Problem Set 7

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

Name: Lydia Yu

Collaborators: Herbert Turner, Pranit Nanda, David Magrefty

Problem 7-1.

(a) $\delta(s, v) = \{s : 0, a : 8, b : 9, c : 7, d : 8, e : 11, f : \infty, g : 12, h : 10\}$
order: s, c, a, d, b, h, e, g, f

(b) Vertex c: the algorithm tries to change c's path estimate after c is already removed from the queue because of the negative edge weight going into c.

Problem 7-2. We will run DFS on the graph to determine if there are any edges with negative weights; if there are negatively-weighted edges, then you would end up with a negative cycle and return that there is no shortest path. This takes O($|V| + |E|$), which is just O($|V|^2$) since $|E| = \Omega(|V|^2)$. If there are no negative edges, then we run Dijkstra's algorithm on the graph starting from the source vertex, which will give us the shortest paths from s to all of the other vertices in the connected part of the graph. This takes O($|V| \log |V| + |E|$), which is also O($|V|^2$) because $|E| = |V|^2$. Compared with PS5-2, this running time is slower because the edges are weighted; however, the basic logic of finding the shortest paths to every vertex from every vertex, taking the largest of those shortest paths for each vertex (getting the eccentricity), then taking the smallest of these eccentricities is the same for both problems.

Problem 7-3. We want to find the smallest of all of the shortest paths to the farthest vertex for each vertex in the graph. Run Johnson's algorithm to get all of the shortest paths from each vertex to every other vertex in the graph. For each starting vertex, we will look for the largest shortest path, which gives us the shortest path to its farthest vertex: this is the eccentricity of that starting vertex. Then, out of all of these shorts paths to the farthest vertex for each vertex in the graph (all of the eccentricities), we take the smallest one to get the radius of the graph. The number of edges in the graph is $|V|^2$ because the graph is directed and connected, which means that each vertex can be connected to every single other vertex (there are $|V|$ vertices that each can be connected to $|V| - 1$ other vertices). Johnson's algorithm runs in O($|V||E| + |V|^2 \log |V|$); plugging in $|E| = |V|^2$, we get a running time of O($|V|^3 + |V|^2 \log |V|$) = O($|V|^3$).

Problem 7-4. Create an undirected graph with the junctions as the vertices and the pipes as the edges, weighted by their length. Also, create a supernode that connects to all of the junctions that have sensors and set the weights of those edges to 0. Run Dijkstra's algorithm from the supernode to get the shortest path lengths from the supernode to every other vertex in the graph. The information that we get from running the algorithm also tells us the distances from the sensors to the vertices that are closest to them (this is because algorithm returns the shortest distance from the supernode to all other vertices, and since the edge weight of the supernode to the sensors are 0 and we must pass through a sensor in order to get to other vertices in the graph from the supernode, we know that the distance from the supernode to vertices in close proximity to a sensor is equal to the distance from a sensor to the vertices that are closest to it). Running Dijkstra's takes O($|E| + |V| \log |V|$); we know that $|V| = n$ and $|E| = n - 1$ since the graph is connected (every junction is reachable from every other junction), so the running time of this part is O($n \log n$).

Now that we have all of the distances from the closest vertices to the sensors, we merge sort them so that we can find which threshold distance allows her to find a path from the entrance to her destination while staying as far away from the sensors as possible. Merge sort takes O($n \log n$) since there may be n vertices maximum that are all closest to the sensors. With this sorted list of distances, we essentially do a binary search to find the optimal distance threshold.

Starting at the middle distance, we create an unweighted, undirected graph with the junctions as vertices and pipes as edges, excluding the junctions that have distances less than the chosen threshold distance from a sensor (which we know from the Dijkstra result above). We run BFS from the entrance to the mansion. If a path is not found, this means that the the start and end nodes are not connected anymore because we have removed too many junctions, so we have to decrease the sensitivity (distance threshold) to be more inclusive of nodes by going to the middle distance in the lower half of our sorted list and running BFS based on a graph with that new distance threshold (add back the vertices are now no longer smaller than this new threshold). However, if a path is found, then we will want to check if we can tighten our threshold by increasing sensitivity to be less inclusive of vertices. We go to the middle distance in the upper half of our sorted list and run BFS on a graph with this new threshold (remove the vertices with distances that are now less than the threshold). We continue this binary search until we find the optimal distance (increasing the sensitivity to a higher distance results in a failure to find a path from start to end).

Binary search takes O($\log n$), so we will have to run BFS, which takes O(n) because there are n vertices and O(n) edges, O($\log n$) times, for a total of O($n \log n$). Thus, the overall running time of this algorithm is O($n \log n$).

Problem 7-5. Construct an undirected graph where the vertices are represented by the clearings and the edges are represented by the trails, which are weighted by trail length. For each clearing v, we want to create k+1 vertices (1 vertex for each possible amount of spheres that she can be holding at a time, from 0 to k): $\{v_0...v_k\}$. If there is a trail connecting two clearings u and v, create a directed edge from u_i to v_{i-c_t} for $i \in \{c_t...k\}$ (we only want valid non-negative vertex subscripts) if v does not contain a store. Since c_t is the capacity of critters along that trail, we ensure that two clearings are connected only if she has enough spheres at the starting node to collect the capacity of critters along the trail to the ending node because she wants to collect all of the critters along the trail. However, if v does contain a store, connect a directed edge weighted by trail length from u_i to v_k for $i \in \{0...k\}$ because when she visits a store, she will deposit full spheres and buy empty spheres until she reaches her backpack's capacity.

