Generating Knight's Tours Without Backtracking from Errors

Jefferey A. Shufelt
Hans J. Berliner

May 21, 1993
CMU-CS-93-161

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Abstract

We describe research on the problem of generating multiple closed tours of an $m \times n$ chessboard by a knight, subject to the constraint that the search scheme used to solve the problem is non-backtracking; i.e., that the search engine never visits a node in the search tree that will ultimately lead to a dead end. We describe our experiences and results in the context of KTC, a search program developed to undertake this task. We describe the implementation of KTC, the search constraints we discovered, and KTC's performance to date, illustrating that a limited amount of domain knowledge can lead to near-perfect search on this class of Hamiltonian circuit construction problems. We close by suggesting promising directions for achieving a perfect search on this problem and the implications of such an achievement.

The first author was supported by an Augmentation Award for Science and Engineering Research Training sponsored by the Army Research Office, Department of the Army, under Contract DAAH04-93-G-0092. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office, the Department of the Army, or the U.S. government.

Keywords: Knight's tour, Hamiltonian circuits, perfect search, non-backtracking search, robust heuristics, domain knowledge

1. Introduction

In graph search problems, the typical goal is to find a single (perhaps optimal) solution. The problem of finding *all* solutions to a graph search problem has seen little interest, perhaps due to the limitations established by complexity theory. In the general case, graph search is NP-hard, and enumeration and construction are, for all intents and purposes, intractable. The obvious question arises: why attempt a complete solution? A simple answer is that one may wish to know the cardinality of a solution set, or perhaps even know each and every member of the set.

With these goals, another issue arises. Heuristics can not be used to obviate search combinatorics in such problems, as we have no guarantee that these will not discard subtrees with valid solutions. If we wish to know all members of a solution set, we must utilize constraints which are guaranteed to discard invalid subtrees, and leave every valid solution path untouched. This leads us to a second question: can we accomplish a search under these conditions without backtracking from error? More generally, for a given search problem, can we solve the corresponding NBFF (No Backtracking From Failure) problem?

There are several points to consider:

- If a certain set of rules can be shown to solve a frequently occurring NBFF problem which is an instance of a general intractable problem, one then obtains a powerful technique for handling this intractability in practice.
- If a certain set of rules, while failing to solve the NBFF problem, comes close in the sense that the ratio of solutions to dead ends is substantially increased, then one may still be able to use the set effectively, or use automatic theorem proving techniques to extend the set.
- If NBFF generation of solutions is possible for a given sub-domain with a given set of rules, one is led to determine the extent to which other sub-domains may be solvable.
- NBFF problems represent another domain in which limited amounts of knowledge may be able to conquer state-space combinatorics. The degree to which this is true is of interest.

For the knight's tour problems we consider, which are a sub-domain of Hamiltonian circuit construction problems, connectivity of the graph is important. A completely connected graph has $n!$ circuits. In a sparsely connected graph, it may be possible to enumerate the circuits by a generative procedure. That such a procedure would be NP-hard is due to the fact that the degrees of vertices in an arbitrary graph can rise as the number of vertices in the graph. For a graph with some upper limit on degree, this class of problems might be solvable with a sufficient set of rules, as, for instance, the four-color problem was.

From a practical point of view, one should be able to partition all such problems into those that are tractable and those that are not. It would seem that tractability for a generator is largely based upon the number of vertices in the graph, and the average degree. A graph with high average degree is unlikely to yield to the methods we describe in this paper. However, many interesting graphs have low average degree relative to the number of vertices in the graph, so we hope that the development of effective rules for these graphs yields useful solutions. Further, for

simple domains, a few rules can in fact eliminate backtracking from error, so there is hope that for graphs of sufficiently low degree, rules can be found that eliminate backtracking completely.

In this paper, we describe our research on the NBFF knight's tour problem. Sections 2 and 3 describe the knight's tour problem and its history. Section 4 discusses the combinatorics of the construction problem and bounds on the number of solutions for an 8×8 board. Section 5 describes the concepts and details of some basic rules used to constrain search; more elaborate rules are described in the appendices. Section 6 outlines the search strategies and techniques we employed in KTC, the search engine, and Section 7 describes and analyzes a set of experiments we performed with KTC. We conclude in Section 8 by summarizing the performance of KTC and posing some conjectures for future work.

2. The knight's tour construction problem

The knight's tour problem can be stated as follows. Given a location x on a chessboard, find a sequence of moves that will cause a knight located at x to visit each square exactly once, with the final move returning the knight to x. This is also known as a *re-entrant knight's tour* in the literature; we refer to it as a knight's tour for brevity. Figure 1 shows one such tour for the standard eight-by-eight chessboard. It should be clear that the starting position of the knight is not an issue; if a tour exists, it must pass through every square on the board. We refer to the obvious extension of this problem as the knight's tour construction problem: given a location x on a chessboard, find n possible tours from that position, where n can be the cardinality of the set of possible tours. This is distinct from the enumeration problem, which only asks how many knight's tours exist for a board, and does not ask what the tours are.

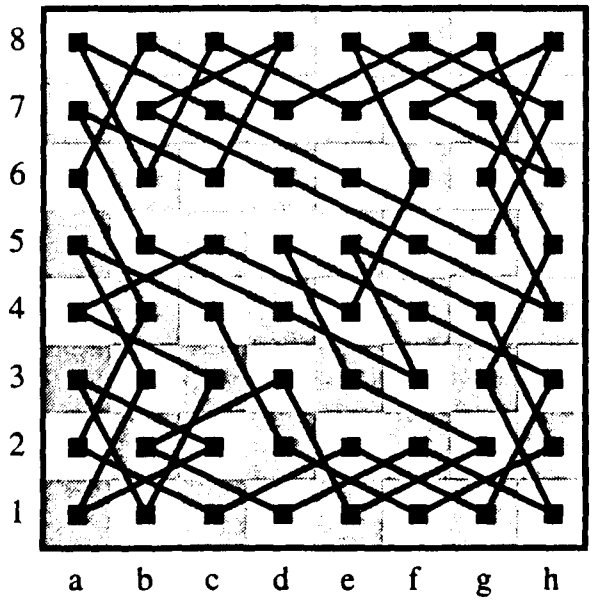

Figure 1: A knight's tour

	a	b	c	d	e	f	g	h
8	2	3	4	4	4	4	3	2
7	3	4	6	6	6	6	4	3
6	4	6	8	8	8	8	6	4
5	4	6	8	8	8	8	6	4
4	4	6	8	8	8	8	6	4
3	4	6	8	8	8	8	6	4
2	3	4	6	6	6	6	4	3
1	2	3	4	4	4	4	3	2

Figure 2: Liberty count at each square

3. Previous work

The knight's tour problem has been a popular pastime among mathematicians for centuries. Rouse Ball [1] provides an excellent account of early solutions to the knight's tour problem and several variants. We do not mention most of these, referring the interested reader to Rouse Ball for details, but we do note the first serious attempt at a mathematical analysis of the problem by Euler in 1759 [3, 1]. Euler's technique consisted of moving the knight at random over the board until no moves were open to it; he then applied a set of rules for inserting unvisited squares into the existing path in such a way as to form increasingly longer closed paths, culminating in a closed tour of the board. He found these points by searching backwards for places to splice unvisited squares into the tour while satisfying the constraints of knight movement. Euler's technique is of particular interest; although it was originally phrased in terms of numerical orderings of positions, the similarity of his basic technique with backtracking search is striking.

More recent work on search problems of a similar nature includes that by Rivin and Zabih [10], who developed a dynamic programming solution for determining $Q(n)$, the number of solutions for the n-queens problem. Their algorithm is exponential in n, but they posit that the value of $Q(n)$ is super-exponential; if so, their approach is superior in the enumerative version of n-queens. Takefuji and Lee [12] developed a neural network for generating knight's tours on rectangular chessboards, and provided examples of the network's performance on boards of varying sizes. The network was reported to more frequently converge on states consisting of several subtours as board size increased, however.

Kale's work on the n-queens problem [7] considers some of the issues we address here, both in terms of addressing the construction problem rather than the enumeration problem, and in terms of considering the robustness of the heuristics for generating solution sets of varying sizes. Kale described a heuristic for n-queens which was capable of finding single solutions without backtracking in many cases, and which also exhibited small amounts of backtracking for sets of multiple solutions.

4. The combinatorics of tour construction

Both knight's tour problems, non-constructive and constructive, can be couched in graph-theoretic terminology, if each square on the chessboard is regarded as a vertex in a graph, and if each legal knight move between two squares is represented by an edge between the corresponding vertices. In graph-theoretic terms, the knight's tour problem is an instance of the general problem of finding a Hamiltonian circuit in a graph, i. e., the problem of finding a closed traversal of a graph which visits every vertex in the graph exactly once. It is well known that the decision problem of finding a Hamiltonian circuit in an arbitrary graph is NP-complete; further, it is known that the Hamiltonian circuit construction search problem is NP-easy, and hence of equivalent complexity to the decision problem [4].

In our case, however, we are faced with a potentially much more difficult task. Assuming that the solution set for the Hamiltonian circuit problem is non-empty, we wish to obtain not just a singleton member of the solution set, but the set in its entirety. The cardinality of this set is not known for the knight's tour construction problem, to the best of the authors' knowledge. Rouse

Ball gives bounds for the cardinality, for a standard 8×8 chessboard [1]: the upper bound is the number of combinations of 168 items taken 63 at a time (roughly 1.18×10^{47}), due to de Jaenisch [6], and the lower bound is 122,802,512, the number of closed tours of a specific type, due to Kraitchik [9].

We can improve the upper bound by a combinatorial argument, and by noting some basic properties of a knight's tour. There are 168 potential knight moves on an 8×8 chessboard, but 8 of these moves must appear in every knight's tour; these are the two moves from each of the four corners of the board. These moves must appear since they are the only means by which each of the corner squares can be entered and exited in a knight's tour. We can then choose 56 more moves out of the remaining 160 to complete a tour, assuming for our purposes that every possible combination of 56 moves with the 8 corner moves will yield a valid knight's tour. The resulting upper bound is the number of combinations of 160 items taken 56 at a time, or roughly 6.44×10^{43}. This may be a pessimistic upper bound, but even in the best case it is clear that the construction of the entire set of knight's tours for a chessboard is impractical. Further, the upper and lower bounds are separated by over 30 orders of magnitude; this suggests the possibility that solutions could be extremely sparse relative to the size of the search tree. For the purposes of this paper, we will restrict ourselves to the problem of finding the first n knight's tours in the solution set, for some suitable n. This implies that an ordering of knight's tours is possible; as will be seen later, the choice of an initial square and a search scheme will uniquely dictate such an ordering.

