KNIGHT'S TOUR PROBLEM USING BLIND AND HEURISTIC SEARCH ALGORITHM

A documentation presented to Mr. Lee Javellana

In Partial Fulfillment

of the requirements in Artificial Intelligence

Lim, Lyka Raquel C.
Oabel, Cliff Kenneth P.

May 2022

INTRODUCTION

A knight's tour is a classic chess problem in graph theory that was first posed around the 19th century, over 1000 years ago studied by great mathematicians including Leonhard Euler. It consists of a knight starting at any square, in this experiment, from d4 (index found in figure 01) and moving into the remaining 63 squares without landing on the same square twice. The knight piece must only visit each square exactly once following an L-shaped path.

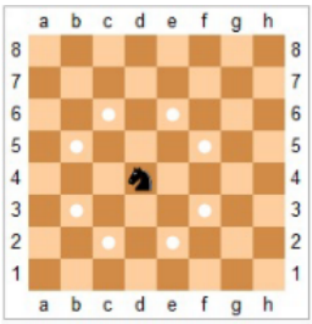


Figure 1: Knight's Tour

This problem is solved on an 8x8 chessboard and the possible tours for this chessboard is $1.305x10^{35}$ where the knight could end on any other square. This problem focuses on finding out whether there is a path from the beginning to end and how it can cover all the squares without revisiting them. There are two experiment parameters/algorithms that are used for this Knight's tour problem, the blind search and the heuristic search algorithm. For this experiment, we will be using JAVA as our main programming language and Eclipse as our Integrated Development Environment (IDE).

Blind Search Algorithm

Blind search or uninformed search is an algorithm wherein it operates in a brute force way and has no information about its domain. In this experiment, the researchers used the depth-first search algorithm wherein a stack is used to remember the last knight placement in case a dead end occurs in any iteration and uses the idea of backtracking. Backtracking is a

technique that was used for finding a solution using recursion, it means that if there are no more nodes in the current path, it will go back on the same path to find nodes to traverse. One step at a time is considered and if it does not satisfy the constraint then it backtracks it and checks for another solution. Figure 02 shows how the Depth-First Search Algorithm works.

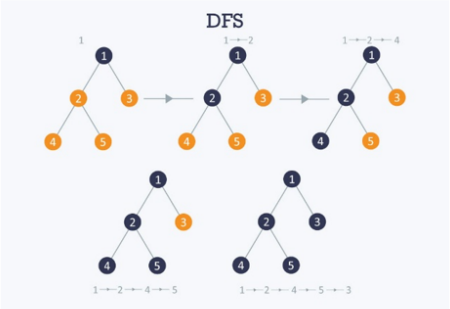


Figure 02: Depth-First Search

Heuristic Search Algorithm

Heuristic algorithm, on the other hand, is a search technique that is designed to decrease the time complexity by solving problems as quickly as possible but may not produce the best solution. Warnsdorff's rule is a heuristic search algorithm used by the researchers for finding the knight's tour. In this algorithm, there's a restriction added in which the knight must move to the square from which it will have the fewest onward moves, disregarding the count moves that revisit any square. Figure 03 below indicates the movement rule for Warnsdorff's Algorithm.

Figure 03: Warnsdorf's Rule

METHODOLOGY

Problem

For this experiment, our goal is to find out how different algorithms work on solving the knight's tour problem wherein it could solve legal moves made by a knight in a chessboard. A knight is placed in one block on an empty chess board, the rule is we must find a path wherein each square in the board must be visited by the knight exactly once. There will be two algorithms that will be used and compared in solving this puzzle namely, the blind search and the heuristic search algorithm. Provided below is the detailed step by step on how the experiments were run and the code's corresponding data structures.

Blind Search: Depth-First Search

Description of data structures

Variables

- int boardLength = 8; - Used to indicate that the chessboard has an 8x8 dimension.
- int initialX = 4 and int initialY = 3; - Indicates the initial positions of the knight in the chess board.
- int moveCount = 0; - Used as a counter so that the Knight could proceed to its next step
- boolean flag = false; - Flag for printing the initial chess board
- int side = 8; - Used as a counter to print files (column) and rank (row)
- int x = 0 and int y = 0; - Usually used to count the number of loops when implementing a For loop.
- long endTime; - Used to keep track of the program's end time.

