# Heuristic Strategies for the Knight Tour Problem

Luis Paris
*Institute for Advance Education in Geospatial Sciences*
*The University of Mississippi*
lparis@olemiss.edu

## Abstract

This paper presents three heuristic functions that aim to reduce the search cost for the *knight tour* problem. The first heuristic, *h1a*, is an interesting case of study that illustrates heuristic analysis, although it fails to obtain solutions. Heuristic *h2* is an enhancement of the *Warnsdorff* method, discussed as heuristic *h1b*. All heuristics are used in conjunction with a greedy algorithm to decide which node to expand next on a best-first basis. Tests show that heuristics *h1b* and *h2* narrow the search space considerably, in most cases to $O(N^2)$ time when no backtracking occurs. However, the *Warnsdorff* method, *h1b*, fails on some boards, sharply declining as $N$ grows larger. Heuristic *h2* adds a new criterion to address this issue, being capable of finding complete tours on very large boards in $O(N^2)$.

## 1. Introduction

The *knight tour* problem is another interesting puzzle among the domain of chess problems - another one being the famous *8-Queens* problem. However, unlike the latter with only 192 arrangements on an 8x8 board, the number of solutions to the *knight tour* problem becomes even larger and more intractable as $N$, the dimension of the board, increases. For instance, while for $N = 5$ there are only 304 solutions taken in just a few seconds, on a 6x6 board about half an hour is required to get the 524,486 solutions[1]. For $N = 8$, the number of solutions grows incredibly large, with 33,439,123,484,294 possible tours, obtained after running an enumeration program on 20 Sun Workstations for about 4 months [1]. However, later an error was presumed to be in the calculation since the number of tours must be, at least, divisible by 4. *Mordecki* [2] suggests a much higher upper bound of $1.305 \times 10^{35}$, and states that it is divisible by 8.

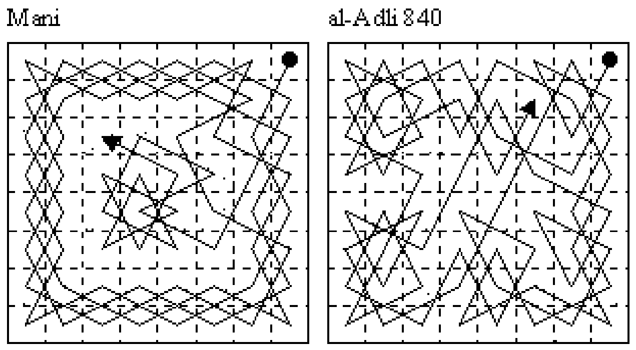


Fig.1. First two recorded tours on an 8x8 chessboard [7]

The original *knight tour* problem can be stated like this: Place a knight on an arbitrary square of the chessboard and visit every other square exactly once by performing knight moves until all 64 squares have been visited. The problem can be extended to *NxN* boards, of course, completing a tour when all *NxN* squares have been visited. Fig. 1 shows two different solutions on a regular 8x8 board.

The first recorded tours, depicted in Fig. 1, date back from 840AD when they were found in an Arabic manuscript [3]. In 1725, the *knight tour* problem seems to have been rediscovered and studied in Europe by many mathematicians, including *Euler* [4], without knowledge of the medieval work. However, it wasn't until the advent of computers when other algorithmic methods became available for testing.

Different problem classifications can be made from the *knight tour* problem. The number of possible tours that can be arranged in an *NxN* board is just one of them. Another one is finding particular tours that exhibit some esthetic property such as *symmetry*, like the one on Fig. 2, or tours that meet some particular requirement such as reentrancy, also called *closed*, *cyclic*, or *reentrant* tours. Note that the second tour in Fig. 1 is reentrant. Another type of problem, more suitable for large boards, is simply finding one solution, any complete tour. As $N$ grows large, the total number of possible tours a knight can make on an *NxN* board is no longer interesting due to the exponential complexity of the *search space*. However finding any solution among this vast universe of sequences, and reducing the cost involved in doing so, is interesting.

