Analysis of Algorithms

Minimum Spanning Trees

Andres Mendez-Vazquez

November 8, 2015

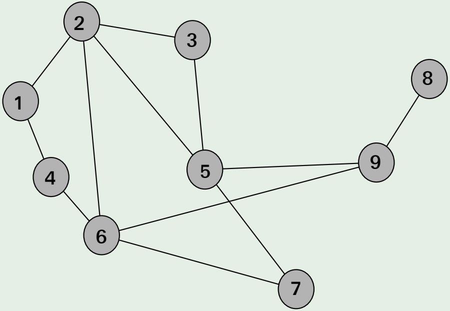
Outline

Outline

Cinvestav

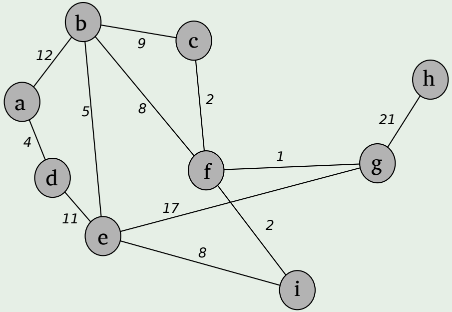
Originally

We had a Graph without weights



Then

Now, we have have weights

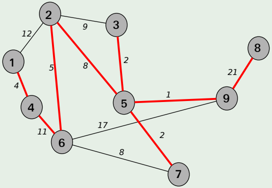


Finally, the optimization problem

We want to find

$$\min_{T} \sum_{(u,v)\in T} w(u,v)$$

Where $T \subseteq E$ such that T is acyclic and connects all the vertices.



This problem is called

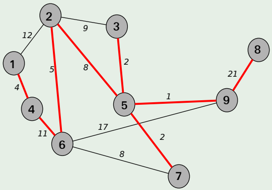
The minimum spanning tree problem

Finally, the optimization problem

We want to find

$$\min_{T} \sum_{(u,v)\in T} w(u,v)$$

Where $T \subseteq E$ such that T is acyclic and connects all the vertices.



This problem is called

The minimum spanning tree problem

When do you need minimum spanning trees?

In power distribution

We want to connect points x and y with the minimum amount of cable.

In a wireless network

Given a collection of mobile beacons we want to maintain the minimum connection overhead between all of them.

When do you need minimum spanning trees?

In power distribution

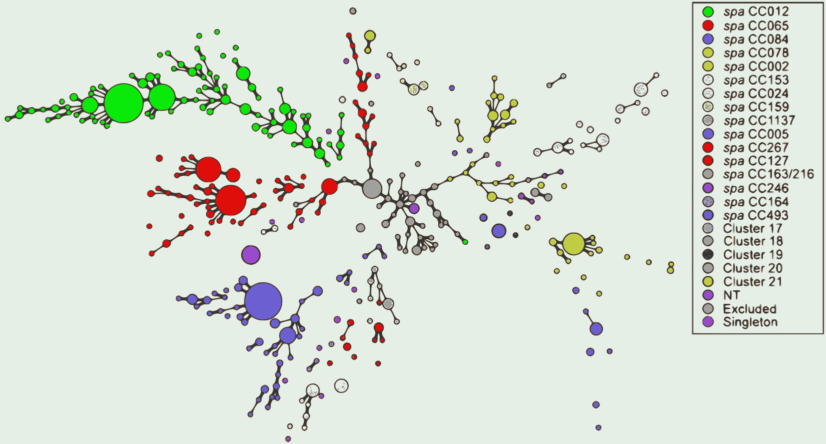
We want to connect points x and y with the minimum amount of cable.

In a wireless network

Given a collection of mobile beacons we want to maintain the minimum connection overhead between all of them.

Some Applications

Tracking the Genetic Variance of Age-Gender-Associated Staphylococcus Aureus



Some Applications

What?

Urban Tapestries is an interactive location-based wireless application allowing users to access and publish location-specific multimedia content.

- Using MST we can create paths for public multimedia shows that are no too exhausting

These models can be seen as

Connected, undirected graphs $G = (V, E)$

- E is the set of possible connections between pairs of beacons.
- Each of the this edges (u, v) has a weight $w(u, v)$ specifying the cost of connecting u and v.

These models can be seen as

Connected, undirected graphs $G = (V, E)$

- E is the set of possible connections between pairs of beacons.
- Each of the this edges (u, v) has a weight $w(u, v)$ specifying the cost of connecting u and v.

Outline

Cinvestav

Growing a Minimum Spanning Tree

There are two classic algorithms, Prim and Kruskal

Both algorithms Kruskal and Prim use a greedy approach.

Growing a Minimum Spanning Tree

There are two classic algorithms, Prim and Kruskal

Both algorithms Kruskal and Prim use a greedy approach.

Basic greedy idea

- Prior to each iteration, A is a subset of some minimum spanning tree.
- At each step, we determine an edge (u, v) that can be added to A such that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.

Growing a Minimum Spanning Tree

There are two classic algorithms, Prim and Kruskal

Both algorithms Kruskal and Prim use a greedy approach.

Basic greedy idea

- Prior to each iteration, A is a subset of some minimum spanning tree.
- At each step, we determine an edge (u, v) that can be added to A such that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.

Generic minimum spanning tree algorithm

A Generic Code

Generic-MST(G, w)

1. $A = \emptyset$
2. while A does not form a spanning tree
3. do find an edge (u, v) that is safe for A
4. $A = A \cup \{(u, v)\}$
5. return A

Generic minimum spanning tree algorithm

A Generic Code

Generic-MST(G, w)

1. $A = \emptyset$
2. while A does not form a spanning tree
3. do find an edge (u, v) that is safe for A
4. $A = A \cup \{(u, v)\}$
5. return A

This has the following loop invariance

Initialization: Line 1 A trivially satisfies.

Maintenance: The loop only adds safe edges.

Termination: The final A contains all the edges in a minimum spanning tree.

Generic minimum spanning tree algorithm

A Generic Code

Generic-MST(G, w)

1. $A = \emptyset$
2. while A does not form a spanning tree
3.   do find an edge (u, v) that is safe for A
4. $A = A \cup \{(u, v)\}$
5. return A

This has the following loop invariance

Initialization: Line 1 A trivially satisfies.

Maintenance: The loop only adds safe edges.

Termination: The final A contains all the edges in a minimum spanning tree.

Generic minimum spanning tree algorithm

A Generic Code

Generic-MST(G, w)

1. $A = \emptyset$
2. while A does not form a spanning tree
3.   do find an edge (u, v) that is safe for A
4. $A = A \cup \{(u, v)\}$
5. return A

This has the following loop invariance

Initialization: Line 1 A trivially satisfies.

Maintenance: The loop only adds safe edges.

Termination: The final A contains all the edges in a minimum spanning tree.

