Minimum spanning tree

Textbook Ch 23

Outline

- Definition and applications
- Prim’s algorithm
- Kruskal’s algorithm

Spanning trees

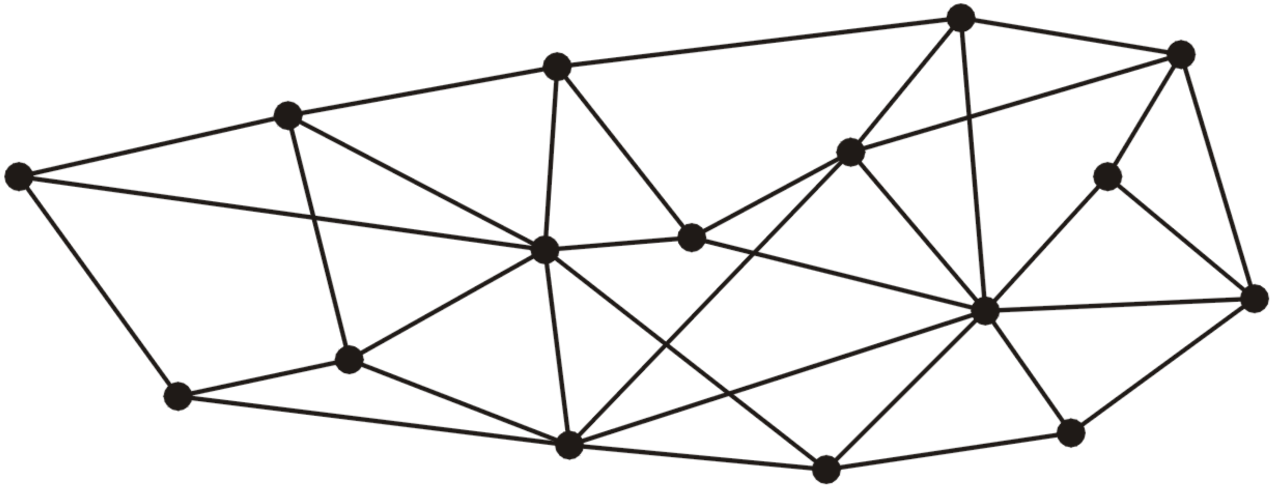
Given a connected graph with n vertices, a spanning tree is defined as a subgraph that is a tree and includes all the n vertices

