A Brute Force Approach to Solving the Knight's Tour Problem using Prolog

Robert Borrell
Wells Fargo Bank, Phoenix, Arizona, USA

Abstract - *The knight's tour problem is an old problem. It was investigated by Euler as well as a number of other researchers in recent years. This work describes the author's history with the problem, reviews the current work and the implementation. The work describes the algorithm, a depth first search with no bias or heuristic, and provides a discussion along with future work of this work. The work uses the Prolog language to develop a brute force algorithm in finding open circuit solutions to the knight's tour problem. The program is tested on various prolog implementations. An improved verify/2 predicate is discussed as it improved the overall execution in finding solutions. Future work may investigate the Warnsdorf's algorithm and magic squares.*

Keywords: Artificial Intelligence, Knight's Tour problem, Prolog

1 Introduction

The knight's tour problem on an 8x8 chess board is an old problem. From [2] the first recorded knight's tour was recorded in 840A.D. Figure1 illustrates the earliest known knight's tours. The objective of the problem is to find a path on the chess board in which the chess knight traverses each square only once. The first serious investigation was performed by Euler, who presented his analysis of his work to the Academy of Sciences in Berlin in 1759, but he published the work in 1766. Other well-known mathematicians worked on the problem include Taylor, de Moivre and Lagrange. Subsequent work has identified this problem as a Hamiltonian, a graph that has a path which traverses each node once.

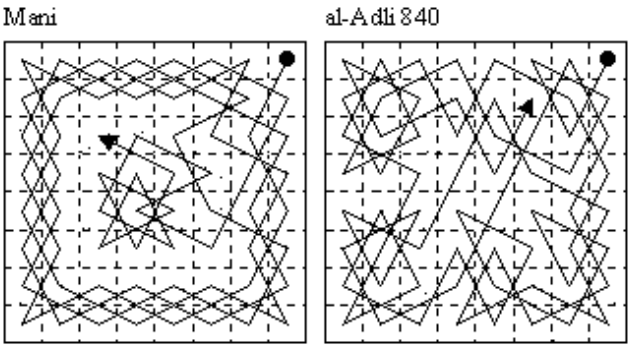

Figure 1: Two of the earliest known tours [2]

As well as interest in finding *open* tours, there is also interest in finding *closed* tours. A closed tour has the extra property that the 63rd move ends on a square that is a knight's move away from the start square, so that the knight could complete a Hamiltonian circuit with a 64th move. Closed tours are more difficult to find.

Other work has attempted to find efficient algorithms to resolve the knight's tour problem as well as counting techniques. For example, an article describes genetic algorithms using the Ant Colonization Optimization (ACO) [2] to find solutions to the knight's tour. The authors discussed how a genetic algorithm was used to find knight's tours. Unlike those authors, the best they could obtain by was 207 knight's tours per million attempts. The authors from [2] elected an Ant Colonization Optimization approach to finding knight's tours. Their work compared their approach against a depth first algorithm and the genetic algorithm. The result improved the number of knight's tour found.

Another approach to the knight's tour was done with a neural network by Parberry [6] in which he compares a neural network method against standard algorithms. Parberry was clarifying some points by some other authors regarding the effectiveness of neural networks to find knight's tours. His conclusions are the neural network could work for knight's tour using 26x26 boards. However, standard algorithmic approaches such as Euler and Warnsdorf heuristics as well as divide and conquer approach surpass that very easily by finding knight's tours using 78x78 boards.

Articles for counting techniques to the number of knight's tours were researched by Shroer and Wegener [7] and Lobbing and Wegener [3]. According to [3], there are over 33 quadrillion knight's tours in a 8x8 chess board. The technique uses the binary decision diagrams (BDDs) to arrive the solution.

Logic Programming has its roots in Logic. Logic programs consist of logical formulas, and logic computation is the process of deduction or proof construction. This makes logic programming fundamentally different from most other programming languages. The main difference between logic programming and conventional programming languages is the *declarative* nature of logic.

