Math 182: Problem Set 5
Kenny Guo

Question 1:

Every day, you and your friend both walk from your homes to your jobs. You live relatively close to each other so you often pass by each other while walking to work. However, you are both very busy and don't often see each other, so it is very important to you that you see your friend while walking to work each day. You also both dislike walking, so you each prefer to find a route to work that is as short as possible. Design an efficient algorithm to find routes for you and your friend to walk to work so that the two routes meet in at least one point (so that you get a chance to see your friend) and are as short as possible given that constraint (so that you don't have to walk too much).

Assume that you are given an undirected, weighted graph $G = (V, E)$ where the vertices represent the intersections of streets in your city, the edges represent the roads between those intersections and for each edge $(u, v) \in E$, the edge weight $w(u, v)$ represents the amount of time it takes you to walk down that road. You are also given vertices s_1, t_1, representing your home and your workplace and s_2, t_2, representing your friend's home and workplace. You need to find a path from s_1 to t_1 and a path from s_2 to t_2 such that the two paths have at least one vertex in common (it is okay if they have more than one vertex in common) and the sum of the lengths of the two paths is as short as possible given this constraint.

For convenience, your algorithm only needs to report the sum of the lengths of the two paths, not the paths themselves (though you should think about how you would find the actual paths efficiently). If no such paths exist, your algorithm should return "∞." For full credit, your algorithm should run in time $\Theta((|V| + |E|) \log |V|)$.

Idea: Let $d(x, y)$ be the shortest time from node x to y. If we suppose the paths meet at some vertex v, then the best path for $i = 1, 2$ has time $d(s_i, v) + d(v, t_i)$, which we can find using Dijkstra 4 times. We then minimize the sum of these over all meeting points v.

For this algorithm, assume Dijkstra is implemented where it returns an array of the times for the shortest possible paths between the start node and all other nodes in G.

```
ShortestPathsWithMeet(G, w, s1, t1, s2, t2):
    d_s1 = Dijkstra(G, w, s1)
    d_t1 = Dijkstra(G, w, t1)
    d_s2 = Dijkstra(G, w, s2)
    d_t2 = Dijkstra(G, w, t2)

    best = infinity
    for v in V:
        total = d_s1[v] + d_t1[v] + d_s2[v] + d_t2[v]
        best = min(best, total)
    return best
```

This has $\Theta((|V| + |E|) \log |V|)$ runtime.

Proof. The algorithm performs Dijkstra four times, for a total runtime of $\Theta((|V| + |E|) \log |V|)$. Then, it finds the minimum by looping over all $v \in V$, for $\Theta(|V|)$ time. The former dominates. □

This algorithm returns the shortest sum of times of the two paths with at least one vertex in common.

Proof. For any vertex v, any path from s_i to t_i that passes through v can be split into a path from s_i to v and one from v to t_i. Dijkstra (along with the fact the graph is undirected) ensures these are minimized in weight, so the claim made in the 'Idea:' section above is true.

If we consider an optimal solution that exists, it must have at least one meeting vertex, which our algorithm considers and thus returns the minimum weight of its respective path. If no solution exists, it means that either t_i is not reachable from s_i (in which case, Dijkstra would return at least one of $d(s_i, v)$ or $d(v, t_i)$ as ∞), or there is no vertex that can be shared in any paths of the two players (in which case, any v would have to be unreachable by at least one of the players, and their distances to it would be ∞ by Dijkstra). Thus, the algorithm returns the correct solution. □

Question 2:

Given a weighted graph $G = (V, E)$ and vertices $u, v \in V$, it is possible that the shortest path from u to v is not unique–that is, it is possible that there are multiple distinct paths from u to v that all have the same minimal length. Design an efficient algorithm that, given an undirected, weighted graph $G = (V, E)$ with positive edge weights and a vertex $s \in V$, will find an array U such that for each $v \in V$, $U[v] =$ True if and only if the shortest path from s to v is unique (and $U[v] =$ False otherwise).

Idea: We modify Dijkstra's algorithm and keep an array U of bools where $U[v]$ indicates whether the shortest path from s to v is unique. We initialize all as True, but when we find a different path with same shortest distance, we mark it as False. When we find a strictly shorter path to a vertex, we update the shortest distance and whether or not the path was unique.

```
UniqueShortestPaths(G=(V,E), w, s):
    d = new array length |V|
    U = new array length |V|
    for v in V:
        d[v] = infinity
        U[v] = True
    d[s] = 0

    Q = new priority queue containing each v in V with priority d[v]
    while Q is not empty:
        x = DeleteMin(Q)
        for each neighbor y of x:
            dist = d[x] + w(x,y)
            if dist < d[y]:
                d[y] = dist
```

```
            U[y] = U[x]
            ChangePriority(Q, y, d[y])
        else if dist = d[y]:
            U[y] = False
return U
```

This algorithm runs in $\Theta((|V| + |E|) \log |V|)$.

