athen gotoL;
ifreccount<34andXX[1,a]=1then
begin write text(70,[FORCED*LOOP]);
gotoL;
end;
stepcount(a);
end
else
begin newline(70,1);
gotoN;
end;
L: newline(70,1);
end;
N: storepath[reccount]:=0;
reccount:=reccount-1;
write text(70,[PRESENT*VALUE*OF*RECCOUNT*IS]);
write(70,f1,reccount);
newline(70,1);
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
end of procedure nextstep;
procedure stepcount(B);
integerB;
begin integer l,m,n;
```

```
reccount:=reccount+1;
write text(70,[[36s]STEP]);
write(70,f1,reccount);
newline(70,1);
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
m:=reccount-1;
forl:=1step1until36donewdeg[reccount,l]:=newdeg[m,l];
newdeg[reccount,B]:=0;
forl:=1step1until8do
begin m:=XX[l,B];
ifm≠0thenn:=newdeg[reccount,m]else gotoP;
ifn≠0thennewdeg[reccount,m]:=n-1;
P:
end;
write text(70,[STATE*OF*NEWDEG]);
form:=1step1until36dowrite(70,f,newdeg[reccount,m]);
newline(70,1);
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
storepath[reccount]:=B;
ifreccount<35then nextstepelse
begin printroute(storepath);
storepath[reccount]:=0;
reccount:=reccount-1;
end;
end of procedure stepcount;
real procedure time;
KDF9 3/0/0/0;
SET3;OUT;SET23;FLOAT;
EXIT;
ALGOL;
procedure printroute(A);
valueA;integer arrayA;
begin integer p,q;
p:=A[35];
write text(70,[KNIGHTS*TOUR[c]]);
ccount:=ccount+1;
forq:=0step1until34do
begin write(70,f,A[q]);
write text(70,[,]);
end;
write(70,f,p);
newline(70,1);
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
end of procedure printroute;
comment this program prints out all the knights tours on a 6×6
chessboard. The recursive tree-search is liberally sprinkled wit
time-checks to furnish some idea of the relative times spent on
different parts of the program;
open(20);open(70);
```

```
write text(70,[D*J*W*STONE[c]KNIGHTS*TOURS*ON*6*BY*6*BOARD[3c]]
forh:=1step1until36do
begin origdeg[h-1]:=x:=read(20);
storepath[h-1]:=0;
fori:=1step1untilxdoXX[i,h]:=read(20);
fori:=x+1step1until8doXX[i,h]:=0;
end;
ifread(20)≠999then gotoM;
comment this reads in the condensed connection matrix XX and the
vector containing the degrees of all the squares;
f1:=format([ndd]);f:=format([nd]);f2:=format([nddsddd.dddccc]);
klokin:=time;
storepath[0]:=reccount:=1;
newdeg[0,1]:=1;
fori:=2step1until36donewdeg[0,i]:=origdeg[i-1];
newdeg[0,12]:=5;
ccount:=0;
storepath[reccount]:=XX[1,1];
fori:=1step1until36do
begin ifi=XX[1,1]then newdeg[reccount,i]:=0;
else
begin ifi≠1then
forh:=1step1until8do
ifXX[h,i]=XX[1,1]then
begin newdeg[reccount,i]:=newdeg[0,i]-1;
gotoMM;
end;
newdeg[reccount,i]:=newdeg[0,i];
end;
MM:
end;
nextstep;
newline(70,1);
write text(70,[TOTAL*NO.*OF*KNIGHTS*TOURS*EQUALS**]);
write(70,f1,ccount);
gotoend;
M: write text(70,[INCORRECT*DATA]);
end: close(70);close(20);
end→
⇒
Data:-
2;12;31;
4;9;16;27;34;
8;7;9;14;19;24;29;34;36;
8;12;14;18;21;22;25;29;31;
4;16;18;25;27;
2;19;24;
3;3;13;35;
4;14;16;31;36;
4;2;3;15;17;
3;12;16;18;
```

```
2;13;17;
6;1;4;10;17;30;35;
6;7;11;18;19;27;31;
4;3;4;8;20;
3;9;16;19;
8;2;5;8;10;15;20;26;30;
6;9;11;12;21;24;27;
4;4;5;10;13;
6;3;6;13;15;23;26;
4;14;16;22;24;
3;4;17;23;
3;4;20;26;
4;19;21;27;29;
6;3;6;17;20;28;30;
4;4;5;30;33;
6;16;19;22;31;32;34;
8;2;5;13;17;23;28;33;35;
3;24;27;34;
4;3;4;23;35;
6;12;16;24;25;32;36;
6;1;4;8;13;26;33;
2;26;30;
3;25;27;31;
4;2;3;26;28;
4;7;12;27;29;
3;3;8;30;999;→
```

```
DD174AS00KP4+30100951RST→
begin libraryA0,A6,A12;
integer a,d,f,f1,f2,h,i,j,k,l,m,n,x,ccount,reccount;
integer array XX[1:8,1:36],storepath,origdeg[0:35],newdeg[0:35,
1:36],jj[1:34],board[1:36];
 realklokin;boolean array link[1:34];
real procedure time;
KDF9 3/0/0/0;
SET3;OUT;SET23;FLOAT;
EXIT;
ALGOL;
procedure printroute(A);
valueA;integer arrayA;
begin integer q;
write text(70,[KNIGHTS*TOUR[c]]);
ccount:=ccount+1;
forq:=1step1until36do
beginwrite(70,f1,A[q]);
ifq=q÷6×6thennewline(70,1);
end;
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
end of procedure printroute;
comment this program prints out all the knights tours on a 6×6
chessboard. The iterative tree-search is liberally sprinkled
with time-checks to furnish a comparison between this and a
 recursive method. Speeding up is further effected by use of the
array BOARD which depicts graphically which square is visited or
which move and facilitates checking whether a square has been
visited before;
open(20);open(70);
write text(70,[d*J*W*STONE[c]KNIGHTS*TOURS*ON*6*BY*6*BOARD[3c]])
forh:=1step1until36do
beginorigdeg[h-1]:=x:=read(20);
board[h]:=storepath[h-1]:=0;
fori:=1step1untilxdoXX[i,h]:=read(20);
fori:=x+1step1until8doXX[i,h]:=0;
end;
ifread(20)≠999then gotoM;
comment this reads in the condensed connection matrix XX and the
vector containing the degrees of all the squares;
close(20);
f:=format([nd]);f1:=format([ndd]);f2:=format([nddsddd.dddccc]);
klokin:=time;
storepath[0]:=board[1]:=reccount:=1;
newdeg[0,1]:=1;
fori:=2step1until36donewdeg[0,i]:=origdeg[i-1];
newdeg[0,9]:=5;
ccount:=0;
storepath[reccount]:=XX[1,1];
board[XX[1,1]]:=2;
```

