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

Problem 1

We want to prove that: an edge e is necessary $\Leftrightarrow$ for every cycle C in G that contains it, e is not the maximum weight edge in this cycle. We will show each direction separately:

- *e is necessary $\Rightarrow$ e is not the maximum weight edge in* any *cycle.*
  To show this, we will show that the contrapositive holds, that is we will show that if there exists a cycle C_k such that e is the maximum weight edge in that cycle $\Rightarrow$ e is not a necessary edge.

  We can actually show something stronger: if e is the maximum weight edge in some cycle, then it can not be part of the MST (hence not necessary). Let us take an MST T that contains e. If we removed e, we would create a *cut* with two distinct sets of nodes S and $V \setminus S$. We would then use another edge e' to cross the cut, and specifically we would choose one from C_k since we know $w_{e'} < w_e$ for all edges e' in C_k. This would give us a new spanning tree T' with $W(T') < W(T)$, which is a contradiction since we supposed T was an MST.

- *e is not the maximum weight edge in* any *cycle $\Rightarrow$ e is necessary.*
  We know that the MST is unique (due to the unique edge weights) so we only need to show that e will always belong to this MST. If e belongs to this MST, e will also be necessary.Let $e = (u, v)$ where $u, v \in V$, and C_i be the cycles that $e \in C_i$. Lets assume e does not belong in the MST. Then there should be a path connecting u an v, which will use all other edges of one of the cycles C_i. But since e is not the maximum weight edge in any cycle, this means we will be forced to use the maximum edge in C_i. This contradicts the fact that we have an MST, since the maximum weight edge in a cycle cannot belong to an MST.

Problem 2

You're trying to run through an obstacle course, but there's a twist. Each step on the course is marked either "left" or "right". You're only allowed to step on squares labeled "left" with your left foot, and likewise for your right foot. Of course, you can decide which foot to take your first step with. Is it possible for you to run from your starting location to a target location while obeying the course constraints?

You are given an undirected graph $G = (V, E)$ in the form of an adjacency list, a start node s, a target node t, and a partition of E into two sets L and R. Call a path "feasible" if it alternates between edges in L and edges in R.

(a)

Give a linear time (i.e. $O(n + m)$) algorithm to compute the shortest feasible path from s to t or determine that no such path exists. Show your algorithm is correct and achieves the desired runtime.

Algorithm Design:

We can solve the problem by running BFS on a modified graph. The modified graph is (mostly) bipartite, where each partite set is a copy of the node set in the original graph. We can ensure that only feasible paths are considered by making the edges directed, with left edges leaving one partite set and right edges leaving the other.

The purpose of the supersource s' is to allow paths starting from s_r or s_ℓ so that the first edge taken can be from either L or R. Similarly, the supersink t' allows the path to finish at either t_r or t_ℓ.

Note that the edges of E' are directed but the edges of L and R are undirected, so for example an undirected edge $\{u, v\} \in R$ becomes the two directed edges (u_r, v_ℓ) and (v_r, u_ℓ) in E'. The interpretation of a node v_r is that the next edge taken must be a right edge, and likewise with v_ℓ the next edge must be a left edge.

Algorithm 1: Left-Right Alternating BFS

```
Input: Graph $G = (V, L \cup R)$ and vertices $s, t$.
1 Define $G'$ to be a directed bipartite graph with node set $V' = \{v_\ell, v_r : v \in V\}$ and edge set
    $E' = \{(u_\ell, v_r) : \{u, v\} \in L\} \cup \{(u_r, v_\ell) : \{u, v\} \in R)\}$;
2 Add to $G'$ a "supersource" node $s'$ with edges $(s', s_r)$ and $(s', s_\ell)$ and a "supersink" node $t'$ with edges
    $(t_r, t')$ and $(t_\ell, t')$ (making it no longer bipartite);
3 Run BFS on $G'$ with source $s'$ and sink $t'$;
4 If BFS does not find an $(s' - t')$ path, return "No Feasible Path";
5 If BFS finds an $(s' - t')$ path, return the same path with the supersource $s'$ and supersink $t'$ removed and all
    subscripts removed from the internal nodes ;
```