The work is unique as other researchers have not used Logic Programming and Prolog to find solutions to the knight's tour problem. The remainder of the document will cover the current work, review the implementation, a brief discussion, and end with future work.

2 Current Work

The development of logic programming began in the 1970s with the pioneering work of Kowalski from the University of Edinburgh. A logic program consists of a set of logic formulas called clauses. Clauses have a head and body. The body can be more clauses as well. Other components are terms, variables, and functors. As result of this declarative approach, prolog was created. The programming language Prolog was developed by Alain Colmerauer in 1972. Later in 1979, Warren, Pereira, and Pereira developed the Edinburgh syntax of Prolog. Finally, today the standard is the ISO Prolog standard used by all Prolog interpreters and compilers. Although other procedural languages can solve this problem, the declarative nature of prolog was used in this work. This section will discuss the history, the hardware platforms tested, the first solution found, the software used, and the program enhancement.

This effort began in the late 1980s when an article influenced the decision to research on solving the knight's tour problem. At the time the software available for IBM PC running MSDOS 6.1 was Turbo Prolog v2.0 [8 and 9]. The research began on developing a knight tour's problem algorithm using Prolog language using the Turbo Prolog Inference Engine. In addition, after the acquisition of the classic Clocksin and Mellish [1] prolog book, it provided the direction in developing the approach.

In 1995, when the internet was accessible, the use of archie was used to find a prolog interpreter, and the source code for the UNSW Prolog Version 4 interpreter was found. The UNSW Prolog interpreter was compiled on a SCO UNIX platform. The effort to resolve a solution resumed. Upon completion of the prolog implementation, a knight's tour solution was found using 45 minutes of computing time. Later that year, the UNSW Prolog interpreter was compiled using Gnu C compiler on SUN Solaris server on a Sparc Center 2000. Since the processor speed was double, the solution was found in 22 minutes of computing time (See Table 1).

Table 1: Time to first solution

Year	Platform	Time to First Solution
1995	SCO UNIX 33MHz	45min
1995	Sparc Center 2000 66MHz	22min
2008	PC 3GHz	35s

The current work began in 2007. The experiments were done in an Intel platform running 3.0 GHz single core processor with 1.5 GB of RAM running Windows XP. The software for the experiments was a guest VMware Linux server running Red Hat Linux AS4.5. Initially, the work was performed with UNSW Prolog interpreter. After validating some experiments on UNSW Prolog, the GNU Prolog was used. However, since the GNU Prolog was not being maintained, the SWI Prolog interpreter and compiler were used.

In Figure 2, the first solution is identified by the program. The starting square is [1,8] and ends at [8,2]. This is an open circuit solution. Since the chess board is symmetrical, the chess board can be rotated with a similar solution path in solving the knight's tour. Because of the symmetry, the starting square can be [8,8], [8,1], and [1,1] with a similar path. Additional sample solutions can be found in Appendix A.

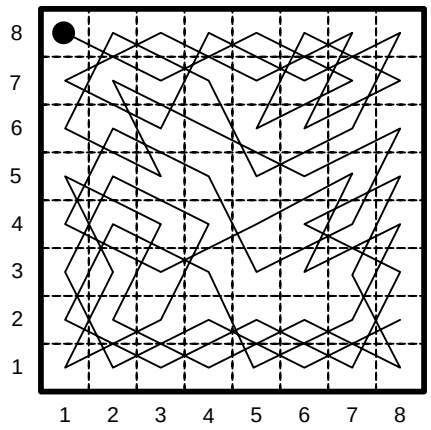

Figure 2: First solution

First, using the UNSW Prolog interpreter [10], the original prolog code from 1995 was tested unmodified. The experiments resolved the first solution in about 35 seconds of computing time, which was much faster than the original test in 1995 of 45 minutes of computing time. Next the GNU Prolog compiler was tested. The code was modified because the member, the length, and the reverse predicates were built-in. The experiments ended with the same result. Finally, the same code was tested with SWI Prolog with same time result of 35 seconds (see Table 2).