The existence of an ordering allows us to address an important aspect of the construction problem. Finding the first n knight's tours under an ordering means that we must not use heuristics which might discard valid solutions. The search constraints we present in this work all share the key property that they discard subtrees of the search which are guaranteed to possess no valid knight's tours.

5. Search constraints

In this section, we describe the constraints we developed to guide the search for knight's tours. To begin a search, an initial position for the knight must be supplied; we refer to this position throughout as the *start square*. We refer to the position of the knight during an intermediate point in the search as the *current square*. Throughout, we say that two squares are *adjacent* if a knight can move from one square to the other in one move. We also use the terms *vertex* and *square* interchangeably, as well as the terms *liberty* and *edge*, to highlight their equivalence in this problem.

5.1. Liberties and basic constraints

The first version of our search mechanism utilized only one constraint, based on the square-specific notion of a *liberty*. A liberty is a single access route to and from a particular square on the board. Figure 2 shows the number of liberties available at each square of the board prior to the initiation of search; this table is updated after each move in the search to indicate the remaining number of liberties at each square. The rule captures the fact that the knight can never be allowed to work itself into a dead end:

> RULE 1: If a move of the knight to a new square would cause any square, excepting the start, current, or new square, to possess only one liberty, then the move should not be taken.

In graph-theoretic terms, the rule ensures that no vertices of degree 1 can appear in the graph, other than the current vertex and the start vertex. After the first move, the problem of finding the Hamiltonian circuit has been reduced to the problem of finding the Hamiltonian path between the current vertex and the start vertex. Certainly, these two vertices can possess degree 1. It should be clear that the presence of any other vertices of degree 1 in such a situation renders the construction of a Hamiltonian path impossible.

The next rule captures the idea that a knight can never place itself in a position where it must choose one of two mandatory moves. In graph-theoretic terms, this rule prevents the possibility of the search reaching a vertex which is adjacent to at least two vertices of degree 2.

> RULE 2: If a move of the knight to a new square would cause at least two squares adjacent to the new square to possess exactly two liberties, then the move should not be taken.

5.2. Backplanning, forced edges, and forced paths

The majority of the constraints presented in this work rely on a simple intuition about the effects of knight movement from square to square. Each move removes liberties from specific squares on the board; it seems reasonable to expect that these deletions would place limitations on the allowable paths a knight might traverse. In practice, these limitations are often severe enough that they can mandate specific sequences of moves to finish a tour; they can mandate specific sequences from the current position; and they can mandate specific sequences that must occur somewhere in the middle of a tour. In this section, we explain the key ideas behind these constraints, which permit a powerful analysis of board state.

Backplanning is based on the idea that every move eliminates potential liberties from intermediate squares on the board, and the removal of these liberties may force a specific sequence of moves for the completion of a Hamiltonian path; we refer to this ending sequence as the *endpath*. Backplanning is implemented as a recursion which begins at the start square, and attempts to work backwards, assigning squares a position in the tour.

The notion of a *forced edge* first comes into play during backplanning. A forced edge is a liberty which must be traversed in every Hamiltonian path from the current vertex to the start vertex. Trivial examples of forced edges are the paths from a1 to c2 and a1 to b3. Since a1 has only two liberties, any Hamiltonian circuit of the chessboard must traverse these liberties to enter and exit a1. Forced edges can be created during the search as unused liberties are removed from squares already visited during a partial tour, potentially reducing vertices in the graph to vertices of degree 2; it should be clear that both edges of such vertices must be forced edges, since one edge must be used as the entrance to the vertex and the other as the exit. We conclude the discussion

of forced edges with what may be an obvious point, but one worth noting nonetheless: if the current square has a forced edge, then there is only one possible move for which the search can proceed.

> RULE 3: **Backplanning** - Do not move to squares which have already been assigned an ordering in the endpath. The ordering assignment is recursive, beginning at the start square (which occupies position $n=m\times k$ in the list of moves, on an $m\times k$ board). If the square is connected to another square by a forced edge, recurse to that square and assign it position $n-1$ in the ordering. If the square is connected to only one other square, make the connecting edge a forced edge; recurse to that square and assign it position $n-1$.

The *end square* is defined as the last square the knight needs to reach to successfully complete a knight's tour. One might initially think that this is the same as the start square defined earlier, but the start square is fixed for all tours, whereas the end square can change as search progresses. Consider a tour with start square a1, in which the knight's first move is to c2. Upon this move, the end square changes from a1 to b3; the knight need only reach b3 now to complete a tour, since the edge from b3 to a1 has now become forced. If the search reaches the end square, then no more search is necessary to reach the start square, as there exists a unique sequence of moves from the end square to the start square. The backplanning mechanism is used to compute the end square for any given board state.

The next rule addresses the set of situations by which a square has its entrance and exit liberties planned, either by possessing only two liberties or by possessing two forced edges. In either case, a relative tour position for the square is determined.

> RULE 4: If a square is adjacent to two 2-liberty squares *A* and *B*, then remove all liberties from that square, excepting those that connect it to *A* and *B*. If a square has only two liberties, mark both as forced edges. If a square has two forced edges, remove all other edges connected to that square.

We introduce some additional terminology here. A *Plan1* vertex is a vertex with one forced edge. A *Plan2* vertex possesses two forced edges, and hence has been assigned a relative position in any Hamiltonian path. Prior to the initiation of search, the corner squares of a chessboard are all Plan2 nodes, since they only possess two liberties (and thus both liberties are forced); c2 would be a Plan1 node, since it possesses one forced edge to a1, but none of its remaining edges are forced.

With these definitions, we introduce a very powerful concept, which is used in virtually every constraint hereafter. A *forced path* is a sequence of at least two connected vertices, the first and last of which are Plan1 vertices, and the intermediate vertices of which are Plan2 vertices. Such a sequence of vertices is called a forced path because the knight is, quite literally, forced into taking the path. If the knight arrives at either end of a forced path, it must traverse the entire sequence from end to end.

RULE 5: **Simple cycle removal** - If the vertices at the ends of a forced path are connected by an unforced edge, remove the edge (unless the forced path connects the start square with itself).

RULE 6: If a square *B* is adjacent to a square *C* having three liberties, two of which connect to the endpoints of a forced path, then the liberty between *B* and *C* is marked as a forced edge.

The concepts of backplanning, forced edges, and forced paths are heavily used in the remainder of the ruleset, and form the basis for analyzing board state. As rules are fired, they may create forced paths, which can interact with other portions of the graph to create new forced paths, and so on. To ensure that all such interactions are obtained, the entire ruleset is applied repeatedly at every node in the search tree until the graph undergoes no modifications. Rule ordering is still an issue to be addressed, of course; Appendix 2 contains a description of an interaction effect between Rules 17 and 21. As described there, however, such ordering effects need not be cause for alarm, and in fact can lead to new constraint knowledge. Section 7.2 also addresses this issue.

In the appendices of this paper, we describe the remainder of the ruleset using a graphical notation, and we describe the relations they exploit to constrain search. In the current implementation of the searching mechanism, there are a total of 22 constraints.

6. Search strategy

KTC (Knight's Tour, Chess) is an implementation of a searching mechanism and a set of search constraints designed to attack this problem. In this section, we briefly describe some implementation details of KTC and motivate our choices for search techniques.

Since we were attempting to constructively enumerate solutions to the knight's tour problem, and since we knew the depth of the search tree beforehand, a natural choice for the searching mechanism was a stack-based depth-first search. A board state was represented by an $m \times n$ grid of pointers to nodes, each of which possessed pointers to other nodes according to their connectivity via knight moves. This representation allowed direct access to any square on the board via the grid, and access to neighbors (in the knight's sense) via the node-to-node pointers. (It should be noted that the original representation consisted only of the $m \times n$ grid, with square-specific information stored at each entry in the grid. It was not until after the implementation of a few constraints that it became clear that the underlying graph structure needed to be directly represented.)

Subtrees were always visited in a specific order. A knight has eight possible moves; each move was tried in clockwise order, beginning with the move which takes the knight one row forward and two columns to the left (the 10 o'clock move). The conjunction of this ordering with a choice of initial square for the search uniquely dictates an ordering of knight's tours. This proved useful for testing and debugging purposes; the first n tours from a square were generated by an early version of KTC, and then used as a comparison test for later versions to ensure that

constraints were coded correctly, since every version of the program had to generate the same tours in the same order.

The search mechanism does not utilize any form of lookahead; during the evaluation of a state in state-space, it considers only the information available at that state. Recall that we seek a searching mechanism that will never evaluate a node which ultimately leads only to dead ends. Any lookahead scheme must violate this constraint, albeit indirectly. By formulating the problem in this fashion, we can attribute performance gains solely to the constraint knowledge we add to KTC.

The search constraints described in the previous section and the appendices constitute the key machinery of KTC. These were developed intermittently over the course of a year, by examination of the dead end board states in which KTC found itself (on 8×8 boards), and design of rules to handle the most common dead end situations. KTC was heavily instrumented, to provide graphical output of board states in chess board form, and in graph form by vertices and edges; this instrumentation allowed us to easily inspect board states and determine the causes of dead ends in the search. Despite this, as the performance of KTC increased, it became steadily more difficult to discover the root causes of dead ends in the search, and to develop rules for these complex board states. It is our belief that any substantial improvements to the ruleset employed by KTC will ultimately come from automatic generation and testing mechanisms similar in spirit to those employed by theorem proving systems. Nonetheless, the manually derived ruleset in place as of this writing produces impressive results; we consider these results next.

7. Experimental data and analysis

In this section, we describe a set of experiments with KTC and the results of those experiments. We provide an analysis of the results and discuss their implications, including some thoughts on future directions for extending these experiments. Section 7.1 discusses the performance of the complete ruleset on a variety of boardsizes, including data for partial runs on an 8×8 board. Section 7.2 discusses the problem of rule ordering and its impact on knight's tour construction; it also describes the results of experiments which show the effects of incrementally adding constraints to KTC.

