Analysis and Design of Algorithms

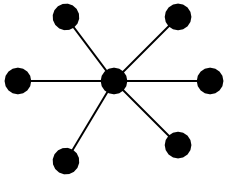
Graphs

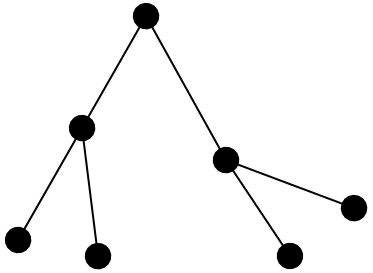
Part III: Minimum Spanning Trees (MST)

Instructor: **Morteza Zakeri**

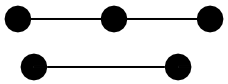
Tree

- We call an undirected graph a **tree** if the graph is *connected* and contains *no cycles*.
- Trees:

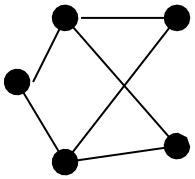




- Not Trees:



Not connected



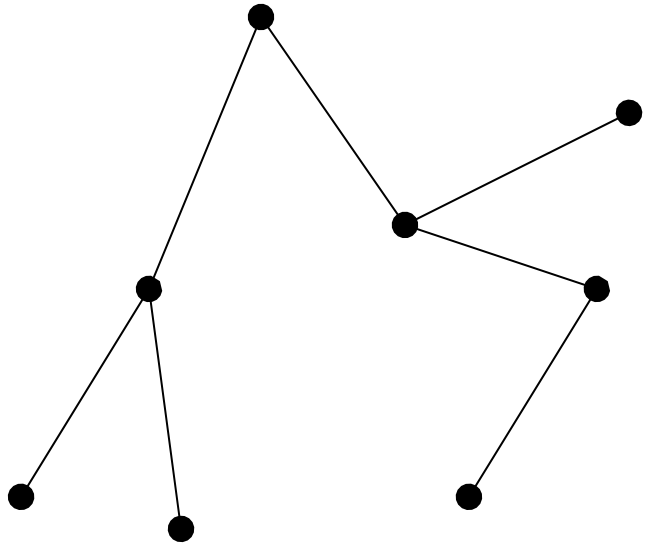
Has a cycle

Number of Vertices

- If a graph is a tree, then the number of edges in the graph is one less than the number of vertices.
- **A tree with *n* vertices has *n* – 1 edges.**
  - Each node has one parent except for the root.
    - Note: Any node can be the root here, as we are not dealing with rooted trees.

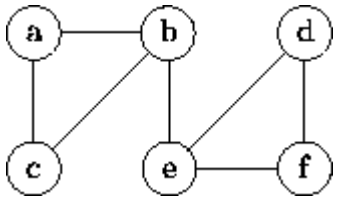
Connected Graph

- A **connected graph** is one in which there is *at least one path* between each pair of vertices.

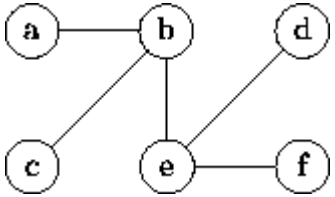


Spanning Tree

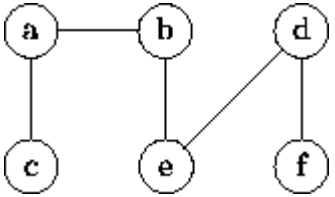
- In a tree there is always **exactly one path** from each vertex in the graph to any other vertex in the graph.
- A **spanning tree** for a graph is a subgraph that includes every vertex of the original, and is a tree.



(a) Graph G



(b) Breadth-first spanning tree of G rooted at b



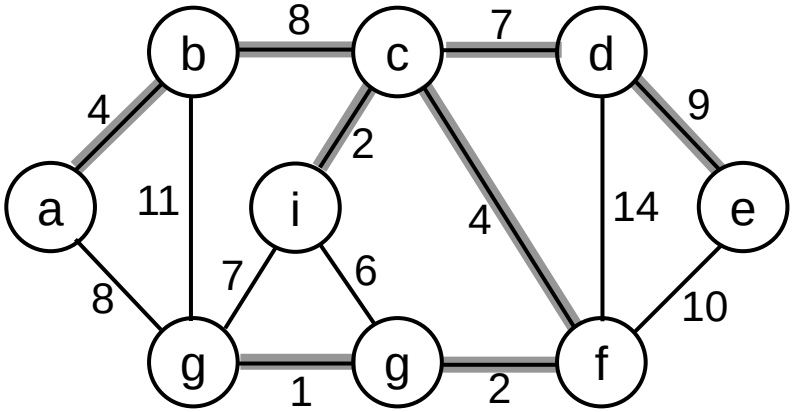
(c) Depth-first spanning tree of G rooted at c

Non-Connected Graphs

- If the graph is not connected, we get a spanning tree for each **connected component** of the graph.
  - That is we get a forest.

Minimum Spanning Trees

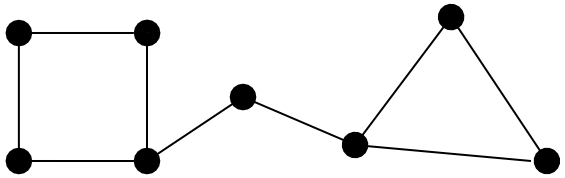
- **Spanning Tree**
  - A tree (*i.e.*, connected, acyclic graph) which contains all the vertices of the graph
- **Minimum Spanning Tree**
  - Spanning tree with the **minimum sum of weights**



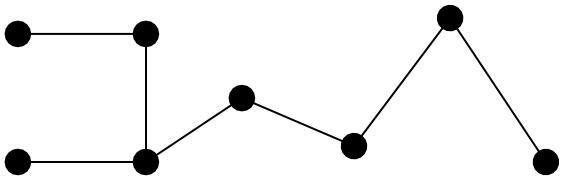
- **Spanning forest**
  - If a graph is not connected, then there is a spanning tree for each connected component of the graph

Finding a Spanning Tree

Find a spanning tree for the graph below.



We could break the two cycles by removing a single edge from each. One of several possible ways to do this is shown below.

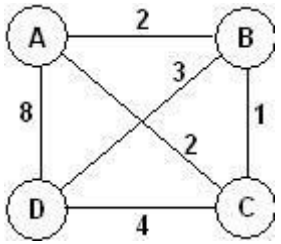


Minimum Spanning Tree

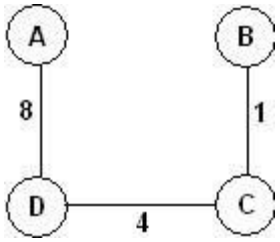
- A spanning tree that has minimum total weight is called a **minimum spanning tree** for the graph.
  - Technically it is a minimum-weight spanning tree.
- If all edges have the same weight, breadth-first search or depth-first search will yield minimum spanning trees.
  - For the rest of this discussion, we assume the edges have weights associated with them.

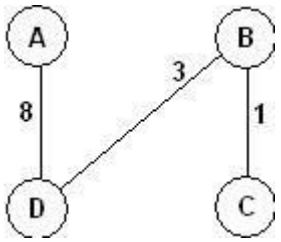
Minimum Spanning Tree

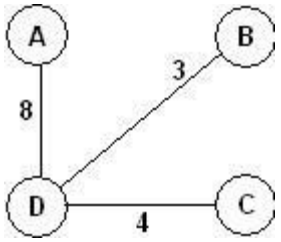
- Consider this graph.

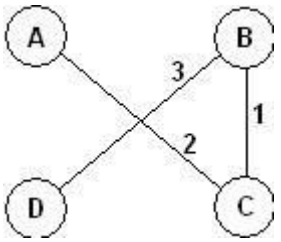


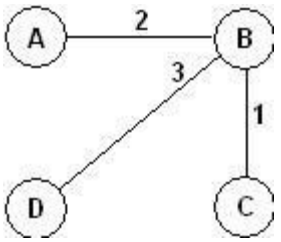
- It has 20 spanning trees. Some are:



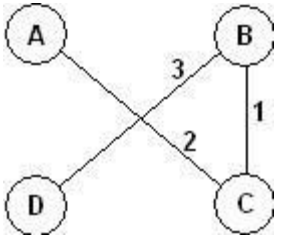


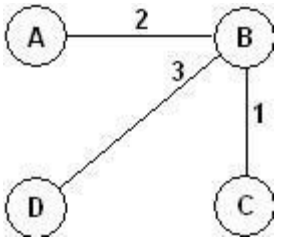






- There are two minimum-cost spanning trees, each with a cost of 6:

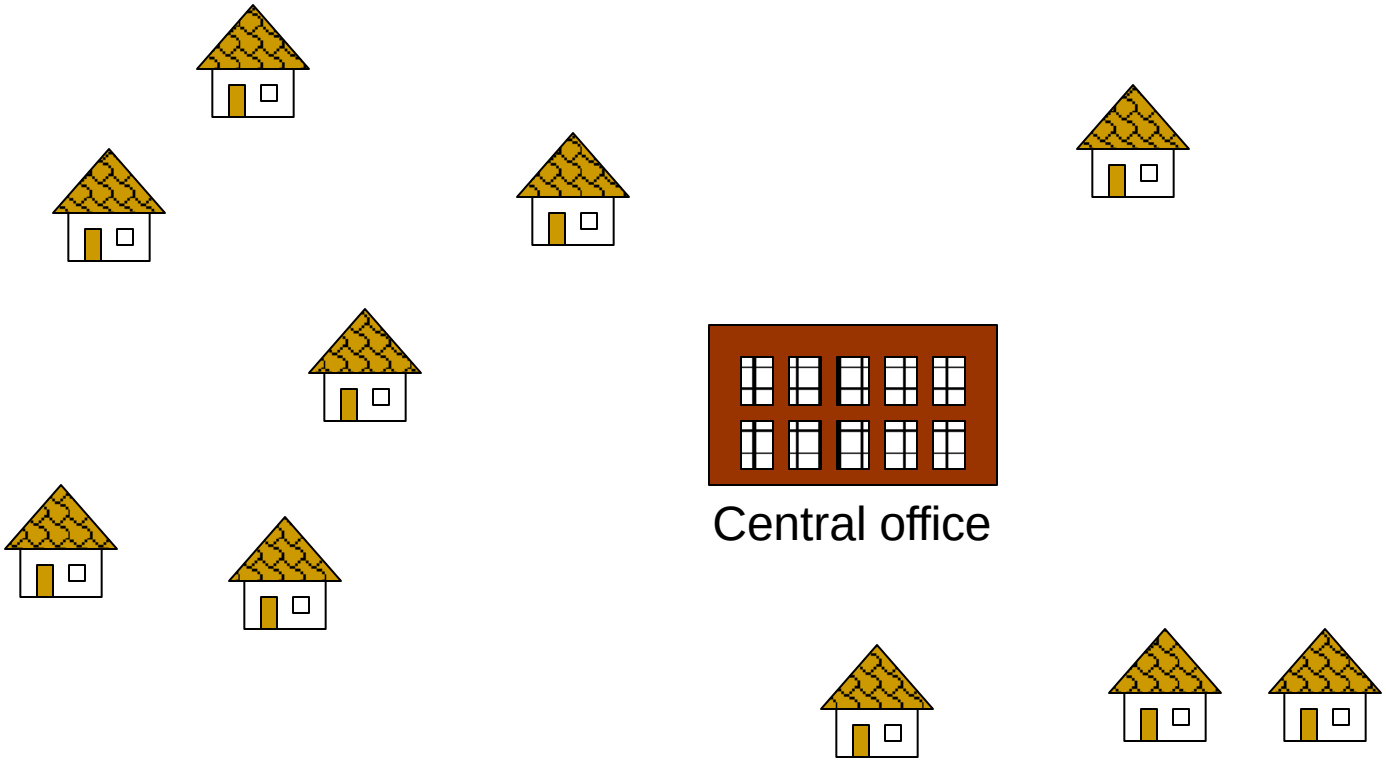




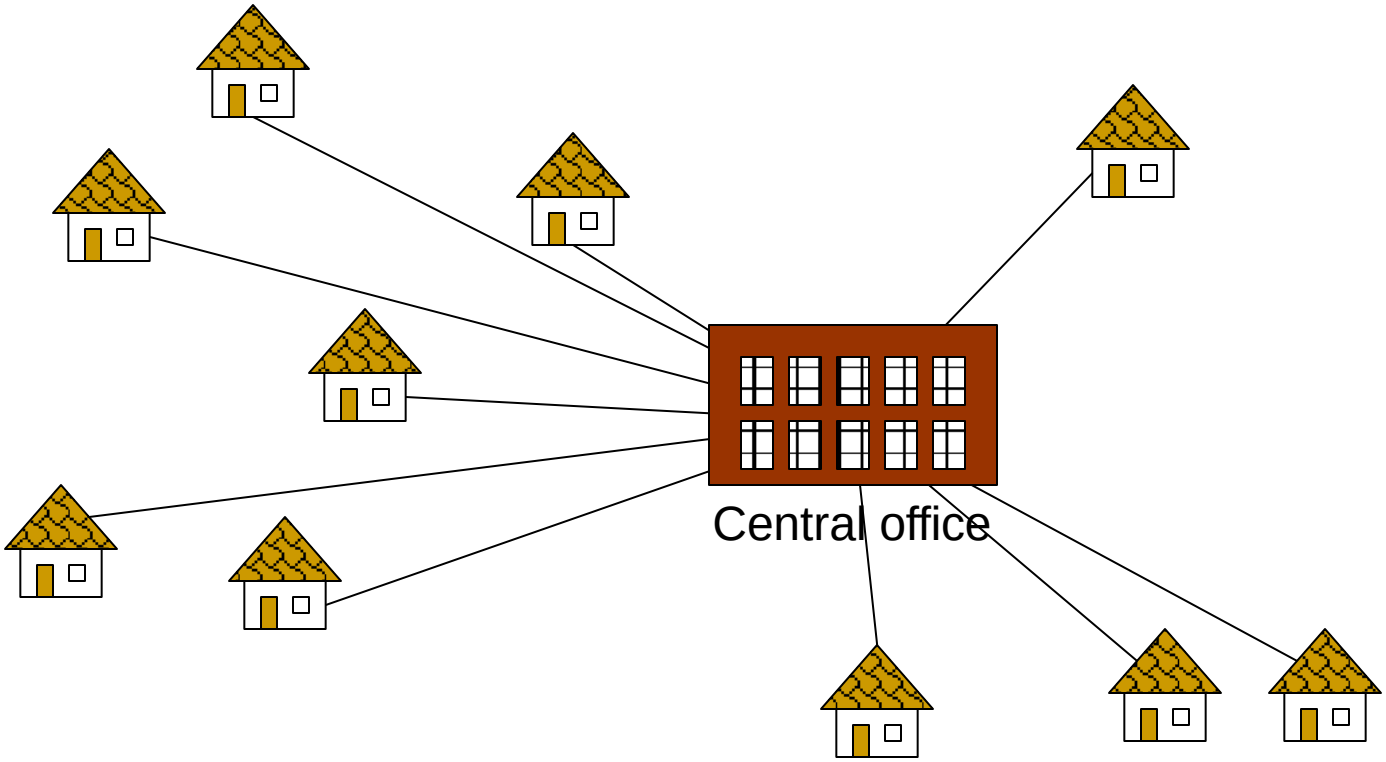
Applications of MST

- Minimum-cost spanning trees have many applications.
  - Building cable networks that join *n* locations with minimum cost.
  - Building a road network that joins *n* cities with minimum cost.

Problem: Laying Telephone Wire

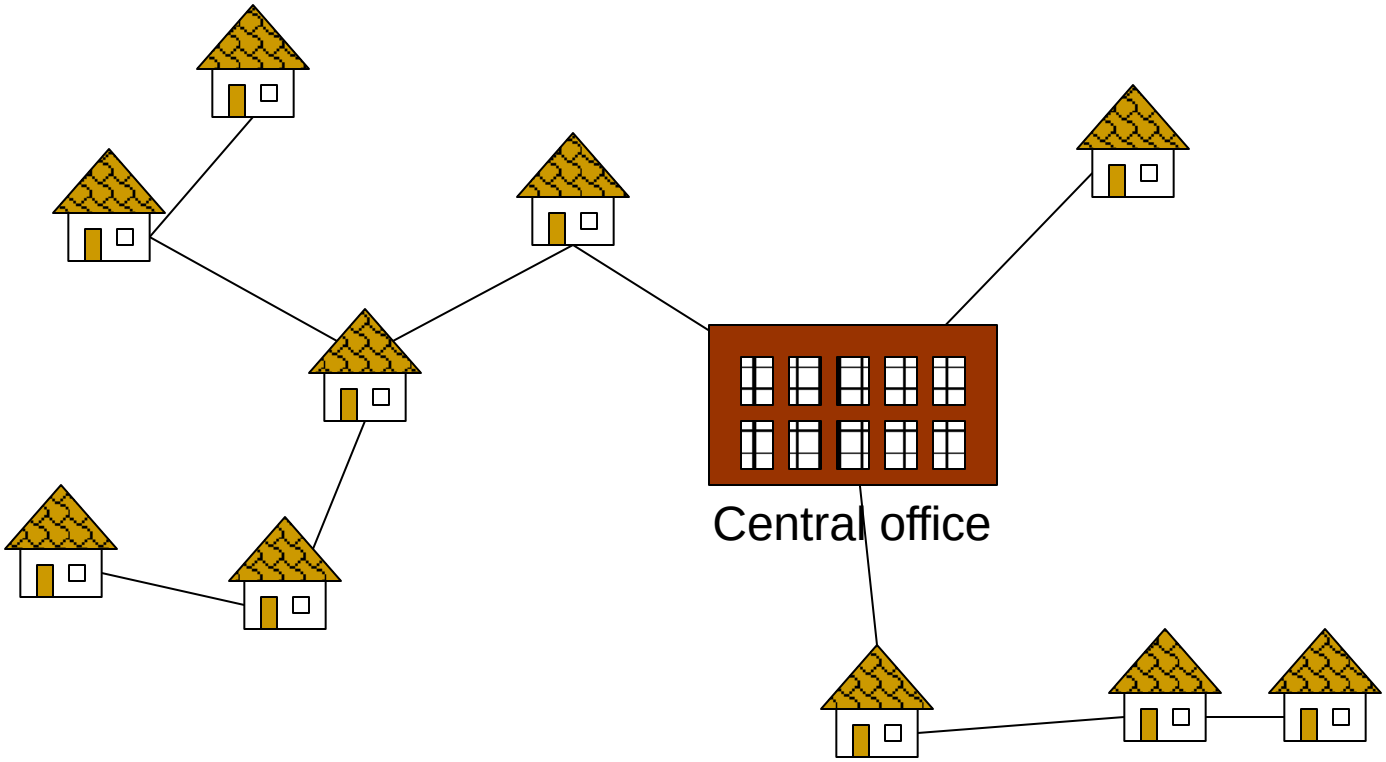


Wiring: Naïve Approach



Expensive!

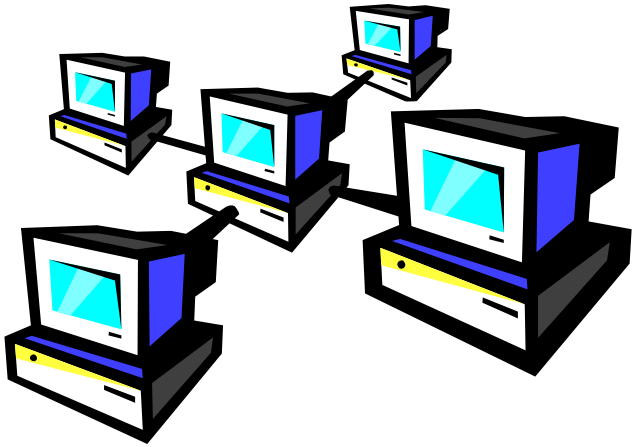
Wiring: Better Approach

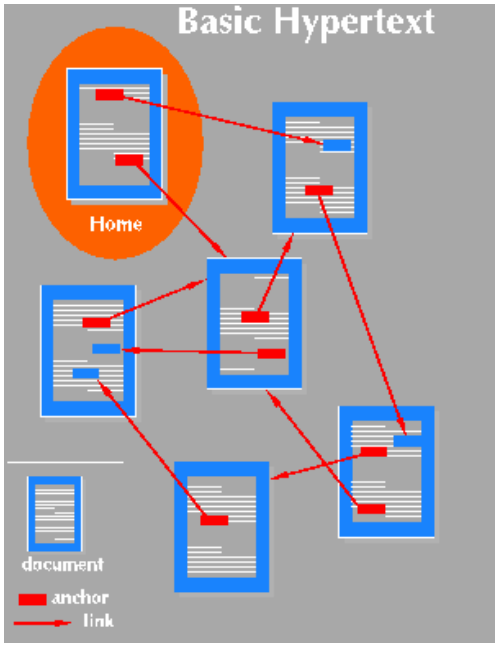


Minimize the total length of wire connecting the customers

Applications of MST

– Find the least expensive way to connect a set of cities, terminals, computers, etc.

