CSOR W4231 Analysis of algorithms I
Assignment 4

Xijiao Li (xl2950)

October 18, 2020

Problem 1

Based on Dijkstra's algorithm, we need an additional array `P-LEN` to store the minimum number of edges on the shortest path for each node to s known so far. Then, we can use `P-LEN` as a tiebreaker when selecting which edge we mark to connect the next closest neighbor. The algorithm will be as follow:

```
SORTEST-MIN-E(G,w,s)
  initialize Min-Queue Q
  for each v in N do
    color v white
    set d[v] <- INF
    set p[v] <- ⊥
    Insert(v, Q)
  set d[s] <- 0
  while Q is not empty do
    set u <- Extract-Min(Q)
    color u black
    for each v in Adj[u] that is white do
      if d[v] > d[u] + w(u,v) then
        set p[v] <- u
        set P-LEN[v] <- P-LEN[u]+1
        set d[v] <- d[u] + w(u,v)
        Decrease-Key(v)
      if d[v] == d[u]+w(u,v) and P-LEN[u]+1 < P-LEN[v] then
        set p[v] <- u
        set P-LEN[v] <- P-LEN[u]+1
Backtrace from t to s based on p to output the path
```

Correctness:

Suppose that x is the neighbor of the tree with smallest distance $d(x)$. Let P denote the true shortest path from s to x, which contains the fewest non-tree edges if there are ties. In the proof of original Dijkstra's algorithm, we have already shown that the last edge in P must come directly from the tree, as discussed in class.

Now we want to show that if there are two nodes in the tree u, v that both connect to x with (u, x) and (v, x) respectively, and P_u, P_v are both the shortest path from s to x, our algorithm will choose the one with the fewest nodes along the path.

Without the loss of generality, suppose v is added after u. When we add u to the tree, we go through all the neighbors of u to update their distance, and $d(x)$ is set to the length of P_u, `P-LEN[x]` is set `P-LEN[u]+1`. Then after some time we add v to the tree, and we go through all the neighbors of v to update their distance. Now we find $d(x)$ is equal to the length of P_v, meaning

that there is another shortest path. We compare `P-LEN[x]` with `P-LEN[v]+1`. If `P-LEN[x]` $>$ `P-LEN[v]+1`, we update `P-LEN[x]` to `P-LEN[v]+1` and change $p(x)$ to v; otherwise, no change. In this way, we can show that we always chose the shortest path with fewest nodes known so far.

Since we modify by just adding a new array, the time complexity is not changes. Using Heap: $O((n+e)\log n)$. Using Fibonacci Heap: $O(e+n\log n)$

Problem 2

a.

In this question, let p denotes "the set C is not consistent" and q denotes "C contains a strict inequality $x_u < x_v$ such that the graph G has a path from x_v to x_u".
i. Prove that $p \to q$.
Proof by contradiction. Suppose that p and $\neg p$, i.e., the set C is not consistent, and assume that C does not contain a strict inequality $x_u < x_v$ if there is a path from x_v to x_u in G. There are two cases:

1. For any path from x_v to x_u, C does not contain $x_u \leq x_v$.
   Thus we now there is no edge (x_u, x_v) in the path, so G is acyclic. We can then do the topological sorting on G to find an ordering of the nodes such that all edges go forward in the ordering. And we can assign positive integer values to the nodes from small to large based on this ordering (i.e., assign the first node 1 and the second node 2, etc).

