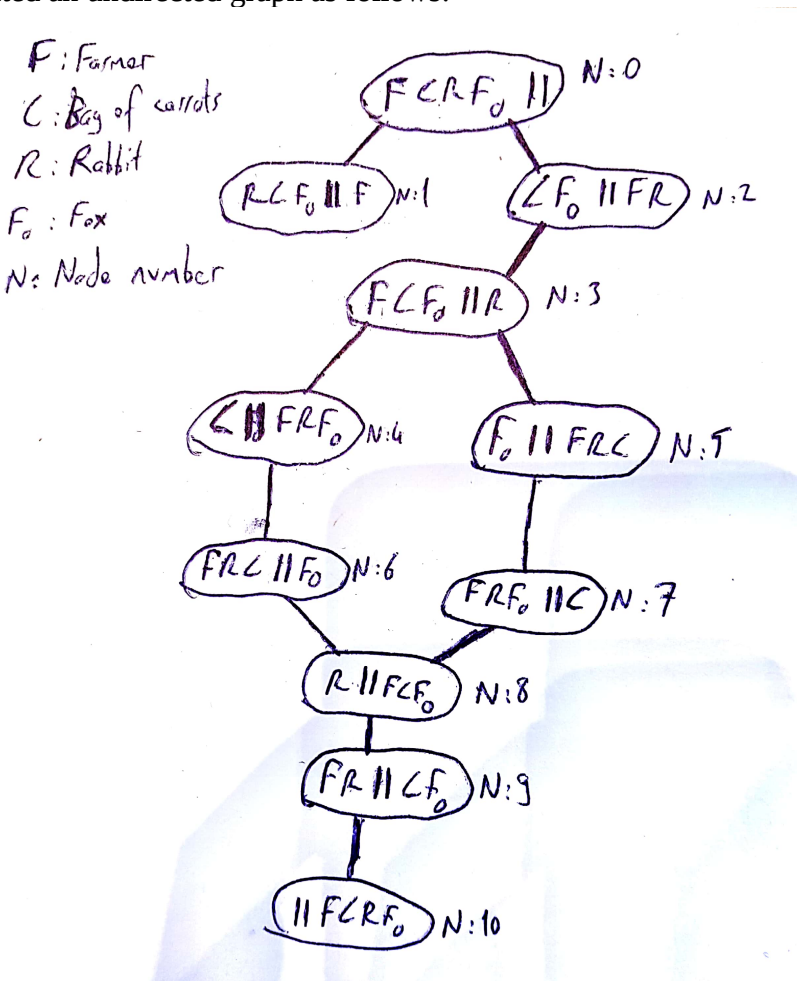
In this project, I tried to solve a famous problem. There is a farmer, his rabbit, bag of carrots and fox on the east side of the river, and all of them want to pass the river. However, there are many things to handle, e.g. only farmer can row, fox can eat rabbit if they stay alone.

For solving this problem, I have written all the possible situations. Then, I have numbered them and have created an undirected graph as follows:



This graph is an example of bipartite graph because the nodes can be colored red or blue such that every edge has one red and one blue end. None of this graph's node has more than 2 edges. So, it's possible to color them which satisfies the condition of being bipartite graph.

I thought I needed two classes to turn this graph into C++ code and I have created class Node and class Graph.

```
class Node{
public:
    std::string east[4],west[4];
    int no;
    static Node create(int, int, int, variables, variables, variables,
variables);
};
```

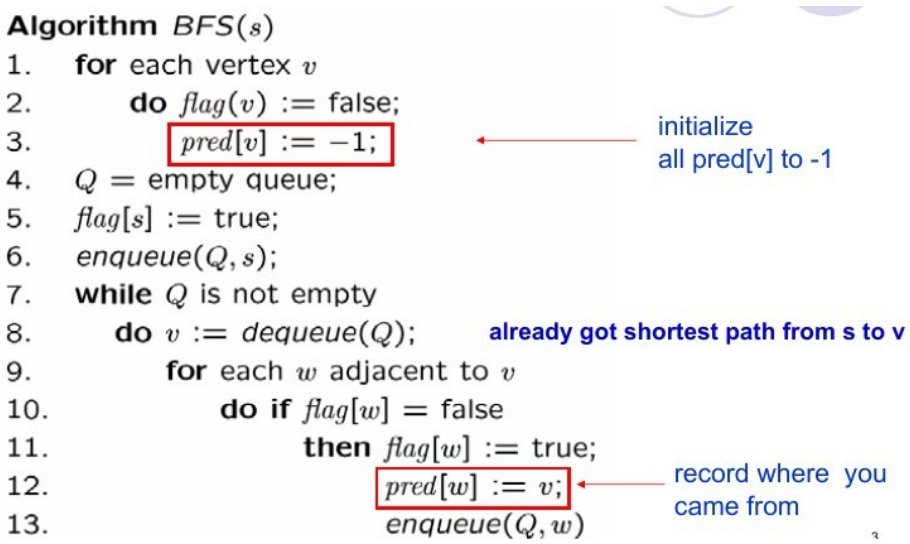
I have used this class for creating the nodes of the graph.

```
class Graph{
  std::list<class Node> *list;
   int visitedNodes;
   int movCount;
public:
   Graph(); //constructor
   void insertion(int, class Node);
   int* BFS(int s);
   int* DFS(int s);
   void print_path(std::list<int>, class Node*);
   //helper methods
   void print_stack(std::stack<int>);
   void print_queue(std::queue<int>);
};
```

This class has a list pointer of nodes, in other words, adjacency lists for each node. For instance, if node-0 is connected with node-1 and node-2, it means that list-0 has two nodes, node-1 and node-2.