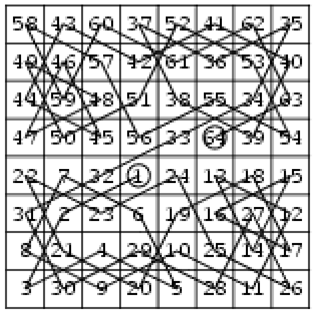


Fig.2. One symmetric, re-entrant, 8x8 tour [8]

In this paper, two of these problem types are addressed. First, the enumeration problem is implemented using a *depth-first search* approach to enumerate all possible tours on a generic board. Of course due to the exponential complexity of the search space, this is only tractable for small boards ($N<8$). Then another, perhaps more interesting problem is addressed: Finding any tour on a particular board, especially a large one, starting from a user selected square. This is achieved by implementing a greedy *informed search* algorithm with a particular *heuristic* as the *evaluation function*.

[1] On an Intel Pentium 4 CPU, running at 1.8GHz, with 1GB of RAM.

## 2. Problem Formulation

The *knight tour* problem, at least in its original inception, is a search problem whose path cost is of no interest in terms of cost minimization. Unlike the *Traveling Salesman*, the *knight tour* problem has no associated path cost to minimize. Any sequence of *NxN* moves that reaches a goal state is a perfectly valid solution. In other words, only the final state counts when performing the goal test. Solutions, if they exist, must reside necessarily at the last level of the search tree. Moreover, this result is independent of any formulation strategy used to abstract the problem.

Thus, the problem can be initially formulated as follows:

- *Goal test*: any sequence of exactly *NxN* different positions, obtained after performing valid knight moves.
- *Path cost*: zero
- *Operators*: perform a *knight move* from a previously visited square to a new unvisited square (see Fig. 8).
- *States*: any sequence of 0 to NxN visited squares.

The proposed formulation strategy is *incremental* rather than *complete-state* [5], because new valid squares are added one by one to the sequence. If the sequence has exactly *NxN* different squares, then a new solution is found. There could be instances where no further move is possible to an unvisited square, a situation known as a *dead-end*. In such a scenario, a typical search algorithm will probably enter a backtracking mode in which previously added squares are removed from the sequence, and other moves are tested. Obviously a good search algorithm needs to avoid such *dead-ends*, in order to minimize the search cost.

Note that a *depth-first search* (DFS) implementation would follow almost naturally from the above formulation, assuming that the operator is performed in sequential order. However, other more intelligent search schemes prove to be more effective than DFS, especially when specific *heuristics* are used to select successive operators. In section VI, this will be explored by presenting three heuristic functions.

## 3. General Search Algorithm

Most search algorithms can be implemented using a *general search* algorithm with a *specific search* strategy [5]. The *general search* algorithm basically determines the main strategy that aims to narrow the search space in an effort to find solutions, given a particular *problem* as input. However, it does not evaluate which nodes will be expanded and added to the search path. That is the purpose of the *specific search* strategy, which dictates the sequence of nodes to expand based on an *evaluation function*.

A *node* holds information associated with one level of the search tree. The most relevant information attached to a node is the *state*. What the *state* means and how it is interpreted depends on the particular problem. The set of all possible states make the *state space* for the problem. In addition to *states*, a node can optionally have other associated *values* that must be stored and book-kept at each level of the search tree. Such *values* typically make sense in *informed search* schemes and are defined by the *specific search* strategy.

