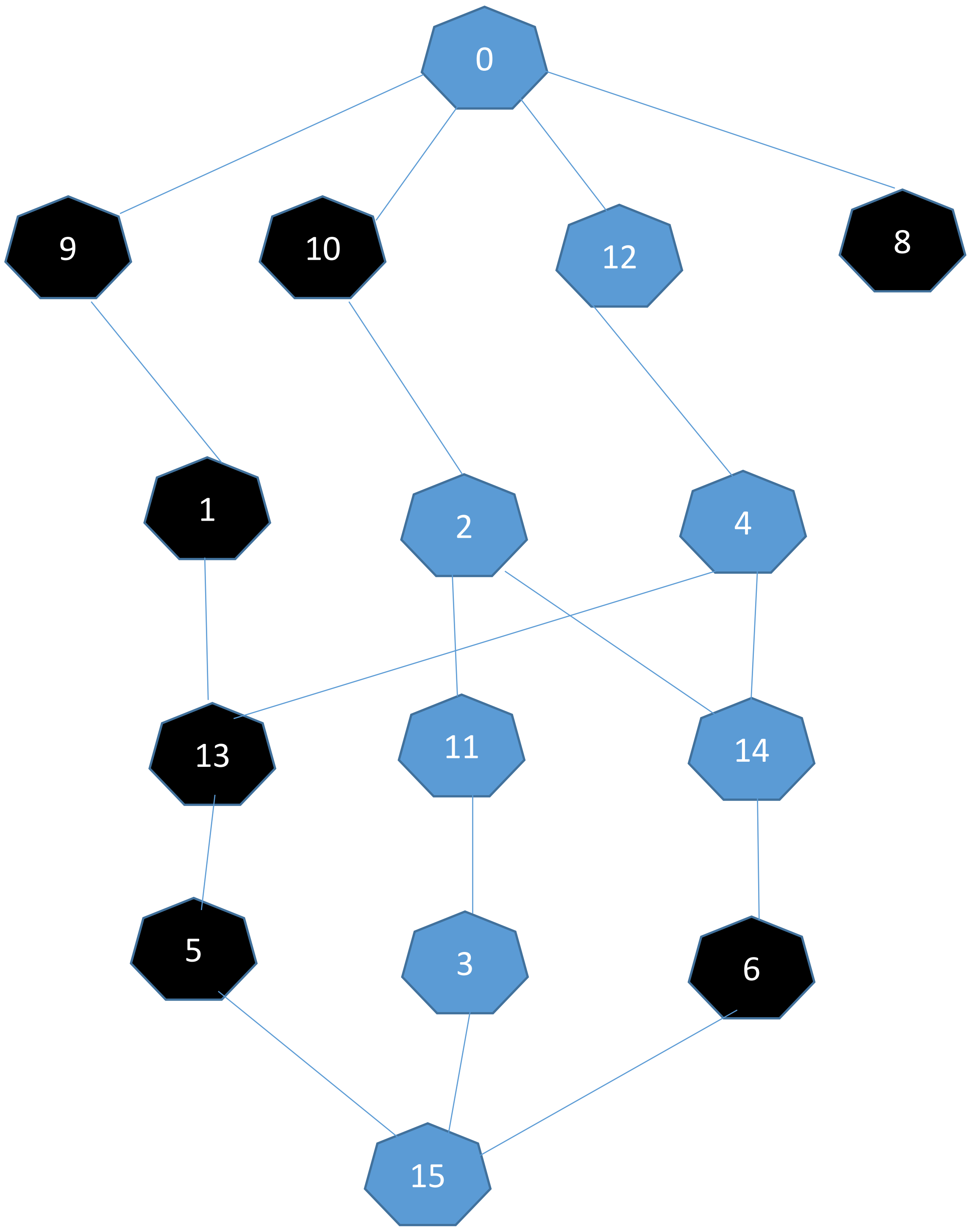
Emre KÖSE - 150130037

REPORT

1) Present your problem formulation, state and action representations in detail.