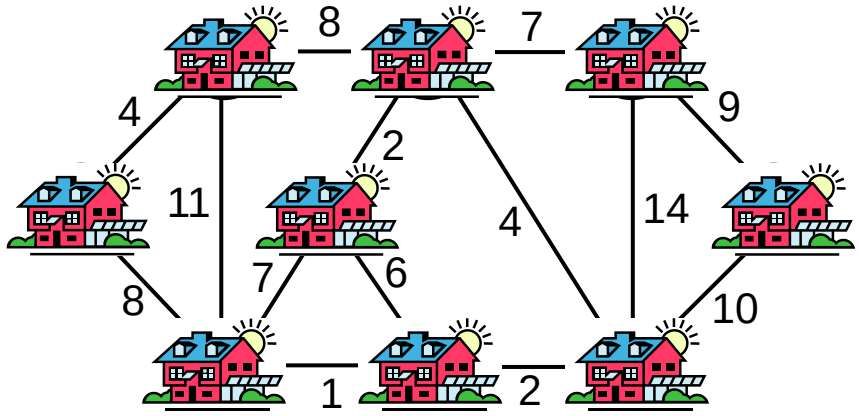




Example

Problem

- A town has a set of houses and a set of roads
- A road connects 2 and only 2 houses
- A road connecting houses u and v has a repair cost w(u, v)



Goal: Repair enough (and no more) roads such that:

1. Everyone stays connected

   i.e., can reach every house from all other houses

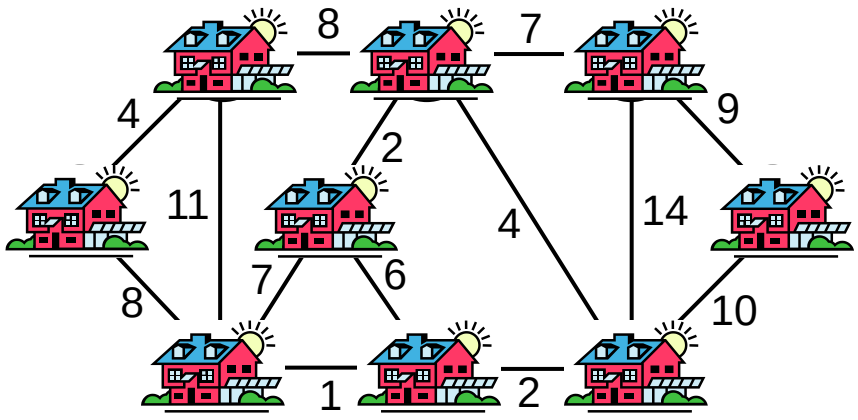
2. Total repair cost is minimum

Minimum Spanning Trees

- A connected, undirected graph:
  - Vertices = houses, Edges = roads
- A **weight** $w(u, v)$ on each edge $(u, v) \in E$

Find $T \subseteq E$ such that:

1. T connects all vertices
2. $w(T) = \Sigma_{(u,v)\in T} w(u, v)$ is minimized



Properties of Minimum Spanning Trees

- Minimum spanning tree is **not** unique

- MST has no cycles – see why:
  - We can take out an edge of a cycle, and still have the vertices connected while reducing the cost
- # of edges in a MST:
  - |V| - 1

Brute Force MST

- Brute Force option:
  1. For all possible spanning trees
     i. Calculate the sum of the edge weights
     ii. Keep track of the tree with the minimum weight.
- Step i) requires N-1 time, since each tree will have exactly N-1 edges.
- If there are M spanning trees, then the total cost will *O(MN).*
- Consider a complete graph, with *N(N-1)* edges. How big can M be?

Brute Force MST

- For a **complete graph**, it has been shown that there are N^{N-2} possible spanning trees!
- Alternatively, given N items, you can build N^{N-2}
- distinct trees to connect these items.

Greedy MST

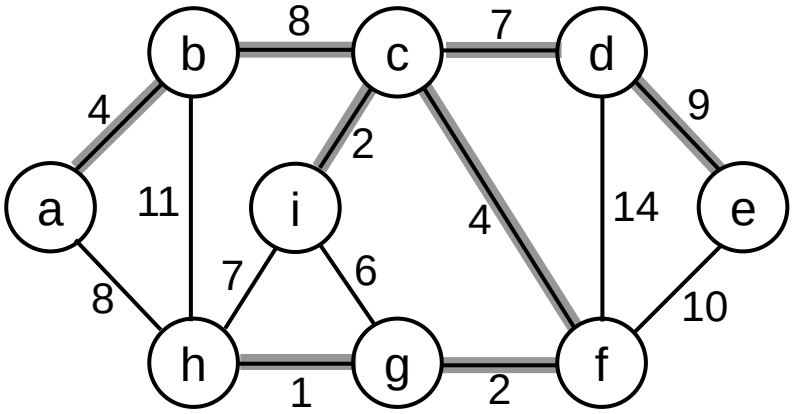
- There are many approaches to computing a minimum spanning tree. We could try to **detect cycles** and **remove edges**, but the two algorithms we will study build them from the bottom-up in a *greedy* fashion.

- **Kruskal's Algorithm – *starts with a forest of single node trees*** and then adds the edge with the minimum weight to connect two components.

- **Prim's Algorithm – *starts with a single vertex*** and then adds the minimum edge to extend the spanning tree.

Growing a MST – Generic Approach

- Grow a set A of edges (initially empty)
- Incrementally add edges to A such that they would **belong to** a MST

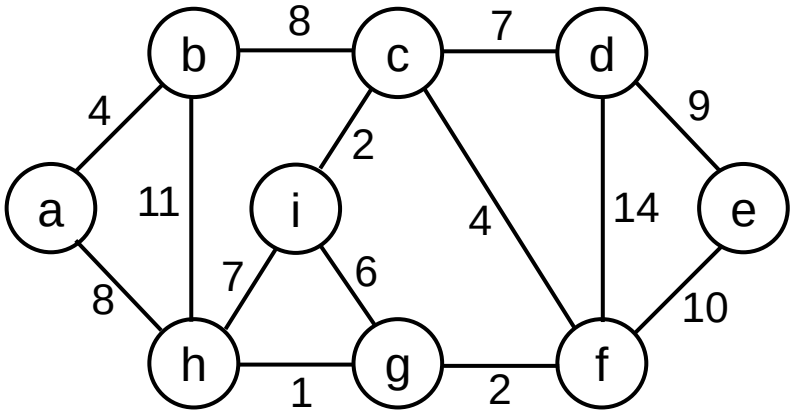


Idea: add only "safe" edges

– An edge (u, v) is **safe** for A if and only if $A \cup \{(u, v)\}$ is also a subset of **some** MST

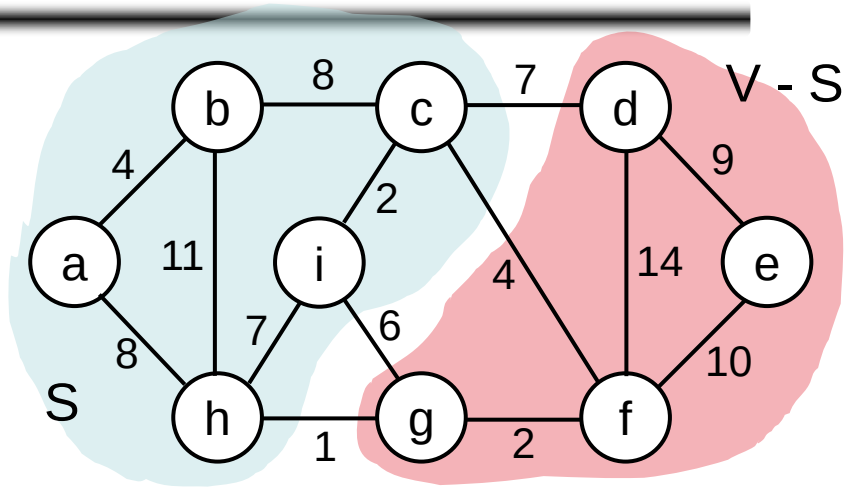
Generic MST algorithm

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



- How do we find safe edges?

Finding Safe Edges



- Let's look at edge (h, g)
  - Is it safe for A initially?
- Later on:
  - Let $S \subset V$ be any set of vertices that includes h but not g (so that g is in V - S)
  - In any MST, there has to be one edge (at least) that connects S with V - S
  - Why not choose the edge with **minimum weight** (h,g)?

Definitions

- A **cut** (S, V - S)

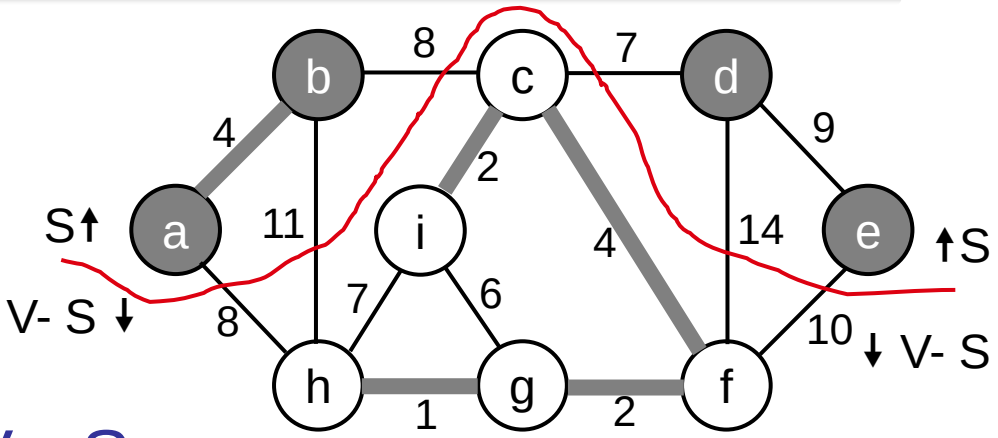
is a partition of vertices

into disjoint sets S and V - S

- An edge **crosses** the cut

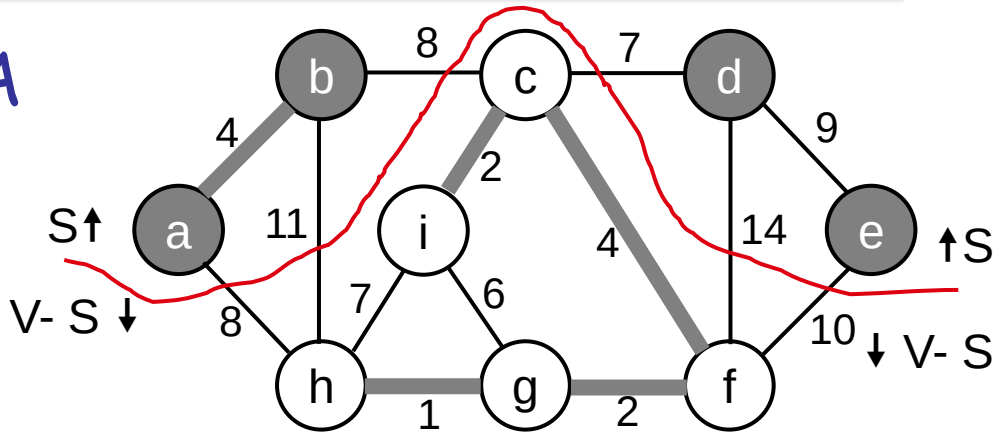
(S, V - S) if one endpoint is in S

and the other in V – S



Definitions (cont'd)

- A cut **respects** a set ***A*** of edges ⇔ no edge in ***A*** crosses the cut



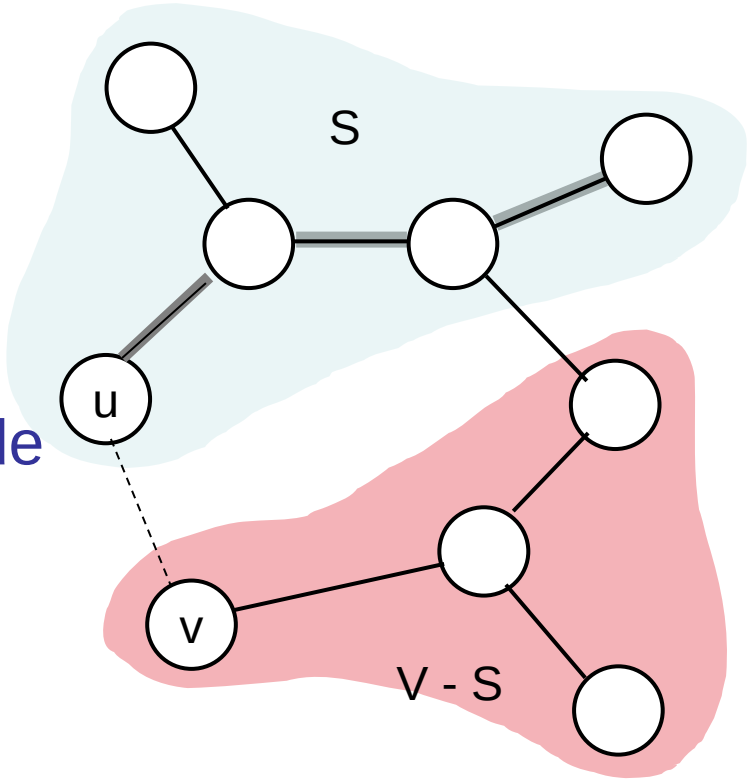
- An edge is a **light edge** crossing a cut ⇔ its weight is minimum over all edges crossing the cut
  - Note that for a given cut, there can be > 1 light edges crossing it

Theorem

- Let *A* be a subset of some MST (*i.e.*, T), (S, V - S) be a **cut** that respects A, and (u, v) be a **light edge** crossing (S, V-S). Then (u, v) is safe for *A*.

Proof:

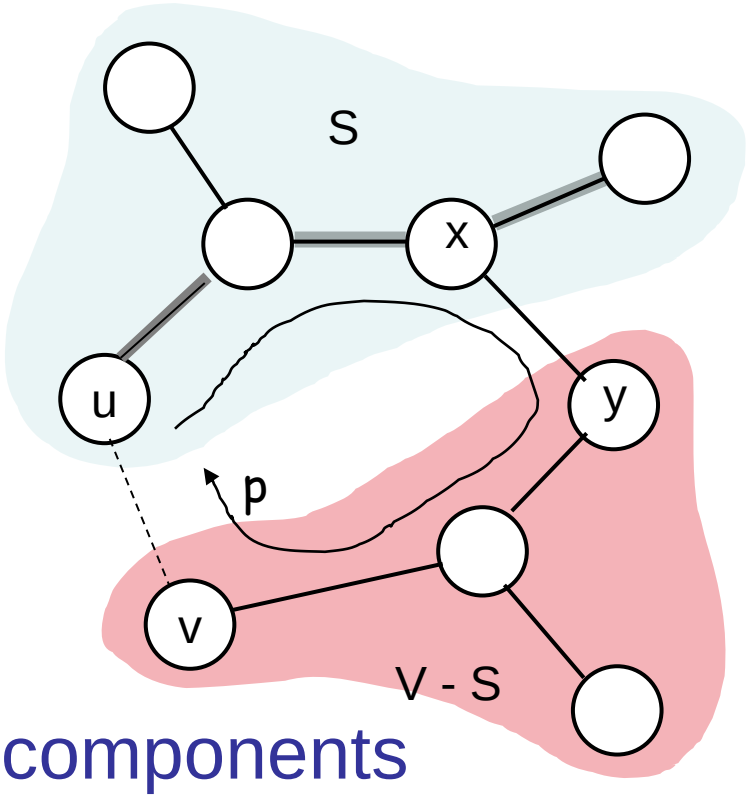
- Let T be an MST that includes A
  - edges in A are shaded
- Case1: If T includes (u, v), then it would be safe for A
- Case2: Suppose T does not include the edge (u, v)
- **Idea**: construct another MST T' that includes A ∪ {(u, v)}



Theorem - Proof

- T contains a unique path p between u and v
- Path p must cross the cut (S, V - S) at least once: let (x, y) be that edge
- Let's remove (x,y) $\Rightarrow$ breaks T into two components.
- Adding (u, v) reconnects the components

$$T' = T - \{(x, y)\} \cup \{(u, v)\}$$



Theorem – Proof (cont.)

$T' = T - \{(x, y)\} \cup \{(u, v)\}$

Have to show that T' is an MST:

- (u, v) is a light edge

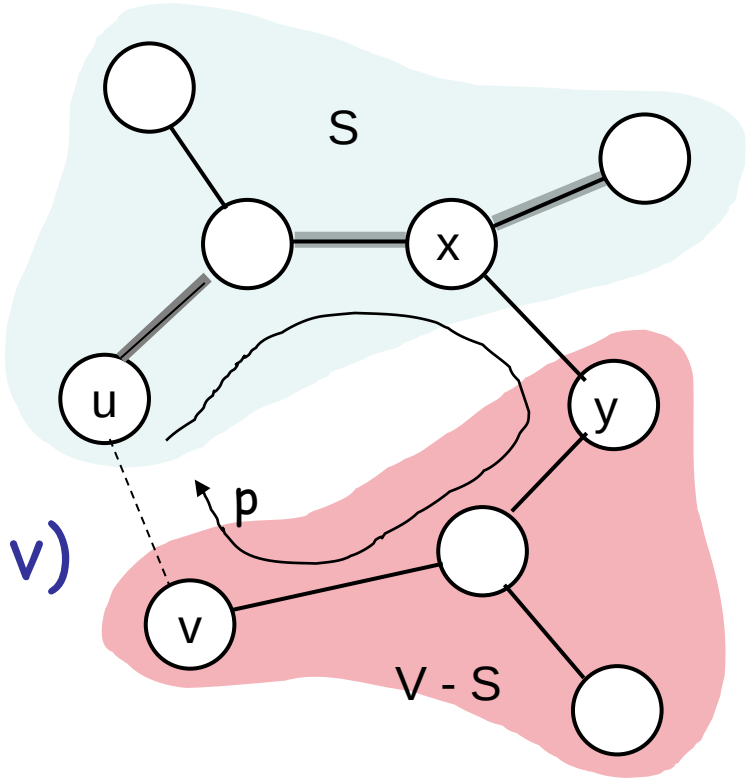
  $\Rightarrow w(u, v) \leq w(x, y)$

- $w(T') = w(T) - w(x, y) + w(u, v)$

  $\leq w(T)$

- Since T is a spanning tree

  $w(T) \leq w(T') \Rightarrow$ T' must be an MST as well



Theorem – Proof (cont.)

Need to show that (u, v) is safe for A:

i.e., (u, v) can be a part of an MST

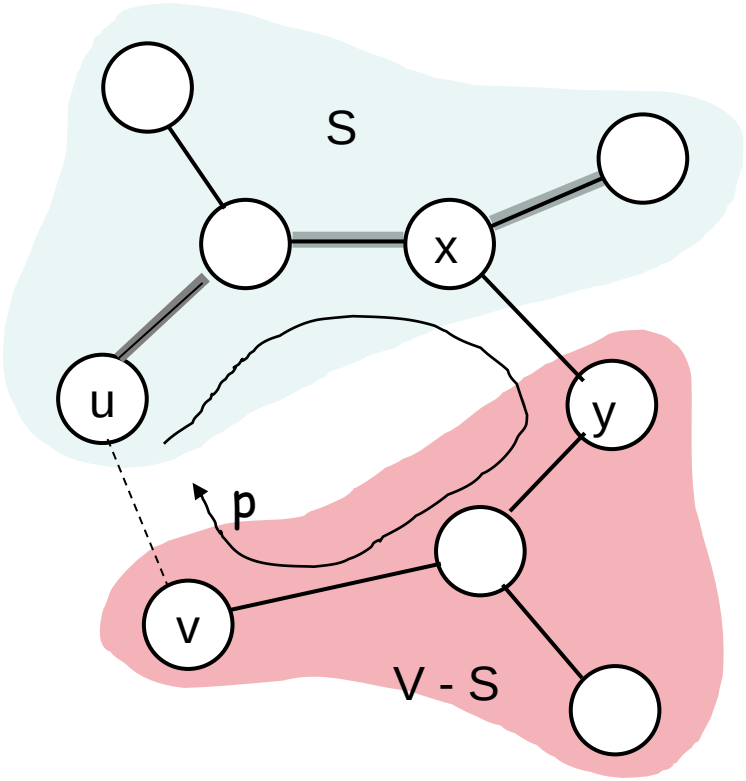
- $A \subseteq T$ and $(x, y) \notin T \Rightarrow$

  $(x, y) \notin A \Rightarrow A \subseteq T'$

- $A \cup \{(u, v)\} \subseteq T'$

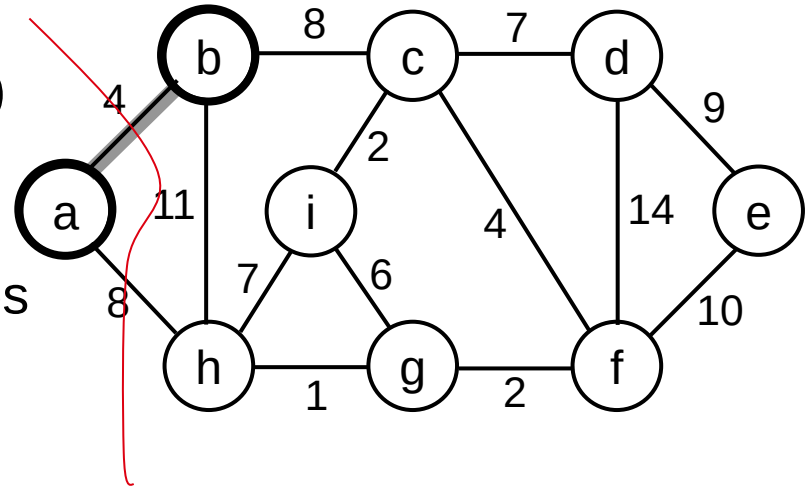
- Since T' is an MST

$\Rightarrow$ (u, v) is safe for A



Prim's Algorithm

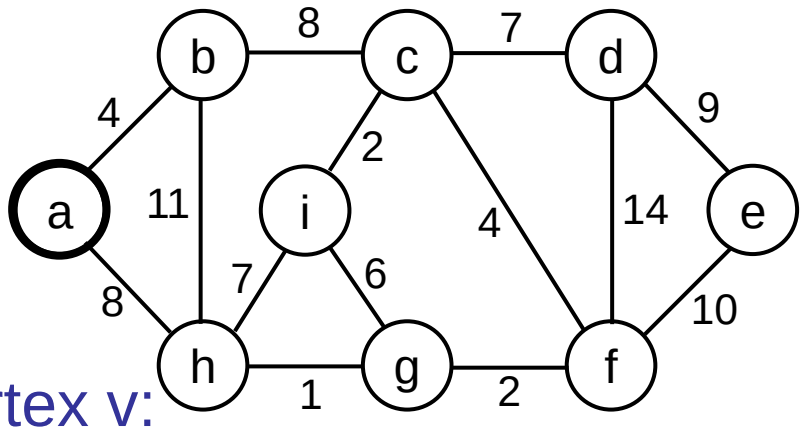
- The edges in set A always form a single tree
- Starts from an arbitrary "root": $V_A = \{a\}$
- At each step:
  - Find a light edge crossing $(V_A, V - V_A)$
  - Add this edge to A
  - Repeat until the tree spans all vertices



How to Find Light Edges Quickly?