7.1. Initial experiments

In our first set of experiments, we ran KTC with all 22 rules in place on a variety of boardsizes, to obtain values for the number of knight's tours for these boards. It is worth mentioning a basic fact about the existence of knight's tours for $m \times n$ chessboards, noted in [1]. No board with an odd number of squares can possess a knight's tour, since every knight's move alternates the color of the square on which the knight resides. This allows us to consider only those boards with even numbers of squares.

Table 1 gives data for a number of $m \times n$ chessboards, showing the boardsize and number of positions for each boards. The table also presents the number of knight's tours for each board, and the number of dead ends encountered during the search, implying backtracking was necessary. The final column indicates the rules which were actually applied during the search.

Boardsize	Squares	Solutions	Dead ends	Critical nodes	Rules applied
3×10	30	16	0	0	3,4,6
3×12	36	176	50	36	1,3-5,6,21,22
3×14	42	1536	312	260	1,3-8,11,12,14,15,17,21,22
3×16	48	15424	4572	3468	1-15,17,19,21,22
3×18	54	147728	47250	34104	1-15,17,19,21,22
5×6	30	8	152	8	3-8,15
5×8	40	44202	3584	3149	all 22
6×6	36	9862	5115	2304	1-8,10-19,21,22
8×8*	64	20000	1438	1028	all 22

Table 1: Dead ends, critical nodes, and rules applied for solved boards

The fifth column gives the number of critical nodes encountered during the search. *Critical nodes* are those nodes in the search tree where there exists at least one subtree which possesses at least one solution, and where there also exists at least one subtree which possesses no solution. By this definition, the number of critical nodes for a search is a function of the ruleset, since a perfect search would never encounter a node with a subtree which contained no solutions. As the rule set increases the number of critical nodes will decrease monotonically until, for a perfect search, the number is zero. The number of critical nodes for each board is of interest because it represents an upper bound on the number of constraints necessary to achieve perfect search, assuming in the worst case that a unique rule is necessary to address each critical node.

To alleviate a potential source of confusion, note that the number of solutions listed for each board assumes that clockwise and counterclockwise tours are equivalent. Due to its depth-first searching mechanism, KTC does make this distinction, and generates twice the number of solutions shown here, producing both the clockwise ordering and the counterclockwise ordering from the start square as unique solutions. The number of dead ends and critical nodes are taken directly from KTC without modification, however.

Note that this provides a sanity check; any search completed by KTC must produce an even number of solutions, since each solution occurs twice. One other sanity check is that KTC must produce the same number of solutions on every complete search, regardless of the choice of starting square. We used these checks throughout our experimentation.

In addition to the boards shown in Table 1, we also ran KTC to completion on several other boards; these boards had no solutions. These were the 3×6 board, the 3×8 board, and the 4×*m* boards, *m*=3..14. These results suggest that 4×*m* boards might never have solutions, but we know of no proof for this assertion.

*The 8x8 data are, perforce, for partial runs. We chose 20000 solutions as the search termination point.

Boardsize	Dead ends rule 1 only	Dead ends all rules	Dead end gain factor	Nodes expanded rule 1 only	Nodes expanded all rules	Node expansion gain factor
3×10	2290	0	∞	5686	675	8.4
3×12	36435	50	728.7	91521	7549	12.1
3×14	514288	312	1648.4	1286484	71791	17.9
3×16	7119826	4572	1557.3	17938699	755539	23.7
3×18	98453338	47250	2083.7	248436509	7588831	32.7
5×6	18460	152	121.4	36504	849	43.0
5×8	10991529	3584	3066.8	22531739	1697531	13.3
6×6	1595962	5115	312.0	3321679	351110	9.5
8×8*	1974352	1438	1373.0	4412534	335726	13.1

Table 2: Performance gains on various chessboards

We found that searches on the narrow boards could proceed to greater depths (3×18=54 squares represents the deepest search run to completion). This is very likely due to the limited numbers of liberties (and hence smaller branching factors) available on these narrow boards. Any square on a 3×*m* board will possess at most four liberties; on a 4×*m* board, at most six liberties are available. The combinatorics of the problem are forcefully manifested on boards where 8-liberty squares are available, the 5×*m* and larger boards.

Only three of the rules were used in the only perfect search achieved thus far, that for the 3×10 board. More rules were used as the board was extended in size along its longer dimension, until the 3×16 board, after which only three rules (16, 18, 20) remain unused. It is interesting to note that these are the only 7-vertex rules in our ruleset; as can be seen in the appendices, all other rules use fewer vertices. It may be the case that these 7-vertex lattices can never arise on a 3×*m* board, but this also remains to be established.

The first set of data primarily serve as an exhibit of KTC's ability to complete constructive versions of the knight's tour problem for a variety of boards. Table 2 serves as an exhibit of KTC's power in reducing both the size of the visited state space and the number of dead ends encountered. The second and third columns of Table 2 contrast the number of dead ends encountered by two versions of the system: one utilizing only Rule 1 as a constraint, and the other utilizing all 22 rules. We note that the use of at least one rule is necessary, as the searches become unmanageable otherwise. The fourth column gives the multiplicative improvement between these two numbers. The remaining columns present a similar comparison on the number of nodes evaluated in the state space.

The dead end data show the effectiveness of KTC's ruleset. In all cases, the number of dead ends encountered was reduced by two orders of magnitude, and in some cases, well over three. This illustrates two points:

- Solutions are sparse relative to the number of dead ends.
- Limited amounts of domain knowledge can effectively address the majority of these dead end situations.

In the case of perfect search on 3×10, KTC was able to reduce the number of node expansions by a factor of 8.4. In other cases, the number of nodes to be evaluated was reduced by factors ranging from 9.5 to 43.0, showing that only a small fraction of state space needs to be visited during tour construction.

7.2. Rule ordering and incremental performance

In this section, we present performance data for a variety of measures while incrementally adding rules to KTC. We begin by addressing the rule ordering question, which always arises in search systems of this nature.

