INTERNATIONAL ISLAMIC UNIVERSITY MALAYSIA

KNIGHT’S TOUR WITH (DFS) SEARCH ALGORITHM

SEMESTER 1 2017/2018

GROUP 7

ECIE 3101

MUHAMMAD NAJIB BIN MOKHTAR 1510955

MUHAMMAD AZRI BIN MAHANIF 1512761

MUHAMAD SYAHMI BIN CHE YUSOFF 1513091

MUHAMMAD ZHAFIR BIN IBRAHIM 1511393

OBJECTIVE

• To practice on tree traversals, graph traversal dealing with real world problem.

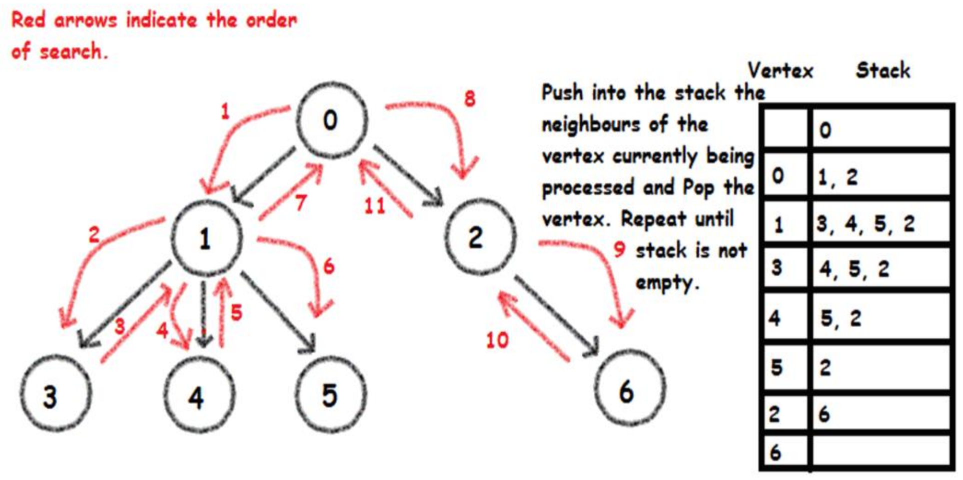
• To implement possible heuristics to reduce the complexity while traversing.

• To solve general Knight’s tour problem.

INTRODUCTION

DEPTH FIRST SEARCH(DFS)

The DFS algorithm is a recursive algorithm that use the idea of backtracking. It involves in the exhaustive search to all nodes by going ahead. There is possible by using backtracking. The word of backtracking means to moving forward until three is no more node at the current path.it will move backwards on the same path to find nodes to traverse. All node will be visited on the current path till all the unvisited nodes have been traversed after the which path will be selected


Red arrows indicate the order of search.
Push into the stack the neighbours of the vertex currently being processed and Pop the vertex. Repeat until stack is not empty.
0
1
2
3
4
5
6
1
2
3
4
5
6
7
8
9
10
11

Vertex	Stack
	0
0	1, 2
1	3, 4, 5, 2
3	4, 5, 2
4	5, 2
5	2
2	6
6	

Depth First Search

DFS can be implemented by using stacks. The basic idea is:

1)pick a starting node and push all its adjacent into stack

2)pop a node from stack to select the next node to visit and push all adjacent node into a stack

3)repeat the process until stack is empty

The time complexity is O(V+E)

Basic code for (DFS)

1)

```
graph1 = {
  'A' : ['B','S'],
  'B' : ['A'],
  'C' : ['D','E','F','S'],
  'D' : ['C'],
  'E' : ['C','H'],
  'F' : ['C','G'],
  'G' : ['F','S'],
  'H' : ['E','G'],
  'S' : ['A','C','G']
}

def dfs(graph, node, visited):
  if node not in visited:
    visited.append(node)
    for n in graph[node]:
      dfs(graph,n, visited)
  return visited

visited = dfs(graph1,'A', [])
print(visited)
```

