PBL2 The Knights Tour

Abstract

Our joint report on the research and exploration into solving the knight's tour via Artificial Intelligence. This report highlights the methods involved, namely the use of State-Space Search, Bruteforce and Heuristics within our algorithms to truly understand the dynamics and possibilities of solving the Knights tour for a completed tour. Depending on the starting conditions of the problem in relation to the board dimensions and starting position of the knight. This report will highlight our findings directly correlated to the efficiency of our implemented algorithms compared to other well-known methods of finding a complete tour during these conditions and the methodology behind them with emphasis on the limitations and constraints of what we saw with the investigated processes and how dependant the algorithm is on the starting position of the knight with relation to the board dimensions.

Table of Contents

Introduction

The knight's tour problem consists in moving a knight on a chess-board in such a manner that it shall move successively on to every cell once and only once [1]. These tours can be either closed (re-entrant) where the knight ends on a square one move from where it started or otherwise open. Closed tours may only exist on boards with an even number of squares. Many mathematicians and chess masters have considered methods and identified tours including Euler, Warnsdorff and Roget [2],[3].

What are the methods of solving the tour?

By hand

In theory, it is perfectly possible to draw a 6 x 6 grid on a piece of paper or whiteboard and solve it manually by selecting a starting square and selecting all subsequent squares that are legal for the knight to move to. However, this is an arduous task that may take a long time to solve, and you are very likely to backtrack at many points. This was done by the famous mathematician Euler completing what is now known as a random walk tour [1].

Algorithms

State-space search

A more efficient and effective way of solving the Knight's Tour is making use of the state space search structure. This involves splitting up the chess board into nodes, or in the language of graph theory, vertices. Each vertex in a graph represents a unique square on the chess board, so that means we can then "point" each vertex to another vertex that represents a legal move for the knight at that position, otherwise known as the edges. If we do this for all vertices, we essentially have a "map" of the chessboard as a graph that the knight can utilise. This is effectively a depth-first-search.

Now that we have our graph, how can we use this to solve the Knight's Tour problem? As mentioned earlier, each vertex represents a unique square on the chessboard, so we can say that if we have a list that has every vertex inside of it, and the knight has only moved legally to each one, we have solved the knight's tour. So how does our knight know which vertex it can travel to from any given vertex it stands on? Aside from not going to vertices it has already visited, we need a way to decide the next vertex. This is where our heuristics come in, we have a choice of going with a brute force or Warnsdorff heuristic to determine the next vertex.

Artificial neural networks

When creating an algorithm to identify knight's tours there are varying methods for representing a chess board. A common representation is a graph where each square is a node and the set of moves are represented as edges connecting each node together. Takefuji and Lee propose the use of a neural network to represent this problem [4]. Each neuron in this network represents a legal knight's move, where the number of neurons for a square board of size n may be calculated using the formula n(n-1)/2, together forming a network akin to the previously described knight's graph. Each neuron has either an active or inactive state, represented by 1 and 0 respectively. Upon starting the network each neuron is assigned a random state, each neuron fluctuates between states becoming stable when two

neighbouring neurons are also active. Once all neurons are stable a solution has been found. This may either be one continuous path or multiple separate paths.

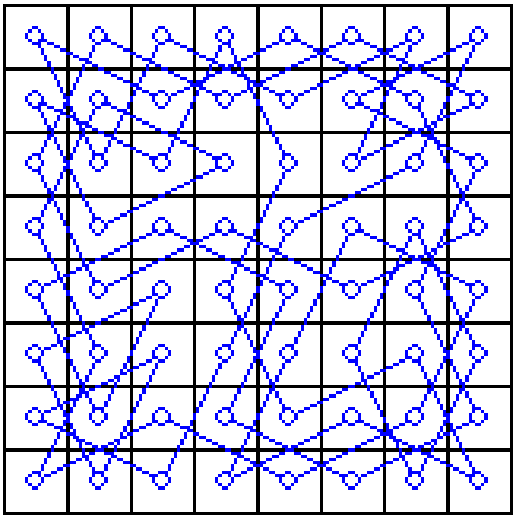

Fig. 1. A failed knight's tour generated using a neural network. It contains two separate disconnected paths [5].