```
fori:=1step1until36do
begin ifi=XX[1,1]then newdeg[reccount,i]:=0else
begin ifi≠1then
forh:=1step1until8do
ifXX[h,i]=XX[1,1]then
begin newdeg[reccount,i]:=newdeg[0,i]-1;
gotoMM;
end;
newdeg[reccount,i]:=newdeg[0,i];
end;
MM:
end;
NXST:d:=storepath[reccount];
forj:=ifreccount=1then2else1step1until8do
begin ifXX[j,d]≠0thena:=XX[j,d]else gotoH;
if newdeg[reccount,a]=1then
begin ifreccount<34andXX[1,a]=1then gotoN;
fork:=j+1step1until8do
ifXX[k,d]=0then gotoGelse ifnewdeg[reccount,XX[k,d]]=1then goto
G: write text(70,[GOTO*SQUARE]);
write(70,f1,a);
link[reccount]:=false;
gotoSTCT;
end;
end;
H: jj[reccount]:=1;
I: a:=XX[jj[reccount],d];
write text(70,[TEST*SQUARE]);
write(70,f1,a);
ifa≠0then
begin ifboard[a]≠0then gotoL;
ifreccount<34andXX[1,a]=1then
begin write text(70,[FORCED*LOOP]);
gotoL;
end;
link[reccount]:=true;
gotoSTCT;
end
else gotoN;
L: newline(70,1);
jj[reccount]:=jj[reccount]+1;
ifjj[reccount]≤8then gotoI;
N: newline(70,1);
storepath[reccount]:=board[d]:=0;
reccount:=reccount-1;
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
ifreccount=0then gotoEelse gotoF;
STCT: reccount:=reccount+1;
write text(70,[[36s]STEP]);
write(70,f1,reccount);
newline(70,1);
write text(70,[TIME*SO*FAR*]);
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
m:=reccount-1;
forl:=1step1until36donewdeg[reccount,l]:=newdeg[m,l];
newdeg[reccount,a]:=0;
forl:=1step1until8do
begin m:=XX[l,a];
ifm≠0then n:=newdeg[reccount,m]else gotoP;
ifn≠0thennewdeg[reccount,m]:=n-1;
 P:
end;
write text(70,[STATE*OF*NEWDEG]);
form:=1step1until36dowrite(70,f,newdeg[reccount,m]);
newline(70,1);
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
storepath[reccount]:=a;
board[a]:=reccount+1;
ifreccount<35then gotoNXSTelse
beginprintroute(board);
storepath[reccount]:=board[a]:=0;
reccount:=reccount-1;
end;
F: d:=storepath[reccount];
 iflink[reccount]then gotoL else gotoN;
E: newline(70,1);
write text(70,[TOTAL*NO.*OF*KNIGHTS*TOURS*EQUALS**]);
write(70,f1,ccount);
goto end;
M: write text(70,[INCORRECT*DATA]);
close(20);
end: close(70);
end→
→
Data:-
2;9;14;
3;10;13;15;
4;7;11;14;16;
4;8;12;15;17;
3;9;16;18;
2;10;17;
3;3;15;20;
4;4;16;19;21;
6;1;5;13;17;20;22;
6;2;6;14;18;21;23;
4;3;15;22;24;
3;4;16;23;
4;2;9;21;26;
6;1;3;10;22;25;27;
```

```
8;2;4;7;11;19;23;26;28;
8;3;5;8;12;20;24;27;29;
6;4;6;9;21;28;30;
4;5;10;22;29;
4;8;15;27;32;
6;7;9;16;28;31;33;
8;8;10;13;17;25;29;32;34;
8;9;11;14;18;26;30;33;35;
6;10;12;15;27;34;36;
4;11;16;28;35;
3;14;21;33;
4;13;15;22;34;
6;14;16;19;23;31;35;
6;15;17;20;24;32;36;
4;16;18;21;33;
3;17;22;34;
2;20;27;
3;19;21;28;
4;20;22;25;29;
4;21;23;26;30;
3;22;24;27;
2;23;28;999;->
```

```
DD174AV00KP5+30100951RST;
begin library A0,A6,A12;
integer a,b,bb,d,f,f1,f2,h,i,j,k,m,x,ccount,reccount;
integer arrayXX[1:8,1:36,0:35],storepath[0:35],jj[1:34];
real klokin;boolean gate;
real procedure time;
KDF9 3/0/0/0;
SET3;OUT;SET23;FLOAT;
EXIT;
ALGOL;
procedure printroute(A);
integer array A;
begin integer q;
write text(70,[KNIGHTS*TOUR[c]]);
ccount:=ccount+1;
forq:=1step1until36do
begin write(70,f1,A[1,q,35]);
ifq=q÷6x6then newline(70,2);
end;
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
end of procedure printroute;
comment this program prints out all the knights tours on a 6x6
 chessboard. The iterative tree-search is liberally sprinkled
with time-checks to furnish a comparison between this and a
recursive method. All record of the past and present state of
the board is maintained in a dynamic stack of connection
matrices, the 3-dimensional array XX. There is no need for
origdeg, newdeg, or board;
open(20);open(70);
write text(70,[D*J*W*STONE[c]KNIGHTS*TOURS*ON*6*BY*6*BOARD[3c]]);
forh:=1step1until36do
begin x:=read(20);
storepath[h-1]:=0;
fori:=1step1untilxdoXX[i,h,0]:=read(20);
fori:=x+1step1until8doXX[i,h,0]:=0;
end;
ifread(20)≠999then gotoM;
comment this reads in the condensed connection matrix XX[0] and
the vector containing the degrees of all the squares;
close(20);
f:=format([ndd]);f1:=format([nddd]);f2:=format([nddsddd.dddccc]);
klokin:=time;
storepath[0]:=1;
a:=XX[1,1,0];
fori:=1step1until7doXX[i,a,0]:=XX[i+1,a,0];
XX[8,a,0]:=0;
XX[1,1,0]:=1;
fori:=2step1until8doXX[i,1,0]:=0;
reccount:=ccount:=0;
storepath[reccount]:=a;
gotoSTCT;
```

```
NXST: d:=storepath[reccount];
gate:=true;
fori:=1,i+1whilei<9andXX[i,d,m]≠0do
ifXX[3,XX[i,d,m],m]≠0then
else if gatethen
begin a:=XX[i,d,m];
gate:=false;
end
else gotoN;
if notgatethen
begin ifreccount<34and a=14then
begin write text(70,[[s]FORCED*LOOP]);
gotoN;
end;
jj[reccount]:=0;
write text(70,[GOTO*SQUARE]);
write(70,f,a);
gotoX;
end;
jj[reccount]:=1;
I: a:=XX[jj[reccount],d,m];
write text(70,[TEST*SQUARE]);
write(70,f,a);
if a=0then gotoN;
ifreccount<34andXX[1,a,m]=1then
begin write text(70,[[s]FORCED*LOOP]);
gotoL;
end;
X: gotoSTCT;
L: newline(70,1);
jj[reccount]:=jj[reccount]+1;
ifjj[reccount]<8then gotoI;
N: newline(70,1);
storepath[reccount]:=0;
reccount:=reccount-1;
m:=reccount-1;
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
ifreccount=0then gotoEelse gotoF;
STCT: reccount:=reccount+1;
write text(70,[[36s]STEP]);
write(70,f,reccount);
newline(70,1);
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
m:=reccount-1;
k:=0;
fori:=1,i+1whilei<9andXX[i,a,m]≠0do
begin b:=XX[i,a,m];
V: k:=k+1;
```

```
if k=athen
begin forj:=2step1until8doXX[j,a,reccount]:=0;
XX[1,a,reccount]:=reccount+1;
gotoV;
end;
if k≠bthen
begin forj:=1step1until8doXX[j,k,reccount]:=XX[j,k,m];
gotoV;
end
else
begin forj:=1step1until7do
begin bb:=XX[j,b,m];
XX[j,b,reccount]:=ifbb<athenbbelseXX[j+1,b,m];
end;
XX[8,b,reccount]:=0;
end;
end;
if k<36then
fork:=k+1step1until36do
if k=athen
begin fori:=2step1until8do
XX[i,a,reccount]:=0;
XX[1,a,reccount]:=reccount+1;
end
else fori:=1step1until8doXX[i,k,reccount]:=XX[i,k,m];
write text(70,[STATE*OF*XX[c]]);
fori:=1step1until8do
begin forj:=1step1until36dowrite(70,f,XX[i,j,reccount]);
newline(70,1);
end;
write text(70,[TIME*SO*FAR*]);
write(70,f2,time-klokin);
storepath[reccount]:=a;
ifreccount<35then gotoNXSTelse
begin printroute(XX);
storepath[reccount]:=0;
reccount:=reccount-1;
end;
F: d:=storepath[reccount];
ifjj[reccount]=0then gotoNelse gotoL;
E: newline(70,1);
write text(70,[TOTAL*NO.*OF*KNIGHTS*TOURS*EQUALS**]);
write(70,f1,ccount);
gotoend;
M: write text(70,[INCORRECT*DATA]);
close(20);
end: close(70);
end→
→
Data:-  As for program DD174AS00......
```