The graph representation is like above. The number that is inside of these nodes represents the id of the node which is generated with following logic:

Let’s examine state with id 0. This state represents the initial state which is all farmer, rabbit, fox and carrots are in west side of the river. I represented this state in object like (w,w,w,w) and then the ‘id’ of the state is calculated with logic of binary numbers. In this representation, the ‘w’ is accepted as a value of zero and ‘e’ is accepted as a value of one. So, (w,w,w,w) is calculated as 0000 and this is equal to 0. For an another example the last state is (e,e,e,e) which is represented as 1111(binary) and this is equal to 15 in decimal. When action is realized, new state is generated in code. This is done with changing only appropriate parameters of the state. So, what is appropriate means in this situation? The appropriate means the parameters can be changed, if this is same with farmer’s situation. As an extra state, farmer can cross the river alone.

The colors show us which states are dangerous and which are not. The blue ones are safe states and black ones are dangerous. So, in searching we should go over the blue ones.

```
class State{

    int id;
    char Farmer;
    char Rabbit;
    char Fox;
    char Carrots;

public:
    int edgesNumber;
    State *edges;
```

The class which is given above represents our nodes, these are our states.

2) How does your algorithms work?

I followed up different approaches among BFS and DFS. In BFS, because we need track to all nodes in each layer, firstly the graph was constructed and then the solution was found. But in DFS there is no need to check all nodes in one layer firstly, so the solution was found directly in this situation.

a) Write your pseudo-code.

As intended, we have two works, these are bfs and dfs.
Pseudo-codes:
In BFS:

```
➔ costructGraph(root)
        -push root to queue
        -while(queue is not empty)
            -temp = get element from front of the queue
            -assign neighbors of the temp node
            -for each neighbor of temp node
                -push neighbor to queue
                -assign true for flag of neighbor(visited flag)

            -control reached node count for graph(if all is visited, exit from while)
```

```
➔ findSolution(root)
        -get neighbors of root node
        -for each node in neighbors
            -check the id of the node for validity
                -If(node is valid)
                    -add this node to solution path
                    - findSolution(validNode)
                    -break
```

In DFS:

```
➔ dfs()
        -push root to stack
        -while(stack is not empty)
            - temp = pop state from stack
            - get neighbor states of temp
            - if( flag[id of temp node] == false)           // if node is unvisited
                - flag[id of temp node] = true
                - pop from stack
                - find the most appropriate state among neighbors
                    - push other neighbors
                    - push the most appropriate state
            -else
                -pop from stack                             // already visited
```

b) Show complexity of your algorithm on pseudo-code.

For both the searching complexity is m + n which m is total edge number and n is total node number. Actually, complexity is lower than 33. m=18+ n=15

In bfs, the calculation results with 33 steps which is actual value of complexity and in dfs calculation results with 21 steps.

3) In Depth-First Search algorithm, why should we maintain a list of discovered nodes?

In DFS, the flag for discovered node leads the algorithm to check about this node's neighbors are visited or not. This control flow is provided with stack and if we do not marked discovered node as discovered, it will always push all neighbors of the current node to stack and this situation will lead the algorithm's running time increase too much or in some situations, it will not reach the unvisited nodes and will not end up the algorithm. Actually maintaining is the key step of this algorithm.

4) Is the graph constructed by the given problem formulation a bipartite graph? Why or why not?

In the graph which was shown in question-1, shows us the graph is bipartite. Because it has no cycle with odd number and there is no edge between nodes which are in same layer.

5) Analyze and explain the algorithm results in terms of:

- The number of visited nodes: I think this is about nodes which was discovered to find solution. So, in my program all nodes are visited in bfs. But, in the dfs, only the nodes which are in solution path are visited.

- The maximum number of nodes kept in the memory: I assume this question as the max. number of nodes those memories are taken with 'new' function. In my program, in bfs I firstly constructed graph with fully connected, this value is bigger than in dfs. In addition, I only did a memory allocation with 'new' for root node.

- The running time: I could not see real results in millisecond range. However, I think the running time in BFS is bigger than the DFS because of the visited node number and node number which is kept in memory.

- The number of move steps on the found solution: Because of the solution is unique in my representation the step numbers are equal each other.