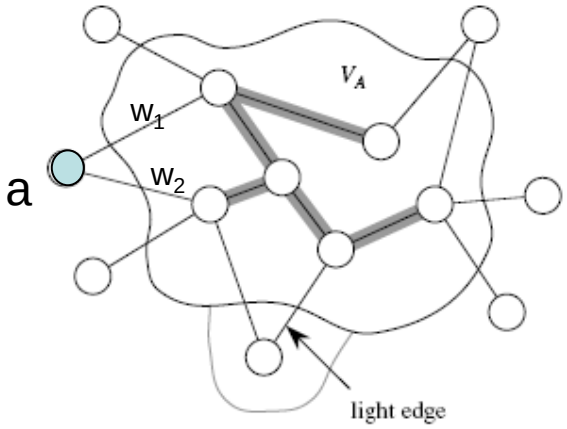
Use a priority queue Q:

- Contains vertices not yet included in the tree, i.e., $(V - V_A)$
  - $V_A = \{a\}$, $Q = \{b, c, d, e, f, g, h, i\}$
- We associate a key with each vertex v:



key[v] = minimum weight of any edge (u, v) connecting v to V_A

Key[a] = min(w_1,w_2)

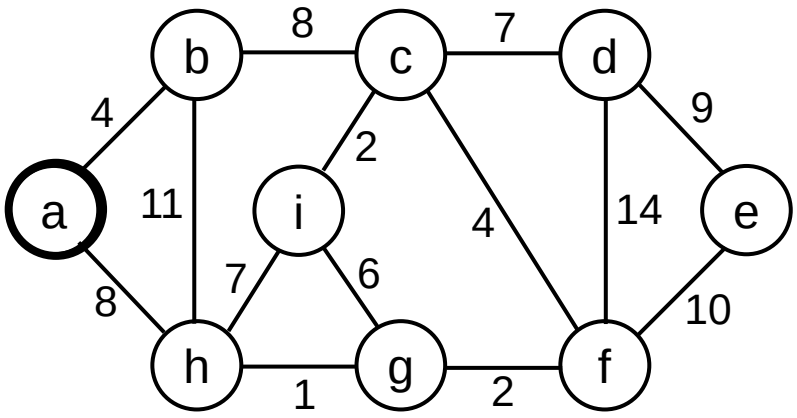


How to Find Light Edges Quickly? (cont.)

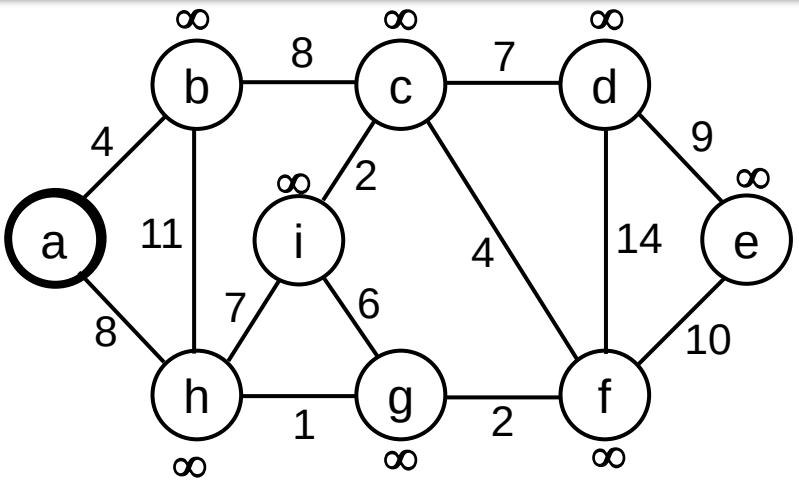
- After adding a new node to V_A we update the weights of all the nodes adjacent to it

  e.g., after adding a to the tree, k[b]=4 and k[h]=8

- Key of v is ∞ if v is not adjacent to any vertices in V_A



Example

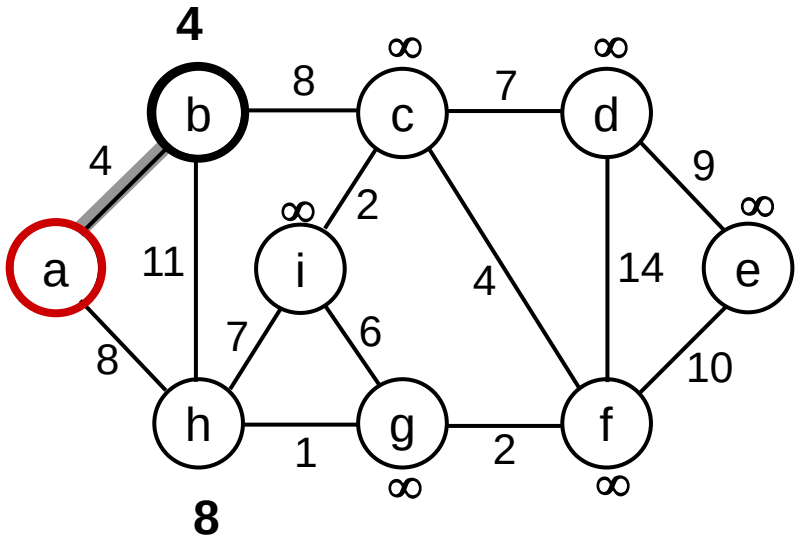


0 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞

Q = {a, b, c, d, e, f, g, h, i}

$V_A = \varnothing$

Extract-MIN(Q) ⇒ a



key [b] = 4 π [b] = a

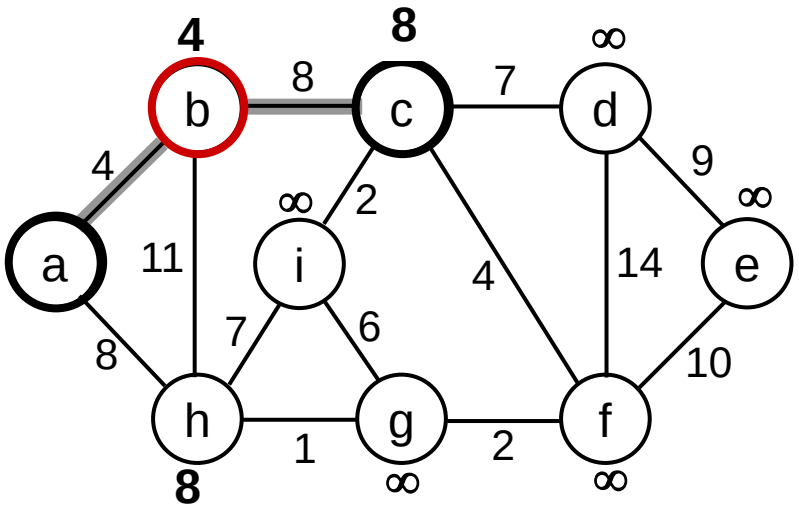
key [h] = 8 π [h] = a

4 ∞ ∞ ∞ ∞ ∞ **8** ∞

Q = {b, c, d, e, f, g, h, i} V_A = {a}

Extract-MIN(Q) ⇒ b

Example



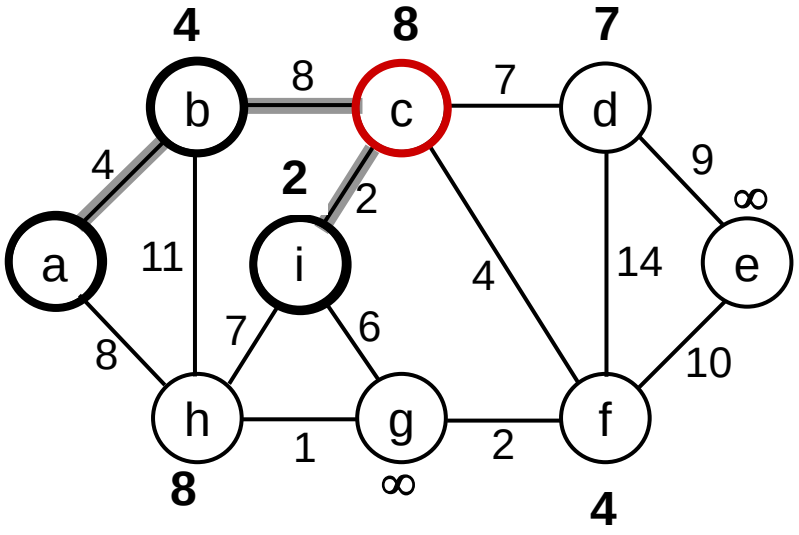
key [c] = 8 π [c] = b

key [h] = 8 π [h] = a - unchanged

8 ∞ ∞ ∞ ∞ **8** ∞

Q = {c, d, e, f, g, h, i} V_A = {a, b}

Extract-MIN(Q) ⇒ c



key [d] = 7 π [d] = c

key [f] = 4 π [f] = c

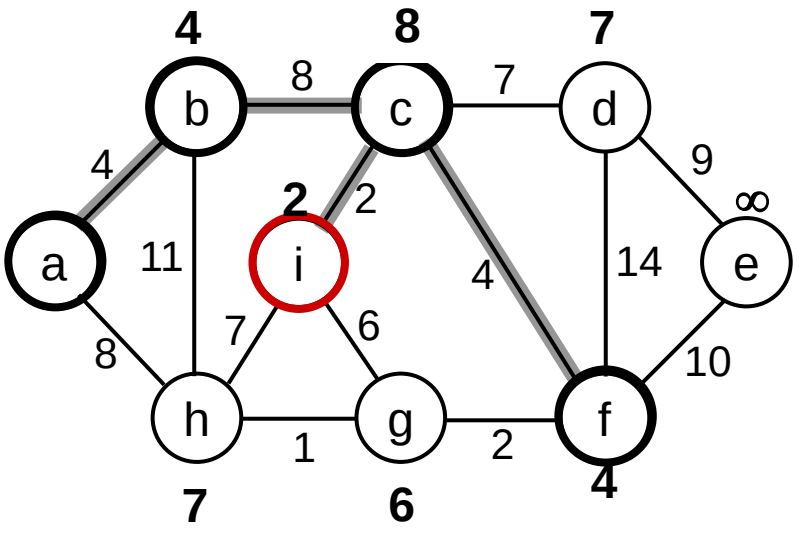
key [i] = 2 π [i] = c

7 ∞ **4** ∞ **8** **2**

Q = {d, e, f, g, h, i} V_A = {a, b, c}

Extract-MIN(Q) ⇒ i

Example



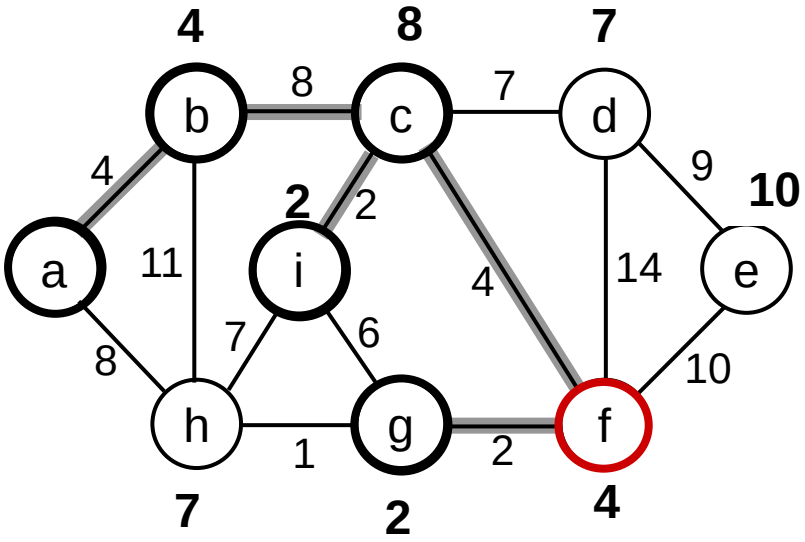
key [h] = 7 π [h] = i

key [g] = 6 π [g] = i

7 ∞ 4 6 8

Q = {d, e, f, g, h} V_A = {a, b, c, i}

Extract-MIN(Q) $\Rightarrow$ f



key [g] = 2 π [g] = f

key [d] = 7 π [d] = c unchanged

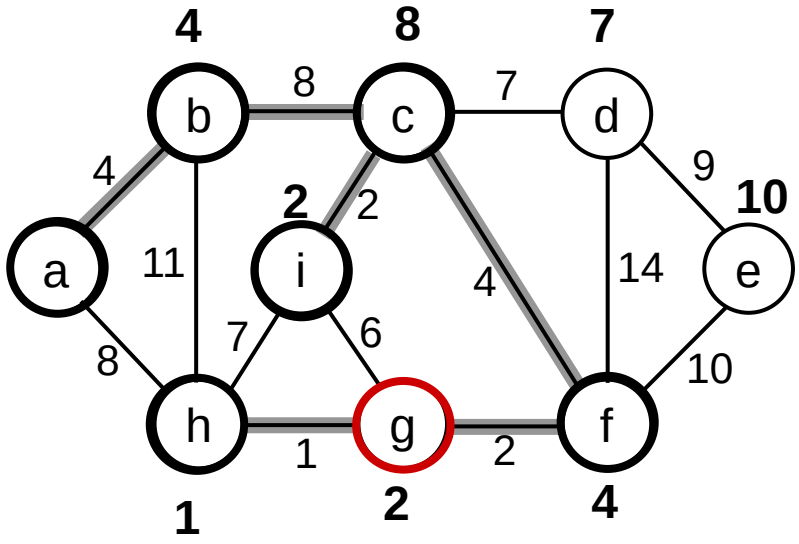
key [e] = 10 π [e] = f

7 10 2 8

Q = {d, e, g, h} V_A = {a, b, c, i, f}

Extract-MIN(Q) $\Rightarrow$ g

Example

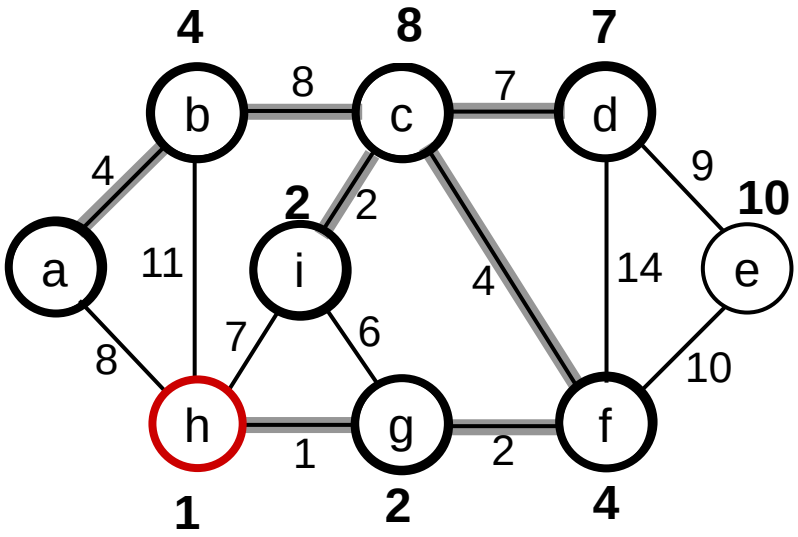


key [h] = 1 π [h] = g

7 10 1

Q = {d, e, h} V_A = {a, b, c, i, f, g}

Extract-MIN(Q) $\Rightarrow$ h

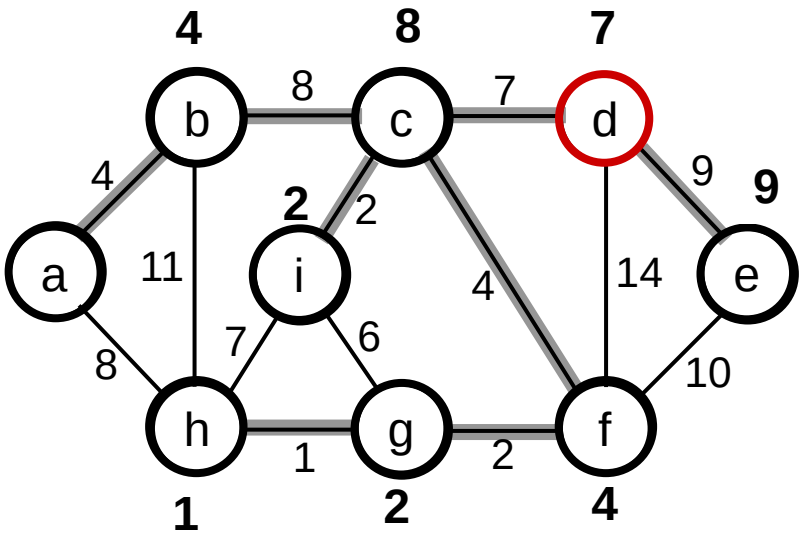


7 10

Q = {d, e} V_A = {a, b, c, i, f, g, h}

Extract-MIN(Q) $\Rightarrow$ d

Example



key [e] = 9 π [e] = f

9

Q = {e} V_A = {a, b, c, i, f, g, h, d}

Extract-MIN(Q) $\Rightarrow$ e

Q = $\varnothing$ V_A = {a, b, c, i, f, g, h, d, e}

PRIM(V, E, w, r)

```
1.   Q ← ∅
2.   for each u ∈ V
3.        do key[u] ← ∞
4.           π[u] ← NIL
5.           INSERT(Q, u)
6.   DECREASE-KEY(Q, r, 0)        ▶ key[r] ← 0
7.   while Q ≠ ∅
8.        do u ← EXTRACT-MIN(Q)
9.           for each v ∈ Adj[u]
10.               do if v ∈ Q and w(u, v) < key[v]
11.                  then π[v] ← u
12.                       DECREASE-KEY(Q, v, w(u, v))
```

O(V) if Q is implemented as a min-heap

Total time: $O(V \lg V + E \lg V) = O(E \lg V)$

O(lgV)

Executed |V| times

Takes *O*(lgV)

Min-heap operations: *O*(VlgV)

Executed O(E) times total

Constant

Takes *O*(lgV)

O(ElgV)

Using Fibonacci Heaps

- Depending on the heap implementation, running time could be improved!

	EXTRACT-MIN	DECREASE-KEY	Total
binary heap	O(lgV)	O(lgV)	O(ElgV)
Fibonacci heap	O(lgV)	O(1)	O(VlgV + E)

Prim's Algorithm

- Prim's algorithm finds a minimum cost spanning tree by selecting edges from the graph one-by-one as follows:
- It starts with a tree, T, consisting of a single starting vertex, x.
- Then, it finds the shortest edge emanating from x that connects T to the rest of the graph (i.e., a vertex not in the tree T).
- It adds this edge and the new vertex to the tree T.
- It then picks the shortest edge emanating from the revised tree T that also connects T to the rest of the graph and repeats the process.

Prim's Algorithm Abstract

Consider a graph G=(V, E);

Let T be a tree consisting of only the starting vertex **x;**

while (T has fewer than I V I vertices)

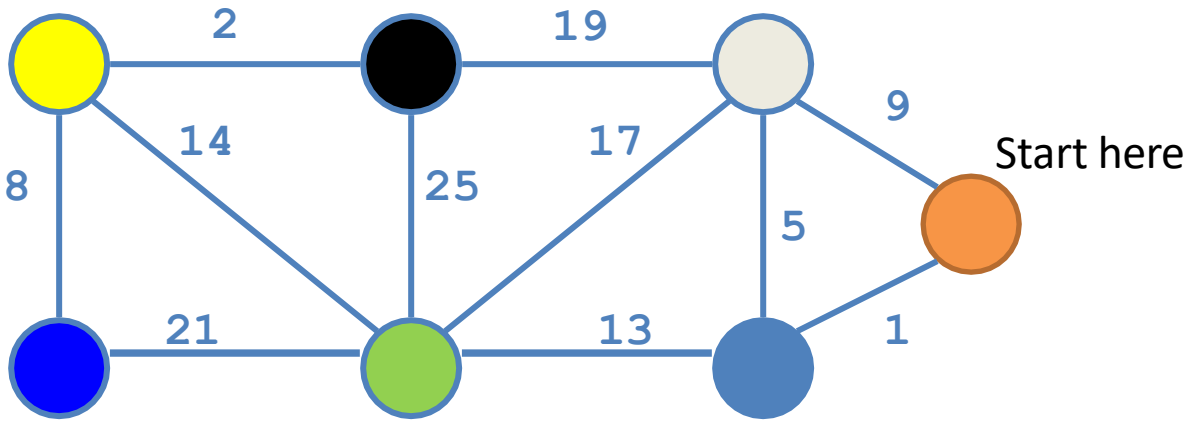
{

find a smallest edge connecting T to G-T;

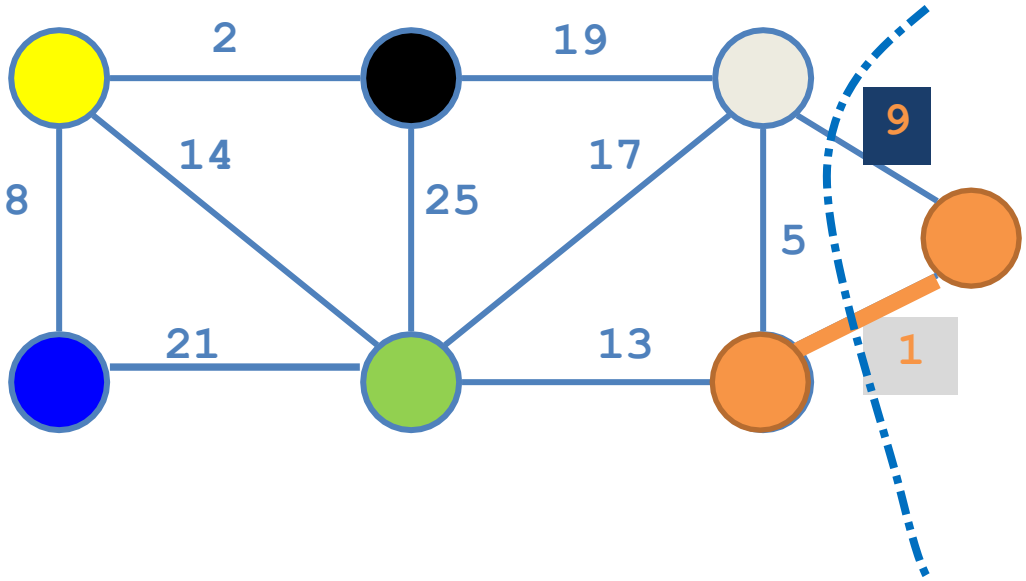
add it to T;

