CS521 Fall 2011 \ Assignment #3

Ariel Stolerman

1)

Let $G = (V, E)$ be a directed graph with $|V| = n, |E| = m\big(= O(n^2)\big)$. Also let A be the adjacency matrix of G and $adj[v]$ be the adjacency list for $v \in V$.

From the definition of G^2, E^2 should contain an edge (u, v) between each $u, v \in V$ that hold either $(u, v) \in E$ or $(u, w), (w, v) \in E$ for some $w \in V$.

Compute G^2 for an adjacency-matrix G:

```
1. Initialize $A^2$ to be the adjacency matrix for $G^2$ of size $n \times n$ (all initial values are $0$).
2. For each row $i = 1, \dots, n$ in $A$:
    2.1. For each column $j = 1, \dots, n$:
        2.1.1. If $A_{ij} == 1$ then:
            2.1.1.1. $A_{ij}^2 = 1$
            2.1.1.2. For each column $k = 1, \dots, n$:
                2.1.1.2.1. $A_{ik}^2 = \max\{A_{ik}^2, A_{jk}\}$
3. Return $G^2 = (V, E^2)$ where $E^2$ is defined by the adjacency matrix $A^2$.
```

Correctness and running time:

Row 2.1.1.1. makes sure that all direct edges $(u, v) \in E$ will be included also in E^2. Row 2.1.1.2.1. that is executed for all 2nd level vertices of the currently checked vertex, makes sure that if (u, v) is already in E^2 it stays that way, but if $(u, v) \notin E^2$ and $(u, w), (w, v) \in E$ then (u, v) is included in E^2 as well.

The initialization of A^2 will is $\Theta(n^2)$. The first two loops are always executed, adding $\Theta(n^2)$ to the total running time. The last loop executes only for edges that existed in G, so each edge will add $\Theta(n)$ cost, concluding to a total addition of $\Theta(mn)$. The total running time is therefore $\Theta\big(n(n + m)\big) = \Theta(n^2 + nm)$.

Compute G^2 for an adjacency-list G:

I will use an adjacency matrix for G^2 for intermediate calculations, and address all vertices and their indices the same way (i.e. for vertices $i, j \in V$ the corresponding edge is in A_{ij}).

```
1. Initialize $A^2$ to be the adjacency matrix for $G^2$ of size $n \times n$ (all initial values are $0$).
2. For each $i \in V$:
    2.1. For each $j \in adj[i]$:
        2.1.1. Set $A_{ij}^2 = 1$
        2.1.2. For each $k \in adj[j]$:
            2.1.2.1. Set $A_{ik}^2 = 1$
3. Build the adjacency list for $G^2$ from $A^2$.
```

4. Return the corresponding $G^2 = (V, E^2)$.

<u>Correctness and running time</u>:

Each original edge from E is copied to A^2 in row 2.1.1., and each vertex that is distant by 2 from the one we're now checking is also added to A^2 in the loop on line 2.1.2., resulting with a correct build of G^2. We use an intermediate adjacency matrix for G^2 as it won't cost less to immediately use adjacency lists – the latter will require for each $k \in adj[j]$ to check if it's already in $adj[i]$ and only if not then add it, going through all $adj[i]$ for that.

The initialization of A^2 is $\Theta(n^2)$. Setting A^2 to hold all original edges in E is $\Theta(m)$, and for each original edge we add $O(n)$ edges to A^2, a total of $O(mn)$. Creating an adjacency list from A^2 takes $\Theta(n^2)$. The total is therefore $O\big(n(n+m)\big) = O(n^2 + mn)$.

Emailing with Tom, I was told that in order to give a Θ bound (which is required), I can say something like this: suppose for each $(u, v) \in E$ there are k edges to be added, then the tight bound would be $\Theta(n^2 + km)$. This doesn't give much information… if we mark m_i as the total number of edges per node $i \in V$ ($\sum_{i \in V} m_i = m$ in a directed graph) then we can say the total number of A^2 updates is $\sum_{i \in V}\big(m_i + \sum_{j \in adj[i]} m_j\big)$, and I don't think there's a way to simplify it to a constant expression of m, n, thus the O-bound given above is the closest answer that can be given.

2)

The main diagonal of the matrix BB^T holds the total number of edges that row/column vertex has (in and out). That is, for $i \in V$: $(b_{ii}) = |\{(i, v) \in E\} \cup \{(v, i) \in E\}|$. That can be seen from the fact that the total number of 1s or -1s for a certain row i, i.e. vertex $i \in V$, represents the number of edges i has in/out of it, and that number is calculated in the main diagonal due to matrix multiplication rules (sum of index multiplications, transferring every -1 to 1).

