Problem Set 7

All parts are due Friday, November 1 at 6PM.

Name: Diego Escobedo

Collaborators: Nick Baginski, RItaank Tiwari, Ben Dover, Noah Lee

Problem 7-1.

(a) A: 8 B: 9 C: 7 D: 8 E: 11 F: infinity G: 12 H: 10 S: 0
Order: S, C, A, D, B, H, E, G, F

(b) The vertex that this happens to is vertex C. Vertex C is deleted from the priority queue second, right after S, and its weight is 'locked in' at 7. However, when the priority queue deletes G (which has a weight of 12 at this point) and looks at all its edges, it sees that 12 + (-6), its potential weight for (G,C) has a lower weight than 7. However, Dijkstra's should never update the weight of a vertex that has already been removed from the priority queue.

Problem 7-2. The first thing we are going to do is create a graph G' that has all the same vertices and all the same edges, but they are now going to be unweighted. Because they are unweighted, we can run BFS in $O(V+E) = O(V^2)$ time, and it will tell us all of the connected components. Now, if we analyze the connected component that contains vertex s, which is composed of vertices I and edges J, we can search it for negatively weighted edges in $O(I + J) <= O(V + E) = O(V^2)$ by comparing it to our original G. We know that having an undirected, negatively weighted edge in a graph will cause an infinite loop. So, any vertices not in the connected component containing s will have a shortest path of infinity. Further, for the vertices in the connected component containing s, there are two options: If there is a negative edge in the connected component, then the shortest paths are all negative infinity. If there are no negative edges, then we can run Dijkstra's from s to find the shortest paths, and we will be guaranteed to find the correct shortest paths in $O(Vlog(V)+E) = O(V^2)$ time. Because we are running a constant number of $O(V^2)$ operations, the whole process runs in $O(V^2)$.

Problem 7-3. This problem is quite simple. We simply run Johnson's algorithm in $O(VE + V^2 log(V)) = O(V^3)$ time. Now, we have every node's list of shortest paths. From here, we simply pick each node's longest shortest path, which because we have to look through V nodes V times, takes $O(V^2)$ time. Now that we have our list of longest shortest paths, we can pick the shortest longest shortest path in $O(V)$ time. Further, we know that this algorithm is correct because Johnson's algorithm works for general graphs with any weights (as long as they don't have negative weight cycles), which means it works for the graph we are given. Thus, we can find the radius correctly and in $O(V^3 + V^2 + V) = O(V^3)$ time.

Problem 7-4. We start by creating a graph G, where the nodes are the junctions and the edges are the paths between junctions, and the the weights between edges are the lengths of the paths.

The advantage to this problem is that we do not need to use shortest path algorithms to find the path from the entrance junction to the mansion. We simply need to find out if a path exists from the junction to the mansion. A good idea is to iteratively try to remove nodes that are arbitrarily close to the camera junctions. One strategy that we could do is to create a supernode connected to the motion sensor junctions by weight 0 edges. From there, we could run Dijkstra's and find, for every node, its shortest distance to a camera node. We can store this information in a list, and then run binary search on it to figure out how many nodes to remove. In other words, we will have an ordered list with n nodes and their distance from a camera junction. We can create a temporary graph without the first $\frac{n}{2}$ nodes, and run BFS to see if we can get to the mansion. If we can, we can be more ambitious and remove $\frac{3n}{4}$ nodes, and if there is no viable path, then we would instead try the first $\frac{n}{4}$ nodes. This way, because of the way binary search works, then we would only have to run BFS logn times. This argument is correct because we are successively checking what the furthest possible shortest distance to a camera node is, and narrowing it down to the minimum possible without ruining our chances of getting to the mansion.

Runtime: because the E is bounded by 4N, $E = O(V)$. Running Dijkstra's from the supernode takes $O(m + nlogn) = O(n + nlogn) = O(nlogn)$. Running BFS takes $O(m + n) = O(n)$ time, and since we are running it every time we do a binary search, we run it logn times, meaning the runtime of the BFS's is $O(nlogn)$ time. Thus, because all of our operations are bounded by $O(nlogn)$, our whole agorithm runs in $O(nlogn)$.

Problem 7-5. For this problem, we are going to use the graph layering technique that we have learned in class. We are going to create a graph G' with k+1 layers. Layer 0 will represent her having 0 empty pocket spheres, and layer k will represent when she has k empty pocket spheres. In layer i, a clearing c is represented as a vertex c_i. Our trails are going to be represented as edges, but we are going to be using some special rules to connect edges. The weights of our edges are going to be the length of the edges.