Fig. 3 shows the *general search* function for the *knight tour* problem. The variable *Position* is an array of bi-dimensional vectors, each representing a knight placed on the board. Thus, a *node* consists of a *position* and other optional *values* that depend on the search strategy. The *queuing function*, explained below, is the most important part since it specifies the order in which prospective nodes are selected and iterated, for each level, along the search path.

```
function GENERAL_SEARCH( Queueing-Fn)
   solution_found = FALSE
   Position[ 0] = (x_0, y_0)  //Place 1st knight arbitrarily
   SetKnightPosition( 1, Queueing-Fn)  //place 2nd knight until last
   return solution_found //return TRUE if found, FALSE otherwise
end function  //GENERAL_SEARCH

function SetKnightPosition( n : integer, //n : node number
                                   Queueing-Fn)  //Queueing function
  pos_sequence = Queueing-Fn( n) → seq{ 1..8 }
  loop i in pos_sequence  //position sequence stored per node
    Position[ n] = Position[ n-1] + KnightMove[ i]
    if Position[ n] within range AND NOT visited
      if (n is last node)
       solution_found = TRUE  //SOLUTION FOUND
     else
       Mark Position[ n] as visited
       SearchKnightPosition( n + 1)  //recursion
       Clear Position[ n] as visited
  end loop
end function  //SetKnightPosition
```

Fig.3. General search algorithm for the *knight tour* problem

For informed search methods, the *specific search* strategy is defined in terms of an *evaluation function*. For uninformed search methods, no evaluation is performed. The purpose of the *evaluation function* is to minimize the search cost involved in looking for solutions. In terms of efficiency, an *evaluation function* is relatively better than another one if its overall *search cost* is lower. It will also be *optimal* if it is guaranteed to reach a solution in the minimum possible number of steps. In the context of the *knight tour* problem, an optimal solution is found when a tour is completed in exactly *NxN* knight moves without getting trapped in any *dead-end*, and hence no backtracking occurred. Hence, an optimal solution is in $O(N^2)$.

## 4. Best-First Search

The three heuristics to be discussed, designed to evaluate the best square to visit next from the current node, can be implemented using a *best-first* search algorithm, having each prospective heuristic as the *evaluation function*.

```
function BEST-FIRST-SEARCH( Eval-Fn)
   returns success (TRUE) or failure (FALSE)
   inputs: Eval-Fn, an evaluation function
   Queueing-Fn ← a function that orders nodes by Eval-Fn
   return GENERAL-SEARCH( Queueing-Fn)
end function
```

Fig.4. *Best-first* search algorithm

Fig. 4 above illustrates the *best-first search* algorithm. The *queuing function* "Queueing-Fn" simply sorts, in ascending order in this case, the prospective nodes to be expanded, according to the node *value* returned by the *evaluation function*. Therefore, general search is invoked on the least-valued node first, continuing with the second-valued, and so on, up to the most-valued node, rather than going sequentially as in *depth-first search*. The *value* of a node in this context is defined by the particular *heuristic*, which in turn is measured in terms of its *evaluation function*.

## 5. Enumeration Problem

The goal in an enumeration *knight tour* problem is to find all possible tours for a given *NxN* board. If uninformed search is used, a simple *depth-first search* scheme should suffice to obtain all of the solutions. However, for large boards ($N > 6$), the exponential *state space* of the problem clearly bounds the search in such a way that any brute force method will prove very inefficient.

On the other hand, informed search methods help find solutions more efficiently, but the time required for such methods to complete execution becomes more intractable as *N* increases, since the number of solutions grows exponentially as well. Table 1 shows the number of solutions as a function of *N*, and the approximate execution time.

Table 1. Number of solutions, and Execution time, for *N*

| ***N*: Board dimension** | **Number of solutions** | **Execution time** |
|---|---|---|
| 4 | 0 | – |
| 5 | 304 | < 1 second |
| 6 | 524,486 | 19 minutes |
| 7 | ? | ? |
| 8 | $< 1.305 \times 10^{35}$ | > 1 year |

## 6. Inventing Heuristics

The main reason why brute force methods like *depth-first search* algorithms are very inefficient is because most of the execution time is lost in expanding and backtracking nodes that will never reach a goal state. *Backtracking* is just the process of resuming after a failed attempt while trying to reach a goal state. Understanding why it occurs in this particular problem is the first step to avoid failed states, and when doing so, heuristic ideas come almost naturally.