Furthermore, for every other entry in BB^T, if we take its absolute value it indicates the total number of edges between that entry's corresponding row and column vertices. That is, the number of edges between $i, j \in V$ is $|(b_{ij})| = |(b_{ji})|$. That can be seen from the fact that when multiplying row i of B by column j of B^T, the sum will be contributed only by indices that aren't 0 for both i and j, and when that happens (and edge between i and j) we know for sure that one is the inverse of the other (as an edge between i and j will always leave one and enter the other), thus contributing -1 to the sum. Therefore the entry (b_{ij}) will hold the total number of edges between i and j (disregarding direction), times -1. It immediately implies symmetry w.r.t. the main diagonal of BB^T.

The above also implies that the sum of each row or column of BB^T will always be 0.

3)

<u>Calculation of G^T for adjacency matrix represented G</u>:

1. Let A be the adjacency matrix of G.
2. Initialize the transposed adjacency matrix A^T (initially all values are 0).
3. Calculate A^T and set it as the adjacency matrix of G^T.

If it is not trivial, I will detail:

```
    3.1. For $i = 1, \dots, n$:
        3.1.1. For $j = 1, \dots, n$:
            3.1.1.1. $A^T_{ij} = A_{ji}$
4. Return $G^T = (V, E^T)$ where $E^T$ is defined by the transposed adjacency matrix $A^T$.
```

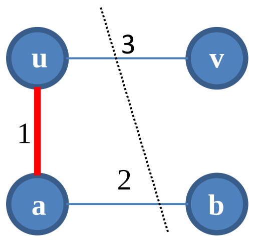
The running time of initializing and computing the transposed matrix is $\Theta(n^2)$, making the entire running time $\Theta(n^2)$.

<u>Calculation of G^T for adjacency list represented G</u>:

```
1. Initialize list of size $n$ of $adj^T$ lists (all initially empty).
2. For each $v \in V$:
    2.1. For each $u \in adj[v]$:
        2.1.1. $adj^T[u] \leftarrow v$
3. Return $G^T = (V, E^T)$ where $E^T$ is defined by the list of $adj^T$.
```

The initialization takes $\Theta(n)$. We then go over all m edges in E (actually, all adjacent nodes for each node in V) and for each apply a constant-time operation (row 2.1.1.), thus concluding to $\Theta(m + n)$ running time.

4)



In this counterexample we show an edge (u, v) that is safe for $A \subseteq E$, however it is not the next edge to be added to A according to Kruskal's MST algorithm (but we don't care, as it satisfies the terms of the question).

Here $S = \{u, a\}, V - S = \{v, b\}, A = \{(u, a)\}$, and the edge (u, v) is definitely safe for A (any MST in this case is the graph itself) but it is not light for the cut $(S, V - S)$, as (a, b) is.

5)

Let T be a minimum spanning tree of a graph G w.r.t. the weight function $w: E \to \mathbb{R}$, and let w' be another weight function such that for some $(x, y) \in E(T)$ it satisfies $\forall (u, v) \in E: w'(u, v) = \begin{cases} w(u, v), & (u, v) \neq (x, y) \\ w(u, v) - k, & (u, v) = (x, y) \end{cases}$, where k is some positive integer. We will prove that T is a minimum spanning tree also w.r.t. w'.

Assume by contradiction that T is not a MST under w', and according to the change of the weight of (x, y) under w', $w(T) = w'(T) + k$ (since no other weight is changed except for (x, y)). Let S be a MST under w', therefore $w'(S) < w'(T)$ (since we assume T is not a MST under w').

If $(x, y) \notin S$, since w' changed only the weight of (x, y), then $w'(S) = w(S)$. But then $w(S) = w'(S) < w'(T) = w(T) - k < w(T) \Rightarrow w(S) < w(T)$. But T is a MST under w, a contradiction.

If $(x, y) \in S$, since $w'(S) < w'(T)$ then $w(S) = w'(S) + k < w'(T) + k = w(T) \Rightarrow w(S) < w(T)$, but T is supposed to be a MST under w, therefore no spanning tree with strictly smaller weight should exist, a contradiction.

Thus we have proven that T must be a MST also under w'.

6)

We will describe an algorithm for finding the new MST, show its correctness and running time. I define a tree as a set of edges, rather than a graph (edges and nodes; the nodes are simply the original set of nodes of the given graph).

