BLG336E - Analysis of Algorithms II
2017-2018 Spring, Project 1 Report
Submission Deadline: 18.03.2018, 23:59

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

I used a special well-known method which is called **bit masking** to represent the states with only numbers. In this method the bits of the numbers are used to store the state, and it provides mainly two advantages: reducing the used memory, and making easier to control states.

In this problem, there are just 16 different states which are specified below, and we can express them with numbers from 0 to 15. Let's consider all these numbers in **binary form**, they have 4 bits and each bit is responsible for one character (Farmer, Fox, Rabbit, Carrot respectively). If a bit is 0, it means the corresponding character is at the east side, otherwise it is at the west side.

Number	Binary Representation	State
0	0000	Farmer Fox Rabbit Carrot ||
1	0001	Farmer Fox Rabbit || Carrot
2	0010	Farmer Fox Carrot || Rabbit
3	0011	Farmer Fox || Rabbit Carrot
4	0100	Farmer Rabbit Carrot || Fox
5	0101	Farmer Rabbit || Fox Carrot
6	0110	Farmer Carrot || Fox Rabbit
7	0111	Farmer || Fox Rabbit Carrot
8	1000	Fox Rabbit Carrot || Farmer
9	1001	Fox Rabbit || Farmer Carrot
10	1010	Fox Carrot || Farmer Rabbit
11	1011	Fox || Farmer Rabbit Carrot
12	1100	Rabbit Carrot || Farmer Fox
13	1101	Rabbit || Farmer Fox Carrot
14	1110	Carrot || Farmer Fox Rabbit
15	1111	|| Farmer Fox Rabbit Carrot

Table 1. Number-State Transitions
(Invalid states are red-colored)

According to the number-state transition table, we can easily construct the state diagram: **red** cells represent invalid states, and **green** cells represent valid states. The source cell is colored with **yellow** and sink cell is colored with **blue**. Actually all state actions are bi-directional, however turning back to the same state would not advantageous, hence I just use directional edges.

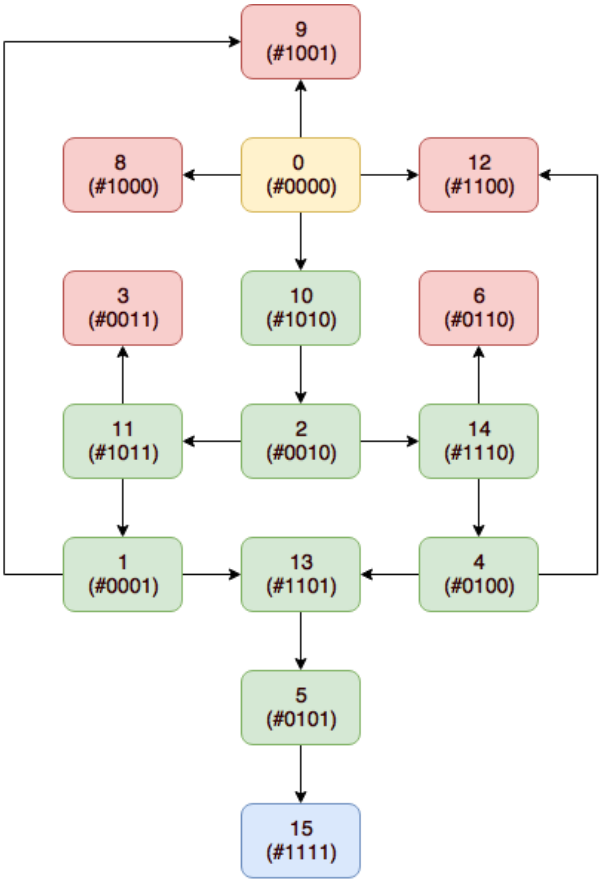

Figure 1. State Diagram

Now all states and actions are determined, and we can just use these transitions instead constructing a graph explicitly.