In the particular context of the *knight tour* problem, there are two situations that may cause a sequence to prevent from completing a tour:

- *Inaccessible square*. There is an isolated square on the board impossible to visit, because all valid squares from where it is accessible have already been visited.
- *Dead-end*. There is no unvisited square that can be visited from the current square.

Fig. 5a depicts a *dead-end* situation that occurs after the 15th move, or node 15. The knight is about to reach a *dead-end* by moving to the only square available (2,1), after which no further move is possible. Fig. 5b shows an *inaccessible square* situation, where (2,1) is inaccessible.

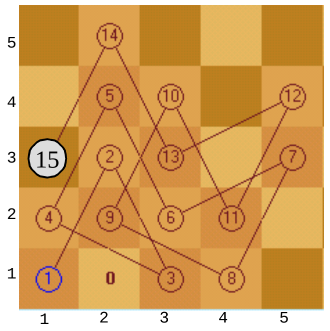


Fig.5a. Square (2,1) is a *dead-end*.

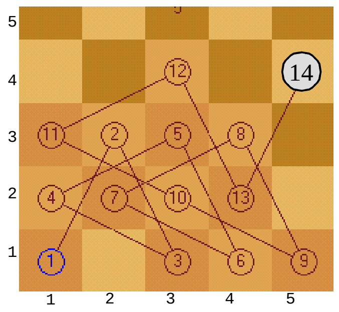


Fig.5b. Square (2,1) is *inaccessible*.

It is helpful to analyze the first case. Clearly the knight at node 15, in Fig. 5a, realizes it is in a *dead-end* because from (2,1) there are no further available squares to move next. However, it would be desirable to detect this fact before approaching the actual *dead-end*, in order to prevent backtracking. By looking at node 15 in Fig. 5a, the only plausible move is (2,1) where the *dead-end* occurs. Again, could it have been detected earlier? To do so it is obvious that the current node needs to know in advance all the possible move options for all the squares that are accessible from it. This suggests enumerating the number of options that each prospective successor square currently has. In Fig. 5a, the only successor square from node 15 is at (2,1), and from there, the number of options is *zero*. This confirms that such a square is a *dead-end*. However, backtracking still needs to be performed. Therefore, the question arises, from which move can the *dead-end* be prevented? By visual inspection it is trivial to see that from node 14, the choice of square (1,3) for node 15, instead of (4,4), produced the *dead-end*. Fig. 6a shows the number of options for successor squares from node 14, one move from (1,3), and five moves from (4,4). This suggests that by choosing the square with *most* options to move, *dead-ends* can be prevented. Let this heuristic be *h1a*.

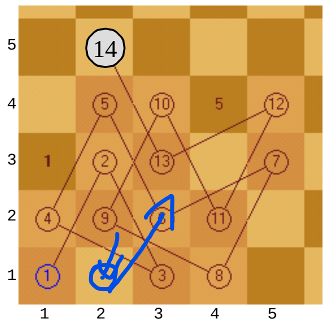


Fig.6a. Preventing "dead-ends."

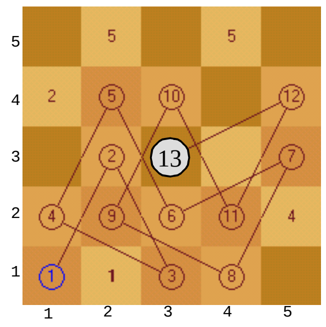


Fig.6b. Preventing "inaccessible" nodes

However, even if *dead-ends* were minimized by choosing the squares with most options, there is still a problem. In Fig. 6a, by choosing (4,4) for node 15, square (1,3) becomes *inaccessible*. Again, backing up one node from 14 and resuming the search, Fig. 6b shows the number of options for successor squares starting from node 13. Note that if square (2,1), the one with least options, is chosen, both (2,1) and (1,3) now become *accessible*. Therefore, by choosing the node with least options, *inaccessible* squares can be prevented. Let his heuristic be *h1b*.

