Solving the Knight’s Tour Problem with Backtracking, Heuristics, and Divide and Conquer Algorithms.

Evan Dreher, Christian Previtali, Christina Kasunick

**Abstract:    
The knight’s tour is a series of moves made by the knight on a chessboard such that it visits each square on the board exactly once. This paper discusses the comparison of solving the knight’s tour using a brute force depth-first recursive backtracking algorithm with a divide and conquer algorithm.**

**Keywords: Knight’s Tour, Backtracking, Divide and Conquer, Warnsdorff’s rule,**

I. Introduction

The knight’s tour problem (KTP) starts with a knight on the corner of a chess board, typically a square one but the problem may be extended to rectangular boards. From that starting corner, the knight must jump to every square on the board without ever landing on the same square twice. The problem is also typically split into two types, the closed knight’s tour (CKT) and open knight’s tour (OKT). A CKT is one where after hopping to every square on the board exactly once, the knight can hop from the square they ended on to the square they began on creating a big loop. An OKT is simply a knight’s tour that is not closed. It is a specific version of the Hamiltonian-cycle problem in which you attempt to find a path on a graph that visits every vertex exactly once. The KTP has existed for hundreds of years and has been studied by many great mathematicians such as Leonhard Euler. From these hundreds of years of research and calculations, we have a perfectly precise idea of which boards are and are not solvable. For example, in 1991, Allen Schwenk proved that a rectangular m × n board has a CKT unless m and n are both odd; m or n equals 2, 3, or 4; or m or n equals 3 and the other variable equals 4, 6, or 8 [1]. With Schwenk’s proof we can know that something as large as a 10,000 × 10,000 board has a CKT on it but knowing that a path exists is trivial, the real problem is finding that path which requires a lot more time and computing power.

II. Backtracking

The slow and simple backtracking algorithm uses a brute force depth-first recursive search of every move that the knight can make until eventually a complete OKT is found. It works quickly enough on small boards but begins to fall short when it tries to solve even a standard 8 × 8 board. The algorithm uses recursion so to analyze the runtime complexity we must set up a recursion for it. First let’s look at the pseudo code,

1. Tour(row, col):
2. // Mark that current square is marked as step[step]
3. Path[row][col] = steps
4. Visited[row][col] = true;
5. Steps++
7. // gets all moves from the square [row][col]
8. moves = findMoves(row, col)
10. // for every move from this square move there if it hasn‘t been visited
11. For move in moves:
12. If(not visited[moves[0]][moves[1]]):
13. Tour(moves[0], moves[1])
15. // end the algorithm if the tour is solved
16. If tourComplete:
17. Return;
19. // go back if no child paths work
20. Visited[row][col] = false
21. Path[row][col] = 0
22. Steps - -

*Figure 1*

From a given square on the board, the knight can have up to 8 squares that it can move to. Towards the edge of the board, it has less squares to move to with as low as 2 from a corner square. Since we can never jump on the same square twice then when we hop to a square there are up to 7 squares to hop to which results in up to 7 recursive calls per call. Marking the current square is a constant time operation, finding the moves from the current square is also a constant time operation, checking if the tour is complete is O(1) where n is the area of the board, and unmarking the square at the end of the call is constant time as well. So, in total, the work we do per call is O(n). This gives us a final recursion of

T(n) = 7T(n+1) + 1

In this recursion, n will start at 0 and go up to n times n, the total number of square on the board, because we start at move 0 and have to check all possible paths on every square. This results in a grand total of 7n calls. With only a constant amount of work per call we get a final runtime of O(7n). That is abysmal and will likely struggle to run on boards much larger than the standard 8 x 8 chessboard, but it is only the slow algorithm so a poor runtime is to be expected.

III. Warnsdorff’s Rule

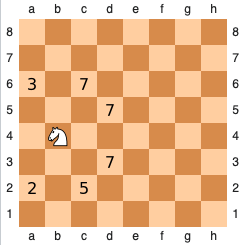
While the backtracking algorithm has the power to solve the Knight's tour problem, an improvement can be made on the algorithm by adding a heuristic.  In 1823 H. C. von Warnsdorff developed a heuristic to aid in the solution to the Knight's tour problem. This heuristic would later be known as Warnsdorff's rule. The rule is as follows:

"Let the (n+1)th square of the path be the square which

1. is adjacent to the nth square,

2. is unvisited (that is, it does not appear earlier in the path), and

3. has the minimal number of adjacent, unvisited squares. " [2]



*Figure 2*

In Figure 2, we can see that the knight is positioned at B4. From this position the knight can move to A2, A6, C2, C6, D3, and D5. Each number in these positions represent the possible moves that can be made from these specific positions. According to Warnsdorff's Rule, that path to be taken must be the one that has the smallest possible moves. In this example, the knight would move to position A2 because the knight can only move to two positions from that location. The algorithm implemented for Warnsdorff's Rule is as follows:

1. WarrnsdorffMoves(row, col){
2. moves[] = findMoves(row, col)
3. //find amount of continuing moves there are from each move
4. let branches [0...moves.size] be a new array
5. for i=0 to i<branches.length{
6. branches[i] = findMoves(moves
7. }
8. //counting sort to sort the moves calculated
9. counting-sort(branches, Warnsdorff-moves, k)
10. }

*Figure 3*

In figure 3 we can see that the runtime is O(1). Normally, a counting sort algorithm would run in O(n + k), but since we know that a knight can only move up to 7 squares then the size of the array we are sorting is at most 7 and k would also be at most 7. Since n and k are both constants we can assert that using counting sort on a constant amount of elements will take a constant amount of time.

IV. Divide and Conquer

The divide and conquer algorithm uses recursion to find a solution. In the Knights Tour Problem, divide and conquer will produce a base case where the smallest board that it will solve will be a board where the length and width are less than 10. The algorithm divides the board into quadrants until it reaches the smallest subproblem. Once it calculates the base case, it combines the quadrants for each subproblem, working its way back to its original size.

After some testing with the backtracking algorithm on boards of many sizes, it was discovered that board sizes smaller than a 10x10 were able to be solved very quickly. This means that boards of sizes 10x10 and larger could be broken down into smaller, solvable boards. When the base cases have been solved, there must be a way to connect all of the sub boards together. The knight would hop from one smaller board to a point on the next board and repeat until all of the sub boards are connected to form a solved knight’s tour on the original size. Because of the properties of boards that size, we will be splitting up our large board into manageable chunks. To do this we will use the following algorithm to divide the height, and width of the board into chunks of that size, “draw” lines dividing our large board into a grid composed only of ideal boards, and then solving those boards linearly. Here is the recursive algorithm to divide the board into chunks.

1. DNC(length, width):
2. Tour t
3. // Base Case
4. If length < 10 and width < 10
5. t = new Tour(length, width)
6. t.solveBoard()
7. Else
8. newLength = length/2
9. newWidth = width/2
10. Tour t1 = DNC(newLength, newWidth)
11. Tour t2 = DNC(newLength, newWidth)
12. Tour t3 = DNC(newLength, newWidth)
13. Tour t4 = DNC(newLength, newWidth)
14. t = join(t1,t2,t3,t4)
15. Return t

*Figure 4*

In figure 4 we can see that the runtime of the algorithm is O(nlogn). This is because in lines 10-14, it takes lgn time to calculate the sub board. It takes an additional n time to reassemble the board, represented in line 14

Dr. Ian Parberry proposes that the divide and conquer method can solve the knight’s tour in O(n2) time. Dr. Parberry proposes that the recurrence derived by using the divide and conquer method is as follows:

*T(n) = 4T(n/2) +O(1)*

Here, Dr. Parberry explains that the board is divided into 4 sub boards that are determined by taking the original length and width of a board of size n x n and dividing them by two. The constant time at the end is the amount of time it takes for the quadrisected boards to be reassembled.[3]

To get a better grasp of how this divides the board up, let us see how it would run on a 31 × 31 board. Since 31 is greater than our base case of 10, it would split it into a chunk of 15 and a chunk of 16. Each chunk of 15 would be split into a chunk of 7 and a chunk of 8 and the chunks of 16 would be split into two chunks of 8. Since these splits were made on both axes, we would get the following table of subproblems

|  |  |  |  |
| --- | --- | --- | --- |
| 7 x 7 | 7 x 8 | 8 x 8 | 8 x 8 |
| 8 x 7 | 8 x 8 | 8 x 8 | 8 x 8 |
| 8 x 8 | 8 x 8 | 8 x 8 | 8 x 8 |
| 8 x 8 | 8 x 8 | 8 x 8 | 8 x 8 |

*Figure 5*

The algorithm would then solve 4 subproblems and merge them. So, after solving each quadrant in the example above we would have a solved 15 x 15 in the top left, a solved 15 x 16 in the top right, a solved 16 x 15 in the bottom right, and a solved 16 x 16 in the bottom left. The last step would then be to merge each quadrant such that it completes a 31 x 31 CKT. However, ensuring that the four quadrants can be joined together in a way that makes one large tour requires a fair amount of logic. The pseudocode is rather lengthy and difficult to read, but Dr. Parberry outlines the process with a nice simple picture. The first step to doing this is to make sure that each quadrant is solved as a “structured” knight’s tour (SKT) as Dr. Parberry puts it. An SKT is a CKT that also has the following specific formation of moves around each of the 4 corners of the board.

