Problem Set 5

All parts are due April 26th, 2016 at 11:59PM.

Name: Arturo Chavez-Gehrig

Collaborators: Sabrina Ibarra

Part A

Problem 5-1.

(a) In order to find the largest possible net calorie loss for Donald's run, we will calculate the net calorie loss on each edge. To do this, we will iterate through all the edges and calculate $gain(e) - loss(e)$ and assign that value as the weight for each edge. Now that we have a directed acyclic graph, so we can find the shortest path by using the DAG-SP algorithm. DAG-SP finds the shortest paths by first, performing a topological sort of the vertices. Then, it relaxes the edges coming out of each node, doing so in topologically sorted order. Once we have the shortest paths to all the vertices, we can reconstruct the optimal path from start to target by using the parent pointers. The run time for DAG-SP is $O(V+E)$. The run time of calculating the net calorie loss of each edge is $O(E)$. Total run time to find the shortest path with the maximum calorie loss is $O(V+E)$.

(b) Given the constraint that Donald will never have net positive calorie gain during his run, we will augment the algorithm designed in part a to restrict Donald from traversing those edges where the constraint is binding. By adding the if statement that prevents the traversal of edges that cause positive calorie gain, we do not affect the order of the run time. The overall run time is still $O(V+E)$.

Constrained-DAG-SP(Graph,weights,start,end)

topologically sort the vertices of G

for each vertex v in V[G]

$d[v] = \infty$

$p[v] = NIL$

```
(d[s]=0
for each vertex u, taken in topologically sorted order
    for each vertex v in Adj[u]
        if d[u] + w(e) > 0:
            Continue
        else:
            RELAX(u,v,w)
x = target
path = [ ]
while p[x]! =NIL
    add p[x] to the path
    x = p[x]
return path
```

(c) Now that Donald has reduced the constraints on his paths as well as the definition of the graph, we will need to use a different shortest path algorithm. We can use Bellman Ford on a graph with net calorie loss edge weights calculated the same as in part a. We will find either the shortest path or identify a negative weight cycle. If we find a negative edge cycle, we will return 'unbounded weight loss'. Calculating the net calorie change for each edge takes $O(E)$. Running Bellman Ford takes $O(V * E)$, so total run time is $O(V * E)$.

(d) Given this new constraint for Donald's run being greater than or equal to $enough$, we will run Bellman Ford identically to in part c. If no negative weight cycles exist, then Bellman Ford will return the correct shortest path that defines the largest possible calorie loss for Donald's run. However, in the case that there is a negative weight cycle detected by the usual Bellman Ford algorithm, instead of returning 'unbounded weight loss', we will construct a path that allows Donald to run $enough$ distance. Instead of relaxing all edges once to detect the negative weight cycle, we will add an if statement that checks if we have reached $enough$ distance for Donald's run. We will continue relaxing all edges until the if statement is true, then we will halt. To construct the path, we will compare the path returned by the first main loop of Bellman Ford, tracing parent pointers, with the path returned once the if statement is satisfied (again tracing parent pointers). We will be able to deduce the path to the cycle using the first path, and we will be able to deduce the rest of the path using the path returned

by the second main loop of Bellman Ford. We will count the number of iterations taken in the second loop of Bellman Ford until the if statement is satisfied. This will inform us of the number of loops in the negative weight cycle that were taken for Donald to reach $enough$ distance. These facts to deduce Donald's path. The run time of regular Bellman Ford takes $O(VE)$ and the second main loop of Bellman Ford (usually taking $O(E)$) is now bounded by the $enough$ weight. So now this second loop runs in $O(|V|^C)$. So the total run time of this algorithm is $O(VE + V^C)$.

Problem 5-2.

(a) Proof by induction

Induction hypothesis: If u is any node in V, then $d[u] \geq \delta[u]$

Base case:

In Dijkstra algorithm, d[u] is initialized to ∞, so this satisfies the property since $\infty \geq \delta[u]$, since $\delta[u]$ is a constant in a connected graph or is ∞ if no path exists to the node.

Inductive step:

Assuming that the induction hypothesis holds for the nth iteration of the main loop in Dijkstra. The only way for d[u] to be decreased is if an edge leading to a vertex u is relaxed. When an edge is relaxed, the value of d[u] is updated to $d[x] + w(x, u)$ if this new value is smaller than what was previously stored in d[u]. This cannot be smaller than the value of $\delta[u]$. By the triangle inequality $\delta(u) = \delta(x) + \delta(x, v)$, d[u] remains greater than or equal to $\delta[u]$. For it to be smaller then $d[u] < d[x] + w(x, u)$, which is impossible since that is how d[u] is updated in the first place. This process is repeated until d[u] has the shortest length in Q. At this point, it is popped from Q and added to S. As we will prove in part b, $d[u] = \delta[u]$ and the value of d[u] will not be updated again.

