Josh Hillerman, Vikram Thirumaran, Devin Vliet

Prof. Moaaz Alqady

MATH 232

June 5, 2024

Knight's Tour Project Report

Throughout the different project options. Our group chose the Knight's Tour problem to work on since it was at first easy to conceptualize, and one of the few problems that we felt could be discussed both in and out of mathematical context. This problem, however, ended up being quite a rabbit hole and had many different layers of complexity for us to delve into. Many different developments have been made since the first conception of the problem in the 9th century(300 years after the invention of chess), and new ways to solve it have been discovered less than 30 years ago. The problem, at a base level, consisted of a standard 8x8 chess board, and the task was to find out if a knight could touch every single tile exactly once: a Hamiltonian Path, or an open tour of the board. The aforementioned complexity arose when we had to think about closed tours, or Hamiltonian Cycles, larger and smaller boards, rectangular boards, three-dimensional boards, and even non-Euclidean Boards. Thinking of the tiles on the board as vertices, and the edges as an indication that we can move there legally, the problem could be represented using concepts and theorems learned in graph theory, we just had to find the algorithms and proofs that could help us solve the problem. Some theorems and laws we used were Warnsdorff's rule, Ira Pohl's tie-breaking, and color theory. On top of all of this, implementing this problem into a coding language was another challenge, so with our work cut out for us, we made our first attempt at the problem.

While researching theorems provided in the project description, our first implementation (Implementation 1)of the problem used C code and a depth-first search algorithm(DFS). We had learned about the DFS algorithm in previous computer science classes, and the graph theory equivalent in class the prior week, so we started implementation 1. This was bound to fail, as we knew that it was inefficient and simply a brute force technique, but getting a benchmark test of the problem would help us see how we have progressed. Implementation 1 began with taking user input in the format ./hKnight <n> <m> <starting corner>, initializing a chessboard with dimensions n * m. The chessboard was a 2D array of integers, and every value started off as -1. After that, the list of all 8 moves a Knight could 2 was split up into 2 arrays (vertical and horizontal movement), and stored so that it could be used to traverse the board later. After that, the starting corner was picket(by user input), the integer in the 2D array was assigned a 0 (instead of the previously initialized -1) and we could begin traversing the board. We first loop through the list of moves we just defined above, and pick the first legal one. We keep doing this until we complete the board or we reach a dead end, where all 8 moves are illegal. If the board is completed, we are done, but if there is a dead end, we backtrack, going to the previous tile. From the previous tile, we continue iterating through the list of legal moves, and select the next available move in our list, and as far as we can again until success or failure. Using this guess-and-check method, we eventually find the solution or keep backtracking until we run out of moves on our starting corner, which returns a failure. This code had its pros and cons, but it was nowhere near what we needed it to be. Since it just followed the first available path instead of choosing where to go by rules defined in an algorithm, the program was bound to be wildly different when it came to the input the user provided. The size of the chessboard would dramatically lengthen the runtime, and while that makes sense, the code was so inefficient a 9x9

board was impossible to solve. Furthermore, the corner in which the piece started influenced the runtime tremendously, due to inefficient move selection by our DFS algorithm. The DFS worked fine for smaller boards, but anything over 8x8 was a failure, so we moved on to the next implementation.

WARNSDORFF'S HEURISTIC

To improve our algorithm, we first studied previously existing theorems and heuristics. The project description mentioned Warnsdorff's rule, used to choose the best next move based on the "importance" of each tile on the board. German mathematician H.C. von Warnsdorff developed this rule in 183, aiming to reduce the complexity of the problem by mapping each step and choosing a move that minimizes future difficulties. This rule works by calculating the "onward move" and assigning a number to every tile. The number of onward moves is determined by counting the number of legal moves a knight can make from that tile, excluding previously visited tiles. After that, the knight moves to the tile with the smallest onward number, or the tile with the fewest amount of options for a subsequent move.

The step-by-step process of the heuristic is as follows:

1. Choose a starting position for the knight on the chessboard.
2. For each of the possible knight moves from the current tile, calculate how many onward moves (next possible knight moves) are available from each candidate square.
3. Move the knight to the square with the smallest number of onward moves. If there are multiple squares with the same number of onward moves, select any one of them, randomly or based on a secondary rule - (this secondary rule will be discussed later)
4. Repeat the process from the new position until the knight has visited every square on the board exactly once.

