IM1202 Knight Tour Project Report

Stijn de Preter (852726504)
Arjan Broer (850166428)

November 30, 2025

Abstract

This document presents a comprehensive overview of the research conducted for the IM1202 project. It includes an introduction to the problem, methodology, related work, experiments, results, and conclusions. The focus of the project is on solving the Knight Tour problem using Answer Set Programming (ASP). Key findings and insights are discussed throughout the document.

Contents

1 Introduction

The Knight's Tour problem is a classic problem in computer science. It involves moving a knight piece on a chessboard such that it visits every square exactly once. The challenge lies in the unique movement capabilities of the knight, which moves in an "L" shape: two squares in one direction and then one square perpendicular, or one square in one direction and then two squares perpendicular. More formally the knight's tour can be defined as a graph for an $n \times n$ chessboard defined by $G = (V, E)$ where $V = (i, j)|1 \leq i, j \leq n$ and $E = ((i, j), (k, l))||i - k|, |j - l| = 1, 2$. [5]

Research questions that we discuss in this report are:

- How can the Knight's Tour problem be logically modeled so that it is solvable with Answer Set Programming (ASP)?
- Which different logical models can be used to formulate the problem?
- How can heuristics be applied to improve the efficiency of solving the Knight's Tour problem with Answer Set Programming (ASP)?
- What other technique (other than heuristics) can be used to improve the performance of the algorithms?

To limit the scope of the project and allow for comparisons, we will focus on finding five results for the problem on a standard board (8x8 squares). We look at Open tours meaning: The knight visits every square exactly once, but the ending square is possibly not a knight's move away from the starting square. If no performance difference is observed, the board can be expanded until a noticeable difference in performance is achieved. The starting point will also be fixed at a corner of the chessboard.

2 Methodology

There are several methods to solve this problem. The following two methods are available:

- Method 1: A sequence is used. This means that we search for the next square of a chessboard from a specific square.
- Method 2: "Connected" fields are searched for all squares. This searches for combinations of all the fields where they are connected.

Other methods may also exist.

3 Related Work

In [5] the author discusses an efficient algorithm for finding knight's tours on larger chessboard. The scalability of the algorithm is proven by showing that boards can be split up in smaller boards. Following a structure of knight steps in the corners creates a structure that enables the boards to be connected. Solving the problem for the smaller boards and copying this solution to stitch together the larger board allows for an efficient solution. This algorithm goes beyond the scope of this project because this report limits the board size to 8×8. The algorithm is interesting for future research on larger boards. The

article [1] describes the open and closed knight's tour problem and provides a solutions in a CLINGO listings. The listing for the open knight's tour problem is designed by defining the $n \times m$ chessboard, the knights possible steps and the constraint of visiting a square only once. The solution to be found should be exactly $n \times m$ steps long following the set rules. This will guarantee a solution to the open knight's tour problem.

4 Experiments and Results

4.1 Methods to solve the knight problem

There are several methods to solve the knight problem. In this section two methods are discussed.

4.1.1 Method 1

This approach models the problem as a sequence of steps (or time points). The first step starts at position (x=1,y=1). The second step might be at (x=2,y=3). Each such combination of coordinates and step number is called a visit: at step "z", the knight occupies square (x,y). Finally, a constraint ensures that every square on the board is visited exactly once within a total number of steps equal to the number of squares.

```
% Board size
#const n = 8.

% Board
square(1..n, 1..n).

% Knight moves
move(X1,Y1,X2,Y2) :- square(X1,Y1), square(X2,Y2), 1 = |X1-X2|, 2 = |Y1-Y2|.
move(X1,Y1,X2,Y2) :- square(X1,Y1), square(X2,Y2), 2 = |X1-X2|, 1 = |Y1-Y2|.

% Steps
step(1..n*n).

% Initial position at step 1
visit(1,1,1).

% For each step > 1, the knight must move from the previous square to a new square via a legal knight move.
% The choice rule ensures exactly one move is made at each step.
{ visit(X2,Y2,Z2) : visit(X1,Y1,Z1), move(X1,Y1,X2,Y2), Z2 = Z1 + 1 } = 1 :- step(Z2), Z2 > 1.

% Constraint: no square may be visited more than once.
:- visit(X,Y,Z1), visit(X,Y,Z2), Z1 != Z2.


#show visit/3.
```

Listing 1: Methode 1

4.1.2 Method 2

The second method is based on reachability. For each square on the board, a possible predecessor square is selected (except for the starting square). The final solution only includes configurations that satisfy two conditions:

- Every square must be reachable from the start (no isolated loops or disconnected regions, such as two squares pointing only to each other).
- Each square can have at most one outgoing edge, ensuring the structure forms a single continuous path rather than branching.

