CSCI-GA.1170-001/002 Fundamental Algorithms December 7, 2014

Solutions to Problem 1 of Homework 11 (15 points)

Name: Keeyon Ebrahimi *Due: Wednesday, December 5*

Dijkstra's algorithm solves the single-source shortest-path problem on a weighted directed graph $G = (V, E)$, when all edge weights are non-negative. Suppose we wish to modify the algorithm so that it works on graphs which has negative weight edges as long as there is no *negative cycle*. Consider the following modified Dijkstra's algorithm.

```
ModifiedDijkstra(G, w, s)
1.  Initialize-Single-Source(G, s)
2.  S = ∅
3.  Q = G.V
4.  While Q ≠ ∅ Do
5.      u = Extract-Min(Q)
6.      S = S ∪ {u}
7.      For each vertex v ∈ G.Adj[u]
8.          If v.d > u.d + w(u, v)
9.              v.d = u.d + w(u, v)
10.             v.π = u
11.             If v ∈ Q, Then Decrease-Key(Q, v, v.d)
12.             Else Insert(Q, v) and S = S \ {v}.
```

(a) (3 points) Note that the only step where the above algorithm differs from the original Dijkstra algorithm is in Step 12. Give an example with the smallest possible number of vertices to show that if we remove step 12 from ModifiedDijkstra, then it does not solve the single-source shortest-path problem if the edge weights may be negative, even if there are no negative weight cycle.

Solution:
Here is an example demonstrating when Dijkstra's doesn't work with negative edges.

Lets say we have 3 nodes, A, B, C, with the source being A

$A \to B$ with weight 2.
$A \to C$ with weight 3.
$C \to B$ with weight -10.

Bellman-Ford with A source: B distance $= -7$

(i) A *distance* : 0
A is source

(*ii*) C *distance* : 3
$A \to C$

(*iii*) B *distance* : -7
$A \to C \to B$

Dijkstra's with A source: B distance $= 2$

1. With this graph, the priority queue will dequeue node B first. Set it's distance to 2. B has no outgoing edges as well, so it will not enqueue up anything new.
2. We will then dequeue C, and give it a correct distance of 3. We then check all of C's outgoing edges, and we will find a path to B with a total of -7. B is no longer in the priority queue though, it will not have it's key decreased. Without line 12 it will not be placed back in the queue as well, making B keeps its distance as 2.

As we can see, Dijkstra's algorithm will give us the incorrect distance for node B when we have a negative edge weight. □

(b) (3 points) Assume that the input graph G has no negative weight cycle, although there may be some edges with negative weight. Show that the total number of times the value of $v.d$ changes in the above algorithm is finite. Hence argue that the algorithm terminates in a finite number of steps.

Solution:
There are a few points we need to prove this.

1. In order to change $v.d$, we must decrease the cost of a path to $v.d$. This if from line 8 of the algorithm

2. The total amount of paths that can lead to $v.d$ is E. Path's are constructed of edges, making the maximum amount of paths be E.

3. This means that we must visit the same edges infinite times if we want to infinitely change the value of $v.d$.

4. The only time we can visit the same more than once is if a cycle exists.

5. Cycles can only be negative cycles or positive cycles.

6. Positive cycles mean they add a positive value to the total cost of the path. We only change $v.d$ if the cost to the path to $v.d$ is decremented, meaning we will not change $v.d$ in a positive cycle, meaning that positive cycles would still lead to a finite amount of $v.d$ changes.

7. The only other option is a negative cycle. A negative cycle can repeat multiple times, and also by definition, a negative cycle will decrease the total cost. This means that a negative cycle will infinitely decrease the total path cost value, which in tern could potentially infinitely decrease the value $v.d$.

Summary: We either have a cycle or we don't. If there is no cycle, we cannot infinitely change $v.d$, as explained in proof line 5. If there is a cycle, we can only have a positive and negative cycle. Positive cycles cannot cause infinite amount of $v.d$ changes as shown in proof line 6. Negative cycles can change $v.d$ infinite amount of times as shown in proof line 7. We have exhausted all possible outcome, and have deduced that the only potential infinite changes to $v.d$ is only when there are negative cycles. □

(c) (4 points) Notice that for all $v \in V$, the value of $v.d$ is at least the shortest distance from s to v in an execution of the above algorithm. Argue that when the algorithm terminates, for all $v \in V$, the value of $v.d$ is equal to the shortest distance from s to v. Hence conclude the above algorithm correctly solves the single-source shortest path problem even for graphs with negative weight, as long as there is no negative weight cycle. Explicitly state where you need step 12 of the above algorithm in your proof.