Functions

- currentTimeMillis(); - Method that returns time in milliseconds
- InitializeChessBoard(); - Initializes board as a 2d array to simulate a chessboard
- SolveTour(); - This function is called in the main function. When called, this function uses the MoveKnightRecursively(); method to determine if there exists a solution, if there is a solution, it will print the board along with the knight's path and when there is no solution, it prints out a message indicating that there is no solution.
- PrintChessBoard(int board[][]) - Prints the board with the knight's path
- PrintInitialBoard(int board[][]) - Prints the initial board with the starting position of the knight indicated

- MoveKnightRecursively(int x, int y, int moveCount, int[][] board, int[] MoveX, int[] MoveY) - A recursive function where it takes parameters from the SolveTour(); method and check every possible moves whether it can do that move or not by checking if that cell is already visited.

Arrays

- int[][] chessBoard - Array containing the chessboard with a size of the initialized boardLength
- int board[][] - Containing the chess board with solution
- int moveX[] and int moveY[] - Movement patterns of the knight based on the changes in x and y coordinates.

Conditional Statements

- If Statements: a conditional statement that if "if" is true, the program will perform the function or display the information.

Loops

- While Loop: Loop wherein this condition is implemented if the program still doesn't get a solution.
- For Loop: Loop used to repeat a block of code given a specified number of times.

Algorithm (Pseudocode)

```
Initialize the starting point of the knight (startX and startY), the
board and its size (board[][]), and the movement tracker (moveN);

Main function(){
     Call SolveTour();  //Driver code;
     Print run time;
}

PrintChessBoard(board[][]){
     Print the board with the possible solution;
}

PrintInitialBoard(board[][]){
     Print the initial board where the knight will start its tour;
}
```

```
InitializeChessBoard(){
     Sets all value of all cells of the chess board to -1 to enable
backtracking
}

SolveTour(){

     Initialize and define the x and y array, its possible moves and
     its sequence of the knight;

     Calls InitializeChessBoard(board);

     Set the starting count of the knight

     Calls PrintInitialBoars(board);

     Calls the MoveKnightRecursively() and checks whether a solution
     is found

     Else Print the found solution and return true;

}

Boolean SolveRecursively (int x, int y, int moveN, int[][] board,
int[] MoveX, int[] MoveY){

     Initialize nextX and nextY;

      Checks if the move count matches with the board dimensions and
      if  true return true;

     determine the next move of the knight;

     Check whether the next move is a safe move, it will check if
     the cell is already visited and if it is on the edge of the
     board;

     If true place the move in the cell, if not backtrack;

     If all else fails, return false;

}
```

Heuristic Search: Warnsdorf 's Rule

Description of data structures

The implementation of this algorithm is contributed by Saeed Zarinfam obtained from https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/. A little modification was made for the tracking of runtime and printing of the chessboard. Below is the detailed description of the variables and functions used for this algorithm.

Libraries Used

- import java.util.Arrays: Imported so that the program will allow arrays to be viewed as lists.
- import java.util.concurrent.ThreadLocalRandom: Imported so that the program could generate random numbers.

Variables

- private static long startTime: Used to keep track of the execution's start time.
- long endTime: Used to keep track of the program's end time.
- public static final int N = 8: Used to indicate that the chessboard has an 8x8 dimension.
- int x and int y: Used to identify the board limits
- int i: Usually used to count the number of loops when implementing a For loop.
- int count: Used to get the number of iterations in a corresponding loop.
- int min_deg_idx: Used to indicate that the square is not yet occupied.
- int min_deg: Used to find the minimum degree adjacent from the knight's current position
- int start: Trying all N adjacent of (*x, *y) starting from this random adjacent.
- int sx and int sy: Indicates the initial positions of the knight in the chess board.

Functions

- public class warnsdorf: This function serves as the main function for this algorithm.
- public static void main(String[] args): This is the JAVA main method that serves as the entry point of the program.
- boolean limits(int x, int y): Function that restricts the knight to remain on its board.

- boolean isempty(int a[], int x, int y): Function that checks whether a corresponding square in the chessboard is empty or not.
- int getDegree(int a[], int x, int y): Function that returns the count of the empty squares adjacent to the knight's current position.
- Cell nextMove(int a[], Cell cell): Function that picks what the knight's next move using the warnsdorff's algorithm. This function will return false if there will be no possible move.
- void print(int a[]): Display the chess board with all the knight legal moves from the current time.
- void printInitial(int a[]): Function used to display the initial chess board indicating that all squares are empty except for the starting position.
- boolean neighbour(int x, int y, int xx, int yy): Function used to check the neighboring squares of the knight's current position.
- boolean findClosedTour(): Function used to generate the knight's legal moves. Returns false if no move is possible.
- class Cell: Constructor used to initialize objects.