– It has $n - 1$ edges

A spanning tree is not necessarily unique

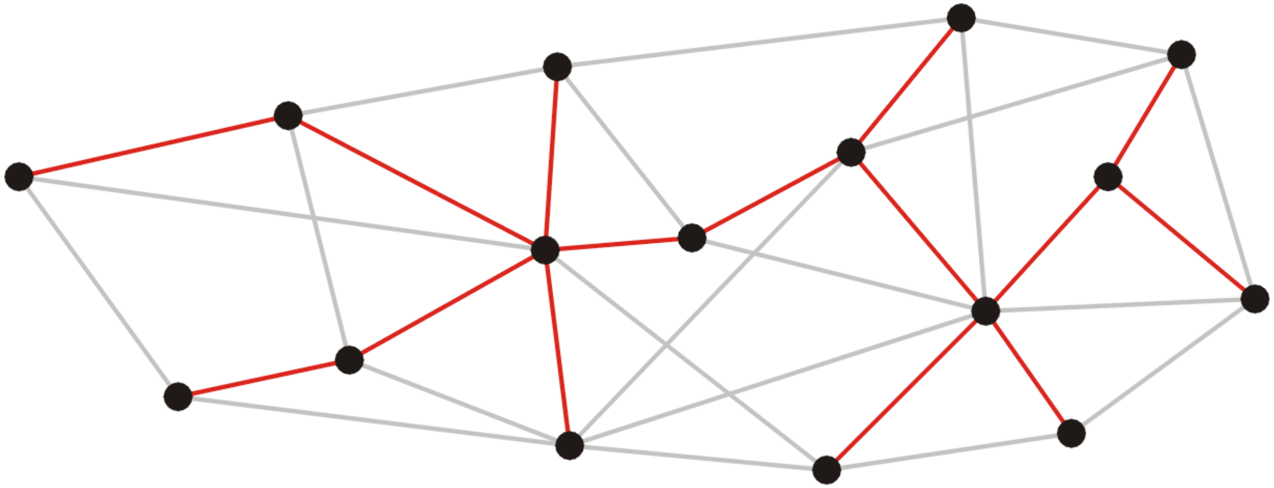
Spanning trees

This graph has 16 vertices and 35 edges



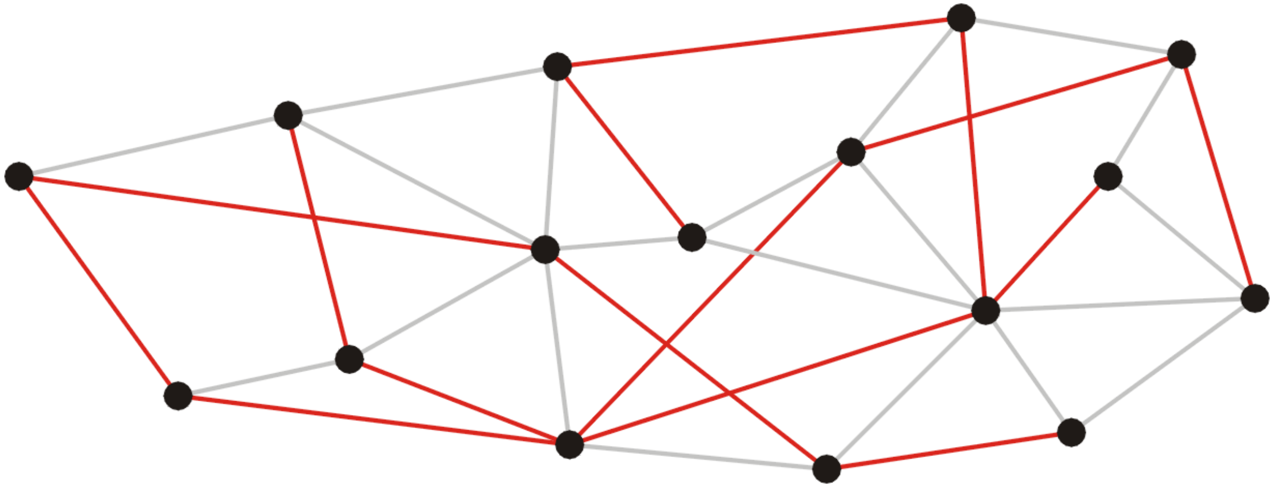
Spanning trees

These 15 edges form a minimum spanning tree



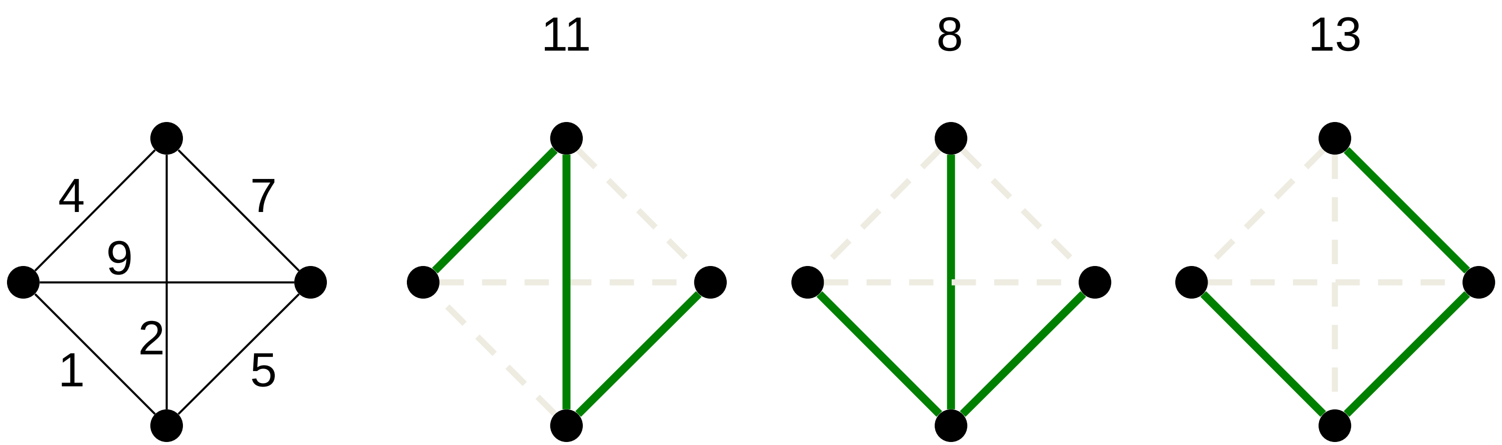