Table 2. Proposed heuristics and their intended action

| ***Heuristic*** | ***Avoids dead-ends ?*** | ***Avoids inaccessible squares?*** |
|---|---|---|
| *h1a*: Visit square with "most" options | Yes | No |
| *h1b*: Visit squares with "least" options | Yes , indirectly (not intentionally) | Yes |

Table 2 above summarizes the two analyzed prospective heuristics *h1a* and *h1b*. Note that in Fig. 6b, after choosing the square with least options at (2,1), the previous *dead-end* at that position (see Fig. 5a) was indirectly prevented.

## 7. Results of Applying Heuristics

The question arises, which heuristic is better? In theory, the last proposed heuristic in Table 2, *h1b*, looks more promising since it seems to prevent *dead-ends* indirectly, in addition to *inaccessible* squares. The best way to check this is empirically through a test program. Such a program should receive as inputs both the *dimension* of the board, *N*, and an *initial square* selected by the user. It should consist of an informed search algorithm, with each proposed *heuristic* as the *evaluation function*. Each returned value, indicating the number of options that each prospective successor square currently has, will then be sorted in either descending or ascending order by a *queuing function*, depending on whether heuristic *h1a* or *h1b* is chosen, respectively. Figs. 3 and 4 show the *general search* and *best-first search* algorithms.

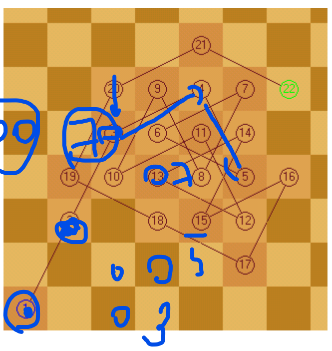


Fig.7a. Tour using heuristic *h1a*.

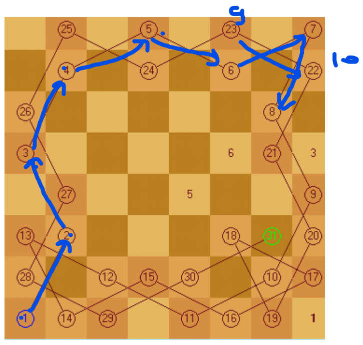


Fig.7b. Tour using heuristic *h1b*

After running the program using heuristic *h1a*, expanding the node with most options, ironically every single tour ends in a *dead-end* after about half the board is visited. Also in critical moves such as visiting a *corner square* before it gets *inaccessible*, the heuristic fails miserably since it obviously chooses other higher-valued squares. A visual inspection on Fig. 7a, where a tour sequence is generated using heuristic *h1a*, reveals that visits tend to quickly agglomerate in the center of the board, where squares have a higher *branching factor*, and hence more options to move. This eventually exhausts all natural paths connecting lower-valued squares from the four corner sections. Dead-ends become more likely.

Results for heuristic *h1b* turn out to be far more optimistic than heuristic *h1a*. As Fig. 7b depicts, by expanding the node with least options to move, visits tend to gather at the border and corner squares initially, finally working their way to the higher-valued center squares. Note that natural paths connecting the four corners are not broken as occurs with heuristic *h1a*. They are simply displaced to occupy the center squares where the knight has more freedom to move, and hence more chances to perform complete tours.

Performance of heuristic *h1b* is much better than expected. Using heuristic *h1b* as the *evaluation function* for a *best-first search* algorithm, complete tours for generic *NxN* boards can be obtained with ease, without backtracking. The algorithmic reasoning behind heuristics *h1a* and *h1b* seem to follow almost naturally after studying why *dead-ends* and *inaccessible* squares occur. It is therefore quite reasonable to expect that some other enthusiasts have thought about such practical methods to perform complete *knight tours*. Indeed, heuristic *h1b* presented in this paper is actually the same practical method introduced by *H. C. Warnsdorff* in 1823 [6].

