MPCS 55001 Algorithms Autumn 2023
Homework 7

Zhiwei Cao zhiweic

Collaborators: Student A, Student B

Instructions: Write up your answers in LaTeX and submit them to Gradescope. **Handwritten homework will not be graded.**

Collaboration policy: You may work together with other students. If you do, state it at the beginning of your solution: **give the name(s) of collaborator(s)** and the nature and extent of the collaboration. Giving/receiving a hint counts as collaboration. Note: you must write up solutions **independently without assistance**.

Internet sources: You must include the url of any internet source in your homework submission.

1 Dynamic MST (13 points)

Suppose that we have a weighted graph $G = (V, E, w)$ with **distinct** edge weights and an MST T already computed. We want to update a minimum spanning tree for G as vertices and edges are added or deleted.

For each part below, write **pseudocode** for an algorithm which takes $G = (V, E, w)$ and its MST T, the modified graph $G' = (V', E', w')$, and the vertices/edges being modified as inputs, and outputs an MST T' for G'.

(a) (3 points) G' has the same vertices as G and one new edge e is added. Your algorithm must run in $O(V)$ time. **Prove your solution is correct**, and analyze its running time.

Algorithm 1 1 (a): NewEdgeAdded

```
1: function NewEdgeAdded($G$, $T$, $G'$, $e$)
2: Input: $G = (V, E, w)$ and its MST $T$, the modified graph $G' = (V', E', w')$, and the vertices/edges $e = (u, v)$ with wight $w(e)$.
3: Output: an MST $T'$ for $G'$
4: $T' \leftarrow T \cup e$ // add the new edge to MST
5: $cycleDetected, maxWeightEdge \leftarrow applyDFSfromutofindthecycleandmax-weighted-edgeintheT'$
6: // apply cut property
7: if $w(maxWeightEdge) > w(e)$ then
8:     $T' \leftarrow T' - maxWeightEdge$ // remove the edge with higher weight
9: end if
10: return $T'$
```

Correctness proof: need to prove 1. T' contains all vertices; 2. T' is a tree, which doesn't contain any cycles; In the beginning, T is an MST that contains all vertices of the G. Add a new edge e in the G to build T' in G' doesn't affect the vertices. At the same time, new edge e must produce a cycle in the T'. By the cut property, we could remove an edge with the max weight that is greater or equal to the weight of e, otherwise, it will remove the e. It could break the cycle condition and ensure all vertices are still connected in the T'. Moreover, the edge removal by the cut property also makes sure that the new edge e added to the T' could provide a better MST in G' because the cut property makes sure that $w(T') \leq w(T)$.

Running Time: Adding edge and removing edge take $O(1)$, DFS takes $O(E + V)$, but in this case, E only has $|V| - 1$ in a tree, then DFS takes $O(V + (V - 1))$. In conclusion, the running time is $O(V)$

(b) (4 points) G' has the same vertices as G and one edge e is removed. You may assume that G' is still connected. Your algorithm must run in $O(E)$ time. **Prove your solution is correct**, and analyze its running time.

Algorithm 2 1 (b): EdgeRemoved

```
function EdgeRemoved($G$, $T$, $G'$, $e$)
Input: $G = (V, E, w)$ and its MST $T$, the modified graph $G' = (V', E', w')$, and the vertices/edges $e = (u, v)$ with wight $w(e)$.
Output: an MST $T'$ for $G'$
if $e$ is not in $T$ then
    return T // is the removed edge is not in T, doesn't affect the MST
end if
// prove the case that e is an edge of MST
$X$ and $Y$ be the two separate part of the tree, where $X \in S$ and $Y \in V - S$
$minEdge \leftarrow NIL$
$minEdgeWeight \leftarrow -\infty$
for each edge $e' = (u, v)$ in (E-{$e$}) do
    if (u in $X$ and v in $Y$) or (u in $Y$ and v in $X$) then
        if $w(u, v) < minEdgeWeight$ then
            $minEdge \leftarrow (u, v)$
            $minEdgeWeight \leftarrow w(u, v)$
        end if
    end if
end for
$T' \leftarrow T - \{e\} + minEdge$
return $T'$
```

