CS 31: Homework 6

Thomas Monfre

March 1, 2019

Problem 6-1

Exercise 24.3-9

First, let S denote the set of visited vertices, and let $V - S$ denote the elements in the priority queue Q. Then, in order to implement Dijkstra's algorithm such that we can compute the shortest paths from a given source vertex s in $O((V + E) \lg W)$ time, let us take advantage of the fact that the weight of each edge is bounded to $\{0, 1, ...W\}$ for some constant W. Then, there must be at most $W+2$ distinct shortest-path estimates in $V-S$ at a given time. We know this to be the case because there are $W + 1$ values an edge weight can take on, and we initialize the shortest-path estimate for all vertices $v \in V - \{s\}$ to be ∞. Then, since we extract the minimum path estimate of a vertex v in the priority queue Q on each iteration of the outer for-loop, upon removing v from Q and inserting it into S, $v.d \leq u.d$ for all $u \in Q$. Then, for some vertex x such that $(v, x) \in E$, upon calling RELAX on (v, x), it must be the case that $x.d \leq v.d+W$ since the edge weights are bounded to 0...W. Then, since all vertices in $V - S$ with a finite edge weight must have previously had their shortest-path estimates changed by the relaxation of some edge, and calling EXTRACT-MIN removed the vertex v with the smallest estimate from Q, it must be the case that $v.d \leq x.d \leq v.d + W$, or $x.d = \infty$. Then, since there are $W+1$ finite values a shortest path estimate could take on, as well as ∞, there must be $W + 2$ distinct shortest path estimates in $V - S$ at a given time.

Then, since we know there are at most $W+2$ distinct shortest-path estimates in $V-S$ at any given time, let us modify DIJKSTRA such that we implement the priority queue with a binary min-heap. Let the key of each node in the heap represent a distinct shortest-path estimate. Then, since more than one vertex in $V - S$ may have some shortest-path estimate d at a given time, each node in the heap will hold a linked list of vertices, each of which have a shortest-path estimate of d. In order to maintain this, let us hold an array of length $W + 2$ in addition to the array representing the heap, where each index in this array holds a pointer to the node in the heap corresponding to some shortest-path estimate. Let us require that each index in the array is equal to the shortest path estimate of all elements in the linked list stored in that node MOD $(W + 2)$. Note that constructing this array allows for efficient look-up of nodes representing specific path estimates in the heap. Since there are $W + 2$ distinct values a shortest path estimate could take on, but no bound on the magnitude of those values, by using the modulo operation, we allow the array to wrap around and store different values while still maintaining order.

Then, since we proved above that there are at most $W+2$ distinct shortest-path estimates in $V-S$ at a given time, using this implementation, there are at most $W + 2$ nodes in the heap at a given time. Therefore, the EXTRACT-MIN operation will take $O(\lg W)$ time, since we simply remove the first element in the linked list of the root node. Then, if the linked list contains more elements, we are done. Otherwise, we remove the entire node, move a leaf node up and bubble down, then also

remove our array pointer to the node. Similarly, we can implement the DECREASE-KEY operation in $O(\lg W)$ time as well, since we simply take a reference to a vertex, then make a constant-time lookup in the pointer array using the modulo operation for the node in the heap for the desired new key value. If such a node exists, we insert into the linked list in constant time. If not, we insert a new node into the heap, bubble up at most $\lg W$ times, then construct a new pointer in the pointer array. In this case, we have a worst-case run-time of $O(\lg W)$ for DECREASE-KEY. Note that the same process of checking for a pre-existing node and constructing one if necessary is a safe implementation of INSERT.

Then, as per CLRS Chapter 24.3, since DIJKSTRA makes $|V|$ calls to EXTRACT-MIN, $|V|$ calls to INSERT, and at most $|E|$ calls to DECREASE-KEY, and we can implement each in $O(\lg W)$ time, we then have $O(V \lg W) + O(E \lg W) = O((V + E) \lg W)$.