```
*XEQ
*FRONTSHEET
      P COMMONSTOURS,
      YA1296,
      END,
*PRELUDE
*FORTRAN
      DIMENSION INTAR1(36),INTAR2(36,8),INTAR3(36)
      COMMON INTAR1,INTAR2,INTAR3
      PUBLIC KTTOT
      CALL BOCAF(I)
      INTAR1=I-36
      INTAR2=INTAR1-36*8
      INTAR3=INTAR2-36
      CALL BOCAS(INTAR3)
      CALL PREOUT(1)
      END
*CHAIN 1
*FORTRAN
      INTEGER XX,ORGD
      COMMON KEEPP,XX,ORGD
      PUBLIC KTTOT
      DIMENSION KEEPP(36),XX(36,8),ORGD(36)
      PRINT 3
    3 FORMAT(20HMAIN ROUTINE ENTERED)
      READ 1,(ORGD(I),I=1,36)
    1 FORMAT(36I2)
      DO 5 I=1,36
      KEEPP(I)=0
      J=ORGD(I)
      READ 2,(XX(I,K),K=1,J)
    2 FORMAT(30Z)
      J=J+1
      DO 4 K=J,8
    4 XX(I,K)=0
    5 CONTINUE
      READ 6,I
    6 FORMAT(I3)
      IF(I-999)10,7,10
    7 KTTOT=0
      CALL SETUP
      CALL RECURSIVE NEXTEP(1)
      GOTO 9
   10 PRINT 8
    8 FORMAT(14HDATA INCORRECT)
    9 CALL EXIT
      END
*FORTRAN
      RECURSIVE SUBROUTINE NEXTEP(KTREC)
      KA=KTREC
      CALL SQTEST(KA)
   11 IF(KA)10,8,9
    8 CALL RECURSIVE STEPCT(KTREC)
      KA=KTREC
      CALL LOOP(KA)
      GOTO 11
    9 CALL RECURSIVE STEPCT(KTREC)
      KA=KTREC
      CALL RUBST(KA)
      GOTO 11
   10 KTREC=KTREC-1
      RETURN
      END
*FORTRAN
      RECURSIVE SUBROUTINE STEPCT(KB)
      KB=KB+1
      KC=KB
      CALL UPDATE(KC)
      IF(KC)13,12,13
   12 CALL RECURSIVE NEXTEP(KB)
      GOTO 14
   13 KB=KB-1
   14 RETURN
      END
*FORTRAN
      SUBROUTINE PRINKT
      COMMON KD
      PUBLIC KTTOT
      DIMENSION KD(36)
      PRINT 1,(KD(I),I=1,36)
    1 FORMAT(1H0,36I3)
      KTTOT=KTTOT+1
      PRINT 2,KTTOT
    2 FORMAT(12HKNIGHTS TOUR,I5)
      RETURN
      END
*FORTRAN
      SUBROUTINE PRINTT(TIME)
      PRINT 1,TIME
    1 FORMAT(11HTIME SO FAR,F8.3)
      RETURN
      END
*USERCODE
C THIS IS AN ORGANISING ROUTINE
 ENTRYNAMES
      SETUP*
 ROUTINE
 P1,V3*
      Q1, =V1, Q2, =V2,
      SET 122, OUT, SET 23, FLOAT,
      =V3,(STORES STARTING TIME)
      JSP2, JSP3,
C SEE RESPECTIVE P-ROUTINES FOR THEIR PURPOSE
      V1, =Q1, V2, =Q2,
      EXIT 1,
      END,
*USERCODE
C THIS ROUTINE SETS UP THE FIRST ROW OF NEWDEG AND ESTABLISHES
C SQUARE 1 AS THE STARTING SQUARE
 ROUTINE
      P2,V3*
      V2=1, V3=Q36/36/AYA1,
      V2, Z1, =M1, =MOM1,(SETS STOREPATH(0) TO 1)
      V3, =Q1, Z3, =RM2,
 *,1, MOM2Q, =MOM1Q, *, J1C1NZS,
  3,  V2, =YA1, SET 5, =YA397,
C NEWDEG(0) SET UP
      EXIT 1,
      END,
*USERCODE
C A ROUTINE WHICH MAKES THE SPECIAL FIRST MOVE AND UPDATES
C NEWDEG AND STOREPATH ACCORDINGLY
 ROUTINE
      P3,V8*
      V2=36, V3=AYA1,
      V6=Q36/1/0, V7=Q8/36/0, V8=12,
      V8, Z1, =RM6, M+I6,
      =MOM6Q,
C FIRST MOVE TO SQUARE 12
      V6, =Q1,(OUTER LOOP COUNTER)
      V3, =RM3, Z2, =M6,
  1,  DC1, M+I1, V7, =Q2, M1, MOM6, J2NE,
C TEST IF I=XX(1,1)
      ERASE, ZERO, =YA398, M+I3, J7,
C NEWDEG(1,XX(1,1)) SET TO 0
  2,  V2P2, -, DUP, =M2,
      DUP, V2, MULTD, CONT, =M4, J5=Z,(TEST IF I IS 1)
 *,3, M6M2Q, V8, -, J4=Z, J3C2NZS,
      J5,
  4,  M4M3Q, V2P2, -, =M4M3,
C SETS NEWDEG(1,I)=NEWDEG(0,I)-1
      J7,
  5,  M4M3Q, =M4M3,
  7,  M-I3, J1C1NZ,
C NEWDEG(1) FORMED FROM NEWDEG(0)
      EXIT 1,
      END,
*USERCODE
C THIS ROUTINE PERFORMS THE LOOKAHEAD AND TESTS VIABLE SQUARES
C TO SEE IF THEY HAVE ALREADY BEEN VISITED
 ENTRYNAMES
      SQTEST,RUBST,LOOP*
 ROUTINE
      P4,V83,R61,R12*
      V4=1, V5=36, V6=B77, V7=8, V10=AYA0,
      Q1, Q2, =V2, =V1,
      DUP, =V9, =M3,
      MOM3, =V3,(KTREC STORED IN V3)
```

