CMSC 162 - MACHINE PROBLEM 1
Blind and Heuristic Search Methods on the Knight's Tour Problem

Chua, Mary Elizabeth E.
Nisay, Deiondre Judd V.

Introduction

Knight's Tour is a sequence of moves made by a knight piece in a chess board such that the knight only visits each tile exactly once starting from an arbitrary position. The moves that the knight can only make will follow a L-shaped movement wherein it travels one tile along one axis and two tiles along another axis. In this experiment, we tested both the blind and informed searches to determine if *d4* is a valid starting position in an 8x8 chessboard.

Methodology

The programming was done using the C Language through a source-code editor namely Virtual Studio Code, or VSCode. There are 10 trials on each of the searches, totalling to 20 trials, and the runtime of each trial was collected manually through the repeating process of recording the runtime then running the search algorithm again. To fully optimize time-efficiency, we decided to separate the two search methods in different C files so that both can be run simultaneously, while recording each trial and comparing them to one another will be optimized as well. Furthermore, by separating the two C files, one may be run before the other, giving the user the opportunity to choose which method to execute first and avoiding delays.

To set up the program, the variables that will determine the tile positions and their order for the knight to travel in the chessboard are integers *mySelectSize*, *myVertical*, *and myHorizontal*, wherein *mySelectSize* is a user input that determines the chessboard size, then *myVertical* and *myHorizontal* determines the starting position of the knight, which will then be used as point of references for the search algorithm later on. Another variable that will play an important role in determining the sequence of moves the knight will make would be the 2D integer array *myLastPosition* where the size is determined by the number of total tiles in the chessboard with n size which will be determined in the next step. This number of tiles will be the number of rows while there will be two columns for the 2D integer array, which then all positions in the array are initialized to 0. One last array that plays a vital role in knight's tour will bythe integer *myCounter* which is used as a position value for listing down the sequence of moves the knight will be taking.

Next in the setup process in the knight's tour program, the user will be asked to input a chessboard size not greater than 9 tiles to simulate a realistic set up for the knight's tour while considering the efficiency in terms of runtime. Once a chessboard size is determined, the function *createChessBoard()* will execute which simply prints out the chessboard filling each tile with "XX".
The program will then ask the user to input a starting position, which in this case would be *d4*, and will print the initial state of the chessboard in separate text files, namely *blindSearch.txt* and *heuristicSearch.txt*, with the starting position in place labeled with "01". After the user has inputted a starting position, a timer starts which is used to record the total runtime for a search algorithm. Since the origin of the chessboard is located on the bottom-left corner in contrast to the origin of a 2D array, which is located on the top-left corner, we implement the function *initialPosition()* to convert our alphabet positions into their number counterparts relative to a chessboard. After the setup process, we then start the blind searching.

```
Part 1: Knight's Tour Using Blind Search

INITIAL STATE:
-----------------------
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 01 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
```

Figure 1: Initial state of the chessboard with starting position *d4*

blindSearch()

The function *blindSearch()* is a depth-first search algorithm wherein it uses recursion to find different pathing combinations, in this case the knight movements, along branches in a tree for as long as possible and backtracks when no solutions are found along the path it took.
The integer *myCounter* also has another purpose which indicates the depth level of this function.
The integers *myHorizontal* and *myVertical* are the points of references that indicate the current position or tile of the knight in the chessboard.

The integer *knightMovement* is used in 8 while loops and each of them, at different *knightMovement* values, dictates which path the knight should take, from its current tile, under certain conditions such as the tile it plans to visit should never go beyond the chessboard and it shouldn't already be visited, whose value at that tile should be 0. If the path the knight plans to visit deems to be invalid, the *knightMovement* value will increase and go to the next while loop that matches the *knightMovement* value. However, if the path the knight plans to visit meets the conditions, the tile to be visited will be inserted to the *myLastPosition* array relative to the depth level and will now be the current tile, in which that current tile in the chessboard will be given a value of 1, meaning that it's a tile that has been already visited. Afterwards, the function *blindSearch()* will call itself, creating a recursion in the code. Pseudocode of the procedure is as shown below in Figure 2:

```
blindSearch(parameters)
     knightMovement = n
     While knightMovement == n do
          Move 1 along one axis from current tile
          Move 2 along on other axis from current tile
          If new tile exceeds chessboard border
               knightMovement = n+1
          Else if new tile is already visited
               knightMovement = n+1
          else
               Insert new tile position to
               myLastPosition array
               Mark new tile in chessboard as already
               visited
               current tile = new tile
               blindSearch(parameters)
               knightMovement = n+1
     End While
     While knightMovement == n+1 do
          ...
```