Problem 6-2

Exercise 24.4-7

First, let us observe that when solving a constraint graph problem in the traditional sense, $\delta(v_0, x)$ for some $x \in V$ is less than or equal to 0. We know this to be the case because the weight of each edge $(v_0, x) \in E$ for all $x \in V$ is initialized to 0. Then, on the first iteration of the outer loop in Bellman-Ford, we set the shortest-path estimate for each vertex to be at most 0 since there exists an edge (and therefore a path) from v_0 to each vertex of weight 0. Then, since there exist no directed edges going to v_0 and all shortest path estimates are set to at most 0 on the first iteration of the loop, the final shortest path estimate from v_0 to some vertex $x \in V$ will simply be the shortest path achieved between some other vertex $y \in V$ and x. We know this to be the case because the weight of each edge from v_0 is 0, so the minimum weight path from v_0 to x will simply be the minimum weight path achieved between some other vertex y and x plus the edge from v_0 to y.

Then, since we set the shortest path estimate of all vertices to be at most 0 on the first iteration of the for-loop, let us solve a system of difference constraints in a Bellman-Ford-like manner without the extra vertex v_0, by initializing the shortest path estimate of all vertices to be 0. Then, we will run Bellman-Ford in the traditional manner by picking an arbitrary vertex to be our source then relaxing each edge $|G.V| - 1$ times. Since we initialized each shortest path estimate to be 0, when relaxing each edge, the shortest path between the source and each vertex x will then be the shortest path between any other vertex y and x since initializing the shortest path estimates of each vertex to be 0 does not require a path from a particular vertex, but rather allows the Relax method to choose the smallest edge weights possible.

Problem 6-3

Exercise 25.1-10

Let us observe that in the Slow-All-Pairs-Shortest-Paths algorithm, we construct the matrix $L^{(m)}$ for $m = 0...n-1$ where $L^{(x)}$ for $0 \leq x \leq m$ holds the shortest path weights between all vertices such that each path contains at most x edges. Using this, let us note that the diagonal of $L^{(x)}$ holds the weights of cycles in the graph that have at most x edges, since a path from a vertex i to a vertex i is cyclical. Then, since each L holds the shortest path weights between all vertices and we initialize $l_{ij}^{(0)}$ to 0 if $i = j$, each value stored in the diagonal will be less than or equal to 0. Therefore, we have found a negative weight cycle of x edges or less in the graph if a value stored in the diagonal of $L^{(x)}$ is less than 0. Note that with this approach, we can ignore all values in the L matrices that are not on the diagonal, and thus ignore the possibility that having negative weight paths causes the shortest-path estimate for values off the diagonal to be incorrect.

Then, since we seek to find the number of edges in a *minimum* length negative weight cycle, let us iterate over the diagonals of each $L^{(m)}$ from $m = 2$ to $(n-1)$. As we iterate over the diagonals in these matrices for increasing m-values, the first negative value we find on a diagonal will be a minimum length negative weight cycle. We know this to be the case because all cycles in a specific matrix hold the same maximum number of edges and the first time we will come across a specific negative weight cycle in $L^{(x)}$ will be when it is of exactly length x. Note that we call Slow-All-Pairs-Shortest-Paths because it computes each $L^{(m)}$ for $m = 0...n-1$, allowing us to iterate over the diagonal in each L matrix. Since Fast-All-Pairs-Shortest-Paths skips L matrices, using this algorithm may over-estimate the minimum number of edges in a minimum length negative weight cycle.

Therefore, we can find the true minimum number of edges in a minimum length negative weight cycle by iterating over the diagonals of each L matrix in increasing values of m, then return the specific m value we are on when we first find a negative value on the diagonal. In the worst case, we must call Slow-All-Pairs-Shortest-Paths n times if the first negative weight cycle is of length $(n-1)$. Therefore, we have $O(n^4)$ for all calls to Slow-All-Pairs-Shortest-Paths plus $O(n^2)$ for iterating over at worst n values on each diagonal. This yields a final run-time of $O(n^4)$.

Problem 6-4

Exercise 25.2-4

Let us first note that by dropping the subscripts, we are replacing values in the matrix D in place. Then, in order to show this process is correct, let us simply prove that the values at each index in D that we access are correct. Specifically, let us show that $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$ will always equal $d_{ij}^{(k)} = \min(d_{ij}^{k-1}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$, the implementation used with the superscripts. Then, we have three cases to cover.

