BLG336-E PROJECT REPORT

Project No	1
Student	Alperen KANTARCI – 150140140
Instructor	Zehra Çataltepe
Date	18.03.2018

1. Introduction

As stated in description of project, we are asked to solve well known riddle with graph data structure and Breadth First Search & Depth First Search algorithms.

2. Development and Environment

The application is written on latest Arch Linux operating system with using Atom text editor and Clion IDE. Program compiled at Arch Linux and ITU SSH server successfully. The important note that in order to compile following command should run. **g++ -std=c++11 main.cpp**

3. Questions

3.1-) I have approached to the question with object oriented model. I created State object which keeps the where elements are and it is kind of how you see the riddle at that moment. There is an important variable which is nullNode. If node is created by illegal move nullNode will be set to true so it won't be used in graph creation. The second object of my implementation is Node which is exactly graph node. Every graph node contains currentState (State object) and it has parent,child nodes in order to create link between graph nodes. My last object is Move object that holds which animal currently in the boat with farmer. First i create the whole graph which contains every state of the riddle and creation process so much similar to the breadth first search since i should traverse whole graph with every solution and graph creation is just traverse the whole graph while creating new nodes. After creation, according to the user selection i apply breadth first search and depth first search.

3.2.a-)

```
main:
GET mode
CALL createGraph
IF mode is "bfs" or "dfs" THEN
   CALL solutionTrace with bfs or dfs
   CALL clearGraph
ELSE
   PRINT error message
ENDIF
```

```
createGraph:
CREATE initial state
CREATE a queue
CREATE a visited list with no visited nodes
ENQUEUE root node to the queue
WHILE queue is not empty
   DEQUEUE current from queue
   FOR every possible moves
      CREATE state that move and current state will create
      IF new state is not nullNode
         CREATE new Node named child with new state
         ASSIGN parameters(parent,level) of child
         IF this state didn't happen before
            ADD child to the parent's childList
            IF this state is allowed state
               ENQUEUE child
            ENDIF
         ENDIF
      ENDIF
   ENDFOR
ENDWHILE
```

```
bfs:
CREATE a queue
CREATE a visited list with no visited nodes
ENQUEUE root node to the queue
WHILE queue is not empty
   DEQUEUE current from queue
   IF current node state is the solution state
      RETURN current
   ENDIF

   FOR every child of the current node
      GET child node
      IF child node is visited
         CONTINUE
      ENDIF
      INSERT current to the visited list
      ENQUEUE child node to the queue
   ENDFOR
ENDWHILE
```

```
dfs:
CREATE a stack
CREATE a visited list with no visited nodes
PUSH root node to the stack
WHILE stack is not empty
   POP current from queue
   IF current node state is the solution state
      RETURN current
   ENDIF
   IF current node is not visited
       ADD current node to the visited list
       FOR every child of the current node
          GET child node
          PUSH child node to the stack
       ENDFOR
   ENDIF
ENDWHILE
```

```
cleanGraph:
IF current is nullptr
   return
ENDIF

FOR each child of the current
    CALL cleanGraph with child node
ENDFOR

DELETE current node's state
DELETE current node
```

3.2.b-) Complexity: BFS algorithm is implemented with adjacency list so complexity for bfs algorithm will be O(V+E). V stands for number of nodes, E stands for number of edges between two nodes.

DFS algorithm is implemented with adjacency list so complexity will be same as bfs algorithm and it is O(V+E). V stands for number of nodes, E stands for number of edges between two nodes.

But creating the graph has complexity as well and graph creation happen before running any search algorithms. I used very same algorithm as BFS only extra operation is trying 4 moves every time and searching any cycle. So creation of the graph structure complexity will be O(V*K+E) where V is number of nodes , E is number of edges between two nodes and K for cycle preventing.

At last part of my pseudo code you can see it only traverse whole graph and remove their state so it's complexity is O(2*N) so it's O(N). As you can see the worst runtime in the pseudo code is O(V*K+E) or O(V+E) it depends on cycle preventing and number of nodes.

3.3-) Since we go until the end of to the graph we push node to the stack at every layer and if we don't keep discovered we can visit the same node again. So in order to not create infinite loops we have to keep visited nodes with additional list or any data structure.

3.4-) I would say yes to this question since we can color the edges of the graph with two color and there will be different layers for this connected graph. Also we can ensure it from there is no cycle with odd numbered edge so if our graph is connected and doesn't have a kind of cycle that i mentioned then by lemma in the slides it is bipartite graph and every odd layer level can be colored as black and every even layer level can be colored as red in graph.

3.5-)

	BREADTH FIRST SEARCH	**DEPTH FIRST SEARCH**
# visited nodes	26	13
maximum # nodes in memory	8	8
Running time	0.029 ms	0.022 ms
# steps of solution	7	7

We can see that depth first search found the solution with visiting less nodes since in our riddle graph we didn't iterated after the not allowed state so the only deepest path in the graph is the solution path, therefore depth first search found that quickly. However if it wasn't the case and the if all the branches would contain deeper nodes then depth first search would consume more time to find correct node. Also if i will keep the graph in another form (Changing the order of the move) dfs and bfs visited nodes count would differ.