Using Backtracking to Solve the Knight's Tour Problem

Weston McEvoy
Department of Electrical and Computer Engineering
University of Colorado Boulder

Boulder, United States

weston.mcevoy@colorado.edu

***Abstract*—This paper will explore one of many algorithms that solve the infamous Knight's Tour Problem. Everything from the construction of the algorithm to the limitations of its structure will be covered. Detailed proofs, examples, and figures are included.**

Keywords—recursion, complexity, induction, runtime, backtracking, validity

I. Introduction

For centuries, the game of chess has challenged society's greatest minds. The game appears simple and even trivial at first; but many have honored it as a good measure of one's intelligence due to the strategy it demands for the player to experience success. Despite featuring only 32 pieces and 64 squares, the possible outcomes of the contest have proven to number in the trillions and trillions. Given this scale, it becomes incredibly difficult, or even impossible, for super computers to compute the total number of outcomes within a timeframe that is anywhere near reasonable. The purpose of this report is to better understand programming and arithmetic principles that are utilized when calculating large numbers of possible outcomes. In order to reduce computation time so that program results may be more easily examined, this report will explore a simplified version of chess, the knight's tour problem.

II. The Knight's Tour

D. The Rules of the Game

The knight's tour is a chess puzzle that first came about in a ninth century manuscript published by Abu Zakariya Yahya ben Ibrahim al-Hakim. The puzzle was later made popular by Euler in the 18th century [1]. The challenge is as follows: Start one knight on any given square on an empty board and find a path that allows the knight to hit every square of the board without hitting the same square twice. In accordance with the rules of chess, the knight can only move in an "L" shape. That is, two squares in one direction and one square in the other.

D. Approaching the Synthesis of the Algortihm

Before the algorithm can be written, the process of finding a path must be broken down into smaller tasks. The first step in solving the puzzle is choosing a square on which to start the knight. This step is relatively straight forward. Since the knight can start on any square, there are 64 possible initial conditions. Following the choice of an initial condition, the next step is to make the first move. For this move in only, any square on the board is game since none of the other squares have been touched yet. However, it is necessary to ensure that the square of the first move is in fact on the board. When a human performs the knights tour, they can simply perform this check visually. A computer does not have this luxury. Thus, before every move is finalized, it must be verified by the computer that the next square is in fact on the

board. For all moves after the first one, in addition to verifying that the next square is in fact on the board, it must also be checked that the next square hasn't already been touched. On the second move, this is easy. The only square that has been touched is the initial condition and the one that the knight is currently on. Therefore, the only square that is invalid is the previous square. As more moves are made, more and more squares become invalid and therein lies the difficulty of the problem. The problem is solved when all previous conditions have been satisfied and all squares on the board have been touched. Now that the proceedings of solving the problem are evident, each individual step can be broken down further.

III. Definitions

C. *Board Setup*

For a programming approach, it is essential to define the board numerically in order to easily keep track of moves and set initial conditions. This can easily be done by creating an 8x8 array of zeros. Using this method, squares of the board can easily be referenced by row and column. In this instance, let the variable *board* be the array that contains the layout of the board. The first position in *board, board{1,1},* corresponds to the top left corner of the board.

C. *Possible Knight Moves*

Basic arithmetic can be used to determine the moves of the knight. In order to determine all possible next positions, it is necessary to assume it starts in the middle of the board (depicted by the green circle in Fig. 1). This ensures that all possible moves are accounted for. If the knight starts on the edge of the board, some next moves may be impossible since they are off the board. With this assumption made, the knight has eight possible next positions. In the following descriptions, the "L" shaped move is broken down into two sub-moves. The first sub-move is two squares long and the second is only one. The inverse case is not needed since it would produce the same positions but with different paths. In this application, the path to the position is irrelevant. The first sub-move can either be row-wise or column-wise. If row-wise, this involves adding or subtracting two to the current column. The second sub-move is then completed by adding or subtracting one to the current row (depicted by the red circles in Fig. 1). Intuitively, this procedure produces four possible outcomes. The other four are found by repeating the same procedure but moving column-wise first rather than row-wise (depicted by the blue circles in Fig. 1).