(b) After any number of iterations in the main loop, if u in S then $d[u] = \delta[u]$.

Proof by Contradiction:

A vertex u is added to S if it popped from the priority queue, meaning all paths of length less than d[u] have been explored. This is because all weights are nonnegative so none of the paths that will be explored in the next iterations will return a shorter distance to u. Suppose $d[u] > \delta[u]$ and u exists in S, but vertices are added to S only once, so if $\delta[u] < d[u]$ then u would have already been added to S and no updates to d[u] would have been made in the main loop of Dijkstra. Contradiction. If d[u] is added to S then all further paths will be larger, so d[u] won't be updated once it is in S.

(c) Proof by induction:

Induction hypothesis: If u is any node, then $d[u] = \delta_S[u]$.

Base case: When there is no such path $\delta_S[u] = \infty$ all values d[u] are initialized to ∞. Therefore, the property holds at initialization.

Inductive step: This property is an example of the optimal substructure property of shortest paths. For every node in the graph, the shortest path is made up of other shortest paths. All confirmed shortest paths are included in S as proven in part b. Edges are only relaxed if their source node is popped from the priority queue (when also the node is added to S). Because of this ordering of the relaxation of edges, all nodes that do not have an edge connecting them to a node in S will have d[node]$= \infty$ and $\delta_S[u] = \infty$ as well. However, nodes that do have a connecting edge (or are in S) will have a path made up of edges between vertices in S (with perhaps one additional edge to a node not in S). Therefore, for all nodes, after any number of iterations through the main loop, d[u] will equal $\delta_S[u]$.

(d) If u is in S and v is in Q, then $\delta[u] \leq \delta[v]$.

Proof by Contradiction:

Dijktra's algorithm is commonly described as if you had water flowing out from the start node at a constant flow rate and the algorithm discovers the shortest paths to the nodes as the frontier is expanded. The most recently added node to S defines the frontier of which the algorithm has explored shortest paths up to that distance. Suppose u is in S and v is in Q, but $\delta[u] > \delta[v]$. It's intuitive to see that nothing in Q can have a shorter path than anything in S otherwise it would have been explored already since it falls inside the frontier. Dijkstra's algorithm expands the frontier radially outwards. More formally, since u is in S, the frontier is defined by the length of its shortest path. If $\delta[u] > \delta[v]$, then v falls within the frontier and must exist in S. Contradiction, v was defined to exist in Q, and a node v cannot exist in both Q and S, because Q is defined to be $V - S$. Therefore, for all nodes u in S, their shortest path is less than or equal to the shortest path to all nodes v in Q.

Problem 5-3.

(a)

(b)

(c)

(d)

(e) Dijkstra $= 30000$

Bidirectional Dijkstra $= 12799$

A* $= 8355$

In Dijkstra, edges are explored as distances increase radially outward. Despite the target being in the corner, all directions are explored in the same depth. In the case of this situation, where the start node is in the center and the target is in the corner, much of the graph is explored, perhaps unnecessarily. This leads to the highest number of Q operations before reaching the target.

In bidirectional Dijkstra, edges are explored as distances increase radially outward, but this time from both source and target. The algorithm precedes in a similar way to single source Dijkstra, but this time when the frontiers overlap and the stopping condition is met, this algorithm finds the shortest path. The termination condition is when the current shortest path between start and target is smaller than the sum of the lowest priority elements on the forward and backward min priority queues. At that point, we are guaranteed to have found the shortest path. This method is able to decrease the number of Q operations than regular Dijkstra due to this particular test graph. More generally, when a graph of with n nodes and branching factor b, then $n = b^d$. It follows that $d = log_b n$ and $d \geq \delta[s, t]$. We can simplify the number of Q operations to be $b^{(log_b n)^2/2} = \sqrt{n}$. For this reason, we were able to decrease the number of Q operations.

A* takes the fewest number of operations since it biases the exploration of nodes if they are 'closer' to the target, as defined by the potential function (Euclidean distance in this case). This uses the assumption that the shortest path between s and t involves traversing edges that primarily go in the direction of the target. Since this test graph fits this assumption, A* is able to find the shortest path with fewer Q operations than either bidirectional or regular Dijkstra.