}

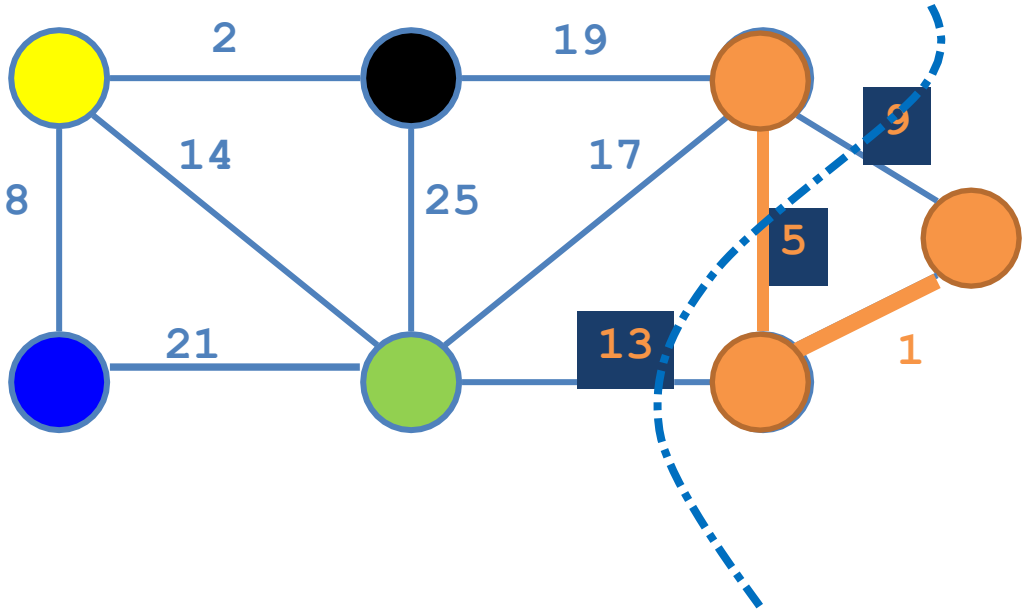
Prim's Algorithm



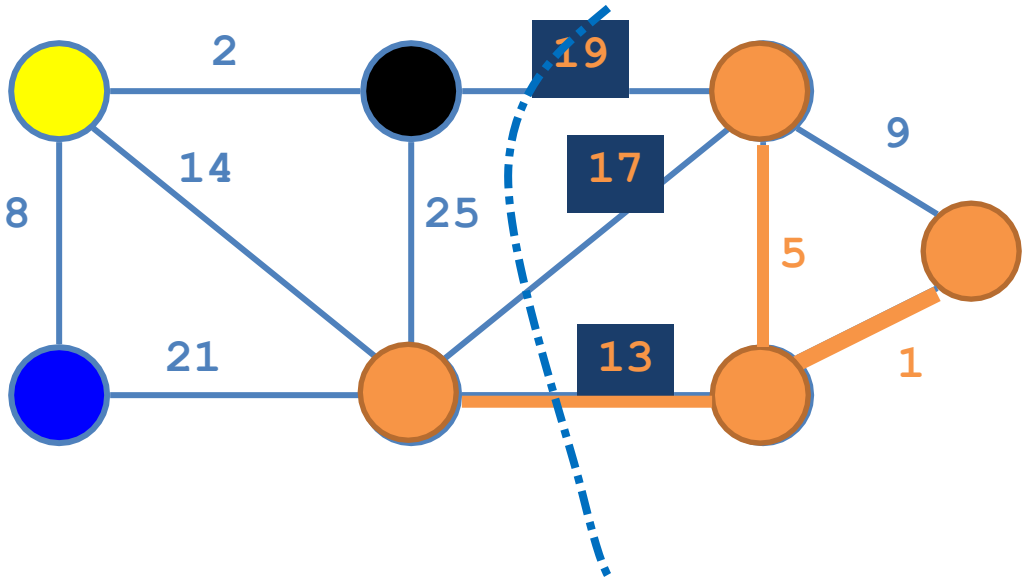
Prim's Algorithm



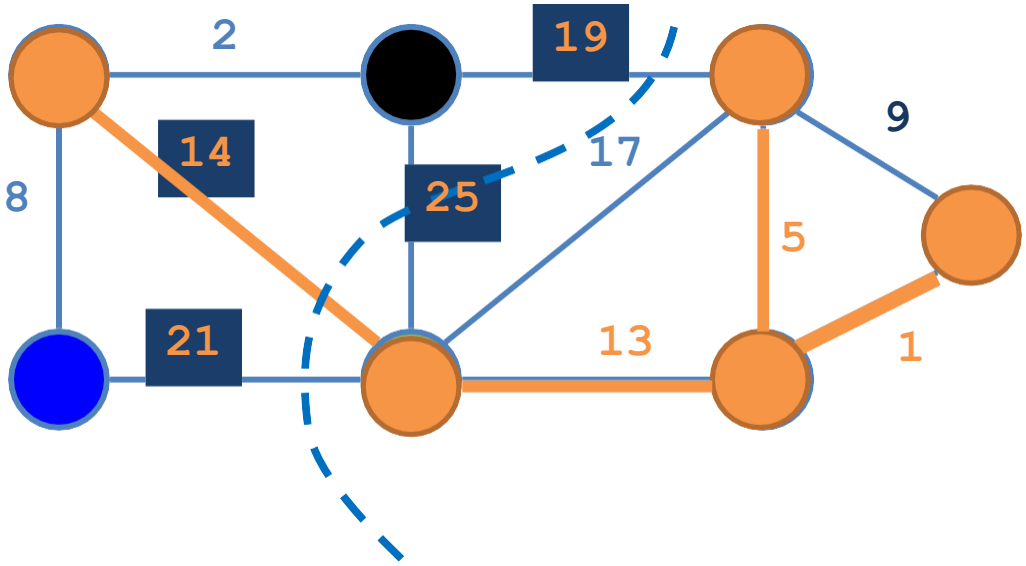
Prim's Algorithm



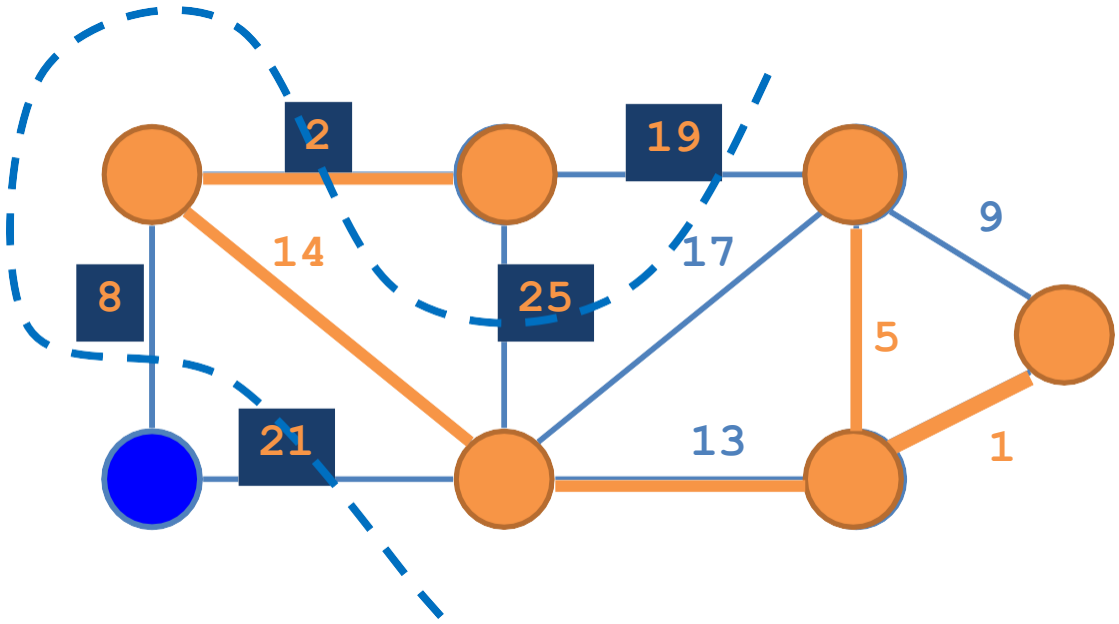
Prim's Algorithm



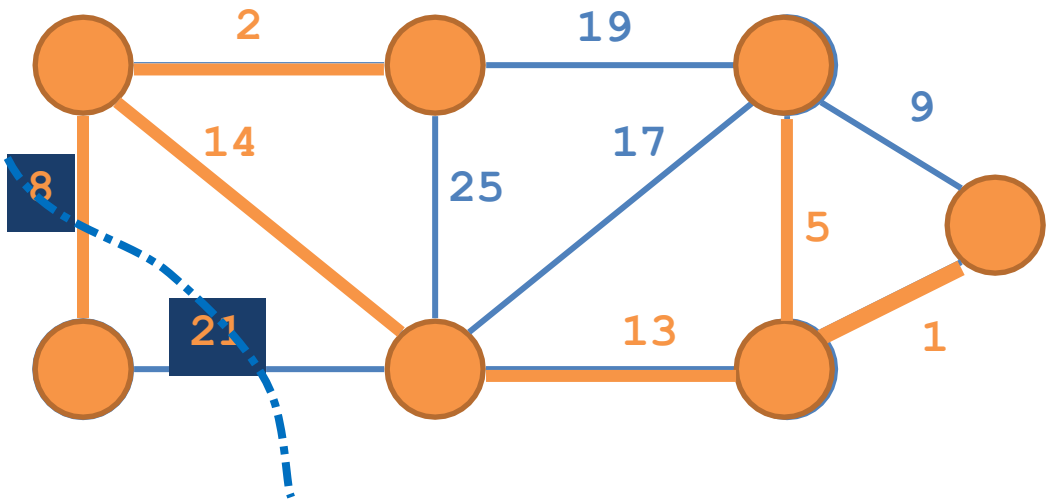
Prim's Algorithm



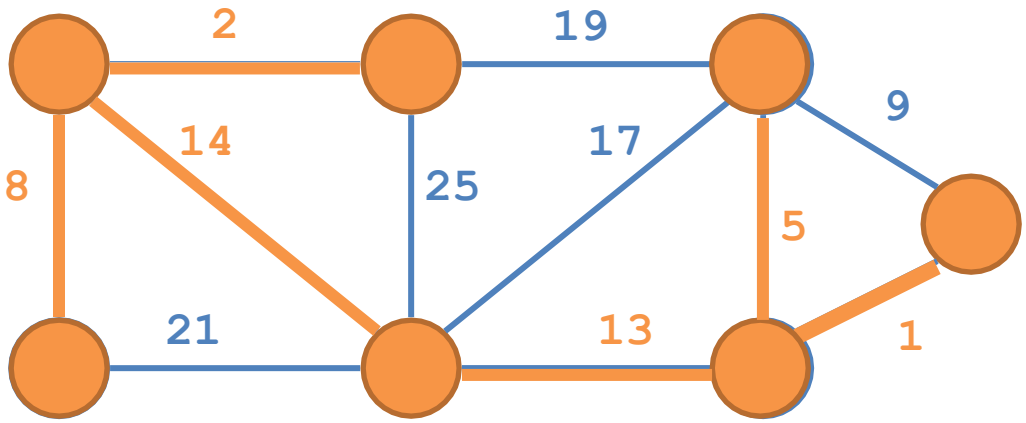
Prim's Algorithm



Prim's Algorithm



Prim's Algorithm

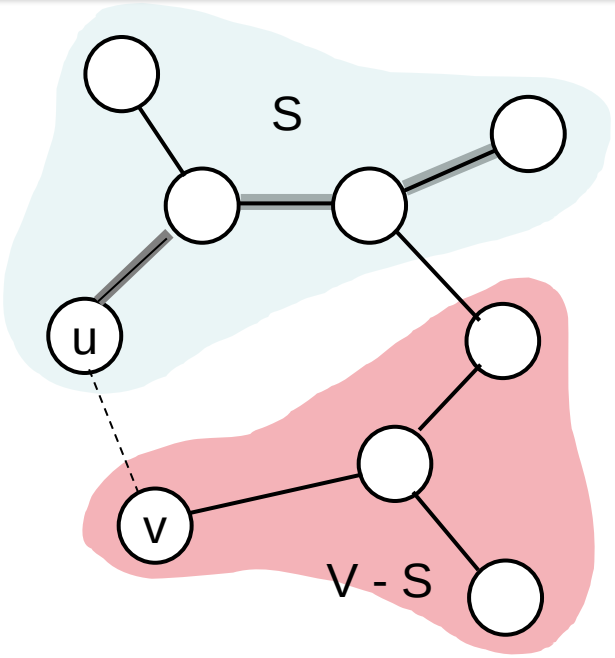


Prim's Algorithm

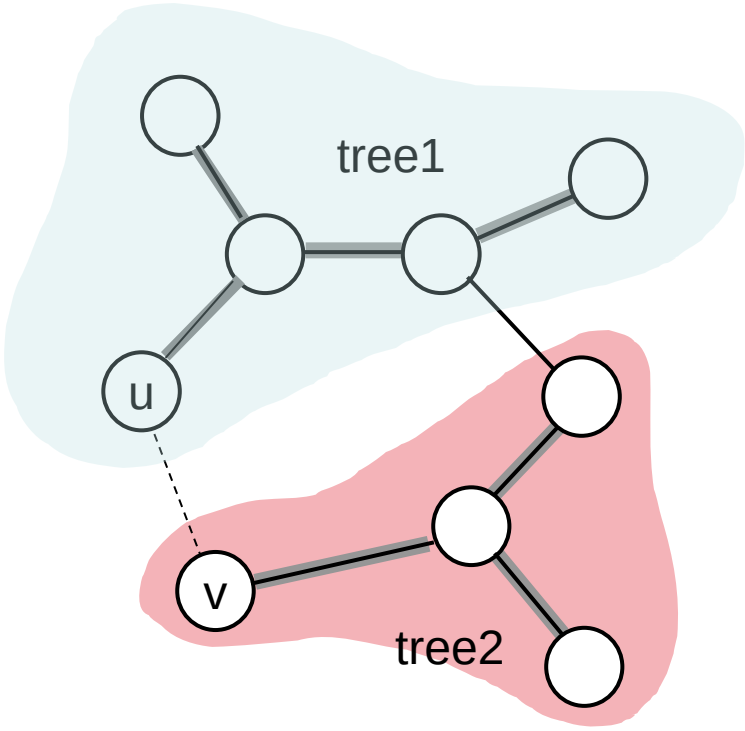
- Prim's algorithm is a **"greedy"** algorithm
  - Greedy algorithms find solutions based on a sequence of choices which are **"locally"** optimal at each step.
- Nevertheless, Prim's greedy strategy produces a **globally optimum solution**!
  - See proof for generic approach (i.e., slides 12-15)

A different instance of the generic approach

(instance 1)



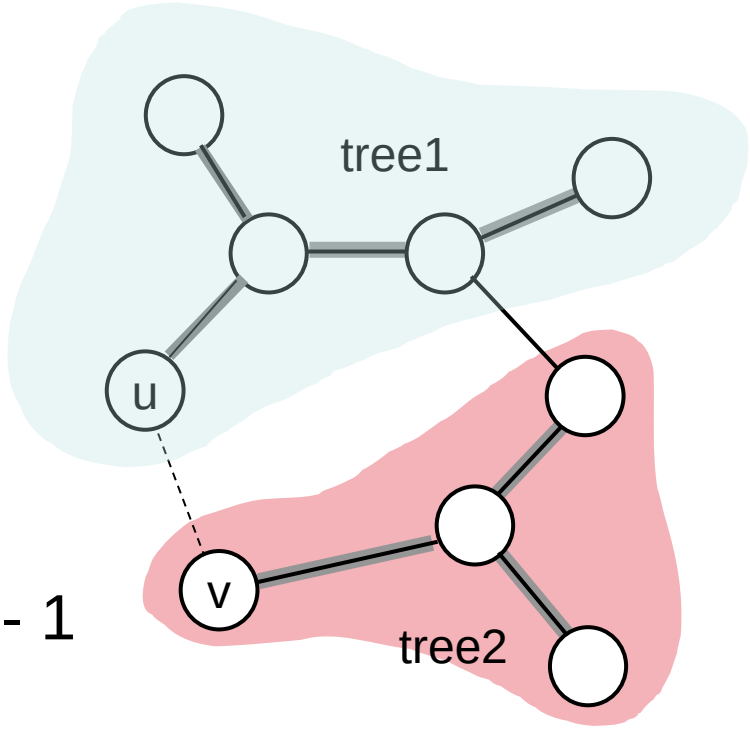
(instance 2)



- *A* is a forest containing connected components
  - Initially, each component is a single vertex
- Any safe edge merges two of these components into one
  - Each component is a tree

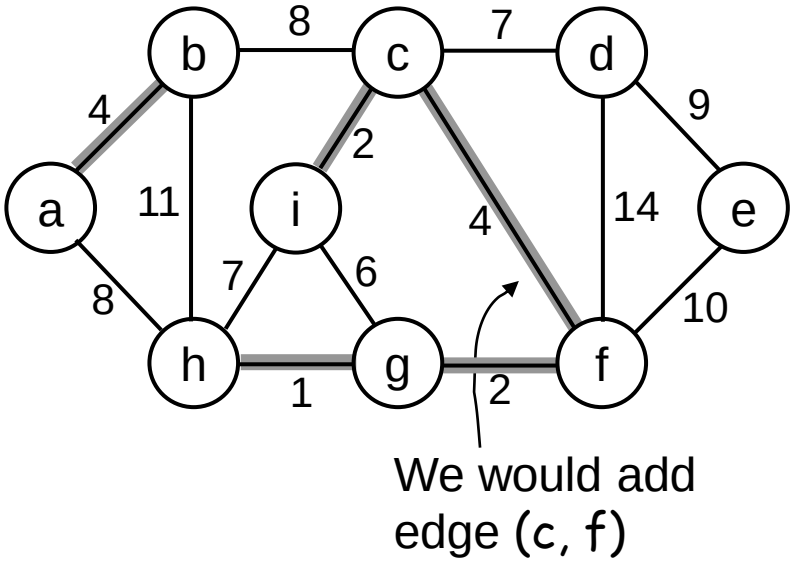
Kruskal's Algorithm

- How is it different from Prim's algorithm?
  - Prim's algorithm grows one tree all the time
  - Kruskal's algorithm grows multiple trees (i.e., a forest) at the same time.
  - Trees are merged together using **safe** edges
  - Since an MST has exactly |V| - 1 edges, after |V| - 1 merges, we would have only one component



Kruskal's Algorithm

- Start with each vertex being its own component
- Repeatedly merge two components into one by choosing the **light** edge that connects them
- Which components to consider at each iteration?
  - Scan the set of edges in monotonically increasing order by weight

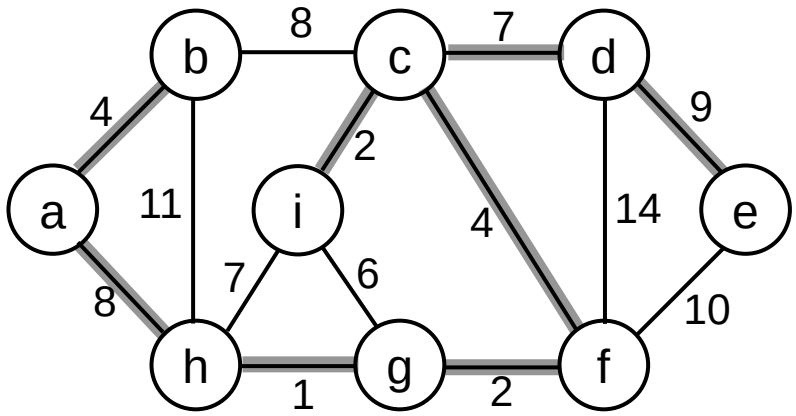


Kruskal's Algorithm

- Greedy algorithm to choose the edges as follows.