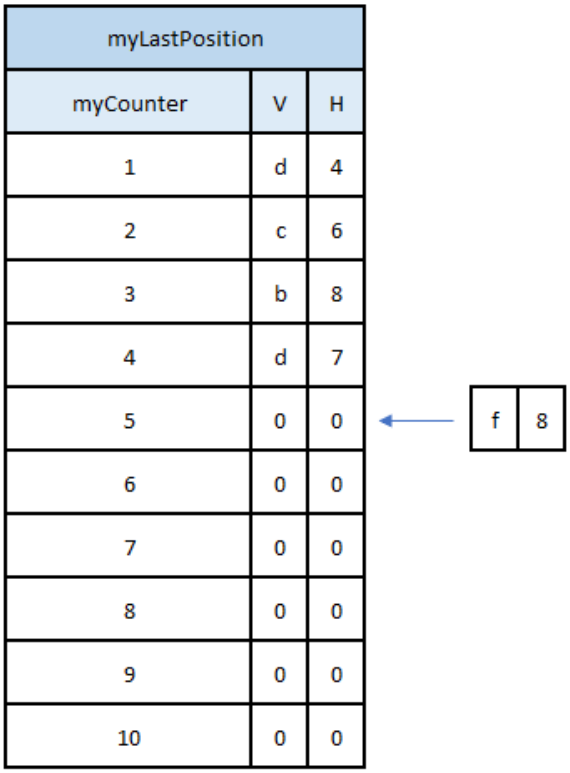
Figure 2: Pseudocode for determining the tile that the knight will visit next

When the program has exhausted all its plan-to-visit tiles, the *knightMovement* value will be given a 9 which would then be given two choices to go through with certain conditions. If the depth level, which is the *myCounter* value, is greater than 1, the program will revert the values of *myHorizontal* and *myVertical* to the values they were on

that level, then backtrack to the while loop on where it last left off before it recursed. To add to that, before it has exhausted all of its options, if there has been any valid moves in that certain depth level, we will set the last recorded tile position in the chessboard to 0. We then subtract *myCounter* by 1 to match the depth level of the level the program will backtrack to. Since we're backtracking to a previous depth level, this means a recursion happened, which then points to the fact that the knight's move leading to this recursion was a valid move. Therefore, we will also set the tile position of the chessboard on which the knight moved, which led to the recursion, to 0.

If the *knightMovement's value* at that level isn't 8 yet, it will increase its *knightMovement* value and move on to the next while loop and repeat its process as shown in Figure 2. This will then overwrite the values in the *myLastPosition* array at that depth level.

On the other hand, if the *knightMovement* value is 9 while the *myCounter* value is equal to 1, this means that the program has exhausted all possible sequences of tiles the knight can visit which will deem the knight's tour unsolvable using the starting position. Pseudocode of the procedure is as shown below in Figure 4:



myLastPosition		
myCounter	V	H
1	d	4
2	c	6
3	b	8
4	d	7
5	0	0
6	0	0
7	0	0
8	0	0
9	0	0
10	0	0

f	8

Figure 3: Adding a tile position to the *myLastPosition* array relative to *myCounter*

```
...
current tile = new tile
blindSearch(parameters)
```

```
            knightMovement = 9
If knightMovement == 9 && myCounter > 1
     If a valid move was found at depth level
     Position value of valid move in
     chessboard = 0
     myCounter = myCounter - 1
     Position value of previous valid move in
     chessboard = 0
     Current move reverted to its value before
     valid move in previous depth level was made
Else if knightMovement == 9 && myCounter == 1
     Print "Knight's Tour unsolvable with this
     starting position!"
...
End If
```

Figure 4: Pseudocode for checking whether the program backtracks or terminates

To check whether the knight has visited all the tiles exactly once, in every while loop as shown in Figure 2, it will call the function *checkIfZero()* and insert the integer *knightMovement* as a parameter which then the function will go through all the tiles in the chessboard in search for a value of 0. If it succeeds in finding a tile with a value of 0, the value of *knightMovement* won't be changed and the program will continue as scheduled. However, if the function goes through all the tiles in the chessboard and fails to find a tile with a value of 0, the value of knightMovement will be changed to 0. This would mean it would skip all the while loops and head straight to the end of the function and begin the repetitive backtracking until it reaches depth level 1, or the value of *myCounter* reaches 1. This event would always occur at the depth level, or the *myCounter* value, which is equivalent to the number of tiles in the chessboard. Pseudocode of the procedure is as shown below in Figure 5:

```
checkIfZero(knightMovement = n)
     Checks all positions in the chessboard array
     If an array position exists containing 0 then
          knightMovement = n
     Else
          knightMovement = 0
     End if
```