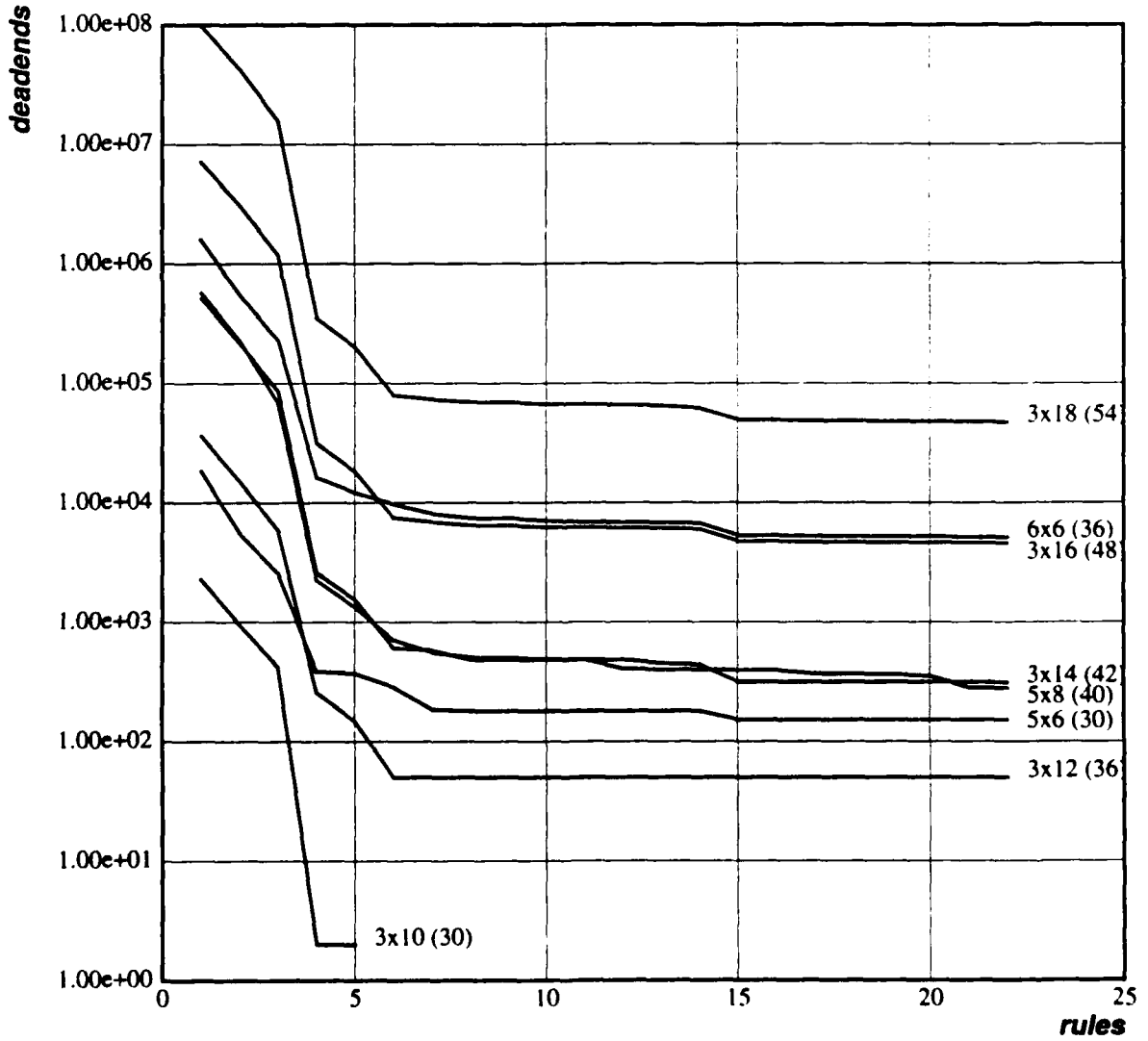

Figure 3: Number of dead ends as rules are added

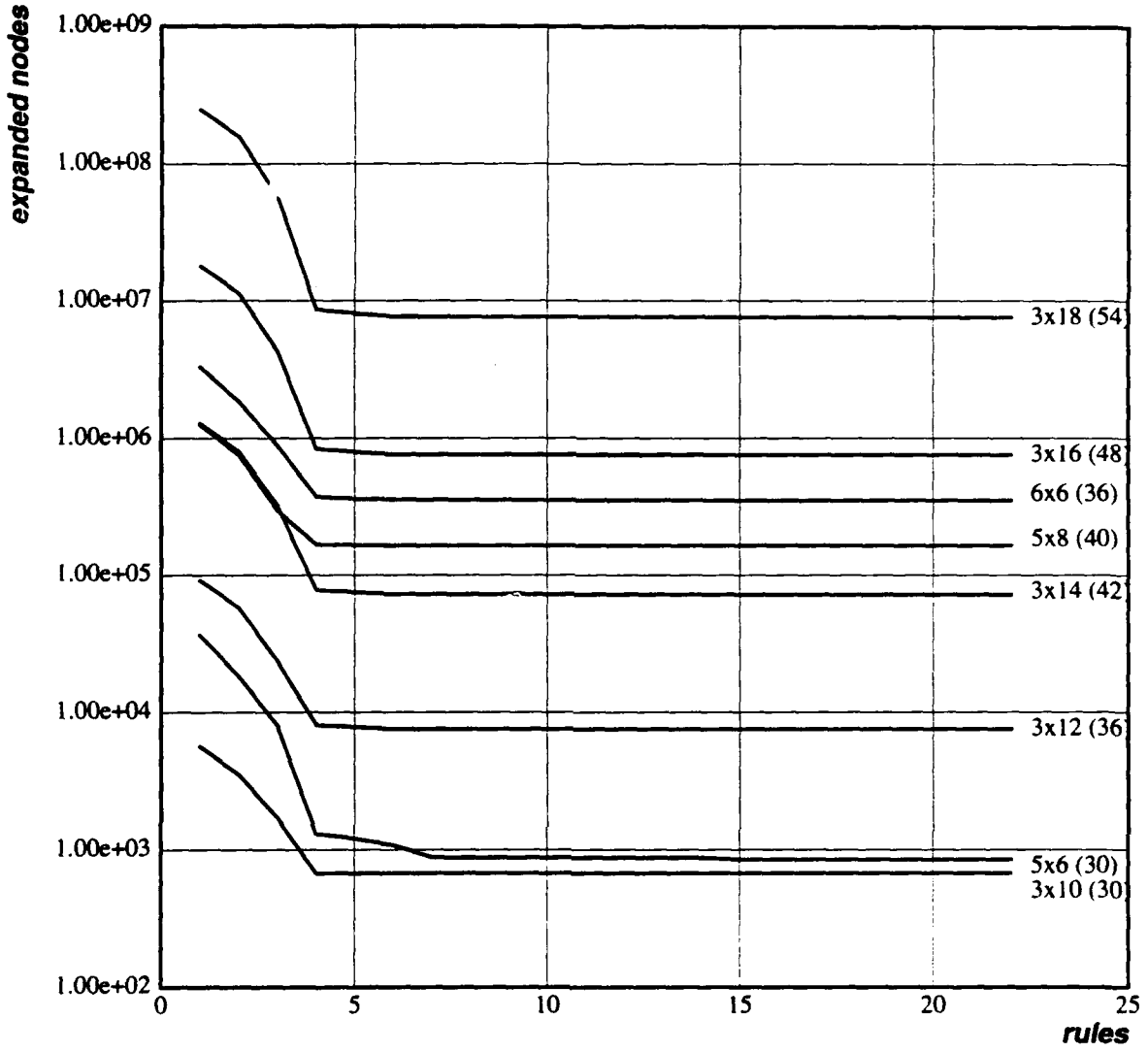

Figure 4: Number of expanded nodes as rules are added

For all experimental data provided in this section, the rules were incrementally added to the system in exactly the order they are presented in this paper. Certainly, we have not exhaustively tried every possible ordering of rules to ascertain which ordering provides the best performance; we have taken some care, however, to see that rules were added in order of decreasing power.

We argue that in constructive searches of the type we study, rule ordering is in fact a tool to be exploited, not a problem to be resolved (as it is typically treated in heuristic search). Two facts lead to this belief:

- Since the constraints we use are expressed as mathematical statements about the necessity or impossibility of certain subgraph traversals, any change in performance due to a reordering of rules implies a non-commutativity in the ruleset.
- The status of any liberty as a forced path or an unused edge in some search node is independent of the ruleset, and hence a change in the status of an edge under different rule orderings implies that an extension of the ruleset is derivable by inspecting the circumstances of that change.

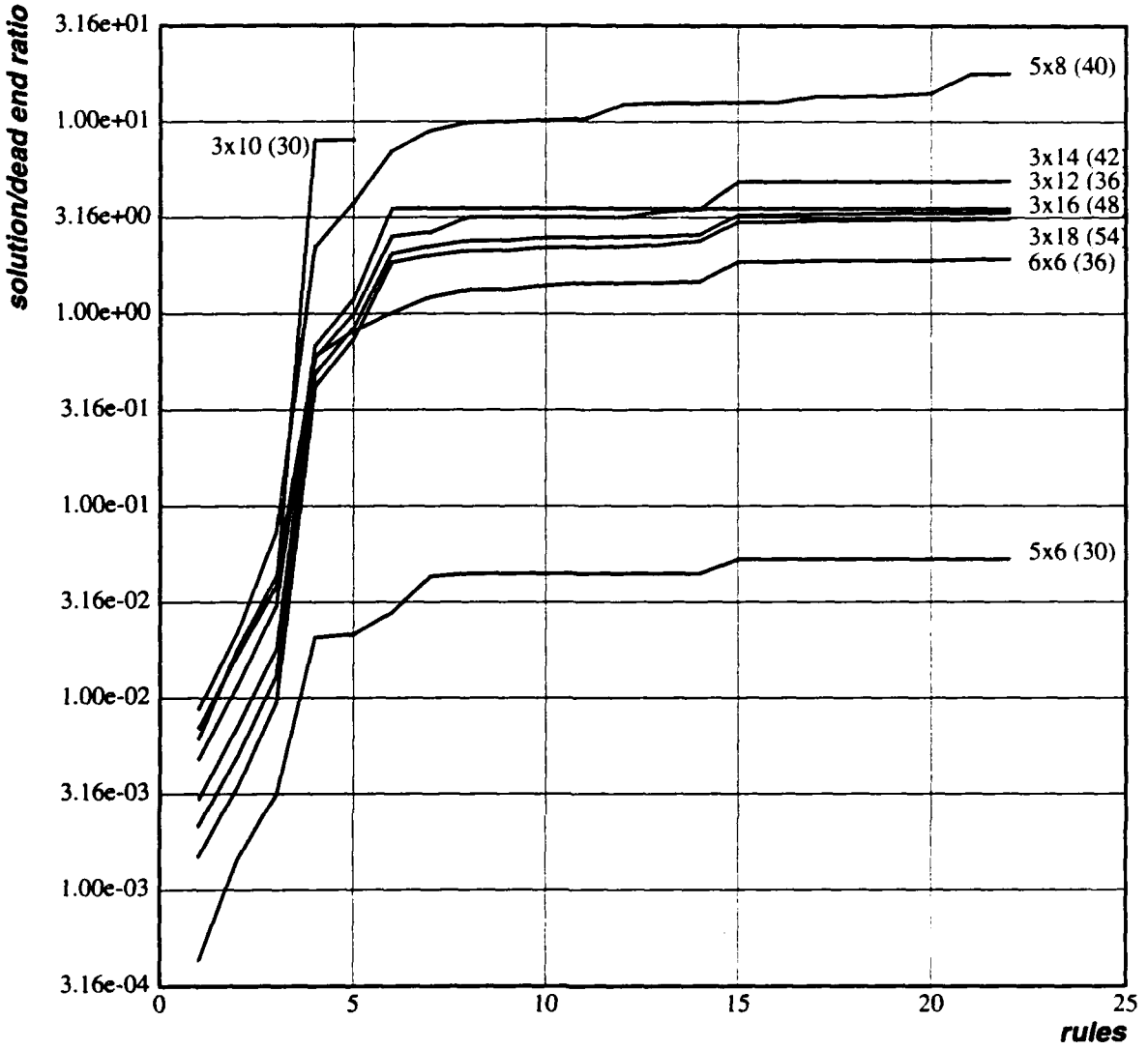

Figure 5: Ratio of solutions to dead ends as rules are added

Hence, in situations where such rule interaction effects arise, we can exploit these two facts to strengthen the ruleset. As discussed in the appendix, Rule 21 was found after such an effect was discovered between Rules 17 and 22. This illustrates that rule ordering should be regarded as a tool for discovering mathematical constraints on graph searches of this type.

Figure 3 depicts the number of deadends encountered by the system as rules were incrementally added to KTC, for each of the eight chessboards we have been considering thus far. In all cases, the vast majority of the dead ends were removed by the application of Rules 1-6 (note the logarithmic scaling of the graph). In fact, as mentioned earlier, a subset of these rules are sufficient to achieve a perfect solution on the 3×10 case.

