23.1-1

Let (u, v) be a minimum-weight edge in a connected graph G. Show that (u, v) belongs to some minimum spanning tree of G.

Step 1 of 1

We prove this by contradiction. Let T be a MST without the minimum weight edge (u,v). Add (u,v) to the MST. We now have a cycle is the resulting tree of which the edge (u,v) is a part. If we remove any other edge (apart from (u,v)) from the cycle. We have another spanning the T (because there will still be a path from any vertex to any other vertex) obviously, weight of the spanning tree ie.e, the sum edge weights, $W(T^1) \leq W(T)\, W(T^1) \leq W(T)$, because we have replaced an existing edge with a minimum weight edge. Since T is a MST $W(T^1) \leq W(T)$. Therefore, we have $W(T^1) = W(T)$ implying that T^1 is also a MST, thus we see that there always exists some MST of which any minimum weight edge is a part.

Note: A minimum weight edge can be left out of a MST if and only if the MST already has at least one another edge also of the same minimum weight. If all the edge weights are distinct there is only a unique MST.

23.1-2

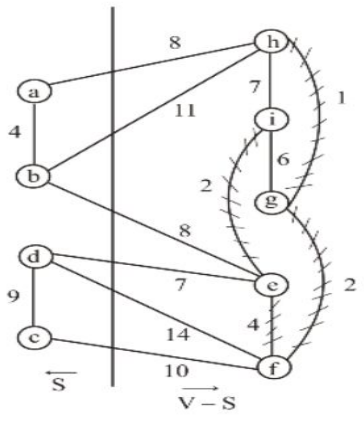
Professor Sabatier conjectures the following converse of Theorem 23.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A crossing $(S, V - S)$. Then, (u, v) is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.

Step 1 of 3

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function ω defined on E. Let 'A' be a subset of E that is included in some minimum spanning tree for G. Let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A crossing $(S, V - S)$. Then (u, v) is a light edge for the cut.

Step 2 of 3

Proof:



DIAGRAM

Step 3 of 3

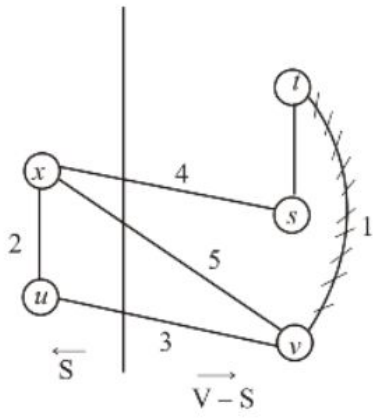
According to the definition of safe edge, it does not violate minimum spanning tree property. Safe edge does not cross $(S, V-S)$ even though it is a light edge that is which crosses the $(S, V-S)$.

23.1-3

Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

Step 1 of 2

Consider a cut with u and v on separate sides.



DIAGRAM

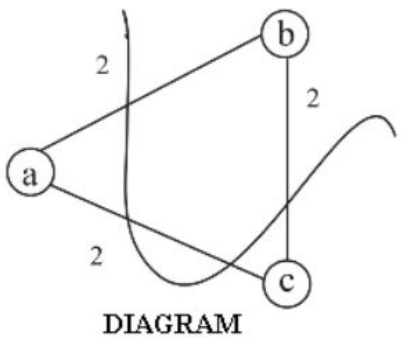
Step 2 of 2

If (u, v) is not a light edge then clearly the graph would not be a minimum spanning tree.

23.1-4

Give a simple example of a connected graph such that the set of edges $\{(u, v)$: there exists a cut $(S, V - S)$ such that (u, v) is a light edge crossing $(S, V - S)\}$ does not form a minimum spanning tree.

Step 1 of 2



DIAGRAM

Step 2 of 2

A triangle whose edge weights are all equal is a graph in which every edge is a light edge crossing some cut. But the triangle is cyclic, so it is not a minimum spanning tree.

23.1-5

Let e be a maximum-weight edge on some cycle of connected graph $G = (V, E)$. Prove that there is a minimum spanning tree of $G' = (V, E - \{e\})$ that is also a minimum spanning tree of G. That is, there is a minimum spanning tree of G that does not include e.

Step 1 of 1

If e be a maximum weight edge in a minimum spanning tree, then replacing it with any other edge on the cycle would yield a better minimum spanning tree.

23.1-6

Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

Step 1 of 3

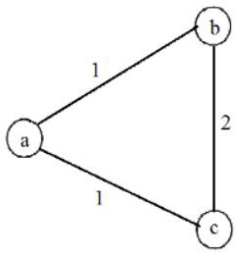
Suppose that for every cut of G, there is a unique light edge crossing the cut. Let us consider two minimum spanning trees T and T^1, of G. We will show that every edge of T is also in T^1 which means that T and T^1 are the same tree and hence there is a unique minimum spanning tree.