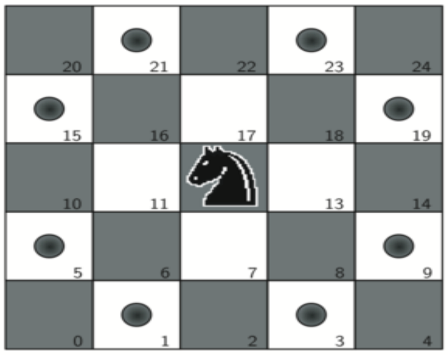
KNIGHT’S TOUR

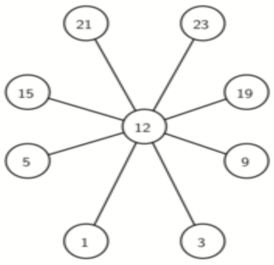
The knight’s tour is played on a chess board with a single chess piece, the knight. The object of the puzzle is to find a sequence of moves that allow the knight to visit every square on the board exactly once. The purpose of this project is to solve the problem using two main steps which are

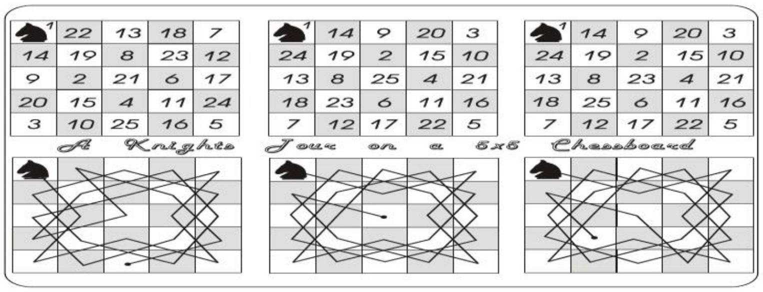
- represent legal move of a knight using (DFS) graph
- use algorithm to find path of the length rowsXcolums for every vertex on the graph is visited only once

to present knight's tour as a graph we need:

- Each square on the chessboard can be represent as a node in the graph
- Each legal move by the knight can be represented as an edge in the graph.







1	22	13	18	7
14	19	8	23	12
9	2	21	6	17
20	15	4	11	24
3	10	25	16	5

1	14	9	20	3
24	19	2	15	10
13	8	25	4	21
18	23	6	11	16
7	12	17	22	5

1	14	9	20	3
24	19	2	15	10
13	8	23	4	21
18	25	6	11	16
7	12	17	22	5

METHODOLOGY

The knight Tour function takes four parameters: n, the current depth in the search tree; path, a list of vertices visited up to this point; u, the vertex in the graph we wish to explore; and limit the number of nodes in the path. The knight Tour function is recursive. When the knight Tour function is called, it first checks the base case condition. If we have a path that contains 64 vertices, we return from knight Tour with a status of True, indicating that we have found a successful tour. If the path is not long enough we continue to explore one level deeper by choosing a new vertex to explore and calling knight Tour recursively for that vertex.

DFS also uses colors to keep track of which vertices in the graph have been visited. Unvisited vertices are colored white, and visited vertices are colored gray. If all neighbors of a vertex have been explored and we have not yet reached our goal length of 64 vertices, we have reached a dead end. When we reach a dead end we must backtrack. Backtracking happens when we return from knight Tour with a status of False. In the breadth first search we used a queue to keep track of which vertex to visit next. Since depth first search is recursive, we are implicitly using a stack to help us with our backtracking. When we return from a call to knight Tour with a status of False, in line 11, we remain inside the while loop and look at the next vertex in nbr List.

```
from pythonds.graphs import Graph, Vertex
def knightTour(n,path,u,limit):
        u.setColor('gray')
        path.append(u)
        if n < limit:
            nbrList = list(u.getConnections())
            i = 0
            done = False
            while i < len(nbrList) and not done:
                if nbrList[i].getColor() == 'white':
                    done = knightTour(n+1, path, nbrList[i], limit)
                i = i + 1
            if not done:  # prepare to backtrack
                path.pop()
                u.setColor('white')
        else:
            done = True
        return done
```