Table 2: Time to first solution in 2008

Prolog Interpreter	Time to First Solution
UNSW Prolog	35s
GNU Prolog	35s
SWI Prolog	35s
SWI Prolog with modified verify/2 predicate	7s

Recently, the LISP implementation of Othello [5], in which the author used a 10x10 array and filled the border cells with a value of -1. This approach to check the border squares influenced an improvement to the Prolog code. Because of this influence, the code for the verify/2 predicate was rewritten to test if the newly generated coordinates landed on the border squares instead of checking the internal squares. This improved implementation reduced the time of finding the first solution from 35 seconds to less than seven seconds of computing time. The first 14 solutions were found in with an average of 2m7s of computing time. This was a significant improvement.

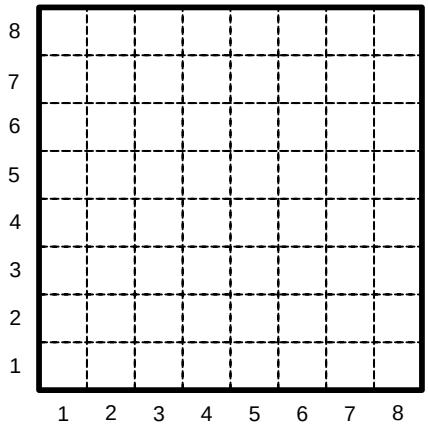

Figure 3: Coordinates of the chess board

3 Implementation

In this section the representation of the objects are discussed as well as the code implementation.

The implementation of the path co-ordinates uses international chess notation for identifying the location of the knight on the board. The left most file is designated as 1, and numbered until it ends at 8. The ranks are number from 1 to 8. The lower left corner square is designated as [1,1] as in Figure 3.

The program has three parts. The first component is the high level clause. Second component is the depth first search clause. Next component is the knight move generator. The last component is the verification clauses. The basic code implementation of the knight's tour code is listed in Figure 4.

```
knight_tour(Start, Solution) <-
   go(Start, Start, Path),
   length(Len, Path),
   Len == 64,
   reverse(Solution).

go(Current, Path, [Current|Path]).

go(Current, Path, Z) <-
   knight_move(Current, Next),
   not(member(Next, Current)),
   go(Next, [Current|Path], Z).

knight move([R,C], [X,Y]) <-
   knight move generator,
   verify([X, Y]).

verify(X,Y) <- check boarder squares.
verify(_,_).
```

Figure 4: The knight's tour program.

The knight_tour/2 predicate calls the go/3 predicate, determines the solutions length, checks if it is 64 in length, and the reverse/1 predicate reverses the solution path. The go/3 predicate calls the knight moves/2, checks if the move is in the current solution path, then recursively calls the go/3 predicate until no moves can be generated. The knight_move/2 generator consists of the eight possible moves of the knight (see Figure 5).

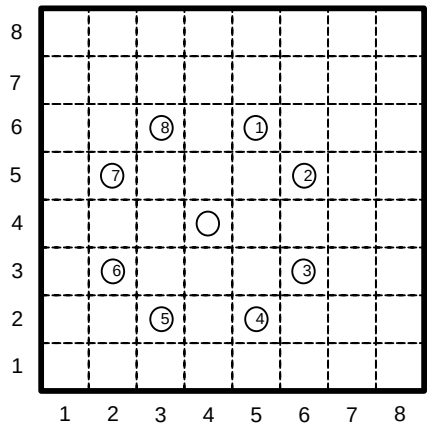

Figure 5: The Legal Knight moves.

The generated move is checked by the verify/2 predicate. If the move falls on the border, then the move has failed and another move is selected.

4 Discussion

In Luger and Stubblefield [4], the knight's tour problem is used as pedagogical device. The authors used a modified 3x3 board to illustrate search in predicate calculus and in production systems. Then they develop Prolog code for the modified problem. The authors develop a frame work for the 8x8 board. However, this work differs from Luger's proposed implementation since there open and closed solutions and the boundary checker, the verify/2 predicate, differs.