Correctness proof: 1. If the removed edge e is not in the MST T, it doesn't affect the result of MST, since the T is untouched; 2. If removed edge e in the MST, it will cause the MST lost the connection and provide two seperate trees X and Y. In order to make sure the property of the MST, the algorithm will check each edge that connect X and Y and find the minimum-weight edge in order to connect two seperate trees and construct the new MST T'.
Running Time: The algorithm check all the edges expect the removed edge if the removed edge is in the MST T, so it will take $|V| - 1$ steps. In conclusion: the algorithm takes $O(V)$

(c) (4 points) G' has one new vertex v and k new edges $e_1, e_2, \ldots, e_k$ are added connecting v to vertices in G, where $1 \le k \le |V|$. The running time of your algorithm should not depend on $|E|$, the number of edges in G. **No credit for $\Omega(E \log V)$ time algorithms.**

Algorithm 3 1 (c): AddKEdge

```
1: function AddKEdge($G$, $T$, $G'$, $e = \{e_1, e_2, \ldots, e_k\}$, $v$)
2: Input: $G = (V, E, w)$ and its MST $T$, the modified graph $G' = (V', E', w')$, the edges $e = newedges\{e_1, e_2, \ldots, e_k\}$ with wight $w(e_i)$ and vertex v.
3: Output: an MST $T'$ for $G'$
4: $minEdge \leftarrow NIL$
5: $minEdgeWeight \leftarrow \infty$
6: for each edge $e_i = (u, v)$ in e do
7:     if $w(e_i) < minEdgeWeight$ then
8:         $minEdge \leftarrow e_i$
9:         $minEdgeWeight \leftarrow w(e_i)$
10:    end if
11: end for
12: $T' \leftarrow T + minEdge$
13: return $T'$
```

Correctness proof: The main idea is to find the minimum edge between the added vertex v and the MST T, then the T' in G' could ensure the tree connectivity and the minimum property of MST. Since there is a new vertex, the only thing I needed to do was find the minimum-weighted edge and connect to the T. Besides, all the vertices of G are in the T

Proof: based on the theorem of cut, assign the added vertex $v \in V - S$ and the original MST $T \in S$. They are in a connected undirected weighted graph T'. The algorithm will iterate all cut edge weight between the added vertex v and all vertex in the T, and find the minimum-weighted edge $e' = (u, v)$ where $u \in S$. Based on the cut property, the latest-weighted edge $e' = (u, v)$ crossing the cut, where $u \in S$ and $v \in V - S$, is in some MST T' of G'

Running Time: The running time is related to the size of k, which means the number of edges we need to check between e and $e' \in T$. And the worst case is O(V), since k is bound in the range of $1 \le k \le |V|$. In conclusion: the running time is $O(k)$

(d) (2 points) Show that T may provide no information about T' when we remove a vertex and the incident edges. Specifically, give an example of $G = (V, E, w)$ and $G' = G - v$ such that the MST of G has no edges in common with the MST of G'.

Assume there is a graph G: G = (V, E, w) where V = A, B, C, D;
E = $\{(A, D), (B, D), (C, D), (A, B), (B, C), (C, A)\}$; w(A, D)=1, w(B, D)=1, w(C, D)=1, w(A, B)=3, w(B, C)=4, w(C, A)=1. The MST of G will contains the edges $\{(A, D), (B, D), (C, D)\}$ with the weight 3. If remove the vertex D and its incident edges which are $\{(A, D), (B, D), (C, D)\}$, then the $G' = G - \{D\}$. In another words, this operation remove the MST T from G, and only $G' = \{(A, B), (B, C), (C, A)\}$ left. At the same time, T' can not get any information from T in this case. Then the G' needs to construct the MST from the beginning, then MST T' will contain $\{(A, B), (B, C)\}$ with the weight 7 in this case. In this case, T' and T have completely different MST.

2 Borůvka's Algorithm (12 points)

Borůvka's MST algorithm (1926) can be described as a distributed version of Kruskal's algorithm. The input is an undirected connected weighted graph $G = (V, E)$ with edge weight function w. We assume that distinct edges have distinct weights. (Alternatively, the algorithm can be modified to work with arbitrary edge weight functions by adding a tie-breaking rule.)