Figure 5: Pseudocode for determining if Knight's Tour is solved

The array of values in *myLastPosition*, which is the sequence of tile positions the knight will visit to solve the knight's tour, will then be given an ordered number wherein the first tile position in the array will be given a value of 01 in the chessboard, indicating the starting position, the last tile position in the array will be given a value of 64 in the chessboard, indicating the last tile position the knight will land on to solve the knight's tour, and the tile positions in between will be given increasing ordered numbers according to the tile positions in the array *myLastPosition*. After all the tiles in the chessboard are assigned to their respective ordered numbers, the program will then call the function *printBlindSearch()* which would print and append the final state of the chessboard in the same text file, *blindSearch.txt*, as the initial state, together with the runtime for the blind search algorithm.

```
z=1
For each row in myLastPosition i
     For each column in myLastPosition j
          If j == 0
               y = vertical position of tile
          Else
               x = horizontal position of tile
               chessboard[x][y] = z
               z=z+1
          End If
     End For
End For
Print chessboard
```

Figure 6: Pseudocode for assigning ordered numbers to the tile positions in the chessboard

To have an overview and simple explanation of each function, we have:

1.) createChessBoard(int mySelectSize, int** chessBoard)

- This function initializes the chessboard by setting each tile to "XX" after the user has inputted the chessboard size

2.) initialPosition(int mySelectSize, char myStart[2], int *myVertical, int *myHorizontal, int** chessBoard, int** myLastPosition, File *bs)

- After the user has inputted the starting position for the knight's tour, the function will first convert the input string into integer values to be placed on the chessboard array while also inserting the integer values to myLastPosition array wherein this array lists down the sequence of movements of the knight for the knight's tour. This function will also write and print the initial state in a text file namely blindSearch.txt.

3.) blindSearch(int mySelectSize, int** chessBoard, int** myLastPosition, int *myHorizontal, int *myVertical, int *myCounter, FILE *bs)

- This is the program proper where the depth first search is being implemented and executed. Since this is a blind search, all the movements for the knight are placed in order wherein the program will check the first movement option to see if it's valid. if the first movement is invalid, it will go to the next movement option to see if it's valid. However, if it's a valid move, it then places the knight in the new tile and calls the blindSearch() function again creating a recursion in the code. However, if it recurses but doesn't find a valid move in the function, it'll backtrack and erase the values it recorded during that recursion or simply overwrite it. After the program has exhausted all the possible sequences of movements through the function checkIfZero(), it will tell the user that the starting position given to the program will not be able to generate a Knight's Tour. On the other hand, the moment the depth-first search finds a valid move for the last tile in the chessboard unvisited, it will mark that tile as visited and the function checkIfZero() will confirm that all tiles have been visited exactly once. It will then record the knight's movement then backtrack until depth level 1. This will then record the sequence of movements to the chess board.

4.) checkIfZero(int mySelectSize, int** chessBoard, int *myMove)

- This function checks if all the tiles in the chessboard have been visited exactly once by checking all the tile positions of the chessboard array to see if they have a value of 0, which is unvisited, or 1, which is visited. To determine if the knight's tour is solved, the function should not be able to find a value of 0.

5.) printBlindSearch(int mySelectSize, int** chessBoard, FILE *bs)

- This function simply appends and prints the final state of the chessboard in the text file namely blindSearch.txt.

heuristicSearch()

On the other hand, heuristic search is a type of method that uses measures in order to make decisions for the next steps in the code. The particular heuristic search algorithm that is used in solving the Knight's Tour problem is called the Warnsdorff's Rule. Instead of blindly searching each and every possible route that the knight piece can take, the Warnsdorff's rule dictates which route to take which would result in a better chance of finding a single knight's tour. With this method, from the starting position of the knight, the code will take note of every next possible move which should be at least 2 and at most 8. Of all of these possible moves, the chosen move will be the one that has the least number of next possible moves from that future position (of course, not counting the squares that have already been visited).

