CSCI-GA.1170-001/002 Fundamental Algorithms December 6, 2014

Solutions to Problem 1 of Homework 11 (15 points)

Name: Sahil Goel *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.  $S = \emptyset$
3.  $Q = G.V$
4.  While $Q \neq \emptyset$ Do
5.      $u$ = Extract-Min($Q$)
6.      $S = S \cup \{u\}$
7.      For each vertex $v \in G.Adj[u]$
8.          If $v.d > u.d + w(u, v)$
9.              $v.d = u.d + w(u, v)$
10.             $v.\pi = u$
11.             If $v \in Q$, Then Decrease-Key($Q, v, v.d$)
12.             Else Insert($Q, v$) and $S = S \setminus \{v\}$.
```

(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: Suppose there are three vertices A,B and C with the following edge weights
A-B : 3
A-C : 10
B-C : 9
C-B : -8
A is the source
Then initially
d(A) = 0, d(B)= infinity and d(c) = infinity
Q = { (0,A), $(\infty, B), (\infty, C)\}$

After first extract-min
S={(0,A)}
Q = {(3,B), (10,C) }

After second extract-min
S={(0,A),(3,B)}
Q = {(10,C)}

After third extract-min
S={(0,A),(3,B),(10,C)}
Q={}

So the minimum distance we get from A to B is 3 but we can see if we follow the path from A-C and then C-B, the distance will be (10+(-8)) = 2 which is lower than the one we get by applying original Dijkstra algorithm. Hence original Dijkstra doesn't solve the problem if graph have negative edges

□

(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: For any node v, the value v.d is changed whenever we find an edge u such that $u.d + w(u, v) < v.d$. Since we know that there are finite number of nodes and no negative cycles, for a node its value will be changed if we found a node amongst its adjacent (worst case : rest (n-1) nodes) which satisfies this condition. Its value will be change max n-1 times, hence the final program will terminate

□

(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: Suppose that when the algorithm terminated, there is a node v whose $v.d > distance(s, v)$
Suppose u is the predecessor of v in the shortest path. Assuming that u.d is correct. When we would have added u to the Solution set S, we would have considered all of its adjacent nodes. If we had found $v.d > u.d + w(u, v)$ we would have relaxed the edge (u,v). Observe that u can't already be in S. Whenever we find a node x.d smaller than previous, we check if it was already in S or not. If it is in S, we remove it from S so as to reflect all the changes to its adjacent nodes. Hence this is a contradiction, therefore for every node $v.d$ will be minimum distance from source.
Base case: When distance(s,s)=0 and $distance(s, G.V - \{s\}) = \infty$
This is correct as when no other node is discovered and their distance = inifnity from the

source.
12th step is used so that, whenever an edge is relaxed, the node is removed from the solution set and added again to queue so that all its adjacent nodes should again be modified. In the proof above, it makes sure that if u is the predecessor of v in shortest path, the shortest distance to u will be found before which will eventually help in finding the shortest distance to v

□

(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: Initially all the nodes are at infinite distance($v.d = \infty$) and s.d=0. Then we start with s, extract it and traverse all its adjacent nodes and replace each of the adjacent node v.d by s.d + w(s,v) = 0

For the worst case, we will extract v_k where k is maximum possible. Initially we will extract v_n

Now we will again update all the nodes(u) by distance u.d = $v_n.d + w(v_n, u)$ since u.d was initially zero and now it will be negative since w(v_n,u) is negative.

Now $v_n.d$=0 and
$v1.d = v2.d = v3.d = v4.d = \ldots = v_{n-1}.d = 0$ since there is no outgoing edge from v_n
Now we will again extract min suppose v_{n-1} and update every node distance in its adjacency list if $u.d > v_{n-1}.d + w(v_{n-1}, u)$
Since now $v_{n-1}.d = 0$ and $w(v_{n-1}, u) = -2^{-(n-1)} = -1/2^{n-1}$,
$v_{n-1}.d + w(v_{n-1}.u) = -1/2^{n-1}$
Now value of v_n will be updated and again it will be added to the queue.
Similarly going in this manner, suppose we extract node v_k, values of all the nodes after v_k i.e. $\{v_{k+1}, v_{k+2}, \ldots, v_n\}$ will be updated and added to queue which finally will repeat again
Recurrence equation at time k will be $T(k) = T(1) + T(2) + T(3) + \ldots + T(k-1)$ since we will repeat the same procedure starting from last node, then last two, then last three and so on
This procedure will go on till the end
Finally recurrence equation $T(n) = T(1) + T(2) + \ldots + T(n-1)$
Now T(1) = 1
T(2) = T(1) = 1
T(3) = T(2) + T(2) = 2
T(4) = T(3) + T(2) + T(1) = 4
T(5) = T(4) + T(3) + T(2) + T(1) = 8
Similarly $T(n) = 1 + (1 + 2 + 4 + 8 + \ldots + 2^{n-2})$
$T(n) = 1 + 2^{n-2} - 1 = 2^{n-2} = \Omega(2^n)$

□

CSCI-GA.1170-001/002 Fundamental Algorithms December 6, 2014

Solutions to Problem 2 of Homework 11 (14 Points)

Name: Sahil Goel *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: The shortest distance to all stationary shops can be calculated by applying Dijkstra's algorithm by assigning source to s and calculating minimum distance of all other nodes from s. Initially s.d = 0 and for all other nodes v, v.d=infinity. After the algorithm is terminates, the final values in b.d will be the shortest distance of that b node from s. Here Dijkstra returns the shortest distance from s to every stationary shop. After the algorithm is terminated arr[i] holds shortest distance from s to node with index i.

