ADA Assignment 2

Ankur Sharma, 2016225

February 2018

Partners: Anvit Mangal (2016135) and Ishaan Bassi (2016238).

Problem 1

Prove the correctness of BFS algorithm implemented using adjacency matrix and derive its time complexity.Does the time complexity reduce when adjacency list is used. If yes, Prove it.

Pseudocode of BFS using adjacency matrix:
Let vertices are numbered from 1 to n. And v_i denotes i^{th} vertex. Let A is an adjacency matrix.

```
for i: 1 → n :
   if v_i is not visited:
      BFS(v_i);
BFS(s):
   Q # An empty queue
   Q.enqueue(s);
   while Q is not empty:
      u = Q.dequeue();
      mark u as visited;
      for v : 1 → n:
         if A[u][v]==1 and v is not visited:
            Q.enqueue(v);
```

Correctness of BFS algorithm implemented using adjacency matrix:
Aim of BFS is to visit all vertices in a graph level by level.
We have to show that:
1) All vertices are visited level by level.
2) Algorithm terminates.
Proof of 1):
We will prove it using induction on level of graph as viewed from source.
Base Case: 0th level. That is source itself. From pseudocode, we enqueue

source and dequeue it in while loop and mark it visited. Hence, Base case holds.
Inductive Hypothesis: All vertices are visited upto level k that are reachable from given source.
Now, we will prove that if vertices at level k+1 can be reached from source then they will be visited.
Vertices at level k+1 can only be reached from level k as viewed from source.
Now, for all vertices at k^{th} level we enqueue their neighbors and they are dequeued in code and hence, marked visited.
and, they are visited from k^{th} level.
Hence, Proved.
Proof of 2): BFS is called for all the vertices that are unvisited. Because a graph may have multiple components. In each iteration of 'while' loop, we dequeue a vertex from queue adn mark it visited. As the 'for' loop in which we are adding only unvisited vertices in queue and also it is finite as we are iterating from 1 to n. Hence, algorightm terminates.

Time Complexity:
Time Complexity is $O(V^2)$. Because, all n vertices in graph check for its neighbors in O(n) (for loop inside while). Hence, $O(V^2)$.
Yes, time complexity reduces to O(V+E) in case of adjacency list. Take a look at the pseudocode of BFS written above. 'for loop' inside while loop runs v from 1 to n to find neighbors of u. In this case, we end up going to all vertices in graph even if they are not neighbors of u. But, in case of adjacency list, we store only those vertices in list of u which are neighbors of u. Hence, we only go to neighbors of u. Therefore, Complexity becomes O(V+E) which in case where each vertex has n-1 neighbors also becomes $O(V^2)$.

Problem 2

Design an algorithm to find minimum number of dices rolled required to win snake and ladder game. Prove the correctness of the algorithm and derive the time complexity for the same. Justify why your algorithm is the best?

Algorithm:
Assign each step (1-n) a vertex. Now, we will design(construct) our graph in such a way that we can run bfs to get the desired result. It is a directed graph with edges between the vertices defined as follows:
1) There is a directed edge from v_i to v_{i+1} for all $1 <= i < n$.
where v_i=i b/c vertex are numbered from 1 to n.
Such edges has weight equal to 1.

2) If there is a ladder or snake from a vertex u to v, then we add
a directed edge from u to v with 0 weight. And remove the edge (u, u+1)
which we added in 1). Also, we add a new edge from u-i to u+j with weight
equal to 2 where i and j are minimum positive integer such that u-i and u+j
has no ladder or snake on them.
This was our graph construction. Now, we will implement bfs
algorithm to get optimal path for this graph.
Here source(s) is v_1 or 1 and destination(t) is v_n or n.
and V = $\{v_i : 1 <= i < n\}$.
Also, the graph is constructed using Adjacency linked list as any vertex
can have maximum of 2 edges. (2 in case when a vertex)
BFS algo:

```
for all v εV-s
   vis[v] = false;
vis[s] = true;
P (Parent array)
Q # An empty queue
Q.enqueue(s)
while Q is not empty:
   u = Q.dequeue();
   for all v ε neighbors(u):
      if vis[v] == false :
         vis[v] = true;
         P[v]=u;
         Q.enqueue(v);
\*We have obtained shortest path path from s to t. Now, we will calculate
minimum no. of dices rolls required using parent array.*\
numOfRolls=0;
count =0 ;
for(i = t; true; i=P[t]):
   if(weight(t, P[t]) == 0 or i==s):
      numOfRolls += ceil(count/6);
      count=0;
   else count++;
   if(i==s):
      break;
print(numOfRolls);
```

