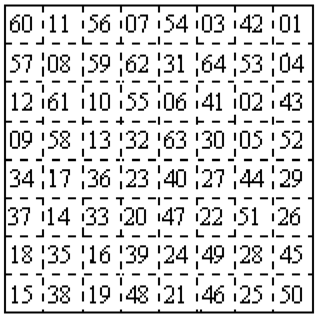
Kefil Tonouewa
CSCI 362
Instructor: Dr Snehasis Mukhopadhyay

Project Report

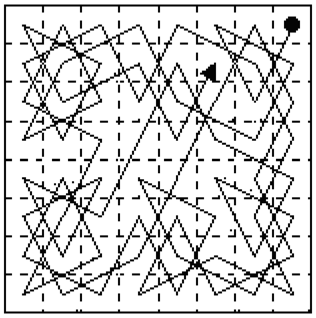
The knight Tour problem is a very common problem among the enthusiastic of chess. For this assignment, we were asked to solve the knight tour problem using both the Warnsdorf heuristic algorithm and backtracking in order to find a new path when the knight get stuck at a location with no issue. The knight tour problem consists of finding a way to make a knight visit all the squares on a chess board. Knowing that in our case we were asked to use both methods to solve the problem, we decided to use the divide and conquer method to come up with vital solution for the Knight tour problem. First, we divided the project into two sperate pieces. In the first Part, we tried to implement Warnsdoff heuristic algorithm that say always move the knight to an adjacent, unvisited square with minimal degree. Warnsdoff heuristic algorithm is described as follow:

Starting from any square, the knight must move to an unvisited square that has the fewest successive moves. Choosing a square with the fewest successors avoids a possible dead-end when traversing the board. However, because Warnsdorff's rule is heuristic, it is not guaranteed to find a solution. Fortunately, the proposed SAS solution determines a complete path beginning from any square – Well, almost. Although there are numerous solutions for a given starting point, this solution attempts to generate sixty-four paths on a standard 8x8 chess board, one for each starting position.

Rather than show the knight's tour as a directional path, it is more expedient to display the ordinal, numerical sequence on the chess board, which more easily indicates the knight's moves. In the example below, the tour begins in the upper-right corner of the board and ends nearby, just one move away.



60	11	56	07	54	03	42	01
57	08	59	62	31	64	53	04
12	61	10	55	06	41	02	43
09	58	13	32	63	30	05	52
34	17	36	23	40	27	44	29
37	14	33	20	47	22	51	26
18	35	16	39	24	49	28	45
15	38	19	48	21	46	25	50



Implementation of Warnsdoff Heuristic's Algorithm

To realize the implementation of Warnsdoff heuristic algorithm, we used two

```
knightLocation  next_location_Generator(knightLocation presentLocation)
int  number_of_next_location_Generator(knightLocation presentLocation)
```

functions denoted:

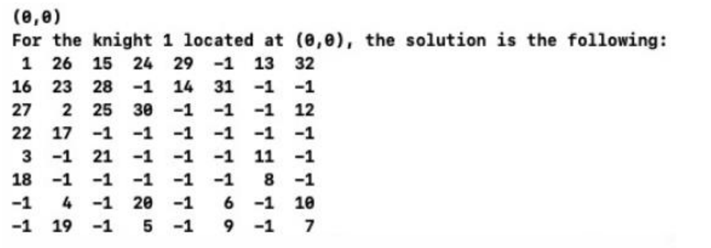
These two functions interact with each other to fulfil the requirement of the Heuristic Algorithm by Warnsdoff by returning the next location with the fewest moves. First, our current location which we chose to implement as a structure like the following

```
struct knightLocation
{
    int xa;
    int ya;
};
```