2. For some path from x_v to x_u, C contains a weak inequality $x_u \leq x_v$.
   Thus we know there is an edge (x_u, x_v) in the path, so there is a loop formed by (x_u, x_v) and the path from x_v to x_u. We also know that for any edge (n_1, n_2) in this loop, there is only weak inequality $n_1 \leq n_2$ rather then no strict inequality $n_1 < n_2$, otherwise the assumption q is violated. We can then consider any of these loop as a SCC and collapse them as one single node. Eventually we will get a DAG, for which we can find an ordering using topological sorting as in case 1. Then similarly, we can assign positive integer values to the nodes from small to large based on this ordering, except that if the node is coming from a SCC, we assign all original nodes in the SCC with that value.

ii. Prove that $q \to p$.
Suppose that the there is a path $x_v e_1 n_1 e2 ... x_u$, based on definition, we know that $x_v \leq n_1 \leq ... \leq x_u$, which leads to $x_v \leq x_u$. If C contains a strict inequality $x_u < x_v$, then there is a contradiction and C is not consistent.

Thus, we have proved the two directions and therefore show $p \Leftrightarrow q$

b.

In order to determine whether C is consistent over the positive integers, we will first make a graph G using the definition in part a. Then the algorithm will be as follows:

```
CHECK(G):
  Compute the SCCs in G
  for every SCC do
    if contains an edge (x_u, x_v) with a strict inequality x_u < x_v do
      return false
  return true
```

Correctness:

We know that C is consistent only if (1) there is no loop in G, or (2) the loops in G do not contain any two nodes with a strict inequality in between them. These two cases are corresponding to the two cases in the proof of part a, which we have already proved. Thus we only need to find all the loops in G and check each of them to see whether it contains a strict inequality. From the discussion in class, we know that all the loops in a graph can be detected and collapsed into SCCs. So we can first find all the SCCs and then go through each of them to do the check.

The time of finding all the SCCs is $O(n+m)$ as discussed in class; the time of checking the existence of strict inequality in SCCs is $O(m)$, since in the worst case we need to go through all the edges (e.g., there is only one SCC which contains all the nodes), each in a constant time (for checking). Thus the overall time complexity is $O(n+m)$.

c.

Similarly, in order to compute the minimum solution of a given consistent set C, we will first make a graph G using the definition in part a. Then the algorithm will be as follows:

```
MIN-ASSIGN(G):
  Run DFS(G) to devide G into connected subgraph G_1...G_k
  for every connected subgraph G_j do
    Compute the SCCs in G_j
    Collapse every SCC as a single node // obtain DAG G_j'
    Sort topologically the nodes in G_j'
    for v_i in topological order v_1...v_n do
      if v_i comes from a SCC then
        assign i to all the original nodes in that SCC
      else then
        assign i to v_i
```

The algorithms for DFS, finding SCC, and topological sorting have been explained in class, so I do not write them in details here.

The time of DFS, finding SCCs and doing topological sorting is $O(n+m)$,

and the time of assigning values to all the nodes is $O(n)$ since we go through each node once with constant time. Thus the overall time complexity is $O(n + m)$.

Problem 3

a.

First, remove the edge (x, y), and let T_x and T_y denote the subtrees obtained after removing (x, y). Run BFS in T_x from x, in other to determine which nodes are in T_x and which are in T_y. Then we only need to find the minimum weight edge e_{min} with one endpoint in T_x and the other in T_y. Connecting T_x and T_y using e_{min}, we got the new MST T_1.

```
MST(G,T)
  Remove edge (x, y) from T
  Color all nodes in T white
  BFS(x) and color any visted node black
  set min <- INF
  set e_min <- null
  for every edge e = (n_1, n_2) in G do
    if n_1, n_2 have different color and w[e] < min then
      set min <- w[e]
      set e_min <- e
  Connect T_x and T_y using e_min and output new tree T_1
```

Correctness:
We want to show that T_1 is the new MST. Proof by contradiction. Assume that T_1 is not the MST and we can add an out-of-tree edge (u, v) to replace the edge (a, b) along the path from u to v in T_1, such that $w(c, d) < w(a, b)$. There are two cases:

1. If (a, b) is not in T, or (a, b) is (x, y) with the updated weight.
   Then it is the edge that we newly added in T_1, which connects T_x and T_y. So we know that (u, v) must also connect T_x and T_y. Thus there is a contradiction since from our algorithm we know (a, b) is the minimum weight edge with one endpoint in T_x and the other in T_y.