Solution:
We know that Dijkstra's algorithm works with all positive edges, because the distance is always increasing as we traverse. Dijkstra's always evaluates the edge creating the next minimum total path. Because all edges are positive, no path ever decreases. Because of this, we know that grabbing the current edge creating the minimum distance will be the next best candidate for a shortest path. We also know that a path to v evaluated at an earlier time will always have a smaller $v.d$, because we are always increasing in total d, and the earlier it is evaluated, the shorter the total cost will be.

Knowing that non negative edges will create an always increasing distance, we know that a path to v evaluated at an earlier time will always have a smaller value than the distance to v calculated at a later time. This is what makes Dijkstra's works for positive edges.

With negative edges, the distance can decrease. This means that we cannot assume that a path discovered at an earlier time will be shorter than a path discovered at a later time. As shown in *part a*, the problem with Dijkstra's is when a shorter path that takes later to discover exists, it will never be evaluated because the previous path to v would take v out of our queue, never allowing this later discovered yet shorter path to v.

This is where *Line* 12 comes into play. If we have those cases where a shorter path to v is discovered later, *Line* 12 will allow us to re-evaluate v, taking care of the skipping over issue we discovered in *part a*, which then fixes the issue with Dijkstra's and negative edges

□

(d) (5 points) Consider an example of a graph G with vertices $(s, v_1, \ldots, v_n)$ and edge weights $w(s, v_i) = 0$, and $w(v_i, v_j) = -2^{-i}$ for all $1 \leq i < j \leq n$. By finding an appropriate recurrence relation, show that ModifiedDijkstra takes $\Omega(2^n)$ time in the worst case, when executed on (G, s, w).

Solution:
We are applying the algorithm one item at a time, giving us a $T(n-1)$ in our recurrence relation. In the worst case analysis for the given G, we know for every iteration, we will be placing something back on the queue. This will give us a recurrence relation of

$$T(n) = 2 * T(n-1) + O(1)$$

This gives us the running time of $\theta(2^n)$

□

CSCI-GA.1170-001/002 Fundamental Algorithms December 7, 2014

Solutions to Problem 2 of Homework 11 (14 Points)

Name: Keeyon Ebrahimi *Due: Wednesday, December 5*

John, who lives in a node s of a weighted undirected graph G (with non-negative weights), is getting late and has to reach the venue of his high school final exam at node h as soon as possible. However, he has to buy some pencils on his way to the examination hall. He can get pencils at any stationary, and the stationary shops form a subset of the vertices $B \subset V$. Thus, starting at s, he must go to some node $b \in B$ of his choice, and then head from b to h using the shortest total route possible (assume he wastes no time in the stationary). Help John to reach the examination hall as soon as possible, by solving the following sub-problems...

(a) (2 points) Compute the shortest distance from s to all stationary shops $b \in B$.

Solution:
Because we know that there are no negative weight edges, we can just run Dijkstra's algorithm. We can also have a set B where we initially store all b. Whenever we visit and find the distance of a b, we remove b from set B. If set B is ever empty, we can stop Dijkstra's algorithm because we know that we have found all the distances to all b. □

(b) (4 points) Compute the shortest distance from every stationary shop $b \in B$ to h. Can one simply add a new "fake" source s' connected to all stationary shops with zero-weight edges and run Dijkstra from s'?

Solution:

In order to find the shortest distance from **every** stationary shop, we need to run Dijkstra's algorithm multiple times. For each b, we will need to run Dijkstra's algorithm using that b as the source. This means that if B contains 10 b's, then we need to run Dijkstra's algorithm 10 times, with each b being used as a source.

We cannot use the fake source method because that will just find which b has the shortest distance to h. This will only find one shortest distance though, and it will not find **every** shortest distance. □

(c) (2 points) Combine parts (a) and (b) to solve the full problem.