Say we add the new node v and edges $E_v = \{e_1, \dots, e_k\}$ to the graph G, and let T be the already calculated MST. The algorithm:

- Define $E' = T \cup E_v$ and $V' = V \cup \{v\}$
- Run Kruskal's algorithm on $G' = (V', E')$ to find the MST T'.
- Return T'.

<u>Correctness</u>:

The correctness derives from the fact that we can build an MST with only the old MST (all vertices in V and edges in T) combined with the new node v and its new edges E_v. This is proven in the following:

Consider the following algorithm:

```
1.   Add the edge $e_i \in E_v$ such that $w(e_i) = \min_{e \in E_v}\{w(e)\}$ to $T$.
2.   For each edge $e \in E_v - \{e_i\}$:
     2.1. Add $e$ to $T$
     2.2. Remove the heaviest edge on the cycle that was created. This is done as follows: for the added edge $e = (u, v)$ where $u$ is the new node, using BFS find the path $v \to u$ on the graph $(V \cup \{u\}, T - \{e_i\})$ (i.e. ignore the lightest edge connecting $u$ to the MST that we added on step 1; since it's a tree there's only one such path) and remove the heaviest edge on that path.
```

The correctness for this algorithm is shown by comparing with the MST that would have been created by Kruskal's algorithm, between each two iterations of the loop over $e \in E_v$. Before that, since e_i is the lightest edge connecting to v, it would have been added along with all other edges that were selected for T, since no cycle would have been created and it's the only edge connecting to v.

Now, consider adding the edge $e \in E_v - \{e_i\}$: Kruskal's algorithm (in short: K) would do everything the same up until encountering e. When getting to e, if it is the heaviest edge on the cycle it creates in the algorithm described above, that means in K we already got all other edges on the cycle (as they are lighter than e), and now e creates a cycle, thus will not be taken – and K continues as before, thus our algorithm creates the same MST as K. If e is NOT the heaviest edge on that cycle, then when K would arrive to it, it would have taken it, and continued regularly up until the heaviest edge, denoted e', that creates a cycle. Then K would not take it, and continue all the rest the same. Thus, here also our algorithm results with the same MST that K would create, by simply removing the heaviest edge on the cycle created with e, which is e'. Therefore the modified T is still a MST through all the iterations in row 2.

Thus, we have proven that after adding v, E_v we can create a new MST only from $T \cup E_v$.

<u>Running time</u>:

Since T is a MST then $|T| = |V| - 1$, by definition. In addition $|E_v| = k \leq |V|$ (in the case v has edges to all $u \in V$). Thus $m = |E| = O(|V|) = O(n)$. Applying Kruskal's algorithm is $O(m \lg n)$, so in our case: $\boxed{O(n \lg n)}$.

The reason we did not simply use the algorithm that helped proving we can construct an MST from $T \cup E_v$ is that its running time is $O(kn) = O(n^2)$, which is asymptotically larger. Eventually $k = |E_v|$ will determine which algorithm is better to use.

7)

The algorithm **does not** compute a minimum spanning tree correctly (not at all times, anyway). Here is a counterexample: let $V = \{u, v, x\}$ and $E = \{e_1 = (u, v), e_2 = (v, x), e_3 = (x, u)\}$ with a weight function w that satisfies $w(e_i) = i$. The proposed algorithm might divide V into $V_1 = \{v\}, V_2 = \{x, u\}$ such that $MST_{V_1} = \emptyset$ and $MST_{V_2} = \{e_3\}$ (the only edge in V_2). The "conquer" phase will take e_1 to connect between V_1, V_2, resulting with the MST $T = \{e_1, e_3\} \Rightarrow w(T) = 4$. But the true sole MST of this graph is $\{e_1, e_2\}$ with the weight 3, proving the proposed algorithm wrong.

8)*

a.

When using Kruskal's algorithm to construct a MST, and at each iteration we take a safe edge to the set A built up to now, by the sorted order of the edges' weights. Since all weights are distinct, that must create a unique MST. To show second-best-MSTs are not unique, here is a counterexample:

Let $V = \{v_1, v_2, v_3, v_4\}$ and a weight function defined over the edges in E: $w(v_1, v_2) = 1, w(v_1, v_3) = 2, w(v_1, v_3) = 3$, $w(v_3, v_4) = 4, w(v_2, v_4) = 5$. The MST is $T = \{(v_1, v_2), (v_1, v_3), (v_3, v_4)\}$ with $w(T) = 1 + 2 + 4 = 7$. However there are two second-best-MSTs: $T_1 = \{(v_1, v_2), (v_2, v_4), (v_2, v_3)\}, T_2 = \{(v_1, v_3), (v_3, v_4), (v_2, v_3)\}$ with $w(T_1) = 1 + 5 + 3 = w(T_2) = 2 + 4 + 3 = 9$. Thus the second-best-MSTs are not unique.

b.