First, let us note that $d_{ij}^{(k-1)} = d_{ij}$ (the version without the superscripts) both if k is an intermediate vertex on the path p used in d_{ij} and if it is not. If k is not an intermediate vertex on the path p, then per CLRS Chapter 25.2, we know all intermediate vertices of path p are in the set $\{1, 2, ...k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, ...k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, ...k\}$. If k is an intermediate vertex on the path p, then per CLRS Chapter 25.2, we can split p into a path p_1 from i to k and a path p_2 from k to j. Then, by Lemma 24.1, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, ...k\}$. Then, because vertex k is not an intermediate vertex of path p_1, all intermediate vertices of p_1 are in the set $\{1, 2, ...k-1\}$. Therefore, p_1 is a shortest path from i to k with vertices in the set $\{1, 2, ...k-1\}$. The same can be said for p_2. Therefore, $d_{ij}^{(k-1)} = d_{ij}$ and in the operations between setting d_{ij} on the $(k-1)$th iteration and coming back to d_{ij} on the kth iteration, we will not have accessed or updated d_{ij}. So, d_{ij} in the version without the superscripts will be equal to $d_{ij}^{(k-1)}$, the value we set it to on the previous iteration of the outer loop.

Second, let us note that d_{ik} represents the minimum edge weight attained from a path p with vertices in the set $\{1, 2, ...k\}$. However, by our definition of intermediate vertices in Floyd-Warshall, k cannot be an intermediate vertex on the path. Then, per CLRS Chapter 25.2 and as shown above, we know all intermediate vertices of path p are in the set $\{1, 2, ...k-1\}$. Thus, a shortest path from vertex i to vertex k with all intermediate vertices in the set $\{1, 2, ...k-1\}$ is also a shortest path from i to k with all intermediate vertices in the set $\{1, 2, ...k\}$. This indicates $d_{ik}^{(k-1)}$ not only holds the value we require for d_{ik}, but that that value remains unchanged in D. This occurs again because the outer for-loop iterates on k, so we cannot have previously accessed d_{ik} in this kth iteration. Therefore, $d_{ik}^{(k-1)} = d_{ik}$, the value used in the version without the superscripts. Since k is not an intermediate vertex on a path with vertices in the set $\{k, k+1, ...n\}$, the same argument applies for d_{kj}.

Then, since we have shown d_{ij}, d_{ik}, and d_{kj} remain unchanged, the values they hold in the matrix D that is updated in-place will be the values they would hold in the matrix $D^{(k-1)}$ if we used the original implementation of this algorithm. Therefore, the implementation without superscripts is correct, allowing us to implement Floyd-Warshall with only $\Theta(n^2)$ space.

Problem 6-5

Exercise 25.3-4

This method of reweighting penalizes paths with more edges, even if they have a low weight, and therefore may not always produce the shortest path between two vertices. For example, suppose we have vertices a, b, c, d in a graph such that there exists an edge (a, b) of weight 1, an edge (b, c) of weight 1, an edge (a, c) of weight 3, and an edge (a, d) of weight -5. Then, the shortest path from a to c is the edges (a, b) and (b, c) since the total weight of such path is 2. This is the result that Johnson's algorithm produces. If we were to use Professor Greenstreet's approach, we would add 5 to each edge weight, yielding $w'(a, b) = 6$, $w'(b, c) = 6$, $w'(a, c) = 8$, and $w'(a, d) = 0$. Then, the shortest path from a to c is the edge (a, c) with weight 8 as opposed to the true shortest path whose weight is now 12. Therefore, Professor Greenstreet's approach does not yield the correct answer. Essentially, by adding a constant to each edge weight, shortest paths that traverse more edges are penalized with a higher total weight than those that traverse fewer paths. In this regard, Professor Greenstreet's method is not guaranteed to produce the true shortest path between two vertices in a graph.

Problem 6-6

Exercise 25.3-6. You may assume that $\infty - \infty$ is undefined and, in particular, not 0.