Solution:
We initially run Dijkstra's algorithm using s as our source. Once we have found the distance to every b, we then run Dijkstra's on every b, finding each b's distance to h. For each b, we then add the distance to b with the distance from b to h, and whichever b gives us the smallest sum, we use this path. □

(d) (6 points) Your solution in part (c) used two calls to the Dijkstra's algorithm (one in part (a) and one in part (b)). Define a new graph G' on at most $2n$ vertices and at most $2m + n$ edges (and "appropriate" weights on these edges), so that the original problem can be solved using a *single* Dijkstra call on G'.

Solution: The case where this doesn't work is when we only find a single b's shortest distance to h. We want to make sure that we go through every b. The way we do this is we modify the graph at each b. At each b, we put a 0 weight edge to all other b's. That ensues that when we hit any b, the path will then no matter what travel to all other b's. This will give us our correct algorithm checking all paths that go through each b. □

CSCI-GA.1170-001/002 Fundamental Algorithms December 7, 2014

Solutions to Problem 3 of Homework 11 (12 points)

Name: Keeyon Ebrahimi *Due: Wednesday, December 5*

You are given a directed graph $G = (V, E)$ representing some financial choices. Each edge $(u, v) \in E$ has a weight $w(u, v)$, where $w(u, v) > 0$ represents a cost, and $w(u, v) < 0$ represents a profit. Your initial portfolio is a vertex $s \in V$, and at each step you are allowed to go from your current node $u \in V$ to a neighboring node $v \in Adj(u)$, incurring a cost $w(u, v)$ if $w(u, v) > 0$, or a profit $-w(u, v)$ otherwise.

(a) (4 points) We say that a vertex s is *super-lucky* if s itself is part of a cycle C of negative weight, so that starting from s one can repeatedly come back to s with some profit. Using the "matrix multiplication" approach, design $O(n^3 \log n)$ algorithm to find all super-lucky vertices.

Solution:
For set up. Imagine 3 Matrices A, B, and C.

1. A_{ij} is total weight to get from vertex A_{ij}
2. B_{ij} is the edge weight of the edge that connects Vertex i and Vertex j.
3. C_{ij} is the computed shortest path from Vertex i to Vertex j.
4. D^x computed shortest paths matrix that contains the shortest paths with a path that has x total edges.

We know that the dynamic solution to this is

$$C_{ij} = Min_{for\ all\ k}(A_{ik} + B_{kj})$$

This will be the same as matrix multiplication, which is

$$C_{ij} = \sum_k (A_{ik} * B_{kj})$$

except we have to replace summation with min and the multiplication with an addition. We will call this math replacement $**$

Now we can see that this gives us the recurrence equation of

$$D^n = D^{n-1} * *A$$

This means that while using that math replacement, we know that

$$D^n = A^n$$

Now we must compute A^n. For this we can use Exponentiation by squaring.

This is computed by:
when n is odd, $x^n = x(x)^{\frac{n-1}{2}}$
When n is even, $x^n = (x^2)^{\frac{n}{2}}$

If we compute A^n using this Exponentiation by squaring method, we will now be running A^2 $log(n)$ times.

Once we do all this, we will have our solution matrix C all filled out. Now we must find which vertexes have negative paths to themselves, we just need to check the diagonal of matrix C, and all negative values will inform us that we have a negative cycle starting and ending at that vertex.

As for the running time, computing matrix multiplication takes $O(n^3)$ running time, and we are doing this $log(n)$ times meaning that our total running time will be $O(n^3\ log\ n)$ □

(b) (4 points) Say that s is *lucky* if there exists a way to eventually make unbounded profit starting from s (but not necessarily coming back to s infinitely many times as with super-lucky vertices). Assume also you know all super-lucky vertices. Give the fastest algorithm you can for finding lucky vertices given super-lucky vertices. State its running time as a function of m and n.
(**Hint**: Make sure you use super-lucky vertices instead of computing from scratch.)

Solution:
To summarize, we will be running a BFS, once we reach a super lucky vertex, we will traverse up the path that got there and add them to a Lucky Set.

1. Set L is initially an empty set that we will fill with Lucky Vertexes
2. Set SL is a set of all the super lucky vertexes that are given.
3. Set Q will be our queue that we will run our BFS on have all vertexes excluding all of the vertexes in SL.