Some basic definitions for the Greedy Choice

A cut $(S, V - S)$ is a partition of V

- Then (u, v) in E crosses the cut $(S, V - S)$ if one end point is in S and the other is in $V - S$.
- We say that a cut respects A if no edge in A crosses the cut.
- A light edge is an edge crossing the cut with minimum weight with respect to the other edges crossing the cut.

Some basic definitions for the Greedy Choice

A cut $(S, V - S)$ is a partition of V

- Then (u, v) in E crosses the cut $(S, V - S)$ if one end point is in S and the other is in $V - S$.
- We say that a cut respects A if no edge in A crosses the cut.
- A light edge is an edge crossing the cut with minimum weight with respect to the other edges crossing the cut.

Cinvestav

Some basic definitions for the Greedy Choice

A cut $(S, V - S)$ is a partition of V

- Then (u, v) in E crosses the cut $(S, V - S)$ if one end point is in S and the other is in $V - S$.
- We say that a cut respects A if no edge in A crosses the cut.
- A light edge is an edge crossing the cut with minimum weight with respect to the other edges crossing the cut.

Cinvestav

Outline

Cinvestav

The Greedy Choice

Remark

The following algorithms are based in the Greedy Choice.

Which Greedy Choice?

The way we add edges to the set of edges belonging to the Minimum Spanning Trees.

They are known as

Safe Edges

The Greedy Choice

Remark

The following algorithms are based in the Greedy Choice.

Which Greedy Choice?

The way we add edges to the set of edges belonging to the Minimum Spanning Trees.

They are known as

Safe Edges

The Greedy Choice

Remark

The following algorithms are based in the Greedy Choice.

Which Greedy Choice?

The way we add edges to the set of edges belonging to the Minimum Spanning Trees.

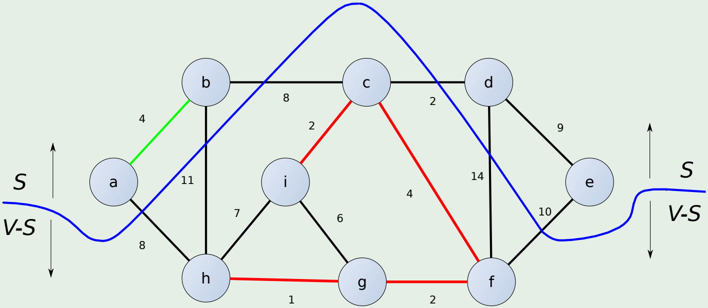
They are known as

Safe Edges

Recognizing safe edges

Theorem for Recognizing Safe Edges (23.1)

Let $G = (V, E)$ be a connected, undirected graph with weights w defined on E. Let $A \subseteq E$ that is included in a MST for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.



Observations

Notice that

- **At any point in the execution of the algorithm the graph $G_A = (V, A)$ is a forest, and each of the connected components of G_A is a tree.**

Thus

- Any safe edge (u, v) for A connects distinct components of G_A, since $A \cup \{(u, v)\}$ must be acyclic.

Observations

Notice that

- **At any point in the execution of the algorithm the graph $G_A = (V, A)$ is a forest, and each of the connected components of G_A is a tree.**

Thus

- Any safe edge (u, v) for A connects distinct components of G_A, since $A \cup \{(u, v)\}$ must be acyclic.

Cinvestav

The basic corollary

Corollary 23.2

Let $G = (V, E)$ be a connected, undirected graph with 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, and let $C = (V_c, E_c)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.

Proof

The cut $(V_c, V - V_c)$ respects A, and (u, v) is a light edge for this cut. Therefore, (u, v) is safe for A.

The basic corollary

Corollary 23.2

Let $G = (V, E)$ be a connected, undirected graph with 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, and let $C = (V_c, E_c)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A, then (u, v) is safe for A.

Proof

The cut $(V_c, V - V_c)$ respects A, and (u, v) is a light edge for this cut. Therefore, (u, v) is safe for A.

Outline

Cinvestav

Outline

Cinvestav

Kruskal's Algorithm

Algorithm

MST-KRUSKAL(G, w)

1. $A = \emptyset$
2. for each vertex $v \in V[G]$
3. do Make-Set
4. sort the edges of E into non-decreasing order by weight w
5. for each edge $(u, v) \in E$ taken in non-decreasing order by weight
6. do if $FIND - SET(u) \neq FIND - SET(v)$
7. then $A = A \cup \{(u, v)\}$
8. Union(u,v)
9. return A

Kruskal's Algorithm

Algorithm

MST-KRUSKAL(G, w)

1. $A = \emptyset$
2. for each vertex $v \in V[G]$
3. do Make-Set
4. sort the edges of E into non-decreasing order by weight w
5. for each edge $(u, v) \in E$ taken in non-decreasing order by weight
6. do if $FIND - SET(u) \neq FIND - SET(v)$
7. then $A = A \cup \{(u, v)\}$
8. Union(u,v)
9. return A

Kruskal's Algorithm

Algorithm

MST-KRUSKAL(G, w)

1. $A = \emptyset$
2. for each vertex $v \in V[G]$
3. do Make-Set
4. sort the edges of E into non-decreasing order by weight w
5. for each edge $(u, v) \in E$ taken in non-decreasing order by weight
6. do if $FIND - SET(u) \neq FIND - SET(v)$
7. then $A = A \cup \{(u, v)\}$
8. Union(u,v)
9. return A

Kruskal's Algorithm

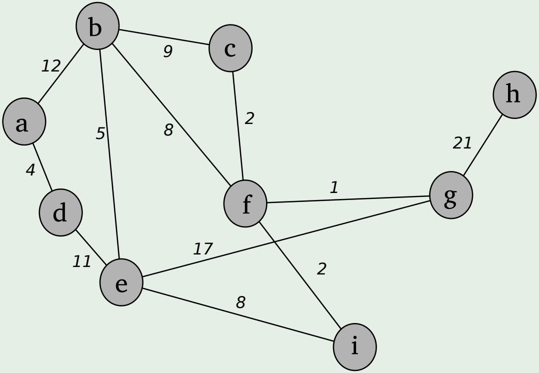
Algorithm

MST-KRUSKAL(G, w)

```
A = ∅
for each vertex v ∈ V[G]
        do Make-Set
sort the edges of E into non-decreasing order by weight w
for each edge (u, v) ∈ E taken in non-decreasing order by weight
        do if FIND − SET(u) ≠ FIND − SET(v)
                then A = A ∪ {(u, v)}
                        Union(u,v)
return A
```