```
main()
{
int arr1[] = Dijkstra(G,s)
}
```

□

(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: We can apply Dijkstra's algorithm on it by using destination(school) as the source. It will give the shortest distance from school to all the stationary shops.

```
Algorithm
{
arr2=Dijkstra(G,school)
return arr2
}
```

arr2[i] will store the minimum distance from that node i to school node

□

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

Solution: Now we can use the combination of solutions (a) and (b), to find the shortest distance. We will run dijkstra as in part (a) as well as in part (b) and then use the values of arr1 and arr2 to find the minimum distance from s to h via any node b in B

We have to find the minimum such that arr1[i] + arr2[i] is minimum and i ∈ B

```
int min=infinity
int finalindex; for all $b \in B$
{
int index=indexof(b);
if($minimum > arr1[b] + arr2[b]$)
{
minimum=arr1[b] + arr2[b]
finalindex=b
}
}
```

The final shortest path will be from s to node[finalindex] to h

□

(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: Initially suppose graph G has nodes $\{s, v_1, v_2, \ldots, v_n, h\}$
We will create a copy of graph G' with nodes
$\{s', v_1', v_2', \ldots, v_n', h'\}$

Now for every b∈B we will create a zero node edge from every b node in G to its corresponding b' node in G'.
Now there are 2n vertices and at max 2m+n edges since there will be m edges in G, m in G' and there are maximum n stationary nodes so n more edges from b to b'.

```
Algorithm
{
Create G' copy of G
for every $v_i \in B$
create edge $v_i$ to $v_i'$
end for
Dijkstra(G,s)
}
```

□

CSCI-GA.1170-001/002 Fundamental Algorithms December 6, 2014

Solutions to Problem 3 of Homework 11 (12 points)

Name: Sahil Goel *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: Using matrix multiplication and repeated multiplication we can calculate minimum distance from every node i to every other node j. After we have applied the matrix multiplication algorithm, we will have a 2D matrix c[][] where c[i][j] gives the shortest distance from node i to node j.
After we have the resulting matrix we can traverse the diagonal element, if any of the element is negative then there is a negative cycle since there is a path from node i to i with negative cost, which means there is negative cycle in the graph. It will take O(n) time. Total time will be $O(n^3 \log n + n) = O(n^3 \log n)$

```
Algorithm
{
c[ ][ ]=MatrixMultiplication(G)
S={}
for i=1 : n
if($c[i][i] < 0$)
{
$S = S \cup i$
}
end for
}
```

At the end S will have all the super lucky nodes

□

(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: Any node is lucky if there exists a path from that node to super lucky node since then we can go to super lucky node from that node and make infinite profit
For this we can reverse the graph edges and start from any super lucky node and traverse all the nodes which are reachable from it. Any node which is now reachable is lucky since in actual graph there will e a path from that node to the super lucky node