There are two types of knight's tour solutions - first, the closed circuit solution is where the path leads to square to the starting square on the 64th move. The open circuit solution is where the path ends on any square and not end near the starting square. The algorithm consists of depth first search algorithm and the knight move generator, which consists of the eight possible moves. Because there is no goal state, the algorithm finds open circuit solutions by searching the state space of the knight's tour with no bias or heuristics. In other words, the algorithm uses a brute force approach.

Initially, the verify/2 predicate was defined in Figure 6.

```
verify(X,Y) :- X > 0, X < 9, Y > 0, Y < 9.
```

Figure 6: The original verify/2 predicate.

This implementation was discovered as inefficient because for each verify operation requires the test of four relations to validate the move. The border values are in the set {-1, 0, 9, 10} for an 8x8 board. Thus, if the file or rank falls in the above set, then the cut prevents further checking, and the move is returned with the fail/0 predicate. In Figure 7, the first eight verify/2 predicates handle the case for the

above condition. The first four handle the case for the file at the border squares, and the latter four for the ranks. The last verify/2 predicate always succeeds for all internal squares.

```
verify(-1, _) :- !, fail.
verify(0, _) :- !, fail.
verify(9, _) :- !, fail.
verify(10, _) :- !, fail.
verify(_, -1) :- !, fail.
verify(_, 0) :- !, fail.
verify(_, 9) :- !, fail.
verify(_, 10) :- !, fail.
verify(_,_).
```

Figure 7: An improved verify/2 predicate.

This method only checks the required squares. The solution is influenced by the Othello program in [5] when the author uses an array larger than 8x8 board and placed -1 value in those cells. The experiments demonstrate this version of the verify/2 predicate improved the execution of the algorithm. The first 14 solutions starting at the [1,8] as in Figure 2 are computed with 2 minutes and 7 seconds of computing time.

The nature of the chess board is symmetrical. The board can be rotated so that one solution can become four solutions. For example, in Figure 2, the first solution found by the prolog program can be used at the other three corners using the same movement becomes three additional solutions to the knight's tour. As discussed in the introduction, other research has concluded there are over 33 quadrillion solutions to the knight's tour problem using an 8x8 board. To find all 33 quadrillion knight's tours solution paths would take years to complete.

5 Further Work

The experiments above suggest the algorithm was stable as well as the new implementation of the verify predicate. Because this is a brute force method, future work may include analysis and implementation of the Warnsdorf's heuristic. Also, since the solution path is a list, other potential approaches such as binary trees will be investigated for performance enhancements to the program. Finally, future work may investigate algorithms for magic square solutions to improve and to find solutions quickly. Finally, the aforementioned works in the introduction suggest there are improved solutions.

6 Acknowledgements

I want to thank my family for their support in this endeavor.

7 References

[1] Clocksin, W.F. and Mellish, C.S., *Programming in PROLOG*. Springer-Verlag. 1984.

[2] Hingston, P. and Kendall, G., Ant Colonies Discover Knight's Tours in *AI 2004: Advances in Artificial Intelligence: The 18th Australian Joint Conference on Artificial Intelligence*. Springer. 2004.

[3] Lobbing, M. and Wegener, I., The Number of Knight's Tours Equals 33,439,123,484,294 – Counting with Binary Decision Diagrams, *Electronic Journal of Combinatorics.* Vol 3. p.5. 1996.

[4] Luger, G. F. and Stubblefield, W.A., *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 2nd ed. Benjamin/Cummings Redwood City CA. 1993.

[5] Norvig, P., *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann San Francisco CA. 1992.

[6] Parberry, I., Scalability of a neural network for the knight's tour problem, *Neurocomputing*. 1996.

[7] Schroer, O. and Wegener, I. The theory of zero-suppressed BDDs and the number of knights tours in *IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*. 1994.

[8] Turbo Prolog v2.0 User's Guide, Borland International, Scotts Valley CA. 1988.

[9] Turbo Prolog v2.0 Reference Guide, Borland International, Scotts Valley CA. 1988.

[10] UNSW Prolog Version 4 Manual, University of New South Wales. 1983.

Appendix A – A sample of solutions found.