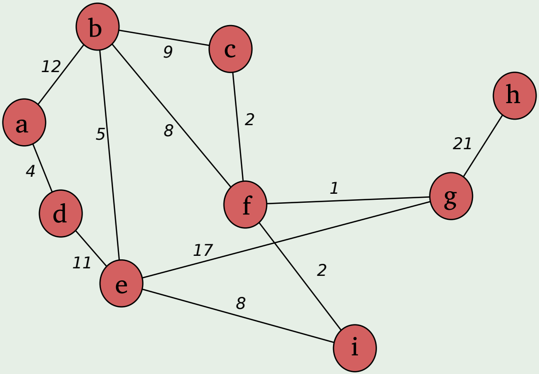
Let us run the Algorithm

We have as an input the following graph



Let us run the Algorithm

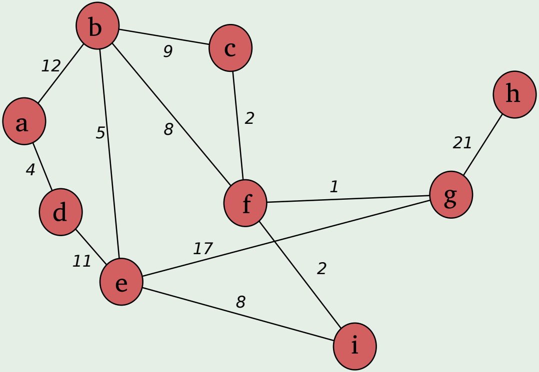
1^{st} step everybody is a set!!!



Let us run the Algorithm

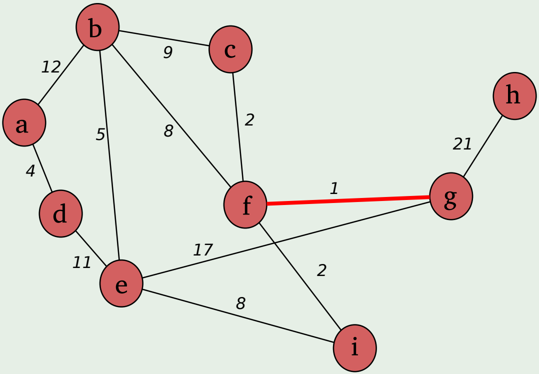
Given (f, g) with weight 1

Question: $FIND - SET(f) \neq FIND - SET(g)$?



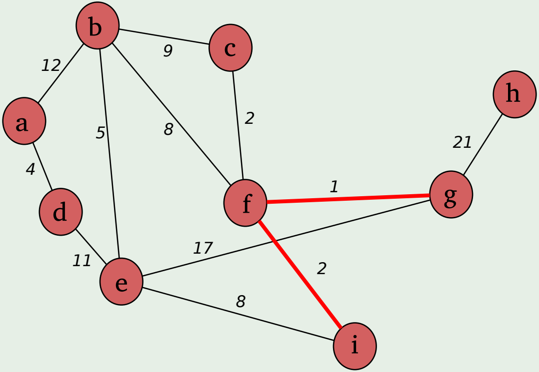
Let us run the Algorithm

Then $A = A \cup \{(f, g)\}$, next $FIND-SET(f) \neq FIND-SET(i)$?



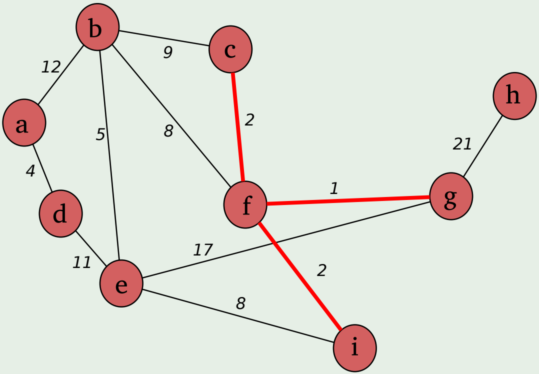
Let us run the Algorithm

Then $A = A \cup \{(f, i)\}$, next $FIND-SET(c) \neq FIND-SET(f)$?



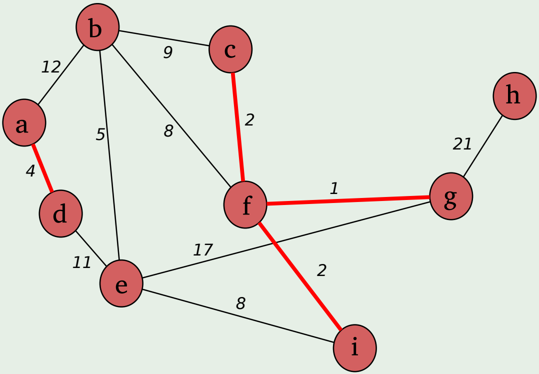
Let us run the Algorithm

Then $A = A \cup \{(c, f)\}$, next $FIND-SET(a) \neq FIND-SET(d)$?



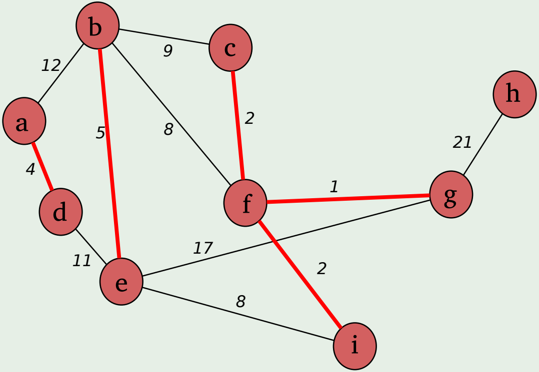
Let us run the Algorithm

Then $A = A \cup \{(a,d)\}$, next $FIND-SET(b) \neq FIND-SET(e)$?



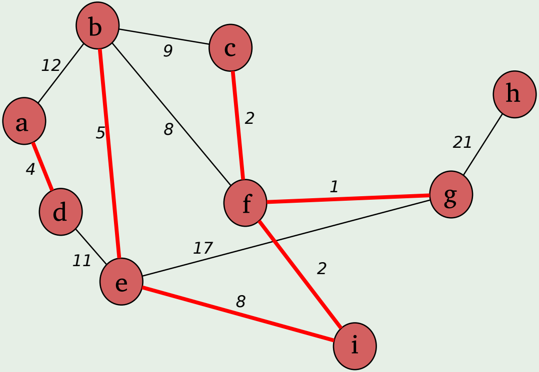
Let us run the Algorithm

Then $A = A \cup \{(b,e)\}$, next $FIND-SET(e) \neq FIND-SET(i)$?



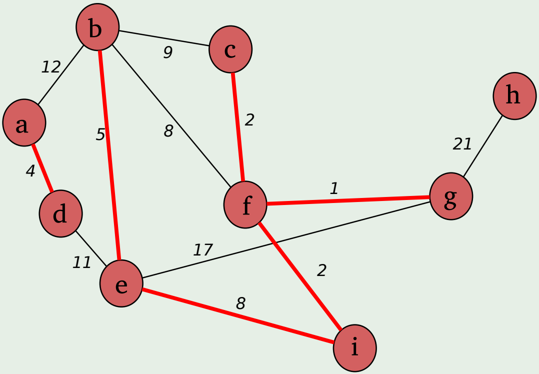
Let us run the Algorithm

Then $A = A \cup \{(e, i)\}$, next $FIND-SET(b) \neq FIND-SET(f)$?