2. If (a, b) is in T and (a, b) is not (x, y).
   Then there is a contradiction since T is the original MST so that any out-of-tree edge (u, v) has weight $\geq$ maximum weight of the edges along the $u - v$ path in tree T, i.e., $w(u, v) \geq w(a, b)$ (this statement still holds since $w(u, v)$ and $w(a, b)$ have not been modified).

The time of deciding whether a vertex is in T_x is $O(n)$, since we just need to go through all the nodes in T_x once. The time of find the minimum weight edge connecting T_x and T_y is $O(e)$, since we just need to go through all the edges once. The total time is thus $O(n + e)$.

b.

The algorithm is very similar to part a:
First, add the edge (u, v) to T, and let T' denote the new tree obtained, which will contain a unique cycle. Run BFS in T' starting from u to get the cycle. Then we remove the maximum weight edge on that cycle, and get T_2.

Correctness:
We want to show that T_2 is the new MST. Proof by contradiction. Assume that T_2 is not the MST and we can add an out-of-tree edge (c, d) to replace the edge (a, b) that is on the path from c to d in T_2, such that $w(c, d) < w(a, b)$. There are two cases:

1. If T_2 has exactly the same edges as T.
   If (c, d) is not (u, v), meaning $w(c, d)$ has not been change, there is a contradiction since T is the original MST so that any out-of-tree edge (c, d) has weight $\geq$ maximum weight of the edges along the $c - d$ path in tree T. If (c, d) is (u, v), there is a contradiction since from our algorithm we know the updated $w(c, d) \geq$ the weight of any edge on the path from c to d in T_2.

2. If T_2 has one different edge from T.
   If (a, b) is not in T, then it is the one we newly add in T_1, which connects T_x and T_y. So we know that (u, v) must also connect T_x and T_y. Thus there is a contradiction since from our algorithm we know (a, b) is the minimum weight edge with one endpoint in T_x and the other in T_y.

The time of finding the loop is $O(n + E_{T_2}) = O(n)$, since according to the definition of MST, $E_T = n - 1$, so $E_{T_2} = E_T + 1 = n$. The time of find the maximum weight edge on that cycle is also $O(n)$. The total time is thus $O(n)$.

Problem 4

We will use dynamic programming here, since this problem exhibits both overlapping subproblems and optimal substructure.

```
COUNT(G, s, t)
  Initialize a list C
  for each node u in G do
    set C[u] <- 0
  set C[t] <- 1
  Sort topologically the nodes in G
  set start <- false
  for each node u in reversed topological order do
    if u == t then
      set start <- true
    else if start == true then
      for each v in Adj[u] do
        set C[u] <- C[u] + C[v]
  return C[s]
```

Correctness:

The recurrence relation is:

$$C(u) = \begin{cases} 1, & \text{if } u = t \\ \sum_{v \in Adj(u)} C(v), & \text{otherwise} \end{cases}$$

If node u only has one out-neighbor v, then $C(u) = C(v)$ since u can always go to t by first going to v. If node u only has more than one out-neighbors $v_1 ... v_k$, then $C(u) = \sum_{v \in Adj(u)} C(v)$ since u can always go to t by first going to an arbitrary out-neighbor v_i and then use $C(v_i)$ ways to go to t (the subproblem).

We want to do DP in an special ordering such that when we are going to compute $C(u)$, we have already compute $C(v)$, $\forall v \in Adj(u)$. Thus we need to do topological sorting on the DAG G, and compute C from the last to the first.

Running time:

The topological sorting takes $O(n+e)$ time. The computing of C takes $O(n+e)$ time since it goes through every nodes once and every edge once. So the overall run time is $O(n + e)$.

Problem 5

a.