We will run a BFS on each node in Q, marking them as we go. The only difference is whenever we detect a super lucky vertex, we trace the path back to the s and insert these vertexes back into our L set.

This will have the same running time as a BFS. The only difference is we can itterate back up now when we find lucky vertexes meaning that the total running time will be

$$O(2V + E)$$

Which then reduces down to

$$O(V + E)$$

□

(c) (4 points) Assume s is not lucky (or super-lucky). Design the best finite strategy to make as much profit starting from s as possible. State the running time of your algorithm. (**Hint**: Think Bellman-Ford.)

Solution:

s will not be lucky or super lucky. s not being lucky or super lucky means that there is no path from s that will lead to a negative cycle. Now we know that G has negative weights and no negative cycles, we know we can run $Bellman - Ford$.

As we are running $Bellman - Ford$ algorithm, we also will store a variable storing location and lowest value, which we will label as Z. Whenever our $Bellman - Ford$ does a relaxation, we will compare our lowest value with the current relaxed value and store the minimum in Z. After $Bellman - Ford$, Z will store the minimum value within our matrix, which gives us the end route spot for the maximum profit path.

We can then look at this point and traverse up the tree to find the path to this maximum profit value.

As for the running time, we first run $Bellman - Ford$ with our Z variable modification, which will take $O(|V||E|)$ time. After this we, traverse up Z's path to find the total path, which takes another $O(V + E)$ worst case.

This gives us a running time of

$$O(|V||E|| + V + E)$$

Which really equals

$$O(|V||E||)$$

□

CSCI-GA.1170-001/002 Fundamental Algorithms December 7, 2014

Solutions to Problem 4 of Homework 11 (22 points)

Name: Keeyon Ebrahimi *Due: Wednesday, December 5*

You are given a map $G = (V, E)$ with cities V connected by roads E. Each road (edge) is labeled with a weight which is a real number. You are located in city $s \in V$. You are also given an array A of boolean values that tells you if there is a toll on that road. More precisely, for road $e \in E$ we have $A[e] = 1$ if and only if there is a toll on e. Being the low budget traveler that you are, your budget allows for at most one such toll to be payed for any given trip (path).

Let $d(s, t)$ denote the minimum possible sum of weights of a path from s to t for any $s, t \in V$. If there is no path from s to t, then $d(s, t) = \infty$, and if there is a path from s to t that contains a negative cycle, then $d(s, t) = -\infty$ (since one can cycle along the path an arbitrary number of times).

Similarly, let $c(s, t)$ denote the minimum possible sum of weights of a path from s to t that passes through at most 1 toll.

(a) (10 points) Construct a graph $G' = (V', E')$ and mappings $f : V \mapsto V'$, $g : V \mapsto V'$ such that $|V'| = 2|V|$, $|E'| \leq 2|E| + |V|$ and for any $s, t \in V$, $c(s, t) = d(f(s), g(t))$. Namely, you reduce the "constrained" problem on G to "unconstrained" problem in G'. Remember to consider the case when $c(s, t)$ is $-\infty$, and prove the correctness of your solution.

Solution: To reconstruct, we run the beginning parts of Johnson's Algorithm. We will be adding a node Q with 0 weight edges to all other Vertexes. Then we will run a Bellman-Ford on G. We will then update all the edges by $w(u, v) + h(u) - h(v)$, just like in Johnson's Algorithm. At this point we will also remove vertex Q and all of it's outing edges that we added. At this point we now have a graph that we can work with that doesn't have the $-\infty$'s anymore. □

(b) (3 points) Give an algorithm that takes as input s and finds a shortest path with at most one toll road from s to all cities in V. Analyze the running time of your algorithm.

Solution:
This will be a modified version of Johnsons Algorithm. We will create the 0 weight edges from an imaginary vertex q to all vertexes, run Bellman-Ford, and adjust the weight of each node to be $w(u, v) + h(u) - h(v)$. We now take out node q and all of it's outgoing edges.

We can now run Dijkstra's algorithm on this graph because we no longer have negative weights. We can now run Dijkstra's from the S and find the shortest distance to each vector that has a toll, marking vertexes on the way. Then we run Dijkstra's from each of these toll vertexes on the graph that marked vertexes when finding the distance to each toll, which includes the tolls as marked.