Spanning trees

As do these 15 edges:



Spanning trees on weighted graphs

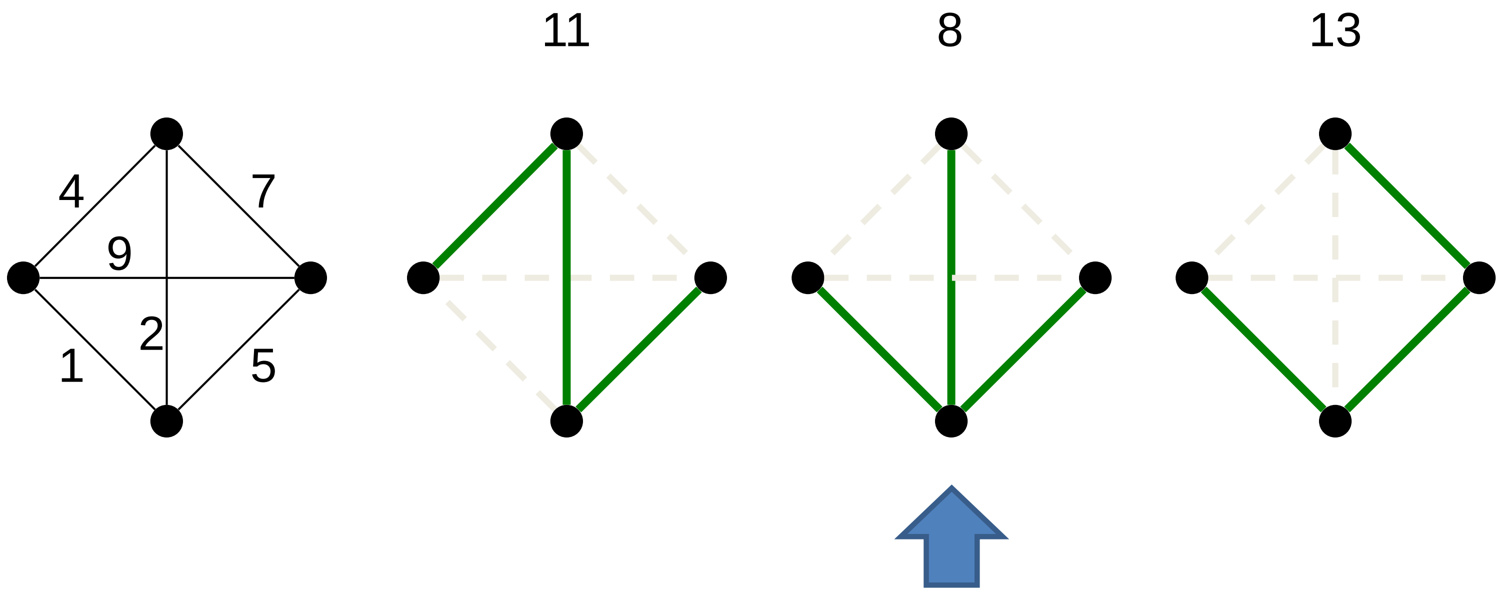
The weight of a spanning tree is the sum of the weights on all the edges which comprise the spanning tree



Minimum Spanning Trees

Which spanning tree minimizes the weight?