Conditional Statements

- If Statements: a conditional statement that if "if" is true, the program will perform the function or display the information.

Loops

- While Loop: Loop wherein this condition is implemented if the program still doesn't get a solution.
- For Loop: Loop used to repeat a block of code given a specified number of times.

Arrays

- public static final int cx[] and public static final int cy[]: Movement patterns of the knight based on the changes in x and y coordinates.
- int a[]: Array containing all the elements of the chess board.

Algorithm (Pseudocode)

```
Import Libraries and Initialize all the required variable and arrays

import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;

public class warnsdorf{
//Main driver
public static void main(String[] args) {
      while(!new warnsdorf().findClosedTour()){
            ;
}
//Print run time;
}

boolean limits(int x, int y){
      //code that must restrict the Knight to be in the 8x8
chessboard
}

boolean isempty(int a[], int x, int y){
      //Code that checks if a square/block is occupied or not
}

int getDegree(int a[], int x, int y){
      //Code that returns the number of empty squares adjacent to
(x,y)
}

Cell nextMove(int a[], Cell cell){
      //Code that will pick the next move using Warndorf's heuristic.
If there's no more possible move, return false.
}

void print(int a[]){
      //Code that will print the chessboard with all the legal moves
it has obtained.
}

void printInitial(int a[]){
      //Code that will print the initial chessboard
}
```

```
boolean neighbour(int x, int y, int xx, int yy){
      //Code that checks the neighboring squares of the current
position of the knight.
}

boolean findClosedTour(){
      //Codes that generate the legal moves of the Knight. If there's
no possible move, return false
}

}
Class Cell{
      //Constructor
}
```

Indicated below is the summary of the Warsdorf's Algorithm,

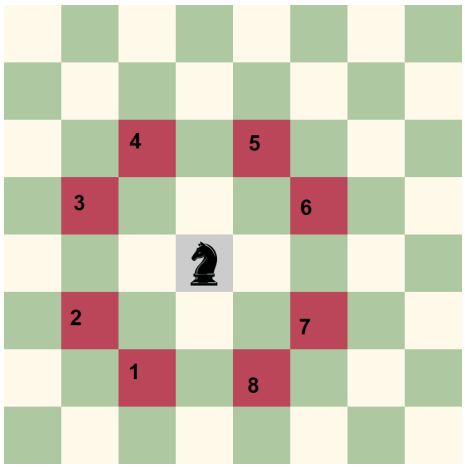
1. Set D4 to be the initial position of the knight on the board.
2. Print the board at D4 with the move number "1".
3. Find the next position which will give the least next unvisited positions.
4. Print the board at that position with the current move number.
5. Each square will be printed with the move number on which it is visited
6. Repeat until a solution is found.

Experimental Setup

For this experiment, two computer setups are used as each researcher compiles and executes the code. The setup are as follows: Computer A is a laptop with Intel i7-1065G7 that has a maximum clock speed of 3.9Ghz with 4gb of DDR4 RAM and computer B is also a laptop but with a lower processing unit having an Intel i3-7100u with a maximum clock speed of 2.40Ghz with 12gb of DDR4 RAM. Two computers will run instances of the code and will be recorded whichever finished first.

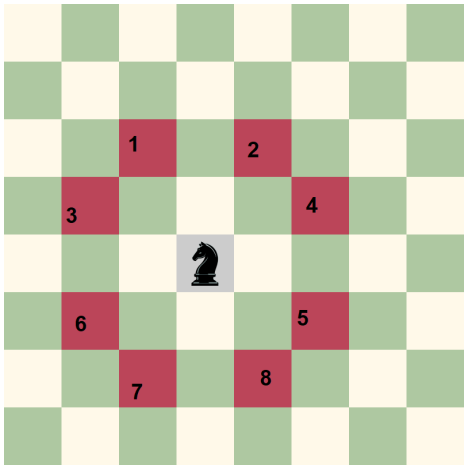
Each computer is installed with Eclipse IDE with the latest version of Java in order to have consistent performance to give more consistent results. It is hypothesized that the cpu can greatly impact the run time of the code. For each algorithm, 10 trials will be done enough to calculate the average times of both algorithms.