Suppose we have a graph G with vertices a, b, c and edges (a, b) with weight -4 and (c, b) with weight 0. Using Professor Michener's approach, we set $G' = G$ and are able to choose an arbitrary vertex as our source. Suppose we choose vertex b as the source. Then, upon running BELLMAN-FORD, we have $\delta(b, c) = \infty$ and $\delta(a, c) = \infty$ because there exist no paths from the source vertex b to vertex c and no paths from vertex a to c in G'. We also have $\delta(b, b) = 0$ because b has no outward edges.

Then, let us consider for a moment just the shortest path from vertex c to vertex b in G'. Because we computed $\delta(b, c) = \infty$ and $\delta(b, b) = 0$ above, we set $h(c) = \infty$ and $h(b) = 0$ in JOHNSON. Because of this, when computing our new weight function $\hat{w}$, we set $\hat{w}(c, b) = w(c, b) + h(c) - h(b) = 0 + \infty - 0 = \infty$. Because of this, upon running DIJKSTRA from vertex c, we find $\hat{\delta}(c, b) = \infty$ because the new edge weight $\hat{w}(c, b)$ we use is ∞ and there are no alternative paths in G' that get us from c to b.

Then, upon storing our final solution $d_{cb} = \hat{\delta}(c, b) + h(b) - h(c)$, we have $d_{cb} = \infty + 0 - \infty$. Since we are told in the problem definition that $\infty - \infty$ is undefined and, in particular, not 0, we then have an incorrect answer for the smallest path weight from vertex c to vertex b since the correct answer would be to take edge (c, b) with original weight 0. However, since we used vertex b as our source and c is not reachable from b because G is not strongly connected, we ended up computing our reweighted edge (c, b) to have a weight of ∞. This led JOHNSON to return an incorrect answer.

Now, let us show that if G is strongly connected, the results returned by JOHNSON with Professor Michener's approach will be correct. To do so, let us show the two conditions the new edge weight function $\hat{w}$ must satisfy hold. First, let us note that if G is strongly connected, then every vertex $v \in G.V$ is reachable from every other vertex. When using Professor Michener's approach, we will then set $G' = G$ and choose an arbitrary source vertex $s \in G'.V$ from which we will run BELLMAN-FORD. Since G and G' are strongly connected, there then exists a path in G' from s to all $v \in G'.V$.

Then, let us assume for a moment that all edge weights in $G.V$ are finite. Since all vertices are reachable in G from s, the minimum total path weight for each path from s to some other vertex in $G.V$ must be finite. Then, $h(v) < \infty$ for all $v \in G'.V$ because we set $h(v) = \delta(s, v)$. Therefore, since BELLMAN-FORD returns the shortest path between the source vertex and all other vertices in G and nothing about the process of setting $\hat{w}$ in JOHNSON has changed, we have $\hat{w}(p) = w(p) + h(v_0) + h(v_k)$ for any path p from v_0 to v_k by **Lemma 25.1** in CLRS Chapter 25.3. Therefore, we have our first condition that $\hat{w}$ must satisfy.

Using this, let us show that $\hat{w} \geq 0$, the second condition that $\hat{w}$ must satisfy. Since $\delta(s, v) = h(v)$, $\delta(s, u) = h(u)$, $h(v) < \infty$, $h(u) < \infty$, and the triangle inequality states that $\delta(s, v) \leq \delta(s, u) + w(u, v)$ for some $v, u \in G$, we then have $h(v) \leq h(u) + w(u, v)$. Since we set $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ and we know $h(v) < \infty$ and $h(v) \leq h(u) + w(u, v)$, it must be the case that $\hat{w}(u, v) \geq 0$ because we know $w(u, v) + h(u)$ is greater than or equal to $h(v)$ and both are finite. So, when subtracting $h(v)$ from $w(u, v) + h(u)$, we will get a non-negative number.