An examination of a Takefuji-Lee Neural Network knight's tour is provided by Parberry, finding it to be significantly slower compared to Warnsdorff's algorithm. This issue arises as a result of failed solutions where multiple simulations must be run to identify a complete joined path [6].

Heuristics

When determining the next vertex to select, we can use either the brute force or Warnsdorff heuristics:

Brute Force

One possible way to select the next vertex is by brute force, what this means is that the knight picks the first available vertex without checking any other conditions. This is essentially doing the Knight's Tour by hand, but relegating it to a computer, which will deliver results much faster, however, the brute force method will stop being effective very quickly. At about a board size of 7 x 7, the brute force method may take many hours or even days to solve.

Warnsdorff heuristic

A more efficient method for identifying a knight's tour is Warnsdorff's heuristic. This heuristic works by identifying all the possible moves from the square the knight is currently on. It then does the same calculation for adjacent squares, moving to the square which commands the fewest possible moves. If two squares command the same number of possible moves it selects either one. This process repeats until all squares have been visited. Using this heuristic, the Knight's Tour can be solved significantly faster.

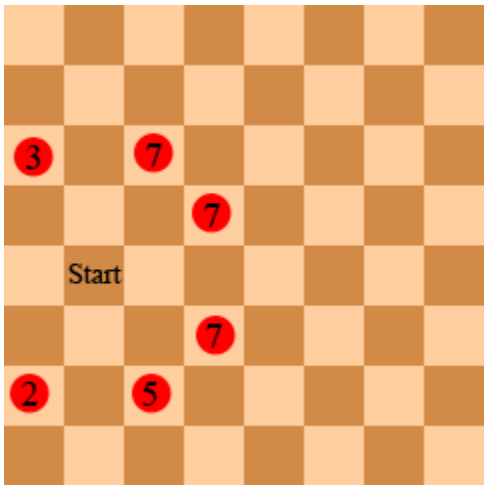

Fig. 2. Shows available moves for a knight on a square on a chessboard. [7]

Chapter 1 – our exploration into solving the Knight's Tour

Methodology

Depth-first-search:

The depth first search algorithm that we implemented starts at the starting vertex and traverses down a determined graph data structure until it cannot go any further, when this occurs and hasn't resulted in a completed tour it will backtrack and continue down another path until the completed tour has been found, this algorithm has been implemented alongside the Warnsdorff's heuristic to act as a more efficient program rather then just brute forcing through all the possible paths until a completed tour has been found.

Our algorithms:

In this report we will be examining two algorithms, one implemented by Norbert and the other by Anthony.

Norbert's algorithm:

Norbert's algorithm is an algorithm written in C# that uses State Space Search to solve the Knight's Tour. It uses Depth-First-Search, backtracking and a choice of Warnsdorff or brute force heuristics.

It starts by defining a Node class which has references to the next and previous nodes,

```
class Node
{
    public static int nodeCounter = 0; //Global counter for how many nodes exist
    3 references
    public int squareNum { get; private set; } //Number representing where on the chessboard this node exists, starts
    13 references
    public ChessCoords chessCoords { get; private set; } //refers to cartesian coordinates this node is located in
    16 references
    public List<Node> deeperVertices { get; private set; } //List of legal nodes reachable from this position
    5 references
    public List<Node> blockList { get; set; } //When backtracking from a node to this one, that node gets temporarily
    8 references
    public Node nextVertex { get; set; } = null; //Reference to next node on the Knight's Tour
    3 references
    public Node previousVertex { get; set; } = null; //Reference to previous node on the Knight's Tour

    public bool visited = false; //Have we been to this node yet?

    private int boardSize; //Size of the board, used to calculate adjacent squares

    public heuristicAction findVertex; //findVertex will point to the correct heuristic function to use

    3 references
    private int numberOfChoices
    {
        get { return deeperVertices.Count; }
    }
```