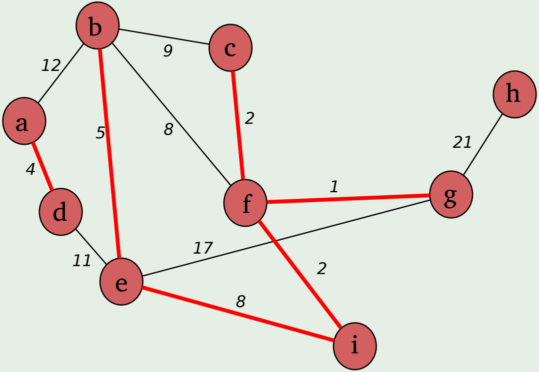
Let us run the Algorithm

Then $A = A$, next $FIND-SET(b) \neq FIND-SET(c)$?



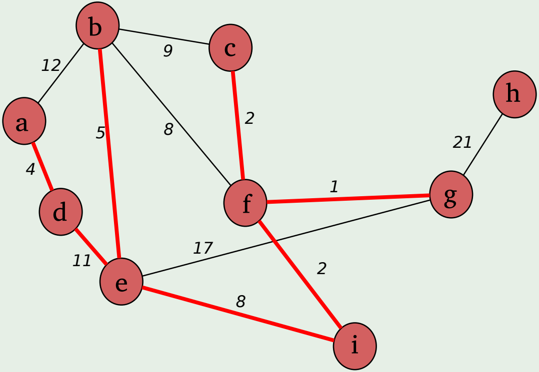
Let us run the Algorithm

Then $A = A$, next $FIND - SET(d) \neq FIND - SET(e)$?



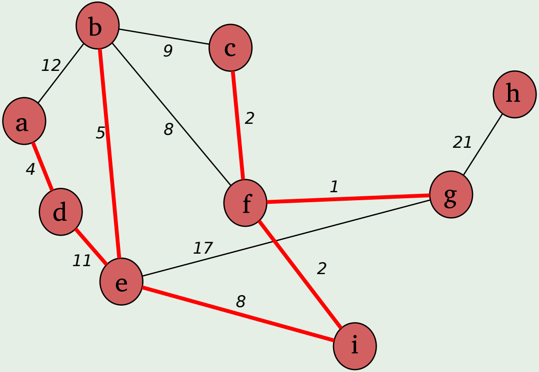
Let us run the Algorithm

Then $A = A \cup \{(d, e)\}$, next $FIND - SET(a) \neq FIND - SET(b)$?



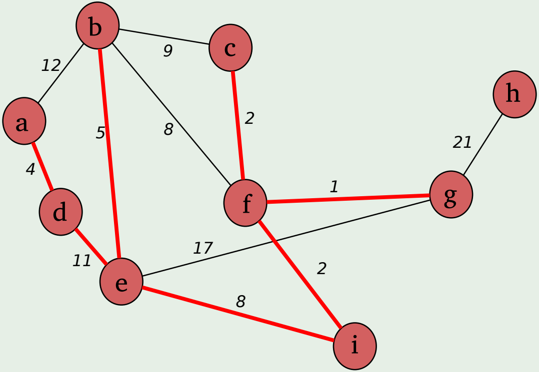
Let us run the Algorithm

Then $A = A$, next $FIND-SET(e) \neq FIND-SET(g)$?



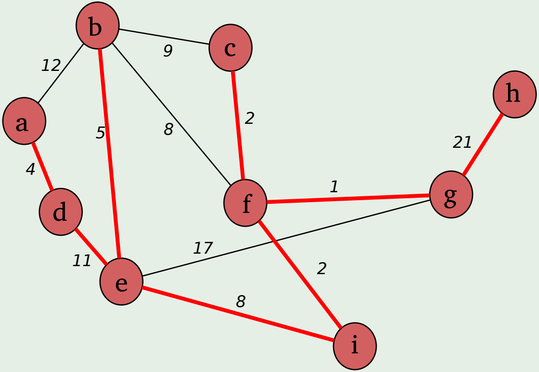
Let us run the Algorithm

Then $A = A$, next $FIND-SET(g) \neq FIND-SET(h)$?



Let us run the Algorithm

Then $A = A \cup \{(g,h)\}$



Kruskal's Algorithm

Algorithm

MST-KRUSKAL(G, w)

$A = \emptyset$
for each vertex $v \in V[G]$
    do Make-Set
sort the edges of E into non-decreasing order by weight w
for each edge $(u, v) \in E$ taken in non-decreasing order by weight
    do if $FIND - SET(u) \neq FIND - SET(v)$
        then $A = A \cup \{(u, v)\}$
            Union(u,v)
return A

Complexity

Explanation

- Line 1. Initializing the set A takes $O(1)$ time.
- Line 2. Sorting the edges in line 4 takes $O(E \log E)$.
- Lines 5 to 8. The for loop performs:
  - $O(E)$ FIND-SET and UNION operations.
  - Along with the $|V|$ MAKE-SET operations that take $O((V+E)\alpha(V))$, where α is the pseudoinverse of the Ackermann's function.

Complexity

Explanation

- Line 1. Initializing the set A takes $O(1)$ time.
- Line 2. Sorting the edges in line 4 takes $O(E \log E)$.
- Lines 5 to 8. The for loop performs:
  - $O(E)$ FIND-SET and UNION operations.
  - Along with the $|V|$ MAKE-SET operations that take $O((V+E)\alpha(V))$, where α is the pseudoinverse of the Ackermann's function.

Thus

- Given that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time and $\alpha(|V|) = O(\log V) = O(\log E)$.
- The total running time of Kruskal's algorithm is $O(E \log E)$, but observing that $|E| < |V|^2 \longrightarrow \log|E| < 2\log|V|$, we have that $\log|E| = O(\log V)$, and so we can restate the running time of the algorithm as $O(E \log V)$.

Complexity

Explanation

- Line 1. Initializing the set A takes $O(1)$ time.
- Line 2. Sorting the edges in line 4 takes $O(E \log E)$.
- Lines 5 to 8. The for loop performs:
  - $O(E)$ FIND-SET and UNION operations.
  - Along with the $|V|$ MAKE-SET operations that take $O((V+E)\alpha(V))$, where α is the pseudoinverse of the Ackermann's function.

Thus

- Given that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time and $\alpha(|V|) = O(\log V) = O(\log E)$.
- The total running time of Kruskal's algorithm is $O(E \log E)$, but observing that $|E| < |V|^2 \longrightarrow \log |E| < 2 \log |V|$, we have that $\log |E| = O(\log V)$, and so we can restate the running time of the algorithm as $O(E \log V)$.