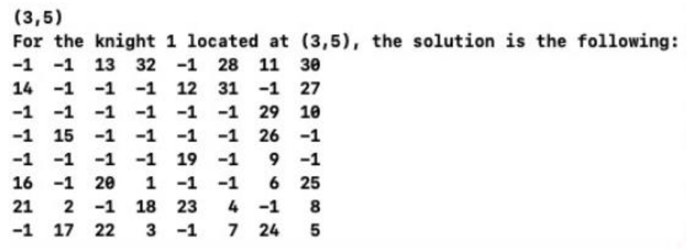
is passed into **next_location_generator** from then, an array of next location

```
knightLocation nextLocation[8]
```

will be generated. While each element of the array of element containing this various calculation $(x-2,y+1),(x-1,y+2),(x+1,y+2),(x+2,y+1),(x+2,y-1),(x+1,y-2),(x-1,y-2),(x-2,y-1)$ is being generated, the generated element is going to get passed into the function called **number_of_next_location_Generator**. The function named **number_of_next_loca**tion**_Generator** is used to generate the number of next possible locations that the location that we passed in the function has. Right after this process is done, we are storing the number of next locations that each next location generated from the present location in an array and returning the index of the element with the least next move. We will then take that index and return the element form the array of next locations that has the same index as our next location to move to (The one with the fewest next move). This process will be repeated until the stack that contains our visited locations `stack<knightLocation> visitedLocation` size's reaches 32. For our implementation of the heuristic algorithm of Warnsdoff if we choose two inputs such as (0,0) and (3,5), we will obtain the following outputs:



```
(0,0)
For the knight 1 located at (0,0), the solution is the following:
 1  26  15  24  29  -1  13  32
16  23  28  -1  14  31  -1  -1
27   2  25  30  -1  -1  -1  12
22  17  -1  -1  -1  -1  -1  -1
 3  -1  21  -1  -1  -1  11  -1
18  -1  -1  -1  -1  -1   8  -1
-1   4  -1  20  -1   6  -1  10
-1  19  -1   5  -1   9  -1   7
```



```
(3,5)
For the knight 1 located at (3,5), the solution is the following:
-1  -1  13  32  -1  28  11  30
14  -1  -1  -1  12  31  -1  27
-1  -1  -1  -1  -1  -1  29  10
-1  15  -1  -1  -1  -1  26  -1
-1  -1  -1  -1  19  -1   9  -1
16  -1  20   1  -1  -1   6  25
21   2  -1  18  23   4  -1   8
-1  17  22   3  -1   7  24   5
```

The main point that the heuristic algorithm raised is that the next location that the knight is going to go to has its coordinate close to the edge of the chess board.

In the second part of the project, we use our own implementation of how we wanted to move from one location to another. Using our own implementation, we have gotten to some position where there was no way out. Therefore, we had to use backtracking algorithm to find a new path. Backtracking algorithm is described as the following by Wikipedia:

The backtracking algorithm enumerates a set of partial candidates that, in principle, could be completed in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of candidate extension step. Conceptually, the partial candidates are represented as the nodes of a tree structure, the potential search tree. Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further. The backtracking algorithm traverses this search tree recursively, from the root down, in depth first search. At each node c, the algorithm checks whether c can be completed to a valid solution. If it cannot, the whole sub-tree rooted at c is skipped (pruned). Otherwise, the algorithm (1) checks whether c itself is a valid solution, and if so, reports it to the user; and (2) recursively enumerates all sub-trees of c. The two tests and the children of each node are defined by user-given procedures. Therefore, the actual search tree that is traversed by the algorithm is only a part of the potential tree. The total cost of the algorithm is the number of nodes of the actual tree times the cost of obtaining and processing each node. This fact should be considered when choosing the potential search tree and implementing the pruning test.

Implementation of Backtracking Algorithm

Before we get stuck at a location a start backtracking, the following is our way of finding the next location our knight. To move our knight, we are a function

```
stack<knightLocation>  next_location_Generator_for_backtracking(knightLocation presentLocation)
```

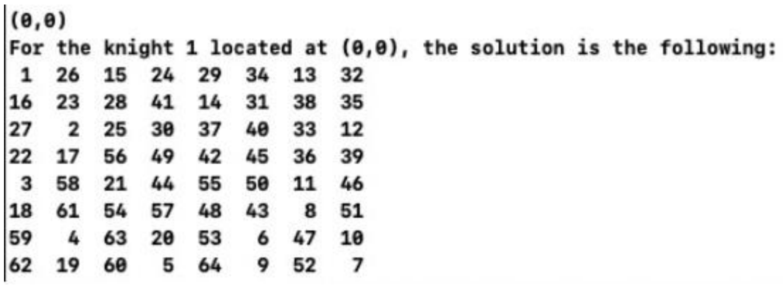
that we have named:

This function will be returning a stack of next possible location that can be visited by the knight from its present location. After the generation of that stack, it will be pushed into `stack<stack<knightLocation>>nextstack`.nextstack is a stack of stack that will be used to store the stack containing the location generated. Every time the stack of next location is generated and it not an empty stack, the program will choose the top of the stack to be the next location that the knight will visit. After that location is chosen, it will be deleted from the stack before being push into nextstack.

Whenever a current location generates an empty stack when passed into the **next_loaction_Generator_for_backtracking** function, it means the knight is stuck at that specific location thus we will start backtracking by using the stack of stack that we implemented.

After a full run of the code we get the following:

Example of a run: for (0,0)



```
(0,0)
For the knight 1 located at (0,0), the solution is the following:
 1  26  15  24  29  34  13  32
16  23  28  41  14  31  38  35
27   2  25  30  37  40  33  12
22  17  56  49  42  45  36  39
 3  58  21  44  55  50  11  46
18  61  54  57  48  43   8  51
59   4  63  20  53   6  47  10
62  19  60   5  64   9  52   7
```

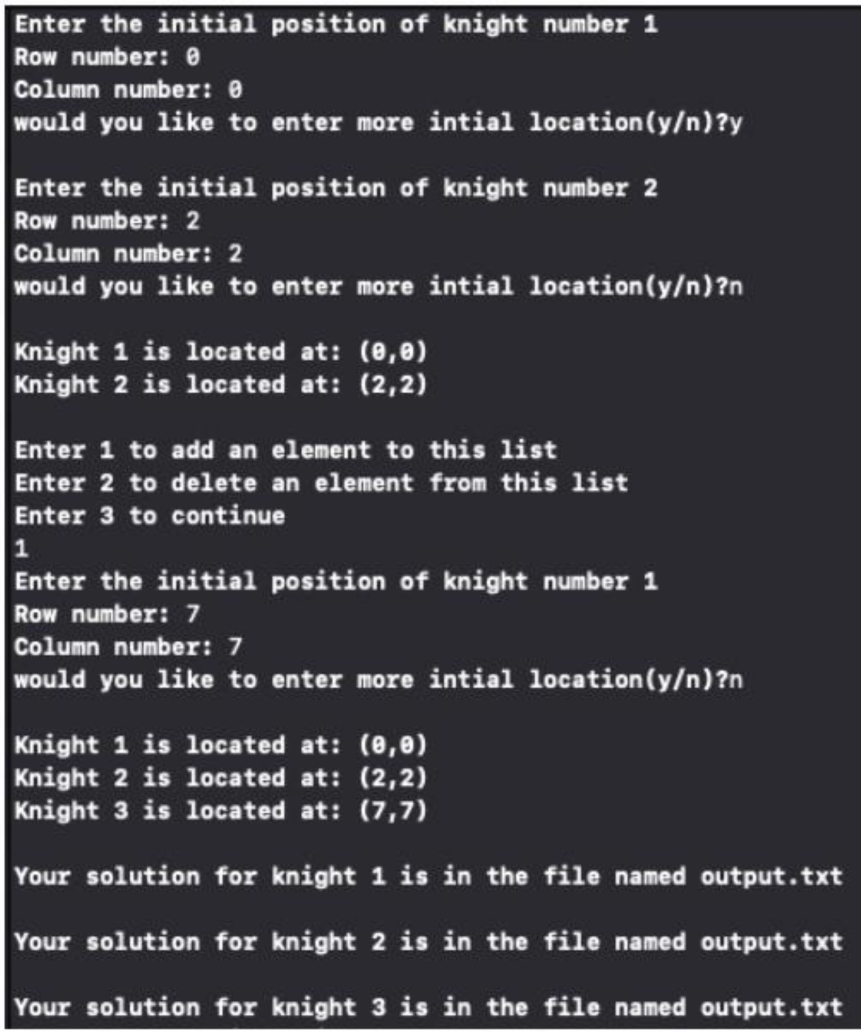
This is the list of the running time of all the squares located on the chessboard:

(0,0)→seconds	(1,0)→seconds	(2,0)→seconds	(3,0)→seconds
(0,1)→seconds	(1,1)→seconds	(2,1)→hour	(3,1)→seconds
(0,2)→hour	(1,2)→seconds	(2,2)→seconds	(3,2)→seconds
(0,3)→seconds	(1,3)→seconds	(2,3)→seconds	(3,3)→seconds
(0,4)→seconds	(1,4)→seconds	(2,4)→seconds	(3,4)→seconds
(0,5)→seconds	(1,5)→hour	(2,5)→ hour	(3,5)→minutes
(0,6)→seconds	(1,6)→seconds	(2,6)→seconds	(3,6)→seconds
(0,7)→seconds	(1,7)→seconds	(2,7)→seconds	(3,7)→seconds

(4,0)→seconds	(5,0)→seconds	(6,0)→seconds	(7,0)→seconds
(4,1)→seconds	(5,1)→seconds	(6,1)→10mins	(7,1)→seconds
(4,2)→seconds	(5,2)→seconds	(6,2)→seconds	(7,2)→seconds
(4,3)→seconds	(5,3)→seconds	(6,3)→seconds	(7,3)→seconds
(4,4)→hour	(5,4)→seconds	(6,4)→seconds	(7,4)→minutes
(4,5)→seconds	(5,5)→seconds	(6,5)→45 mins	(7,5)→minutes
(4,6)→seconds	(5,6)→hour	(6,6)→10mins	(7,6)→seconds
(4,7)→seconds	(5,7)→seconds	(6,7)→seconds	(7,7)→seconds

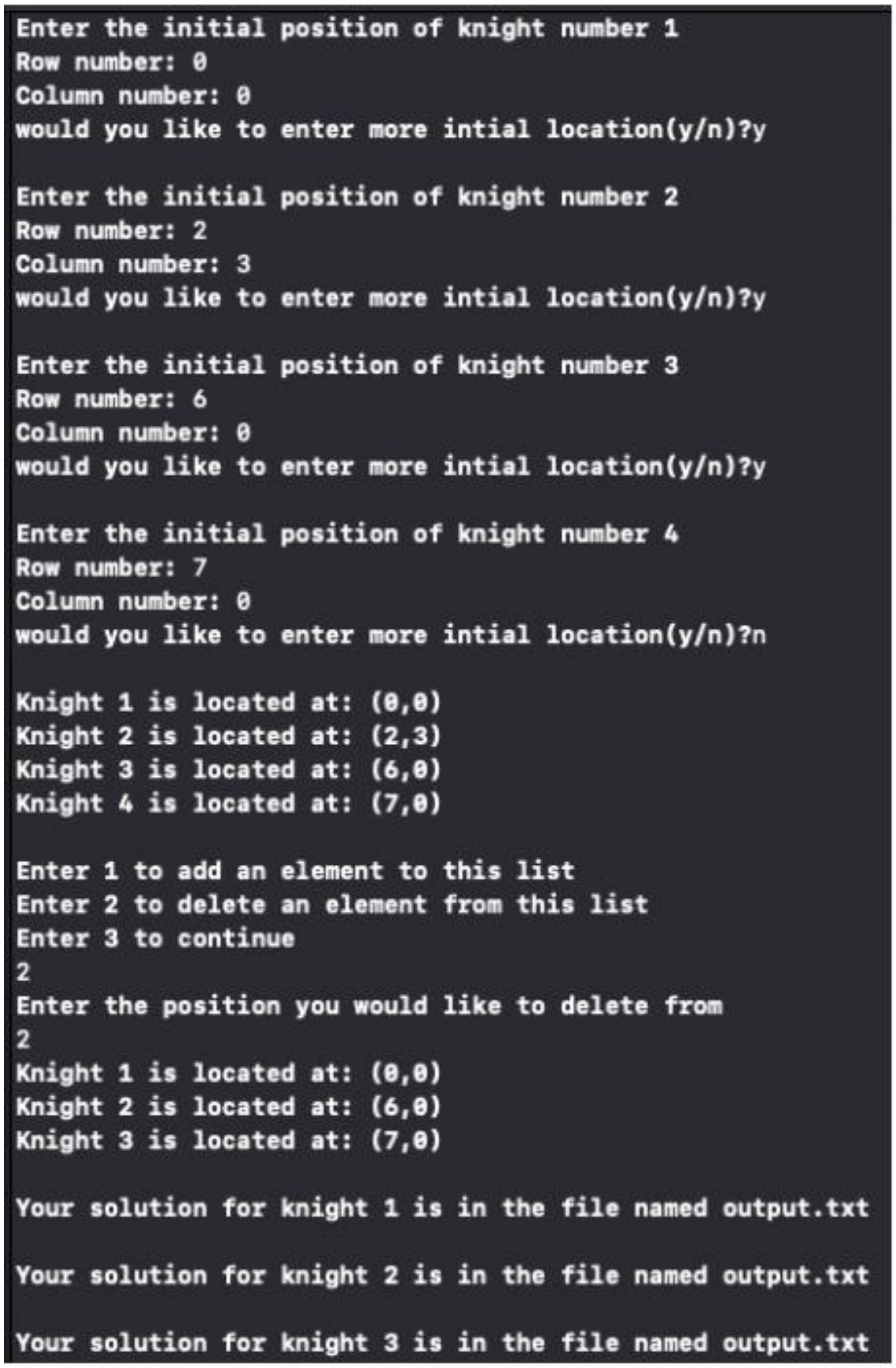
The following is a transcript of run:

Type 1: If the user decides to add to the list of location



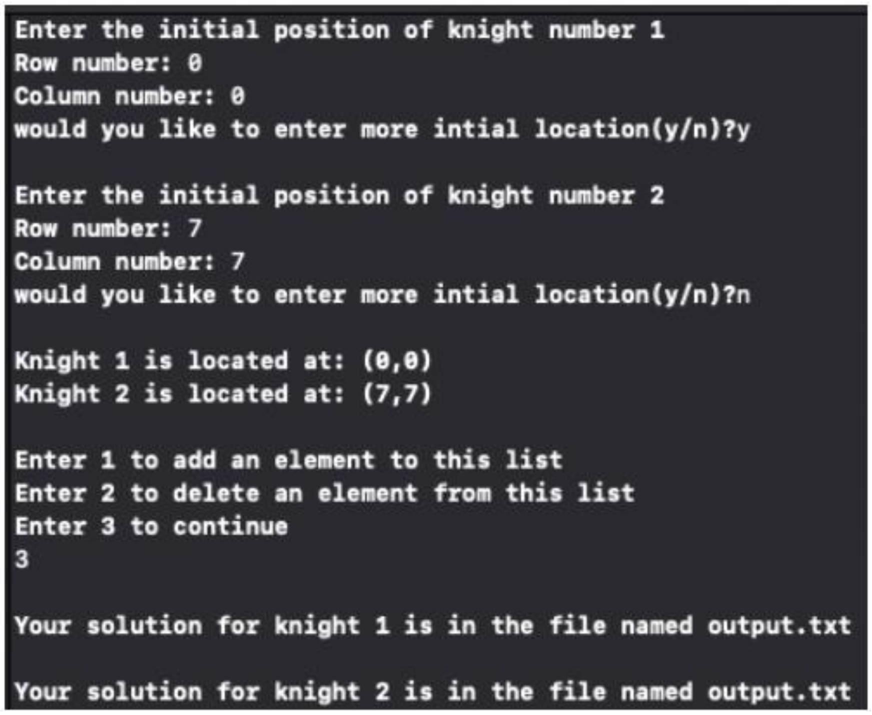
```
Enter the initial position of knight number 1
Row number: 0
Column number: 0
would you like to enter more intial location(y/n)?y

Enter the initial position of knight number 2
Row number: 2
Column number: 2
would you like to enter more intial location(y/n)?n

Knight 1 is located at: (0,0)
Knight 2 is located at: (2,2)

Enter 1 to add an element to this list
Enter 2 to delete an element from this list
Enter 3 to continue
1
Enter the initial position of knight number 1
Row number: 7
Column number: 7
would you like to enter more intial location(y/n)?n

Knight 1 is located at: (0,0)
Knight 2 is located at: (2,2)
Knight 3 is located at: (7,7)

Your solution for knight 1 is in the file named output.txt

Your solution for knight 2 is in the file named output.txt

Your solution for knight 3 is in the file named output.txt
```