After understanding the Warndorff Heuristic, we were able to start thinking about how we could implement that into our code to solve our problem on a larger scale and with a lower time complexity.

COLORING PRINCIPLE:

For a closed knight's tour, the knight must visit every tile on the chessboard once and then return to the starting position. We know that the knight can only travel between disjoint tile sets (black to white and vice versa) by definition of the board tiling and knight movement. Therefore if there is an odd number of tiles on the board, we must start on a tile that belongs to the vertex set with more vertices. As we traverse the board and cover the last square, we will end on a tile of the same color as the starting position. However, we know that the knight cannot travel within its disjoint vertex sets, making it impossible to return to the starting position. If we start on a tile that is not part of the larger vertex set, we cannot reach the final tile, and thus cannot return to the start.

The definition of an odd number is: $2k + 1$, where k is an integer. The formula for the number of white and black squares in a chess board is as follows: Black = floor[(n*n)/2], White = (n*n) - floor[(n*n)/2]. If we use our definition of an odd number in place of n, we find this: Black = floor(1/2($4k^2 + 4k + 1$)), White = $4k^2 + 4k + 1$ - Black, Black = $2k^2 + 2k$, White = $2k^2+2k+1$. This clearly shows that for an odd number of n, we will have one more white cell than black. Each knight's move must go from black to white or white to black, thus we must start on the color with an additional cell. Generally, we must start on a cell that lies on the diagonal or its subsequent alternating cells to find a knight's tour.

There are interesting parallels to bipartite graphs due to this rule: you can think of the black and white cells on the checkerboard as members of disjoint vertex sets, where the only edges that can exist are between the two disjoint sets but not within them. This bipartite graph nature enforces the fact that the knight can only move between black and white tiles and cannot form a cycle on a board with odd dimensions.

For a closed knight's tour, the knight must visit every tile on the chessboard once and then return to the starting position. Given that the board is odd, we must start on a tile of the larger vertex set (either white or black). To create a closed knight's tour, we must first create an open knight's tour and then return to the starting position. As we traverse our odd board and arrive at the last open tile, it will be of the same color set as the starting vertex, because there is one more tile of that color on the board. However, to create a closed knight's tour, we must return to our starting position, which is of the same color as the tile we currently occupy. This is impossible due to the rules outlined above. If we start on a tile of a different color to circumvent this, on our second-to-last move we will occupy a tile of the same vertex set as the one remaining, making it impossible to complete the tour. Therefore, no closed knight's tours can exist on odd boards.

IRA POHL TIE BREAK:

Ira Pohl’s tie breaker rule is not necessarily a new rule, but more so an extension of Warnsdorff’s heuristic. It states that in the event of a tie - meaning multiple vertices of the same non-zero degree exist- one should pick the first vertex of the path which results in the minimum sum across the neighborhood of the vertices. “To improve on Warnsdorff’s arbitrary selection in case of ties, the following tie-breaking method was proposed and tested. For each tie move, sum

the number of moves available to it at the next level and pick the one yielding a minimum. In theory this can be carried through as many levels as necessary for tie-breaking." [Pohl]

Pohl describes a generalization of this method as a multilevel approach in his paper: "This modification to Warnsdorff's rule is a generalization of the original concept. In terms of the connections of the unreached squares it is again a maximizing rule, with sufficient power always to work with a knight on a chessboard. The generalized rule (for method of order k) is: Consider all paths of k moves and count the remaining number of connections for each path. Select the first move of the path whose number is maximum, providing this path is not a dead end. Ties are broken by going to k + 1 moves."

This statement is articulating the same point, that selecting the minimum sum leaves the most open paths left and thus maximizes the amount of moves available.