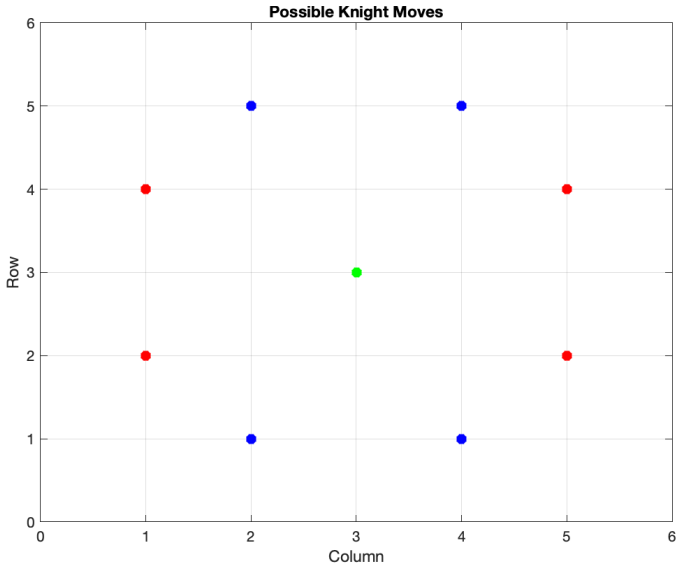

Fig. 1. The eight possible moves of a knight.

These moves will be stored in two separate matrices. The first, *x,* will store row incrementing/decrementing values and the second, *y,* will store column incrementing/decrementing values. Each position of these matrices corresponds to each other. For example, if *x(1)* is 1 then *y(1)* cannot also be 1. The combination of these two values would result in the knight moving one column and one row which is an invalid move. The vectors in Fig. 2 show the correct matching between the x and y positions.

```
x = 1×8
     1     2     2     1    -1    -2    -2    -1

y = 1×8
     2     1    -1    -2    -2    -1     1     2
```

Fig. 2. *x* and *y* move vectors.

The vector values don't necessarily have to be in this order as long as those same eight pairs are always present. Namely, x*(1)* and *x(8)* could be switched if and only if *y(1)* and *y(8)* were also switched.

C. Valid Moves

Whenever a move is made, a check is needed to determine whether the next positions are valid. Within the context of the problem, a next position is valid if it satisfies two conditions. First, if the position is a square that exists on the board and second, if the square has not been touched by the knight yet. The algorithm will perform the first of these checks by ensuring that both the row and column of the position is less than 8 and greater than 0. The second check will be done by checking that the value of the position, *board{row, column},* is equal to 0. Recall that *board* was initialized with each position taking on a value of 0. The validity check just described is performed within its own function.

IV. Moving Through the Algorithm

All operations described in this section all occur in one function, *solve.* It is important to note that there is a call to the validity function within *solve.*

B. Moving the Knight

After the board has been setup, it is time to begin solving the problem. The first step is obvious: move the knight! This is done by adding one of the values of the *x* move vector to the current row and adding the corresponding value of the *y* move vector to the current column. Once a move has been made, the validity check described earlier is checked. If valid, the value of the new position is set to the number of move it is (if it's the first move, the position gets set to 1). The process just described is then repeated over and over until the number of moves is 63 and the problem is solved or an invalid position is encountered.

B. Handling Invalid Moves

If a move is made and it is determined to be invalid, another pair from the move vectors must be used to try a new move. This is repeated until a valid move is found. At that point, another move can be made.