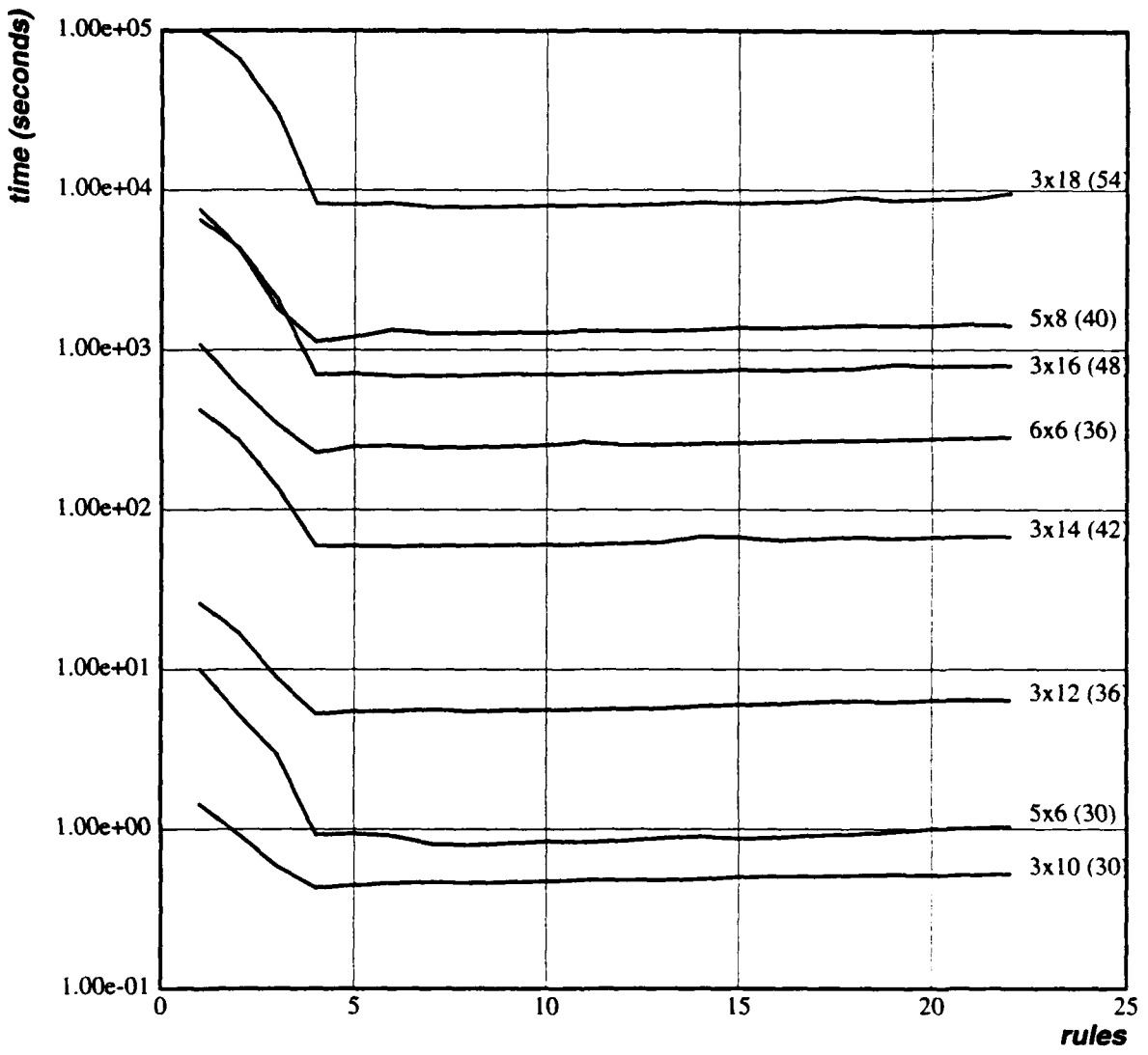

Figure 6: Search time as rules are added

Figure 4 depicts the decrease in the number of nodes visited by KTC as rules were incrementally added. In all cases, the majority of the unnecessary node evaluations were removed by the first four rules. The flat lines for later rules show that increasing knowledge has a very small effect on the total number of nodes visited in the state space; this suggests that their utility lies primarily in detecting more complex dead end states. To see this more clearly, we consider the next figure.

Figure 5 depicts the ratio of solutions to dead ends as rules were incrementally added to KTC. Again, we see the power of the first six rules in eliminating dead end subtrees from consideration. However, we can also observe more subtle effects of the later rules. For example, Rule 15 seems to have some power for detecting more complex dead end configurations; it is interesting to note that this was the first and simplest of several rules we discovered which exhibited a lattice shape. It is also interesting to observe that later rules induce a slow but relatively steady improvement on the 5×8 board.

The 5×6 board exhibits the same asymptotic behavior as the other boards, but with a much lower limit; recall from Table 1 that the 5×6 board had 8 solutions, but 152 dead ends. We do not have an explanation for this anomalous behavior, but we can conjecture at least one possibility, based on the following two observations. First, the critical nodes for 5×6 are quite complex; second, the 5×6 board is the smallest board we examined which possesses 8-liberty squares. It may be that the graph shows our failure to address the extremely complex critical nodes which could arise in the presence of 8-liberty nodes. This effect is less noticeable on 6×6 and 5×8, which could be explained by a large increase in state space (and solutions) relative to the number of these complex critical nodes. If this is true, then one encouraging implication is that the number of complex critical nodes might be quite limited, and hence one might not need to develop an impossibly large set of rules in order to achieve perfect search on the other boards. This remains to be seen, however.

Figure 6 depicts the search time as rules were incrementally added to KTC. Times were measured on an unloaded Dec Alpha AXP 3000/400 running OSF/1.

There are two important observations that can be made about the search time behavior:

- The exponential nature of the construction problem is quite obvious in this graph; the 3×*m* boards show that each addition of a 3×2 section to the board results in an **order of magnitude** slowdown in search time.
- The first four rules together provide an order of magnitude speedup over Rule 1 alone, and the incremental addition of the remaining rules produces a steady but slow increase in search time. This shows that the application of simple domain knowledge can produce substantial performance improvements. It also suggests that one might be able to achieve NBFF search with running times comparable to that achieved by searching with only one rule; this, however, remains to be seen.

Finally, we have also performed very large runs on the 8×8 board. In the initial stages of this research, when KTC used only Rule 1, we observed solution-to-dead-end ratios of 0.09 on the 8×8 board. The current version of KTC, utilizing all 22 rules, has been run on an unloaded Omron Luna 88K running MACH, for 30 days. During that time, it explored 510,851,013 nodes in the state space, encountering 1,627,809 dead ends en route to 27,544,000 solutions. This leads to a solution-to-dead-end ratio of 16.9, an improvement of two orders of magnitude over the initial ratio. This result is consistent with performance gains observed for smaller runs on 8×8, as well as runs on other boards.

7.3. Discussion

In this section, we present an informal discussion of the results we have obtained thus far, the implications of these results, and several future directions for this work. We begin the discussion by considering some specifics of this domain and the ruleset employed in KTC.

As noted in Section 7.1, the only rules which failed to fire on the 3×*m* boards were also the only 7-vertex rules in the ruleset. This suggests at least three possibilities:

- It may be the case that constraint knowledge need be no more complicated than 6-vertex rules for the $3 \times m$ sub-domain, and we have yet to discover the remaining n-vertex rules ($n<7$) for which NBFF search is achieved;
- It may be that certain critical nodes can only be solved by rules which must consider the majority of the nodes in the graph. If this is so, then the problem combinatorics will present another roadblock, as graph matching is not feasible for large graphs. As noted in [2], bounding the complexity of patterns is essential for efficient integration of pattern recognition into search.
- It may be that the remaining critical nodes hinge on relationships involving 4-liberty squares (or higher), instead of the 3-liberty squares which predominate in our ruleset. We feel certain that there exist many such relationships which we have yet to elucidate.

At this stage of KTC's development, elucidating these relationships manually would be a formidable task. Initially, this was not so; the first two rules are quite obvious. As we began to develop the notion of a forced path, however, substantial insight was required to see and exploit their effects. The lattice-shaped graphs we found later in our explorations seemed more amenable to analysis, and it is clear that many more graphs of this type remain to be discovered.

By inspection of some critical nodes for the simple domains of 3×10 and 5×6, it is clear that there exist other structures than the ones we have located thus far. The inspection process followed this general form:

- Locate a critical node in the search tree.
- Find a square in the critical node with few options for movement.
- If possible, find a particular move from this square that causes a deadend.
- By inspection, distill this information into a rule.

While there was a certain regularity to the rule discovery process, it quickly exceeded our abilities for analysis.

We highlight the difficulty of this approach by exhibiting a critical node from the 5×6 board in Figure 7. Using the same graphical notation as in the appendices, the square symbol represents the current square, the double circle represents the end square, and the other circles represent other squares on the board. The thick line represents a forced path, and the other lines represent liberties between squares. The numbers represent the count of available liberties at each square.

The node in Figure 7 is typical of the critical nodes we have observed. At present, we have no rule which can handle this node. Although there are undoubtedly some simple relationships which have escaped our attention, it seems clear that further rules must be obtained by computer-aided analysis of critical nodes. We believe natural candidates for such analysis are theorem-proving techniques, which might well uncover many more complex relationships, as well as simpler ones which have thus far eluded discovery.

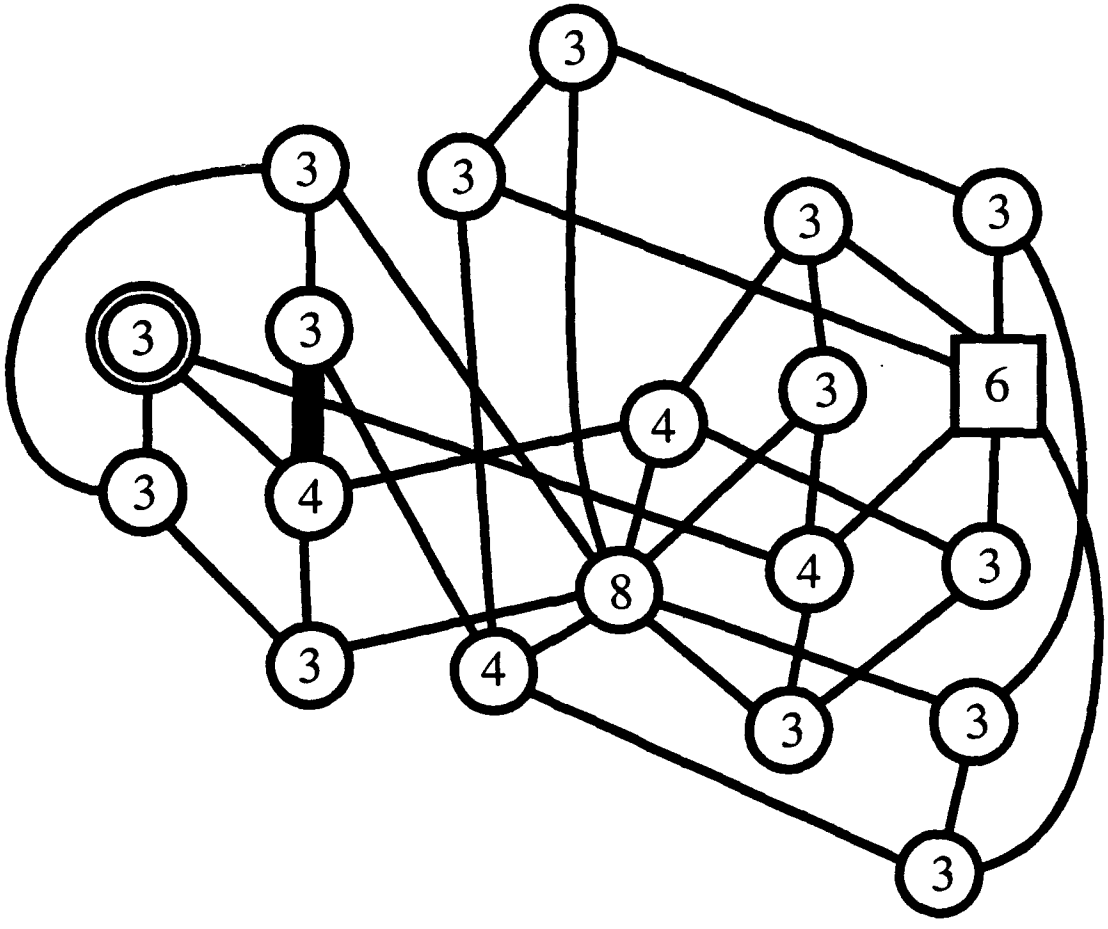

Figure 7: Critical node encountered on 5×6

If, in fact, all rules can be bounded in the number of vertices they possess, then a generate-and-test scheme might suffice for rule discovery, by positing various edges in an n-complete graph as either nonexistent, unforced, or forced, and evaluating the resulting rule. This might well be intractable for even small n, however.