MATRIX OVERLAY SOLUTION:

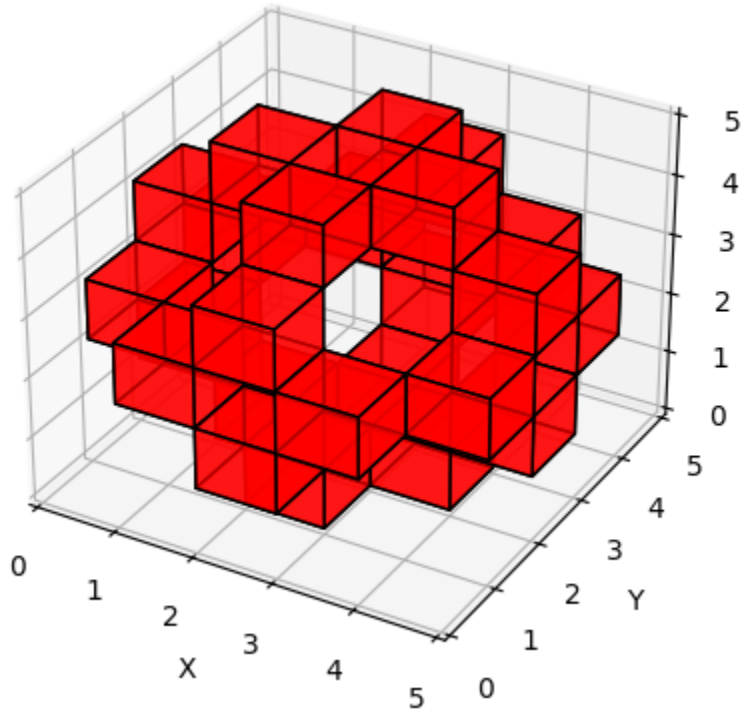
This algorithm implements Warnsdorff's Rule, The Coloring rule, and Ira Pohl's Tie Breaker to solve Knights tours using matrices. It determines effective start positions using the coloring rule and layers matrices to determine moves of minimum degree, with ties being severed via Ira Pohl's Rule.This algorithm implements Warnsdorff's Rule, The Coloring rule, and Ira Pohl's Tie Breaker to solve Knights tours using matrices. It determines effective start positions using the coloring rule and layers matrices to determine moves of minimum degree, with ties being severed via Ira Pohl's Rule.

This algorithm begins with generating a NxN matrix and initializing it with zeros as every element. It then proceeds to overlay this starting matrix with a border of 2x2 ones, so now the dimensions of the matrix is N+2 x N+2. Now we apply the coloring principle to the board and color the elements that are initialized with zeros in alternating black and white squares, in the case of N being an odd number we will mark the larger set of tiles with ones and then select any one of the elements in the matrix that is equal to zero and mark it with our starting position. This ensures that we will select a starting position that will have an open knight's tour associated with it. Any square in the matrix has a 5x5 “moves matrix” associated with it where every element is initialized as a one, except the valid moves from the center position of the matrix of which there are eight. This “moves matrix” is how we select valid and open moves throughout our algorithm. Next we will take a 5x5 sub-matrix of our board with the selected position as the center position, And logically “OR” it together with the moves matrix, this returns a 5x5 matrix with zeros in the position that are open on the main board and also a valid move via the moves matrix. Since this new matrix has different indices than the main board we now convert the indices that represent open and valid moves back into the indices relative to the main board. Next we apply Warnsdroff’s Rule. For each open and valid position we run this submatrix moves matrix process again but instead of returning the valid moves we simply count the number of the moves, i.e determine the degree of the open and valid moves. According to Warnsdorff we then select the open and valid move that is of least degree, or least accessible and continue our process from that position. In the case that both possible moves have an equal degree or there is no minimum we will apply Ira Pohl’s tiebreaker and sum the degree of each potential moves adjacent legal cells and select the minimum of those. This algorithm continues until every tile on the board is visited in this manner and returns the list of path vertices and the filled board.

This implementation is very quick to provide results in fact so quick that it can produce an open tour of a 100 x 100 chess board in less than three seconds and a 400x400 open tour in around 30 seconds. This is due to the selection process and the tiebreaker method. The classical approach to this problem utilizes the Warnsdorff heuristic and a random tie-breaker which is extremely effective in small dimensions but fails as values of n grow. An excerpt from a paper by Oregon State University states, “As expected, the rule is very successful for small boards, succeeding more than 50% of the time for almost all m < 100. However, the success rate drops sharply as m increases; when m > 200 we find that fewer than 5% of the attempts produce tours, and when m > 325 we find no successes at all. These observations provide strong evidence that the success rate of a random tiebreak method goes rapidly to 0 as m increases.” (Squirrel and Cull)