Proof. It is the same as Dijkstra, along with a $O(|V|)$ U array initialization, and additional constant work for U for each shortest path update or checking if there are multiple shortest paths. □

This algorithm is correct: $U[v]$ is True if and only if the shortest path from s to v is unique.

Proof. Since distances are computed via Dijkstra, $d[v]$ are the shortest distances from s to v. We consider the following cases when U of some vertex y is updated:

1) If the algorithm finds a strictly shorter path to a vertex y through a vertex x, then the new shortest paths to y are the shortest paths to x, followed by the edge (x, y). Thus $U[y]$ should be equal to $U[x]$, since if there is only one shortest path to x, there is only one shortest path to y, and if there are more than one shortest path to x, there are more than one shortest path to y.

2) If the algorithm finds another path to y with length equal to $d[y]$, then there are at least two shortest paths to y. Thus, the algorithm sets $U[y] =$ False.

Dijkstra considers all possible shortest paths, and so these cases show $U[v]$ is True iff the shortest path from s to v is unique. □

Question 3:

Consider the following algorithm to find a minimum spanning tree. At each step, find the heaviest edge e which is part of a cycle, and remove it; do so until there are no edges remaining. Let T be the graph resulting from this. (Note that T is a tree, since it is connected and has no cycles.) Prove that T is a minimum spanning tree.

Proof. We argue via a "reverse" exchange argument. Let G be the original graph, and suppose on some given step, e in some cycle C is removed, which is the heaviest.

We claim there exists a MST that doesn't contain e. Let T be a MST of G. If $e \notin T$, we are done. Suppose $e \in T$. Removing e from T results in the tree being cut into two components, and there must be some edge $e' \in C$ that crosses this cut, since e was part of a cycle so there is still a path between its endpoints after its removal. Adding e' to T reconnects the tree, call this T', and since e was the heaviest edge in the cycle, so $w(e) \geq w(e')$, we see

$$w(T') = w(T - e + e') = w(T) - w(e) + w(e') \leq w(T),$$

meaning T' is still optimally a MST that doesn't contain e.

We iterate until there are no more cycles left. Since removing an edge from a cycle doesn't disrupt connectivity, we reach a spanning tree at termination (i.e. the number of edges left is $|V| - 1$). Finally, since after every step, there is some MST that doesn't contain any deleted edges, we know the spanning tree we end up with after deleting $|E| - |V| + 1$ edges must be optimal. □

Question 4:

The width of a path in a weighted graph is the minimum edge-weight of an edge in the path. Design an algorithm that, given a weighted connected undirected graph G and vertices s and t, finds the width of the widest path from s to t in G.

Hint: Maximum spanning tree.

Idea: Find a maximum spanning tree (adapting the Prim algorithm). Then take the unique path from s to t and find its width.

```
PrimMax(G = (V,E), w):
    E' = new empty list
    used = new length |V| array. Initialize all to false.
    Q = new empty priority queue
    Pick any v in V and set used[v] = True
    For all neighbors u of v:
        add (v,u) to Q with priority -w(v,u)
    while length(E') < |V|-1:
        (x,y) = Pop(Q)
        if used[y] = False:
            used[y] = True:
            add (x,y) to E'
            for all neighbors z of y:
                add (y,z) to Q with priority -w(y,z)
    return (V, E')

WidestPath(G, w, s, t):
    T = PrimMax(G,w)
    Use BFS in T to find the unique path P from s to t
    width = infinity
    for each edge e in P:
        width = min(width, w(e))
    return width
```

The runtime is $O(|V| + |E| \log |V|)$.

Proof. PrimMax takes $O(|V| + |E| \log |V|)$. Using BFS on a tree to find the path between s and t takes $O(|V|)$ time since $|E| = |V| - 1$. Thus, total time is $O(|V| + |E| \log |V|)$. □

This algorithm correctly returns the width of the widest path from s to t in G.

Proof. We use an exchange argument. Let T be a maximum spanning tree of G, and let P be the unique path from s to t. Let

$$e' \in \text{argmin}_{e \in P} w(e)$$

be an edge with minimum width $\bar{w}$ in P.

Remove e' from T, cutting T by its components. Any path from s to t must cross this cut. Suppose for contradiction there is some edge e" with width $w > \bar{w}$ that crosses this cut. Then

$T' = T - e' + e$" is a spanning tree, but $w(T') > w(T)$, contradicting T being a maximum spanning tree.

Thus, every edge crossing the cut has weight at most $\bar{w}$. Since any path from s to t must cross this cut, they all must have an edge with weight at most $\bar{w}$. Since the path in T already maximizes this with a width of $\bar{w}$, the algorithm returns the widest path. □