– Such a tree is termed a *minimum spanning tree*

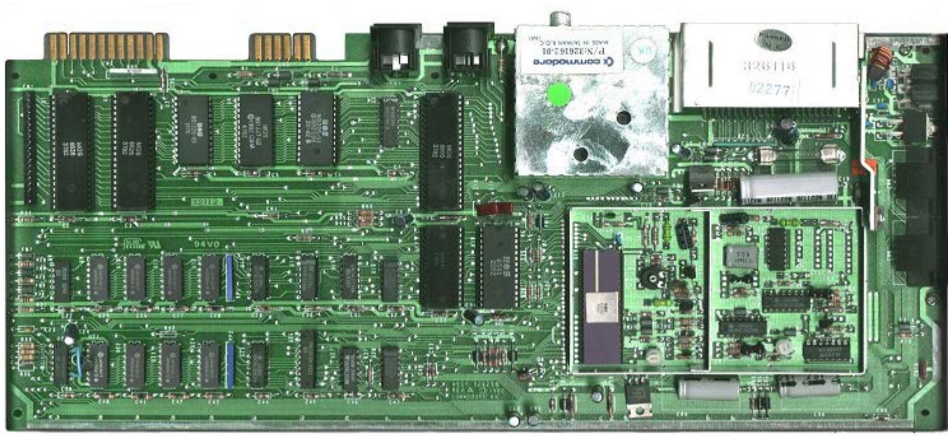


Application

Consider supplying power to

- All circuit elements on a board
- A number of loads within a building

A minimum spanning tree will give the lowest-cost solution



www.kpmb.com

www.commedore.ca

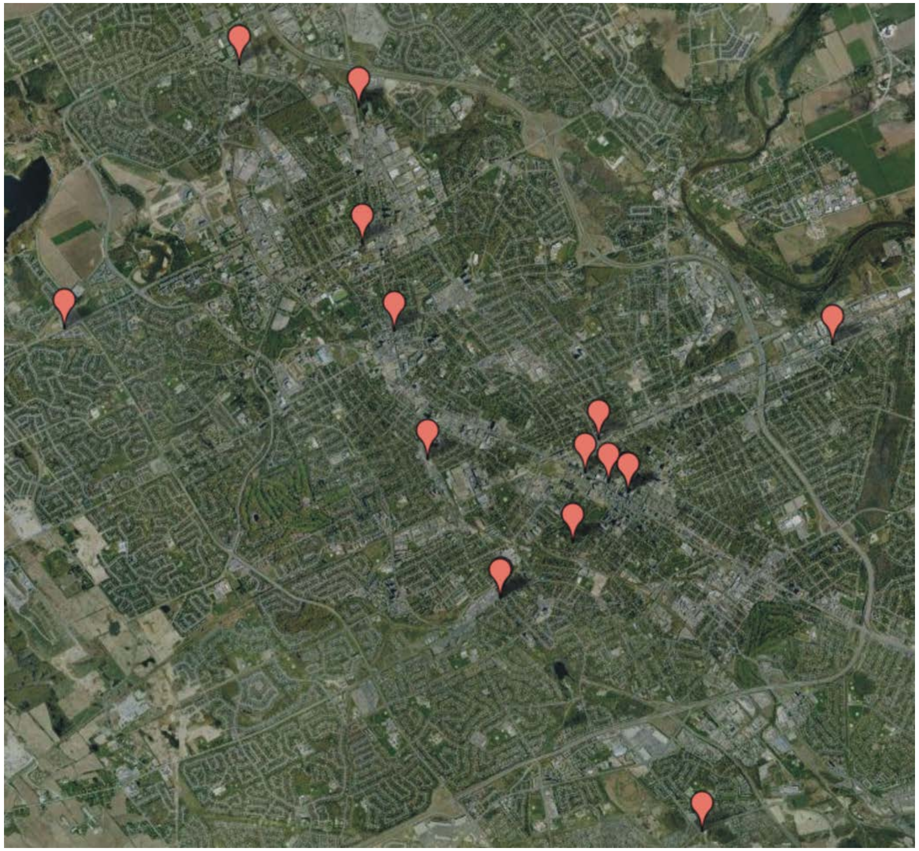
Application

The first application of a minimum spanning tree algorithm was by the Czech mathematician Otakar Borůvka who designed electricity grid in Moravia in 1926