Proof of Correctness:

There is a correspondence between paths in G' and paths in G, as implied by the final step of our algorithm. Suppose $P = s, v^1, v^2, \ldots v^k, t$ is a feasible path in G. Let $f(P) = \begin{cases} s, s_r, v_\ell^2, v_r^3, \ldots, t_q, t & \text{if } (v^1, v^2) \in R \\ s, s_\ell, v_r^2, v_\ell^3, \ldots, t_q, t & \text{if } (v^1, v^2) \in L \end{cases}$, where q is either ℓ or r depending on the parity of the length of the path.

We claim that $f(P)$ is a path in G' if and only if P is a feasible path in G. Then, since the length of $f(P)$ is just 2 more than the length of P, finding the shortest path in G' is equivalent to finding the shortest feasible path in G. This reduces the correctness of our algorithm to the correctness of BFS, and we know that BFS correctly finds the shortest path in an unweighted graph.

($\Rightarrow$) Suppose $f(P) =$ is a path in G'. We want to show P is a feasible path in G. Since $G' \setminus \{s, t\}$ is bipartite, $f(P)$ must alternate between ℓ nodes and r nodes (again exlcluding the supersource and supersink). But all out-edges from ℓ nodes correspond to edges in L in G, and similarly out-edges from r nodes correspond to edges in R in G. Therefore, P alternates between edges in L and edges in R, and so is feasible.

($\Leftarrow$) Suppose P is a feasible path in G. Suppose without loss of generality that P begins with an edge in R, and $P = s, v^1, v^2, \ldots, v^k, t$. Then $f(P) = s, s_r, v_\ell^1, v_r^2, \ldots, t_q, t$. This will be a path in G' since any edge $\{v^{2i}, v^{2i+1}\}$ in P must be an edge in R and so $(v_r^{2i}, v_\ell^{2i+1})$ will be an edge in G'. Similarly, any edge $\{v^{2i-1}, v^{2i}\}$ in P must be an edge in L and so $(v_\ell^{2i-1}, v_r^{2i})$ will be an edge in G'.

Runtime Analysis:

We can construct the graph G' in $O(n + m)$ since it requires a constant amount of work for each node and edge in G. Note also that $|V'| = O(|V| = n)$ and $|E'| = O(|E| = m)$. Hence, since BFS runs in linear time, running BFS on G' is also $O(n + m)$. Computing the shortest path from the predecessors requires only $O(m)$, and all other steps in the algorithm run in constant time. Hence, the overall algorithm is $O(n + m)$.

(b)

You've decided that you'd be willing to hop, stepping with the same foot twice in a row, if it shortens your path. However, you aren't very coordinated so you can only hop once during the run. Call a path "1-feasible" if it alternates between edges in L and R, with the possible exception of a single pair of consecutive edges in either L or R. Note that feasible paths are necessarily 1-feasible. Give a linear time (i.e. $O(n + m)$) algorithm to compute the shortest 1-feasible $s - t$ path in G or determine that none exists. Show your algorithm is correct and achieves the desired runtime.

Algorithm Design:

Notice that the algorithm from part (a) can be modified to take a source node and compute the shortest paths from the source to every other node, as well as the lengths of the paths. This modification is simply using the version of the standard BFS that finds all shortest paths rather than searching for a particular target.

We'll use this modified version of (a) as a subroutine to compute the shortest feasible path from s to every other vertex. We'll also use it to compute the shortest feasible path from t to every other vertex.