Complexity

Explanation

- Line 1. Initializing the set A takes $O(1)$ time.
- Line 2. Sorting the edges in line 4 takes $O(E \log E)$.
- Lines 5 to 8. The for loop performs:
  - $O(E)$ FIND-SET and UNION operations.
  - Along with the $|V|$ MAKE-SET operations that take $O((V+E)\alpha(V))$, where α is the pseudoinverse of the Ackermann's function.

Thus

- Given that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time and $\alpha(|V|) = O(\log V) = O(\log E)$.
- The total running time of Kruskal's algorithm is $O(E \log E)$, but observing that $|E| < |V|^2 \longrightarrow \log|E| < 2\log|V|$, we have that $\log|E| = O(\log V)$, and so we can restate the running time of the algorithm as $O(E \log V)$.

Complexity

Explanation

- Line 1. Initializing the set A takes $O(1)$ time.
- Line 2. Sorting the edges in line 4 takes $O(E \log E)$.
- Lines 5 to 8. The for loop performs:
  - $O(E)$ FIND-SET and UNION operations.
  - Along with the $|V|$ MAKE-SET operations that take $O((V+E)\alpha(V))$, where α is the pseudoinverse of the Ackermann's function.

Thus

- Given that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time and $\alpha(|V|) = O(\log V) = O(\log E)$.
- The total running time of Kruskal's algorithm is $O(E \log E)$, but observing that $|E| < |V|^2 \longrightarrow \log |E| < 2 \log |V|$, we have that $\log |E| = O(\log V)$, and so we can restate the running time of the algorithm as $O(E \log V)$.

Complexity

Explanation

- Line 1. Initializing the set A takes $O(1)$ time.
- Line 2. Sorting the edges in line 4 takes $O(E \log E)$.
- Lines 5 to 8. The for loop performs:
  - $O(E)$ FIND-SET and UNION operations.
  - Along with the $|V|$ MAKE-SET operations that take $O((V+E)\alpha(V))$, where α is the pseudoinverse of the Ackermann's function.

Thus

- Given that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time and $\alpha(|V|) = O(\log V) = O(\log E)$.
- The total running time of Kruskal's algorithm is $O(E \log E)$, but observing that $|E| < |V|^2 \longrightarrow \log |E| < 2 \log |V|$, we have that $\log |E| = O(\log V)$, and so we can restate the running time of the algorithm as $O(E \log V)$.

Complexity

Explanation

- Line 1. Initializing the set A takes $O(1)$ time.
- Line 2. Sorting the edges in line 4 takes $O(E \log E)$.
- Lines 5 to 8. The for loop performs:
  - $O(E)$ FIND-SET and UNION operations.
  - Along with the $|V|$ MAKE-SET operations that take $O((V+E)\alpha(V))$, where α is the pseudoinverse of the Ackermann's function.

Thus

- Given that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time and $\alpha(|V|) = O(\log V) = O(\log E)$.
- The total running time of Kruskal's algorithm is $O(E \log E)$, but observing that $|E| < |V|^2 \longmapsto \log |E| < 2 \log |V|$, we have that $\log |E| = O(\log V)$, and so we can restate the running time of the algorithm as $O(E \log V)$.

Outline

Cinvestav

Prim's Algorithm

Prim's algorithm operates much like Dijkstra's algorithm

- The tree starts from an arbitrary root vertex r.
- At each step, a light edge is added to the tree A that connects A to an isolated vertex of $G_A = (V, A)$.
- When the algorithm terminates, the edges in A form a minimum spanning tree.

Prim's Algorithm

Prim's algorithm operates much like Dijkstra's algorithm

- The tree starts from an arbitrary root vertex r.
- At each step, a light edge is added to the tree A that connects A to an isolated vertex of $G_A = (V, A)$.
- When the algorithm terminates, the edges in A form a minimum spanning tree.

Prim's Algorithm

Prim's algorithm operates much like Dijkstra's algorithm

- The tree starts from an arbitrary root vertex r.
- At each step, a light edge is added to the tree A that connects A to an isolated vertex of $G_A = (V, A)$.
- When the algorithm terminates, the edges in A form a minimum spanning tree.

Problem

Important

In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A.

For this, we use a min-priority queue Q

During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute.

There is a field key for every vertex v

- It is the minimum weight of any edge connecting v to a vertex in the minimum spanning tree (THE LIGHT EDGE!!!).
- By convention, $v.key = \infty$ if there is no such edge.

Problem

Important

In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A.

For this, we use a min-priority queue Q

During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute.

There is a field key for every vertex v

- It is the minimum weight of any edge connecting v to a vertex in the minimum spanning tree (THE LIGHT EDGE!!!).
- By convention, $v.key = \infty$ if there is no such edge.

Problem

Important

In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in A.

For this, we use a min-priority queue Q

During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a key attribute.

There is a field key for every vertex v

- It is the minimum weight of any edge connecting v to a vertex in the minimum spanning tree (THE LIGHT EDGE!!!).
- By convention, $v.key = \infty$ if there is no such edge.

The algorithm

Pseudo-code

MST-PRIM(G, w, r)

```
for each u ∈ V[G]
        u.key = ∞
        u.π = NIL
r.key = 0
Q = V[G]
while Q ≠ ∅
        u = Extract-Min(Q)
        for each v ∈ Adj[u]
                if v ∈ Q and w(u, v) < v.key
                        π[v] = u
                        v.key = w(u, v) ▷ an implicit decrease key in Q
```

The algorithm

Pseudo-code

MST-PRIM(G, w, r)

for each $u \in V[G]$

  $u.key = \infty$

  $u.\pi = NIL$

$r.key = 0$

$Q = V[G]$

while $Q \neq \emptyset$

  $u =$Extract-Min(Q)

  for each $v \in Adj[u]$

    if $v \in Q$ and $w(u, v) < v.key$

      $\pi[v] = u$

      $v.key = w(u, v)$▷an implicit decrease key in Q

The algorithm

Pseudo-code

MST-PRIM(G, w, r)

for each $u \in V[G]$

$\quad u.key = \infty$

$\quad u.\pi = NIL$

$r.key = 0$

$Q = V[G]$

while $Q \neq \emptyset$

$\quad u =$Extract-Min(Q)

$\quad$ for each $v \in Adj[u]$

$\quad\quad$ if $v \in Q$ and $w(u, v) < v.key$

$\quad\quad\quad \pi[v] = u$

$\quad\quad\quad v.key = w(u, v)$▷an implicit decrease key in Q

The algorithm

Pseudo-code

MST-PRIM(G, w, r)

for each $u \in V[G]$
  $u.key = \infty$
  $u.\pi = NIL$
$r.key = 0$
$Q = V[G]$
while $Q \neq \emptyset$
  $u =$Extract-Min(Q)
  for each $v \in Adj[u]$
    if $v \in Q$ and $w(u, v) < v.key$
      $\pi[v] = u$
      $v.key = w(u, v)$ ▷an implicit decrease key in Q

Explanation

Observations

1. $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $\pi[v] \neq NIL$, then $key[v] < \infty$ and $key[v]$ is the weight of a light edge $(v, \pi[v])$ connecting v to some vertex already placed into the minimum spanning tree.