Estimation of the size of the search tree would provide another avenue for future work, as one might be interested in obtaining good approximations to the cardinality of the solution sets for various chessboards. There has been a great deal of work on reliable estimation of search tree size, based on statistical sampling of portions of the tree; we refer the reader to [8] and [11] as examples of this work. These techniques could provide tools for estimating the size of the solution sets; alternatively, one might turn the problem around and use KTC to test the robustness of these sampling methods on a variety of solved boards.

A broad topic of interest concerns the generality of constraint knowledge of the type utilized in KTC. As has been known for some time, many combinatorially large problems can be handled through the use of heuristics or constraints, but formulating these rules is often non-trivial. Gaschnig [5] discussed the idea of problem similarity for devising heuristics, whereby one endeavors to develop heuristics for an easy problem which can be mapped into the harder problem of interest.

We suggest that this idea leads to an interesting experiment. Given the deep underlying similarity of all NP-hard problems, and the existence of polynomial time reductions for mapping from problem to problem, it might be illuminating to map the ruleset described in this work to some other NP-hard sub-domain that arises in practice, and evaluate the performance of the ruleset in that sub-domain. While this might not lead to dramatic performance gains in the new sub-domain, it should permit, at the very least, a qualitative assessment of the generality of this type of domain knowledge.

8. Summary and conclusions

In this paper, we have presented a set of constraints for achieving radical performance gains on knight's tour graph search problems. We have shown that it is possible to dramatically increase the ratio of solutions to dead ends in these searches, by factors ranging from 100 to 3000, and cut the size of state space under consideration by factors of 8 to 40. We have also shown that the combinatorics of these problems lead to an interesting knowledge/search time tradeoff:

- The application of simple domain knowledge can cut the search space dramatically, leading to a large initial speedup over a search using little knowledge.
- The incremental addition of more complex knowledge only modestly increases search time relative to the initial speedup. This implies that it may be possible to achieve NBFF search without paying the price of increased search time over a search using little knowledge, by using rules bounded in size.

This research has led to the following conjectures:

- $4\times m$ boards have no knight's tours.
- The maximum number of vertices in any rule required to perfectly solve a $3\times m$ domain is 6.
- Finding new rules has become tedious at best and will probably require a mechanical method capable of producing rules which have a beneficial effect on performance. We suspect that automatic theorem proving techniques are applicable to the problem of rule generation, and may be able to lead the way to NBFF search on this class of problems.
- If this approach can be mapped to other classes of NP-hard problems, it will be possible to make significant reductions in search times for other construction problems.

9. Acknowledgements

We would like to thank the MAPS group, Dan Stodolsky, and the MACH lab at Carnegie Mellon for the use of several workstations at all hours of the day and night for testing and experimentation purposes. We also thank Dave McKeown for his comments, support, and encouragement throughout the project.

References

[1] Rouse Ball, W. W., and Coxeter, H. S. M.
Mathematical Recreations and Essays.
Macmillan, New York, 1962.

[2] Berliner, Hans J.
Pattern recognition interacting with search.
Technical Report CMU-CS-92-211, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October, 1992.

[3] Euler, L.
Memoires de Berlin for 1759.
pp. 310-337, 1766, Berlin.

[4] Garey, Michael R., and Johnson, David S.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
W. H. Freeman and Company, New York, 1979.

[5] Gaschnig, John.
A problem similarity approach to devising heuristics: first results.
In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 301-307. 1979.

[6] de Jaenisch, C. F.
Applications de l'Analyse Mathematique au Jeu des Echecs.
Vol. 2, p. 268, 1862-3, Petrograd.

[7] Kale, L. V.
An almost perfect heuristic for the N nonattacking queens problem.
Information Processing Letters 34(4):173-178, April, 1990.

[8] Knuth, Donald E.
Estimating the efficiency of backtrack programs.
Mathematics of Computation 29(129):121-136, January, 1975.

[9] Kraitchik, M.
La Mathematique des Jeux.
pp. 360, 402, 1930, Brussels.

[10] Rivin, Igor, and Zabih, Ramin.
A dynamic programming solution to the n-queens problem.
Information Processing Letters 41(5):253-256, April, 1992.

[11] Stone, Harold S., and Stone, Janice M.
Efficient search techniques - An empirical study of the N-queens problem.
IBM Journal of Research and Development 31(4):464-474, July, 1987.

[12] Takefuji, Yoshiyasu, and Lee, Kuo Chun.
Neural network computing for knight's tour problems.
Neurocomputing (Netherlands) 4(5):249-254, August, 1992.

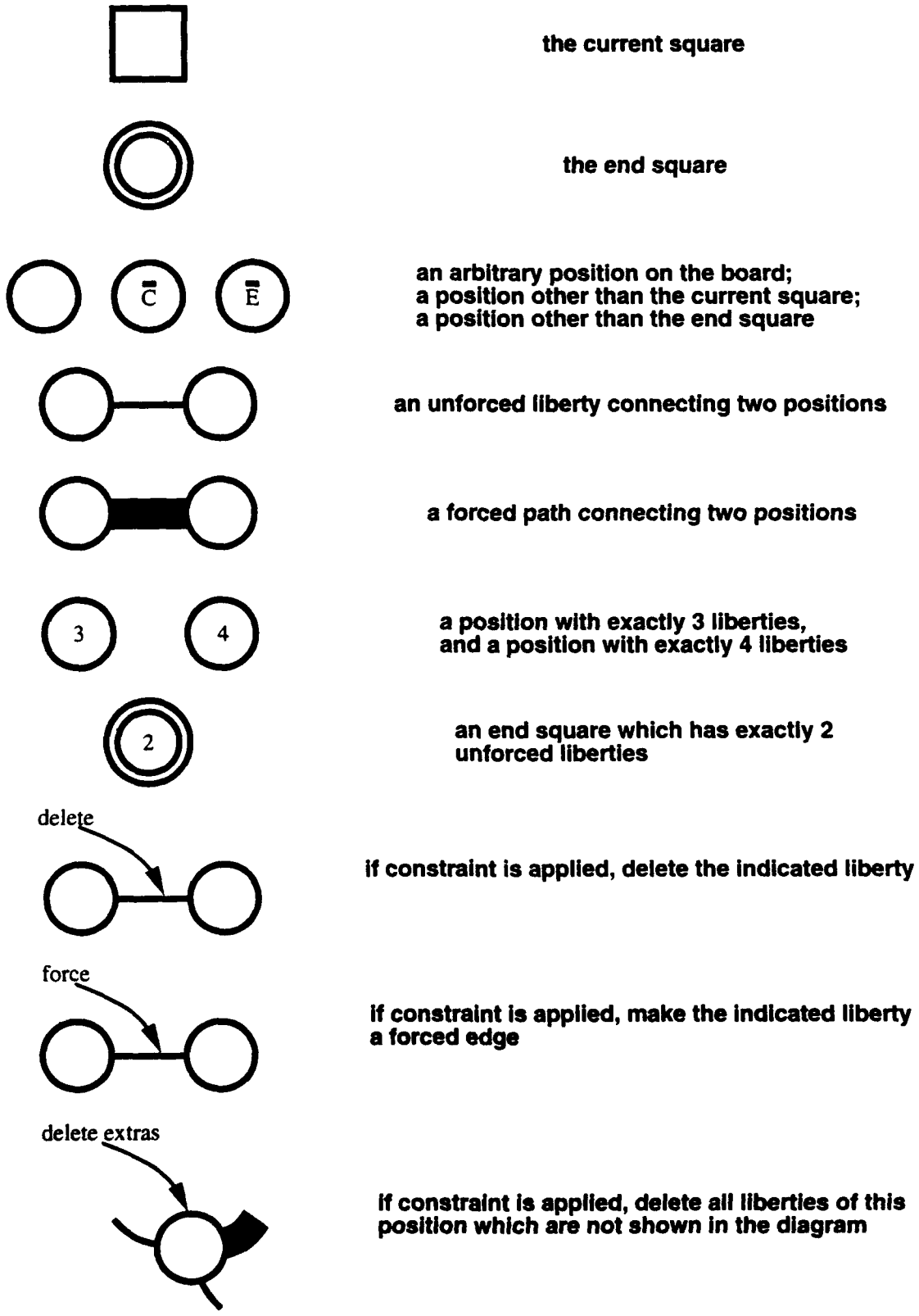

Figure 8: Glossary of graphical symbols

Appendices

In these appendices, we describe the remainder of the ruleset, using a simple graphical notation to show the constraints these rules express. We also provide brief explanations for many of the rules, and mention some interesting patterns and themes we observed during the course of developing the rules.

1. A graphical notation for constraints

In this section, we introduce a simple graphical notation to describe the remainder of the ruleset, as it illustrates the nature of the rules more effectively. We remind the reader once again of the equivalence of the knight's tour problem and the Hamiltonian circuit problem; this equivalence allows us to represent any state during the search as a graph, with one vertex for each unvisited square (including the current square and the start square), and an edge for each liberty, which connects the two squares sharing the liberty. The remainder of the constraints are expressed as relationships in subgraphs of a state graph. Figure 8 presents a glossary of the symbols and notations we use to represent these relationships.

We repeat Rules 5 and 6 in graphical form, to introduce the notation. In Figure 9, we see clearly that if the lower edge were to be taken, then a cycle would be formed. The only allowable cycle is the one that takes the knight through every square on the board, so if there exists any square that is not a member of the forced path, the lower edge must be deleted. We represent the presence of another square not involved in the pattern by another circle, disjoint from the rest of the diagram.