For the blind search algorithm, a set of movement patterns will be used in order to have varying resulting positions and time. The movement patterns are as follows:



The image above shows the sequence of moves that the knight will try, starting from the lower left and then going clockwise, as it tries out the possible neighboring moves. The logic of this sequence is, it would give the knight a more "blind" search because the pattern tries to move in a sequential manner when a move in invalid, thus a kind of brute forcing its way to find a possible solution. The idea that it starts at the bottom is so it could in theory, if it reaches a corner quickly, it has a quicker time on finding a solution. Other clockwise methods are also used with different starting moves, but unfortunately no results were gathered due to the possibility that it already caused an infinite loop. The sequence above is used for the experiment.

Another kind of sequence is used in the experiment, which is a modified version of an alternating sequence of moves.



The reason that the 5th and 6th choice is swapped is because that when the regular version is used, it runs significantly faster compared to the heuristic method that is used, thus

randomly swapping the two positions so it gives the algorithm some kind of “randomness” comparable to a blind search.

RESULTS AND DISCUSSION

Blind Search

Initial Chessboard:

XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	1	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX
XX	XX	XX	XX	XX	XX	XX	XX

BS - Trial 01

(Clockwise Sequence)

63	20	15	6	31	22	59	40
14	5	62	21	60	41	24	33
19	16	7	30	23	32	39	58
4	13	18	61	42	25	34	49
17	8	29	0	53	48	57	38
12	3	10	43	26	35	50	47
9	28	1	54	45	52	37	56
2	11	44	27	36	55	46	51

Runtime: 12212415 milliseconds

BS - Trial 02

(Clockwise Sequence)

63	20	15	6	31	22	59	40
14	5	62	21	60	41	24	33
19	16	7	30	23	32	39	58
4	13	18	61	42	25	34	49
17	8	29	0	53	48	57	38
12	3	10	43	26	35	50	47
9	28	1	54	45	52	37	56
2	11	44	27	36	55	46	51

Runtime: 10975077 milliseconds

BS - Trial 03

(Clockwise Sequence)

63	20	15	6	31	22	59	40
14	5	62	21	60	41	24	33
19	16	7	30	23	32	39	58
4	13	18	61	42	25	34	49
17	8	29	0	53	48	57	38
12	3	10	43	26	35	50	47
9	28	1	54	45	52	37	56
2	11	44	27	36	55	46	51

Runtime: 7286256 milliseconds

BS - Trial 04

(Clockwise Sequence)

63	20	15	6	31	22	59	40
14	5	62	21	60	41	24	33
19	16	7	30	23	32	39	58
4	13	18	61	42	25	34	49
17	8	29	0	53	48	57	38
12	3	10	43	26	35	50	47
9	28	1	54	45	52	37	56
2	11	44	27	36	55	46	51

Runtime: 10547652 milliseconds

BS - Trial 05

(Clockwise Sequence)

63	20	15	6	31	22	59	40
14	5	62	21	60	41	24	33
19	16	7	30	23	32	39	58
4	13	18	61	42	25	34	49
17	8	29	0	53	48	57	38
12	3	10	43	26	35	50	47
9	28	1	54	45	52	37	56
2	11	44	27	36	55	46	51

Runtime: 22037000 milliseconds

BS - Trial 06

(Clockwise Sequence (Top First))

51	48	53	46	23	2	19	10
54	45	50	35	20	11	22	3
49	52	47	24	1	18	9	12
44	55	34	41	36	21	4	17
59	40	43	0	25	8	13	28
56	33	58	37	42	27	16	5
39	60	31	26	7	62	29	14
32	57	38	61	30	15	6	63

Runtime: 362154 milliseconds

BS - Trial 07

(Clockwise Sequence)

63	20	15	6	31	22	59	40
14	5	62	21	60	41	24	33
19	16	7	30	23	32	39	58
4	13	18	61	42	25	34	49
17	8	29	0	53	48	57	38
12	3	10	43	26	35	50	47
9	28	1	54	45	52	37	56
2	11	44	27	36	55	46	51

Runtime: 11610706 milliseconds

BS - Trial 08

(Modified Alternate Sequence)

19	2	15	10	7	4	63	12
16	29	18	3	14	11	8	5
53	20	1	28	9	6	13	62
30	17	54	21	32	61	40	23
55	52	31	0	27	22	33	60
46	49	36	57	34	41	24	39
51	56	47	44	37	26	59	42
48	45	50	35	58	43	38	25