Fig. 3. Partial definition of the Node class from Norbert's program.

sets up a 2D array composed of node objects and then all the nodes in the array compute the legal moves from their position. So in a 6 x 6 board, the node at position [0, 0] will have references to the nodes at positions [2, 1] and [1, 2]. After this initialisation is complete, the starting node defined from a function argument is recognised as the first node, the program then enters a while loop where it will determine what the next legal node will be depending on the heuristic used. If there are no legal moves available, the program backtracks to the previous node, the node backtracked from will be temporarily added to a node-local block list so that the same node isn't visited again. Eventually the while loop's condition will be satisfied, that means the Knight's Tour has been solved, the program will then print out the results.

```
while (nodeCount != totalSpace) //While we haven't accumulated enough unique nodes
{
    nextNode = node.findVertex(); //Selection method determined by Knight's Tour argument

    if (nextNode != null) //If there are valid move options from the current node
    {
        node.nextVertex = nextNode; //Confirm the next node in our path
        node.nextVertex.visited = true; //it then becomes "visited" so we don't select this node in the future unless we backtrack over it
        node.nextVertex.previousVertex = node; //Set the reference to the previous node so we can backtrack to it if need be
        node = node.nextVertex; //move down the graph

        nodeCount++; //Increment the nodes we have as we have found a suitable one
    }

    else //If there are no valid move options from the current node
    {
        Node badNode = node;
        node.visited = false;
        node = node.previousVertex; //go back to the previous node
        node.nextVertex = null;
        node.blockList.Add(badNode); //Add the bad node to our block list so we don't go down this path again
        badNode.blockList.Clear(); //Clear the block list for the bad node as we are no longer there to make use of it

        nodeCount--; //We removed a node so we decrement
    }
```

Fig. 4. The main part of Norbert's algorithm. Will keep traversing until a valid path is found.

Example of the code being run with the starting position (0,0) and dimensions of 8x8:

```
Welcome to the knight's tour solver! Please enter the size of the board you'd like (Will be N*N)
N: 8

Please enter the starting x position of the knight, with 0 being the left-most side and 7 being the right-most side inclusive
X: 0

Please enter the starting y position of the knight, with 0 being at the top and 7 being at the bottom inclusive
Y: 0

Please enter the heuristic you would like to use, 1 for Brute force or 2 for Warnsdorff.
Heuristic: 2

boardSize: 8
startX: 0
startY: 0
Chosen heuristic: WARNSDORFF
```

Fig. 5. Setting up Norbert's program with correct settings.

```
Vertex 035: 003, [2, 0]
Vertex 036: 013, [4, 1]
Vertex 037: 007, [6, 0]
Vertex 038: 024, [7, 2]
Vertex 039: 039, [6, 4]
Vertex 040: 056, [7, 6]
Vertex 041: 062, [5, 7]
Vertex 042: 052, [3, 6]
Vertex 043: 058, [1, 7]
Vertex 044: 041, [0, 5]
Vertex 045: 026, [1, 3]
Vertex 046: 020, [3, 2]
Vertex 047: 030, [5, 3]
Vertex 048: 045, [4, 5]
Vertex 049: 060, [3, 7]
Vertex 050: 050, [1, 6]
Vertex 051: 033, [0, 4]
Vertex 052: 018, [1, 2]
Vertex 053: 035, [2, 4]
Vertex 054: 029, [4, 3]
Vertex 055: 046, [5, 5]
Vertex 056: 036, [3, 4]
Vertex 057: 021, [4, 2]
Vertex 058: 038, [5, 4]
Vertex 059: 044, [3, 5]
Vertex 060: 027, [2, 3]
Vertex 061: 037, [4, 4]
Vertex 062: 022, [5, 2]
Vertex 063: 028, [3, 3]
Vertex 064: 043, [2, 5]

001 016 035 032 003 018 037 022
034 031 002 017 036 021 004 019
015 052 033 046 057 062 023 038
030 045 060 063 054 047 020 005
051 014 053 056 061 058 039 024
044 029 064 059 048 055 006 009
013 050 027 042 011 008 025 040
028 043 012 049 026 041 010 007

Time to complete 1 Knight's Tour(s) of size 8 * 8 using WARNSDORFF: 0.001 seconds.
Number of loop iterations: 395
```

Fig. 6. Results from Norbert's program based on the settings from Fig 5.