```
Algorithm
{
1. Reverse the edges
2. Start from any super lucky node and do DFS traversal. Whichever nodes comes in the path, add those nodes to lucky nodes (L).
3. If there are still super lucky nodes left which are not yet visited, start DFS from this step and repeat from step 2
4. Reverse the edges again and to get the original graph G 4. Return L
}
```

Since each node and each edge is traversed maximum 3 times(2 reversals and 1 DFS), total time complexity if O(n+m)

□

(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: Since s is not lucky we know that there doesn't exist any path from s to any lucky or super lucky node otherwise it would have been lucky too. Therefore there doesn't exist any negative edge cycle which can come in the path from s therefore we can apply bellman ford algorithm from s and calculate the shortest distance node from s. After we apply bellman ford from s, we will get an array arr[] back in which arr[i] will give the minimum distance of ith node from s

```
Algorithm
{
arr[]=bellmanford(G,s)
int min=arr[0],index=i
for int i=1:n
if(min¿arr[i]) min=arr[i]
index=i
end for
}
```

Finally if (min¡0) then we can make maximum profit of $|min|$ by going from node s to node i. We can store the path when we relax the edges in bellman ford. Also if min¿=0 then there is no way we can make profit from node s then we will remain on the same node

Time complexity is O(VE)

$\square$

CSCI-GA.1170-001/002 Fundamental Algorithms	December 6, 2014

Solutions to Problem 4 of Homework 11 (22 points)

Name: Sahil Goel *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 $|V'| = 2|V|$, $|E'| \le 2|E| + |V|$ 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: In this case, suppose the original graph is G_1. Initially we create a copy of graph say G_2. Every node u in G_1 corresponds to u' in G_2. These are the following modifications we do in G_1 and G_2

1. Delete all edges in G_1 and G_2 which have toll on them and store all those edges in say T. Note the edges will be copy of each other from G_1 and G_2 but we don't need to store multiple edges therefore we will only store the toll edges from either G_1(or G_2)

2. Now $\forall$ e $\in$ T(toll edges) if e connects u and v, then we will add an edge connecting G_1 and G_2 with a directed edge from u to v'. The edge weight will be same as it was for the original edge.

3. Now $\forall$ v $\in$ V, we will add an zero weight edge connecting G_1 and G_2 with a directed weight edge from v in G_1 to v' in G_2

Now the whole graph is G' with at max 2m+n edges and 2n nodes. 2m+n edges occur when there is no toll edge. If there are toll edges we remove it twice(from G_1 and G_2) but add it only once(from G_1 to G_2), therefore in that case total number of edges will be lesser We can run Bellman-ford algorithm on G' to find shortest distance of all the nodes from s.

If(d(s,t) = $+\infty$, then we don't need to check for negative cycles since there is no path from s to t with at max 1 toll and we will return otherwise

To check for negative cycle in path from s to t

1. Modify Bellman-Ford to run one extra iteration. If any node's shortest distance decreases(i.e. any edge is relaxed) during this iteration we store these nodes in P(potential nodes). It signifies that nodes in P are a part of negative cycle.

2. Observe, if there is a path(In G') from s to any node in P to t, then the d(s,t) = -∞. Since we can in between loop from P to P infinite times by decreasing the total cost reaches -∞.

3. Now our problem is reduced to whether there is a path from s to any P node to t, is yes then d(s,t) = -∞
(a). To do this, first we do a DFS traversal from s and store all the nodes which occur in the path. If any node occur in the path it means that, that node will be reachable from s through some path. Store all reachable nodes in R_s(Reachable from s)
(b). Now take intersection of R_s and P and update $P = R_s \cap P$ because only those nodes will matter now(since other nodes of P are not reachable from s).
(c). Now we have to check if there is a path from any of these P nodes to to t. We can do that in one DFS traversal by reversing the edges and do DFS traversal from t. Store all the nodes in R_t(Reachable from t). Since the edges are reversed if a node u is reachable from t, it means that in the original graph there is a path from u to t. Now we can take intersection of P and R_t and if it is not null, then it means there is a path from a p node to t node. Since all the p nodes were now reachable from s, then there is a path from s to p to t, hence d(s,t) = -∞ and exit otherwise simple exit

Total Time Complexity will be O($V'E'$) + O($V' + E'$) + O($V' + E'$) = O($V'E'$)
Since $|V'| = 2|V|$ and $|E'| \leq |2E| + |V|$
O($V'E'$) $= O(2V(2E + V)) = O(4VE + 2V^2) = O(VE)$

Correctness Proof
1. The path has at max 0 or 1 toll
Notice, since all the toll edges are deleted from the graph and only toll exists in edges from G1 to G2. Also there is no edge which comes back from G2 to G1. We can reach G2 only from one of these (1 toll edges) r from 0 toll edges from v to v'. There is no path from G1 to G2 which has more than 1 toll and since we can't come back to G1, when we enter G2, we can't encounter more tolls

2. The path found is shortest
It has 2 cases, (a) Path has 0 tolls, (b) Path has 1 toll
(a). Observe in G', we have added those edges of zero weights of u to u'. We know that the shortest distance is through a path without any tolls. Suppose we obtain a path to d(s,t') which was not the shortest. Since there is a zero edge path from t to t', therefore d(s,t) =

d(s,t'). Now in graph G_1 we know that bellman ford algorithm gives the shortest distance from s to every other node. Therefore d(s,t) should be minimum as we have only deleted the edges with tolls in G1 and we are not using them. Since d(s,t) is minimum and d(s,t')=d(s,t), d(s,t') should be minimum

(b). If the path has 1 toll then it must go through from G_1 to G_2 through any of the toll edge. Suppose d(s,t') is larger than the shortest distance between s to t in original graph. d(s,t')=d(s,v1) + w(v1,v2') + d(v2',t'), where w(v1,v2') is one toll edge from Graph G1 to G2. Now we know that for every node u d(s,u) will be minimum because of correctness of bellman ford. Also d(v2',t') will be minimum using same argument. When we reach at v1, we check all the adjacent nodes which are reachable from v1, therefore we will relax all possible nodes. Therefore we will always select a v1 and v2' for which the total is minimum. Hence it is correct

$\square$

(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: Modify graph to G' from previous example and run modified Bellman Ford which return d array where d[i] specifies shortest distance from s to i. P signifies the nodes which are part of a negative cycle