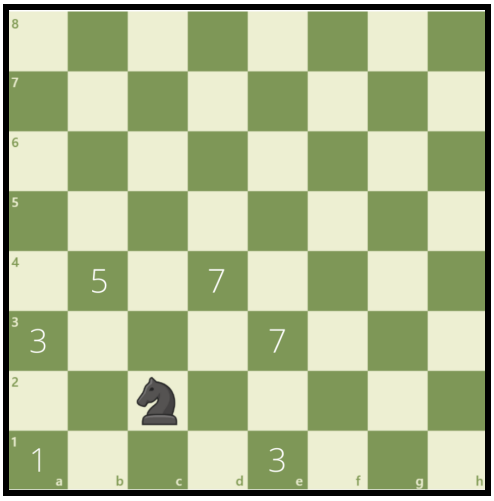


Figure 7: Chessboard Showing the Knight's Number of Possible Moves on the Next Move

For example, in figure n, the knight piece is on position *c2*. Its next possible moves are *a1, a3, b4, d4, e3,* and *e1*. The number of next possible moves for the next moves are 1, 3, 5, 7, 7, and 3, respectively. Using the Warnsdorff's Rule, the knight must move to *a1* since it has the least number of next possible moves. Let's visit the pseudocode for this algorithm.

```
Enter size of chessboard
Enter starting position
Initialize chessboard
Start from starting position
While there are still unvisited squares
      If there are possible moves,
            Get next move with least number of next moves
            Jump to the square
            Mark square visited
      Else
            Backtrack until there are possible moves
                  If backtracked until back to start
                  Then exit and print that the algorithm failed
Endwhile
Print to text file the solution
```

Figure 8: Pseudocode for Heuristic Search using Warnsdorff's Rule

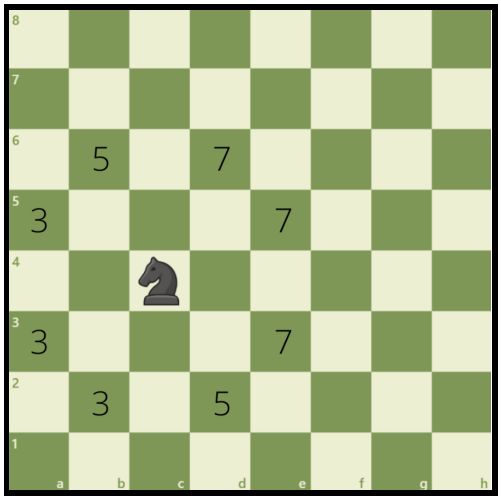


Figure 9: Chessboard Showing the Knight's Number of Possible Moves on the Next Move with

Another example of applying the Warnsdorff's Rule can be seen on figure n. However, this poses a problem since there are three possible moves with the least number of next possible moves. With our current algorithm, it doesn't have a function to break the tie yet and it will just go with the first possible move that is written in code. The Warnsdorff's Rule is built upon the fact that the piece will have a better chance of finding the single knight's tour if it goes through squares that have the least number of next moves in every iteration. An extra measure here is that added weight is on the next move with the next move with the ***next*** move that has the least number of next possible moves. This may apply even to the moves after the next move after the next move, after the next move, and so on, so forth. Breaking this tie seems to have another heuristic function in and of itself, but another way of approximating which of the tied moves should the knight go next is using Arnd Roth's proposition, this states that among the tied moves, the successor must have the largest Euclidean distance from the center of the board. This follows that the next moves from the next moves will have a smaller number of possible moves than the other choices. Adding a few more lines of code to our pseudocode earlier we have:

```
Enter size of chessboard
Enter starting position
Initialize chessboard
Start from starting position
While there are still unvisited squares
     If there are possible moves,
          Get next move with least number of next moves,
               If there are ties,
                    Choose square farthest from the center
```

```
                Jump to the square
                Mark square visited
          Else
                Backtrack until there are possible moves
                      If backtracked until back to start
                      Then exit and print that the algorithm failed
    Endwhile
    Print to text file the solution
```

Figure 10: Pseudocode for Heuristic Search using Warnsdorff's Rule and Arnd Roth's Proposition

Now that we have the final pseudocode, the code heuristicSearch.c may be explained better. The source code was borrowed from Jwalin Bhatt's code, and was slightly changed so that the conditions of the machine problem can be met, such as changing the input of the user, removing some print features, and then reconstructing the order of some of the functions. We have 4 functions including the main function in this C program:

1.) Void jump (int d)
- This function will pass the horizontal and vertical values for the knight to move to. Only two values will be passed to *d*. 1 and -1. 1 will have the piece move normally, -1 will have the piece move the opposite way.