A 1-feasible path has at most 1 vertex w at which the "hop" occured, meaning the edges to and from w belong to the same set. The length of the shortest $s-t$ path that permits a hop at w (it may not actually have a hop at w, which is also fine) is equal to the length of the shortest feasible $s-w$ path plus the length of the shortest feasible $w-t$ path. Simply search over all n possible hop vertices to compute the shortest overall path.

```
Algorithm 2: 1-feasible BFS
  Input: Graph $G = (V, L \cup R)$ and vertices $s, t$.
1 Use the modified subroutine from (a) to compute, for every node $v$, the shortest path $P_v$ from $s$ to $v$;
2 Use the modified subroutine from (a) to compute, for every node $v$, the shortest path $Q_v$ from $v$ to $t$;
3 If $\min_{v\in V}(|P_v| + |Q_v|) = \infty$, where $|\cdot|$ denotes the length of the path, return "Not Feasible" ;
4 Otherwise, return $P_w + Q_w$, where $w = \arg\min_{v\in V} |P_v| + |Q_v|$ and $+$ denotes path concatenation;
```

Alternative Solution: An alternative solution is to modify the graph further. In the new graph, for every original node v we add 4 nodes, $v_{r,\text{not taken}}$, $v_{\ell,\text{not taken}}$,$v_{r,\text{taken}}$, $v_{\ell,\text{taken}}$. These nodes represent all possible states one can arrive at the original node, i.e. with the left foot or right foot and having taken the double hop or not. The edges in this graph among the "not taken" nodes are the same as in part (a), i.e. $\{u, v\} \in R$ becomes the two directed edges $(u_{r,\text{not taken}}, v_{\ell,\text{not taken}})$ and $(v_{r,\text{not taken}}, u_{\ell,\text{not taken}})$ in E'. The edges in this graph among the "taken" nodes are also the same as in part (a). Finally, for any edge $\{u, v\} \in R$, one adds the directed edge $(u_{r,\text{not taken}}, v_{r,\text{taken}})$ and for any edge $\{u, v\} \in L$, one adds the directed edge $(u_{\ell,\text{not taken}}, v_{\ell,\text{taken}})$ to signify that a double hop was taken allowing you to stay in the same foot.

Proof of Correctness:

A 1-feasible path P in G has at most 1 vertex w at which the "hop" occured, meaning the edges to and from w belong to the same set (both belong to L or both to R) and aside from at w, P alternates between edges in L and R. Therefore, P can be thought of a concatenation of two feasible paths: a feasible path from s to w and one from w to t. If P is actually a feasible path with no hops, it can still be written in the form above, where $w = s$ and one of the paths is empty.

We've just shown that any 1-feasible path can be written as the concatenation of two feasible paths. We also claim the converse, that every path P of the form $P_1 + P_2$, where P_1 and P_2 are feasible paths, is 1-feasible. This is straightforward. We know that P_1 alternates edges in L and R. If the first edge of P_2 is in the same set as the last edge in P_1, then our overall path will be feasible and hence 1-feasible. Otherwise, P will have two consecutive edges in the same path at w, but thereafter P_2 alternates so there will be no more repeated edges and P will be 1-feasible.

Finally, note that G is undirected so finding the shortest paths from t to every other vertex is equivalent to finding the shortest paths from arbitrary vertices to t. In minimizing over the possible "hop" vertices as we do in the algorithm, we consider only 1-feasible paths (by ($\Leftarrow$)). Also, a shortest feasible path has some hop vertex (possibly s), and since we consider every possible hop vertex, we are guaranteed to consider a shortest 1-feasible path overall. In other words, it is not possible for us the "miss" the shortest 1-feasible path since we considered its hop vertex and picked another path that was no longer than it. Therefore, since we minimize over all hop vertices, our algorithm returns the shortest 1-feasible path, as desired.

Runtime Analysis:

Our algorithm does 2 runs of the algorithm from a), which each run in $O(n + m)$. We also take the minimum over a set of n lengths, which runs in $O(n)$. Hence, the overall algorithm is $O(n + m)$, as desired.

(c)

Suppose there are also some edges that you are allowed to step on with either foot. Give a one sentence explanation of how to modify your algorithms to work with these new edges.

In the construction of G', just add these new vertices to both directions of edges, as if they were in both L and R.