The classical solution to this problem has a zero likelihood of producing an open tour with an NxN sized chessboard where n > 325, however this implementation produced one in thirty seconds (with a clever starting position). While this method is very effective in producing paths of large boards quickly it does not necessarily find a path every time in large dimensions. To produce a 400 x 400 open tour solution, it took three tries, however to produce one in a single attempt you can think creatively about where you start off from. If it is an even board then we don’t have to consider the coloring rule for starting position but we can instead select the start position of least degree which in any board there are four starting positions with degree = 2, the corners. By selecting the corner position you are effectively forcing the algorithm to check off the least accessible tiles first before continuing on to the whole of the board, in which there are many many opportunities to effectively close off the tour from the corner positions. This initial selection process is a bit of a naive fix to the large board problem, but it does provide results with a higher probability of success.

THREE DIMENSIONAL APPLICATIONS:



Extending this algorithm to three dimensions can be done in a variety of ways, the first implementation solved a knight's tour of a cube by adding an additional parameter to the algorithm, start position. It would solve the open knight's tour in the manner described above then take the final index on the returned index list and use it as the start position for a new board, and repeat this process N times. The culmination of this process would be an NxN cube that had a knight's tour "spiraling" down the interior. This is a fairly trivial solution to the three-dimensional problem, and does not actually address all the possible moves in a cube, it entirely omits the surfaces of the cube and worries about the interior. The true solution to this issue was to extend the moves matrix described above into a three dimensional cube that has the center cubic section acting as the start position and all the possible moves you can make being charted in the cube. The representation on the right shows all the red squares as possible moves that can be made from the center (3,3) position. Now you just run the algorithm using this cube instead of the matrix and you will have a solved cube that traverses both the surface of the cube and the interior.

SQUIRREL AND CULL:

Though Ira Pohl's tie-breaker seems to be thoroughly effective, it does not ensure that you will always find a tour or have potential to find every tour that exists in the board, there exists another well known solution to the tie-breaker scenario by Squirrel and Cull. This solution is a bit dense but provides a successful method of tie breaking. We know there are 8 possible moves for a knight on a chess board, Squirrel and Cull call a Move-Ordering a predefined sequence of these possible moves used to break ties systematically. These sequences prioritize certain moves over others when multiple options are available. (A tie) Squirrel and Cull predefined squares as switching points in the board where they would switch to a different permutation of move ordering to prevent loops and dead-ends. The processes to finding a tour using this method is as follows:

1. **Initialize the Board**: Start from a given position, usually the upper left-hand corner (1,1).
2. **Apply Initial Move-Ordering**: Use the first move-ordering sequence to resolve any ties that occur. Continue using this sequence until a predetermined switching square is reached.
3. **Switch Move-Ordering**: Once the knight reaches a specific square (the switching point), switch to the next move-ordering sequence. This helps to diversify the path and reduce the chances of creating a dead-end.
4. **Repeat Switching**: Continue this process, switching move-orderings at each predetermined square until the tour is complete or a dead-end is reached.
5. **Final Move-Ordering**: Use the final move-ordering sequence until the tour is complete.

By switching these sequences at specific points, the method improves the knight's tour's success rate, especially on larger boards. This approach helps navigate through ties more efficiently and reduces the risk of getting stuck, leading to more successful and efficient tours.

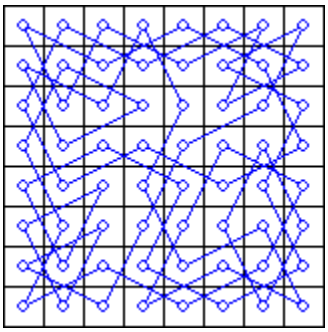
But how did Squirrel and Cull determine which permutations of possible moves to use and when to switch? In the case of the switching points, switching points are usually strategically selected to avoid creating loops or dead-ends. They are often chosen at significant milestones in the tour, such as corners, the center, or positions that tend to result in minimal accessibility. In some cases these points are determined with modulus arithmetic to adjust to the size of the board. As far as move-ordering permutations go, through empirical testing in smaller board you can determine which patterns provide better exploration of the board and lead the paths away from critical points and dead ends, there is not a lot of documentation on selecting move-ordering permutations but as long as the move-ordering permutation is not repeated at any of the subsequent switching points the pattern should be preserved.