Step 2 of 3

Consider any edge $(u,v) \in T$. If we remove (u,v) from T, then T becomes disconnected resulting in a cut $(s, v-s)$. The edge (u,v) is a light edge crossing the cut $(s, v-s)$. Now consider the edge $(x,y) \in T^1$ that crosses $(s, v-s)$. It, too, is a light edge crossing this cut. Since the light edge crossing $(s, v-s)$ is unique, the edges (u,v) and (x,y) are the same edge. Thus $(u,v) \in T^1$. Since we choose (u,v) arbitrary, every edge in T is also in T^1.

Step 3 of 3

Counter example for the converse: Consider the following graph with 3 vertices a, b, c and weights $w(a,b) = w(a,c) = 1$ and $w(b,c) = 2$.



Here, the graph is its own minimum spanning tree, so the graph has a unique minimal spanning tree containing edges (a, b) and (a, c), however cut ({a}, {b, c}) doesn't have a unique light edge crossing the cut.

23.1-7

Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

Step 1 of 1

For a graph with all positive edge weights, let S be an edge subject that connects the graph and has minimum total weight. Suppose for a contradiction that S is not a tree. Then S must contain a cycle. By removing any single edge from this cycle, we could create a new edge subset. So with lower weight than S that still connects all the vertices. Thus S did not have minimum total weight after all. Proof by contradiction.

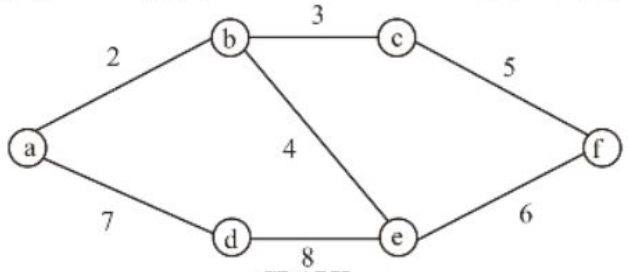
Now suppose that some edge weights may be non-positive. A minimum weight edge subset connecting the graph need not form a tree. For example, consider the complete graph on vertices $\{x,y,z\}$ with edge weights $\omega(x,y)=\omega(x,z)=\omega(y,z)=0$. The set of all 3 edges $\{(x,y),(x,z),(y,z)\}$ connects all the vertices and has minimum total weight 0, but does not form a tree.

23.1-8

Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T. Show that for any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights of T'.

Step 1 of 2

Let consider a graph $G=(V,E)$ in which the vertices $V\{a,b,c,d,e,f\}$.



GRAPH

Step 2 of 2

With weights as above. Let T be a minimum spanning tree with vertices b,c,e,f and L is the list of spotted edge weights.

$$L=\{3,4,5,6\}.$$

Consider another minimum spanning tree T^1 with vertices a,b,d,e,f the edge weights are $7,8,6,2,4$. The sorted list are $\{2,4,6,7,8\}$ in which the L is sorted in T^1 also.

23.1-9

Let T be a minimum spanning tree of a graph $G = (V, E)$, and let V' be a subset of V. Let T' be the subgraph of T induced by V', and let G' be the subgraph of G induced by V'. Show that if T' is connected, then T' is a minimum spanning tree of G'.

Step 1 of 1

We assume the contradict condition " M ' is a minimum weight spanning tree and T ' is not a minimum weight spanning tree in graph G ' " to prove.

If we joined the subset of edges T ' to M ', then we would obtain a spanning tree M in the graph G. The weight of M would be smaller than the weight of T but it contradicts the condition that T is a minimum weight spanning tree.

Thus, our assumption is false and T ' is a minimum weight spanning tree in the graph G '.

23.1-10

Given a graph G and a minimum spanning tree T, suppose that we decrease the weight of one of the edges in T. Show that T is still a minimum spanning tree for G. More formally, let T be a minimum spanning tree for G with edge weights given by weight function w. Choose one edge $(x, y) \in T$ and a positive number k, and define the weight function w' by

$$w'(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \neq (x, y) , \\ w(x, y) - k & \text{if } (u, v) = (x, y) . \end{cases}$$

Show that T is a minimum spanning tree for G with edge weights given by w'.

Step 1 of 1

Let $\varpi(T) = \sum_{(x,y)\in T} \varpi(x,y)$. We have $\varpi^1(T) = \varpi(T) - K$. Consider any other spanning Tree T^1, so that $\varpi(T) \le \varpi(T^1)$.

If $(x,y) \notin T^1$, then $\varpi^1(T^1) = \varpi(T^1) \ge \varpi(T) > \varpi^1(T)$

If $(x,y) \in T^1$, *then* $\varpi^1(T^1) - \varpi(T^1) - K \ge \varpi(T) - k = \varpi^1(T)$

Either way, $\varpi^1(T) \le \varpi^1(T^1)$ and so T is a minimum spanning tree for weight function ϖ^1.

23.1-11 ★

Given a graph G and a minimum spanning tree T, suppose that we decrease the weight of one of the edges not in T. Give an algorithm for finding the minimum spanning tree in the modified graph.