This method differs most from method 1 in that no time/step is kept track of. While method 2 works with the concept of reachability [4] .

```
1
2  #const n = 8.
3
4  % Board
5  square(1..n, 1..n).
6
7  % Knight moves
8  knight_move(X1,Y1,X2,Y2) :- square(X1,Y1), square(X2,Y2), 1 = |X1-
       X2|, 2 = |Y1-Y2|.
9  knight_move(X1,Y1,X2,Y2) :- square(X1,Y1), square(X2,Y2), 2 = |X1-
       X2|, 1 = |Y1-Y2|.
10
11 % Fix starting square
12 start(1,1).
13
14 % At most one incoming knight move per square except for the start
15 1 { route(X1,Y1,X2,Y2) : knight_move(X1,Y1,X2,Y2) } 1 :- square(X2
       ,Y2), not start(X2,Y2).
16
17
18 % Each square must be reachable from the start
19 reachable(X,Y) :- start(X0,Y0), route(X0,Y0,X,Y).
20 reachable(X,Y) :- reachable(X1,Y1), route(X1,Y1,X,Y), not start(X,
       Y).
21
22 % each square must be reachable (start not included)
23 :- A < n*n-1, A = #count { X1,Y1 : reachable(X1,Y1) }.
24
25 % there must be n*n-1 routes (edges) in total and from each square
       exactly one outgoing edge
26 :- N < n*n-1, N = #count { X1,Y1 : route(X1,Y1,_,_) }.
27
28 #show route/4.
```

Listing 2: Methode 2

4.2 Performance of the methods

Both methods are asked to generate five answers. The first method takes 31.880 seconds to do this, while the second method takes 0.342 seconds. To better test the performance of the second model, the second method is asked to provide 10000 answers, which takes 3.688 seconds.

To better understand performance, we look at the statistics. Although we cannot find any good documentation of CLINGO's statistics, we can make some assumptions.

- Choices are the number of branches the algorithm had to make.
- Conflicts would be the number of times the solver should backtrack.

The following results come from the statistics of both methods (see Table 1).

Table 1: Performance Comparison of Methods

Method	**Choices**	**Conflicts**	**Time (s)**
Method 1	1,546,701	370,082	31.880
Method 2	27,908	23,864	0.342

4.3 Heuristics

Heuristics have been explored to potentially solve the problem faster. There is a well-known heuristic for the knight tour: Warnsdorff's algorithm [3]. This heuristic should ensure that the algorithm first searches for combinations, whereby at each step the step is chosen from which the fewest possible future steps are possible. The algorithm does this by determining the next square based on its accessibility. For example, a square at the edge of the board, and therefore less accessible in the future, will be visited more quickly than a square in the center of the board that can still be reached from multiple squares.

The heuristic was implemented in method 1 as follows:

```
1
2 % Define valid target squares for heuristic
3 valid_target(X3,Y3,Z3,Z2) :- move(X2,Y2,X3,Y3), visit(X2,Y2,Z2),
     visit(X3,Y3,Z3), Z3 > Z2.
4 heuristic(H,X2,Y2,Z2) :- visit(X2,Y2,Z2), H = #count { X3,Y3 :
     valid_target(X3,Y3,Z3,Z2)}.
5
6 #heuristic visit(X2,Y2,Z2) : heuristic(H,X2,Y2,Z2). [-H@8, true]
```

Listing 3: Methode 1 with Warnsdorff's algorithm

The code above adds to method 1. First, we need to determine how many more steps can be taken from a given field. This heuristic will then be used in heuristic directives.

The solver takes longer (89.270s) when the heuristic is implemented this way, even though the statistics (see Table 2) show less backtracking and fewer choices being reviewed. This might be explained by the fact that the heuristic needs to be recalculated repeatedly. As a result, each node will take longer. This leads to a better choice of nodes, but at the cost of additional computations (for the heuristic).

Table 2: Performance with Heuristic Implementation

Configuration	**Choices**	**Conflicts**	**Time (s)**
Heuristic-based	579,352	100,768	89.270

4.4 Adding rules for improvement of the performance

In method 1, there are two ways to define that a field may only be accessed once. These are listed in Listing 4 and Listing 5.

```
1        :- visit(X,Y,Z1), visit(X,Y,Z2), Z1 != Z2.
```

Listing 4: Methode 1 - Constrain option 1

```
1        :- square(X,Y), not visit(X,Y,_).
```

Listing 5: Methode 1 - Constrain option 2

The performance of both constraints was tested. The performance of adding both constraints to the program was also tested. All the results are shown in Table 3.