2) How does your algorithm work?
   a. Write your pseudo-code.
   b. Show your complexity of your algorithm on pseudo-code.

function BFS():	O(N)
Q = new Queue() // holds current_state, previous_state pairs	O(1)
add initial pair (0, 0) to Q	O(1)
while Q is not empty:	O(N)
increase number_of_visited_nodes by 1	O(1)
maximum_node_count = max(maximum_node_count, Q.size())	O(1)
current_state, previous_state = Q.pop()	O(1)
mark the current state visited	O(1)
trace[current_state] = previous_state // we can reach the current state from previous state	O(1)
if (current_state is destination (15)):	O(1)
return true	O(1)
for each valid movement from current_state:	O(3)
if next_state is not visited:	O(1)
add (next_state, current_state) pair to Q	O(1)
return false	

function DFS (int state, int previous, int depth)	O(N)
increase number_of_visited_counts by 1	O(1)
max_node_cnt = max(max_node_cnt, depth);	O(1)
trace[state] = previous; // it means to reach the state, first you have to reach "previous" state	O(1)
if state is destionation (15) :	O(1)
return true	O(1)
mark the current state visited	O(1)
for each valid movement from current_state:	O(3)
if next_state is not visited and DFS (next_state, current_state, depth+1) is true:	O(N)
return true	O(1)
return false	O(1)

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

Because if the algorithms encounter a node which was discovered already, it causes an infinite loop and it can never move out the loop.

4) Is the graph constructed by the given problem formulation a bipartite graph? Why or why not? (Implementation is not required for this part.)

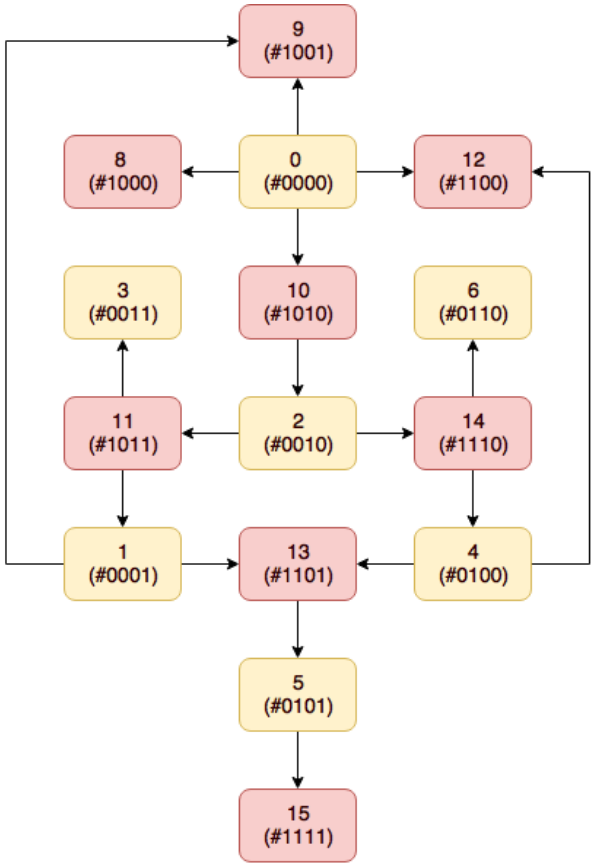

Figure 2. Bipartite Graph

As we can see the state diagram in Figure 1 the graph is a bipartite graph, we can decompose the graph into two disjoint set such that no two graph nodes within the same set are adjacent. Colors represent the set I composed.

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

- **The number of visited nodes:**

According to the program outputs, Depth-First Search algorithm visits **8** nodes (states) and Breadth-First Search algorithm visits **12** nodes (states).

- **The maximum number of nodes kept in memory:**

In DFS, the maximum number of nodes kept in memory is the maximum depth of the recursion, and the maximum depth is **8** for this problem.
In BFS, the maximum number of nodes kept in memory is the maximum size of queue structure that is reached at any moment, and it is just **2** for given problem.

- **The running time:** 0.001 s

DFS and BFS algorithms have O(E + V) time complexity and because E and V are both very small (16), the program finds the solution rapidly.

- **The number of move steps on the found solution:** 7

There is only two solutions in this problem as we can see the state diagram in Figure 1, and both of them has 7 move steps without repetition. DFS and BFS algorithms that I implemented for this problem can find these solutions too.