Diagram, engineering drawing

Description automatically generated

*Figure 6[3]*

If those four moves are ensured in our solution, then when the 4 quadrants are lined up next to each other they will looks like this

A picture containing text, shoji

Description automatically generated

*Figure 7*

Now Fig. 6 does not show the moves on the actual corners as in Fig. 7, only the moves adjacent to them, however since a knight only has 2 moves from a corner it is guaranteed that it will move in that formation. So now that the 4 quadrants have been solved in a structured way and the formation in Fig. 7 is achieved, the 4 quadrants can be merged by replacing the moves in Fig. 8 (a) with the edges in Fig. 8 (b).

Diagram

Description automatically generatedDiagram

Description automatically generated

*Figure 8 (a on the left, b on the right)*

Since each quadrant was a SKT and therefore a CKT then by rearranging the edges in that way not only do we create a larger knight’s tour, but the larger knight’s tour is also a SKT allowing to to be merged with larger boards so that the recursion can work through multiple iteration of dividing.

V. Runtimes and Analysis

The slow algorithm greatly struggled to solve a board larger than 11 x 11 in a reasonable amount of time, but enough data was still able to be gathered that the runtime was able to be accurately analyzed. From 5 x 5 up to 11 x 11 the simple backtracking algorithm clocked in at the following runtime.

|  |  |
| --- | --- |
| Size | Runtime (milliseconds) |
| 5 x 5 | < 1 |
| 6 x 6 | 1 |
| 7 x 7 | 627 |
| 8 x 8 | 14 |
| 9 x 9 | 55 |
| 10 x 10 | 2570 |
| 11 x 11 | 20486 |

*Table 1* *and Figure 9*

After plugging those numbers into a regression line calculator, we can see that the observed runtime of the algorithm is c\*5.955n with a coefficient correlation value of 0.88 for an n-by-n board. This function is the blue line in Fig. 9. This runtime is actually slightly better than what was estimated in section 1. This is because in the earlier estimation it was assumed that a knight could move to 7 squares from any position. However, this is not in several cases. For example, if the knight is on a corner or edge then it can only move somewhere in the range of 2-4 squares meaning it has less options, and the fewer options a backtracking algorithm has the lower its runtime will be. However, it is still apparent that this simplistic algorithm will not suffice on larger boards which brings us to Dr. Parberry’s algorithm. After extensive testing we collected the following data

|  |  |
| --- | --- |
| Size | Runtime (milliseconds) |
| 32 x 32 | 283 |
| 64 x 64 | 906 |
| 96 x 96 | 1213 |
| 128 x 128 | 4064 |
| 160 x 160 | 52 |
| 192 x 192 | 3812 |
| 224 x 224 | 1110 |
| 256 x 256 | 10986 |

*Chart, line chart

Description automatically generatedTable 2 and Figure 10*

This data is much more scattered and less uniform than the last set, but after plugging these values into the same calculator, we get an observed runtime of n4 with a coefficient correlation of 0.91 on an n x n board. This is immensely better than 8n but not quite the expected output given our implementation of the algorithm. Like Parberry’s proposed algorithm, our algorithm splits the board length wise and height wise in half, thus into 4 quadrants, and solves each quadrant in the same fashion and then joins them together. However, our algorithm is only able to join the four quadrants in O(n2) time instead of O(1). This results in the following recursion for our algorithm

Since, , then by case 2 of the master method we can say that our implementation of the algorithm runs in Θ(n2 lg(n)) time. While this is not quite as good as Dr. Parberry’s, it is blisteringly fast compared to the simple backtracking algorithm, even when it used Warnsdorff’s rule as a heuristic

VI. Conclusion

There are many approaches to solve the Knight’s Tour problem. This paper discusses two approaches of solving the Knight’s Tour, a backtracking depth-first search algorithm, with and without a heuristic, and a divide and conquer algorithm. Through testing on boards of various sizes it was determined that there was a significant improvement when running the divide and conquer algorithm compared to running the backtracking algorithm. These results were compared to theoretical time results proposed by Dr. Ian Parberry. When comparing the results of our algorithm’s implementation with Dr. Parberry’s theoretical time, it was observed that our algorithm performed slower than Dr. Parberry’s theoretical time. The knight’s tour is not limited to backtracking and divide and conquer algorithms. There are other ways to solve this problem that may result in faster runtimes on much larger boards, some of which venture into Artificial Intelligent algorithms. The knight’s tour is a problem that will be used to analyze many different algorithms in the future, some of which have already been explored and have different implementations, while others have yet to be discovered.

References

[1] A. Schwenk, “Which Rectangular Chessboards Have a Knight’s Tour?”, *Mathematics MAgazine,* vol. 64, no. 5, pp 325-332, Dec. 1991.

[2] D. Squirrel, P. Cull. “A Warnsdorff-Rule Algorithm for Knight’s Tours on Square Chessboards.” Pp. 111-131, 1996

[3] I. Parberry, “An efficient algorithm for the Knight’s Tour Problem”, *Discrete Mathematics Applied*, 1997.