The first special rule for our edges that we are going to have is that we are going to connect unweighted directed edges from stores on different layers, starting from bottom to top. This means that a store node u_0 on layer 0 will be connected to the same shop u_1, which will in turn be connected to $u_1, u_2,$ etc. successively all the way to layer k. The store at layer k will not be connected to any of the other stores. This would allow Ashley to drop off any amount of filled pocket spheres and replenish with empty ones when she gets to a store.

If we have an edge going from u to v in our original graph, weighted by length l and containing c critters, then we are going to draw directed edges from u_0 to v_c, from u_1 to v_{c+1}, so on and so forth until u_{k-c} to v_k, because if we drew any edges on any of the higher layers, then we would be sad, which we don't want. Because the paths are undirected, we also do the above step bu tswitch the order of u and v. These edges are all weighted by l. They are climbing the layers to represent the fact that Ashley is 'losing' empty pokeballs as she travels the paths.

Now that we have Our graph, we simply run dijkstra's from trundle town on layer k to blue bluff on layer 0. If we can't reach it, then we know that it is impossible to get there without having some sadness. This argument is correct because dijkstra's fits udner tehse parameters and our graph perfectly models the real world scenario and covers all edge cases.

Runtime: edges are bounded by $O(nk)$, and instead of there being n edges, there are nk edges. Therefore, Dijkstra's runs in O(nklog(nk)), which is our target time.

Problem 7-6.

(a) We can do a proof by contradiction. For the first statement we cant to prove, assume that b(s,t) < min(b(s,v),w(v,t)) for some v. If b(s,v) is smaller than w(v,t), then we could have to have the inequality b(s,t) < b(s,v) hold. However, because v is a neighbor of t, then the path from s to v is a subpath of a path from s to t. Therefore, this can't be true, because the maximum bottleneck from s to v would be a maximum bottleneck from s to t also. The other alternative is that b(s,t) < w(v,t), which can't be true, because if w(v,t) has to be less than b(s,v) which means that it has to be the bottleneck which means it would be a bottleneck itself, meaning that it would have to be equal to b(s,t) and not greater than it.

For the second statement, we have a similar argument to the first. Because (s,v*) is a subpath of (s,t), the bottlenecks must be the same. Except, if the last segment is shorter than the bottleneck, then it must in itself be the bottleneck, meaning that the bottleneck overall is equal to the longest segment.

(b) This stuff is easy cheese (not actually) and its like 3am as I write this so apologies for the brevity and kind of bad explanation.

The first pass is going to look a little like this:

```
def dijkstra_wts(succ, wt_capacities, source_name, target_name):
    width = dict()
    for city in succ:
        width[city] = float('-inf')

    width[source_name] = float('inf')
    Q = set(city for city in succ)
    while Q:
        u = max(Q, key= lambda key: width[key])
        Q.discard(u)
        if width[u] == float('-inf'):
            break

        for v in succ[u]:
            alt = max(width[v],min(width[u], wt_capacities[u][v]))
            if alt > width[v]:
                width[v] = alt

    return width[target_name]
```

Why is this correct? At any point in the algorithm, there will be 2 sets of vertices A and B. The vertices in A will be the vertices to which the correct maximum minimum capacity path has been found. And set B has vertices to which we haven't found the answer.

IH: At every step, all of the nodes in A have accurate values of maximum minimum capacity.

Correctness of base case: Trivial when tehre is one node in A and tehre is infinity capacity to move things into itself.

When we iterate, we are setting val[W] = max(val[W], min(val[V], widthbetween(V-W))).

IS: Assume W is the vertex in our set of unoptimized nodes B with the biggest val[W]. So, W is removed from the queue and we temporarily assigned val[W] to node W .

Now, we show that every other path from S to W $<=$ val[W]. This will be always true because all other ways of reaching W will go through some other vertex (call it X) in the set B.

And for all other vertices X in set B, val[X] $<=$val[W]

Thus any other path to W will be constrained by val[X], which is never greater than val[W].

Thus the current estimate of val[W] is optimum and hence algorithm computes the correct values for all the vertices.

Great, now we have the value of w. Now we simply run dijkstra's, but in the line where we usually say length[v] $>$ length[u] + costs[u][v]: we now say length[v] $>$ length[u] + costs[u][v] and widths[u][v] $>=$ maxwidth:, because this way we are metaphorically removing paths that do not have the capacity for w.

This argument is correct. Please believe me I just explained hella.

Runtime: We are running Dijkstra's twice, and dijkstra's is O(m+nlogn). However, I was a bad coder and i think it actually runs in O(m+n^2) because I'm using dictionary iteration to find the min/max instead of fibonacci queues. oopsie. However, its chill because we are told that the number of edges is actually n and the number of nodes is bound by $O(\sqrt{n})$. So, it actually runs in $O(n + \sqrt{n}^2) = O(n)$. Lowkey kinda lit. Thanks peace out sorry for my language dawg love u.

(c) Submit your implementation to `alg.mit.edu`.