Compilation Command: g++ kod.cpp -o kod -std=c++11
Running Command: ./kod bfs

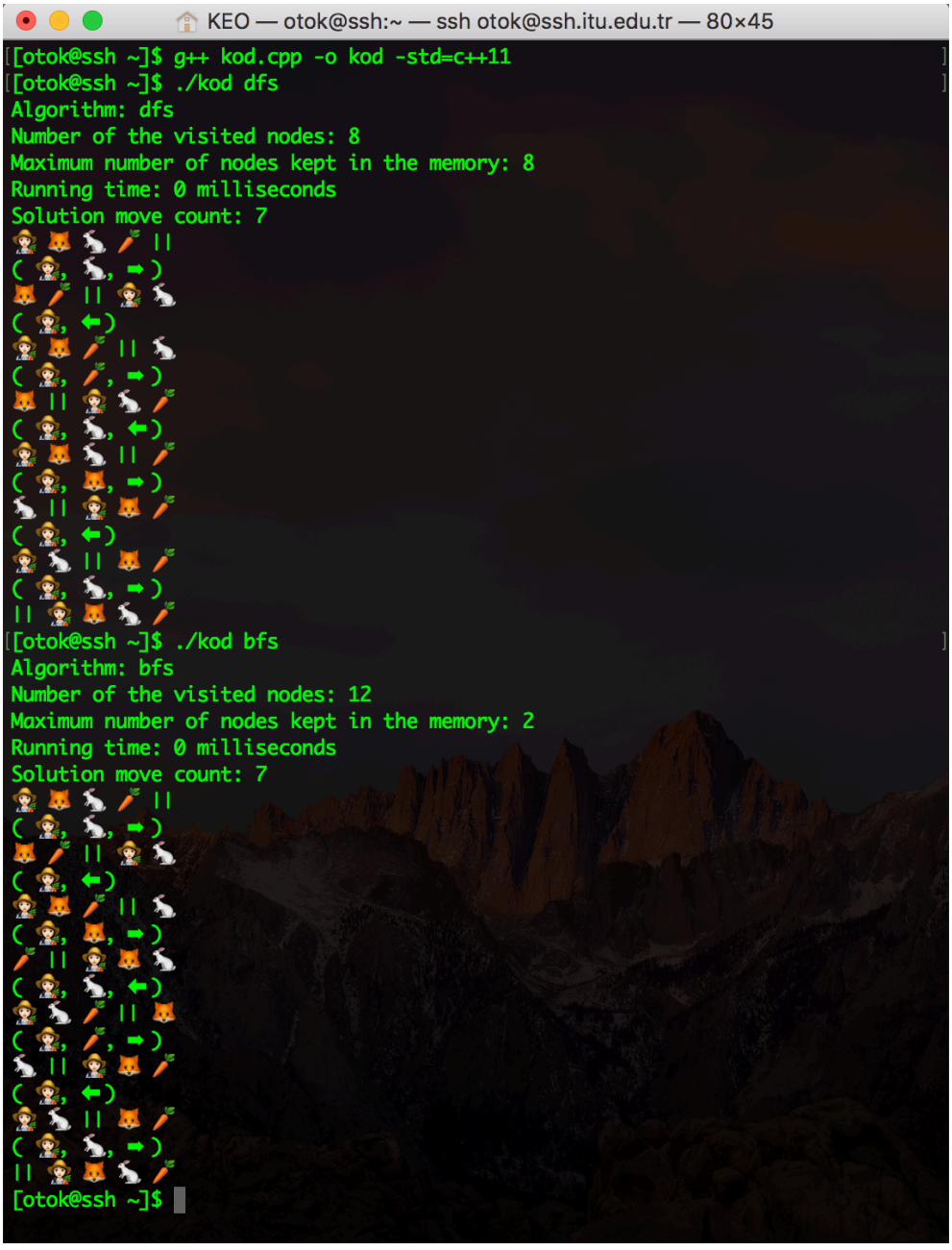

Figure 2. The funny output of program with emojis