NEURAL NETWORKS FOR KNIGHTS TOURS:

There is an interesting solution for this problem that utilizes a neural network in order to find tours. This method is not very effective and finds virtually zero tours in boards with large dimensions but it is sufficiently interesting that it is worth including. The neural network is designed such that each legal knight’s move on the chessboard is represented by a neuron. Which can be initialized to either 0 or 1 respectively if the neuron is 1 we consider it to be active, and if it is 0 we consider it to be inactive. Therefore, the network basically takes the shape of the *knight’s graph* over an NxN chess board. (A knight’s graph is simply the set of all knight moves on the board) If a neuron is active, it is considered part of the solution to the knight’s tour. Once

we begin the network each neuron that is active is configured such that it reaches a “stable” state on the condition that it has exactly two adjacent neighbor neurons that are also deemed active, forming a path; if this does not occur then the state of the neuron changes to inactive. As this cycle continues once the whole network is stable then a solution is achieved.

In the case the network becomes stable it returns sets of degree- 2 subgraphs that are themselves hamiltonian circuits, each neuron that is active in the stable state solution is marked subsequently with its move number and the circuits are patched together. Here is an example I got from Dmitry Bryant:



This solution was produced using a neural network and if you look closely you can identify the four independent circuits that composed this path. As the values of N grow, the network becomes increasingly bad at stabilizing. The creator of this visualization stated: “I obtained one knight's tour out of about 200,000 trials for n = 28 (three days' worth of calculation on my Pentium IV). Parberry wisely asserts that attempting to find a knight's tour for n > 30 using this method would be futile.” [2]

This solution is an interesting take on the problem using some of the new tools that are now available to us through Recurrent Neural Networks, however just as in the classical Warndorff implementation as values of N grow the success rate exponentially trends towards zero.

DATA ABSTRACTION:

This section will primarily be split into three subsections; main motivation behind abstraction, the implementation presented, and the freedoms allowed through abstraction. I will refer to tiles as nodes and the space they exist in as boards, but will use the terminology of graph theory on them.

Section I

First of all, what is data abstraction. Data abstraction is a term in computer science referring to the act of reducing a particular group of data into a simplified representation. This is done for two main reasons, first is to increase readability and comprehension for the programmer, and secondly it allows for data to be packaged and referred to in an organized way. In python, data abstraction can be done via the use of *classes*. Classes are user defined representation of data, which stores attributes and methods. Attributes are characteristics of an instant of that data type, and methods are functions which are specialized for that class's attributes. Through attributes and methods, coding can become much more simplified.

Section II: Node Class, Initializing Board, Algorithm Function

Node Class:

Attributes:

- Coordinate location on the board; an ordered list and in this implementation is an ordered pair. *Cord

- Tracker to see if the node should be included on the board; a boolean variable. *Valid
- The degree of the node, an integer. *Degree
- Nodes that are adjacent to itself - i.e. its neighborhood, a list of node data types. *Connections

Methods:

- Set_connection; takes a node data type as input and appends it to Connections
- Sort_connections; sorts the nodes in Connections by least to greatest depending on their Degree

Board Class: Initializing

Attributes:

- The array representation/input to create a board, list/list of list/list of list of list/... *Array
- Collection of all nodes marked as valid, list of nodes. *Valid Nodes
- Translation between a coordinate and the node given by that coordinate, dict, *Dictionary
- The number of valid nodes, *Order

Methods:

- Set_edge; takes two nodes as input, if both are marked as valid, use set connection on each node with the other

- Sort_nodes; sorts all nodes within Valid Nodes from least to greatest base on degree
- Set_board
    1. Make note of how many rows and how many columns
    2. For each row:
        a. For each column index:
        b. Create a Node given via the coordinate given by row and column index
        c. Add both to the reference dictionary
        d. If the coordinate value is marked within the input Array, mark Node attribute Valid as false
        e. Else:
            i. Append the node to Valid Nodes
            ii. Increment the order
            iii. Check to see any nodes with coordinates strictly above the node, lie within the range of the board. If so call Set_edge on the pair
    3. Call Sort_nodes

Board Class: Algorithm method

*The way this is done in python can be a little confusing to non-programmers, but there are different scopes of variables, local vs global. Because of this, sometimes there is a requirement for what is called a nested function, a function which is defined solely within the frame of another. For simplicity, I'll describe them as two separate functions.