The algorithm proceeds by phases. In the first phase, we begin by having each vertex mark the edge incident to it whose weight is minimum. (For example, if the graph were a 4-cycle with edges of weights 1, 3, 2, and 4 around the cycle, then two vertices would mark the "1" edge and the other two vertices would mark the "2" edge.) This creates a forest F of marked edges. In the next phase, each tree F_i in F marks the minimum weight edge incident to it, i.e., the minimum weight edge having one endpoint in the tree F_i and one endpoint not in F_i, creating a new forest F'. The algorithm repeats these phases until there is only one tree.

(a) (2 points) Given the forest $F \subseteq E$, write **pseudocode** to compute the connected components F_i in $O(V + E)$ time.

Algorithm 4 2 (a): $FindF_i$

```
1: function $FindF_i(F)$
2: Input: a forest F of G
3: Output: the connected components $F_i$ in $F$
4: at the very beginning, each vertex takes one component, and there is no connection between any two vertices
5: $component \leftarrow dict()$ // an map to store edges'
6: $componentMin \leftarrow dict()$
7: $component_{id} \leftarrow 0$
8: for each $u$ in F.V do
9:     if $component[u] == NIL$ then // $u$ doesn't belong to any component
10:        $component_{id} \leftarrow component_{id} + 1$
11:        $componentMin[component_{id}] \leftarrow (NIL, \inf)$ // store the edge and the weight
12:        $BFS(F, u, component, component_{id}, componentMin)$ // find all reachable vertices and assign all vertex $v$ that are reachable to: component[v] $\leftarrow edge_{id}$
13:    end if
14: end for
15: Initialize the cheapest edge for each component to "None"
16: return component, componentMin // each element of the component represent a $F_i$ of the forest $F$
17:
18: function $BFS(F, u, component, component_{id}, componentMin)$
19: $q \leftarrow \emptyset$
20: ENQUEUE(Q,u)
21: while $Q \neq \emptyset$ do
22:     $v \leftarrow DEQUEUE(Q)$
23:     if $component[v] == NIL$ then
24:         $component[v] \leftarrow component_{id}$ // assign component
25:         for each $x$ in Adj[v] do
26:             if $component[x] == NIL$ then
27:                 ENQUEUE(Q,x)
28:             end if
29:             if $component[x]$ is not $component_id$ then
30:                 if $componentMin[component_id] == (NIL, \inf)$ or $w(v, x) < componentMin[component_id][1]$ then
31:                     $componentMin[component_id] \leftarrow ((v, x), w(v, x))$
32:                 end if
33:             end if
34:         end for
35:     end if
36: end while
37: return $component$, $componentMin$
```

(b) (3 points) Write **pseudocode** for Borůvka's algorithm.

Algorithm 5 2 (b): Borůvka's

```
function Borůvka's($G$)
Input: $G = (V, E, w)$ is a graph $G = (V, E)$ with the weight function $w$
Output: a MST $T$ of $G$
reference form Wikipedia: Borůvka's algorithm
Initialize $F \leftarrow (V, E')$ where E' is $\{\}$
$allChecked \leftarrow False$ // check if all trees in the forest are merged
while $allChecked == False$ do
    $component, componentMin \leftarrow FindF_i(F)$
    //Initialize the cheapest edge for each component to "None"
    for each $c$ of component do
        $componentMin[c][0] \leftarrow NIL$
        $componentMin[c][1] \leftarrow$ inf
        // each component at least has one vertex, and the componentMin is stored by the $component_{id}$, and the $component_{id}$ is stored by the vertex
    end for
    for each edge $e = (u, v)$ in G.E do
        if $component[u] \neq component[v]$ then // u, v are in F, make sure e is not in tree
            $u_{min} \leftarrow componentMin[component[u]][0]$ the cheapest edge for the component of u in F
            $w(u_{min}) \leftarrow componentMin[component[u]][1]$
            if $u_{min}$ == NIL or $w(e) \leq w(u_{min})$ then
                $componentMin[component[u]] \leftarrow (e, w(e))$ set e is the cheapest edge for the component of u
            end if
            $v_{min} \leftarrow componentMin[component[v]][0]$ the cheapest edge for the component of v in F
            $w(v_{min}) \leftarrow componentMin[component[v]][1]$
            if $v_{min}$ == NIL or $w(e) \leq w(v_{min})$ then
                $componentMin[component[v]] \leftarrow (e, w(e))$ // set e is the cheapest edge for the component of v
            end if
        end if
    end for
    if all components have the cheapest edge set to $NIL$ then
        $allChecked \leftarrow True$
        // means $componentMin[c_i[0]] \leftarrow NIL$
        // means there is only one tree or one component, and the algorithm cannot find an edge $e = (u, v)$ that $component[u] \neq component[v]$, which means there is no replacement happened, so the cheapest edge keep being $NIL$
    else
        $allChecked \leftarrow False$
        for each component c in componentMin do
            if $componentMin[c][0] \neq NIL$ then
                add $componentMin[c][0]$ to $F.E'$ // append MST edge into $F$
            end if
        end for
    end if
end while
return $F$
```