```
        Z1, =M8,
        V3, V4, -, J1=Z,
        V7, =RC3, J2,(SETS COUNTER FOR OUTER LOOP)
   1,   V7, =RC3, M+I3, DC3,(SPECIAL CASE)
   2,   V3, =M4, M8M4, =M4,(STOREPATH(KTREC) IN M4)
        Z2, V4, -, =M12, V10, =M11,
  21,   M3, V5, MULTD, CONT, M4, +, =M5,
        M12M5, DUP, =M5,(A IN M5), J8=Z,
        V5, M5, V4, DUP, V3, +,
        PERM, -, CAB, MULTD, CONT, +, =M6,
        M11M6, V4, J7NE, ERASE,
C TEST IF NEWDEG(KTREC,A) IS 1
        V3, SET 34, -, J3=Z,
        M12M5, V4, -, J14=Z,
   3,   V7, DUP, M3, V4, +, DUP, CAB, J5=,
        V5, XD, CONT, M4, +, =RM6,
        V5, =I6, -, =C1,(COUNTER SET FOR INNER LOOP)
   4,   M12M6Q, DUP, =V8, J6=Z,(XX(D,K) IN V8)
        V8, V4, DUP, PERM, -,
        V5, XD, CONT, V3, CAB, DUP,
        PERM, +, CAB, +, =M7,
        M11M7, -, J14=Z,
C TEST IF NEWDEG(KTREC,XX(D,K)) IS 1
        DC1, J4C1NZ(INNER LOOP), J6,
   5,   ERASE, ERASE, ERASE,
   6,   M5, =V81,(A IN V81)
        V1, =Q1, V2, =Q2,
        V9, =M9, V4, =MOM9, EXIT 1,
  61,   Q1, =V1, Q2, =V2,
        DUP, =V9, =M2, MOM2,
        =V3,(KTREC IN V3)
        J14,
C DO STEPCT THEN JUMP OUT OF LOOP
   7,   ERASE, DC3, M+I3, J21C3NZ,(OUTER LOOP)
   8,   V7, =RC3,(COUNTER SET FOR OUTER LOOP)
  10,   M3, V5, MULTD, CONT, M4, +, =M6,
        M12M6, DUP, =V81(A IN V81), J14=Z,
        V3, V4, +, =RC6,(COUNTER FOR SHORT INNER LOOP)
 *,9,   M8M6Q, V81, *, -, J13=Z, J9C6NZS,
C TEST IF STOREPATH(K) EQUALS A
        V3, SET 34, -, J11GEZ,
        V81, =M11, M12M11, V4, -, J13=Z,
  11,   V3, =M2, Q3, =V11M2(PRESERVES COUNT DURING STEPCT),
        M4, =V46M2, V1, =Q1, V2, =Q2,
        V9, =M9, ZERO, =MOM9, EXIT 1,
  12,   Q1, =V1, Q2, =V2,
        DUP, =V9, =M2, MOM2,
        =V3,(KTREC IN V3)
        V3, =M2, Z2, V4, -, =M12,
        V11M2, =Q3, V46M2, =M4, Z1, =M8,
  13,   DC3, M+I3, J10C3NZ,(OUTER LOOP)
  14,   V3, =M3, ZERO, Z1, =M8, =M8M3,
      V1, V2, =Q2, =Q1,
      V9, =M9, V4, NEG, =MOM9, EXIT 1,
      END,
*USERCODE
C THIS ROUTINE UPDATES NEWDEG ACCORDING TO MOVE FOUND BY SQTEST
 ENTRYNAMES
      UPDATE*
 FLIST PRINKT,PRINTT*
 ROUTINE
      P5,V5*
      Q1, =V1, Q2, =V2,
      DUP, =V4, =M1, MOM1, =V3,(KC IN V3)
      V10P4, V4P4, +, =M3,
      V5P4, DUP, =C4, =I4,
      Q4TOQ6, V3,
      DUP, NEG, NOT, =M4,
      =M6,
 *,1, M3M4Q, =M3M6Q, *,
      J1C4NZS,(SETS NEWDEG(KC,L)=NEWDEG(M,L))
      V5P4, V81P4, V4P4,
      -, XD, CONT, V3, +, =M4,
      ZERO, =M3M4,(SETS NEWDEG(KC,A)=0)
      Z2, V4P4, -, =M4, V81P4, =M6,
      V7P4, =C6, V5P4, =I6,
  2,  M4M6Q,
      DUP, J4=Z,(TEST IF M IS 0)
      V4P4, DUP, PERM, -, V5P4, XD, CONT,
      V3, +, =M1,
      M3M1, DUP, J3=Z,(TEST IF N IS 0)
      REV, -, =M3M1, J5,(UPDATES NEWDEG)
  3,  ERASE,
  4,  ERASE,
  5,  J2C6NZ,
      Z1, =M6,
      V81P4, V3, =M5, =M6M5,(A IN STOREPATH(KC))
      V3, SET 35, -, J6GEZ,
      V1, =Q1, V2, =Q2,
      V4, =M4, ZERO, =MOM4, EXIT 1,
  6,  V1, =Q1, V2, =Q2,
      LINK, =V0,
      JSF1,(ENTER PRINKT AND OUTPUT STOREPATH)
      SET 122, OUT, SET 23, FLOAT,
      V6P1, -F, =V2, SETAV2, JSF2,
C ENTER PRINTT AND OUTPUT THE TIME TAKEN
      V0, =LINK, Q1, Q2, =V2, =V1,
      V3, =M1, ZERO, Z1, =M2, =M2M1,
      V1, =Q1, V2, =Q2,
      V4, =M4, V4P4, =MOM4, EXIT 1,
      END,
*DATA
 2 4 8 8 4 2 3 4 4 3 2 6 6 4 3 6 6 4 6 4 3 3 4 6 4 6 8 3 4 6 6
 2 3 4 4 3
12 31
9 16 27 34
```

```
7  9 14 19 24 29 34 36
12 14 18 21 22 25 29 31
16 18 25 27
19 24
3 13 35
14 16 31 36
2  3 15 17
12 16 18
13 17
1  4 10 17 30 35
7 11 18 19 27 31
3  4  8 20
9 16 19
2  5  8 10 15 20 26 30
9 11 12 21 24 27
4  5 10 13
3  6 13 15 23 26
14 16 22 24
4 17 23
4 20 26
19 21 27 29
3  6 17 20 28 30
4  5 30 33
16 19 22 31 32 34
2  5 13 17 23 28 33 35
24 27 34
3  4 23 35
12 16 24 25 32 36
1  4  8 13 26 33
26 30
25 27 31
2  3 26 28
7 12 27 29
3  8 30
999
*ENDJOB
```

```
*XEQ
*FRONTSHEET
      P NONRECURSKT,
      YA1296, YB34,
      END,
*PRELUDE
*FORTRAN
      DIMENSION INTAR1(36),INTAR2(36),INTAR3(36)
      COMMON INTAR1,INTAR2,INTAR3
      PUBLIC KTTOT
      CALL BOCAF(I)
      INTAR1=I-36
      INTAR2=INTAR1-36
      INTAR3=INTAR2-36
      CALL BOCAS(INTAR3)
      CALL PREOUT(1)
      END
*CHAIN 1
*FORTRAN
      INTEGER ORGD,BOARD
      COMMON KEEPP,BOARD,ORGD
      PUBLIC KTTOT
      DIMENSION BOARD(36),ORGD(36),KEEPP(36)
      READ 2,(ORGD(I),I=1,36)
  2   FORMAT(B16.0)
      DO 3 I=1,36
      KEEPP(I)=0
  3   BOARD(I)=0
      READ 4,I
  4   FORMAT(I3)
      IF(I-999)6,5,6
  5   KTTOT=0
      CALL SETUP
      GOTO 8
  6   PRINT 7
  7   FORMAT(14HDATA INCORRECT)
  8   CALL EXIT
      END
*FORTRAN
      SUBROUTINE PRINKT(TIME)
      COMMON KA,KB
      PUBLIC KTTOT
      DIMENSION KA(36),KB(36)
      KTTOT=KTTOT+1
      PRINT 1,KTTOT
  1   FORMAT(13HOKNIGHTS TOUR,I5)
      PRINT 2,(KB(I),I=1,36)
  2   FORMAT(6I5//)
      PRINT 3,TIME
  3   FORMAT(11HTIME SO FAR,F8.3)
```

```
      RETURN
      END
*USERCODE
C THIS IS AN ORGANISING AND INITIALISING ROUTINE
 ENTRYNAMES
      SETUP*
 ROUTINE
      P1,V5*
      V3=Q36/36/AYA1, V5=Q1296/1/AYA1,
      Q1, =V1, Q2, =V2,
      SET 1, Z1, =M1, DUP, =MOM1,(SETS KEEPP(0) TO 1)
      Z2, =M1, =MOM1,(SETS BOARD(1) TO 1)
      V5, =Q1,
 *,1, ZERO, =MOM1Q, *, J1C1NZS,(ZEROS DYNAMIC XD)
      SET 122, OUT, SET 23, FLOAT,
      =V4,(STORES STARTING TIME)
      V3, =Q1, Z3, =RM2,
 *,2, MOM2Q, =MOM1Q, *, J2C1NZS,
      ZERO, DUP, =YA1, NOT, NEG,
      SHC-1, NOT, YA289, AND, =YA289,(XD(0) SET UP)
      SET 9, Z1, =RM1, M+I1,
      =MOM1,(FIRST MOVE TO SQUARE 9)
      V1, =Q1, V2, =Q2,
      JSP2,(UPDATES XD AND EFFECTS TREE-SEARCH)
      EXIT 1,
      END,
*USERCODE
C THIS ROUTINE CONDUCTS THE TREE-SEARCH FOR KNIGHTS TOURS
 FLIST PRINKT*
 ROUTINE
      P2,V7*
      V3=36, V5=Q36/36/0,
      Q1, =V1, Q2, =V2,
      Z1, =RM1, M+I1, MOM1,(UPDATE XD FOR MOVE TO 9)
      ZERO, =RM1,(COUNT IS 0), DUP, J14,
 100, Z1, =M2, M2M1,(D)
      DUP, =M7, NEG, NOT,
      V3, DUP, PERM, XD, CONT, =M5,(ROW MODIFIER FOR D)
      SETAYA0, M1, +, =M3,(LEVEL -1 BA)
      =C2,(SHC COUNTER)
      M3M5,
      ZERO, =YBOM1,(TEST-CELL FOR GOTO MOVE)
  1,  J2C2Z, ZERO, SHLD1, DC2, J1=Z,(TEST FOR NONZERO BITS)
      V3, DUP, C2, NOT, NEG, -, XD, CONT, =M4,
      M3M4N, BITS, NEG, NOT, J1NEZ,(TEST IF SQ.SINGLY CONNECTED)
      YBOM1, J7NEZ,
      C2, NOT, =YBOM1,(NOT INDICATES GOTO)
      J1C2NZ,(GOTO MOVE TESTING LOOP)
  2,  YBOM1, DUP, J3=Z,(NO GOTO)
```