Figure 10 shows Rule 6 in graphical form. The leftmost edge must be forced in this situation for any knight's tour to be formed; this can be seen by considering the outcome if the edge was not taken. Since the central vertex initially possesses exactly three liberties, it would possess two if the third edge was removed. But, the application of Rule 4 would mark the remaining two edges as forced edges, and hence a cycle would be formed by the central and rightmost vertices. Thus, the leftmost edge must be forced to prevent a cycle.

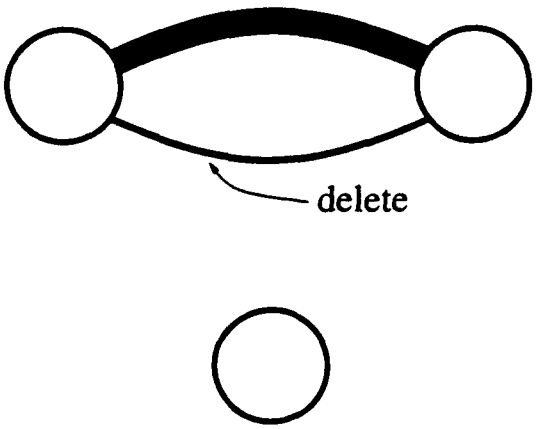

Figure 9: RULE 5

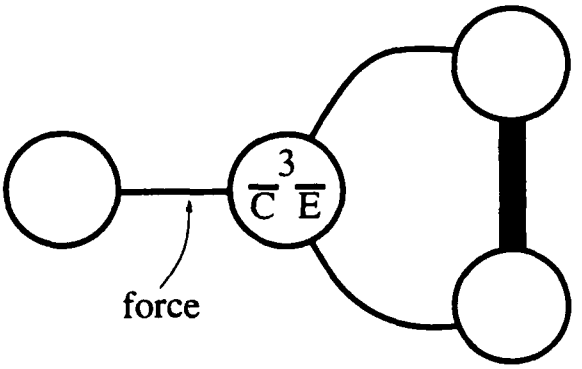

Figure 10: RULE 6

2. More constraints

In this section, we consider the remainder of the ruleset. We will not endeavor to provide a thorough discussion of each rule; we will, however, offer explanation for some of the more involved rules. We will also note certain interesting relationships which appeared frequently during our experimentation, and which may merit further investigation in future work.

Rules 7 and 8 capture certain situations that occur between the current and end squares, which must be resolved by the deletion of an edge. In both cases, traversal of the edge to be deleted would force the knight to prematurely reach the end square without visiting some other square first.

Rule 9 illustrates a case where several edges can be deleted at once from a single vertex. The operation is denoted by "delete extras" in the diagram. In this case, a successful knight's tour is

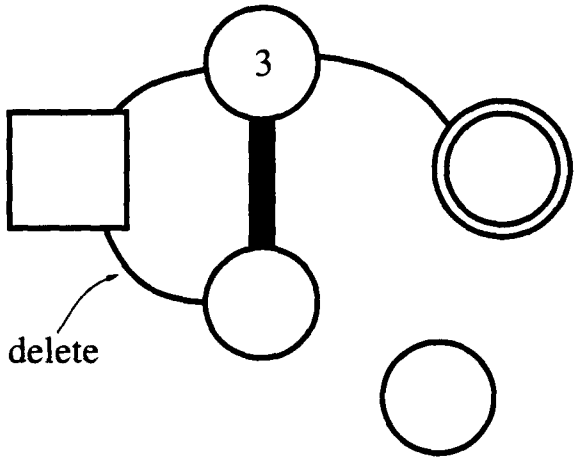

Figure 11: RULE 7

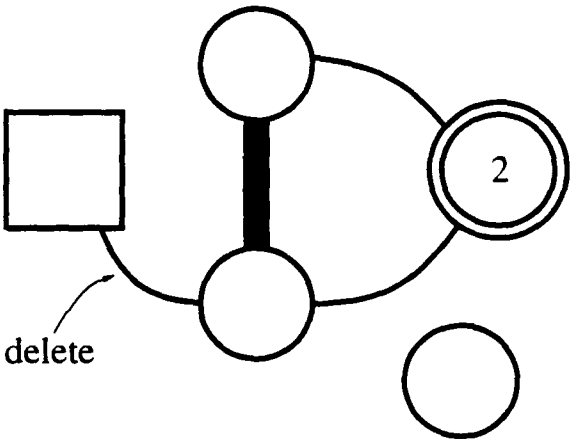

Figure 12: RULE 8

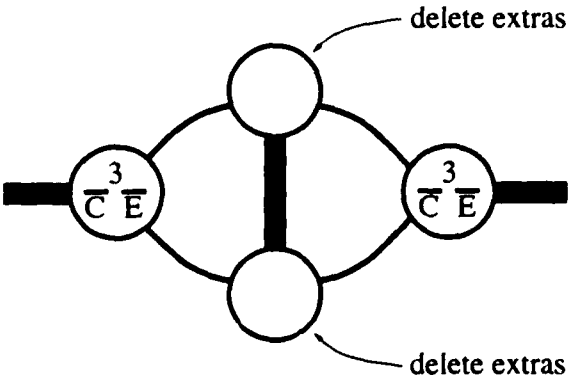

Figure 13: RULE 9

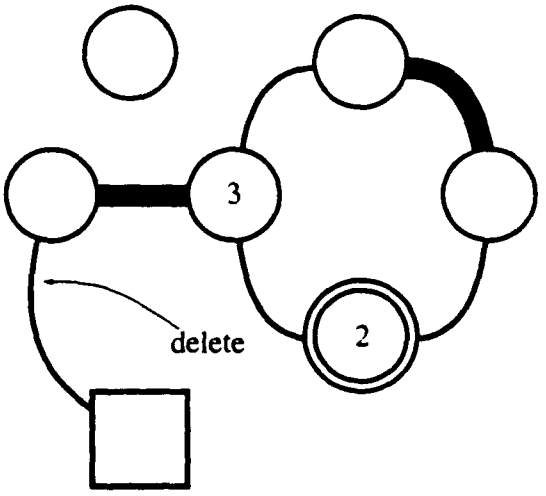

Figure 14: RULE 10

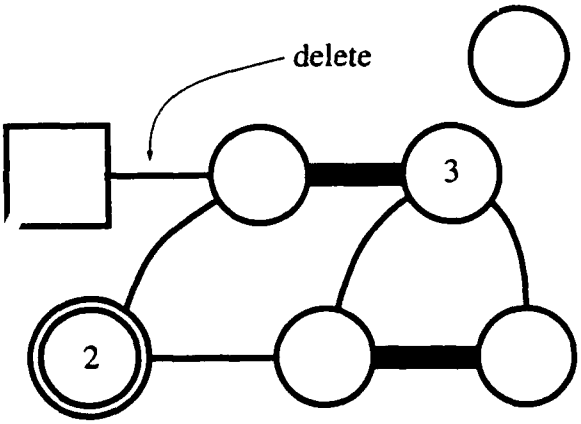

Figure 15: RULE 11

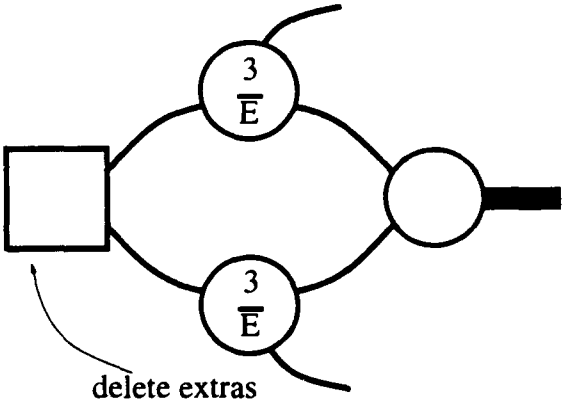

Figure 16: RULE 12

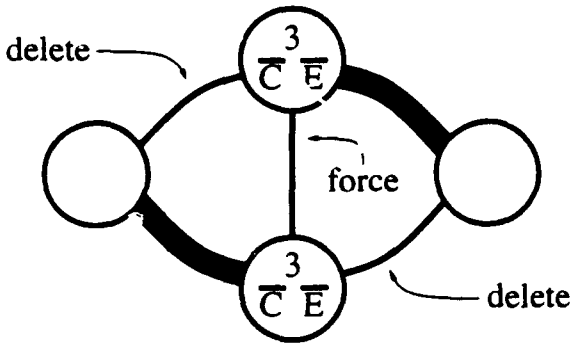

Figure 17: RULE 13

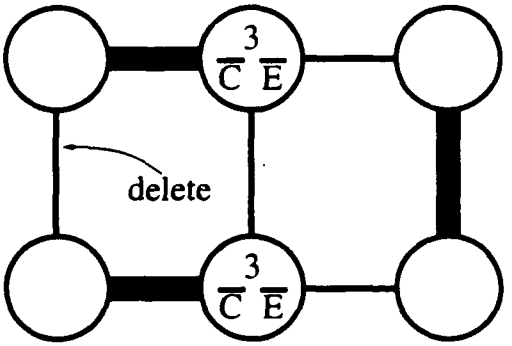

Figure 18: RULE 14

limited to the use of two of the four unforced edges in the diagram; if the knight were to take some other edge out of the top or bottom vertex, the result would be a vertex with three forced edges, an impossibility.

Rules 10 and 11 depict two slightly more complex situations that occur between the current and end squares. In both cases, traversing the edge to be deleted would result in a premature arrival at the end square, and the knight would hence never reach the detached position indicated in both diagrams. Rule 12 depicts a situation similar in nature to Rule 9, where the failure to delete edges resulted in a vertex with three forced edges.

Rule 13 illustrates a situation in which a moderately complex configuration of vertices can be directly reduced to a forced path. The edge connecting the vertices with three liberties must be used, since a cycle would be formed otherwise. But in forcing this edge, the other two edges must be removed to prevent the creation of smaller cycles, and hence a forced path is created between the left and right vertices in the diagram.