Table 3: Performance with Method Implementation

Configuration	**Choices**	**Conflicts**	**Time (s)**
Method1 - Option 1	1,546,701	370,082	36.672
Method1 - Option 2	7,505,290	1,024,750	193.595
Method1 - Combined Options	124,376	31,424	1.473

"Method 1 - Option 1" is the same program as in Listing 1 and therefore also Table 1. However, we see different values here. This is because this code is run once, and that is the result in Table 1. The value can vary from computer to computer and can also change over time. The important thing is that the order of magnitude is considered, rather than the exact result.

5 Discussion

5.1 Methods

Two different model methods have been found for solving the Knight's Tour problem. Other methods may exist. The first method involved maintaining an index to navigate through the steps. The second method relied on reachability. The first method (As set out in Listing 1) was less performing than the second method (As set out in Listing 2). The difference in performance can be explained by the number of choices and conflicts that arise during the solving process. Method 2 showed a significantly lower number of choices and conflicts compared to Method 1.

5.2 Heuristics

A heuristic can be applied to method 1. This is according to the Warnsdorff's algorithm [3]. This heuristic itself must be calculated, making solutions easier to find and requiring less backtracking during the solving process. However, the heuristic itself takes a lot of time to calculate. This ultimately leads to a less performant program.

5.3 Adding rules for improvement of the performance

The performance of method 1 (As set out in Listing 1) is clearly improved by adding the code for Listing 5, although this doesn't change the program's output. The exact reason is difficult to determine because grounding and solving programs can be very complex. However, presumably, many more choices can be eliminated by the presence of the two rules (they complement each other well), resulting in a very efficient program.

It's also striking to see in Table 3 that only adding the code of Listing 5 and removing Listing 4 from the code results in a very inefficient program, with many more choices and conflicts.

5.4 Multi shot solving

Multi-shot solving was explored to see if it could improve performance. The idea behind multi-shot solving is to break the problem into smaller subproblems that can be solved sequentially. For the knight's tour, this could involve solving for smaller sections of the board and then combining these solutions. This approach for the Knight's tour problem is described in detail in [5]. The technique of multi-shot solving is described in [2]. A base program is created that defines the basic rules for the problem to solve. Then an additional program is created that adds specific constraints or goals for each subproblem. When this approach is applied to the knight's tour problem, the challenge lies in finding and defining appropriate subproblems that can be solved independently while still leading to a valid overall solution. Although the performance of solving the smaller subproblems will be faster, the number of found solutions will be less and thus incomplete compared to single shot solving. For an implementation example, a board is defined of size 10×10. This board is then split into four 5×5 sub-boards. The size is chosen because this minimal board size for an open knight's tour is 5×5. Open knight's tours can only be found on square boards of $n \times n$ with $n \geq 5$ as described in [5].

6 Conclusion

The Knight's Tour problem can be effectively modeled using Answer Set Programming (ASP) through different logical models. Two distinct methods were explored: one based on sequential steps and another based on reachability. The reachability-based method demonstrated superior performance, likely due to its reduced number of choices and conflicts during the solving process. These choices and conflicts are critical factors influencing the efficiency of ASP solvers.

Heuristics, such as Warnsdorff's algorithm, can be integrated into the ASP models to guide the search process. However, while heuristics can reduce backtracking and improve the quality of choices, they may introduce additional computational overhead, potentially leading to longer overall solving times. The researched methods in this report showed

a significant performance decrease when heuristics were applied due to the additional calculations needed.

Multi-shot solving was found as related word to this project, but was not implemented due to time constraints. The technique holds promise for breaking down the Knight's Tour problem into smaller, more manageable subproblems, which could enhance solving efficiency in future research. This technique could be particularly beneficial for larger chessboards, where the complexity of the problem increases significantly. For smaller boards this method can not be applied because a board has a minimum size of 5×5 to find an open knight's tour. Consequence of applying multi-shot solving on smaller boards would be that not all solutions are found. The path of the knight is limited by the sub-board solutions and thus can not explore all possible paths.

References

[1] Edeilson Milhomem da Silva Carvallho, Ary Henrique de Oliveira, Glenda Michele Botelho, and Glêndara Aparecida de Souza Martins. Two classic chess problems solved by answer set programming.

[2] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811, 2017.

[3] Kartik. Warnsdorff's algorithm for knight's tour problem, July 2025. Accessed: 2025-10-21.

[4] Liu Liu and Miroslaw Truszczynski. Encoding selection for solving hamiltonian cycle problems with asp. *Electronic Proceedings in Theoretical Computer Science*, 306:302–308, September 2019.

[5] Ian Parberry. An efficient algorithm for the knight's tour problem. *Discrete Applied Mathematics*, 73(3):251–260, 1997.