Let’s look at a simple example of knight Tour in action. You can refer to the figures below to follow the steps of the search. For this example, we will assume that the call to the get Connections method on line 6 orders the nodes in alphabetical order. We begin by calling knight Tour (0, path, A,6)

knight Tour starts with node A Figure 1. The nodes adjacent to A are B and D. Since B is before D alphabetically, DFS selects B to expand next as shown in Figure B. Exploring B happens when nighteries called recursively. B is adjacent to C and D, so knight Tour elects to explore C next. However, as you can see in Figure C Node C is a dead end with no adjacent white nodes. At this point we change the color of node C back to white. The call to knight Tour returns a value of False. The return from the recursive call effectively backtracks the search to vertex B (see Figure D). The next vertex on the list to explore is vertex D, so knight Tour makes a recursive call moving to node D (see Figure E). From vertex D on, knight Tour can continue to make recursive calls until we get to node C again (see Figure F, Figure G, and Figure H). However, this time when we get to node C the test n < limit fails so we know that we have exhausted all the nodes in the graph. At this point we can return True to indicate that we have made a successful tour of the graph. When we return the list, path has the values [A, B, D, E, F, C], which is the order we need to traverse the graph to visit each node exactly once.

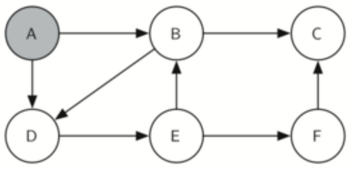


Figure A

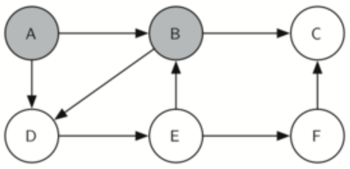


Figure B

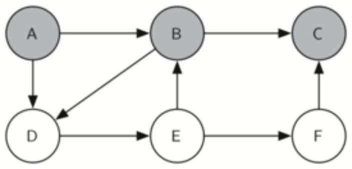


Figure C

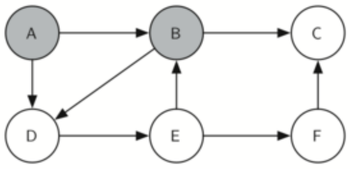


Figure D

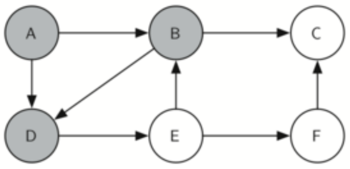


Figure E

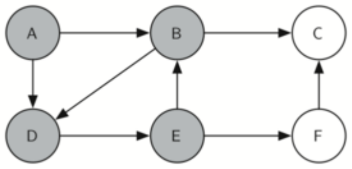


Figure F

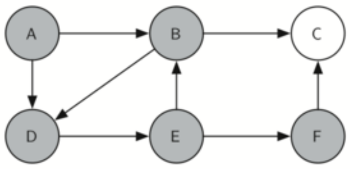


Figure G

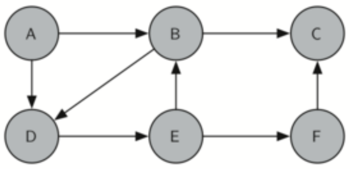
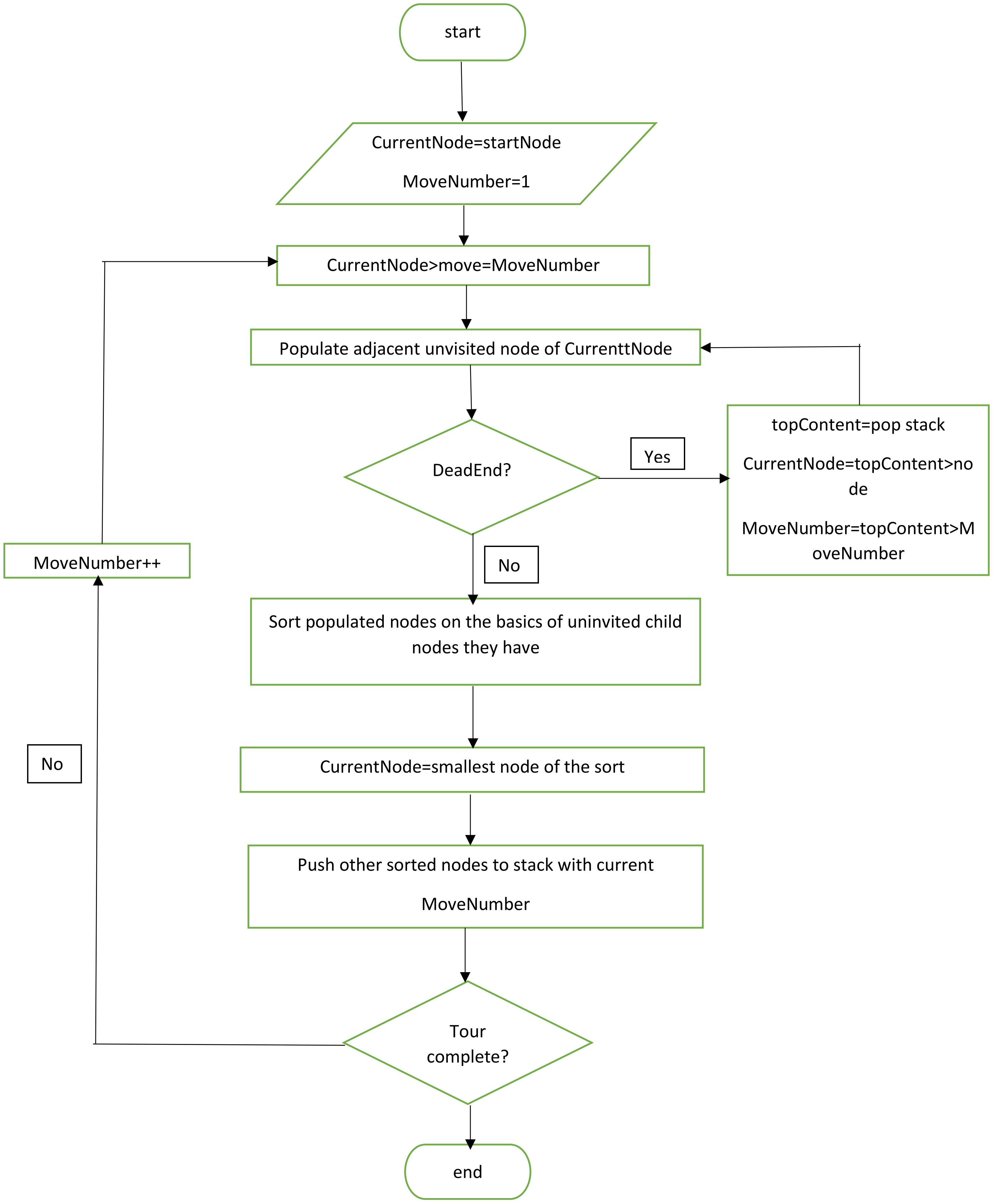


Figure H

FLOWCHART:


start
CurrentNode=startNode
MoveNumber=1
CurrentNode>move=MoveNumber
Populate adjacent unvisited node of CurrenttNode
DeadEnd?
Yes
topContent=pop stack
CurrentNode=topContent>node
MoveNumber=topContent>MoveNumber
No
Sort populated nodes on the basics of uninvited child nodes they have
CurrentNode=smallest node of the sort
Push other sorted nodes to stack with current MoveNumber
Tour complete?
No
MoveNumber++
end