```
        NOT, SET 22, -, J8NEZ,
        M1, SET 34, -, J7LTZ,(TEST FOR FORCED LOOP)
        J8,(JUMP TO UPDATE XD)
   3,   ERASE, V3, =C2,(BRANCH COUNTER FOR LEVEL), ERASE,
        M3M5,(SQUARES CONNECTED TO PRESENT POSN.)
   4,   J7C2Z, ZERO, SHLD1, DC2, J4=Z,(FIND NONZERO BITS AGAIN)
        C2, SET 22, -, J5NEZ,
        M1, SET 34, -, J5GEZ,(TEST FOR FORCED LOOP)
        J4,(REJECT IF PRESENT)
   5,   C2, =YBOM1,(NO NOT INDICATES TEST MOVE)
        J8,(JUMP TO UPDATE XD)
   7,   ERASE, ZERO, DUP, =M2M1,(ZEROS LAST EL.OF KEEPP)
        Z2, NEG, NOT, =M6,
        =M6M7,(ZEROS CORRESP.EL.OF BOARD)
        M-I1,(UP ONE LEVEL)
        M2M1, =M7,(REDEFINES D)
        M1, DUP, J11NEZ,
        J13=Z,
   8,   ERASE, V3, YBOM1, DUP,
        J9GTZ, NOT,
   9,   -,(CHOSEN SQUARE A), DUP,
  14,   =V7,
        M+I1,(DOWN ONE LEVEL)
        M1, =V6,
        DUP, NEG, NOT, DUPD,
        V3, XD, CONT, =M2,(ROW MODIFIER)
        CAB, NEG, =C2,(SHIFT CYCLIC MODIFIER)
        SETAYAO, V6, DUP, =RM1,(RESTORE LEVEL COUNTER)
        +, =M3,(STACK LEVEL BA)
        ZERO, NOT, NEG, SHCC2, NOT,(MASK)
        =Q4, V5, =Q5,
 *,10,M3M5, Q4, *, AND, =M3M5QN, J10C5NZS,(ZEROS COL.A OF XD)
        ZERO, =M3M2N,(ZEROS ROW A OF XD)
        Z1, =M2, =M2M1,(UPDATES KEEPP)
        Z2, =M6, =M7,
        M1, NOT, NEG, =M6M7,(UPDATES BOARD)
        M1, SET 35, -, J100LTZ,(JUMP TO CHOOSE NEXT SQUARE)
        ZERO, =M2M1,(ZEROS LAST EL.OF KEEPP)
        LINK, =V0, V1, =Q1, V2, =Q2,
        SET 122, OUT, SET 23, FLOAT,
        V4P1, -F, =V4, SETAV4,
        JSF1,(PRINT KT AND TIME TAKEN)
        Q1, =V1, Q2, =V2, V0, =LINK,
        Z2, =M2, V7, NEG, NOT, =M3,
        ZERO, =M2M3,(ZEROS CORRESP.EL.OF BOARD)
        V6, NEG, NOT, =RM1,(UP ONE LEVEL)
        Z1, =M2, M2M1, =M7, J12,
  11,   ERASE,
  12,   YBOM1, DUP,
        J7LTZ,(QUICK BACKTRACK FOR GOTO AT LEVEL -1)
```

```
       DUP, V3, DUP, CAB, -,
       =C2,(RESTORES COUNTER FOR TEST SQ.LOOP)
       M2M1, NEG, NOT, XD, CONT, =M5,
       SETAYAO, M1, +, =M3,
       M3M5, ZERO,
       SHLDC2,(RESTORES NONZERO BIT-SEARCH AT LEVEL -1)
       ERASE, REV, =C2, J4,(RESUME TEST SQ.LOOP AT LEVEL -1)
 13,   V1, =Q1, V2, =Q2, EXIT 1,
       END,
*DATA
0000200000000000
0004500000000000
0042240000000000
0021120000000000
0010050000000000
0004020000000000
1000102000000000
0400045000000000
0200422400000000
2100211200000000
1000100500000000
0400040200000000
2010001020000000
5004000450000000
2442004224000000
1221002112000000
0510001005000000
0204000402000000
0020100010200000
0050040004500000
0024420042240000
0012210021120000
0005100010050000
0002040004020000
0000201000100000
0000500400040000
0000244200420000
0000122100210000
0000051000100000
0000020400040000
0000002010000000
0000005004000000
0000002442000000
0000001221000000
0000000510000000
0000000204000000
999
*ENDJOB
```

```
*XEQ
*FRONTSHEET
      P SEGSEARCH,
      W159,
      END,
*PRELUDE
*FORTRAN
      DIMENSION INTAR1(36),INTAR2(36),INTAR3(36,36)
      DIMENSION INTAR4(34)
      COMMON INTAR1,INTAR2,INTAR3,INTAR4,TIMELT
      PUBLIC MODE,RUNRUN,KTTOT
      CALL BOCAF(I)
      INTAR1=I-36
      INTAR2=INTAR1-36
      INTAR3=INTAR2-1296
      INTAR4=INTAR3-34
      CALL BOCAS(INTAR4)
      READ 1,TIMELT,MODE
    1 FORMAT(F8.3/I1)
      CALL PREOUT(1)
      END
*CHAIN 1
*FORTRAN
      INTEGER BOARD,XD
      COMMON KEEPP,BOARD,XD,JJ
      PUBLIC MODE,RUNRUN,KTTOT
      DIMENSION BOARD(36),JJ(34),KEEPP(36)
      DIMENSION XD(36,36)
      IF(MODE.NE.0)9
      RUNRUN=0.0
      CALL ZERO
      READ 2,(XD(1,I),I=1,36)
    2 FORMAT(B16.0)
      DO 3 I=1,36
      KEEPP(I)=0
    3 BOARD(I)=0
      READ 4,I
    4 FORMAT(I3)
      IF(I-999)6,5,6
    5 KTTOT=0
      CALL SETUP
      GOTO 8
    6 PRINT 7
    7 FORMAT(14HDATA INCORRECT)
    8 CALL EXIT
    9 CALL RESUME
      GOTO 8
      END
```