Therefore, since $\hat{w}(u, v) \geq 0$ and all edge weights in G being finite indicates all edge weights in G' are finite, we then know the two conditions required of $\hat{w}$ are satisfied and can conclude that JOHNSON returns the correct answer using Professor Michener's modifications. Let us note that we assumed all edge weights in $G.V$ are finite. If we remove this assumption, our proof will still hold since all vertices will be reachable from the arbitrarily chosen source vertex because G is connected. Therefore, if the minimum weight of a path p from u to v in G is infinite, $\hat{w}(u, v)$ will be infinite causing d_{uv} to be infinite. Let us note that this answer is correct since $\delta(u, v) = \infty$ in this case.

Therefore, if G is strongly connected, the results returned by JOHNSON with Professor Michener's modifications will be correct.

Problem 6-7

Exercise 26.1-4

First let us note that a flow on the graph G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the capacity constraint and the flow conservation property. In order to prove that the flows in a network form a convex set, let us show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1-\alpha)f_2$ for all $0 \leq \alpha \leq 1$. To do so, let us assume f_1 and f_2 are flows. We will then use this assumption to show $\alpha f_1 + (1-\alpha)f_2$ is a flow.

First, since $0 \leq \alpha \leq 1$, it is clearly true that f_1 being a function mapping vertex pairs to reals implies αf_1 is a function mapping vertex pairs to reals. The same can be said for $(1-\alpha)f_2$. Therefore, since a real number plus a real number is a real number, $\alpha f_1 + (1-\alpha)f_2$ is a function mapping vertex pairs to reals. Then, let us show $\alpha f_1 + (1-\alpha)f_2$ abides by the capacity constraint and the flow conservation property.

We know $\alpha f_1 + (1-\alpha)f_2$ abides by the capacity constraint because we are told $(\alpha f)(u,v) = \alpha \cdot f(u,v)$. Therefore, we seek to show $0 \leq \alpha f_1(u,v) + (1-\alpha)f_2(u,v) \leq c(u,v)$ for some $u, v \in V$. Since we have assumed f_1 and f_2 are flows, we then know each satisfy the capacity constraint. Therefore, $0 \leq f_1(u,v) \leq c(u,v)$ and likewise for f_2. Then, in order to show the bounds of 0 and $c(u,v)$ are not violated by this summation, the worst case possibility is that $f_1(u,v)$ and $f_2(u,v)$ are their minimum or maximum possible values. The minimum value of a flow is 0 and the maximum value of a flow is $c(u,v)$.

We can see that if $f_1(u,v) = f_2(u,v) = 0$, we have the inequality $0 \leq 0 + 0 \leq c(u,v)$ which clearly satisfies the capacity constraint. For the maximum value worst case, we can see that if $f_1(u,v) = f_2(u,v) = c(u,v)$, we must show $0 \leq \alpha c(u,v) + (1-\alpha)c(u,v) \leq c(u,v)$. Since $1 - \alpha + \alpha = 1$, we then have $\alpha c(u,v) + (1-\alpha)c(u,v) = c(u,v)$. Therefore, we have $0 \leq c(u,v) \leq c(u,v)$ which clearly satisfies the capacity constraint. Therefore, since we have shown $\alpha f_1 + (1-\alpha)f_2$ abides by the capacity constraint in each worse case, values in between 0 and $c(u,v)$ for $f_1(u,v)$ and $f_2(u,v)$ will also abide by this constraint. Hence, we know $\alpha f_1 + (1-\alpha)f_2$ abides by the capacity constraint for all $u, v \in V$.

Then, we can show $\alpha f_1 + (1-\alpha)f_2$ abides by the flow conservation property because we know f_1 and f_2 are both flows and $0 \leq \alpha \leq 1$. Then, we seek to show: $\sum_{v \in V} \alpha f_1(u,v) + (1-\alpha)f_2(u,v) = \sum_{v \in V} \alpha f_1(v,u) + (1-\alpha)f_2(v,u)$

Since f_1 and f_2 are flows, we then have $\sum_{v \in V} f_1(u,v) = \sum_{v \in V} f_1(v,u)$ and likewise for f_2. So, by subtracting the equal parts of each equation from each other on either side, we get $\alpha \sum_{v \in V} f_1(u,v) - \alpha \sum_{v \in V} f_1(v,u) = 0$, and $(1-\alpha)\sum_{v \in V} f_2(u,v) - (1-\alpha)\sum_{v \in V} f_2(v,u) = 0$. Then, since $0 = 0$, we have that $\sum_{v \in V} \alpha f_1(u,v) + (1-\alpha)f_2(u,v) = \sum_{v \in V} \alpha f_1(v,u) + (1-\alpha)f_2(v,u)$. Hence, we know $\alpha f_1 + (1-\alpha)f_2$ abides by the flow conservation property.