Step 1 of 3

We decrease the weight of an edge e that is not on the T (Minimum Spanning Tree). It's possible that the weight of this edge will be less than some light edges of G. In this case, we should remove all the unsafe edges in T, and replace then with new edges. We will compare e's new value with all T's edges crossing all the possible cuts of G. The two parts of those cuts will contain either end of e. We can see that, together with e, those possible unsafe edges form a circle in G.
Thus, we should remove the largest edge on the circle and include the new edge e and get a new MST T'. By doing so, we can guarantee that we create a new MST based on the argument of Theorem 23.1. When we decrease the weight of edge (u, y), we need to check edges (u, y), (u, v), (v,w), (w, x), and (x, y). And we will add (u, y) into the graph and remove one edge with the maximum weight among those edges we checked.

Step 2 of 3

The inputs of our algorithm include the graph G, the minimum spanning tree T, and the edge e = (u, v) whose weight changes. The output is the new minimum spanning tree T'.

```
AdjustWeight(G,T,e)
if ( e's weight decreased and e is in T)
   then return T
if (e's weight decreased and e ∉ T)
 then tempWeight ← w(e)
      tempEdge ← e
      using u as the start point, do depth-first search on T until reach v.
      for each edge f on DFS path from u to v
        if w(f) > tempWeight
          then tempWeight ← w(f)
               tempEdge ← f
      if tempEdge ≠ e
        then remove tempEdge from T
        add e into T
      return T
```

Step 3 of 3

According to Theorem 23.1 and Corollary 23.2, our algorithm can guarantee the correctly rebuilding of the minimum spanning tree.

The DFS will run with O(T) time and the comparison will take at most O(T). Thus, the running time of case 4 will be less than O(E). The total running time of our algorithm, therefore, will be of O(E).

23.2-1

Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted into order. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskal's algorithm so that the algorithm returns T.

Step 1 of 1

For each edge e in T simply make sure that it precedes any other edge not in T with weight $w(e)$.

23.2-2

Suppose that we represent the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(V^2)$ time.

Step 1 of 2

Prim's algorithm:

```
MST-PRIM (G, r)      // G = (V, E)
    1. for u=1 to |V|
    2. u.dist = ∞
    3. u.pred = ∞
    4.   r.dist = 0
```

5. Create a min-priority queue Q for indexes of vertices according to values of dist
6. While Q is not empty
7. u = EXTRACT-MIN(Q)
8. For v = 1 to |V|
9. if m[u][v]=1
10. if v exists in Q and weight of (u, v) < v.dist
11. v.pred = u
12. v.dist = weight of (u,v)

Step 2 of 2

The Algorithm has two nested for loops with V steps each, thus is in $O(V^2)$.

23.2-3

For a sparse graph $G = (V, E)$, where $|E| = \Theta(V)$, is the implementation of Prim's algorithm with a Fibonacci heap asymptotically faster than the binary-heap implementation? What about for a dense graph, where $|E| = \Theta(V^2)$? How must the sizes $|E|$ and $|V|$ be related for the Fibonacci-heap implementation to be asymptotically faster than the binary-heap implementation?

Step 1 of 2

For Standard Heaps $0(|E|\lg|v|)$

For Fibonacci Heaps $0(|E|+|V|\lg|V|)$

For $|E| \in O(|V|)$ we get $O(|V|\lg|V|)$ for standard heaps and $O(|V|+|V|\lg|V|)=O(|V|\lg|V|)$. for Fibonacci heaps so that the complexities are the same. For $|E| \in O(|V|^2)$ we get $O(|V|^2\lg|V|)$ for standard heaps and $O(|V|^2+|V|\lg|V|)=O(|V|^2)$ for Fibonacci heaps, so that Fibonacci heaps win out.

Step 2 of 2

For the third question, we try to find the point where the Fibonacci implementation becomes faster

$$
\begin{aligned}
e\lg v &> e+v\lg v \\
e\lg v - e &> v\lg v \\
e(\lg v - 1) &> v\lg v \\
e &> \frac{v\lg v}{\lg v - 1} \\
&= \frac{v}{1-\frac{1}{\lg v}} \\
&\approx v
\end{aligned}
$$

That is, for large v, Fibonacci heaps are almost always faster than standard heaps.

23.2-4

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

Step 1 of 2

We know that Kruskal's Algorithm takes O(V) time for initialization $O(E \lg E)$ time to sort the edges, and $O(E\alpha(V))$ time for the disjoint-set operations, for a total running time of $O(V + E\lg E + E\alpha(V)) = O(E\lg E)$.

If we knew that all of the edge weights in the graph were integers in the range from 1 to $|V|$, then we could sort the edges in $O(V+E)$ time using counting sort. Since the graph is connected, $V = O(E)$, and so the sorting time is reduced to O(E). This would yield a total running time of $O(V+E+E\alpha(V)) = O(E\alpha(V))$, again since $V = O(E)$, and since $E = O(E\alpha(V))$. The time to process the edges, not the time to sort them, is now the dominant term.