```
*FORTRAN
      SUBROUTINE PRINKT(TIME)
      COMMON KA,KB
      PUBLIC KTTOT
      DIMENSION KA(36),KB(36)
      KTTOT=KTTOT+1
      PRINT 1,KTTOT
  1   FORMAT(13HOKNIGHTS TOUR,I5)
      PRINT 2,(KB(I),I=1,36)
  2   FORMAT(6I5//)
      PRINT 3,TIME
  3   FORMAT(11HTIME SO FAR,F8.3)
      RETURN
      END
*FORTRAN
      SUBROUTINE TRACE1
      PRINT 1
  1   FORMAT(10HP1 ENTERED)
      RETURN
      END
*FORTRAN
      SUBROUTINE TRACE2
      PRINT 1
  1   FORMAT(10HP2 ENTERED)
      RETURN
      END
*FORTRAN
      SUBROUTINE TRACE3
      PRINT 1
  1   FORMAT(10HP3 ENTERED)
      RETURN
      END
*FORTRAN
      SUBROUTINE TRACE4
      PRINT 1
  1   FORMAT(13HP3 SIDENTERED)
      RETURN
      END
*USERCODE
C THIS IS AN ORGANISING AND INITIALISING ROUTINE
 ENTRYNAMES
      SETUP,ZERO*
 FLIST TRACE1*
 ROUTINE
      P1,V4,R11*
      V3=Q36/36/AYA1,
      JSF1, Q1, =V1, Q2, =V2,
      SET 1, Z1, =M1, DUP, =MOM1,(SETS KEEPP(0) TO 1)
      Z2, =M1, =MOM1(SETS BOARD(1) TO 1), J2,
 11,  Q1, =V1, Q2, =V2,
      SET 1296, =RC1, Z3, =M1,
 *,1, ZERO, =MOM1Q, *, J1C1NZS,(ZEROS DYNAMIC XD)
      V1, =Q1, V2, =Q2, EXIT 1,
  2,  SET 122, OUT, SET 23, FLOAT,
      =V4,(STORES STARTING TIME)
      ZERO, Z3, =M2, =MOM2,
      SET 288, =+M2, SET-2,
      SHC-1, MOM2, AND, =MOM2,(XD(0) SET UP)
      SET 9, Z1, =RM1, M+I1,
      =MOM1,(FIRST MOVE TO SQUARE 9)
      V1, =Q1, V2, =Q2,
      JSP2,(UPDATES XD AND EFFECTS TREE-SEARCH)
      EXIT 1,
      END,
*USERCODE
C THIS ROUTINE CONDUCTS THE TREE-SEARCH FOR KNIGHTS TOURS
 FLIST PRINKT,TRACE2*
 ROUTINE
      P2,V9,R101*
      V3=36, V5=Q35/36/0, V8=F10.0,
      V6=Q1/AZ8/AZ8,
      JSF2, Q1, =V1, Q2, =V2,
      V6, SET 110, OUT,
      Z3, NEG, NOT, =V9,
      Z1, =RM1, MOM1N,(UPDATE XD FOR MOVE TO 9)
      MOTOQ1,(COUNT IS 0), J14,
 100, M2M1,(D)
      DUP, =M7, NEG, NOT,
      V3, DUP, PERM, XD, CONT, =M5,(ROW MODIFIER FOR D)
      M1TOQ3, V9, =+M3,(LEVEL -1 BA)
      =C2,(SHC COUNTER)
      M3M5, ZERO, =M8M1,(TEST-CELL FOR GOTO MOVE)
  1,  J2C2Z, ZERO, SHLD1, DC2, J1=Z,(TEST FOR NONZERO BITS)
      V3, DUP, C2, NOT, NEG, -, XD, CONT, =M4,
      M3M4N, DUP, DUP, NEG, NEV, AND,
      J1NEZ,(TEST IF SQ.SINGLY CONNECTED)
      M8M1, J7NEZ,
      C2, NOT, =M8M1,(NOT INDICATES GOTO)
      J1C2NZ,(GOTO SQUARE TESTING LOOP)
  2,  M8M1, DUP, J3=Z,(NO GOTO)
      NOT, SET 22, -, J8NEZ,
      M1, SET 34, -, J7LTZ,(TEST FOR FORCED LOOP)
      J8,(JUMP TO UPDATE XD)
  3,  ERASE, V3, =C2(BRANCH COUNTER FOR LEVEL), ERASE,
      M3M5,(SQUARES CONNECTED TO PRESENT POSN.)
  4,  J7C2Z, ZERO, SHLD1, DC2, J4=Z,(FIND NONZERO BITS AGAIN)
      C2, SET 22, -, J5NEZ,
      M1, SET 34, -, J5GEZ,(TEST FOR FORCED LOOP)
```