Runtime: 1108131 milliseconds

BS - Trial 09

(Modified Alternate Sequence)

19	2	15	10	7	4	63	12
16	29	18	3	14	11	8	5
53	20	1	28	9	6	13	62
30	17	54	21	32	61	40	23
55	52	31	0	27	22	33	60
46	49	36	57	34	41	24	39
51	56	47	44	37	26	59	42
48	45	50	35	58	43	38	25

Runtime: 2013263 milliseconds

BS - Trial 10

(Modified Alternate Sequence)

19	2	15	10	7	4	63	12
16	29	18	3	14	11	8	5
53	20	1	28	9	6	13	62
30	17	54	21	32	61	40	23
55	52	31	0	27	22	33	60
46	49	36	57	34	41	24	39
51	56	47	44	37	26	59	42
48	45	50	35	58	43	38	25

Runtime: 2002824 milliseconds

Heuristic Search using Warnsdorf ’s Rule

Initial Chessboard:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

HS - Trial 01

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

Runtime: 85 milliseconds

HS - Trial 02

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

Runtime: 66 milliseconds

HS - Trial 03

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

Runtime: 76 milliseconds

HS - Trial 04

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

Runtime: 70 milliseconds

HS - Trial 05

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

Runtime: 55 milliseconds

HS - Trial 06

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

Runtime: 83 milliseconds

HS - Trial 07

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

Runtime: 67 milliseconds

HS - Trial 08

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

Runtime: 82 milliseconds

HS - Trial 09

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

Runtime: 59 milliseconds

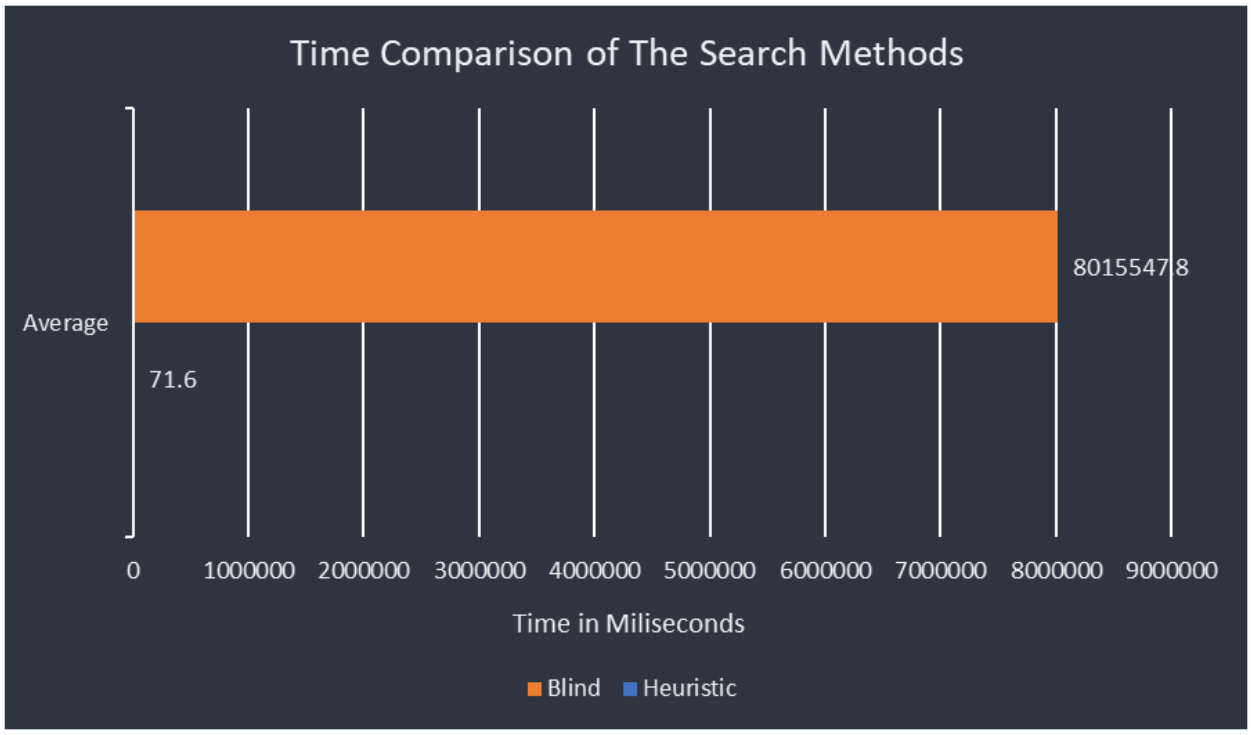
HS - Trial 10

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

Runtime: 73 milliseconds

Data Collection and Comparison

	Blind Search using Depth-First Algorithm Runtime	Heuristic Search using Warnsdorff's Algorithm Runtime
Test 01	12212415 ms	85 ms
Test 02	10975077 ms	66 ms
Test 03	7286256 ms	76 ms
Test 04	10547652 ms	70 ms
Test 05	22037000 ms	55 ms
Test 06	362154 ms	83 ms
Test 07	11610706 ms	67 ms
Test 08	1108131 ms	82 ms
Test 09	2013263 ms	59 ms
Test 10	2002824 ms	73 ms
Average	8015547.8 ms	71.6 ms



Discussion

The results were gathered and then tabulated to calculate each of the algorithm's run time after ten trials and here are the observations

In solving the blind search, as observed trials one to five and trial seven, it used the clockwise sequence prioritizing the moves at the bottom. Trial six had a different result with a significantly faster speed due to the fact that it is compiled with the other clockwise sequence but prioritizing the upper part of the possible moves it can do.

Based on the experimental results from the 10 trials, it was observed that the algorithm for each experiment greatly affected the running times of each trial. When solving the knight's tour problem applying the Blind Search using Depth-First Algorithm , the average running time is 8015547.8 ms or approximately 2.23 hours depending on the path that the researchers have initialized before the actual execution, while when applying the Heuristic Search using Warnsdorff's Algorithm, it only took an average of 71.6 milliseconds for the knight to complete its tour.

Aside from the two algorithms used which greatly affected the run time of each experiment, the researchers have also observed additional three factors that affected the program's compiling and running time.

