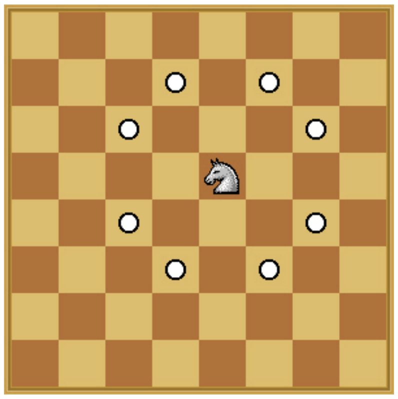
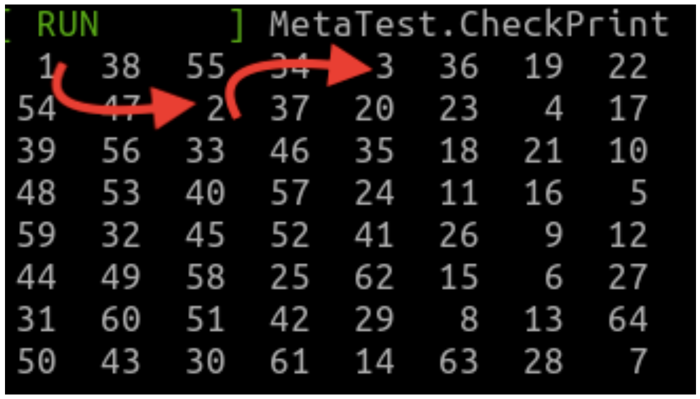
# Solving the Knight’s Tour using Warnsdorff’s Algorithm

Josh Schmitz

All of the work is mine but it should be acknowledged that this project was created as a final project for CPSC-450 at Gonzaga University. Thanks to Dr. Bowers, the professor of the course, for giving me the knowledge necessary for this project.





```
[ RUN      ] MetaTest.CheckPrint
 1  38  55  34   3  36  19  22
54  47   2  37  20  23   4  17
39  56  33  46  35  18  21  10
48  53  40  57  24  11  16   5
59  32  45  52  41  26   9  12
44  49  58  25  62  15   6  27
31  60  51  42  29   8  13  64
50  43  30  61  14  63  28   7
```

## Summary

For my project I implemented Warnsdorff's algorithm to solve the Knight's Tour problem. I also implemented a less efficient backtracking algorithm to accomplish the same task. My goal was to compare the performance of the two algorithms over different sized boards. My results were very clear that Warnsdorff's algorithm is drastically more efficient and, in fact, the only practical implementation.

## 1. ALGORITHM SELECTED

The Knights Tour is a path that a knight can take on a chess board in order to hit every single square exactly once. It is an example of a Hamiltonian path on a graph G(V, E) where V is all the squares on a chess board and every vertex has an edge to all the vertices corresponding to the edges that are reachable by a knight. I implemented both Warnsdorff's algorithm and a more simple backtracking algorithm to solve this problem. Warnsdorff's algorithm is predicated on the aptly named Warnsdorff's Rule: a valid tour will be found by always moving to the square with the minimum degree. In this case the degree of a square is the number of positions a knight can move to from that square.

Here is the Warnsdorff's algorithm pseudocode I designed my C++ implementation around [1]:

```
1. Set P to be a random initial position on the board
2. Mark the board at P with the move number "1"
3. Do following for each move number from 2 to the number of squares on the board
        1. let S be the set of positions accessible from P.
        2. Set P to be the position in S with minimum accessibility
        3. Mark the board at P with the current move number
4. Return the marked board — each square will be marked with the move number on which it is visited.
```

Hamiltonian paths in general are NP-hard, but certain graph patterns such as the one in the Knight's Tour can be solved in linear time. Warnsdorff's algorithm has a time complexity of O(1). [2]

## 2. BASE IMPLEMENTATION

For my project I chose to implement a recursive version of Warnsdorff's algorithm in C++. Instead of making a graph representing the chess board and a knight's move I used a 2d vector of size n*n to represent a chessboard with n squares in each direction. The integer at each index of the 2d vector represents the position of the path it is. For example, the initial position of the knight will be labeled 1, the position after the first move will be labeled 2, and so on until the last position is labeled n*n.

I had three main sections of tests for my code. The tests labeled meta tests were used to visually confirm the validity of test helper functions such as printBoard() and isValid(). The tests labeled Warnsdorff's tests ensure that the implementation is working for boards of varying sizes and with different initial positions. The test labeled backtrack tests perform the same sort of functionality validation but for the backtracking algorithm. Through testing I discovered certain limitations of the algorithms that I was unaware of. For example, I found out that Warnsdorff's algorithm only works with an initial position with a minimal number of valid moves and that the backtracking algorithm is so slow that it is entirely impractical for any board larger than 8*8.

## 3. EXTENDED IMPLEMENTATION

My extended implementation included a couple of performance evaluation tests. I made a CMakeLists file to compile my code and build executables for the unit tests and performance tests.