```
        J4,(REJECT IF PRESENT)
  5,    C2, =M8M1,(NO NOT INDICATES TEST MOVE)
        J8,(JUMP TO UPDATE XD)
  7,    ERASE, ZERO, DUP, =M2M1,(ZEROS LAST EL.OF KEEPP)
        Z2, NEG, NOT, =M6,
        =M6M7,(ZEROS CORRESP.EL.OF BOARD)
        M-I1,(UP ONE LEVEL)
        M2M1, =M7,(REDEFINES D)
        M1, DUP, J11NEZ,
        J13=Z,
  8,    ERASE, V3, M8M1, DUP,
        J9GTZ, NOT,
  9,    -,(CHOSEN SQUARE A)
 14,    DUP, =V7,
        M+I1(DOWN ONE LEVEL), M1, =V6,
        DUP, DUPD, NEG, NOT,
        V3, XD, CONT, =M2,(ROW MODIFIER)
        NEG, =C2,(SHIFT CYCLIC MODIFIER)
        V9, V6, +, =M3,
        V5, =Q5, Z4, NEG, NOT, =M8,
        SET-2, SHCC2,(MASK)
        DUP, =Q4, M3M5, AND,
*,10,=M3M5QN, M3M5, Q4, *, AND, J10C5NZS,
        =M3M5QN,(ZEROS COL.A OF XD)
        ZERO, =M3M2N,(ZEROS ROW A OF XD)
        Z1, =M2, =M2M1,(UPDATES KEEPP)
        Z2, NEG, NOT, =M6, =M7,
        M1, NOT, NEG, =M6M7,(UPDATES BOARD)
        M1, SET 35, -, J100LTZ,(JUMP TO CHOOSE NEXT SQUARE)
        ZERO, =M2M1,(ZEROS LAST EL.OF KEEPP)
        LINK, =V0, V1, =Q1, V2, =Q2,
        SET 122, OUT, SET 23, FLOAT,
        V4P1, -F, Z7, +F, =V4, SETAV4,
        JSF1,(PRINT KT AND TIME TAKEN)
        Q1, =V1, Q2, =V2, V0, =LINK,
        SET 128, OUT, SET 23, FLOAT,
        Z5, -F, V8, SIGNF, J102LTZ,(TEST IF TIME UP)
        SET 122, OUT, SET 23, FLOAT,
        V4P1, -F, Z7, +F, =Z7,(UPDATE RUNRUN)
        V1, =Q1, V2, =Q2, J1P3,(JUMP TO FREEZE ROUTINE)
101,    JSF2, Q1, =V1, Q2, =V2,
        SET 35, =M1, Z3, NEG, NOT, =V9,
        V6, SET 110, OUT,
        SET 13, =V7, M1, =V6,
102,    Z2, NEG, NOT, =M6, V7, =M3,
        ZERO, =M6M3,(ZEROS CORRESP.EL.OF BOARD)
        V6, NEG, NOT, =RM1,(UP ONE LEVEL)
        Z1, =M2, M2M1, =M7, Z4, NEG, NOT, =M8, J12,
 11,    ERASE,
 12,   M8M1, DUP,
       J7LTZ,(QUICK BACKTRACK FOR GOTO AT LEVEL -1)
       DUP, V3, DUP, CAB, -, =C2,(RESTORES COUNTER FOR TEST SQ.LOOP)
       M2M1, NEG, NOT, XD, CONT, =M5,
       M1TOQ3, V9, =+M3,
       M3M5, ZERO, SHLDC2,(RESTORES NONZERO BIT-SEARCH AT LEVEL -1)
       ERASE, REV, =C2, J4,(RESUME TEST SQ.LOOP AT LEVEL -1)
 13,   V1, =Q1, V2, =Q2, EXIT 1,
       END,
*USERCODE
C THIS ROUTINE INITIALISES THE ARRAYS FOR TREE-SEARCHING
C FROM FREEZE AND FREEZES THEM BEFORE RUN ENDS
 ENTRYNAMES
       RESUME*
 FLIST TRACE3,TRACE4*
 ROUTINE
       P3,V12,R1*
       V3=Q0/AW0/AW159, V5=PDS010095, V6=Q0/AZ8/AZ7,
       V4=PDSDATA02, V7=PDSDATA03,
       V8=PFILE TOO, V9=P SMALL  , V10=Q0/0/1,
       V11=Q0/AV8/AV9, V12=Q1/AW0/AW39,
       JSF1, SETAR4, =Z10, J2,
  1,   JSF2, SETAR5, =Z10,
  2,   Q1, =V1, Q2, =V2,
       Z1, SET 35, +,(TOP ADDR.TO BE TRANSFERRED)
       Z4,(BOTTOM ADDR.), DUP, DUP, =M1,
       PERM, -,(AREA OF CORE TO BE READ/WRITTEN)
       SET 40, DUP, DUP, =I1, =I2,
       DIVI, ZERO, SIGN, +,
       =C1,(SECTOR COUNT), SET 39, +,
       =M2,(TOP ADDR.FOR SINGLE TRANSFER)
       V3, V4, V5, SET 130, OUT,(OPEN FILE DATA02)
       C1, -, J98LTZ,(CHECK ON FILE-SIZE)
       =Z9, DC1, M-I1,(ADJUST TRANSFER COUNTER)
  3,   Z9, M0M1Q, ERASE, Z10, =LINK, EXIT,
  4,   M1, =RI3, J41C1NZ, SETAR8, =Z10,
 41,   J6,
  5,   SET 1, =C3, M1, =I3, J6C1NZ, SETAR9, =Z10,
  6,   M2TOQ3,(Q-STORE FOR DISC TRANSFER)
       Q3, SET 106, OUT,(TRANSFER 1 SECTOR)
       Z9, NOT, NEG, =Z9,(ADJUST SECTOR ADDRESS)
       M+I2,(ADDR.LIMITS FOR NEXT TRANSFER)
       J3C1NZ,(KEEPP,BOARD,XD,JJ TO/FROM DISC), M+I1,
       Z10, =LINK, EXIT,
  7,   SET 106, OUT,
       V3, V7, V5, SET 130, OUT,(OPEN FILE DATA03)
       ERASE, Z10, =LINK, EXIT,
  8,   M1, DUP, NOT, NEG, =M3, =I3,
       Z9, Q3, SETAR11, =Z10, J7,
 9,   SET 40, =RC2,
*,10,ZERO, =WOM2Q, *, J10C1NZS,
      MOM1 , =W0, MOM1N, =W1 , Z9, V12, SETAR12, =Z10,
      J7,(SPECIAL TRANSFER OF TWO EXTRA WORDS)
11,   V6, SET 106, OUT,(KTTOT,RUNRUN)
      SET 122, OUT, SET 23, FLOAT,
      =V4P1,(STORES STARTING TIME)
      V1, =Q1, V2, =Q2, JS101P2,
      SET 112, OUT,
      EXIT 1,
12,   Z8, =W0, Z7, =W1,(KTTOT,RUNRUN)
      V12, SET 106, OUT, J99,
98,   V10, V11, SET 104, OUT,
99,   SET 112, OUT,
      END,
*DATA
 600,000
0                      [or 1]
[rest of data as for program NONRECURSKT]
*ENDJOB
```

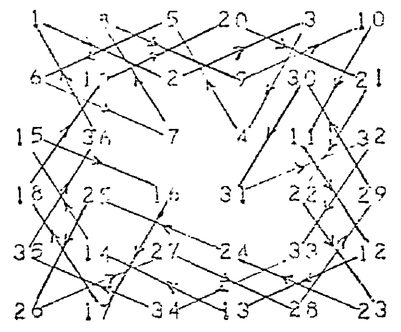
APPENDIX 2

The first piece of output is a sample from the program DD174AM-- of Appendix 1; it shows how the diagnostic output is used to check the validity of the path chosen through the tree and also compare the time taken in various parts of the program. The second and third sheets are from DD174AV--, the former showing the updating of the connection matrix stack XX(see p.85), and the latter the graphic output incorporated from 2nd. generation onwards. Note the 'automatic' choice of an unused square at each step(cf. the 1st. output page) produced by updating the connection matrix rather than the degree vectors. The 4th. page shows graphic output of a 3rd. generation FORTRAN/UCA3 program to print out all the 2- and 4-symmetric tours on a 6x6 board(see Chapter 4). Diagnostics have been omitted and only the tours themselves shown.

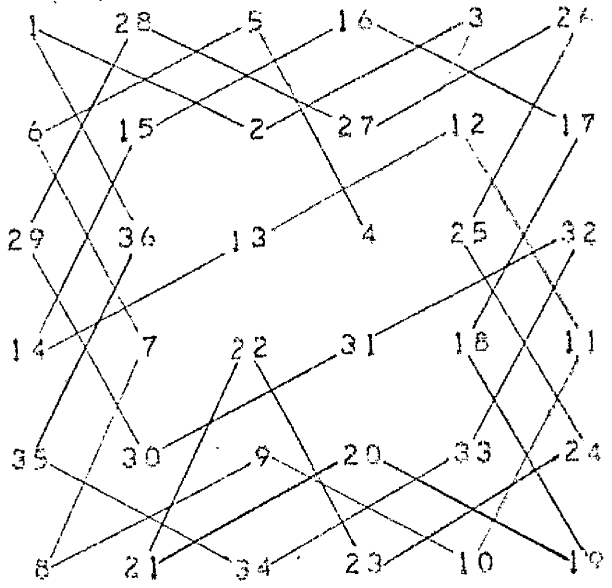
```
 J W STONE
NIGHTS TOURS ON 6 BY 6 BOARD


EST SQUARE  1
EST SQUARE  4                                                   STEP  2
IME SO FAR          0.252


TATE OF NEWDEG 1 4 8 0 4 2 3 4 4 2 2 0 6 3 3 8 5 3 6 4 2 2 4 6 3 6 8
IME SO FAR          0.535


EST SQUARE 12
EST SQUARE 14                                                   STEP  3
IME SO FAR          0.651


TATE OF NEWDEG 1 4 7 0 4 2 3 3 4 2 2 0 6 0 3 8 5 3 6 3 2 2 4 6 3 6 8
IME SO FAR          0.933


EST SQUARE  3                                                   STEP  4
IME SO FAR          1.026


TATE OF NEWDEG 1 4 0 0 4 2 2 3 3 2 2 0 6 0 3 8 5 3 5 3 2 2 4 5 3 6 8
IME SO FAR          1.308


EST SQUARE  7                                                   STEP  5
IME SO FAR          1.404


TATE OF NEWDEG 1 4 0 0 4 2 0 3 3 2 2 0 5 0 3 8 5 3 5 3 2 2 4 5 3 6 8
IME SO FAR          1.684


EST SQUARE  3
EST SQUARE 13                                                   STEP  6
IME SO FAR          1.799


TATE OF NEWDEG 1 4 0 0 4 2 0 3 3 2 1 0 0 0 3 8 5 2 4 3 2 2 4 5 3 6 7
IME SO FAR          2.079


OTO SQUARE 11                                                   STEP  7
IME SO FAR          2.172


TATE OF NEWDEG 1 4 0 0 4 2 0 3 3 2 0 0 0 0 3 8 4 2 4 3 2 2 4 5 3 6 7
IME SO FAR          2.452


EST SQUARE 13
EST SQUARE 17                                                   STEP  8
```

```
TIME SO FAR      193.107


TIME SO FAR      193.139


TEST SQUARE  6
TIME SO FAR      193.191


TIME SO FAR      193.224


TIME SO FAR      193.256


TEST SQUARE 25                                        STEP 16
TIME SO FAR      193.347


STATE OF XX
  1  8  5  6  3 10  6  4  2  9 22  4 15  1  7  4 11 22  8 28 16 11 12 11 17 14 14 20 18 12 20 19 20 13 22 23
  0  0  0 12  0  0  0 19  0  0 24 23  0 22  0  0  0 29 27 31  0 14 27 28  0  0 19 24 33  0 27 28 22  0 24 29
  0  0  0  0  0  0  0  0  0  0  0  0  0 27  0  0  0  0 32 33  0 18 36 35  0  0 23 32  0  0  0  0 29  0 27  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 33  0  0  0  0 31 36  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 35  0  0  0  0 35  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
TIME SO FAR      195.605