Step 2 of 2

If the edge weights were integers in the range 1 to ϖ for some constant ϖ, then we could again use counting sort to sort the edges more quickly. This time sorting would take $O(E+\varpi) = O(E)$ time, since ϖ is a constant. Total running time $O(E\alpha(V))$.

23.2-5

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

Step 1 of 1

Let us consider that all edge weights in a graph are integers in the range from 1 to $|V|$, and then the time taken by prim's algorithm is determined by the speed of the priority queue operations. With the queue implemented as a Fibonacci heap, it takes $O(E+V\log v)$ time. Since the keys in the priority queue are edge weights, it might be possible to implement the queue even more efficiently when there are restrictions on the possible edge weights.

We can improve the running of the prim's algorithm, if the weights are integers in the range from 1 to W, we can implement the queue as an array $Q[0\ldots w+1]$, where each slot i in the array holds all the vertices with that weight as a key. To EXTRACT-MIN, simply scan the array and find the first non-empty slot in $O(1)$ time. Decrease-Key also takes $O(1)$ time, remove the vertex from one list and insert it in another. This gives a total running time of $O(E)$. If the range is 1 to $|V|$, the idea above end up slowing down the algorithm, so using a heap is better.

23.2-6 ★

Suppose that the edge weights in a graph are uniformly distributed over the half-open interval $[0, 1)$. Which algorithm, Kruskal's or Prim's, can you make run faster?

Step 1 of 1

Kruskal's running time is dominated by the time to sort the edge weights. If the weights are uniform over the half-open interval $[0,1]$, then we can use bucket sort to sort them in time $\Theta(n)$ on average. This will bring the average running time of Kruskal down to $O\left(E\lg^{*}(E)\right)$, which is probably linear in E.

23.2-7 ★

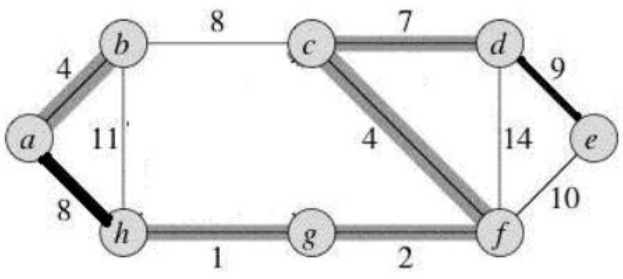
Suppose that a graph G has a minimum spanning tree already computed. How quickly can we update the minimum spanning tree if we add a new vertex and incident edges to G?

Step 1 of 3

Let v be the new added vertex and T be the original MST with root s. Add the lightest incident edge to T and we get a new spanning tree T^1. For each new edge left, say (u,v), add it to the current tree, which must result in a cycle, then remove the edge with the maximum weight on the cycle. Repeat this until all new added edges are processed. To find the maximum-weighted edge in a cycle, first find the path P, from v to s and the path P^1 from u to s. From the root s down, compare the vertices appearing on P and P^1. The first common vertex x on both paths is the least common Ancestor of u and v. Further more, the path $x \to u$, $x \to v$ and edge (u,v) from a cycle. The maximum weighted edge on this cycle is the maximum of edges on paths P and P^1, and edge (u,v). This runs in $O(eV)$ time, where e is the number of new edges.

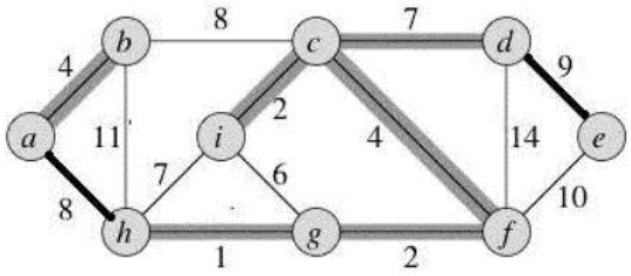
Step 2 of 3

For example, the graph is



Step 3 of 3

After adding the new vertex



23.2-8

Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ

Step 1 of 1

Professor Toole's algorithm falls here. The first one is that when we recurse one or both of the graphs $G^1 = (V^1, E^1)$ and $G^{11} = (V^{11}, E^{11})$ might be disconnected, in which case its minimum spanning tree is not defined. However, the Algorithm still fails, even if we assume that G is a complete graph.

For example, consider the complete graph with vertices x, y, z, w and edge weights, $\varpi(x,y) = 2$, and $\varpi(x,z) = \varpi(x,\varpi) = \varpi(y,\varpi) = \varpi(z,w) = 1$. If the initial recursion splits the graph into $[x,y]$ and $[z,w]$, then we have no choice but to select edge (x,y) in the partition $[x,y]$, producing a non-optimal spanning tree.