For the solution to the knight's tour, Warndorff's Heuristic was implemented as it was simple and easy. Compared to a Neural network, it would require a network system to be created, trained and adjusted in order to give best performing output. For a problem like knight's tour, it was more time efficient to just optimise Warndorff's Algorithm.

Anthony's solution

For the implementation of the solution, python was used. Online resources were used to understand the concept, which helped construct the code [8], [9]. The code has 3 main components: If_move_valid, get_possibilities, and nextMove.

```
def if_move_valid(x, y):
    if ((x in range(0,n)) & (y in range(0,n))): # Check if it is within the bounds
        if (board[x][y] == 0): # 0 indicates that the position has not been visited
            return True
    else:
        return False
```

Fig. 7.

if_move_valid: This function assists by not letting the knight jump out of the chessboard dimensions, and also checks whether the square the knight is moving has been visited or not.

```
# Finds the number of possible moves from (x, y)
def get_possibilities(x, y):
    count = 0
    for i in range(8): # The knight can only make 8 different moves as defined by cx[] and cy[]
        next_x = x + cx[i]
        next_y = y + cy[i]
        if if_move_valid(next_x, next_y): # Only count valid moves
            count += 1
    return count
```

Fig. 8.

get_possibilites: This is a simple function that returns all the possible from a square a knight is on.

```
def nextMove(x, y):
    min_degree = (n+1) #default value of minimum degree will be set to 9.

    move_number = board[x][y] #tracks the order of the path taken by the chess piece to each square

    next_x = 0
    next_y = 0

    has_valid_moves = False

    for i in range(8):
        adj_x = x + cx[i] #get coordinates of adjcaent squares
        adj_y = y + cy[i] #get coordinates of adjcaent squares
        if if_move_valid(adj_x, adj_y): #check if it is unvisited & within chess board
            has_valid_moves = True
            degree = get_possibilities(adj_x, adj_y) #returns number of possible next moves from the adjacent square
            if (degree < min_degree): #select the square that will give the lowest degree
                min_degree = degree
                next_x = adj_x #obtain the move
                next_y = adj_y

    if (has_valid_moves): #if move is valid then add to number
        board[next_x][next_y] = move_number + 1
        movesX.append(x) # append to move lists
        movesY.append(y)
    else:
        return
    return nextMove(next_x, next_y) #loop til no more next move is found
```

Fig. 8.

nextMove: This is a recursive function, which traverses through all the possible moves and checks if they are valid, then visits the square with the lowest degree and reruns the function for which square the knight visits til there are no more to visit.

Chapter 2 – Our results and analysis

Results

Anthony's algorithm has some limitations, the main one being that if it does not instantly find a solution using Warnsdorff's heuristic, it will exit without finding a solution. On the other hand, Norbert's algorithm makes use of backtracking so for any board that has a valid solution, it will be eventually found but can take a while if the board is large.

Effectiveness and efficiency of the implemented Algorithms at 8x8 and 10x10 dimensions for a variety of starting points:

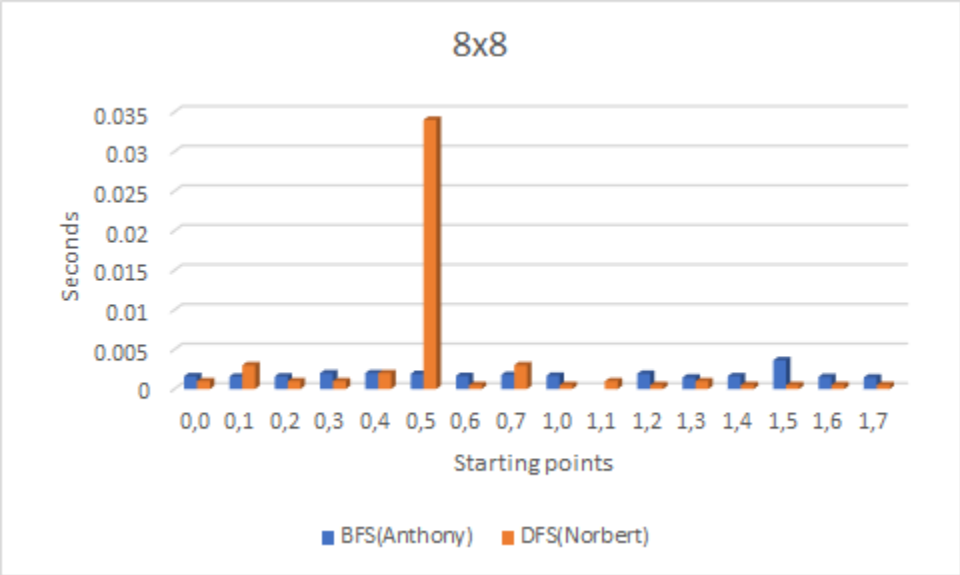

Fig. 1.

As can be seen on the 8x8 board, Anthony's algorithm was unable to find a completed tour starting on position 1,1 however Norbert's was able to. This therefore means that a Warnsdorf only Heuristic is not going to always find a completed tour from a starting position where it is possible to do so much like one that follows the Heuristic but has backtracking capabilities, this can also be seem for starting point 8,1 on the 10x10 board results visualised in fig 2.

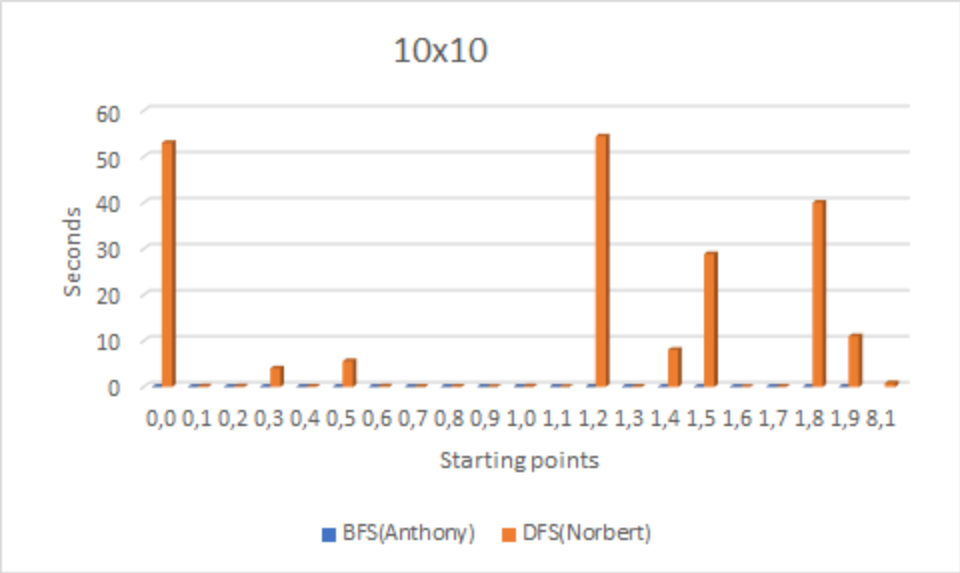

Fig. 2.

From the visualization of the results from the 10x10 in fig 2 it is clear that although Norberts algorithm is able to find a completed tour for every possible square where it is possible, the efficiency of the Algorithm is obviously much slower than anthony's although it is more precise.