Rule 14 depicts a new structure which began to appear at earlier depths in the search. The graph exhibits a lattice shape, as do some of the later rules. We observed variations on this structure frequently during the course of our explorations; we suspect there may be some generalization of

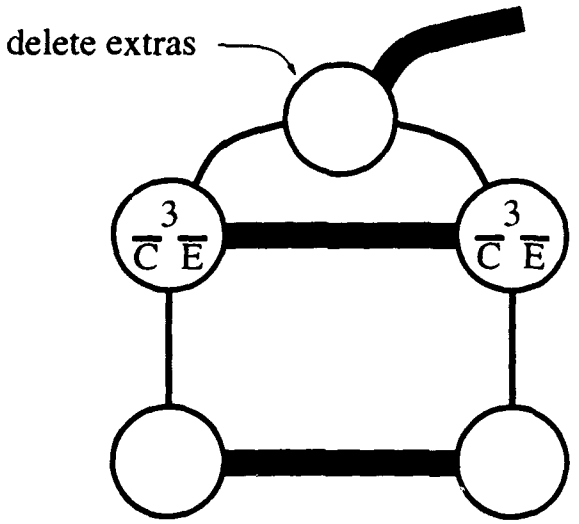

Figure 19: RULE 15

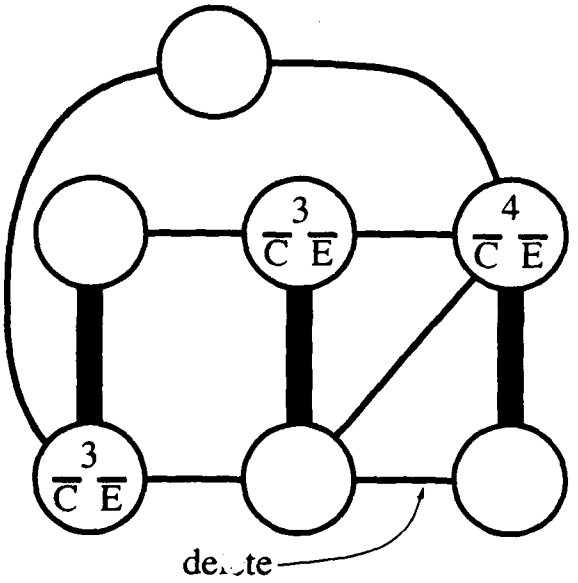

Figure 20: RULE 16

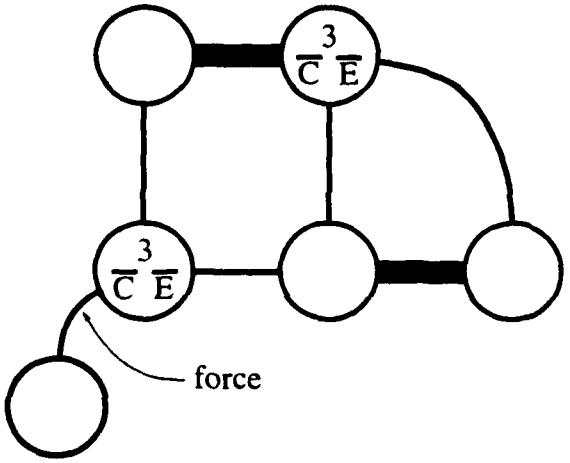

Figure 21: RULE 17

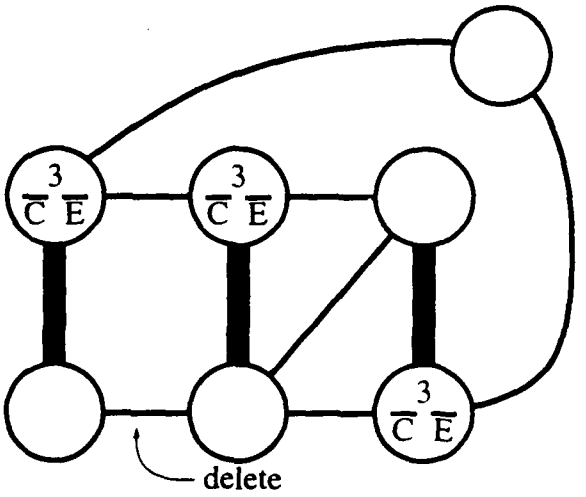

Figure 22: RULE 18

these lattice-shaped rules which will apply more broadly. In this case, if the leftmost edge were to be taken, then the center edge would have to be deleted; otherwise, a cycle would be formed. In its absence, however, both 3-liberty vertices become 2-liberty vertices, and hence a cycle will be formed.

Rule 15 depicts another situation where multiple edges can be removed at once from a vertex, as in Rule 12. Rule 16 is another instance of a lattice-shaped rule, although another external node is present. This rule is also notable because it is the only rule we have discovered which specifies a 4-liberty node. This should not be taken to imply that there are very few constraints which hinge on 4-liberty nodes; our experience suggests that there are almost certainly other such constraints.

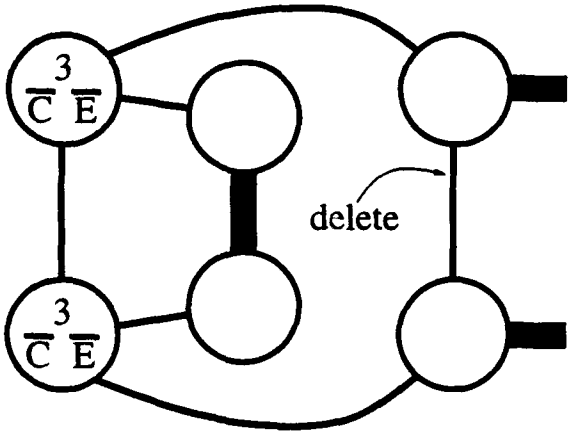

Figure 23: RULE 19

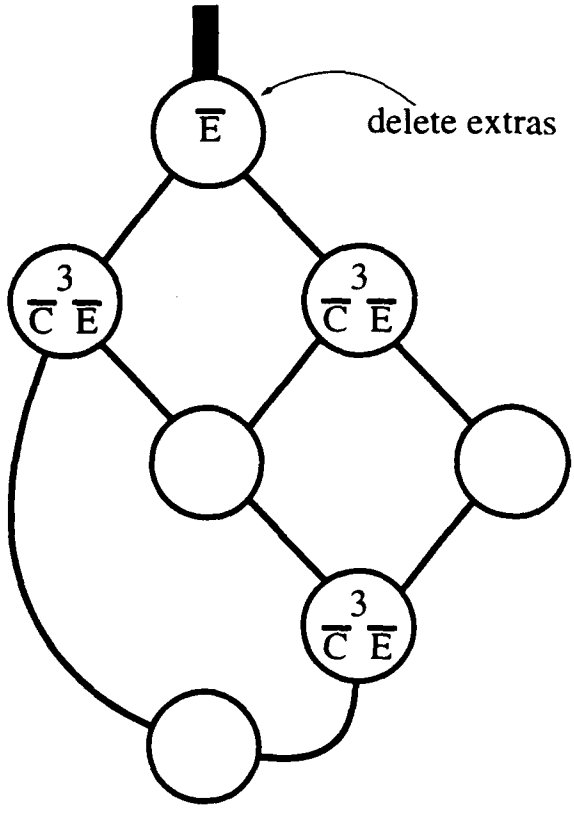

Figure 24: RULE 20

It should be taken to imply that manually finding constraints involving nodes with more liberties becomes very difficult, at least for the authors.

The remaining figures illustrate increasingly more complex configurations, involving up to seven vertices. As with previous rules, these constraints operate on the principle of cycle prevention. The presence or absence of specific edges in these subgraphs is all that prevents the graph from degenerating into a cycle. Rules 17-19 exhibit such behavior.

For completeness, we note that there exists a slight variation of Rule 20 which we have not illustrated for brevity. The top circle with a forced edge can be replaced by a square with no forced edges, i.e., the top circle can be replaced by the start square. In either case, at least one of the edges to the 3-liberty squares must be taken; otherwise, a six-vertex cycle is formed.

Of the remaining rules, Rules 21 and 22 merit special attention, as they shed some light on the independence of the rules in the ruleset. During testing and development of Rule 22, we found that some dead ends, previously eliminated by Rule 17, were reappearing in the search tree. Upon further investigation, we found that Rule 22 was forcing an edge, namely the leftmost vertical edge in Rule 17. This prevented Rule 17 from being activated in these situations.

It is the case, though, that if an edge can be deleted or forced prior to the application of a rule, then it can be deleted or forced afterwards. Its status as a valid or invalid edge in a knight's tour is independent of the ruleset. Hence, if the addition of a rule prevents another from activating,

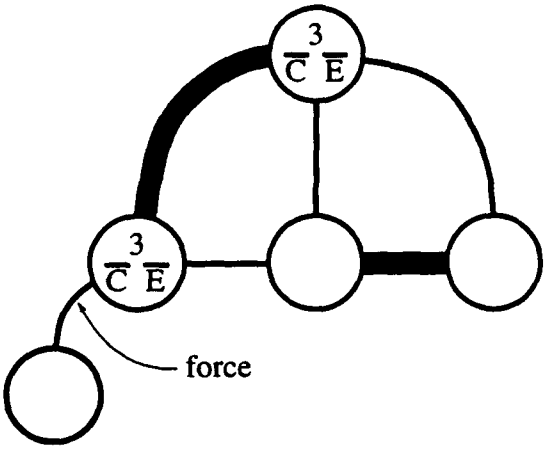

Figure 25: RULE 21

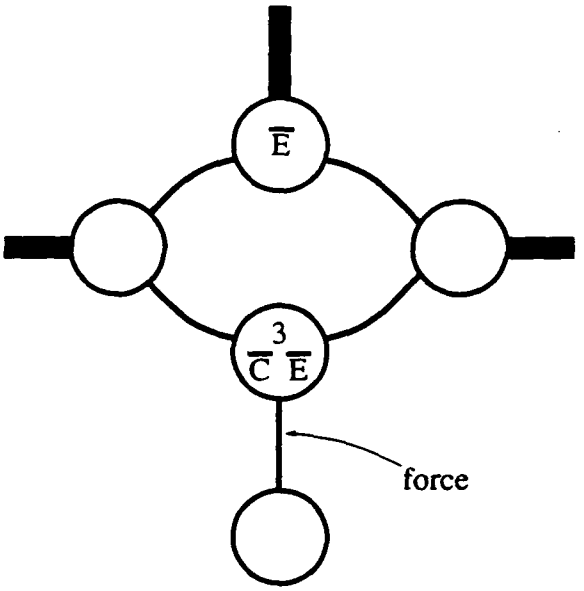

Figure 26: RULE 22

then there must exist some simplification of one of the rules which covers the new situation. In this case, Rule 21 was a simplification of Rule 17, and the similarities between the two rules are obvious. These experiences suggest that while the rules in the ruleset may be order dependent, this order dependence can be exploited to find new rules which will eliminate the dependence and strengthen the ruleset.