23-1 Second-best minimum spanning tree

Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

We define a second-best minimum spanning tree as follows. Let $\mathcal{T}$ be the set of all spanning trees of G, and let T' be a minimum spanning tree of G. Then a ***second-best minimum spanning tree*** is a spanning tree T such that $w(T) = \min_{T'' \in \mathcal{T} - \{T'\}} \{w(T'')\}$.

a. Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.

b. Let T be the minimum spanning tree of G. Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.

c. Let T be a spanning tree of G and, for any two vertices $u, v \in V$, let $max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $max[u, v]$ for all $u, v \in V$.

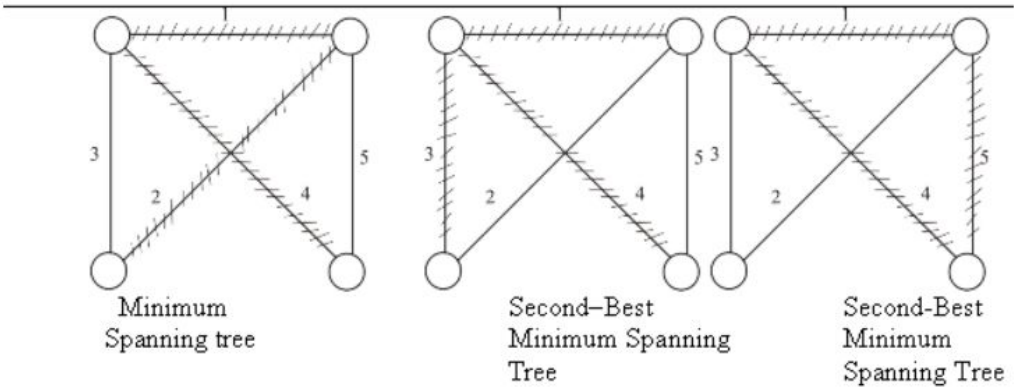
d. Give an efficient algorithm to compute the second-best minimum spanning tree of G.

Step 1 of 5

Let $G=(V,E)$ be an undirected connected graph with weight function $\varpi: E \to R$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

Step 2 of 5

(A) Assume T and T' are distinct minimum spanning trees. Then some edge (u,v) is in T but not in T'. Since all edge weights are distinct the edges on the unique path from u to v in T' must then be strictly lighter than (u,v) contradicting the fact that T is a minimum spanning tree.



Minimum Spanning tree

Second–Best Minimum Spanning Tree

Second-Best Minimum Spanning Tree

Step 3 of 5

(B) Let T be a minimum spanning tree. We wish to find a tree that has the small EST possible weight that is larger than the weight of T. We can insert an edge $(u,v) \notin T$ by removing some other edge on the unique path between u and v. By the above property such a replacement must increase the weight of the tree. Be carefully considering the cases it is seen that replacing two or more edges will not produce a tree better than the second-best minimum spanning tree.

Step 4 of 5

(C) Let max (u,v) be the weight of the edge of maximum weight on the unique path between u and v in a spanning tree to compute this $O(V^2)$ time for all nodes in the tree do the following:

For each node v perform a traversal of the tree. In order to compute $\max(v,l)$ for all $l \in V$ simply maintain the largest weight in encounted so far for the path being investigated. Doing this yields a linear time Algorithm for each node and we therefore obtain a total of $O(V^2)$ time.

Step 5 of 5

(D) Using the idea of the second sub-exercise and the provided Algorithm in the third sub-exercise, we can now compute T_2 from T in the following way:

Compute $\max(u,v)$ for all vertices in T. Compute for any edge (u,v) not in T the difference $\varpi(u,v) - \text{Max}(u,v)$. The two edges yielding the smallest positive difference should be replaced.

23-2 Minimum spanning tree in sparse graphs

For a very sparse connected graph $G = (V, E)$, we can further improve upon the $O(E + V \lg V)$ running time of Prim's algorithm with Fibonacci heaps by preprocessing G to decrease the number of vertices before running Prim's algorithm. In particular, we choose, for each vertex u, the minimum-weight edge (u, v) incident on u, and we put (u, v) into the minimum spanning tree under construction. We then contract all chosen edges (see Section B.4). Rather than contracting these edges one at a time, we first identify sets of vertices that are united into the same new vertex. Then we create the graph that would have resulted from contracting these edges one at a time, but we do so by "renaming" edges according to the sets into which their endpoints were placed. Several edges from the original graph may be renamed the same as each other. In such a case, only one edge results, and its weight is the minimum of the weights of the corresponding original edges.