Efficiency of the implemented algorithms as we progress onto higher dimensions:

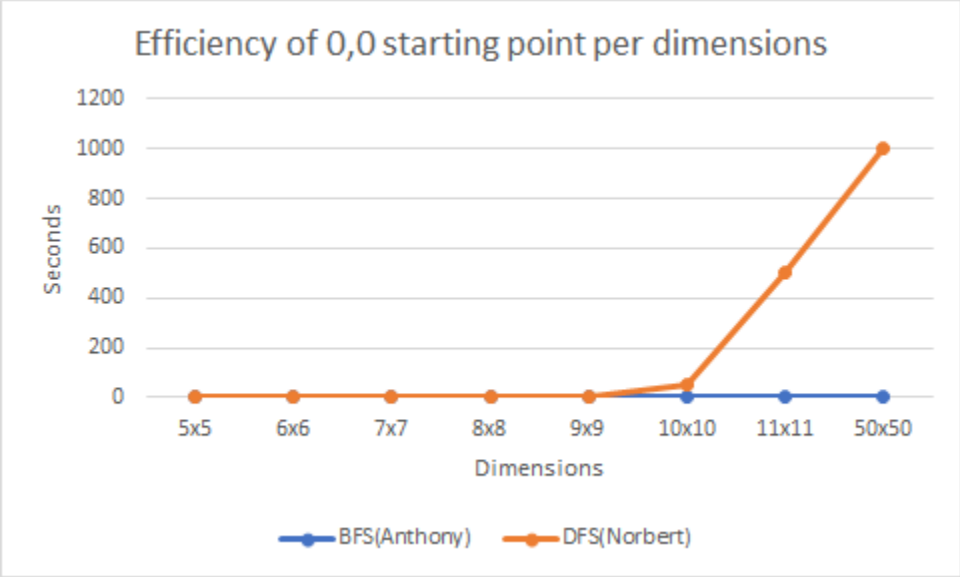

Fig. 3.

In fig 3 it can be see that Norbert's algorithm is capable of solving up to 10 x 10 boards starting at position [0, 0] (top left corner), with an 11 x 11 board possible if you start at [1, 1], all of this is only possible with the use of Warnsdorff's heuristic, with brute force the algorithm can't solve higher than 7 x 7 boards.

Anthony's algorithm can solve up to 50 x 50 being the max at a very efficient rate, the algorithm could also work on higher dimension boards however with this algorithm in its current state it cannot due to the recursive memory limits as it runs out of memory for anything higher than 50x50 dimensions.

Analysis

By using Warndorff's Heuristic, it allows more efficiency in the knight's tour program compared to solving the problem by hand or using brute force methods. When backtracking is implemented, the effectiveness of the algorithm improves. Backtracking does have drawbacks, however. Implementing this extra functionality will reduce the efficiency needed for larger chess boards. Our final solution does not implement backtracking, however it is capable of generating a solution to a 50x50 board in less than a second.

In our final solution, effectiveness is compromised over the efficiency of the algorithm. In one algorithm with backtracking on an 8x8 board, the coordinates (1, 1) should be working. However, these coordinates will not work in our final solution. Our algorithm does implement Warnsdorff's heuristic, which will fill in the edges and the corners, however there is no guarantee that it will fill in every square in the middle without running into a dead end. A board with backtracking is capable of solving every square on the board when N is even. However, there are limitations when N is odd. Under these conditions, when the knight starts on the coordinates (x, y), the solution will not work if x or y but not both is odd. The coordinates (1, 0) or (0, 1) will not work on a 7x7 chessboard, however (1, 1) will work.

A graphical user interface (GUI) to visualise our results was created using the Tkinter python package. The purpose of the GUI is to draw a representation of the Knight's path on a board, thus visualising a completed tour.

To do this, we create a tkinter canvas which will manage a collection of 2D graphics objects. Firstly, a grid of n*n squares are drawn to represent a board. We will then represent the path the knight will take. To do this, we need to keep track of all the coordinates as the tour is calculated. These coordinates are then passed into a line drawing function so that a line can be drawn from each point in order. At each of these points, a connecting 'node' is drawn as a circle.

