BLG336E - ANALYSIS OF ALGORITHMS II

ASSIGNMENT-1 REPORT

Onat Şahin - 150150129

1.

For the states, I created a state class. This class has an array attributed which keeps the locations (left or right) of every individual (farmer, rabbit, carrot, fox). I also keep an 16x16 adjacency matrix. If states m and n are connected (if one can be achieved from the other with a single move), (m,n) and (n,m) cells of the matrix are "1". In this created graph, bfs and dfs algorithms try to get to the final state. If the algorithms enter a failure state (if an animal eats another animal or the carrot), they do not continue to search that route. I do not keep actions as objects and print them using the differences in the adjacent states in the taken route.

2. This is a simplified pseudocode for my algorithms

Class State:

locations[4]

Main program:

vector<State> states

initializeStates(states)

stateNum ← size[states]

vector<bool> visited(stateNum, false)

vector<int> previous(stateNum, -1)

vector< vector<int> > adjacency(stateNum, vector<int>(stateNum))

connectStates(adjacency, states)

if argument = dfs:

then dfs(adjacency, states, visited, previous)

End if

Else: bfs(adjacency, states, visited, previous)

Print results

initializeStates(vector<State>):

toPermute ← “llllrrrr"

for every permutation of toPermute:

  if toPermute.substr(0,4) not in vector:

    then vector.push_back(State(toPermute[0],toPermute[1],toPermute[2],toPermute[3])

  End if

End for

connectStates(adjacencyMatrix, vector<State> states):

For i = 0 to size[states]:

  For j = i to size[states]:

    If isConnected(states[i], states[j]):

      then adjacencyMatrix[i][j] ← 1

        adjacencyMatrix[j][i] ← 1

    End if

    Else: adjacencyMatrix[i][j] ← 0

      adjacencyMatrix[j][i] ← 0

    End else

  End for

End for

int bfs(adjacencyMatrix, states, visited, previous):

queue<int> q

visitedCount ← 0

q.push(0)

While q is not empty:

  current ← front[q]

  q.pop()

  If visited[current] is false:

    visited[current] ← true

```
        visitedCount++
        If current = size[states]-1:
            Then return visitedCount
        End if
        If states[current] is an invalid state:
            Continue
        End if
        For i = 0 to size[states]:
            If adjacencyMatrix[current][i] = 1 and visited[i] is false:
                visited[i] ← true
                prev[i] ← current
                q.push(i)
            End if
        End for
    End if
End while
```

int dfs(adjacencyMatrix, states, visited, previous):

```
stack<int> s
visitedCount ← 0
s.push(0)
While s is not empty:
    current ← top[s]
    s.pop()
    If visited[current] is false:
        visited[current] ← true
        visitedCount++
        If current = size[states]-1:
            Then return visitedCount
        End if
```

```
    If states[current] is an invalid state:
        Continue
    End if
    For i = 0 to size[states]:
        If adjacencyMatrix[current][i] = 1 and visited[i] is false:
            visited[i] ← true
            prev[i] ← current
            s.push(i)
        End if
    End for
  End if
End while
```

Complexities:

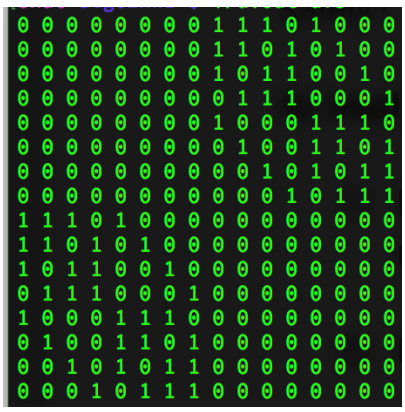
initializeStates : O(n), n being the number of permutations of "llllrrrr"

connectStates: $O(n^2)$ (Actually, the complexity is $O(n^2/2)$)

Bfs and dfs: $O(N+E)$ (N is node number and E is edge number. In these algorithms, we traverse the nodes and for every node, we traverse connected edges.)

3. If we do not keep track of the visited nodes, the depth first search algorithm may enter a cycle and visit the same nodes over and over again.

4.



This is my adjacency matrix after the creation of the graph. It can be seen that there is no connection between the first 8 states. Likewise, there is no connection between the last 8

states. There are only connections from one group to the other. Therefore this graph is a bipartite graph.

5.

While DFS algorithm visited 11 nodes, BFS algorithm visited every node. This is because while DFS continues from a single path until the path ends, BFS traverses the graph layer by layer. DFS algorithm managed to get the solution in the first path. The maximum number of nodes kept in the memory is the same for both algorithms. This is because both algorithms found the same solution. Which is why number of move steps are also the same in both algorithms. Running times are very similar in both algorithms, the reason for which is the graph being very small.