The goal of the performance tests was to compare the efficiency of Warnsdorff's algorithm and the backtracking algorithm. My plan was to run the algorithms on boards on varying sizes and making a table to compare the time it takes the algorithms to run for the different sized boards. However, while doing the unit tests I discovered that my implementation of the backtracking algorithm doesn't work for board sizes larger than 8*8. I left it running for about half an hour on a 9*9 board and it still hadn't solved it. This makes since as the backtracking algorithm requires a huge number  more operations for even just slightly larger boards. As such, I was unable to include a large performance comparison for the two algorithms. However, I have one performance evaluation for both algorithms that only uses relatively small boards and I made another performance evaluation of just Warnsdorff's algorithm that goes to very large sized boards. You can see that it is still way faster to use Warnsdorff's on a huge board than it is to run the backtracking algorithm on much smaller boards.

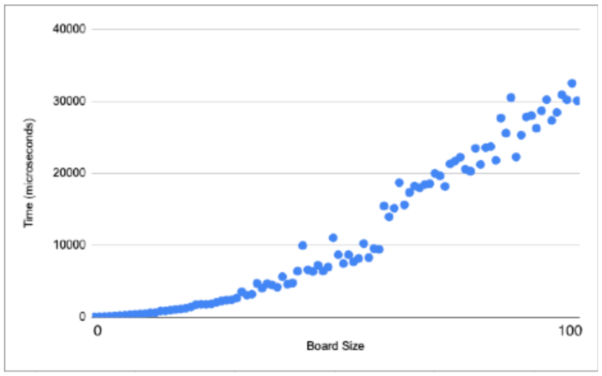
**Table 1. Performance evaluation for Warnsdorff's and the backtracking algorithm**

| Board Size (n*n) | Warnsdorff's (time to complete in microseconds) | Backtracking (time to complete in microseconds) |
|---|---|---|
| 5 | 40 | 8 |
| 6 | 196 | 740954 |
| 7 | 95 | 129180 |
| 8 | 130 | 829404 |

This table shows the output of the first performance evaluation that compares the two algorithms for relatively small sized boards. You can see that the Warnsdorff's algorithm implementation is way faster than the backtracking implementation.

**Figure 1. Time Warnsdorff's algorithm takes to complete a valid path for varying board sizes**

Here is a graph showing the amount of time Warndorff's algorithm takes to solve boards ranging from 5*5 to 100*100. You



can see that it still takes less time for Warndorff's algorithm to solve a 100*100 board than it took the backtracking algorithm to solve a 8*8 board.

## 4. BUILD AND RUN INSTRUCTIONS

I implemented a configuration file CMake so the build and run instructions are simple. I believe there should already be executables in the src directory, but if not you can build them by moving into the src directory and typing `cmake CMakeList.txt` into the command line.

There are two executables: *project_perf* and *project_tests.*

*project_perf* runs the performance evaluation. It takes a single command line argument which specifies which test to run. The valid options are 1 or 2, so you can type either `./project_perf 1` or `./project_perf 2`. The first will evaluate both Warnsdorff's and the backtracking algorithm over a small number of board sizes. The second will evaluate just Warnsdorff's algorithm but over much larger boards.

*project_tests* runs the unit tests for the project. These include a variety of tests to ensure the functionality of both Warnsdorff's and the backtracking algorithm.

## 5. REFLECTION

This ended up being one of the more interesting coding projects I have done so far at Gonzaga. I learned a lot about both the Knight's Tour and Warnsdorff's algorithm. It wasn't a part of my initial plan to include an implementation of a backtracking algorithm for the Knight's Tour, but I am really glad I did. This algorithm ended up being slightly harder for my to implement because: a. Warnsdorff's algorithm turned out to be rather simple and b. I designed my own backtracking algorithm instead of using pseudocode I found. By doing both the backtracking algorithm and Warnsdorff's algorithm I was able to understand the problem at a deeper level and get an appreciation for how people come up with algorithms.

Doing both algorithms also helped tie the project back in to everything else I learned in class because it allowed me to see two different algorithm designs for the same problem. I'm not sure if Warnsdorff's algorithm fits cleanly into one of the algorithm patterns we discussed, but the idea of a backtracking algorithm was certainly something we learned. Additionally, running performance evaluations on two different approaches really helped me gain an appreciation for the importance of knowing an algorithms complexity. While working on the project I discovered that the Knight's Tour is actually a real-world(ish) example of a problem we already learned about: Hamiltonian Paths. It was interesting to see how such different algorithms operating over different data structures were able to accomplish the same task.

In the future I think it would be fun to try to come up with and implement my own algorithm thats even faster than Warnsdorff's algorithm. My idea is to combine a dynamic approach that takes advantage of Warnsdorff's rule to solve the Knight's Tour. The inefficiency I see in Warnsdorff's is that the algorithm has to calculate the degree of up to eight different squares each time the knight moves. This calculation could be replaced by initializing all the squares with their degree and decrementing it each time one of it's 'adjacent' squares gets used. I'm not sure if this would improve run times much, but it would be interesting to see!

## 6. RESOURCES

Describe and cite all of the resources you used in your project. Be sure to add a reference for each citation. Be sure to explain the citation/resource and how you used it. (i.e., this should just be a list of things you used, and instead, needs to describe what it was that was used and how).

1. Geeks for Geeks [1]: I used the this site to gain a better understanding of Warnsdorff's algorithm. I used the pseudocode to design my C++ implementation.

2. Wikipedia (Knight's tour) [2]: I used the wikipedia page on the Knight's Tour to get a better understanding of the problem and possible solutions as well as learn the time complexity of the algorithm.

## 7. REFERENCES

1. https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/
2. https://en.wikipedia.org/wiki/Knight%27s_tour