Correctness:
1) Our aim is to get minimum number of dices rolled which is equivalent to
get an optimal(shortest) path from 1 to n.
2) Algorithm terminates.
Proof:
First, we are running BFS to get optimal path and then, we are computing
minimum number of dices rolled required to traverse this path using parent

array. So, we are starting BFS with 1 as source. As proved in Problem 1, BFS ensures that all the vertices are visited. But here our aim is to get shortest path from 1 to n. Since, we know BFS traverse graph level by level. Therefore, whichever vertex discovers 'n' first, will be the parent of 'n'. Hence, we get the shortest path to 'n' as we are using parent array to store parent of all the vertices. Difficult task is done. Now, we need to compute minimum dice rolls. While travelling from 'n' to '1' using parent array, we will get 1's and 0's as weight of edges. 0 indicates that edge is a ladder or snake and 1 indicates that edge is normal edge from u to u+1. Trick is to calculate consecutive 1's in order to calculate no. of dice rolls required to reach that ladder or snake. Because, in a single roll, suppose we get X number, and if there is a ladder or snake in a way we can't go up or down. because ladder or snake is not at final vertex to which we are going to right now. So, in second part we calculate our min dice rolls by adding consecutive 1's and again, doing so for all 1's seperated by 0's.
Part I: is BFS as proved already in Problem 1, it gets terminated successfully.
Part II: is a 'for loop' for traversing parent and calculating min dice rolls.
We start with 'n' and go up to 's'. Hence, this loops also gets terminated.
That is because, in BFS, we are reaching 'n' from 's'.

Time Complexity:
Part 1 of the algorithm is BFS algo using adjacency list:
therfore, it has complexity of O(V + E)) with E>= n-1.
But each vertex has maximum of 2 edges that would make it O(V) only.
Part 2 is a for loop. If there is no ladder then we have a worst case.
Therefore, Part 2 has complexity of O(V).
Time Complexity: O(V)

My algo is best because the solution to problem requires to find shortest path between two nodes which is done using BFS algorithm b/c there is no other way of doing it better than this. As at least we have to visit all vertices in worst case (no ladder or snake), it's linear. Normally, such problem is done using dijkstra which has higher time complexity than BFS.

Problem 3

Alice gave Bob a 2D box with the box divided into tiny 2D boxes called as pebbles and each having a color. Alice gave a problem to Bob to replace color of a given pebble and all adjacent same colored pebbles with given color. Bob is stuck and your goal is to design an algorithm for him to solve his problem.Write pseudocode for the problem and explain the correctness of your algorithm along with that prove the time complexity for the same.

First of all, we construct an undirected graph in following way:
1) Given an m x n matrix say A. We consider each element a vertex.

2) Each vertex is connected to maximum of 8 other vertices.
For example: A[i][j] may be connected to A[i+1][j], A[i-1][j], A[i][j+1], A[i][j-1], A[i+1][j-1], A[i+1][j+1], A[i-1][j+1], A[i-1][j-1]. Given all these indexes of A are non-negative.
So, this is out graph construction.

Pseudocode:

```
x^th row = input;
y^th column = input;
newColor = input;
currentColor= A[x][y];
visited[][]; # A 2D boolean array of m x n size
# with all its element assigned false initiallly.
Q; # An empty queue which stores two int at each index.
Q.enqueue(x, y); # saves x, y as a list at the first index of Q.
while Q not empty:
   u = Q.pop(); # returns a list of x and y indexes.
   A[u.x][u.y] = newColor; # update color
   visited[u.x][u.y] = true;
   for v ε neighbors(u):
      if visited[v.x][v.y] == false and A[v.x][v.y] == currentColor :
         q.enqueue(v.x, v.y);
```

Note: neighbors(u) returns a list of all vertices index u is connected to in O(1). As described in 2).

Correctness:

Aim here is to show that:
1) All the adjacent vertices with same color are visited and updated with new color.
2) Algorithm terminates.
Proof:
The implemented algorithm is actually BFS with an extra condition that only those nodes are added in queue which has same color as source and are not visited yet. In Problem 1, we proved that BFS visits all nodes in a connected graph(if undirected in one call). Therefore, we call BFS with given input
source and start adding vertices in the queue. As already stated only those vertices are added which has same color as source and are not visited yet. Therefore, as BFS proceeds, we do the same for all the vertices added in queue and mark them visited and update their color. Hence, All the adjacent vertices with same color as source are visited and updated with new color.
Our algorithm terminates when queue gets empty. Since, in each iteration we remove a vertex from queue and mark it as visited. Also, we only adding adjacent vertices with same color and not visited. Hence, in worst case, it adds m*n vertices and gets terminated. Also, it is just a BFS with an extra

condition our algorithm terminates successfully as explained in Problem 1.

Time Complexity:
In worst case, we can have a matrix in which we have all m*n elements of same color. Since, inner for loop runs for maximum of 8 times b/c a vertex can have maximum of 8 neighbors (from 2). And, our outer loop would run m*n times in worst case. Therefore, time complexity is: O(m*n) which was excepted as our algo is similar to BFS where number of vertices = m*n.