Now we know the weight from source to all the different tolls. We also know the distance

from all the tolls to each vertex that wasn't marked on the way to the tolls, including other tolls. This stipulation ensures that we wont visit a node toll node more than once.

At most, the running time is $O(|V|^2\ log(|V|) + |V||E|)$
This is because we are running a Bellman-Ford which takes time $O(|V||E|)$ and then up to v Dijkstra's on each toll, which is what gives us the $O(|V|^2\ log(|V|)$. Put together, we get $O(|V|^2\ log(|V|) + |V||E|)$ □

(c) (3 points) Now assume your buddy Billybob who works at a major airlines company has given you a free plane ticket to any city in V, meaning you can start your trip at any node. As before, once you start your road trip, you are still refusing to pay more then a single toll on any such trip. To help plan the trip, your job is to give an algorithm to find a shortest path with at most one toll road between *all pairs* of cities in V. Analyze the running time of your algorithm. (**Hint**: Remember Johnson.)

Solution:
For this algorithm we will do something pretty simple. We will just be running a Johnson's algorithm. During the stage where we are running the different Dijkstra's algorithm, we will constantly be checking our amount of tolls that we have passed. If we have passed 0 tolls and we pass 1, we do nothing special, but we increment our total tolls passed value by 1. If our tolls visited value is 1 and we visit a node that is a toll, we will do something tricky, like add infinity to potential relaxation phase so that the path through the second toll will never be chosen.

This will give us the shortest path from all vertexes and we will skip over all possible routes that have 2 tolls.
This is a modified version of Johnson's algorithm, that doesn't add anything that will change the running time. All we are doing is modifying the potential value at relaxation, which is constant time.

This gives us a total running time of $O(|V|^2\ log(|V|) + |V||E|)$ □

(d) (6 points) Here you will solve the problem in part (c) directly on graph G, without constructing the helper graph G'. Let $W = \{w(i,j)\}$ be the original edge weight matrix and $W' = \{w'(i,j)\}$ be the same edge matrix except we replace $w'(i,j) = \infty$ if $A(i,j) = 1$ (i.e., never use toll roads in W'). For simplicity, assume W' is pre-computed for you. For $0 \leq k \leq n$, let

- D^k be the matrix of all shortest distances w.r.t. W' which only use nodes $\leq k$ as intermediate nodes.
- C^k be the matrix or all shortest distances w.r.t. W which only use nodes $\leq k$ as intermediate nodes, but *also use at most one toll.*

Fill in the blanks below to directly modify the Floyd-Warshall algorithm to compute the correct answer for problem (c) in time $O(n^3)$. Argue the correctness of your algorithm.

FLOYD-WARSHALL(W, W')

$n = W.rows$
$D^0 = ...$
$C^0 = ...$
For $k = 1$ **to** n **Do**
$C^k = (c^k_{i,j})$ be a new $n \times n$ matrix
$D^k = (d^k_{i,j})$ be a new $n \times n$ matrix
For $i = 1$ **to** n **Do**
For $j = 1$ **to** n **Do**
$d^k_{i,j} = \min(.....)$
$c^k_{i,j} = \min(.....)$
Return .....

Solution:

FLOYD-WARSHALL SOLUTION(W, W')

$n = W.rows$
$D^0 = ...$
$C^0 = ...$
For $k = 1$ **to** n **Do**
$C^k = (c^k_{i,j})$ be a new $n \times n$ matrix
$D^k = (d^k_{i,j})$ be a new $n \times n$ matrix
For $i = 1$ **to** n **Do**
For $j = 1$ **to** n **Do**
$d^k_{i,j} = \min\left(W'_{i,j}, \quad W'_{i,k} + W'_{k,j}\right)$
$c^k_{i,j} = \min(d_{i,j}, \quad d_{i,k-1} + W_{k,k} + d_{k+1,j})$
Return C

What is going to happen here is D is only updating on $'W$, meaning that it will only hold paths that do not include a toll.

When we update C, we will check if we have a smaller path going all the way to the destination using no tolls, the $d_{i,j}$, and we will compare that with no tolls from i to $k-1$ and also from $k+1$ to j. We only check graph W at node k, so we let k be our potential toll. This will ensue that we will always have 0 or 1. □