Insertion method(x, y) has been used for connecting the nodes. It adds the class Node y parameter to the list-x.

BFS method is the implementation of Breadth-First Search algorithm. Pseudo-code of it is as follows:




```
   for(int i=0;i<11;i++) {
      visited[i] = false;
      parent[i] = -1;
   }
   queue.push(s);
   visited[s] = true;
   while(queue.empty() == false){
      u = queue.front();
      queue.pop();
      for(it = list[u].begin(); it != list[u].end(); ++it){
```

```
            if(visited[(*it).no] == false) {
                visited[(*it).no] = true;
                parent[(*it).no] = u;
                queue.push((*it).no);
                movCount++;
            }
            visitedNodes++;
        }
    }
```

DFS method is the implementation of Depth-First Search algorithm. Pseudo-code of it is as follows:

```
Algorithm DFS(s)
     For each vertex v
        do flag(v) := false
           pred(v) := -1;
     Stack S := empty stack;
     s := push S;
     flag(s) = true;
     while (S is not empty)
        do u := pop S;
           if (not visited[u]) then
               visited[u] := true;

           for each w adjacent to v
              do if flag[w] := false;
                    w := push S;
                    flag[w] := true;
                    parent[w] := u;
```

```
    for(int i=0;i<11;i++) {
        visited[i] = false;
        parent[i] = -1;
    }

    stack.push(s);
    visited[s] = true;
    while (stack.empty() == false){
        u = stack.top();
        stack.pop();
        if(visited[u] == false)
            visited[u] = true;
        for(it = list[u].begin(); it != list[u].end(); ++it){
            if(visited[(*it).no] == false) {
                stack.push((*it).no);
                movCount++;
                parent[(*it).no] = u;
            }
            visitedNodes++;
        }
    }
```

Bayram Barış Sarı

Time complexity of both methods are O(number of nodes+ number of edges). Let's explain it:

First for loop of both functions take O(N) time. (N = number of nodes)

While loops have queue and stack respectively for BFS and DFS. Since when one node is visited, it's marked visited, It will take O(N) time, too. However, we have to consider the inner for loop inside of that while loop. It traverses adjacency lists, and it visits all edges twice. For instance, if Node 0 and Node 1 are connected. When 0 is pushed to queue, adjacency list of node 0 will be traversed and the edge between node 0 and node 1 will be traversed. On the other hand, when node 1 is pushed to queue, adjacency list of node 1 will be traversed and the edge between node 0 and node 1 will be traversed again. The rest of the code takes O(1) time. As a result, **time complexity of these methods** are:

$$O(N) + O(N+2E) = O(2N+2E) = O(N+E)$$

In DFS, we need to maintain a list of discovered nodes because otherwise, while loop will be infinite. Stack will never be empty and nodes always will be pushed to the stack.

Method : BFS
Number of visited nodes: 22
Solution move count: 10
The number of nodes kept in memory: 11
Execution time: 3.8e-05 s

Farmer Rabbit Carrot Fox ||
(Farmer, Rabbit, >)
Carrot Fox || Rabbit Farmer
(Farmer, <)
Farmer Carrot Fox || Rabbit
(Farmer, Fox, >)
Carrot || Farmer Rabbit Fox
(Farmer, Rabbit, <)
Farmer Rabbit Carrot || Fox
(Farmer, Carrot, >)
Rabbit || Fox Carrot Farmer
(Farmer, <)
Farmer Rabbit || Fox Carrot
(Farmer, Rabbit, >)
|| Fox Carrot Rabbit Farmer

Method : DFS
Number of visited nodes: 24
Solution move count: 11
The number of nodes kept in memory: 11
Execution time: 4.2e-05 s

Farmer Rabbit Carrot Fox ||
(Farmer, Rabbit, >)
Carrot Fox || Rabbit Farmer
(Farmer, <)
Farmer Carrot Fox || Rabbit
(Farmer, Carrot, >)
Fox || Carrot Rabbit Farmer
(Farmer, Rabbit, <)
Farmer Rabbit Fox || Carrot
(Farmer, Fox, >)
Rabbit || Fox Carrot Farmer
(Farmer, <)
Farmer Rabbit || Fox Carrot
(Farmer, Rabbit, >)
|| Fox Carrot Rabbit Farmer

When we compare the result of BFS and DFS for this project's problem, BFS give better results. It visits less nodes and therefore, it traverses the graph faster than DFS.