## 8. Improving Heuristics

Heuristic *h1b* is not perfect, however. For some boards, in particular *N* = {41,52,59,60,66,74,79,87,88,94}, the choice of squares suggested by the heuristic sometimes prevent the algorithm from detecting *dead-ends*. Unfortunately, even backtracking and resuming the recursive search on the next valued squares do not seem to keep avoiding *dead-ends* and thus finding complete tours. Moreover, heuristic *h1b* seems to fail completely on very large boards, *N* > 300.

A closer inspection to heuristic *h1b* reveals a weakness. It does not address the case when two or more successor squares have the same lowest number of options. If there are two or more lowest-valued successor squares from the current node, the heuristic simply does not contemplate which square to visit next. In the current implementation, the search algorithm chooses the first tied square found, arbitrarily, based on the order in which the knight move *operators* were defined, as shown in Fig. 8. However, is there some way to determine which of the tied squares is better in terms of avoiding *dead-ends* and *inaccessible* squares? This is what the next heuristic, *h2*, attempts to address.

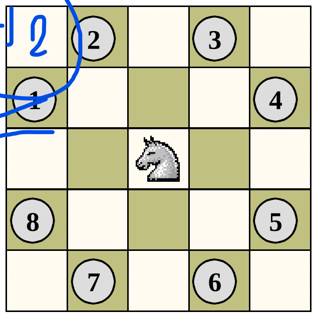
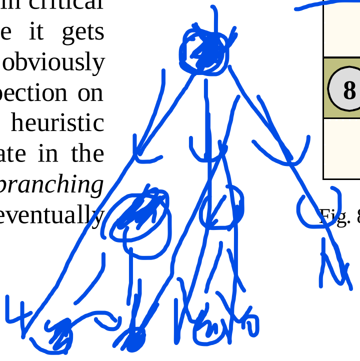


Fig. 8. The 8 knight move *operators*.



good state

From the tour in Fig. 7b, it was argued that visited squares that lie near the center of the board tend to break natural paths connecting squares from the four corners of the board, while visited squares that gather near the borders have more chance of avoiding *dead-ends* and *inaccessible* squares. Therefore, the knight would have more freedom to complete a tour on the last unvisited squares from the center of the board, as Fig. 7b shows. The same reasoning can be argued to break lowest-valued successor squares that are tied. Formally, heuristic *h2* will implement *h1b*, and in addition, in case of tie it will choose the lowest-valued square nearest to any of the four corners of the board, independently of *N*.

When improved heuristic *h2* is used, complete tours can now be constructed without backtracking in $O(N^2)$, including boards for which heuristic *h1b* previously failed, and in very large boards as well. The only limit for constructing complete knight tours using heuristic *h2* seems to be the available memory. Table 3 summarizes the results for heuristics *h1a*, *h1b*, and *h2* according to the dimension of the board, *N*.

Table 3. Results for heuristics *h1a*, *h1b*, and *h2* vs. *N*.

| *N* | **Heuristic *h1a*** | **Heuristic *h1b*** | **Heuristic *h2*** |
|---|---|---|---|
| 1 – 4 | Impossible | Impossible | Impossible |
| 5 – 40 | Fails | Succeeds | Succeeds |
| 41 | Fails | Fails | Succeeds |
| 42 – 51 | Fails | Succeeds | Succeeds |
| 52 | Fails | Fails | Succeeds |
| 53 – 58 | Fails | Succeeds | Succeeds |
| 59 – 60 | Fails | Fails | Succeeds |
| 61 – 65 | Fails | Succeeds | Succeeds |
| 66 | Fails | Fails | Succeeds |
| 67 – 73 | Fails | Succeeds | Succeeds |
| 74 | Fails | Fails | Succeeds |
| 75 – 78 | Fails | Succeeds | Succeeds |
| 79 | Fails | Fails | Succeeds |
| 80 – 86 | Fails | Succeeds | Succeeds |
| 87 – 88 | Fails | Fails | Succeeds |
| 89 – 93 | Fails | Succeeds | Succeeds |
| 94 | Fails | Fails | Succeeds |
| 95 - 100 | Fails | Succeeds | Succeeds |
| … | Fails | ? | Succeeds |
| > 300 | Fails | Fails | Succeeds |

