CS 577: Introduction to Algorithms **Homework 3 Solutions**

Problem 1

Algorithm Design:

Note that the set of nodes we can reach in a single time step depends on both our current position and our current velocity. Our graph should therefore encode information not just on our current position, but also on our current velocity.

To achieve this, we construct a graph with vertices representing the possible states of a car on the race track. Each state will consist of the car's current position and current velocity (before trying to move this step). The current velocity and position each have an x and a y component, so each state will be represented as a 4-tuple of the form (x, y, x_{vel}, y_{vel}). A vertex $u = (x, y, x_{vel}, y_{vel})$ should have edges only to vertices that can be reached by adjusting our current velocity and then letting one time step pass. Since we can only change each velocity component by at most 1 at a time, the neighbors of $u = (x, y, x_{vel}, y_{vel})$ are the vertices of the form

$$v = (x + x'_{vel}, y + y'_{vel}, x'_{vel}, y'_{vel}),$$

where x'_{vel}, y'_{vel} denotes the velocity after making the change, and $|x_{vel} - x'_{vel}| \leq 1$ and $|y_{vel} - y'_{vel}| \leq 1$. Once our graph is constructed, we can determine the minimum number of steps needed to complete the race track by finding the minimum length in our graph of a path from a starting line vertex to a finish line vertex. This can be accomplished efficiently using a breadth first search.

Now we estimate the number of vertices in the graph. The bounds for x and y are simple since we just use every pair (x, y) that is a position on the race track. For the velocity, there is a trivial bound $-n \leq x_{vel}, y_{vel} \leq n$. In fact, we can set stronger bounds on x_{vel} and y_{vel}. Since we can only increase our horizontal velocity by at most 1 at each step, to accelerate to x_{vel} while covering minimal distance, we will have had each horizontal velocity value $1, 2, \ldots, x_{vel} - 1, x_{vel}$ exactly once each. Thus, the minimal distance we can have travelled horizontally while accelerating to x_{vel} is:

$$\Delta_x = \sum_{k=1}^{x_{vel}} k = \frac{x_{vel}(x_{vel} + 1)}{2}.$$

We know that the absolute value of Δ_x is bounded by n, thus we have $|x_{vel}|, |y_{vel}| \leq O(\sqrt{n})$ and the total number of vertices in our graph is $O(n^3)$.

Now that we have bounds on x, y, x_{vel}, y_{vel} and a means of testing whether a given movement is valid, we can construct our graph:

1. For every position (x, y) on the race track, create nodes (x, y, x_{vel}, y_{vel}) for all $|x_{vel}|, |y_{vel}| \leq O(\sqrt{n})$.
2. For each node $u = (x, y, x_{vel}, y_{vel})$ and $x'_{vel} \in \{x_{vel} - 1, x_{vel}, x_{vel} + 1\}$ and $y'_{vel} \in \{y_{vel} - 1, y_{vel}, y_{vel} + 1\}$:
   (a) If $v = (x + x'_{vel}, y + y'_{vel}, x'_{vel}, y'_{vel})$ is in the node set, AND
   (b) the edge (u, v) is not already in the edge set,
   (c) then add the edge (u, v) to the edge set.

Once our graph is constructed, we can find a shortest path from the starting line to the finish line with a breadth first search in which we keep track of our current distance from the starting line:

BFS:

1. Create a queue Q.
2. Add each starting line vertex to Q with distance 0, and mark each such vertex as visited.

3. While Q is nonempty:
    (a) Pop a vertex u from Q.
    (b) If u is a finish line vertex, return its distance.
    (c) Else for each unvisited neighbor v of u:
        i. Add v to Q, with its distance set to be 1 greater than that of u.
        ii. Mark v as visited.

Proof of Correctness:

It is clear that our graph models the given problem faithfully. The correctness of this algorithm therefore follows from the fact that breadth first search can be used to find the shortest path lengths between vertices in an unweighted graph.