Explanation

Observations

1. $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $\pi[v] \neq NIL$, then $key[v] < \infty$ and $key[v]$ is the weight of a light edge $(v, \pi[v])$ connecting v to some vertex already placed into the minimum spanning tree.

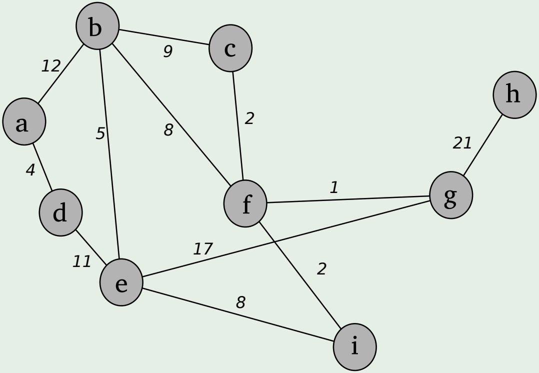
Explanation

Observations

1. $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $\pi[v] \neq NIL$, then $key[v] < \infty$ and $key[v]$ is the weight of a light edge $(v, \pi[v])$ connecting v to some vertex already placed into the minimum spanning tree.

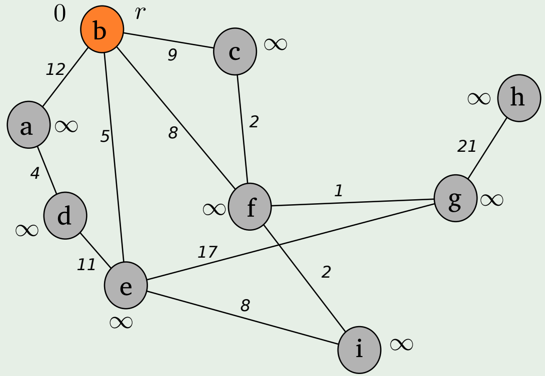
Let us run the Algorithm

We have as an input the following graph



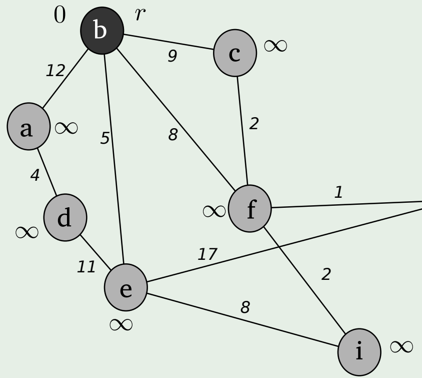
Let us run the Algorithm

Select r =b



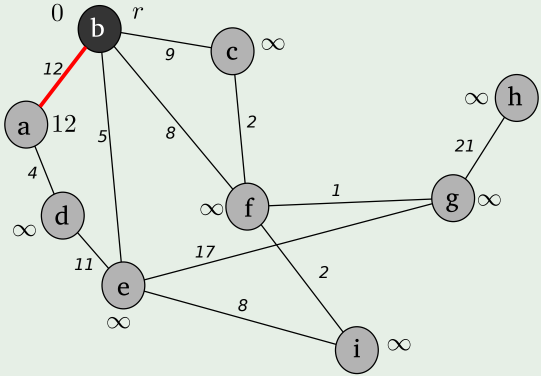
Let us run the Algorithm

Extract b from the priority queue Q



Let us run the Algorithm

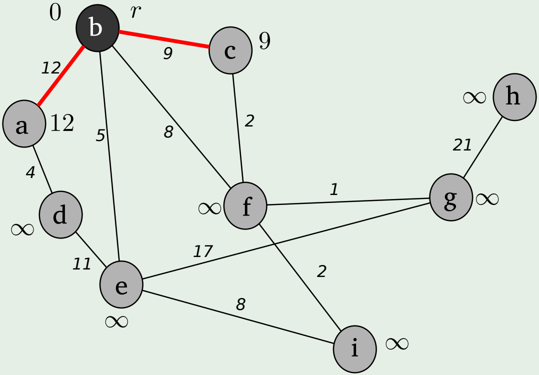
Update the predecessor of a and its key to 12 from ∞