Consider there is a cycle C in G_r of negative weight, i.e.,

$$\sum_{(i,j)\in E_C} w_{ij} = \sum_{(i,j)\in E_C} (rc_{ij} - p_j) = r \sum_{(i,j)\in E_C} c_{ij} - \sum_{j\in N_C} p_j < 0$$

thus,

$$r \sum_{(i,j)\in E_C} c_{ij} < \sum_{j\in N_C} p_j, \quad r < \frac{\sum_{j\in N_C} p_j}{\sum_{(i,j)\in E_C} c_{ij}} = r(C) \leq r*$$

b.

Consider there is a cycle C in G_r of negative weight, i.e.,

$$\sum_{(i,j)\in E_C} w_{ij} = \sum_{(i,j)\in E_C} (rc_{ij} - p_j) = r \sum_{(i,j)\in E_C} c_{ij} - \sum_{j\in N_C} p_j > 0, \ \forall \ C \text{ in } G$$

thus,

$$r \sum_{(i,j)\in E_C} c_{ij} > \sum_{j\in N_C} p_j, \quad r > \frac{\sum_{j\in N_C} p_j}{\sum_{(i,j)\in E_C} c_{ij}} = r(C), \ \forall \ C \text{ in } G$$

since $r* = \max(r(C))$, so $r > r*$.

c.

```
MAX-CYCLE(G)
  set low <- 0
  set high <- R
  while true do
    set r <- r'
    set r' <- (low + hign)/2
    calculate w based on r'
    Run Bellman-Ford(G) to detect negative cycle // obtain d after the
        last update
    if there is a negative cycle C do
      if r' < r and r' < r* and r-r' < ϵ do
        return r'
      else do
        set low <- r'
    else do
      Run FIND-ZERO-CYCLE(G) to detect zero-weight cycle
```

```
    if there is a zero-weight cycle C do
      return r'
    else do
      set high <- r'

FIND-ZERO-CYCLE(G, d)
  Initialize a new graph G'=(V', E') with V = V'
  for each edge (i,j) in E do
    if d(i) + w_{i,j} = d(j) do
      add (i,j) to E'
    Run DFS(G') to detect cycle C
    if exist cycle C do
      output C
      return true
    else
      return false
```

First set 0 as the lower bound of $r*$, and set $R = \max_{(i,j)\in E}(p_j/c_{ij})$ to be the upper bound. Since R refers to the maximum profit-to-cost ratio for any edge in G, so no cycle C would be consistent of edges that have higher ratios and R is the upper bound for $r*$. Also, $p_i > 0, c_{ij} > 0$, 0 is the lower bound for $r*$. We perform binary search in this range:

1. store the former guess in r
2. have a new guess r' of $r*$ as (`low` + `high`)/2.
3. calculate w based on $w_{ij} = r'c_{ij} - p_j$.
4. run Bellman-Ford algorithm to detect negative cycles.
   - if there is a negative cycle, meaning $r' < r*$, increase r';
   - otherwise all cycles are positive, meaning $r' \geq r*$, decrease r'.

We end the search if

1. we have a good enough guess whose difference from $r*$ is within ϵ: for the most recent two guesses r and r', we have $r' < r$, $r' < r*$ (i.e., there is a negative cycle C based on r'), and $r - r' < \epsilon$. Return r' and the cycle C.
2. there is no negative cycle but a zero-weight cycle C, meaning that we have found $r* = r'$, return r' and C.

The binary search has $O(\log(R - 0/\epsilon))$ iterations. During each iteration,

- the time of Bellman-Ford algorithm is $O(|N||E|) = O(|N|^3)$ since there is an edge between any two cities and $|E| = O(|N|^2)$.
- the time of searching zero-weight cycle is $O(|E|)+O(|N|+|E|) = O(|N|^2)$.

so each iteration takes $O(|N|^3)$, and the total running time is $O(|N|^3 \log(R/\epsilon))$.