B. Recursion

It is now essential that the recursive nature of the algorithm is addressed. Whenever a valid move is found within *solve,* the row and column of the new position must be passed into *solve* in order to find the next position. This call to *solve* is placed within itself so that moves can be made one after the other without having to exit the function and change the initial conditions. This approach is extremely effective in using few lines of code to implement the algorithm. However, more is still needed to ensure that the algorithm performs as it should. To illustrate this, it is assumed that fifty moves have already been successfully made. On the 51st move, the algorithm is unable to find a valid move. Rather than exiting the entire function stack and concluding that there is no solution, a method called backtracking can be used instead.

B. Backtracking

Continuing off the previous example, the most recent move can be "undone" by returning 0 from the current *solve.* A check within the previous *solve* recognizes this and resets the current position's value to 0. It then tries a different move. By doing this, the algorithm can retrace its steps instead of starting all over with different initial conditions. This is known as backtracking [1]. This method will eventually successfully result in a solution, like the one seen in Fig. 3.

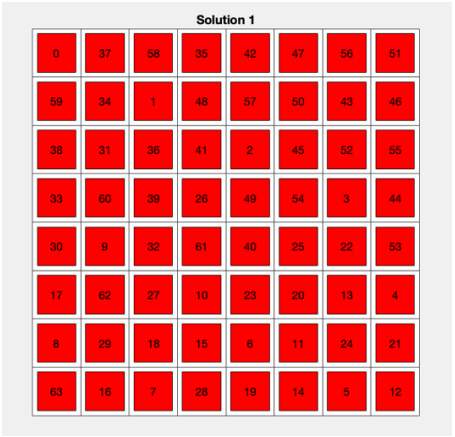

Fig. 3. The first solution found by the backtracking algorithm.

V. Proving The algorithm

Before going into runtime and algorithm complexity, it first needs be proved that the algorithm holds for all positions. The use of recursion warrants a proof by structural induction. The algorithm implemented operates on the rows and columns using simple arithmetic.

$$row = i + x$$

$$col = j + y$$

These two equations can be combined to form one equation that operates on position, or the sum of the row and column:

$$p(t) = p(t-1) + m_t$$

Where *p* is position, *t* is the turn, and m_t is the amount that a position can change by when a move is made.

$$m_t = [-3, -1, 1, 3]$$

Before the proof is performed, it must be kept in mind what is being proved. The proof is attempting to show that if the knight is on a square on the board, then it has at least one move to make to get to a different square on the board that is not the previous square. In other words, it must be proved that:

$$2 \leq p(t) \leq 16$$

The first step is demonstrating that the base case holds. If a square on the board is chosen as the initial condition, this will be true. In the instance of the solution shown in Fig. 4., the starting position is *board{1,1}*. Adding the row and column yields *p(0) = 2*. Note that this satisfies the previous equation, therefore proving the base case. Next it must be shown that the same condition holds for *p(t+1)*. To do this, the following steps are needed:

$$p(t+1) = p\big((t+1) - 1\big) + m_{t+1}$$

$$p(t+1) = p(t) + m_{t+1}$$

$$p(t+1) = p(t-1) + m_t + m_{t+1}$$

$$p(t-1) = p(t+1) - m_t - m_{t+1}$$

Note the use of the inductive hypothesis between steps one and two and the invocation of the inductive step between steps two and three. By the inductive hypothesis, it is known that any previous value of *p* holds so the following equation can be derived:

$$2 \leq p(t+1) - m_t - m_{t+1} \leq 16$$

From this point, some less obvious steps are needed. Examining the equation, it becomes apparent that, in order to satisfy the proof, m_t and m_{t+1} must be equal in magnitude and opposite in sign. Due to our initial condition, $|m_{t+1}| = |m_t| = 3$ and m_t must take on the positive value. Had the initial condition been in the middle of the board, $|m_t|$ and $|m_{t+1}|$ could have taken on the value 1. After applying these conditions, the proof is finally complete:

$$2 \leq p(t+1) \leq 16$$

VI. Breaking Down The Proof

If it wasn't obvious in the previous section, this algorithm is difficult to model mathematically. At first the proof seems poorly constructed and even wrong; however, other implications of the problem qualify it.