Starting from the Trundle Town node with k spheres, run Dijkstra's algorithm on this graph to get the shortest path from Trundle Town to Blue Buff without ever being sad (the nodes in the graph are connected only if she has enough spheres in the starting node to collect the entire capacity of critters along the trail to the end node). If Dijkstra's algorithm returns that the shortest path from Trundle Town to Blue Buff has a length of infinity, then we know that sadness is unavoidable because there is no path that connects the two towns that allows her to collect every critter she sees.

The running time of Dijkstra's is O($|E| + |V| \log |V|$): $|E|$ is a constant multiple of the number of vertices because we know that each clearing connects to at most 5 trails. $|V| = O(nk)$ because there are n clearings, each of which has k+1 states for the k+1 different amounts of spheres she can have at each clearing. Thus, we get O($nk + nk \log(nk)$) = O($nk \log(nk)$).

Problem 7-6.

(a) b(s,t) means that for each of the possible paths from s to t, we will take the minimum edge weight found in each path (which is the bottleneck), and then for all of these bottleneck values, we take the maximum to get the maximum of all of the minimum edge weights for each path. If $b(s,t) < min(b(s,v), w(v,t))$, then that means that one of the other edge weights along one of paths from s to t has a greater weight (since b(s,t) is the maximum of the minimum edge weights out of all the paths from s to v, and w(v,t) is the weight of the edge from v to t). However, if one of these is greater than B(s,t), then that means that the maximum bottleneck should be b(s,v) or w(v,t), which contradicts the statement that b(s,t) is the maximum bottleneck.

b(s,v*) and w(v*,t) account for all the possible edge weights from s to t because b(s,v*) is the maximum bottleneck from s to v* (so it takes all of the minimum weights from all the paths from s to v* and takes the maximum of those values), and w(v*,t) is the last edge's weight in the path from s to t; we know that the maximum bottleneck will need to be one of these values because otherwise we are saying that b(s,v) is not found in this graph since b(s,v*) and w(v*,t) account for all the possible edge weights in paths from s to t.

(b) First, create a directed graph with the cities as the vertices and the routes connecting a starting city to an ending city as the edges, weighted by the weight capacity for a truck going along that route. To get w*, the largest single server that can be shipped from SF to Cambridge, we want to find the largest of all the bottlenecks (the minimum weight capacity along the paths from SF to Cambridge) because that weight capacity tells us the largest server that meets the weight limit on a possible shipping route from SF to Cambridge.

Now, we run Dijkstra's algorithm on the graph from SF; for all of the paths from SF to Cambridge, we will get the bottleneck associated with each path, and then we will return the largest of those bottlenecks. However, we will modify the relaxation step of Dijkstra's algorithm because we are no longer looking for the shortest path from the starting vertex to the other vertices (this is a sum of weights); instead, we want the maximum bottleneck (a singular value and not a sum) for a given path. We initialize all of the bottlenecks for each node in the graph as 0 (the bottleneck for SF will be infinity since it is the starting node), and we relax starting from the node with the highest bottleneck value and then pop that node out of the queue after no more relaxations are possible. Every round of relaxation until the node is popped out of the priority queue will update the estimated bottleneck of that node with a larger value if found. For the Cambridge node, each relaxation represents a new path being found from SF to Cambridge; the estimated bottleneck value for the route will be updated if the minimum weight in this new path is larger than the previously found largest minimum weight. Once no more possible new paths from SF to Cambridge can be found, we pop it out of the queue that is kept in Dijkstra's algorithm and the value that is associated with the Cambridge node will be the maximum bottleneck for the

SF-Cambridge paths.

We know that this modification will actually result in the maximum bottleneck value for each node when it is popped out of the queue because if it were not the maximum value, then the algorithm would continue running and attempting to update the node's value. The vertex is popped out when all possible paths to it have been accounted for, so we know that we will end up with the largest of all of the minimum weights.

Once we know what w* is, we can then find the minimum cost by creating the same graph as the one from above, except this time, the edges are weighted by the cost of shipping. We will not include any edges with weight capacities less than w* in this graph because we will not be able to ship our largest single server along that route. We then run normal Dijkstra's algorithm on this graph starting from SF, and we will get all of the shortest paths based on cost from SF to all other cities in the graph. The minimum cost to ship the server will be the value associated with Cambridge that the algorithm returns.

Running each of the Djikstra's algorithms will take O($|E| + |V| \log |V|$); we know that $|V| < 3\sqrt{n}$ and $|E| = n$ (n trucking routes each with less than $3\sqrt{n}$ cities), so the running time simplifies to O($n + 3\sqrt{n} \log(3\sqrt{n})$) = O(n).

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