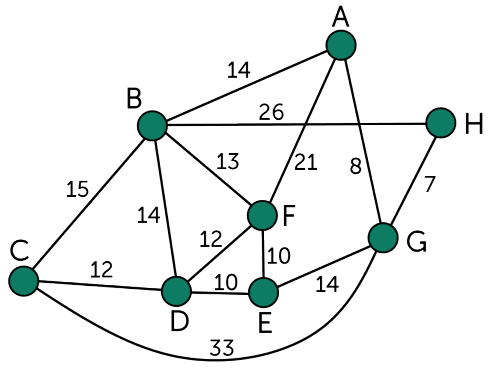
(c) (3 points) **Prove** that each phase can be implemented in $O(V + E)$ time. **Prove** that Borůvka's algorithm has at most $O(\lg V)$ phases. This gives a running time of $O((V + E)\lg V)$ for the entire algorithm.

Phase 1: In order to assign the minimum weight to the vertex. At the very beginning, the algorithm will generate a new graph with all the vertices but no edges. Then assign each single vertex to a single component. This step costs $|V|$ since there are no edges at first. Then The algorithm will check each edge in the original graph G, in order to find the minimum weighted edge which will cost $|E|$. In conclusion, the phase 1 will take $O(V + E)$ times.
Phase 2: After adding the cheapest edges found in the previous step to the forest F, the next step is to find the new connected components. This can be done by performing a BFS (or DFS) starting from each vertex. Since each vertex and edge is visited exactly once during these searches, the total time for this step is $O(V + E)$
Combine two phases $\leftarrow O(E + V)$
Borůvka's algorithm: In each phase, there is at least one edge added to connect two components into one component. The algorithm shrinks the times of operation by half. The algorithm starts with $|V|$ individual components since each vertex takes one component. The number of components halves each phase, as a result, the times that the algorithm needs to generate one single component is $\log V$. In conclusion, the time complexity of Borůvka's algorithm is $O(\log V)$
Entire algorithm: In conclusion: Multiply the running time, then the running time of the entire algorithm $\leftarrow O((V + E)\lg V)$

(d) (4 points) **Prove** the correctness of the algorithm by arguing that $E(F)$ is always contained in the MST. Conclude that the constructed graph F never has any cycles. [Hint: use induction]
Basis Case: At the very beginning, $F = (V, E')$, where $E' = \{\}$, which means F is a graph that contains all vertex of G but doesn't have any edges. An empty set of edges is a subset of any kind of MST because an MST is a subset of the graph's edges. At the same time, if there are no edges in the basis case, then there are no cycles in $E(F)$.
Inductive: Assume that E(F) is contained in the MST and has no cycles after k phases. Prove that E(F) is contained in the MST and has no cycles after k+1 th phases.
In each phase of the algorithm, the algorithm ensures that it will pick the cheapest edge $e = (u, v)$, where u and v are in different components (or trees). Since u and v belong to different components, then the edge e crosses the cut by the cut property of MST. The algorithm will check all edges that can be cut edge between two components. By the cut property, there is one edge that will be a part of an MST. Therefore, the E(F) is contained in the MST after adding the edge e. Moreover, since each component is a tree, and the added edge $e = (u, v)$ where u and v are in different components. Then the merged component will not contribute a cycle in an MST. Then by the induction hypothesis, E(F) is contained in the MST and has no cycles after k phases. Then in the (k+1)th phases, the algorithm will find the suitable edge e based on the component it has, then it can merge them and create no cycles.
Conclusion: By the induction, E(F) is contained in the MST and F doesn't have any cycles in each phase, since F contains trees. The algorithm will keep running until all vertices are connected, then F will span all vertices and F will be the MST of the G eventually.