Care has been taken so that the code is scalable to different sizes of n as input; with a large n, the board will scale down such as to maintain the same window size, and vice versa with a small n the GUI is able to be rendered in greater detail.

The final output of a successful tour will look like this:

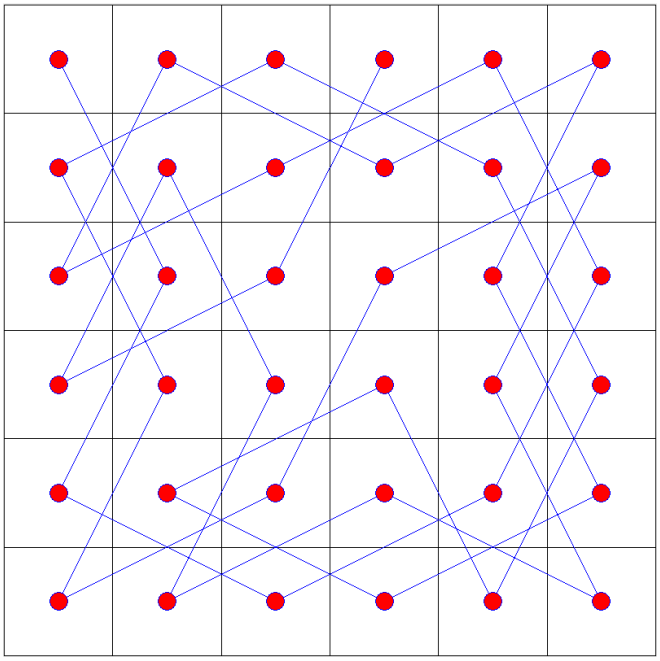

Fig: original GUI representing a completed tour on a 6x6 board.

The gui was used to verify our solution was working and we were able to study its behaviour and make comparisons to other algorithms.

Conclusions

From the limitations discussed in the analysis and findings of the results, it is clear that a Warnsdorff's heuristic Best First Search Algorithm is the most time-efficient compared to a Warnsdorff heuristic Depth First Search, with this in mind it is also evident that the Warnsdorff without backtracking will not find all possible completed knights tour as opposed to the Depth First Search. This clearly shows the differences between the two algorithms, when it comes to efficiency of solving the problem a Best First Search algorithm following Warnsdorf heuristic is the ultimate choice, however when it comes to solving for all the possible tours depending on starting position then a Depth First Search would be the best choice to do so.

References

[1]W. G. and W. Ball, "Mathematical Recreations and Essays", The Mathematical Gazette, vol. 3, no. 54, p. 256, 1905. Available: 10.2307/3603873.

[2]G. Jellis, "Rediscovery of the Knight's Problem", Mayhematics.com, 2021. [Online]. Available: http://www.mayhematics.com/t/1b.htm.

[3]G. Jellis, "Rhombic Tours and Roget's Method", Mayhematics.com, 2021. [Online]. Available: http://www.mayhematics.com/t/1c.htm.

[4]Y. Takefuji and K. Lee, "Neural network computing for knight's tour problems", Neurocomputing, vol. 4, no. 5, pp. 249-254, 1992. Available: 10.1016/0925-2312(92)90030-s.

[5]D. Brant, "Knight’s Tours Using a Neural Network", Dmitrybrant.com, 2021. [Online]. Available: https://dmitrybrant.com/knights-tour.

[6]I. Parberry, "Scalability of a neural network for the knight's tour problem", Neurocomputing, vol. 12, no. 1, pp. 19-33, 1996. Available: 10.1016/0925-2312(95)00027-5.

[7] Egheflin.com. 2021. Untitled Document. [online] Available at: <http://www.egheflin.com/professional/KnightsTourWiki.htm>

[8] GeeksforGeeks. (2017). Warnsdorff’s algorithm for Knight’s tour problem. [online] Available at: https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/

[9] GeeksforGeeks. (2011). The Knight’s tour problem | Backtracking-1. [online] Available at: https://www.geeksforgeeks.org/the-knights-tour-problem-backtracking-1/