Step 1	**First edge: choose any edge with the minimum weight.**
Step 2	**Next edge: choose any edge with minimum weight from *those not yet selected*. (The subgraph can look disconnected at this stage.)**
Step 3	**Continue to choose edges of minimum weight from those not yet selected, except *do not select any edge that creates a cycle* in the subgraph.**
Step 4	**Repeat step 3 until the subgraph connects all vertices of the original graph.**

Example



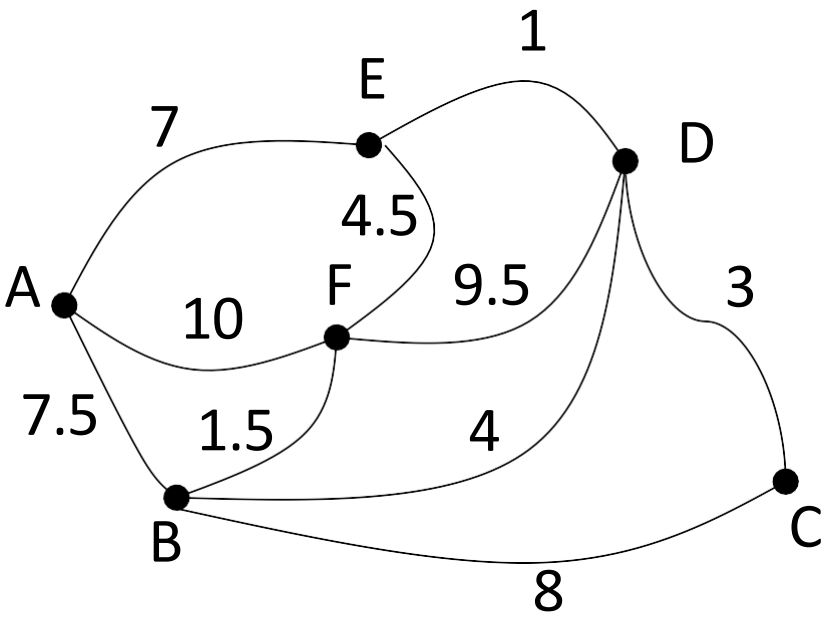
1: (h, g)
2: (c, i), (g, f)
4: (a, b), (c, f)
6: (i, g)
7: (c, d), (i, h)
8: (a, h), (b, c)
9: (d, e)
10: (e, f)
11: (b, h)
14: (d, f)

{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

1. Add (h, g) {g, h}, {a}, {b}, {c}, {d}, {e}, {f}, {i}
2. Add (c, i) {g, h}, {c, i}, {a}, {b}, {d}, {e}, {f}
3. Add (g, f) {g, h, f}, {c, i}, {a}, {b}, {d}, {e}
4. Add (a, b) {g, h, f}, {c, i}, {a, b}, {d}, {e}
5. Add (c, f) {g, h, f, c, i}, {a, b}, {d}, {e}
6. Ignore (i, g) {g, h, f, c, i}, {a, b}, {d}, {e}
7. Add (c, d) {g, h, f, c, i, d}, {a, b}, {e}
8. Ignore (i, h) {g, h, f, c, i, d}, {a, b}, {e}
9. Add (a, h) {g, h, f, c, i, d, a, b}, {e}
10. Ignore (b, c) {g, h, f, c, i, d, a, b}, {e}
11. Add (d, e) {g, h, f, c, i, d, a, b, e}
12. Ignore (e, f) {g, h, f, c, i, d, a, b, e}
13. Ignore (b, h) {g, h, f, c, i, d, a, b, e}
14. Ignore (d, f) {g, h, f, c, i, d, a, b, e}

Kruskal's Algorithm

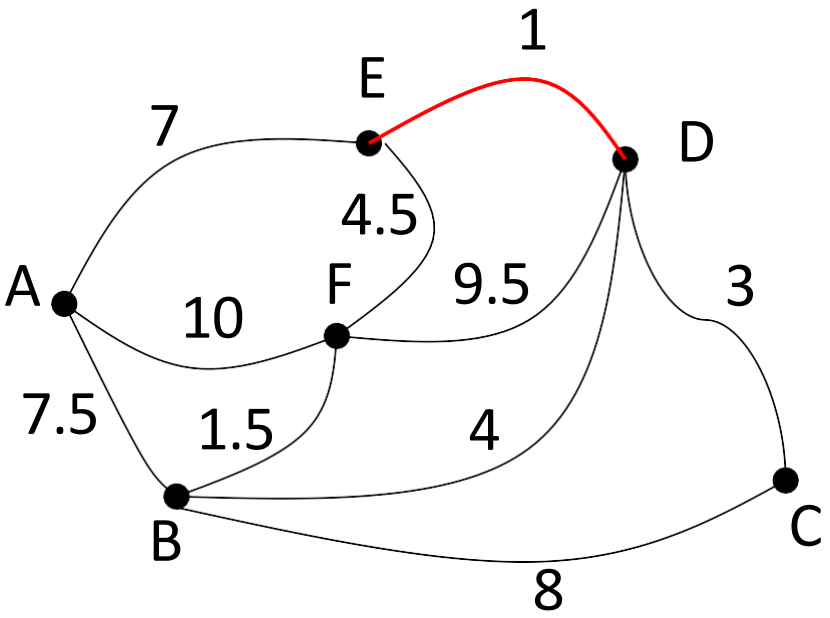
Use Kruskal's algorithm to find a minimum spanning tree for the graph.



Kruskal's Algorithm

Solution

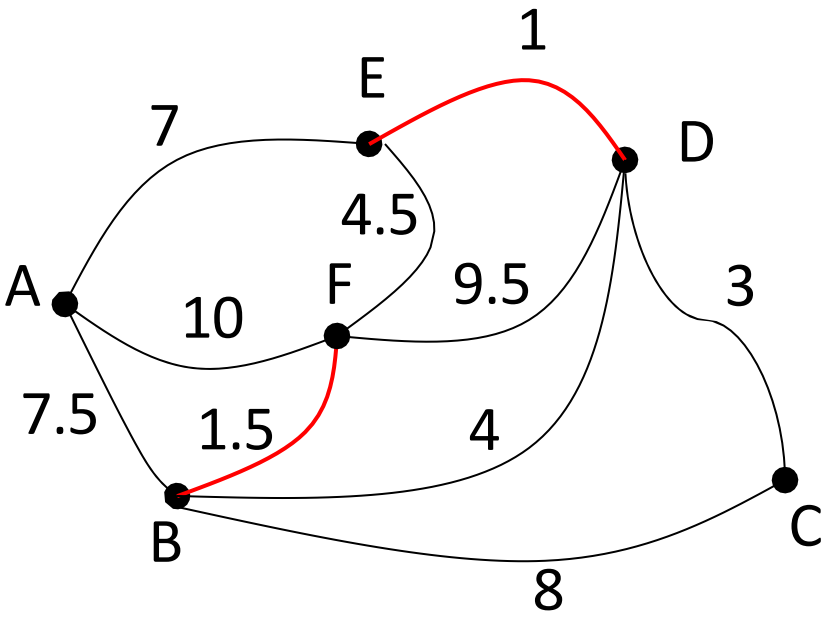
First, choose ED (the smallest weight).



Kruskal's Algorithm

Solution

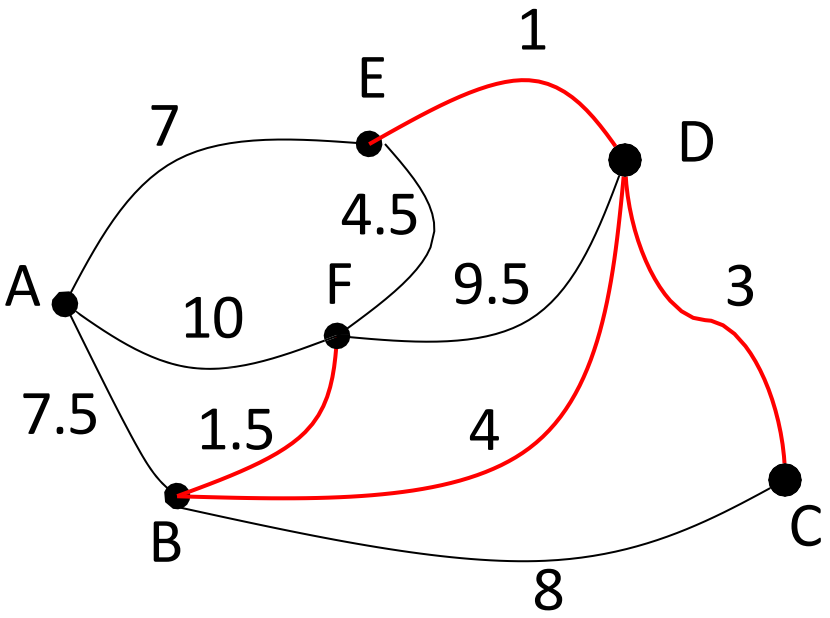
Now choose BF (the smallest remaining weight).



Kruskal's Algorithm

Solution

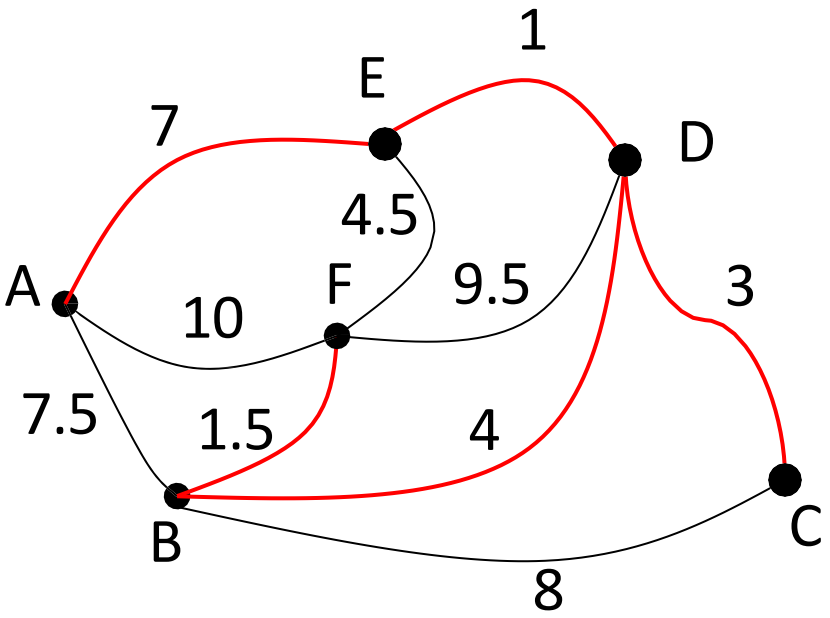
Now CD and then BD.



Kruskal's Algorithm

Solution

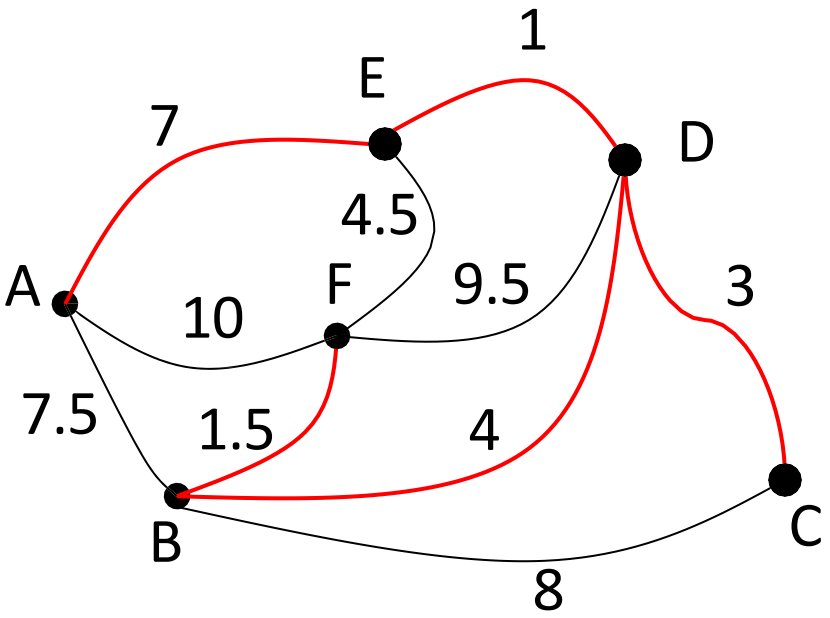
Note EF is the smallest remaining, but that would create a cycle. Choose AE and we are done.



Kruskal's Algorithm

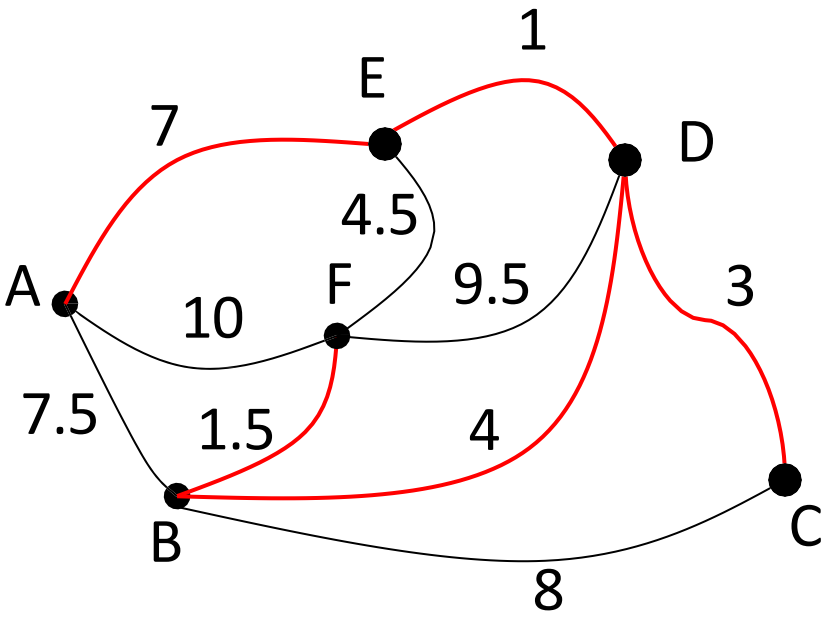
Solution

The total weight of the tree is 16.5.



Kruskal's Algorithm

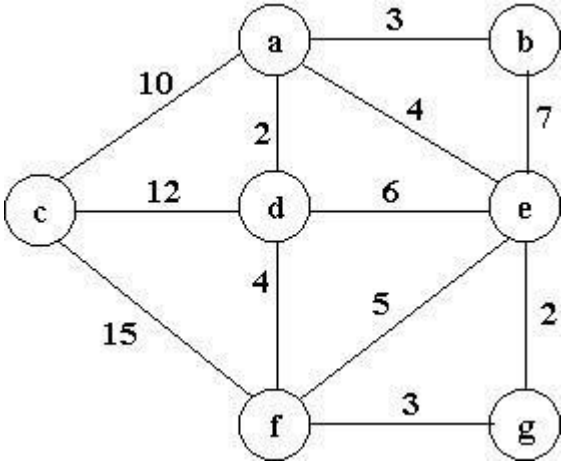
- Question:
  1. How do we know we are finished?

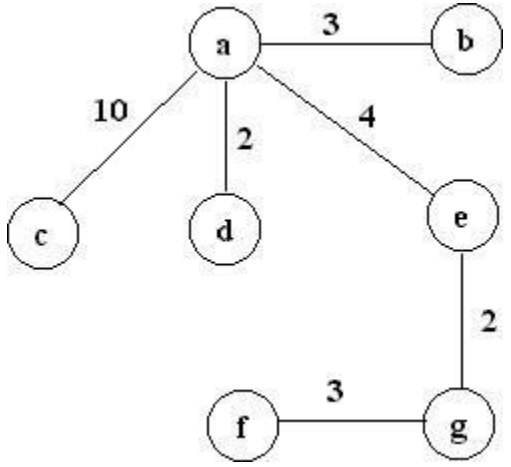


Kruskal's Algorithm

edge	ad	eg	ab	fg	ae	df	ef	de	be	ac	cd	cf
weight	2	2	3	3	4	4	5	6	7	10	12	15
insertion status	√	√	√	√	√	x	x	x	x	√	x	x
insertion order	1	2	3	4	5					6		

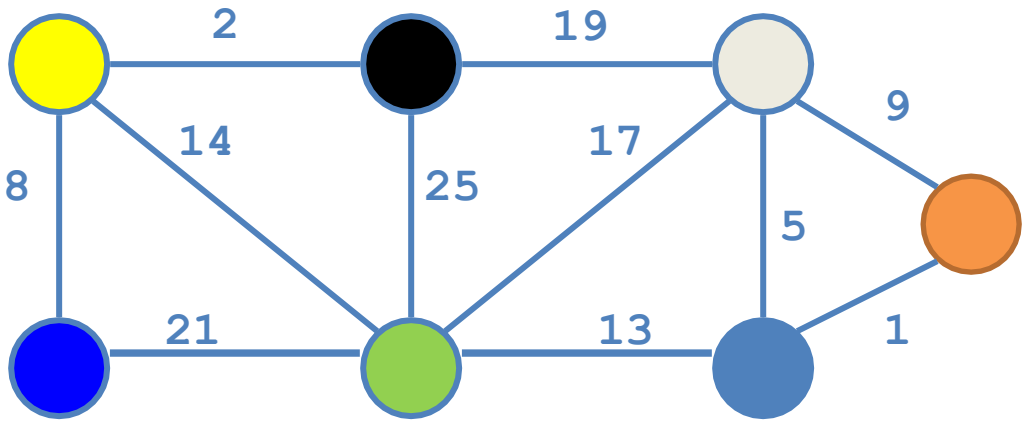
- Trace of Kruskal's algorithm for the undirected, weighted graph:



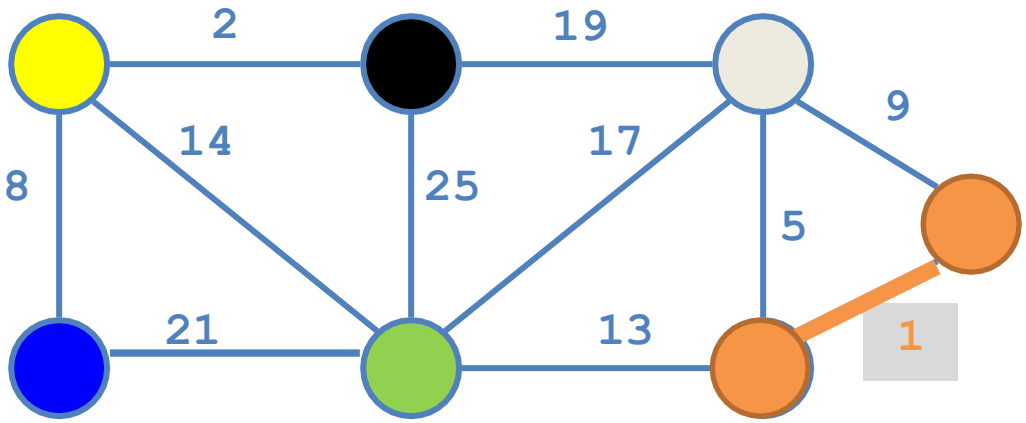


The minimum cost is: 24

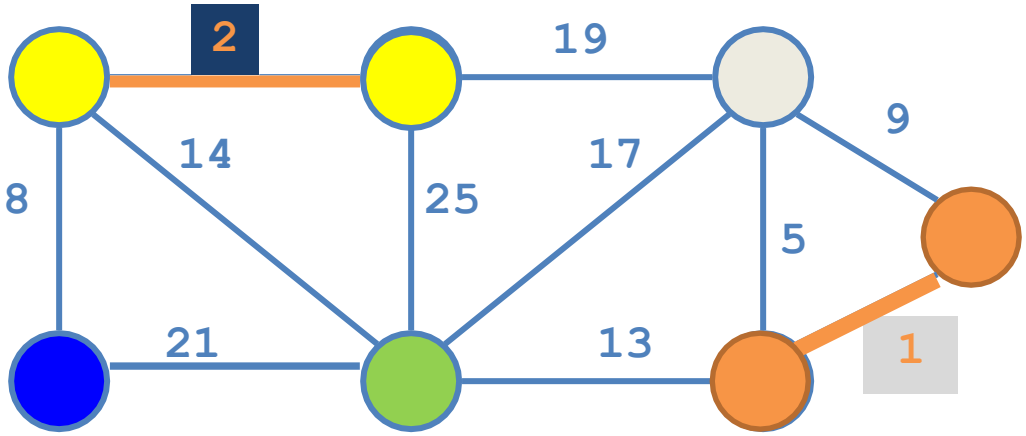
Kruskal's Algorithm



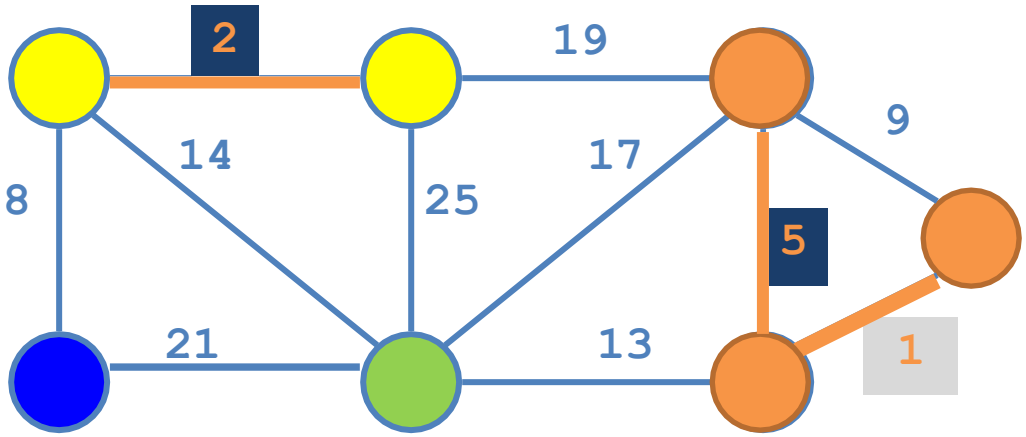
Kruskal's Algorithm



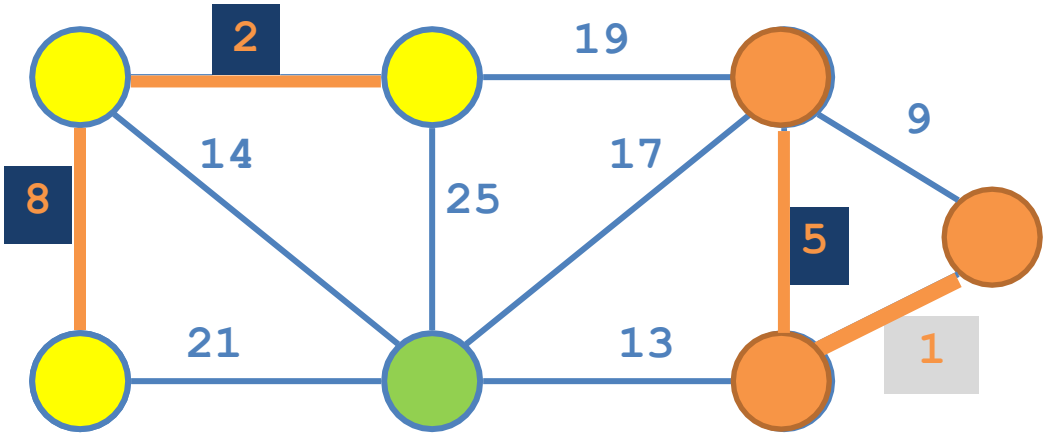
Kruskal's Algorithm



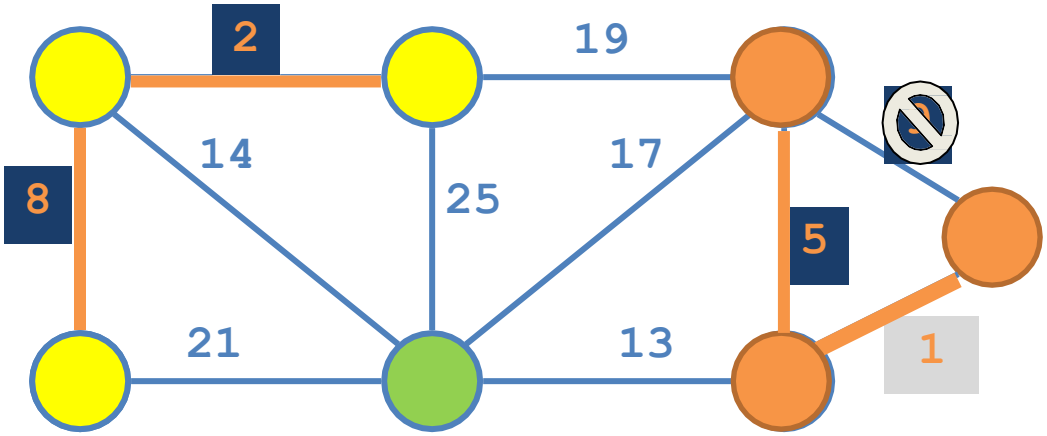
Kruskal's Algorithm



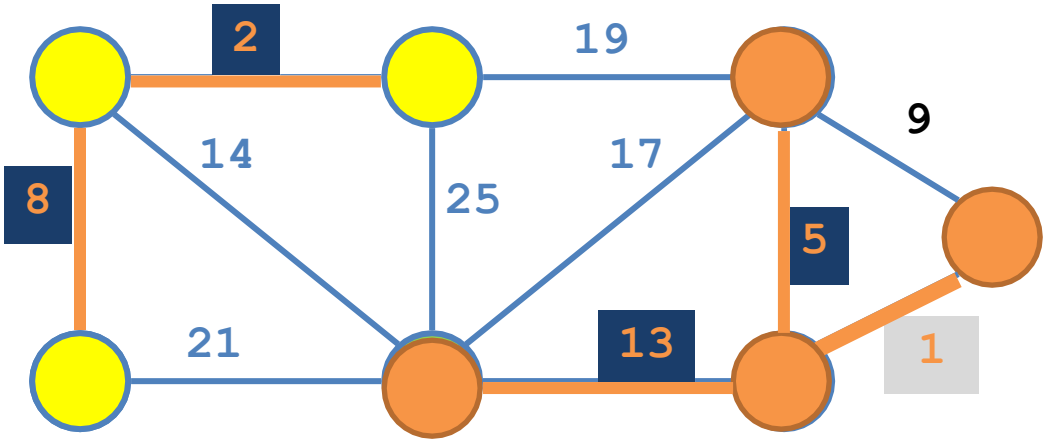
Kruskal's Algorithm



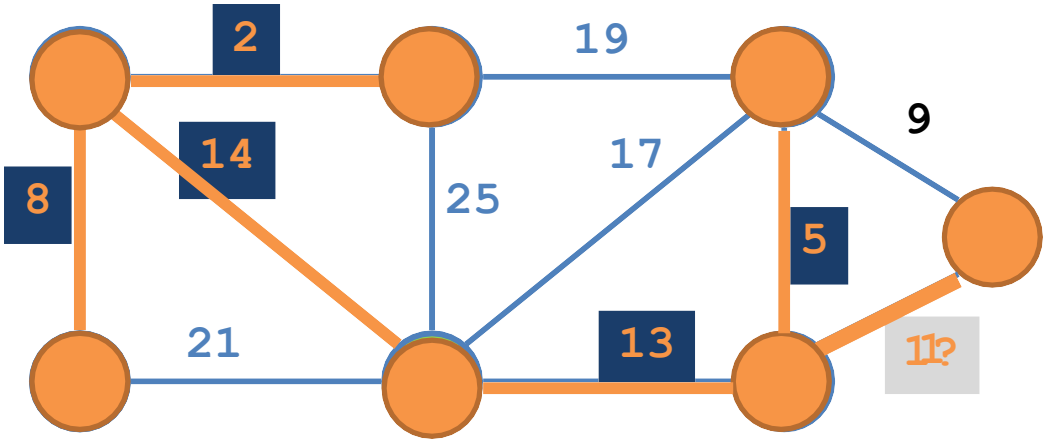
Kruskal's Algorithm



Kruskal's Algorithm

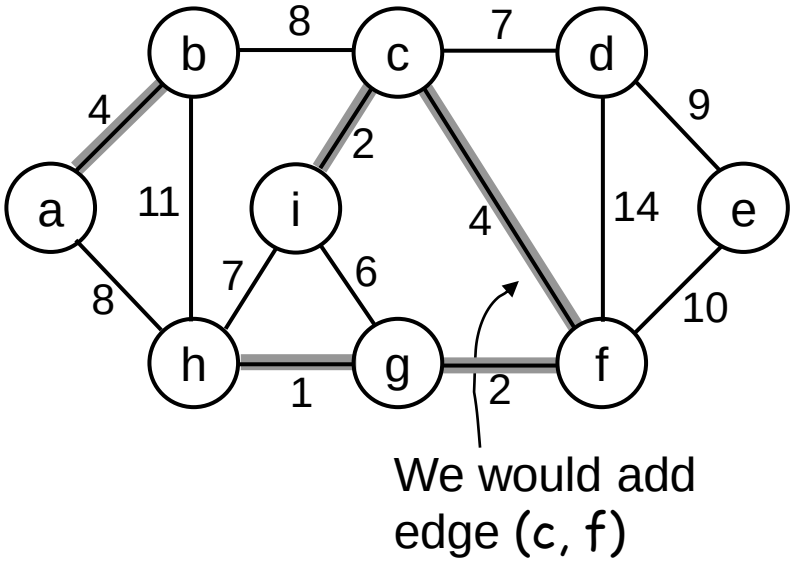


Kruskal's Algorithm



Implementation of Kruskal's Algorithm

- Uses a disjoint-set data structure (see **Chapter 21**) to determine whether an edge connects vertices in different components



Operations on Disjoint Data Sets

- MAKE-SET(u) – creates a new set whose only member is u
- FIND-SET(u) – returns a representative element from the set that contains u
  - Any of the elements of the set that has a particular property
  - *E.g.:* S_u = {r, s, t, u}, the property is that the element be the first one alphabetically

    FIND-SET(u) = r FIND-SET(s) = r
  - FIND-SET has to return the same value for a given set

Operations on Disjoint Data Sets

- UNION(u, v) – unites the dynamic sets that contain u and v, say S_u and S_v
  - *E.g.:* $S_u = \{r, s, t, u\}$, $S_v = \{v, x, y\}$

    UNION (u, v) = {r, s, t, u, v, x, y}
- Running time for FIND-SET and UNION depends on implementation.
- Can be shown to be **α(n)=O(lgn)** where **α()** is a very slowly growing function (see **Chapter 21**)

KRUSKAL(V, E, w)

1. $A \leftarrow \varnothing$
2. **for** each vertex $v \in V$ } **O(V)**
3. **do** MAKE-SET(v) }
4. sort E into non-decreasing order by w } **O(ElgE)**
5. **for** each (u, v) taken from the sorted list ← **O(E)**
6. **do if** FIND-SET(u) ≠ FIND-SET(v) }
7. **then** $A \leftarrow A \cup \{(u, v)\}$ } ← **O(lgV)**
8. UNION(u, v) }
9. **return** A

Running time: $O(V+ElgE+ElgV)=O(ElgE)$ – dependent on the implementation of the disjoint-set data structure

KRUSKAL(V, E, w) (cont.)

1. $A \leftarrow \varnothing$
2. **for** each vertex $v \in V$
3. **do** MAKE-SET(v)

} **O(V)**

4. sort E into non-decreasing order by w } **O(ElgE)**
5. **for** each (u, v) taken from the sorted list ← **O(E)**
6. **do if** FIND-SET(u) ≠ FIND-SET(v)
7. **then** $A \leftarrow A \cup \{(u, v)\}$
8. UNION(u, v)

← **O(lgV)**

9. **return** A

- Running time: $O(V+ElgE+ElgV)=O(ElgE)$
- Since $E=O(V^2)$, we have $lgE=O(2lgV)=O(lgV)$

O(ElgV)

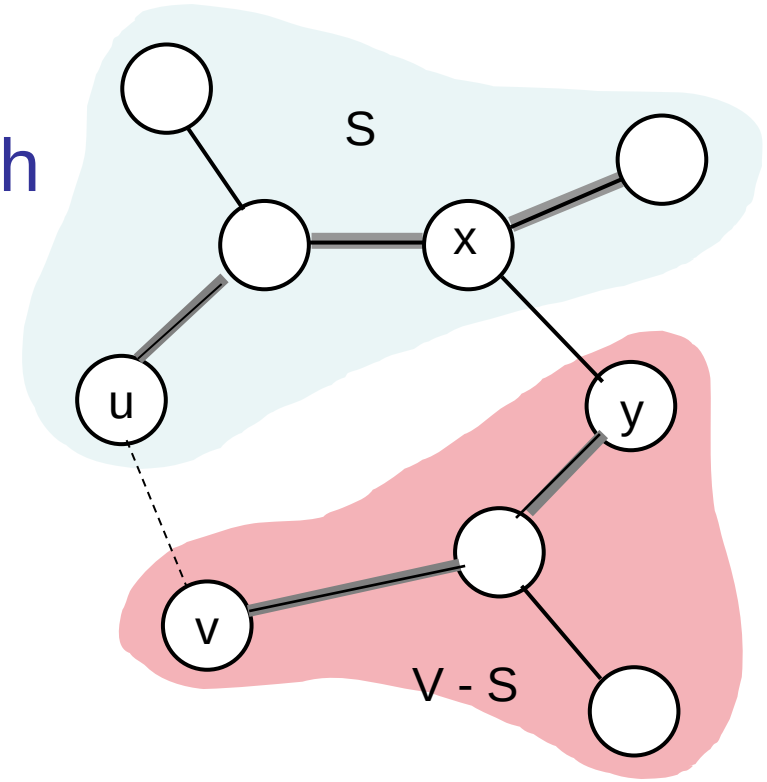
Kruskal's Algorithm – Time complexity

- Steps
  - Initialize forest $O(|V|)$
  - Sort edges $O(|E| \log |E|)$
    - Check edge for cycles $O(|V|)$ x
    - Number of edges $O(|V|)$ $O(|V|^2)$
  - Total $O(|V| + |E| \log |E| + |V|^2)$
  - Since $|E| = O(|V|^2)$ $O(|V|^2 \log |V|)$

  - Thus we would class MST as $O(n^2 \log n)$ for a graph with n vertices
  - This is an *upper bound*, some improvements on this are known.

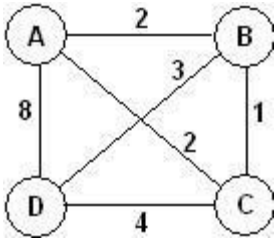
Kruskal's Algorithm

- Kruskal's algorithm is a **"greedy"** algorithm
- Kruskal's greedy strategy produces a globally optimum solution
- Proof for generic approach applies to Kruskal's algorithm too

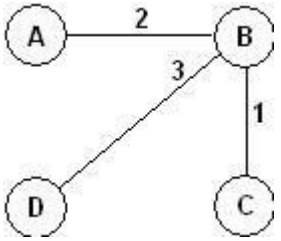


Prim's and Kruskal's Algorithms

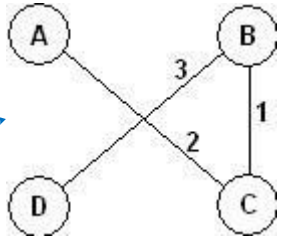
- It is not necessary that Prim's and Kruskal's algorithm generate the same minimum-cost spanning tree.
- For example for the graph shown on the right:



- Kruskal's algorithm results in the following minimum cost spanning tree:
  - The same tree is generated by Prim's algorithm if the start vertex is any of: A, B, or D.



- However if the start vertex is C the minimum cost spanning tree generated by Prim's algorithm is:



Problem 1

- **(Exercise 23.2-3, page 573)** Compare Prim's algorithm with and Kruskal's algorithm assuming:

(a) sparse graphs:

In this case, E=O(V)

Kruskal:

$O(ElgE)=O(VlgV)$

Prim:

- binary heap: $O(ElgV)=O(VlgV)$
- Fibonacci heap: $O(VlgV+E)=O(VlgV)$

Problem 1 (cont.)

(b) dense graphs

In this case, $E=O(V^2)$

Kruskal:

$O(ElgE)=O(V^2lgV^2)=O(2V^2lgV)=O(V^2lgV)$

Prim:

- binary heap: $O(ElgV)=O(V^2lgV)$
- Fibonacci heap: $O(VlgV+E)=O(VlgV+V^2)=O(V^2)$

Problem 2

- **(Exercise 23.2-4, page 574):** Analyze the running time of Kruskal's algorithm when weights are in the range **[1 … V].**

Problem 2 (cont.)

1. $A \leftarrow \varnothing$
2. **for** each vertex $v \in V$ } **O(V)**
3. **do** MAKE-SET(v) }
4. sort E into non-decreasing order by w } **O(ElgE)**
5. **for** each (u, v) taken from the sorted list ← **O(E)**
6. **do if** FIND-SET(u) ≠ FIND-SET(v)
7. **then** $A \leftarrow A \cup \{(u, v)\}$ ← **O(lgV)**
8. UNION(u, v)
9. **return** A

- Sorting can be done in O(E) time (e.g., using **counting sort**)
- However, overall running time will not change, i.e, O(ElgV)

Problem 3

- Suppose that some of the weights in a connected graph G are negative. Will Prim's algorithm still work? What about Kruskal's algorithm? Justify your answers.
  - Yes, both algorithms will work with negative weights. Review the proof of the generic approach; there is no assumption in the proof about the weights being positive.

Problem 4

- **(Exercise 23.2-2, page 573)** Analyze Prim's algorithm assuming:

(a) an adjacency-list representation of G

$O(E.\lg V)$

(b) an adjacency-matrix representation of G

$O(E.\lg V+V^2)$

PRIM(V, E, w, r)

```
1.   Q ← ∅
2.   for each u ∈ V
3.        do key[u] ← ∞
4.           π[u] ← NIL
5.           INSERT(Q, u)
6.   DECREASE-KEY(Q, r, 0)        ▶ key[r] ← 0
7.   while Q ≠ ∅
8.        do u ← EXTRACT-MIN(Q)
9.           for each v ∈ Adj[u]
10.               do if v ∈ Q and w(u, v) < key[v]
11.                  then π[v] ← u
12.                       DECREASE-KEY(Q, v, w(u, v))
```

Total time: $O(V \lg V + E \lg V) = O(E \lg V)$

$O(V)$ if Q is implemented as a min-heap

$O(\lg V)$

Executed |V| times

Takes $O(\lg V)$

Min-heap operations: $O(V \lg V)$

Executed O(E) times

Constant

Takes $O(\lg V)$

$O(E \lg V)$

PRIM(V, E, w, r)

```
1.   Q ← ∅
2.   for each u ∈ V
3.        do key[u] ← ∞
4.           π[u] ← NIL
5.           INSERT(Q, u)
6.   DECREASE-KEY(Q, r, 0)        ▶ key[r] ← 0
7.   while Q ≠ ∅
8.        do u ← EXTRACT-MIN(Q)
9.           for (j=0; j<|V|; j++)
10.             if (A[u][j]=1)
11.               if v ∈ Q and w(u, v) < key[v]
12.                  then π[v] ← u
13.                       DECREASE-KEY(Q, v, w(u, v))
```

$O(V)$ if Q is implemented as a min-heap

Total time: $O(V \lg V + E \lg V + V^2) = O(E \lg V + V^2)$

$O(\lg V)$

Executed |V| times

Takes $O(\lg V)$

Min-heap operations: $O(V \lg V)$

Executed $O(V^2)$ times total

Constant

Takes $O(\lg V)$

$O(E \lg V)$

Problem 5

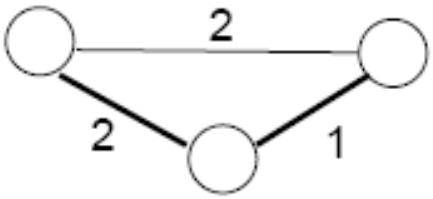
- Find an algorithm for the “maximum” spanning tree. That is, given an undirected weighted graph G, find a spanning tree of G of maximum cost. Prove the correctness of your algorithm.

Problem 5

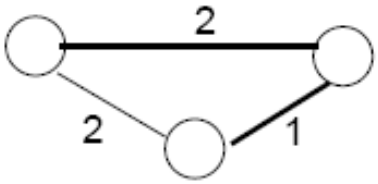
- Find an algorithm for the “maximum” spanning tree. That is, given an undirected weighted graph G, find a spanning tree of G of maximum cost. Prove the correctness of your algorithm.
  - Consider choosing the “heaviest” edge (i.e., the edge associated with the largest weight) in a cut. The generic proof can be modified easily to show that this approach will work.
  - Alternatively, multiply the weights by -1 and apply either Prim’s or Kruskal’s algorithms without any modification at all!

Problem 6

- **(Exercise 23.1-8, page 567)** Let T be a MST of a graph G, and let L be the sorted list of the edge weights of T. Show that for any other MST T' of G, the list L is also the sorted list of the edge weights of T'.



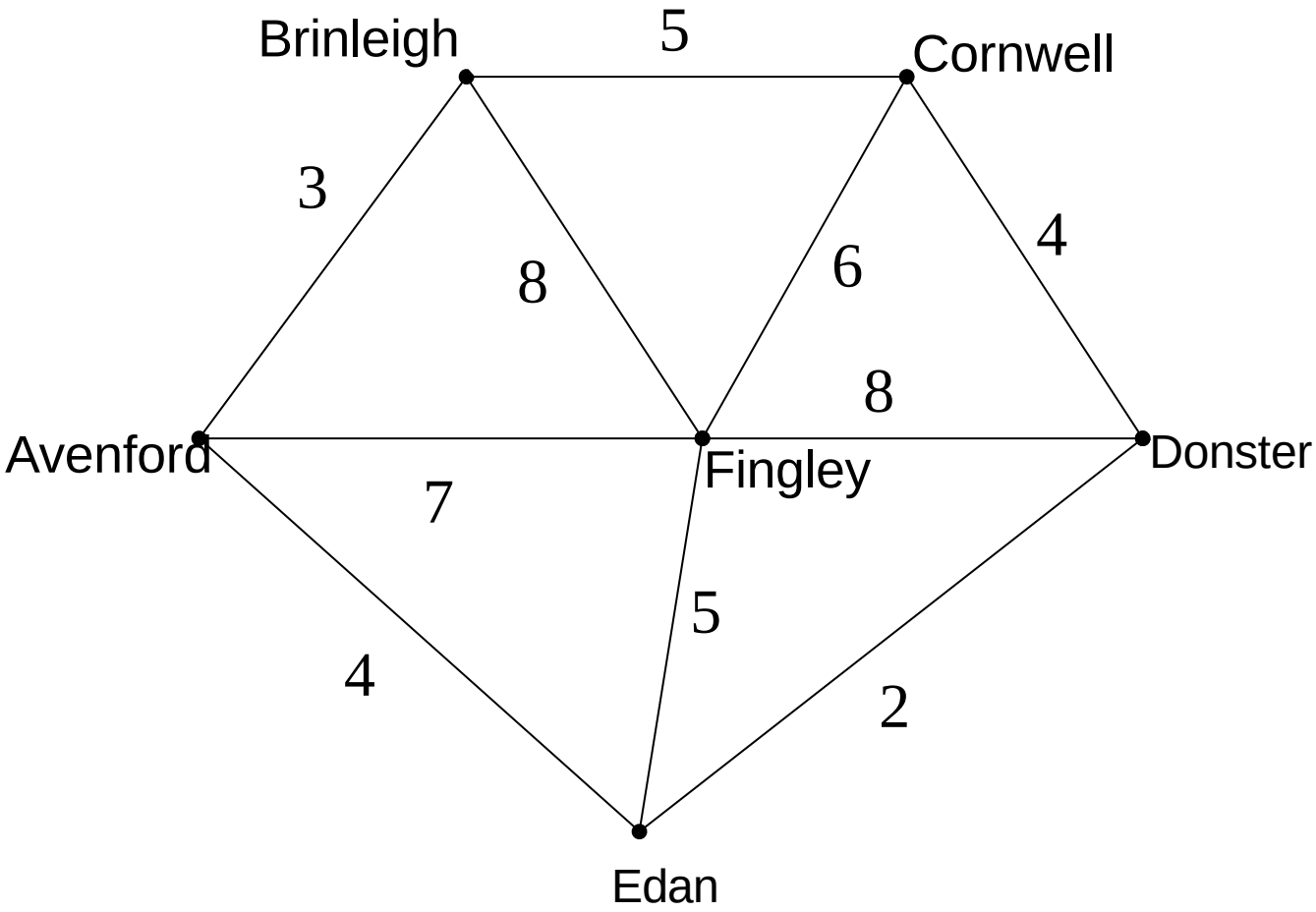
T, L={1,2}



T', L={1,2}

Problem 7: Prim's algorithm with an Adjacency Matrix

A cable company want to connect five villages to their network, which currently extends to the market town of Avenford. What is the minimum length of cable needed?



Problem 7: Prim's algorithm with an Adjacency Matrix

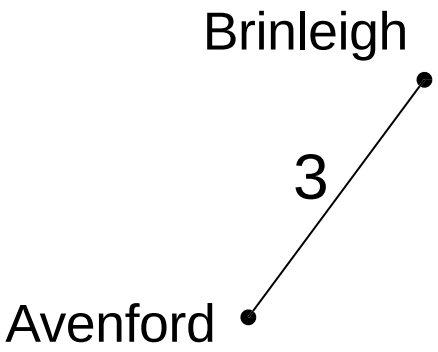
Note, this example has outgoing edges on the columns and incoming on the rows, so it is the transpose of adjacency matrix mentioned in class. Actually, it is an undirected, so $A^T = A$.