Type 2: If the user decides to delete one of the entered locations

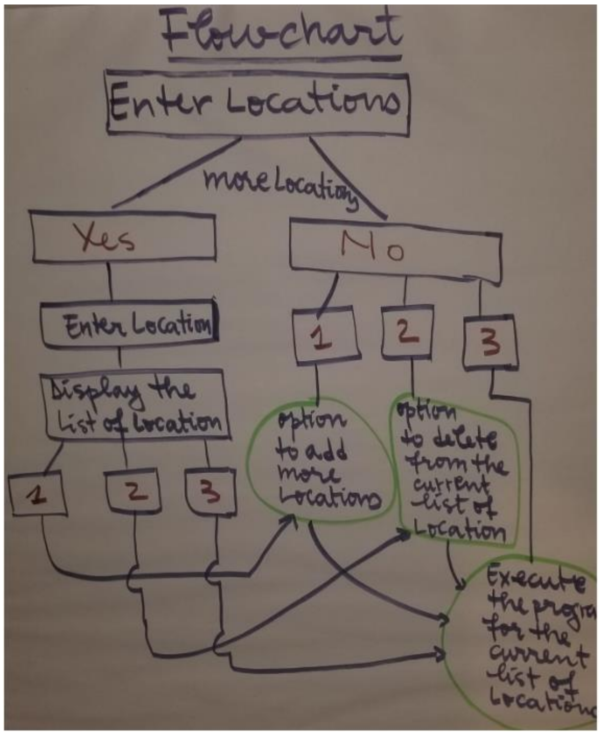


```
Enter the initial position of knight number 1
Row number: 0
Column number: 0
would you like to enter more intial location(y/n)?y

Enter the initial position of knight number 2
Row number: 2
Column number: 3
would you like to enter more intial location(y/n)?y

Enter the initial position of knight number 3
Row number: 6
Column number: 0
would you like to enter more intial location(y/n)?y

Enter the initial position of knight number 4
Row number: 7
Column number: 0
would you like to enter more intial location(y/n)?n

Knight 1 is located at: (0,0)
Knight 2 is located at: (2,3)
Knight 3 is located at: (6,0)
Knight 4 is located at: (7,0)

Enter 1 to add an element to this list
Enter 2 to delete an element from this list
Enter 3 to continue
2
Enter the position you would like to delete from
2
Knight 1 is located at: (0,0)
Knight 2 is located at: (6,0)
Knight 3 is located at: (7,0)

Your solution for knight 1 is in the file named output.txt

Your solution for knight 2 is in the file named output.txt

Your solution for knight 3 is in the file named output.txt
```

Type 3: If the user decide to not choose any of the options



```
Enter the initial position of knight number 1
Row number: 0
Column number: 0
would you like to enter more intial location(y/n)?y

Enter the initial position of knight number 2
Row number: 7
Column number: 7
would you like to enter more intial location(y/n)?n

Knight 1 is located at: (0,0)
Knight 2 is located at: (7,7)

Enter 1 to add an element to this list
Enter 2 to delete an element from this list
Enter 3 to continue
3

Your solution for knight 1 is in the file named output.txt

Your solution for knight 2 is in the file named output.txt
```


Flowchart
Enter Locations
more Locations
Yes
No
Enter Location
Display the list of Location
1
2
3
1
2
3
option to add more Locations
option to delete from the current list of Location
Execute the progra for the current list of Locations