Therefore, since we have shown $\alpha f_1 + (1-\alpha)f_2$ is a function from $V \times V \to \mathbb{R}$ that abides by the capacity constraint and the flow conservation constraint, when f_1 and f_2 are flows, we have that $\alpha f_1 + (1-\alpha)f_2$ is a flow for all $0 \leq \alpha \leq 1$.

Problem 6-8

Exercise 26.2-12

Let us assume we are given a flow network G with edges entering the source s and a flow f in which some edge (v, s) entering the source has $f(v, s) = 1$. Using this, let us prove there must exist some other flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$.

First, let us notice there must exist a cycle on the source vertex s. We know this to be the case because all flow in a flow network originates from the source and there is positive flow entering s from v. Since the flow conservation constraint states the amount of flow entering a vertex v must equal the amount of flow exiting v, we then know at least one unit of flow must enter v. Therefore, there must exist a path from s to v in G since that flow must have originated from the source, implying there must exist a way for that flow to get from s to v. Then, since there exists a path from s to v in G and we know there is an edge from v to s in G, there must exist a cycle on the source vertex s. Note that this cycle must also be of positive flow because we know $f(v, s) = 1$ so each edge (a, b) on the cycle must have a flow $f(a, b) \geq 1$. Let us refer to this cycle as c.

Using this, let us define a new flow f' such that the amount of flow traversing each edge on c is decremented by exactly one unit of flow. Specifically,

$$f'(a,b) = \begin{cases} f(a,b) - 1 & \text{if } (a,b) \text{ is on the cycle } c \\ f(a,b) & \text{otherwise} \end{cases}$$

Then, since (v, s) is on the cycle c and $f(v, s) = 1$, we then have $f'(v, s) = 0$. Furthermore, since c is a positive flow cycle, $f(a, b) \geq 0$ for all (a, b) on the cycle c, so we do not have a negative flow cycle. Since the flow on (v, s) was decremented by 1, we then have one less unit of flow entering the source. Then, since $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$, that is the flow out the source minus the flow into the source, and $f'(v, s) = f(v, s) - 1$, we then must have one less unit of flow exiting the source. Then, $|f'| = (\sum_{v \in V} f(s, v) - 1) - (\sum_{v \in V} f(v, s) - 1) = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) = |f|$. Therefore, there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$ since there must exist a positive flow cycle on s and decrementing the flow on each edge in the cycle by 1 yields the same total flow.

Then, we can compute f' given f in $O(E)$ time assuming all edge capacities are integers with the following. We will set f' to f by iterating over each edge in f' and setting the weight (capacity) of each edge in f' to its corresponding value in f. Then, we will find a positive flow cycle on s such that (v, s) is on the cycle by finding a simple path from s to v using Depth-First-Search. Specifically, we will call DFS from the source vertex s. Note that the first time we come across v in DFS, we will have found a simple path p from s to v. Then, we will iterate over each edge in p and decrement the weight of each edge by 1. We will also set $f'(v, s) = 1$. Given our definition of f' above, we will then be finished. Then, we have $O(E)$ for copying all edges in f to f', $O(V + E)$ for DFS, and $O(E)$ for decrementing the edge weights on p. Since $|E| \geq |V| - 1$ (as noted in CLRS 26.1), we then have $O(E)$ for this algorithm.

Who I Worked With

Problem 6-1: Worked with Kunal Rathi and Emma Rafkin.

Problem 6-2: Worked with Kunal Rathi.

Problem 6-3: Worked with Kunal Rathi.

Problem 6-6: Worked with Emma Rafkin.

Problem 6-8: Worked with Emma Rafkin and Kunal Rathi.