	A	B	C	D	E	F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-

Problem 7: Prim's algorithm with an Adjacency Matrix

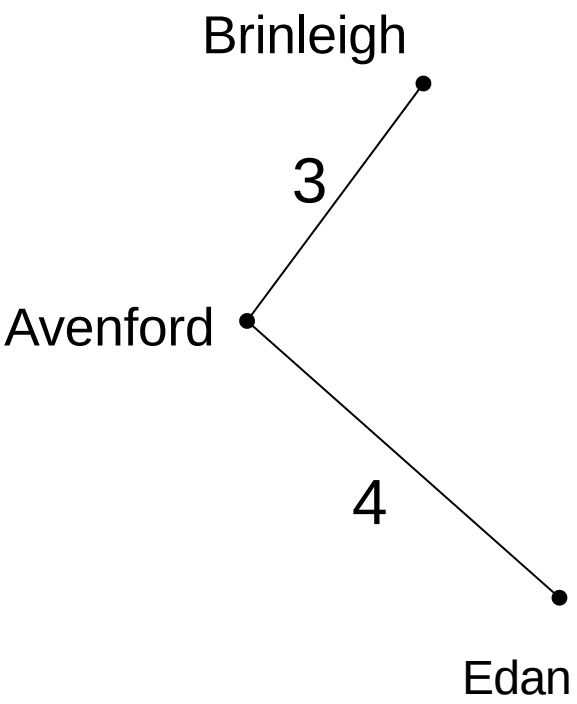
- Start at vertex A. Label column A "1" .
- Delete row A
- Select the smallest entry in column A (AB, length 3)

	1					
	A	B	C	D	E	F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-



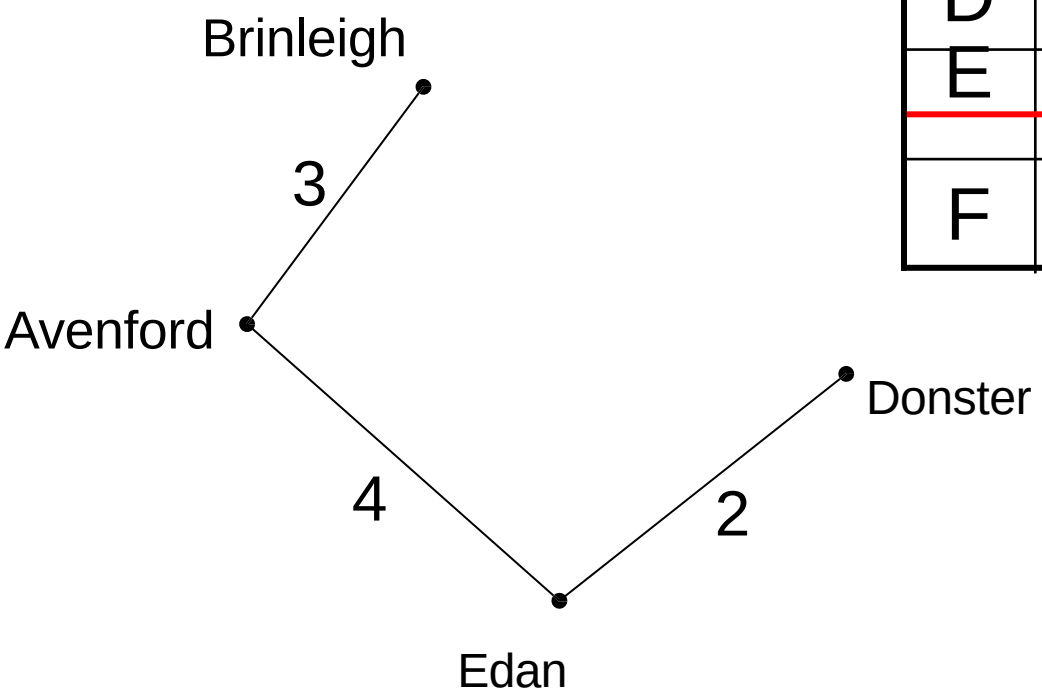
- Label column B “2”
- Delete row B
- Select the smallest uncovered entry in either column A or column B (AE, length 4)

	1 A	2 B	C	D	E	F
~~A~~	~~-~~	~~3~~	~~-~~	~~-~~	~~4~~	~~7~~
~~B~~	3	~~-~~	~~5~~	~~-~~	~~-~~	~~8~~
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-



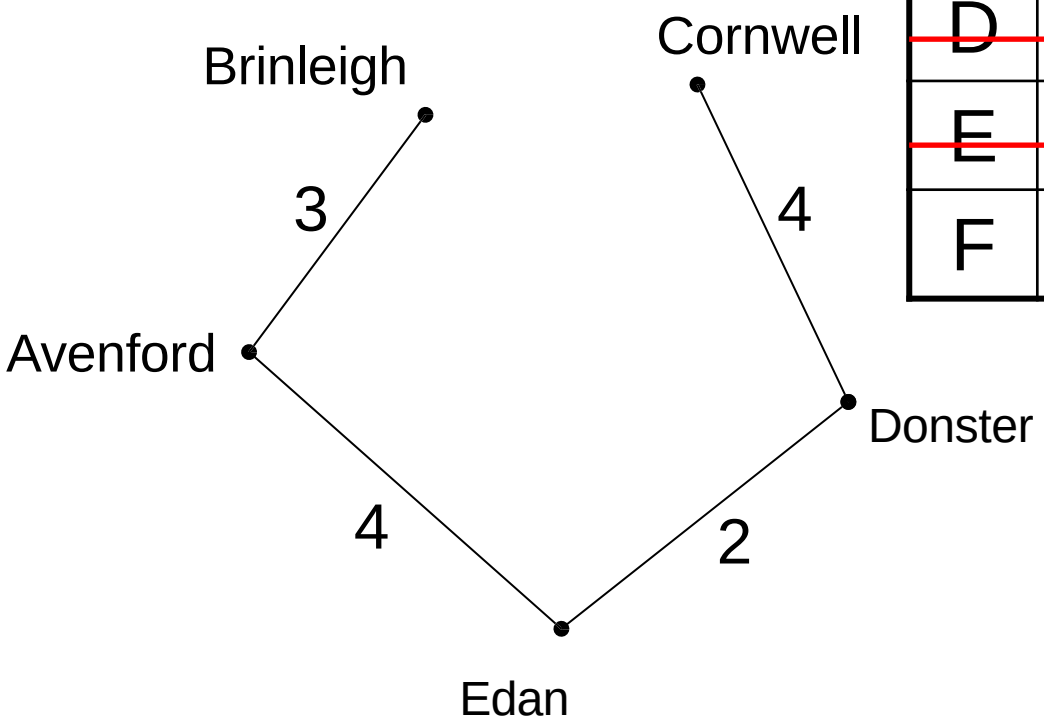
- Label column E “3”
- Delete row E
- Select the smallest uncovered entry in either column A, B or E (ED, length 2)

	1 A	2 B	C	D	3 E	F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-



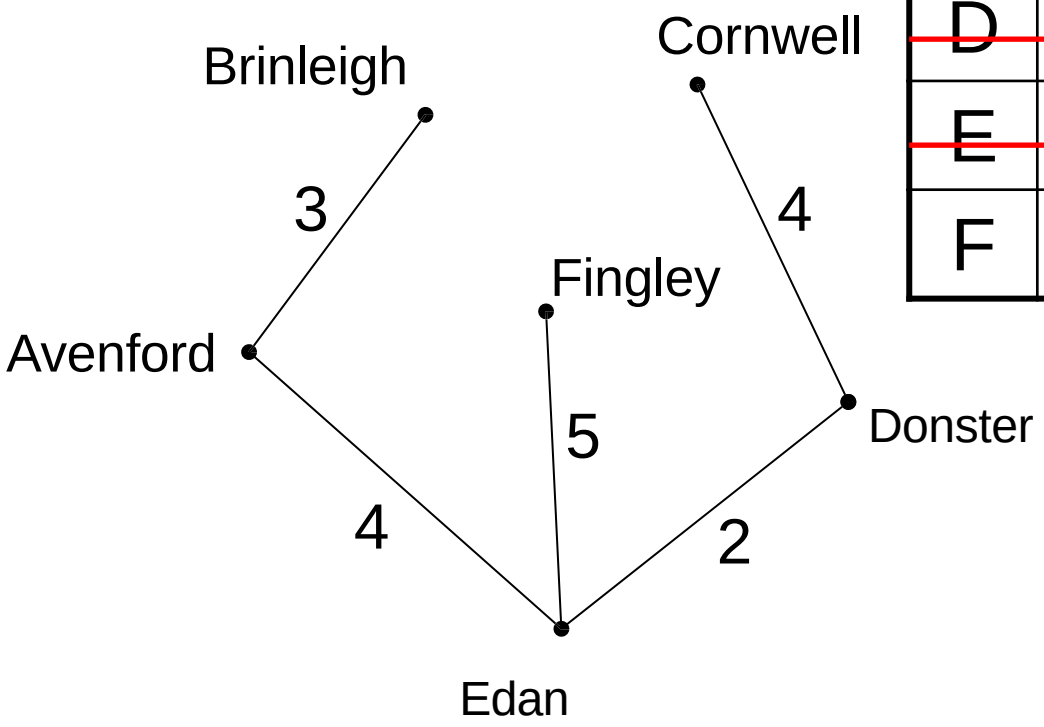
- Label column D “4”
- Delete row D
- Select the smallest uncovered entry in either column A, B, D or E (DC, length 4)

	1 A	2 B	C	4 D	3 E	F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-



- **Label column C “5”**
- **Delete row C**
- **Select the smallest uncovered entry in either column A, B, D, E or C (EF, length 5)**

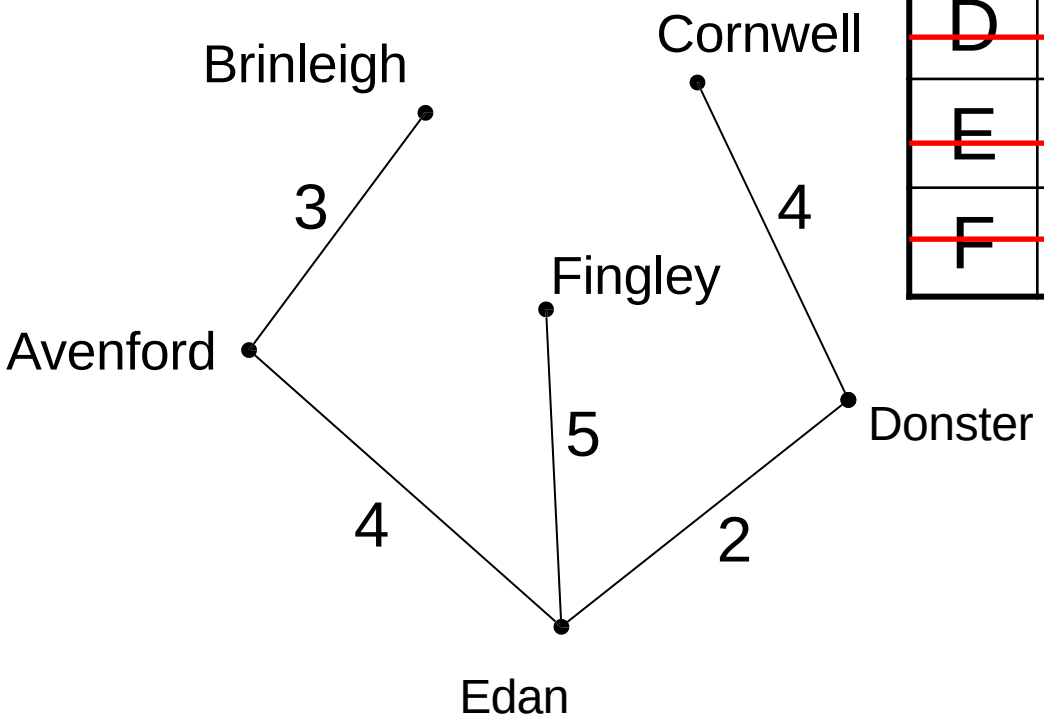
	1 A	2 B	5 C	4 D	3 E	F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-



FINALLY

- Label column F “6”
- Delete row F

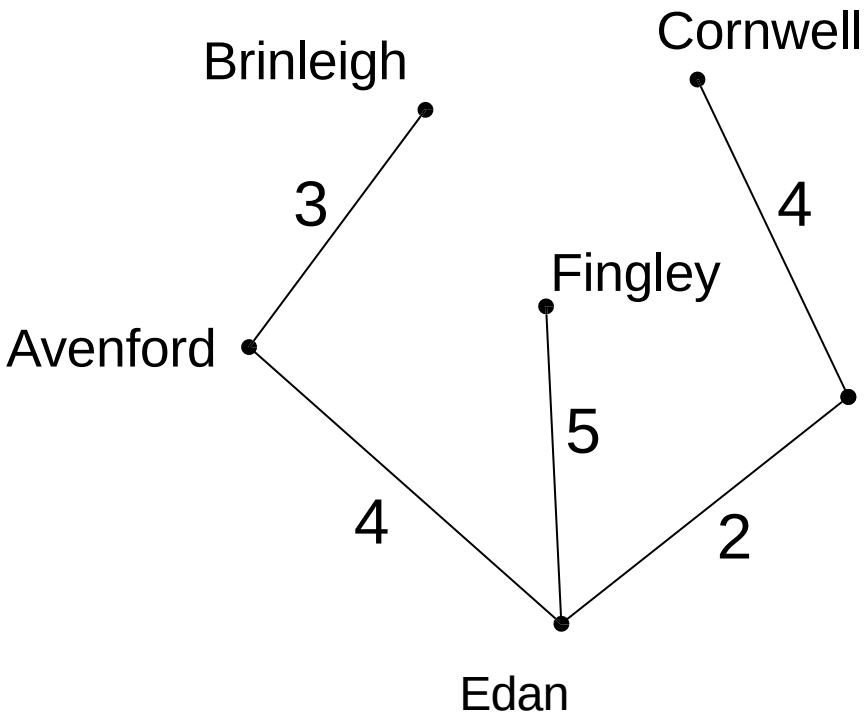
	1 A	2 B	5 C	4 D	3 E	6 F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-



FINALLY

- Label column F “6”
- Delete row F

	1 A	2 B	5 C	4 D	3 E	6 F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-

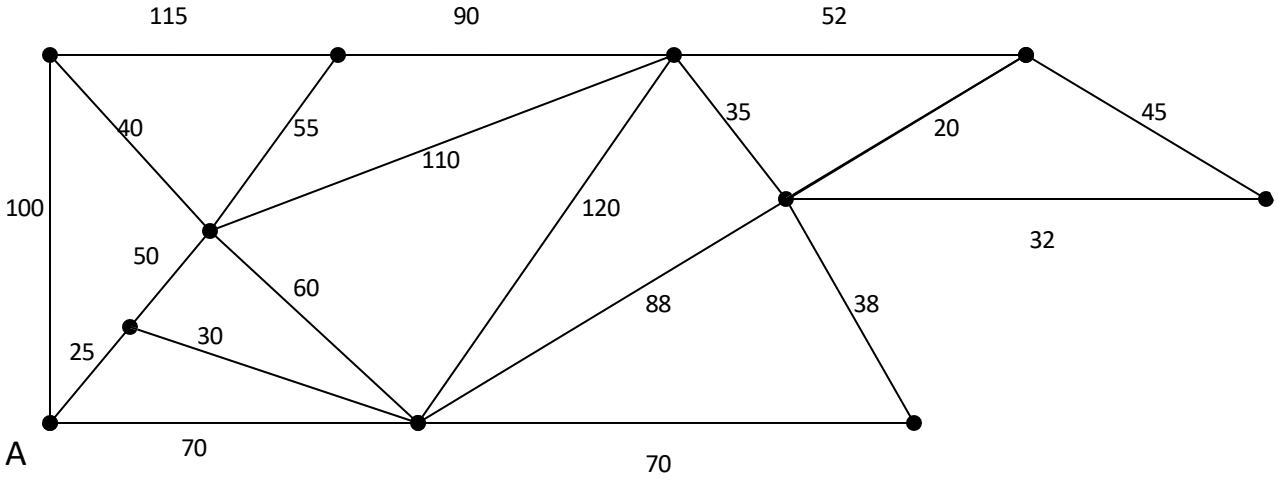


The spanning tree is shown in the diagram

Length 3 + 4 + 4 + 2 + 5 = 18Km

Quiz 1

- Find the minimum spanning tree using Kruskal's Algorithm.

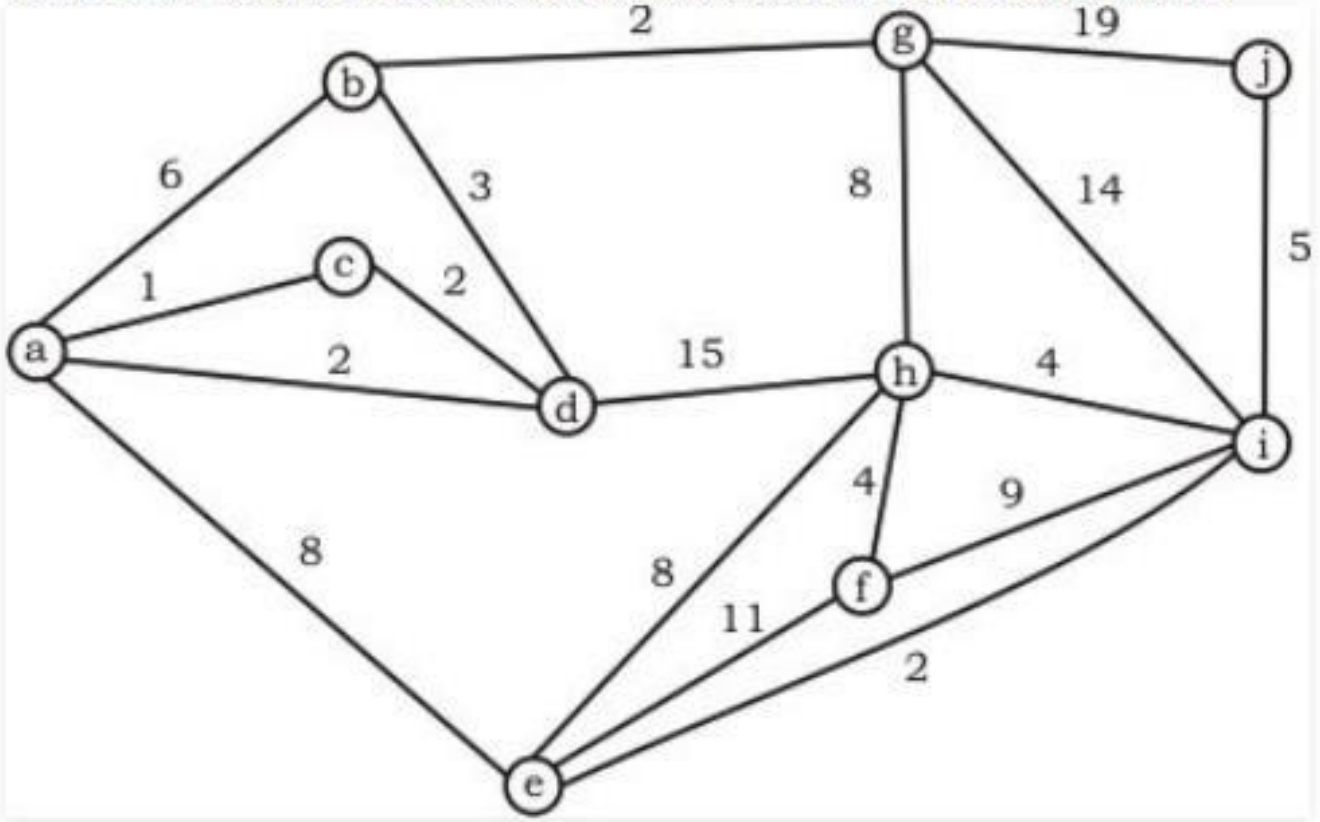


List the edges in increasing order:

20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120

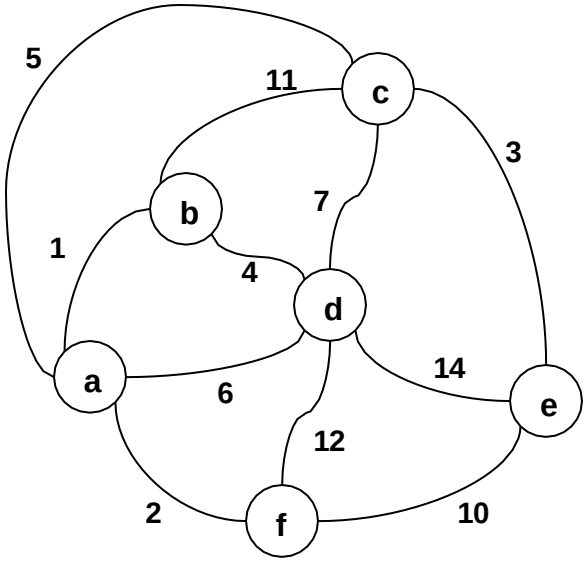
Quiz 2

What is the weight of a minimum spanning tree of the following graph ?