Runtime Analysis:

Each vertex in our graph is an endpoint of at most 9 edges. This is because, starting from any state, there are at most 9 changes we can make to our velocity. Since $|V| = O(n^3)$, we see that $|E| = O(n^3)$ too. A breadth first search in a graph $G = (V, E)$ takes time $O(|V| + |E|)$, so the total runtime of our algorithm is $O(n^3)$.

Problem 2

Algorithm Design:

Note that in the graph each edge has weight 1, thus we can use a modified breadth first search. We will fill in a one-dimensional array A indexed by the vertices of G, and once the algorithm has terminated, $A[w]$ will be the number of shortest v, w paths in G. In standard BFS, we maintain markings for all nodes, so that each node is only added to the queue at most once. Instead of maintaining "visited" markings, we will mark each node with its distance from the source node v, as well as the number of shortest paths to that node.

Algorithm 1: Pseudocode of SPC(G, v, w):

```
   Input: Graph $G$ and vertices $v, w$.
1  Set $A[v] = 1$, and initialize $A[x] = 0$ for every other node $x$;
2  Set $d(v, x) = \infty$ for each node $x$;
3  Create a queue $Q$ containing only vertex $v$ initially, and set $d(v, v) = 0$;
4  while $Q$ is nonempty do
5      Pop the first node $u$ from $Q$;
6      foreach neighbor $x$ of $u$ with $d(v, x) > d(v, u)$ do
7          Update $A[x] = A[x] + A[u]$;
8          if $d(v, x) = \infty$ then
9              set $d(v, x) = d(v, u) + 1$ and add $x$ to $Q$;
10 return $A[w]$.
```

Runtime Analysis:

The initialization of A and the distance values take $O(n)$ time each. The rest of the algorithm is a breadth first search with at most $O(1)$ additional work in each step. Standard breadth first search is $O(n+m)$, so our algorithm has overall runtime $O(n + m)$.

Proof of Correctness:

If v, w belong to different connected component of G, our algorithm will correctly output 0. Without loss of generality, we assume G is connected. To show the correctness of our algorithm, we prove the following claim.

Claim 1. *For every vertex $x \in V$, when x is popped from Q, $A[x]$ is equal to the number of shortest v, x paths in G and it will be never changed in future.*

Proof. We prove by induction on $d(v, x)$. Our base case is $d(v, x) = 0$, which only occurs when $x = v$. The number of shortest v, v paths is 1. In our algorithm, $A[v]$ is set to 1 initially, and is never changed in the "while" loop.

Now assume inductively that, at termination, the value of $A[x]$ will be correct for any node x with $d(v, x) = k$. Now consider a node x with $d(v, x) = k+1$. We show that $A[x]$ must still be correct at termination. Let $x_1, \dots, x_r$ be the neighbors of x with $d(v, x_1) = \dots = d(v, x_r) = k$. Thus the correct final value of $A[x]$ will be $A[x_1]+\dots+A[x_r]$ (where each $A[x_i]$ is the correct final path counts for x_i). This is due to the fact that any subpath of a shortest path is also a shortest path between its two end vertices.

In our algorithm, the value $A[x]$ will be increased by $A[x_i]$ exactly once for each of the x_i since BFS considers vertices in increasing order of distance from the source v. Also each x_i is run through Q before x is popped from Q. The only changes to a given $A[x_i]$ occur during the $k - 1$ layer of BFS, in which all lower distance neighbors of x_i are run through Q. All nodes in subsequent layers have distance from v of at least $d(v, x_i)$, so no further changes can be made to $A[x_i]$. Thus, each $A[x_i]$ has taken on its final value when it is added to $A[x]$. We know by our inductive hypothesis that the final value of each $A[x_i]$ is correct, so we conclude that the correct $A[x_i]$ value is added to $A[x]$ for each i.

Finally, no other changes are made to $A[x]$. There cannot be a neighbor x' of x with $d(v, x') < k$, since we would then have $d(v, x) < k+1$. The only other neighbors x' of x satisfy $d(v, x') \geq k+1$. This means that $d(v, x) \leq d(v, x')$, hence no changes are made to $A[x]$ from $A[x']$. Therefore, we conclude that $A[x] = A[x_1]+\dots+A[x_r]$ at termination and as previously noted, this value is correct. □