We can show a construction for a second-best-MST:

Initialize $T_2 = T$. For each $(x, y) = e \in E - T$:

- Add e to T_2
- Remove the heaviest edge $e_T \in T$ from T_2 on the cycle that is formed when added e to T_2.
- Record the total weight $w(T_2)$

At the end, simply take one of the T_2s that had the smallest weight.

This built is correct since we constructed a tree with the minimum weight possible when T is not available, i.e. when removed one edge from T and replaced it with another. Because of the uniqueness of T we know any T_2 is not a MST, but at most will be a second-best-MST. As shown in exercise 6, creating a cycle in a tree and removing the heaviest edge on that cycle outputs a MST, so here by removing the second-heaviest (as e is the heaviest, otherwise it would have been in T) outputs a second-best-MST.

It is sufficient to replace only one edge in T since replacing more than one will necessarily create a tree T_3 that satisfies $w(T_3) > w(T_2)$, hence is not a second-best-MST: say we replace two edges in T with e_1, e_2, then they replace two original edges from T: $e_{T,1}, e_{T,2}$ that were the heaviest from T on those cycles (excluding e_1, e_2). But, since e_1, e_2 were not originally in T, it means they are the absolute heaviest on those cycles. Hence we can build a tree T_2 with replacing only $e_{T,1}$ by e_1 (or $e_{T,2}$ by e_2), and satisfy $w(T_2) < w(T_3)$ – thus created a second-best-MST T_2 with only one edge replacing original edge from T, as required. This build can be proven for all T_i were $i \geq 3$ (and indicates how many original edges from T we replace with edges in $E - T$).

c.

To find $\max[u, v]$ for all $u, v \in V$ we can use a modification of the BFS algorithm. For running BFS with $u \in V$ as root:

- Set for each adjacent node $v \in V$ (that is, $(u, v) \in E$): $\max[u, v] \coloneqq w(u, v)$
- Set for each non adjacent node $v \in V$: $\max[u, v] \coloneqq \max\{max[u, v.\pi], max[v.\pi, v]\}$, where $v.\pi$ is the predecessor of v on the route from u.

Do the above for all $u \in V$ to get a complete $\max$ defined over all $u, v \in V$ (for all $u \in V$ we can say $\max[u, u] = 0$).

Correctness:

Since it is a tree, there is only one route between each two nodes, otherwise we would have a cycle. Therefore, it is sufficient when starting from some node u to set the maximum-weight on the path to v as the maximum between the maximum weight on the path to v's parent, which is guaranteed to be maximum over the entire path up to it, and the weight of the newly added edge, $w(v.\pi, v)$ (inductive proof).

Running time:

The running time of $BFS(u)$ for all $u \in V$ is $O(n + m)$, and since we apply it on a tree then $m = n - 1 \Rightarrow O(n)$. Finally, applying the modified BFS on all $u \in V$ will conclude with a total of $O(n^2)$.

d.

Given what is proven in the previous sections, to find a second-best-MST we can do the following:

- Find the MST of the graph, denoted T (or assume it is given, like in section b).
- Go over all $(u, v) \in E - T$ and find the edge that minimizes $w(u, v) - \max[u, v]$.
- Add that edge to T and remove from T the edge with weight $\max[u, v]$ (weight is unique over all E).

Correctness:

We showed it is possible to build a second-best-MST from an existing MST by removing one edge from the MST and replacing it with another. Going over the entire $w(u, v) - \max[u, v]$ is calculating the addition in weight to the MST the edge (u, v), which will replace the heaviest edge on $u \to v$ in the MST, will give. Therefore taking the edge that minimizes that size will assure a second-best-MST.

Running time:

Finding the initial MST is $O(m \lg n)$. When calculating $\max$ we can do that only for u ad hoc when checking $(u, v) \in E - T$, a total cost of $O\big((m - n)n\big)$ ($m - n$ edges in $E - T$ and n running time for modified-BFS on each u). Finding the minimum one can be done on the fly. Finally, adding the new edge and removing the one with $\max[u, v]$ is $O(n)$ – finding it over all

$e \in T$ (this can be done in $O(1)$ if for calculating $\max$ we save the edge that gives maximum weight and not just the weight, but asymptotically it doesn't matter). That all concludes to $O(mn)$.