Initially, we set the minimum spanning tree T being constructed to be empty, and for each edge $(u, v) \in E$, we initialize the attributes $(u, v).orig = (u, v)$ and $(u, v).c = w(u, v)$. We use the *orig* attribute to reference the edge from the initial graph that is associated with an edge in the contracted graph. The c attribute holds the weight of an edge, and as edges are contracted, we update it according to the above scheme for choosing edge weights. The procedure MST-REDUCE takes inputs G and T, and it returns a contracted graph G' with updated attributes $orig'$ and c'. The procedure also accumulates edges of G into the minimum spanning tree T.

```
MST-REDUCE(G, T)
 1  for each v ∈ G.V
 2      v.mark = FALSE
 3      MAKE-SET(v)
 4  for each u ∈ G.V
 5      if u.mark == FALSE
 6          choose v ∈ G.Adj[u] such that (u, v).c is minimized
 7          UNION(u, v)
 8          T = T ∪ {(u, v).orig}
 9          u.mark = v.mark = TRUE
10  G'.V = {FIND-SET(v) : v ∈ G.V}
11  G'.E = ∅
12  for each (x, y) ∈ G.E
13      u = FIND-SET(x)
14      v = FIND-SET(y)
15      if (u, v) ∉ G'.E
16          G'.E = G'.E ∪ {(u, v)}
17          (u, v).orig' = (x, y).orig
```

```
18              (u, v).c' = (x, y).c
19          else if (x, y).c < (u, v).c'
20                  (u, v).orig' = (x, y).orig
21                  (u, v).c' = (x, y).c
22      construct adjacency lists G'.Adj for G'
23      return G' and T
```

a. Let T be the set of edges returned by MST-REDUCE, and let A be the minimum spanning tree of the graph G' formed by the call MST-PRIM(G', c', r), where c' is the weight attribute on the edges of $G'.E$ and r is any vertex in $G'.V$. Prove that $T \cup \{(x, y).orig' : (x, y) \in A\}$ is a minimum spanning tree of G.

b. Argue that $|G'.V| \leq |V|/2$.

c. Show how to implement MST-REDUCE so that it runs in $O(E)$ time. (*Hint:* Use simple data structures.)

d. Suppose that we run k phases of MST-REDUCE, using the output G' produced by one phase as the input G to the next phase and accumulating edges in T. Argue that the overall running time of the k phases is $O(kE)$.

e. Suppose that after running k phases of MST-REDUCE, as in part (d), we run Prim's algorithm by calling MST-PRIM(G', c', r), where G', with weight attribute c', is returned by the last phase and r is any vertex in $G'.V$. Show how to pick k so that the overall running time is $O(E \lg \lg V)$. Argue that your choice of k minimizes the overall asymptotic running time.

f. For what values of $|E|$ (in terms of $|V|$) does Prim's algorithm with preprocessing asymptotically beat Prim's algorithm without preprocessing?

Step 1 of 6

Minimum spanning tree in sparse graphs

Minimum spanning tree is the set of all vertices such that the resultant tree is connected and having minimum weight over all other spanning tree.

a)

Since, every minimal-incident edge belongs to any MST for a vertex say v, therefore the edge that is returned by the call MST-REDUCE also called as minimal edges are all cross the cut. Here, A is the minimum spanning tree that has been formed by MST-Prim and in the union operation with T (set of edges), all the edges of minimum spanning tree are taken into consideration. So, this always leads to formation of minimum spanning tree.

Since PRIM-MST returns the safe edges. This leads to the completion of the tree to MST.

Therefore, the resultant tree defined by $T \cup \{(x, y).orig' : (x, y) \in A\}$ is a minimum spanning tree of graph G.

Step 2 of 6

b)

A value has been entered in *G'* if the node has more than one node that is making a minimum span of the tree. It is because if there were more ways by which the spanning of tree would have been done, it was taken into consideration.

Therefore, no of components that are stored in *G'* are always less than $\frac{|V|}{2}$.

That is $|G'.V| \leq \frac{|V|}{2}$

Step 3 of 6

c)

MST-REDUCE can be implemented using the simple data structure such as UNION, MAKE_SET and FIND_SET. Running time of thus implemented algorithm will be $\mathrm{O}(E)$

MAKE_SET (i) : $set[i] = null$

FIND_SET (i) : $return \;\; set[i]$

UNION (i, j) :

if $(set[i] = null)$ then

begin

$last = last + 1$

$set[i] = last$

end

$set[j] = set[i]$

Running time of(MST-REDUCE)=COST of for loop in line 1+COST of for loop in line 4+building $V(G')$ + COST OF for loop in line 12+COST(constructing adjacency list for G')

$= \mathrm{O}(V) + \mathrm{O}(E) + \mathrm{O}(V) + \mathrm{O}(E)$

Since,

$E \geq V - 1$ For connected graphs

Therefore,

$= \mathrm{O}(E)$

Hence proved

Step 4 of 6

d)

In above part it has been proved that MST-REDUCE running time is $O(E)$.

In the k^{th} stage, suppose there are edges E_i

Since,

$$E_k \le E_{k-1} \ \forall i$$

And,

$$E_k \le E$$

Therefore,

Running time of the k stages will be k times $O(E)$

That is $O(kE)$

Step 5 of 6

e)

It can be picked up that $k = \lg \lg V$.