- Longest_path
    - Returns the result of calling long with inputs: Valid Nodes, an empty set, Order
- Long -- Inputs: Options(list), Used(set), Wanted-length(integer)
    1. If Options contains 0 elements, return an empty list
    2. Create variables Longest-length(integer) and New-addition(list)
    3. For every Node in Options, if Valid Node
        a. Create variable path(list) containing Node
        b. Mark the node in Used
        c. Create variable available(list) of non-marked Nodes which are in Connections
        d. Call long with inputs: available, used, wanted-length
        e. Append the result to path
        f. If the length of path is greater than longest_length, update longest-length and new-addition to describe path
    4. Return new-addition.

*In a real application there would be a third step to run analysis, or display graphs, or do something generally based on the output from the method.

Section III:

This is a brief list of examples of things using an abstract representation can assist with. Of course all of this is still technically doable regardless of using abstraction, but this programming style makes it less difficult.

Ease of life:

- Readability is improved, seen very easily through the comparisons of implementations 1 and 2 that abstraction data lets you define how it should act. Therefore complications such as matrices can be ignored. It still does the same thing, but it's more intuitive thinking about a board as a board, rather than a board as a matrix.
- Quick edits to the board are available. Say you created a board but wanted to expand the board, or wanted to delete an edge. In other implementations this isn't possible and would require the creation of an entirely new board. However, in an implementation like this, doing that would be trivial
- Calling some algorithms is a lot easier. For instance, in Ira Pohl's tie break, usually one would have to search through the path in order to collect sums. However, one could very easily create a list of the values with this rule in mind upon initializing. This would look like the 0th index being the number of connections from path length 1, 1st index being the number of connections from path length 2, etc. This saves more time the more data there is, as incrementations like this take constant time, but traversing to find once already created takes polynomial time.

Flexible boards:

- Similarly to implementation 2, multidimensional boards are possible. They could be inserted manually- how it's done in 2, but they could take a lot of time. Luckily that's not the only way possible via this implementation. For example, one could create a method which intakes a collection of 2d boards, and connects the borders of them in order to create a 3d surface such as for a cube, updating the required edges.
- On the same note, these boards are no longer tied to euclidean space. Whereas other implementations required them to retain some form of a 'realistic' board, this is not

limited to that. Since it's an abstract representation, all that needs to be defined is the edges. For example one could make a mobius board by connecting two parallel borders in opposite directions, or connecting both sets of parallel lines, each pair matching in direction to create a torus board.

Enumerated Pieces:

- In a real chess game, it is not only an empty board and a knight, there exists many pieces, each with their own rules for how they may move. How would having these pieces on the board affect the possible paths? Well if the knight and the piece are the same color, then it can just be marked as invalid upon initializing the board. However, if they were on the opposite color, they would have tiles which they could capture on. Meaning that the piece containing tile must be captured first. An implementation is slightly technical so I won't cover in detail, but in essence one could create "symbols"(the pieces) which can be set to correspond with rules(ie how can the piece move), and then add an additional attribute, Locks, to the Node class which contains all the reasons the tile cannot be apart of the path(from initializing, being used in path already, a piece can capture it, and/or any other reason), and the tile is only valid if this attribute is completely empty. (Could implement via a new class instead also)

Works Cited:

1- “Knight’s Tours Using a Neural Network.” *Dmitry Brant*, dmitrybrant.com/knights-tour. Accessed 8 June 2024.

2- A Warnsdorff-Rule Algorithm for Knight’s Tours on Square ..., sites.science.oregonstate.edu/math_reu/proceedings/REU_Proceedings/Proceedings1996/1996Squirrel.pdf. Accessed 8 June 2024.

3- A METHOD FOR FINDING HAMILTON PATHS AND KNIGHT’S TOURS,

Ira Pohl 1967

https://www.researchgate.net/profile/Ira-Pohl/publication/220423597_A_method_for_finding_Hamilton_paths_and_Knight's_tours/links/02e7e51f421860def2000000/A-method-for-finding-Hamilton-paths-and-Knights-tours.pdf