A. Same |m|, Different Moves

There are a few apparent issues with the proof. The first misconception is that since m_t and m_{t+1} take on the same value and opposite sign, the move just made is simply being undone. This may or may not be the case. Recall that moving 1 row and 2 columns results in a different position than moving 2 rows and 1 column but both moves have $|m_t| = 3$.

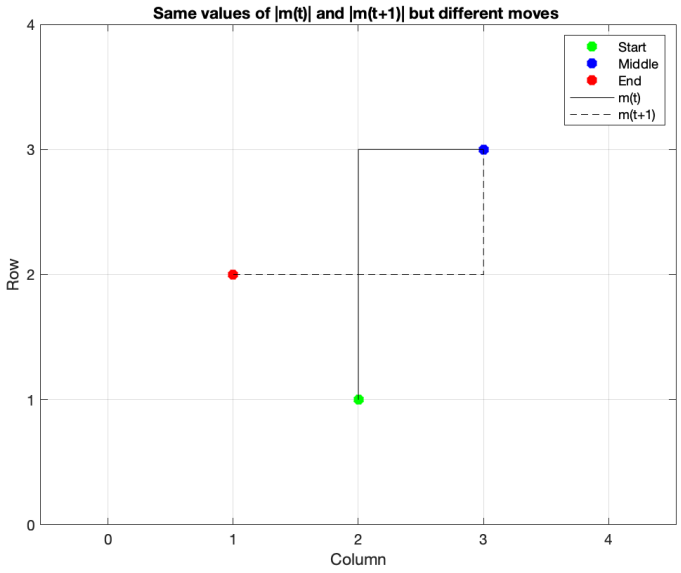

Fig. 4. Same value of *m* but different moves.

B. *Proof Accepted Moves vs. Program Accepted Moves*

It is important to remember that the proof only guarantees the next position will be on the board. It does not guarantee the satisfaction of the algorithm's validity check described in section III.C. Looking at Fig. 4., it can be observed that from the *Middle* position only two moves are possible: back to the *Start* or to the *End*. These both have $|m_{t+1}| = |m_t|$ so in the context of the proof either will do as seen in Fig. 5.

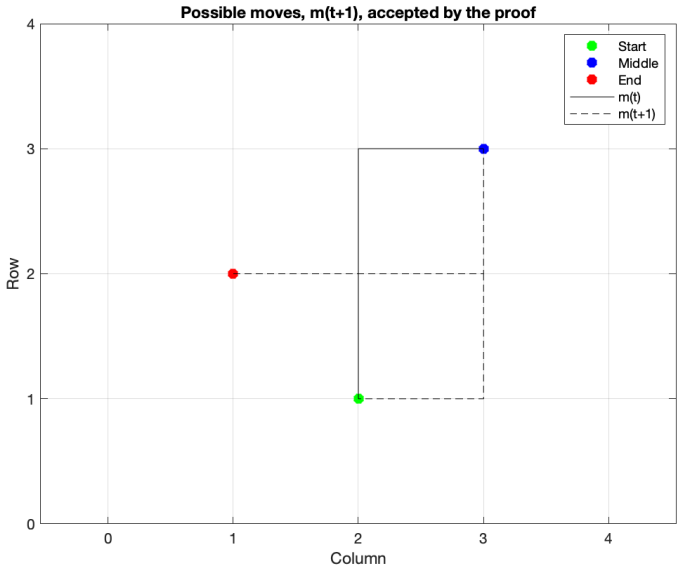

Fig. 5. Proof-accepted moves.

However, this is not the case within the program. The program features something that is very hard to model numerically and was therefore omitted from the proof: the portion of the program's validity check that ensures the next square hasn't been visited. From the program's point of view, the same square can't be touched twice so the choice is obvious. The only possible m_{t+1} is the one shown in Fig. 4.