Note: The RED color represent the field $\pi[v]$

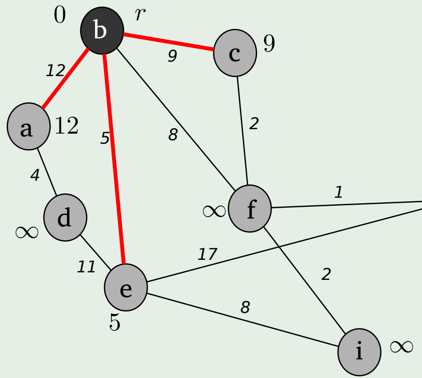
Let us run the Algorithm

Update the predecessor of c and its key to 9 from ∞



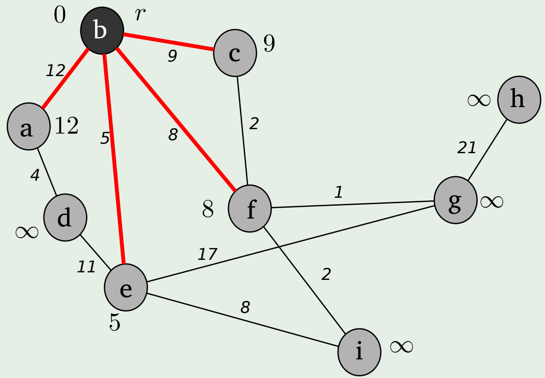
Let us run the Algorithm

Update the predecessor of e and its key to 5 from ∞



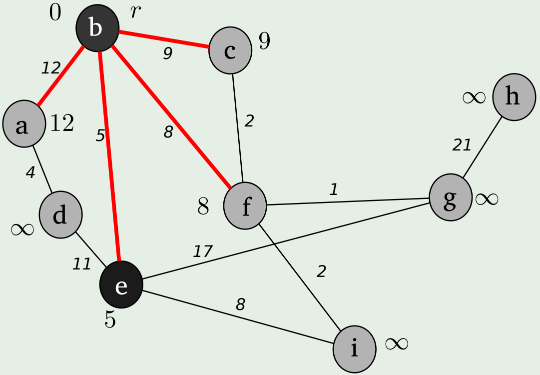
Let us run the Algorithm

Update the predecessor of f and its key to 8 from ∞



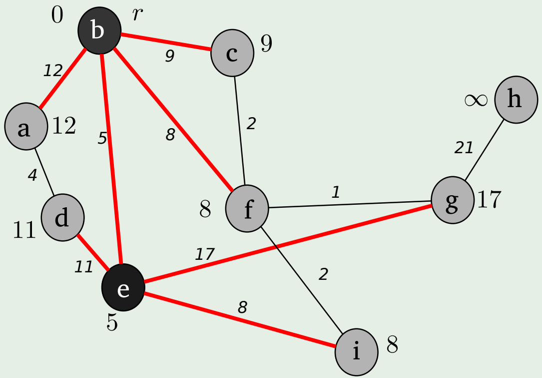
Let us run the Algorithm

Extract e, then update adjacent vertices



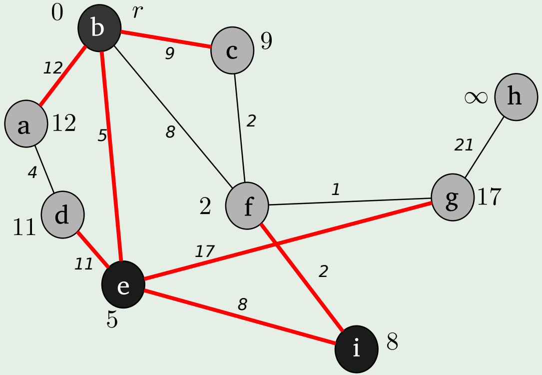
Let us run the Algorithm

Extract i from the priority queue Q



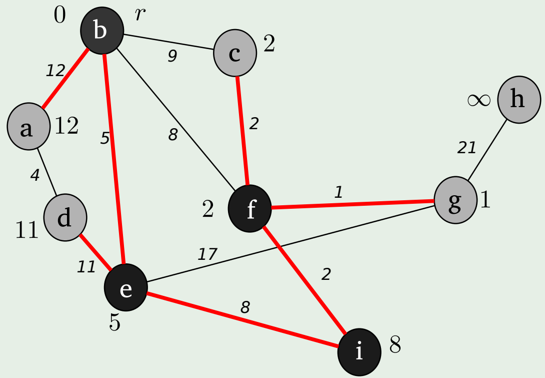
Let us run the Algorithm

Update adjacent vertices



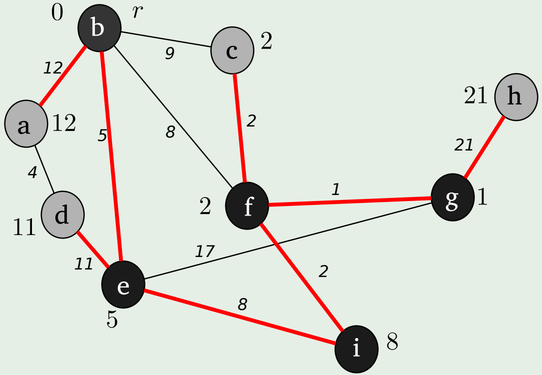
Let us run the Algorithm

Extract f and update adjacent vertices



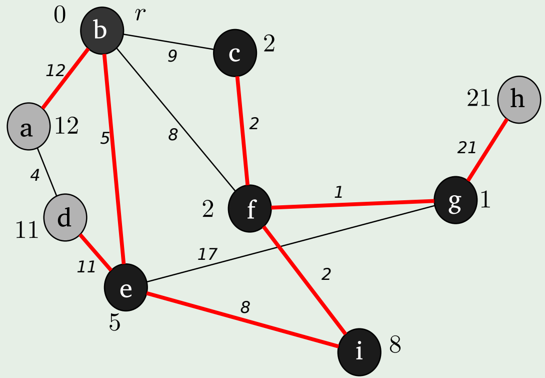
Let us run the Algorithm

Extract g and update



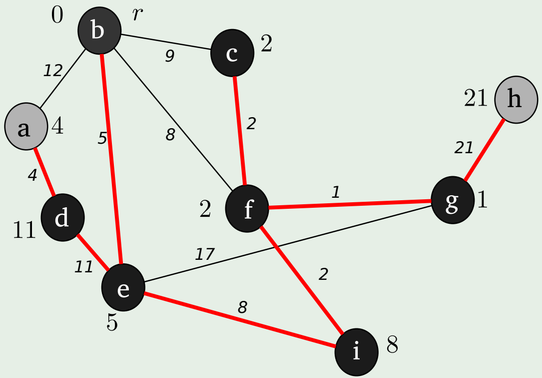
Let us run the Algorithm

Extract c and no update



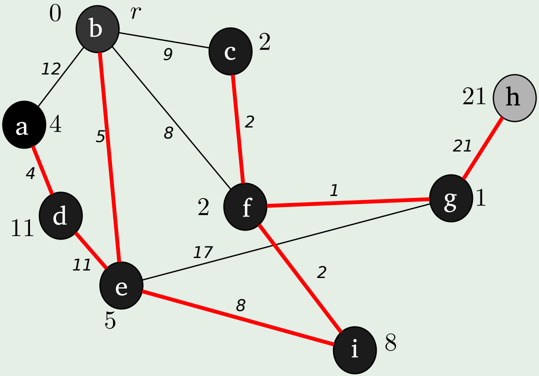
Let us run the Algorithm

Extract d and update key at 1



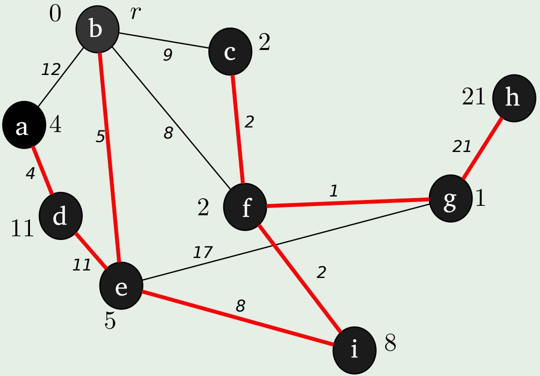
Let us run the Algorithm

Extract *a* and no update



Let us run the Algorithm

Extract h



Complexity I

Complexity analysis

- The performance of Prim's algorithm depends on how we implement the min-priority queue Q.
- If Q is a binary min-heap, BUILD-MIN-HEAP procedure to perform the initialization in lines 1 to 5 will run in $O(|V|)$ time.
- The body of the while loop is executed $|V|$ times, and EXTRACT-MIN operation takes $O(\log V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \log V)$.
- The for loop in lines 8 to 11 is executed $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$.

Complexity I

Complexity analysis

- The performance of Prim's algorithm depends on how we implement the min-priority queue Q.
- If Q is a binary min-heap, BUILD-MIN-HEAP procedure to perform the initialization in lines 1 to 5 will run in $O(|V|)$ time.
- The body of the while loop is executed $|V|$ times, and EXTRACT-MIN operation takes $O(\log V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \log V)$.
- The for loop in lines 8 to 11 is executed $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$.