3 Lightest Branch Spanning Tree (17 points)

Let $G = (V, E, w)$ be an undirected weighted graph. The **heaviest branch** of a spanning tree T is an edge which has the highest weight in T. Moreover, we refer to this weight as the **heaviest branch value**. A **lightest branch spanning tree** (LBST) is a spanning tree that has the least possible **heaviest branch value**.



For example, in the graph above, the tree given by $T_1 = \{(A, B), (B, F), (C, D), (D, E), (E, F), (E, G), (G, H)\}$ is an LBST with the heaviest branch being (A, B) and value 14.

The tree given by $T_2 = \{(A, B), (B, C), (C, D), (D, E), (E, F), (E, G), (G, H)\}$ has (B, C) as the heaviest branch with weight 15; this is **not** an LBST.

(a) (3 points) Prove that any minimum spanning tree (MST) is also a lightest branch spanning tree (LBST).
Prove by the contradiction and assume that there is an MST that is not an LBST.
Find the contradiction condition that there is a spanning tree T' which has a lighter heaviest edge than MST T.
Assign e to be the heaviest edge of T and e' to be the heaviest edge of T'. By the assumption, it is known that $e > e'$. Since T' has a lighter heaviest edge than T', then the sum weight of T' is less than the sum weight of T, which contradicts the MST property. As a result, the sum weight of T' must be greater or equal to the sum weight of T. However, if the total weight of T' is greater than or equal to T, despite T' having a lighter heaviest edge, it implies that T' must have other edges whose combined weights are greater than those in T. It contradicts the property of an MST because we could replace the heavier edges in T' with lighter edges from T to get a spanning tree with a smaller total weight, which would mean T was not an MST. However, T is defined as an MST.
Therefore any minimum spanning tree (MST) is also a lightest branch spanning tree (LBST) by contradiction.

(b) (2 points) Prove or disprove: Every LBST is also an MST.
Disprove by a counterexample: (edges with the same weight are allowed)
w(a,b) = 2, w(b,c) = 1, w(c,d) = 1, w(a,d) = 2, w(a,c) = 1000.
In this case, the MST will be (a,b)(b,c)(c,d), the sum is 4, and an LBST could be (a,b)(a,d)(b,c), the sum is 5. However, it maintains the property of LBST, which makes sure the heaviest branch has a value of 2.

(c) (2 points) Write **pseudocode** for function `LBST_below` which takes as input $G = (V, E, w)$ and a number b and outputs true if G has an LBST with heaviest branch value at most b. Else, it outputs false. Justify why your function works and compute its running time.
The function will check the max edge and compare it with the b, its max weighted edge is less than b, which means it must have an LBST with a restriction of b value. Then the function applied the Kruskal's algorithm, we added edges that could connect an LBST without cycles, and updated the heaviest weight edge, if the value is less than b. If not, the algorithm will break the loop. We will check if all vertices are concluded in the current LBST. If yes, return true, otherwise not.
Time complexity: $O(E \log V)$ since it is based on Kruskal's algorithm.

(d) (2 points) Use the function you wrote in part (c) to write **pseudocode** for an $O\big((V + E) \log V\big)$ time algorithm to compute a LBST. *You may not call Prim or Kruskal.*

(e) (1 point) Prove that if graph $G_b = (V, E_b)$ where $E_b = \{e : (e \in E) \cap (w(e) \leq b)\}$ is connected then the heaviest branch value of an LBST of G is at most b.
Since G_b is a connected graph, then there must exist an MST T_b. At the same time, if all edges' weight is less

Algorithm 6 3 (c): LBST_below

```
function LBST_below($G$,$b$)
Input: $G = (V, E, w)$, a number $b$
Output: outputs true if $G$ has an LBST with heaviest branch value at most $b$
sort edge by weight:$w(e1) < w(e2) < w(e3) < w(e) < \cdots < w(em)$ where m == —E—
if $w(em) < b$ then
    return True
end if
$x \leftarrow \emptyset$
$heaviestBranch \leftarrow -\inf$
for i from 1 to m do
    if ei doesn't create a cycle with edges in X then
        add ei to X
        $heaviestBranch = max(heaviestBranch, w(ei))$
        if $heaviestBranch > b$ then
            break
        end if
    end if
end for
if X contains all vertices of the G then
    return True
else
    return False
end if
```