C. *The Importance of the Validity Check*

If the inclusion of the validity check yields different outcomes, then how could it have been left out of the proof? Recall that there are two portions of the validity check: ensuring the next square is on the board and ensuring the next square is unused. The proof includes the first part of the check and proves that the method used to index rows and columns will always result in a square that is on the board. The second part is used within the program to corroborate the proof. More specifically, the proof yields a square on the board and the second half of the validity check ensures it is untouched. By doing this, the back half of the check simply guides the proof through its moves by eliminating values that m_{t+1} could not take on.

D. m_t *and* m_{t+1} *can be Different*

With a little intuition, it is now apparent that m_{t+1} does not always have to be opposite in value to m_1(within the context of the program anyway). This is most easily observed in larger boards, such as the 8x8 in this application. For example, consider if the same first move was made in Fig. 4. but on a 5x5 instead of a 3x3. There are now more possible values of m_{t+1} than just the one shown. The knight could instead move in the same manner it just did again(+1 column, +2 rows). A sufficiently larger board typically means there are more possible values of m_{t+1} then the one that is guaranteed to hold. The knight has fewer values of m_{t+1} to choose from in corners and along edges because of this same phenomenon.

VII. Limitations of the Algorithm

A. Worst-Case Complexity

Since more than one value of m_{t+1} can be possible, how will the algorithm choose which one to use? In this implementation, this is entirely dependent on the initialization of the *x* and *y* matrices. The program will try the move stored in the first position of these matrices first. If this move is invalid, it will try the next move. This means that worst-case, the program will have to try eight different values of m_{t+1} to find one that works. If none of the eight works, then backtracking takes effect and a new move must be tested for the previous call to *solve.* In order to guarantee that a solution is found, the algorithm must check all moves for every square [3].

$$\sum_{5}^{n} 8^n$$

The above expression represents the number of checks that must be performed given a board where $n = rows * column$. In this application, the program must perform 8^{64} checks worst-case. Therefore, its worst case complexity is O(8^n).

B. Best-Case Complexity

The best-case scenario for solving the Knight's Tour occurs when every move made is correct and no backtracking is required. In other words, it only takes 64 moves to traverse the entire board. However, this is impossible when using *solve* because the program cannot see ahead and determine an invalid move. It must make the move and then determine it is invalid. Fig. 3. is a great visual for this. The knight will move in the same direction until it reaches position 4. It appears as if the program senses a wall and knows it must move another direction than it was heading previously; instead, it moves in the same direction again and then realizes the move was invalid. It then tries the next move in *x* and *y* and realizes it is invalid too. The third move tried works and thus it took 7 moves, rather than 5, to get to position 5. What then is the best-case complexity of the program? It can be estimated by finding the ordering of *x* and *y* that produce a solution in the quickest amount of time given the initial position of *board{1,1}.* This happens to be the order shown in Fig. 2. A simple implementation of an *iteration* variable within *solve* allows the number of checks to be counted. When run with these initial conditions, 66,005,601 checks are performed before the program arrives at the solution. Some simple math helps visualize this complexity:

$$8^n = 66005601$$

$$nln(8) = \ln(66005601)$$

$$n = \ln(66005601)\ /\ln\ (8)$$

$$n = 8.66$$

Thus we can characterize the runtime as o(8^8). Note that this is less than the worst-case complexity of the same algorithm performed on a 9x9 board with the same conditions. Just by inputting a certain order of the *x* and *y* matrices, the runtime of the program can be drastically reduced.

VIII. Finding More Solutions

A. Choosing the Optimal Ordering of x and y

How then can the arrangement of moves within the matrices be chosen to minimize runtime. Looking at Fig. 1., choose one of the moves that is possible given the initial conditions. In this case with the initial condition *board{1,1}*, one possible option is the position *(4,5)* on Fig. 1. Next, by moving counterclockwise around the remaining moves, the matrices in Fig. 3. are generated. Interesting enough, if the other possible first move is chosen, traversing Fig. 2. clockwise yields new *x* and *y* matrices that also produce a solution.