TEST SQUARE 14 FORCED LOOP
TEST SQUARE 33                                        STEP 17
TIME SO FAR      195.729


STATE OF XX
  1  8  5  6  3 10  6  4  2  9 22  4 15  1  7  4 11 22  8 28 16 11 12 11 17 14 14 20 18 12 20 19 18 13 22 23
  0  0  0 12  0  0  0 19  0  0 24 23  0 22  0  0  0 29 27 31  0 14 27 28  0  0 19 24  0  0 27 28  0  0 24 25
  0  0  0  0  0  0  0  0  0  0  0  0  0 27  0  0  0  0 32  0  0 18 36 35  0  0 23 32  0  0  0  0  0  0 27  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 35  0  0  0  0 31 36  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 35  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
TIME SO FAR      197.965


GOTO SQUARE 29                                        STEP 18
```

```
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
TIME SO FAR     283.075


GOTO SQUARE 25                                        STEP 34
TIME SO FAR     283.164


STATE OF XX
  1  8  5 20  3 10  6 19  2  9 30 21 15  1  7  4 11 32 18 25 16 31 22 29 35 14 27 24 33 12 26 17 34 13 28 23
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
TIME SO FAR     285.405


GOTO SQUARE 14                                        STEP 35
TIME SO FAR     285.495


STATE OF XX
  1  8  5 20  3 10  6 19  2  9 30 21 15 36  7  4 11 32 18 25 16 31 22 29 35 14 27 24 33 12 26 17 34 13 28 23
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0



  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
TIME SO FAR     287.736


KNIGHTS TOUR
```

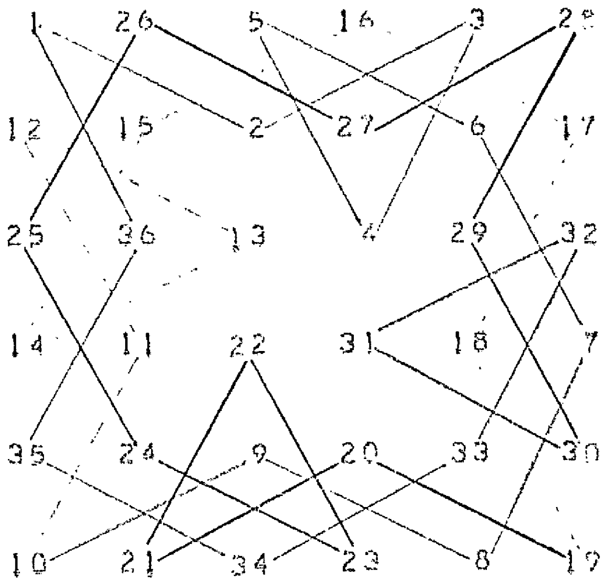


```
TIME SO FAR     288.097
```



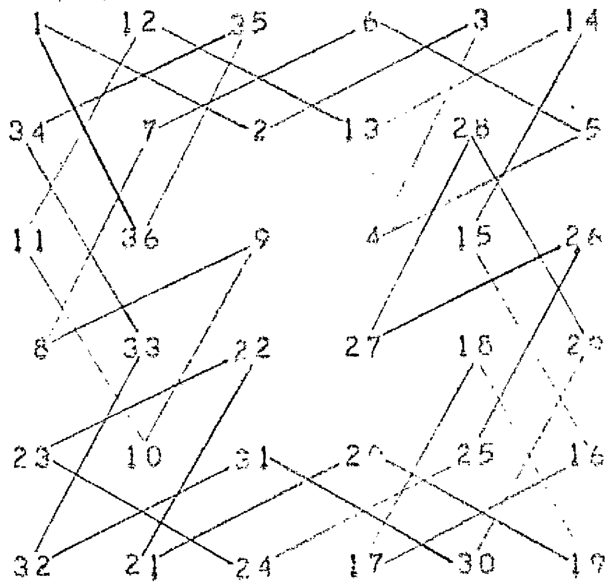
Type ABAB

```
TIME SO FAR    4.014

SYMMETRIC KNIGHTS TOUR      2
```



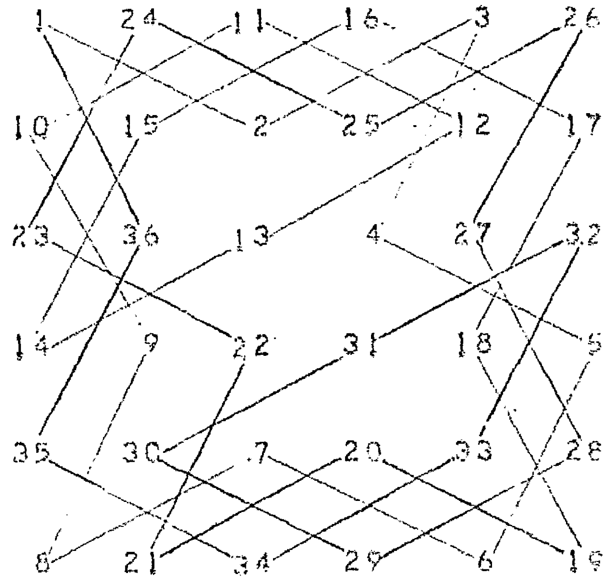
Type AAAA

```
TIME SO FAR    7.950

SYMMETRIC KNIGHTS TOUR      4
```



Type ABAB

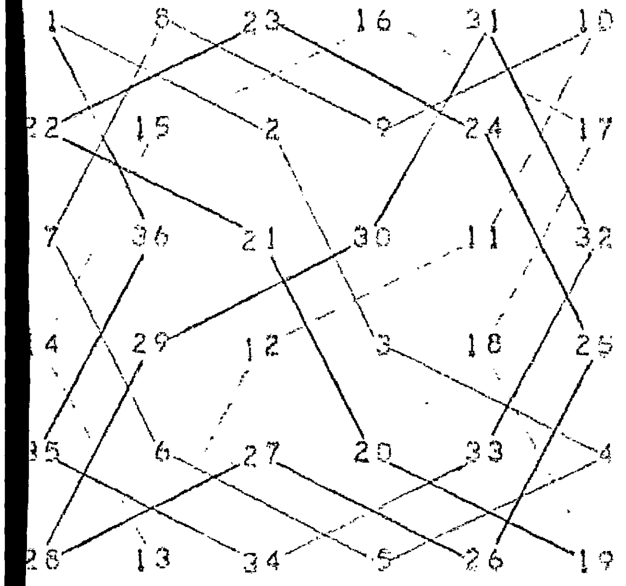
MMETRIC KNIGHTS TOUR 8



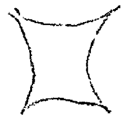
Type ABAB

ME SO FAR 81.508

METRIC KNIGHTS TOUR 77



Type AAAA



(related to No.48)

E SO FAR 309.721

Appendix 3

References

1. C. Berge, 'The Theory of Graphs', Methuen, 1962
2. G. Birkhoff & S. MacLane, 'A Survey of Modern Algebra' (3rd Edition), Macmillan, 1965
3. G. Dantzig, D. Fulkerson, & S. Johnson, 'Solution of a Large-Scale Travelling Salesman Problem', Op. Res., 2 (1954), 393-410
4. J.J. Duby, 'Un Algorithme Graphique Trouvant Tous les Circuits Hamiltoniens d'un Graphe', Etude no. 8, I.B.M. France, 1964
5. O.L.R. Jacobs, 'Introduction to Dynamic Programming', Chapman & Hall, 1967
6. I. Pohl, 'A Method for Finding Hamiltonian Paths & Knight's Tours', CACM, 10 (1967), 44-49
7. M. Pollock & W. Wiebenson, 'Solutions of the Shortest Route Problem - A Review', Op. Res., 8 (1960), 224-30
8. T.L. Saaty & R.G. Busacker, 'Finite Graphs & Networks', McGraw - Hill, 1965
9. H. Scoins, 'Linear Graphs & Trees', in 'Machine Intelligence 1' (ed. Collins & Michie), Oliver & Boyd, 1967
10. 'A User's Guide to COTAN', Computing & Applied Mathematics Group, Culham (U.K.A.E.A.) Laboratory, 1968
11. 'EGDON Programming System Manual', E.E.L.M. Computers Ltd., 1966.