```
Algorithm
{
(d[],P)=Bellman-Ford(s,G')
if(P==NULL)
{
d(s,t')=d[t'];
return
}
if checkpathfrom(s, p,t) then d(s,t)=-∞
return d(s,t')
}

checkpathfrom(s,p,t)
{
P=p;
$R_s = DFS(s, G')$
P = P ∩ $R_s$
G'=Reverse(G')
$R_t = DFS(t, G')$
P= P ∩ $R_t$
if(P is not NULL)
d(s,t') = -∞
```

}

Total Time Complexity will be O($V'E'$) + O($V' + E'$) + O($V' + E'$) = O($V'E'$)
Since $|V'| = 2|V|$ and $|E'| \leq |2E| + |V|$
O($V'E'$) $= O(2V(2E+V)) = O(4VE + 2V^2) = O(VE)$

□

(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: In this case, we can initially re-weight the edges of Graph G' by applying Johnson's algorithm(to make every edge positive). We do this by initially adding a fake source and connecting it to all other nodes with distance 0 and then applying Bellman Ford algorithm from s. Let the shortest distance for every node u from s be h(u).
Now, for every edge e from u to v (u,v), we will update the weight of e by
w'(e) = w(e) + h(u) - h(v)
After we have applied this transformation we will have a graph with no negative edges
Now from every node, we can apply Dijkstra's algorithm. Also we know that Dijkstra algorithm takes $O(E' + V' \log V')$ using fibonacci heap. We need to call Dijkstra algorithm from every node therefore total time complexity will be $O(V) * O(E' + V' \log V') = O(VE + V^2 \log V)$ since (E') = O(E) and V' = O(V). After we have applied Dijkstra algorithm, we can update the shortest distances by applying the inverse transformation i.e. w(e) = w'(e) + h(v) - h(u)
c[u][i] stores the shortest distance between city u and city i Algorithm