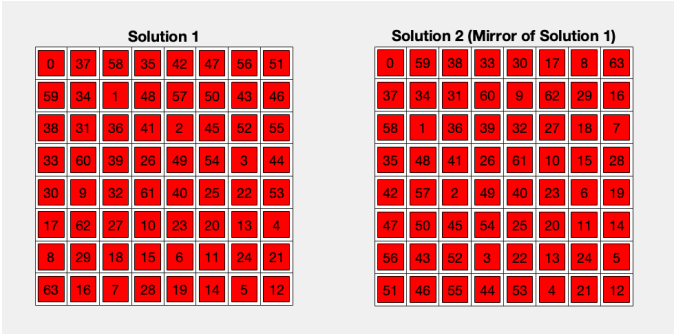

Fig. 6. Two possible solutions

Curiously, these two solutions are mirror images of each other. This same approach can be used to generate 6 more solutions in the other corners.

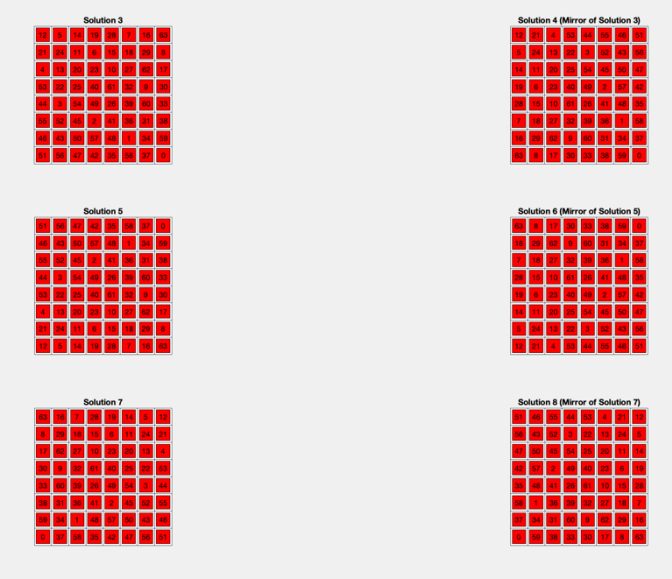

Fig. 7. Six more possible solutions, one from each corner

B. *Solutions for other Squares?*

To locate a possible path for each initial position on one of the diagonals of the board, a *for* loop was used to input different initial positions to a new function: *solve2.* This function is identical to *solve* except for the way it checks if a square has already been used. The board is initialized with values of -1 rather than 0. The new validity check is performed by a slightly modified *valid, valid2.* This allows initial positions other than *board{1,1}* to be inputted. The values of *x* and *y* from Fig. 2. were used for every iteration. The program found the first solution (that of Fig. 1.) within thirty seconds but failed to find a second solution in a reasonable amount of time. To fix this, the variable, *iteration,* was introduced again. A conditional statement in the first few lines of *solve2* terminated the process if iterations grew above one-hundred million. This prevented the program from spending long amounts of time on one solution. After attempting to find a solution for all 8 squares on the diagonal in less than one-hundred million checks, only one was found.

C. *Next Steps*

In order to have success using the procedure described in the previous section, code needs to be introduced that can optimize the order of *x* and *y* based off the initial position. This is an entirely new algorithm in its own, however, and will require a significant amount of problem solving and testing. Perhaps a more sensical direction would be to explore more efficient algorithms that solve the Knight's Tour. For example, Warnsdorff's algorithm involves checking the number of possible moves from all of the valid next positions and moving to that with the fewest [2]. Another possible exploration could include investigating the generation of closed knight's tours [4]. These are tours in which the final move is one move away from the initial position.

References

[1] Carlos Alberto Colodro, "Euler and the Knight's Tour," May 2020.

[2] Uddalak, Bhaduri "The Knight's Tour Problem | Backtracking 1," October 2021.

[3] J. Zelenski, "Exhaustive recursion and backtracking," February 1892, pp.2.

[4] Chess.com haha

Appendix

See submitted .pdf and .m files.