Heuristic *h1a* always fails to obtain solutions for any *N* that is input. However, it is an interesting case of study since it shows that the most promising heuristic not always produces the expected result. In this case, it fails completely from narrowing the *search space* for the *knight tour* problem.

Heuristic *h1b* is able to find complete tours in most cases, but fails to obtain solutions on some particular boards, as Table 3 shows. There does not seem to be any particular pattern or correlation that describes the behavior of *N* when heuristic *h1b* fails. However, somewhere between 100 and 300, heuristic *h1b* seems to fail, invariably. Again, this presumes the *operator* order depicted in Fig. 8.

On the other hand, heuristic *h2* always succeeds for *N* between 5 and 300. The actual upper bound of *N*, for which heuristic *h2* fails, if it ever does, is *unknown*. However, for any arbitrarily selected *N* that is input, between 300 and 600, heuristic *h2* always returns a solution in $O(N^2)$.

## 9. Conclusions

In this paper, the uninformed *depth-first search* method was implemented to enumerate *knight tour* solutions. While this approach is suitable for small boards, the problem is obviously intractable for bigger boards, $N > 6, 7$, or larger, due to the exponential complexity associated not only with the *search space* of the problem (Table 1), but also one that is inherent to the *number of solutions* themselves.

Three heuristics were presented and discussed in detail to find any complete tour on any given *NxN* board.

Heuristic *h1a*, defined to prevent *dead-ends* specifically, clearly fails from doing so since it tends to gather *visited squares* near the center of the board, thus breaking the needed communication paths that link squares from the four corners of the board. This is independent of the board dimension, *N*.

Heuristic *h1b*, the *Warnsdorff* method, is a noticeable improvement over *h1a*, becoming able to perform complete tours on relatively large boards. However, the heuristic fails from detecting *dead-ends* on some particular boards, such as the ones depicted in Table 3. For these particular boards, the heuristic enters backtracking mode, after which it does not converge into any solution rapidly. The heuristic seems to fail completely on very large boards, about $N > 300$.

Finally, heuristic *h2* addresses a limitation of *h1b* by choosing the next lowest-valued square that is closer to any of the four corners of the board, in case of tie. Heuristic *h2* is able to find complete tours in $O(N^2)$, being available memory the only practical limitation.

## 10. References

[1] M. Loebbing and I. Wegener, “The Number of Knight's Tours Equals 33,439,123,484,294 - Counting with Binary Decision Diagrams,” Electronic Colloquium on Computational Complexity (ECCC), Volume 2, 1995.

[2] E. Mordecki, “On the number of Knight's tours,” Prepublicaciones de Matemática. Universidad de la República Oriental del Uruguay, December 27, 2001.

[3] G. Jelliss, “Knight’s Tour Notes,” http://www.ktn.freeuk.com/index.htm, as of Nov. 5, 2003.

[4] M. R. Keen, “The Knight’s Tour,” http://www.markkeen.com/knight.html, as of Nov. 5, 2003.

[5] S. Russell, P. Norvig, “Artificial Intelligence, a modern approach”, Prentice Hall, 1995.

[6] H. C. Warnsdorff, "Des Rösselsprungs einfachste und allgemeinste Lösung." Schmalkalden, 1823.

[7] G. Jelliss, "Early History of Knight’s Tours," http://www.ktn.freeuk.com/1a.htm, as of November 5, 2003.

[8] Mathworld, "Knight’s Tour," http://mathworld.wolfram.com/KnightsTour.html, as of November 5, 2003.