2.) Int count()
- This function will count how many next possible moves are on the current next move square.

3.) Int heuristicSearch()
- This function entails the whole heuristic search algorithm with Warnsdorff's rule, this first checks what are the next possible moves using jump() function and checking if it jumped to a valid square, then counting the next possible moves for that particular square using count(), and then calculating the euclidean distance and then checking if it has found the new position that has the least next possible moves and is the farthest from the center. Otherwise, if there are no valid moves, it will backtrack. This function will continue until there are no more moves left.

4.) Int main()
- This function prompts the user to enter the size of the chessboard and their starting position. This also initializes the chessboard, and calls the heuristicSearch() function.

To further explain, the following are snippets of code ordered in chronological order after the user enters an 8x8 chessboard with a starting position of *d4*.

```
int main()
{
...
//initialize the chessboard
    for(x=0;x<n;x++)
        for(y=0;y<n;y++)
            G[x][y] = -1;
```

The code will first create a chessboard of the size entered by the user via a 2D array G[x][y] and set all values to -1 indicating that each square with the value -1 means that it hasn't yet been visited. Afterwards, the program will ask the user to enter the starting position, which in this case would be *d4*. The main function will then interpret this "*d4*" string to actual indices on the 2D array; in this case, the position of the knight is actually on G[3][4], or x=3 and y=4. We then call heuristicSearch().

```
int heuristicSearch(){
...
for(s=1;;s++){
        G[x][y]=s;
        min=n;
        d=0;
```

In the heuristicSearch() function, there will be a for loop that will count the valid moves in a single knight's tour and save it to G[x][y]. Since this is the first move, the 2D array G[x][y] will be able to save the first move making it G[3][4]=1. The variable min will be set as equal as the size of the board, 8. This will be used to compare if we have a new least number of next moves on a possible square.

```
      for(M[s]=0;M[s]<8;M[s]++){
            jump(1);
            if(x>=0&&x<n && y>=0&&y<n && G[x][y]==-1){
                ...
            }
            jump(-1);
      }
```

Next, we have a nested for loop within the previously introduced loop. This checks all of the possible moves of the piece. It first jumps to every possible square it can move to at

the given instance, and then checks if it's out of bounds, and if it's already visited. If it isn't, it merely jumps back to its original position.

```
            if(x>=0&&x<n && y>=0&&y<n && G[x][y]==-1){
                c=count();
```

However, if the move is proven to be within bounds and has not been visited, then we can go inside the if condition and do some more calculations. The count() function is called and will be stored in the variable c.

```
int count(){
    int count=0;
    if((x-1)>=0 && (y+2)<n && G[x-1][y+2]==-1)count++;
    if((x-2)>=0 && (y+1)<n && G[x-2][y+1]==-1)count++;
      ...
```

The count function basically counts the number of possible next moves in that instance. This accounts for the 8 total possible moves, it checks if the next positions would be within bounds and if it is unvisited. Then it increments every time it finds a valid next move. Now, going back to our previous function:

```
                dist=(x-n/2)*(x-n/2)+(y-n/2)*(y-n/2);
                if(c<min || (c==min&&dist>min_dist)){
                    min=count();
                    d=M[s];
                    min_dist=dist;}
```

Now that we have counted the number of possible next moves for this specific square and saved and kept it safe in variable c, we can move on to measuring its euclidian distance from the center of the chessboard. Next, we start comparing the counts and the distance, but since we're on the first iteration, we have no other squares to compare it yet to. Nonetheless, the code still holds, and in the later iterations where we have squares to compare, it will check if there are new minimums and also check if there are ties. If there are any ties, it will then check the distance if it is larger than the distance of the square from the previous iterations. This is a type of search algorithm wherein it checks if the value is lower than the old value that it is holding, then replaces that old value with the new value since it is our new lowest number. Then it will traverse through the list of numbers until it finds another lower number than the value that it's holding, then replaces it again.