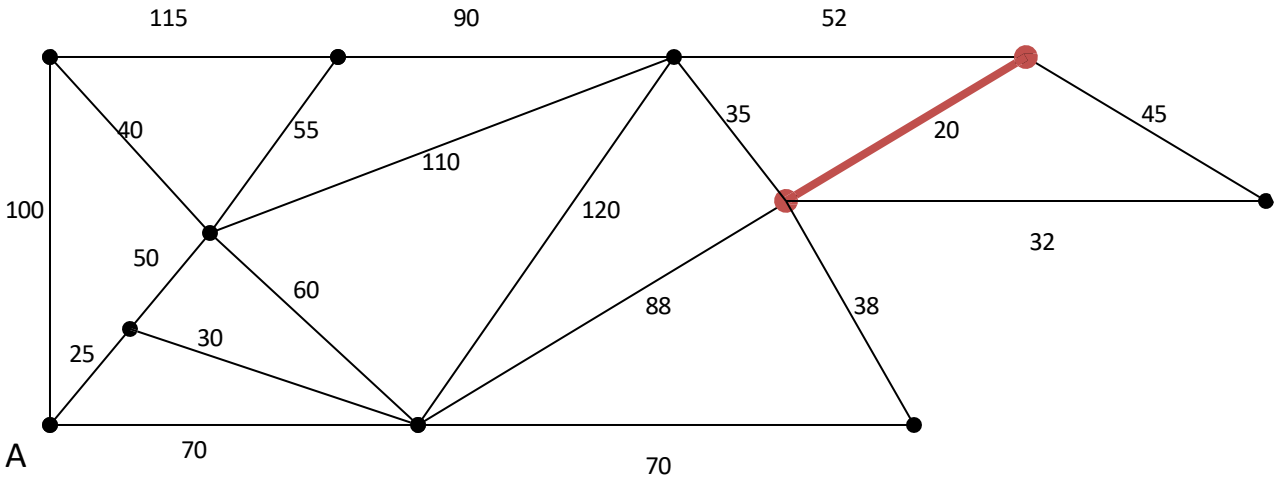


Quiz 3

- Generate 2 minimum spanning tree's for the following graph using Prim's and Kruskal's algorithms

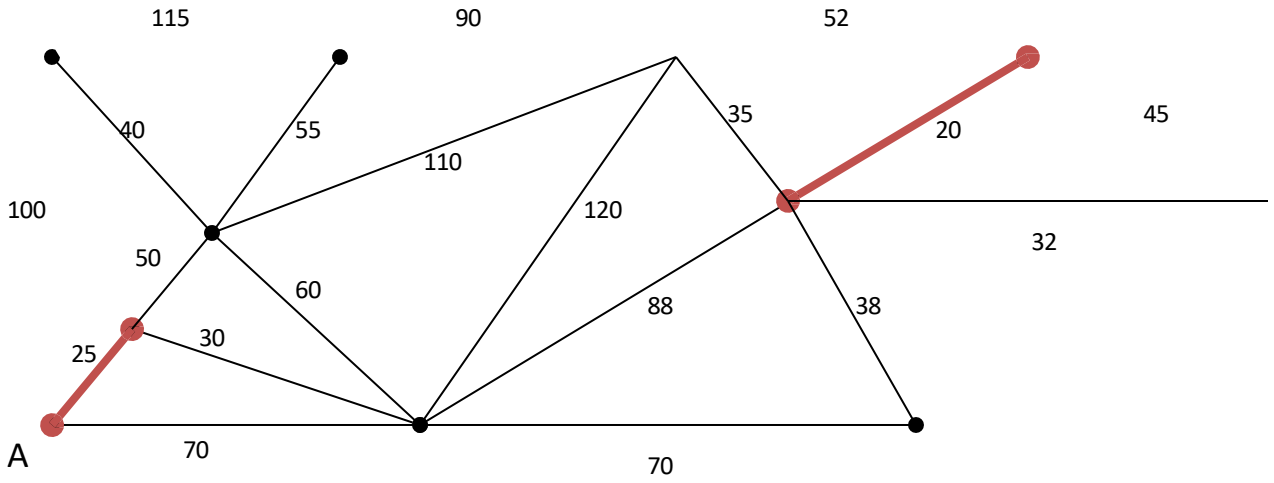


Solution to Quiz 1



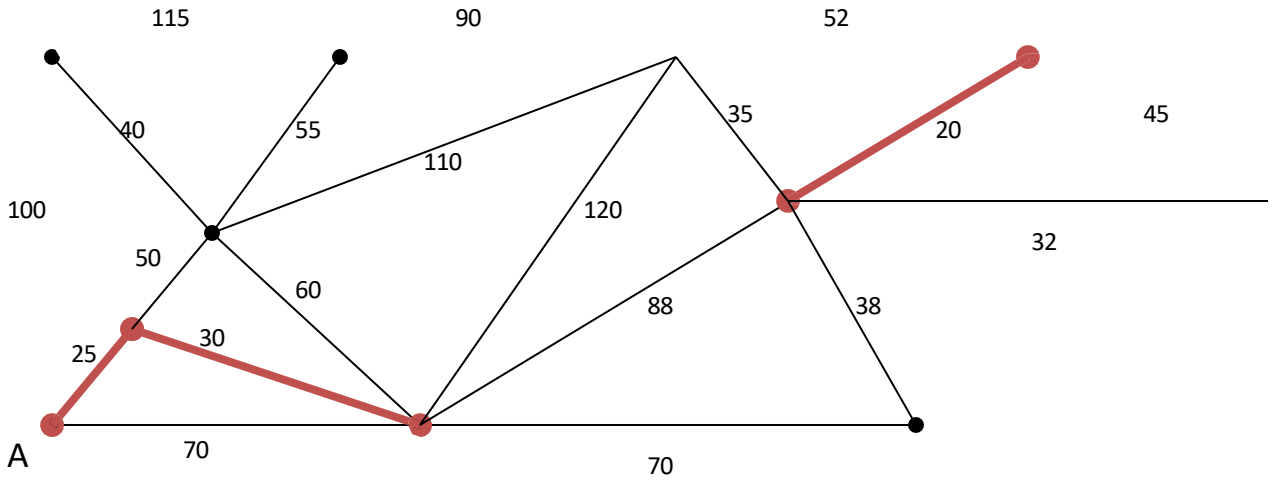
Starting from the left, add the edge to the tree if it does not close up a circuit with the edges chosen up to that point:

20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120



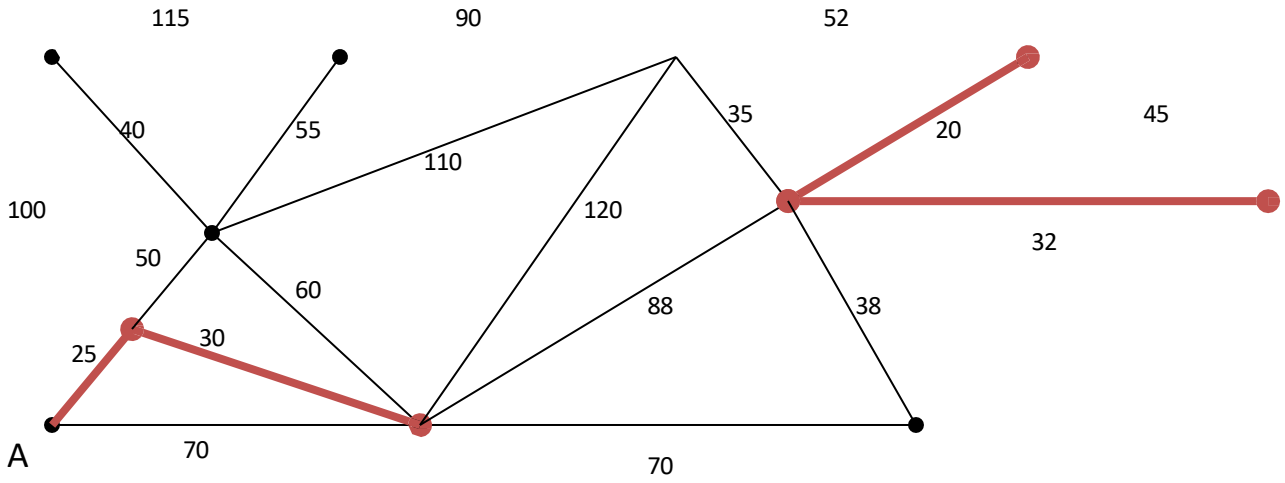
Add the next edge in the list to the tree if it does not close up a circuit with the edges chosen up to that point:

20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120



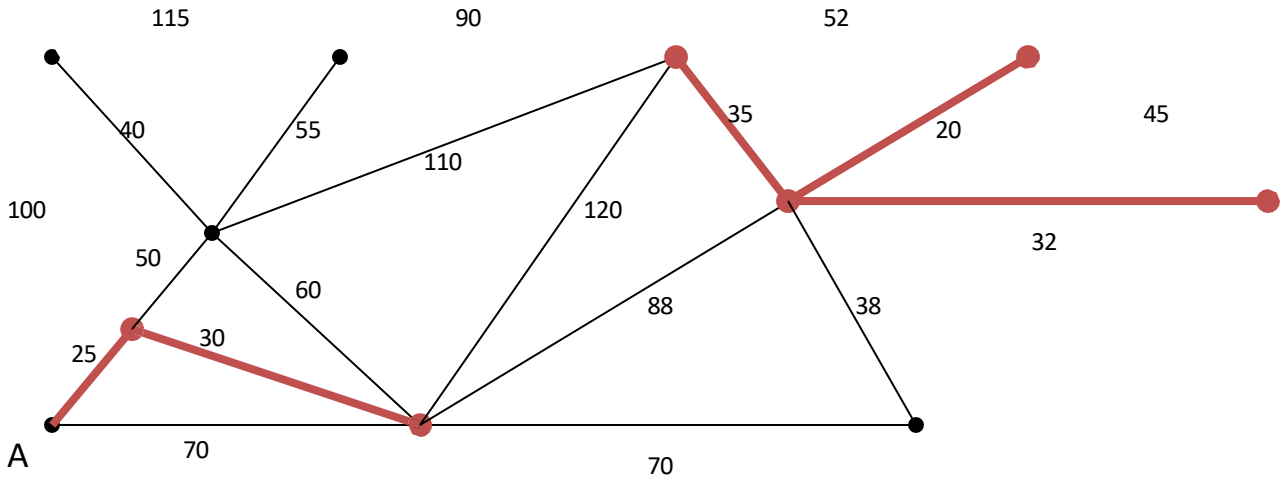
Add the next edge in the list to the tree if it does not close up a circuit with the edges chosen up to that point:

20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120



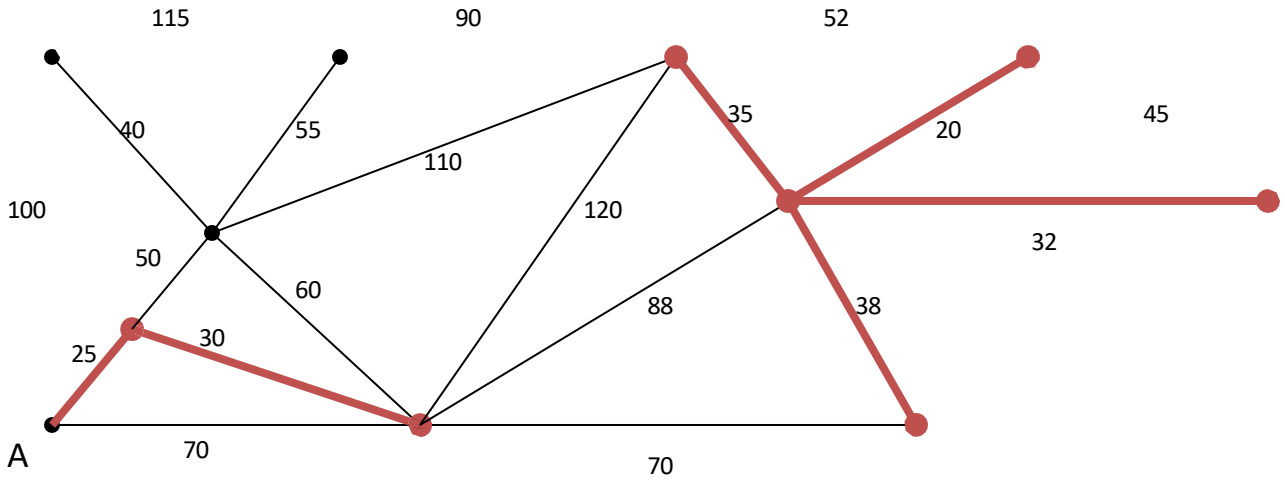
Add the next edge in the list to the tree if it does not close up a circuit with the edges chosen up to that point:

20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120



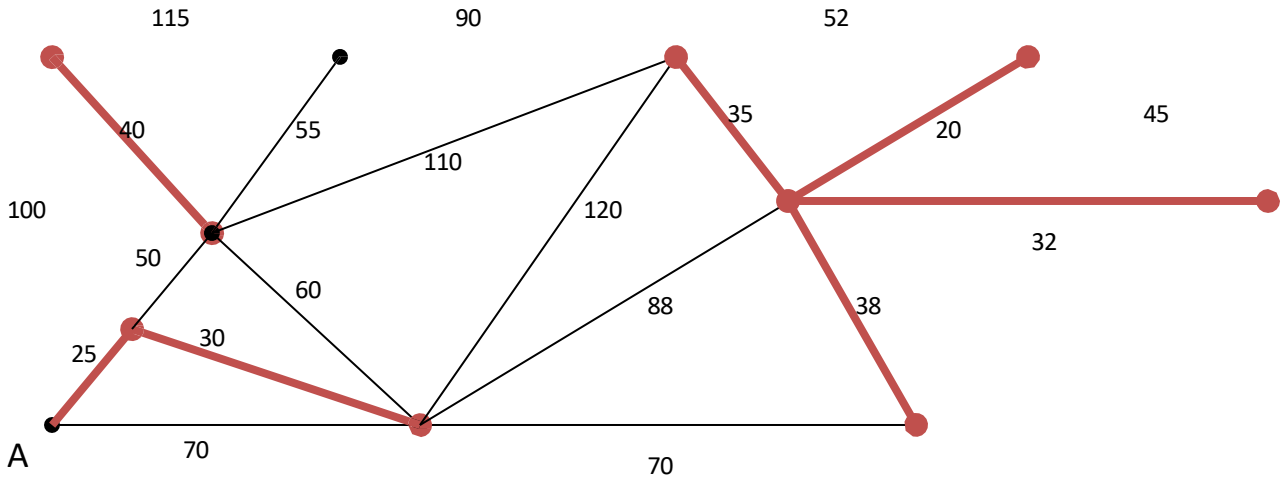
Add the next edge in the list to the tree if it does not close up a circuit with the edges chosen up to that point:

20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120



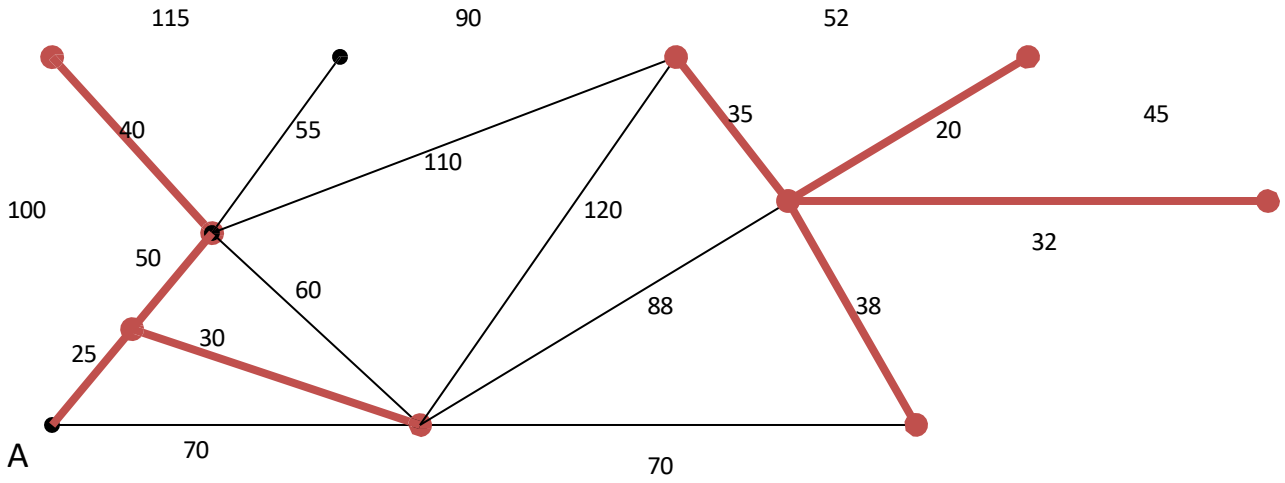
Add the next edge in the list to the tree if it does not close up a circuit with the edges chosen up to that point:

20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120



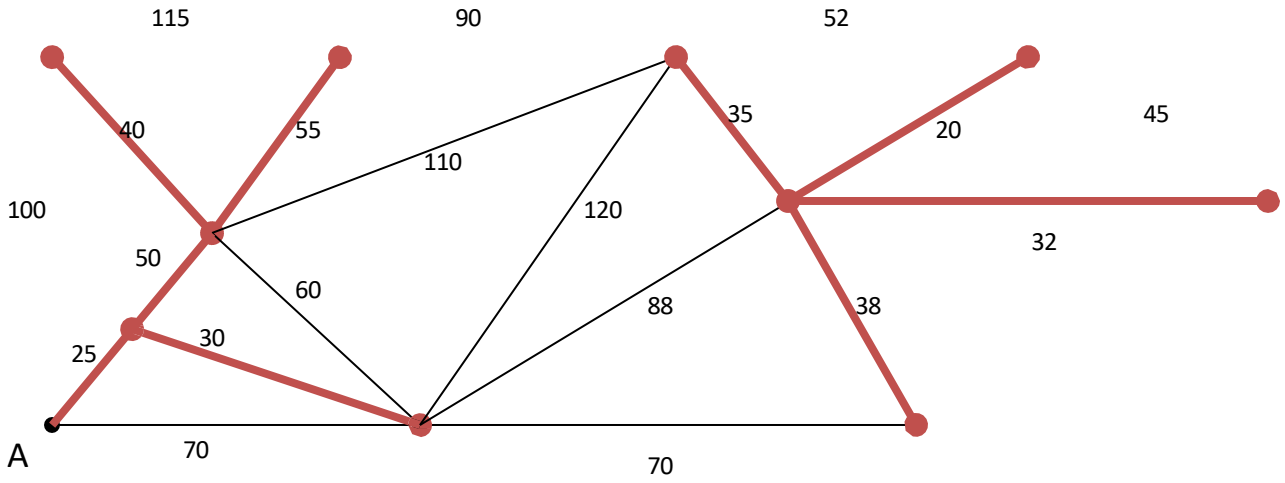
Add the next edge in the list to the tree if it does not close up a circuit with the edges chosen up to that point:

20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120



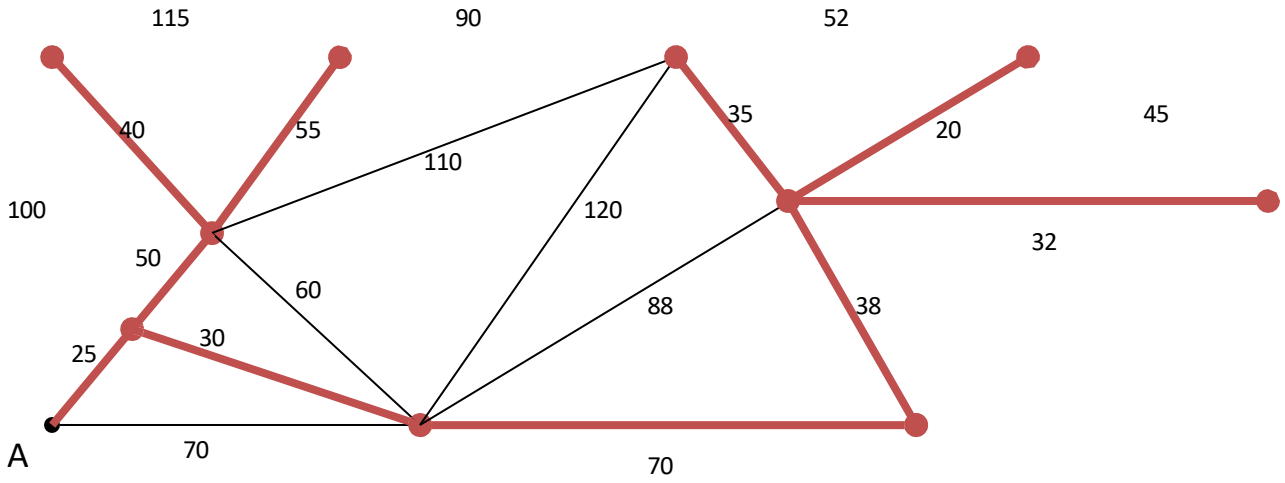
Add the next edge in the list to the tree if it does not close up a circuit with the edges chosen up to that point. Notice that the edge of weight 45 would close a circuit, so we skip it.

20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120



Add the next edge in the list to the tree if it does not close up a circuit with the edges chosen up to that point:

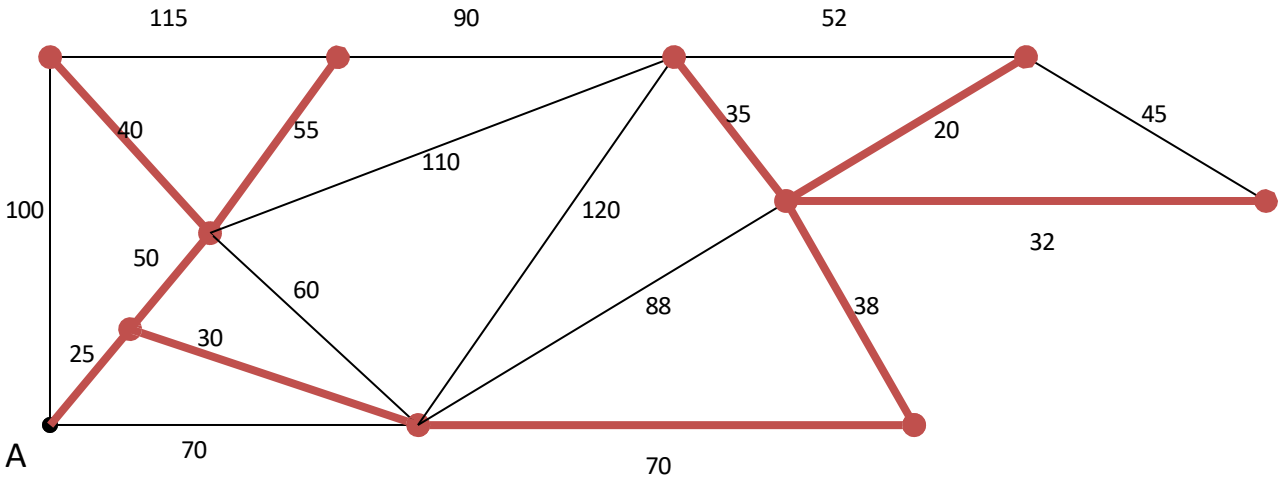
20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120



Add the next edge in the list to the tree if it does not close up a circuit with the edges chosen up to that point:

20, 25, 30, 32, 35, 38, 40, 45, 50, 52, 55, 60, 70, 70, 88, 90, 100, 110, 115, 120

Done !



The tree contains every vertex, so it is a spanning tree. The total weight is 395

Summary Kruskal *vs.* Prim

- Both are Greedy algorithms
  - Both take the next minimum edge
  - Both are optimal (find the global min)
- Different sets of edges considered
  - Kruskal – all edges
  - Prim – Edges from Tree nodes to rest of G.
- Both need to check for cycles
- Both can terminate early
- Kruskal is order of $O(|E| \log |V|)$
- Prim is order of is $O(|V|^2)$ (adjacency matrix implementation).