1. The path the knight follows in blind search: trial 5 has the longest running time which was running for 22037000 ms or approximately 6.12 hours compared to trial eight which was running for 1108131 ms or approximately 18.5 minutes. The two programs have no difference in algorithm but what affected this runtime is its move sequences—where on board the knight makes its move. In trial five, the knight's move sequence was the clockwise sequence, which prioritizes moving downwards while in trial eight to ten, the knight used the modified alternating sequence. Since blind search is an uninformed search, it first continuously searches for a possible route before it explores another if ever the first route is impossible. The reason why trial eight happens to be faster than trial five is because it got lucky that it got its route fast because of its movement sequencing thus, in implementing blind search in solving for the knight's tour problem, how we initialize the movements in our code does matter.

2. It is observed that each instance of running the code, memory was not that quite big of an issue. RAM consumption ranged approximately between 200mb to 600mb per instance of the code and depending how long it is running. The image below shows one instance of code running and its RAM and CPU% consumption

3. Aside from testing different routes for the knight, the researchers have also tried to run as many- sometimes upto four instances of the program that have the same move sequences simultaneously and compared it to the running time of two instances that also have the same move sequences simultaneously. It turns out that the two programs had a faster running time compared to the four instances due to high CPU utilization. Therefore, the CPU performance of the computer affects the running time of a program.
4. Warnsdorff's heuristic algorithm is about 111327 times faster than the blind search in terms of its average run times. Which makes it clear that heuristic search outperforms blind search in an astounding amount.

CONCLUSION

Artificial Intelligence is continuously making its way up our world, computer scientists are in demand to make discoveries and inventions that will make changes on how we view the world and how technologies could affect our daily lives. AI doesn't just consider how it could make the lives of the people easier but also considers the time it could consume in doing something intelligently and how likely people will consider this implementation into their lives. In conclusion based on this experiment, it is shown that the Heuristic method which is the Wardorff's rule was more challenging to implement since it has to know the number of viable moves for each square where the knight visits compared to when using a blind search that doesn't have a domain knowledge to find a solution. Also, Blind search is a time consuming procedure that requires large memory and a lot of processing power which makes it not worthy to implement in real-life situations therefore, it is recommended to implement heuristic search due to the fact that

outperforms blind search in terms of speed, but a downside of a heuristic search as stated is that it needs prior knowledge on the environment, in this case is the chessboard in order for it to find solutions.