Algorithm 7 3 (d): FindLBST

```
function FindLBST($G$)
Input: $G = (V, E, w)$
Output: compute the LBST
$low \leftarrow minimumedgeweightofG$
$high \leftarrow maximumedgesweightofG$
$heaviestBranck \leftarrow high$
while $low \leq high$ do
    $mid \leftarrow (low + high)/2$
    if $LBST\_below(G, mid)$ then
        $heaviestBranck \leftarrow mid$
        $high \leftarrow mid - 1$
    else
        $low \leftarrow mid + 1$
    end if
end while
$A \leftarrow \emptyset$
sort the edge by weight ($w(em)$) and the maximum weight cannot be greater than $haviestBranch$
for i from 1 to m do
    if $ei$ doesn't create a cycle with edges in A then
        add $ei$ to A
    end if
end for
return A
```

or equal to b, then the maximum weighted edge cannot be greater than b. Since we already proved that MST is also an LBST, then the maximum weighted edges of T aslo correspond to the heaviest branch of an LBST of G_b

(f) (1 point) Prove that if graph $G_b = (V, E_b)$ is disconnected, you can greedily produce a partial LBST of G missing only the edges between the connected components of G_b.
Suppose the G_b is disconnected, then there are several connected graphs G_i in G_b. Based on the proof of question e, it is known that each graph G_i could construct a LBST. Each $LBST_i$ also ensures that the maximum weighted edge is less than the value of b, because G_i has the same attribute like G_b.

(g) (1 point) Earlier you studied the Quicksort algorithm, a randomized algorithm for sorting an array of numbers. Describe (2-3 sentences is fine) how you would modify Quicksort to instead select the median value in an array of numbers in linear expected time.
Choose a pivot randomly, then separate the array into two parts: the number less than the pivot and the number larger than the pivot. If the pivot is at the index of (n/2) or between ((n/2), (n/2)+1), then found; if the pivot is less than (n/2), then do recursive on the right part; otherwise, do recursive on the left part.

(h) (5 points) Combine the results of (e) and (f) with a choice of b to reduce the edges you need to consider by half in each iteration and reduce the complexity of your algorithm to $O(V + E)$. Write pseudocode for your algorithm. You may assume that you already have a function to find the median of an unsorted array of numbers in deterministic linear time.

Hint: Consider the relationship between the number of nonisolated vertices (i.e., vertices with adjacent vertices) and the number of edges in a graph. How might you be able to leverage this relationship in analyzing the running time of your algorithm?

Algorithm 8 3 (h): LBST

```
function LBST($G$)
Input: $G = (V, E, w)$
Output: compute the LBST
$mediumWeight \leftarrow$ the median weight of all edges in the G
$smallerGraph \leftarrow$ all vertices; the edges which are smaller than mediumWeight
$largerGraph \leftarrow$ all vertices; the edges which are greater than mediumWeight
// by applying the result of (e)
if smallerGraph is connected then
    call $FindLBST(smallerGraph)$ // assume my solution of question d is correct
    return $FindLBST(smallerGraph)$
else
    // by applying the result of (f)
    $partialLBST \leftarrow \emptyset$
    for each connect component c in smallerGraph do
        $t \leftarrow FindLBST(T)$
        add T to partialLBST
    end for
    $extraReuqiredEdge \leftarrow$ edges of largerGraph to connect the partialLBST
    // then connect the components of smallerGraph using the edges of largerGraph, by applying the cut property
    $components \leftarrow allconnectsub - graphofsmallerGraph$
    $sortedEdge \leftarrow$ sort the edges of largerGraph by its weight
    for each e(u,v) of sortedEdge do
        if u and v in different component then
            add e to extraReuqiredEdge
            merge these two component as one
        end if
        if there is only one componet then
            break
        end if
    end for
    $LBST \leftarrow partialLBST + extraReuqiredEdge$
    return LBST
end if
```

4 Programming: Image Segmentation (20 points)

Follow this GitHub Classroom link to accept the assignment. Your code should be pushed to GitHub; you do not need to include it here.