Now, running time of Prim's algorithm after k stages-

$$T(k) = O(KE) + O(E_k + V_k \lg V_k)$$

Or,

$$= O(kE) + O\left(E + \frac{V}{2^k} \lg \frac{V}{2^k}\right)$$

$$= O\left(kE + \frac{V}{2^k} \lg \frac{V}{2^k}\right)$$

Consider,

$$O\left(kE + \frac{V}{2^k} \lg \frac{V}{2^k}\right) = f(k)$$

Differentiating $f(k)$ and $f'(k) = 0$, the result is

$$E = V \frac{\ln V + 1 - k \lg 2}{2^k}$$

Since,

$$\ln V >> 1 - k \ln 2$$

$$2^k = \frac{V}{E} \ln V$$

$$k = \lg\left(\frac{V}{E}\ln V\right)$$

The value $T_{\min}$ will be minimum value when $E = V$

$$T_{\min} = \mathrm{O}\left(E\lg\ln V + \frac{E}{\ln V}\ln V\right)$$
$$= \mathrm{O}(E\lg\ln V)$$
$$= \mathrm{O}(E\lg\lg V)$$

Therefore, $k = \lg\lg V$ minimizes the total asymptotic running time.

Step 6 of 6

f)

Prim's algorithm with preprocessing has running time $\mathrm{O}(E\lg\lg V)$. On the other hand, Prim's algorithm without preprocessing has running time $\mathrm{O}(E + V\lg V)$.

Therefore, the former is better than latter.

Hence,

$$\mathrm{O}(E\lg\lg V)$$
$$\mathrm{O}(E + V\lg V)$$
$$E\lg\lg V < E + V\lg V$$
$$E(\lg\lg V - 1) < V\lg V$$

$$E < \frac{V\lg V}{\lg\lg V - 1}$$

Till the time the above formula is satisfied, preprocessed Prim's algorithm would win over the Prim's algorithm without preprocessing.

23-3 Bottleneck spanning tree

A ***bottleneck spanning tree*** T of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G. We say that the value of the bottleneck spanning tree is the weight of the maximum-weight edge in T.

a. Argue that a minimum spanning tree is a bottleneck spanning tree.

Part (a) shows that finding a bottleneck spanning tree is no harder than finding a minimum spanning tree. In the remaining parts, we will show how to find a bottleneck spanning tree in linear time.

b. Give a linear-time algorithm that given a graph G and an integer b, determines whether the value of the bottleneck spanning tree is at most b.

c. Use your algorithm for part (b) as a subroutine in a linear-time algorithm for the bottleneck-spanning-tree problem. (*Hint:* You may want to use a subroutine that contracts sets of edges, as in the MST-REDUCE procedure described in Problem 23-2.)

Step 1 of 4

Bottleneck spanning tree

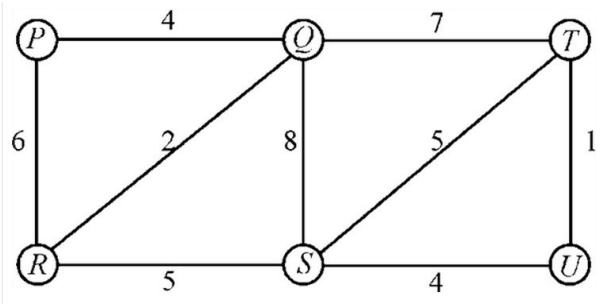
For an undirected graph $G(V,E)$, a bottleneck spanning tree is a spanning tree of G in such a manner that the weight of largest edge is minimum when compared to all spanning trees of G.

That is the maximum-weighted value edge in T is equal to the value of the bottleneck spanning tree.

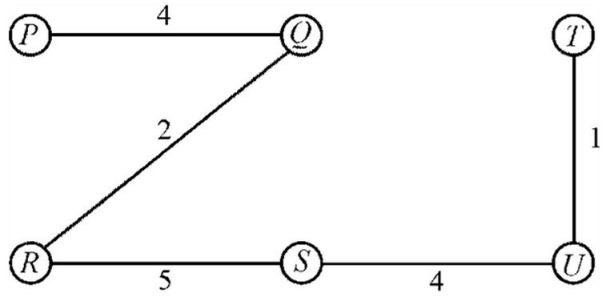
Step 2 of 4

a. Suppose that the graph $G(V,E)$ has a minimum spanning tree MST_1 which is not a BST (Bottleneck Spanning Tree).

Consider the graph given below:



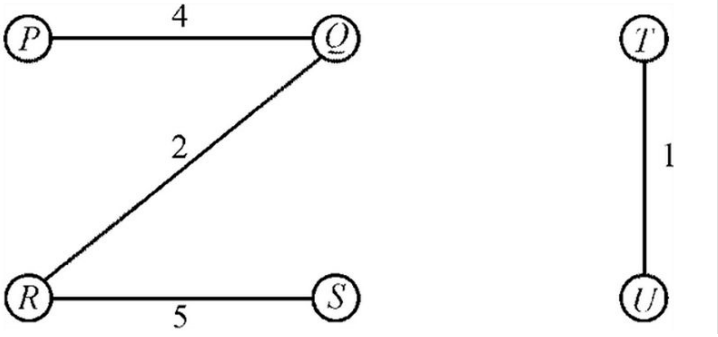
The minimum spanning tree of the graph given above will be:



Assume that the maximum weight edge in MST_1 is (u,v).