Complexity I

Complexity analysis

- The performance of Prim's algorithm depends on how we implement the min-priority queue Q.
- If Q is a binary min-heap, BUILD-MIN-HEAP procedure to perform the initialization in lines 1 to 5 will run in $O(|V|)$ time.
- The body of the while loop is executed $|V|$ times, and EXTRACT-MIN operation takes $O(\log V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \log V)$.
- The for loop in lines 8 to 11 is executed $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$.

Complexity I

Complexity analysis

- The performance of Prim's algorithm depends on how we implement the min-priority queue Q.
- If Q is a binary min-heap, BUILD-MIN-HEAP procedure to perform the initialization in lines 1 to 5 will run in $O(|V|)$ time.
- The body of the while loop is executed $|V|$ times, and EXTRACT-MIN operation takes $O(\log V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \log V)$.
- The for loop in lines 8 to 11 is executed $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$.

Complexity II

Complexity analysis (continuation)

- Within the for loop, the test for membership in Q in line 9 can be implemented in constant time.
- The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which can be implemented in a binary min-heap in $O(\log V)$ time. Thus, the total time for Prim's algorithm is:

$$O(V \log V + E \log V) = O(E \log V)$$

Complexity II

Complexity analysis (continuation)

- Within the for loop, the test for membership in Q in line 9 can be implemented in constant time.
- The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which can be implemented in a binary min-heap in $O(\log V)$ time. Thus, the total time for Prim's algorithm is:

$$O(V \log V + E \log V) = O(E \log V)$$

If you use Fibonacci Heaps

Complexity analysis

- EXTRACT-MIN operation in $O(\log V)$ amortized time.
- DECREASE-KEY operation (to implement line 11) in $O(1)$ amortized time.
- If we use a Fibonacci Heap to implement the min-priority queue Q we get a running time of $O(E + V \log V)$.

Cinvestav

If you use Fibonacci Heaps

Complexity analysis

- EXTRACT-MIN operation in $O(\log V)$ amortized time.
- DECREASE-KEY operation (to implement line 11) in $O(1)$ amortized time.
- If we use a Fibonacci Heap to implement the min-priority queue Q we get a running time of $O(E + V \log V)$.

Cinvestav

If you use Fibonacci Heaps

Complexity analysis

- EXTRACT-MIN operation in $O(\log V)$ amortized time.
- DECREASE-KEY operation (to implement line 11) in $O(1)$ amortized time.
- If we use a Fibonacci Heap to implement the min-priority queue Q we get a running time of $O(E + V \log V)$.

Cinvestav

Outline

Cinvestav

Faster Algorithms

Linear Time Algorithms

- Karger, Klein & Tarjan (1995) proposed a linear time randomized algorithm.
- The Fastest $(O(E\alpha(E, V)))$ by Bernard Chazelle (2000) is based on the soft heap, an approximate priority queue.
  - Chazelle has also written essays about music and politics

Linear-time algorithms in special cases

If the graph is dense $\left(\text{i.e. } \log\log\log V \leq \frac{E}{V}\right)$, then a deterministic algorithm by Fredman and Tarjan finds the MST in time $O(E)$.

Faster Algorithms

Linear Time Algorithms

- Karger, Klein & Tarjan (1995) proposed a linear time randomized algorithm.
- The Fastest $(O(E\alpha(E, V)))$ by Bernard Chazelle (2000) is based on the soft heap, an approximate priority queue.
  - Chazelle has also written essays about music and politics

Linear-time algorithms in special cases

If the graph is dense $\left(\text{ i.e. } \log\log\log V \leq \frac{E}{V}\right)$, then a deterministic algorithm by Fredman and Tarjan finds the MST in time $O(E)$.

Cinvestav

Outline

Cinvestav

Applications

Minimum spanning trees have direct applications in the design of networks

- Telecommunications networks
- Transportation networks
- Water supply networks
- Electrical grids

Applications

Minimum spanning trees have direct applications in the design of networks

- Telecommunications networks
- Transportation networks
- Water supply networks
- Electrical grids

As a subroutine in

- Machine Learning/Big Data Cluster Analysis
- Network Communications are using Spanning Tree Protocol (STP)
- Image registration and segmentation
- Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.
- Etc

Applications

Minimum spanning trees have direct applications in the design of networks

- Telecommunications networks
- Transportation networks
- Water supply networks
- Electrical grids

As a subroutine in

- Machine Learning/Big Data Cluster Analysis
- Network Communications are using Spanning Tree Protocol (STP)
- Image registration and segmentation
- Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.
- Etc

Applications

Minimum spanning trees have direct applications in the design of networks

- Telecommunications networks
- Transportation networks
- Water supply networks
- Electrical grids

As a subroutine in

- Machine Learning/Big Data Cluster Analysis
- Network Communications are using Spanning Tree Protocol (STP)
- Image registration and segmentation
- Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.
- Etc

Applications

Minimum spanning trees have direct applications in the design of networks

- Telecommunications networks
- Transportation networks
- Water supply networks
- Electrical grids

As a subroutine in

- Machine Learning/Big Data Cluster Analysis
- Network Communications are using Spanning Tree Protocol (STP)
- Image registration and segmentation
- Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.
- Etc

Applications

Minimum spanning trees have direct applications in the design of networks

- Telecommunications networks
- Transportation networks
- Water supply networks
- Electrical grids

As a subroutine in

- Machine Learning/Big Data Cluster Analysis
- Network Communications are using Spanning Tree Protocol (STP)
- Image registration and segmentation
- Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.
- Etc

Applications

Minimum spanning trees have direct applications in the design of networks

- Telecommunications networks
- Transportation networks
- Water supply networks
- Electrical grids

As a subroutine in

- Machine Learning/Big Data Cluster Analysis
- Network Communications are using Spanning Tree Protocol (STP)
- Image registration and segmentation
- Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.
- Etc

Applications

Minimum spanning trees have direct applications in the design of networks

- Telecommunications networks
- Transportation networks
- Water supply networks
- Electrical grids

As a subroutine in

- Machine Learning/Big Data Cluster Analysis
- Network Communications are using Spanning Tree Protocol (STP)
- Image registration and segmentation
- Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.
- Etc

Applications

Minimum spanning trees have direct applications in the design of networks

- Telecommunications networks
- Transportation networks
- Water supply networks
- Electrical grids

As a subroutine in

- Machine Learning/Big Data Cluster Analysis
- Network Communications are using Spanning Tree Protocol (STP)
- Image registration and segmentation
- Circuit design: implementing efficient multiple constant multiplications, as used in finite impulse response filters.
- Etc

Outline

Exercises

From Cormen's book solve

- 23.1-3
- 23.1-5
- 23.1-7
- 23.1-9
- 23.2-2
- 23.2-3
- 23.2-5
- 23.2-7