```
{
G' = modify(G) According to first part
Add fake source $s_f$ and connect it to all vertices with zero weight edge
Bellman-Ford($s_f$,G')
For all edges e(u,v)
w'(e) = w(e) + h(u) - h(v)
end For
For all u ∈ $V'$
c[u][ ] = Dijkstra(G',u)
end For
for i = 1 to n
for j = n+1 to 2n
c[i][j] = c[i][j] - h(i) + h(j)
end for
}
```

j varies from n+1 to 2n because in G' the destination cities are in G_2(n+1 to 2n)

□

(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 $0 \le k \le n$, let

- D^k be the matrix of all shortest distances w.r.t. W' which only use nodes $\le k$ as intermediate nodes.
- C^k be the matrix or all shortest distances w.r.t. W which only use nodes $\le k$ 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($W, W'$)
    $n = W.rows$
    $D^0 = W'$
    $C^0 = W$
    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
                $d^k_{i,j} = \min\left(d^{k-1}_{i,j}, (d^{k-1}_{i,k} + d^{k-1}_{k,j})\right)$
                $c^k_{i,j} = \min\left(c^{k-1}_{i,j}, (c^{k-1}_{i,k} + d^{k-1}_{k,j}), (d^{k-1}_{i,k} + c^{k-1}_{k,j}))\right)$
    Return $C^n$
```

Solution:

```
FLOYD-WARSHALL($W, W'$)
    $n = W.rows$
    $D^0 = W'$
    $C^0 = W$
    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
                $d^k_{i,j} = \min\left(d^{k-1}_{i,j}, (d^{k-1}_{i,k} + d^{k-1}_{k,j})\right)$
                $c^k_{i,j} = \min\left(c^{k-1}_{i,j}, (c^{k-1}_{i,k} + d^{k-1}_{k,j}), (d^{k-1}_{i,k} + c^{k-1}_{k,j}))\right)$
    Return $C^n$
```

Proof of Correctness:
Here we need to prove two things.
(a). $c^n_{i,j}$ gives distance from i to j using at max one toll
(b). The distance given by (a). is shortest

Inductive Proof
Initially $c^0_{i,j}$ is equal to original W matrix. c[u][v] = infinity is there is no edge from u to v else c[u][v]= edge weight between u and v. Since W = C^0 represents the shortest distance from one city to another without using any other node in between i.e. maximum 1 length edge is present from u to v, tolls can't be more than 1. It might be either 0 f the edge wasn't a toll or 1 if the edge was a toll. Also it is the shortest distance between u and v without using any other nodes in between.

Now suppose it is true till C^{k-1} and we have to prove for C^k. Lets prove both cases

(a). Since $c^k_{i,j} = \min\left(c^{k-1}_{i,j}, (c^{k-1}_{i,k} + d^{k-1}_{k,j}), (d^{k-1}_{i,k} + c^{k-1}_{k,j}))\right)$, we know that $c^{k-1}_{i,j}$ has at max one toll edge(assumption: true till k-1). Also if the minimum is $(c^{k-1}_{i,k} + d^{k-1}_{k,j})$, we know that there will be maximum one toll edge in $c^{k-1}_{i,k}$ (because of assumption) and $d^{k-1}_{k,j}$ represents the distance from k to j without any toll, therefore there will be at max one toll (from $c^{k-1}_{i,k}$). Similar is the case if min is $(d^{k-1}_{i,k} + c^{k-1}_{k,j})$

(b). Distance given by a is minimum. Since we know that $c^{k-1}_{i,j}$ is the shortest distance from i to j using nodes less than k-1. Now we take $c^k_{i,j} = \min\left(c^{k-1}_{i,j}, (c^{k-1}_{i,k} + d^{k-1}_{k,j}), (d^{k-1}_{i,k} + c^{k-1}_{k,j}))\right)$. If there is a k such that there is path from i to k and from k to j which is lower than the direct path from i to j, we will update the shorter distance here. Since we know that $c^{k-1}_{i,k}$, $c^{k-1}_{k,j}$, $d^{k-1}_{i,k}$, $d^{k-1}_{k,j}$ is minimum(using inductive assumption). Also we know that $c^{k-1}_{i,j}$ is the shortest distance from i to j using all nodes from 1 to k-1. Hence in this iteration, if there is a path from i to k and from k to j which is shorter than the original i to k path, we are updating it here. Also since it is true till (k-1), there can't be a case after kth iteration such that $c^k_{i,j}$ is more than the shortest distance between i to j, because in this iteration for all i and j we are checking if there is a path from i to k and from k to j which is shortest. Hence we will eventually update the distance in this iteration. Hence the proof is correct

□