Removing this edge will result in set of disconnected vertices and that edge can be viewed as a cut of graph G.

This can be given as:



Now since this edge forms a cut therefore each minimum spanning tree must contain that edge or an edge having weight less than that of (u,v).

This leads to the contradiction.

Hence we can say a bottleneck spanning tree is a MST (Minimum Spanning Tree).

Step 3 of 4

b. Consider that $G(V,E)$ is a graph and b is an integer value.

A linear time algorithm is used to decide whether the value provided by bottleneck spanning tree is at most b.

1. **remove** all the edges from graph $G(V,E)$.

2. **for** each edge,

3. **if** (its weight $\leq b$)

add it to graph

end for

If the graph say G_b constructed in above fashion is connected then we can say that we can create a spanning tree having no edges of weight more than b.

Since we are checking for each edges of graph the above algorithm will take $\Theta(E)$ time.

Step 4 of 4

c. Initialize b with the maximum weight edge of the graph $G(V,E)$.

Now,

Check

if *b* is possible (can be checked by above algorithm)

if yes

decrement value of *b*

else

b will be the maximum weight of the bottleneck minimum spanning tree

In this case, every value of *b* is checked until the condition is not satisfied. Since it is being checked for each edges of the graph the above algorithm will take $\Theta(E)$ time.

23-4 Alternative minimum-spanning-tree algorithms

In this problem, we give pseudocode for three different algorithms. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not a minimum spanning tree. Also describe the most efficient implementation of each algorithm, whether or not it computes a minimum spanning tree.

a. MAYBE-MST-A(G, w)

```
1  sort the edges into nonincreasing order of edge weights w
2  T = E
3  for each edge e, taken in nonincreasing order by weight
4      if T − {e} is a connected graph
5          T = T − {e}
6  return T
```

b. MAYBE-MST-B(G, w)

```
1  T ← ∅
2  for each edge e, taken in arbitrary order
3       do if T ∪ {e} has no cycles
4            then T ← T ∪ e
5  return T
```

c. MAYBE-MST-C(G, w)

```
1  T ← ∅
2  for each edge e, taken in arbitrary order
3       do T ← T ∪ {e}
4          if T has a cycle c
5            then let e' be the maximum-weight edge on c
6                 T ← T − {e'}
7  return T
```

Step 1 of 5

(A) The Algorithm clearly produces a spanning tree T. Consider two vertices u and v such that edge (u,v) is in G but not in T. then $\varpi(u,v)$ is at least as large as any weight of an edge e on the path from u to v since otherwise e would have been removed earlier.

MAY BE – MST–A (G, ϖ)

1. Sort the edges into non-increasing order of edge weights ϖ.
2. T← E
3. For each edge e, taken in non-increasing order by weight
4. do if $T\{e\}$ is a connected graph then $T \leftarrow T - e$
5. return T

Step 2 of 5

The sort on the edges can be done in $O(E \lg E) = O(E \lg v)$ time. Uisng a depth first search to determine connectivity in line 4 the total running time is $O(E \lg v + E(V+E)) = O(E^2 + Ev)$. The running time can be reduced using results on the "decremental connectivity problem".

Step 3 of 5

(B) MAY BE – MST - B (G, ϖ)

1) $T \leftarrow \varnothing$
2) For each edge e, taken in arbitrary order
3) do if $T \cup \{e\}$ has no cycles.
4) then $T \leftarrow T \cup e$
5) return T

This Algorithm simply produces a spanning tree, but clearly does not guarantee that the tree is of minimum weight. Line 2 and 3 in the Algorithm can be implemented using disjoint set operations yielding a total running the of $O(E\alpha(V))$.

Step 4 of 5

(D) MAY BE – MST - C(G, ω)

1) $T \leftarrow d$
2) For each edge e, taken in arbitrary order
3) do $T \leftarrow T \cup \{e\}$
4) if T has a cycle C
5) then let e^1 be the max-weight on edge on e
6) $T \leftarrow T - \{e^1\}$
7) return T

The Algorithm produces a spanning tree T. Observe that on completion of any edge (u,v) in G but not in T will be a maximum weight edge on the path from u to v.

Step 5 of 5

We can implement the Algorithm as follows:

- Line 3 can be done in $O(1)$ by appending the edge to the proper adjacency list.
- Line 4 can be done in $O(V)$ time using a depth-first search.
- Since the number of edges in T is at most $O(V)$ time 5 can be done using a linear search on the edges in $O(V)$ time.
- Line 6 can be done using a search on the adjacency list taking at most $O(V)$ time.

The for loop iterates at most $O(E)$ times and the total running time is thus $O(EV)$.