Going back to the algorithm, whenever the condition finds a new lowest value, it saves that particular square position, the count, and its distance from the center of the chessboard. After traversing through the possible moves, it will go back to its position.

```
            jump(-1);   }
        if(min==n)
            break;
        M[s]=d;
        jump(1);}
```

Now, if the algorithm couldn't find a new min value (min==n) (meaning that it never really changed its value since the start of the search), it means that there are no more possible moves, and it must break out of the for loop or it might run in an infinite loop. This could either mean that the code has found a solution, or that the problem cannot be solved and the knight piece is stuck. One way to figure this out is to check if the value of s (the variable that checks how many moves the piece made) is equal to the number of squares on the board, in our case it's n*n = 8*8 = 64.

```
    if(s+1==n*n)
        printf("\nsolution found");
    else
        printf("\nstuck");
```

If indeed our min value was actually changed, then it is guaranteed that we have a next move to get to. We then save the value from d to M[s] (our saved position earlier) and then jump to that position using the jump() function. Then we've reached the end of the iteration and proceed to the next until we find a solution or a dead end.

The rest of the code just prints out the solution by using a nested for loop and printing out the G[x][y] 2D array. The runtime is also provided afterwards.

Results and Discussion

This section presents the results of the experiment tests on the two generated algorithms. The experimentation covers the given parameters from the machine problem: starting position: *d4* and size of the chessboard: 8x8. Furthermore, the trials were conducted 10 times for each search method, giving us a total of 20 trials for the whole experiment. For each evaluation, the runtime is accounted for. The following are the tabulations of the findings:

Table 1: Tabulated Results of the Runtime of Each Trial under Blind Search Method

Trial	Runtime	
1	1 hour, 24 minutes 8.1143 seconds	5048.1143 seconds
2	1 hour, 24 minutes 2.514 seconds	5042.514 seconds
3	1 hour, 24 minutes 14.314 seconds	5054.314 seconds
4	1 hour, 24 minutes 14.476 seconds	5054.476 seconds
5	1 hour, 24 minutes: 19.781 seconds	5059.781 seconds
6	1 hour, 24 minutes: 50.766998 seconds	5090.766998 seconds
7	1 hour, 26 minutes: 29.587000 seconds	5189.587 seconds
8	1 hour, 27 minutes, 4.326 seconds	5224.326 seconds
9	1 hour, 24 minutes, 53.94699 seconds	5093.94699 seconds
10	1 hour, 25 minutes, 38.59 seconds	5138.59 seconds
Average	1 hour, 24 minutes, 55.313965 seconds	5095.314032 seconds

As you can observe, given how the depth-first search algorithm was structured, the runtime for finding the knight's tour with the starting position *d4* is averaged at 1 hour, 24 minutes, and 55.31 seconds with the minimum runtime being at 1 hour, 24 minutes, and 2.514 seconds while the maximum runtime being at 1 hour, 27 minutes, and 4.326 seconds. During the trials, we noticed that the runtime slowly increased per iteration. This is most likely due to rerunning the program consecutively after recording each runtime per iteration. It is without a doubt that there are more efficient algorithms out there with better structuring to minimize runtime. However, a program in this structure that results in a runtime of 1.5 hours, with a starting position of *d4*, using depth-first search, we can confidently say that this algorithm is decent, or even above average at most.

Table 2: Tabulated Results of the Runtime of Each Trial under Heuristic Search Method

Trial	Runtime
1	0.00000000 seconds
2	0.00000000 seconds

3	0.00000000 seconds
4	0.00000000 seconds
5	0.00000000 seconds
6	0.00000000 seconds
7	0.00000000 seconds
8	0.00000000 seconds
9	0.00000000 seconds
10	0.00000000 seconds
Average	0.00000000 seconds

Observing the tabulated runtime of each trial in heuristic search method, every trial is consistently at 0 second, almost instantaneously giving the solution as the user hits enter. During the experimentation and trial period, the number of decimals in the function that runs the runtime tracker was increased to 1000 decimal places to see if the trials really are consistently at 0 second runtime, and every digit was 0. It is certain that this is impossible since there must be at least some microseconds of runtime, but we will keep this an open-ended question.

Conclusion

The purpose of this experiment is to test blind and heuristic search methods to solve Knight's Tour problem using C language. For the blind search program, depth-first is utilized in order to find the single knight's path. For the heuristic search program, Warnsdorff's Rule is implemented along with Arnd Roth's Proposition. With the starting position of *d4* on an 8x8 chessboard, the average runtime for the blind search algorithm for the ten trials is 1 hour, 24 minutes, 55.313965 seconds or 5095.314032 seconds while the average runtime for the heuristic search algorithm for the ten trials is 0 seconds.

With these results, we can conclude that heuristic search is significantly more time-efficient than blind search.

References

SOURCE CODE: Bhatt, J (2020) Knight_s_Tour.c [Source code]. https://github.com/Jwalin1/C-programs/blob/master/Algorithms/Knight_s_Tour.c