Application

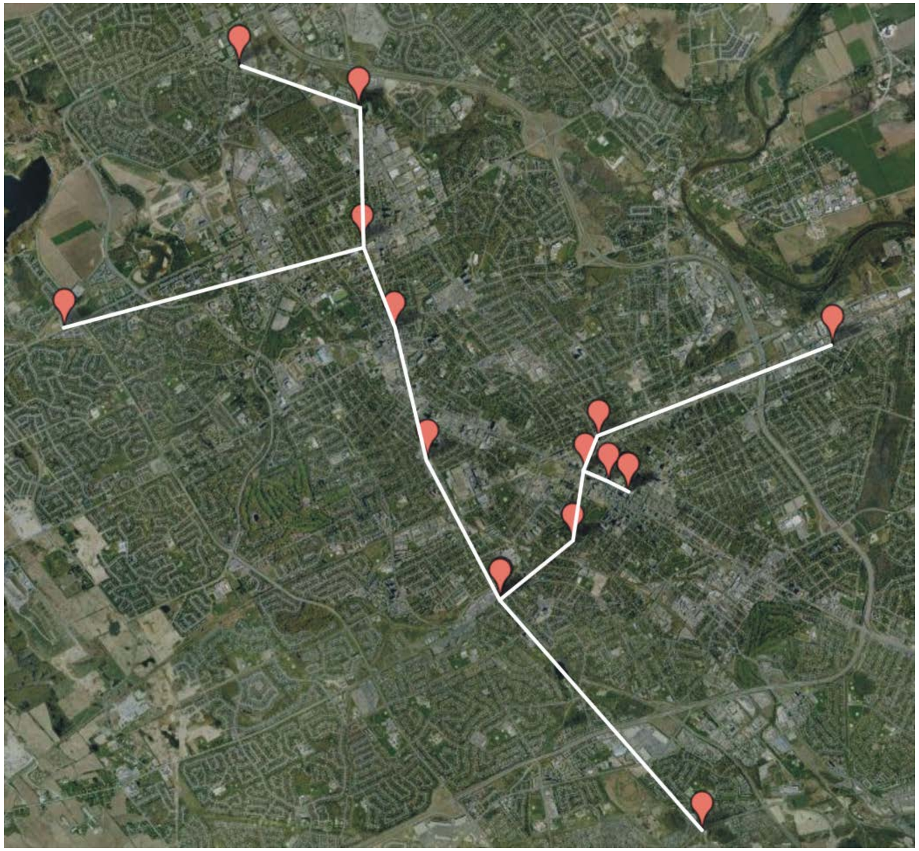
Consider attempting to find the best means of connecting a number of houses

- Minimize the length of transmission lines



Application

A minimum spanning tree will provide the optimal solution



Minimum Spanning Trees

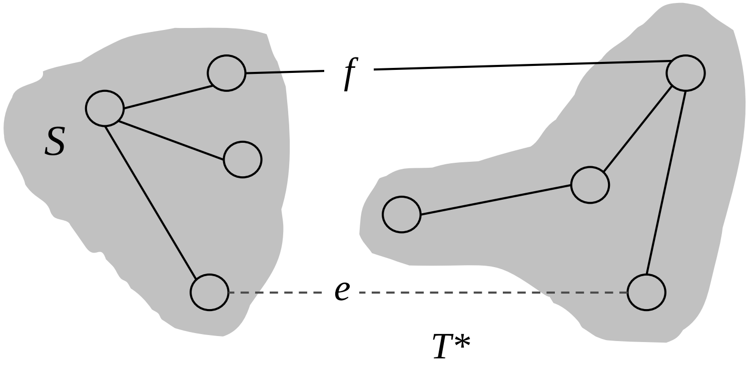
Simplifying assumption:

- All edge weights are distinct

This guarantees that given a graph, there is a unique minimum spanning tree.

Cut property

- Let S be any subset of nodes, and let e be the least weight edge with exactly one endpoint in S. Then the MST T^* contains e.
- Proof
  - Suppose e does not belong to T^*.
  - Adding e to T^* creates a cycle C in T^*.
  - e is in a cycle C with exactly one endpoint in S $\Rightarrow$ there exists another edge f in C with exactly one endpoint in S.
  - $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
  - Since $w_e < w_f$, the weight of T' is smaller than that of T^*.
  - This is a contradiction



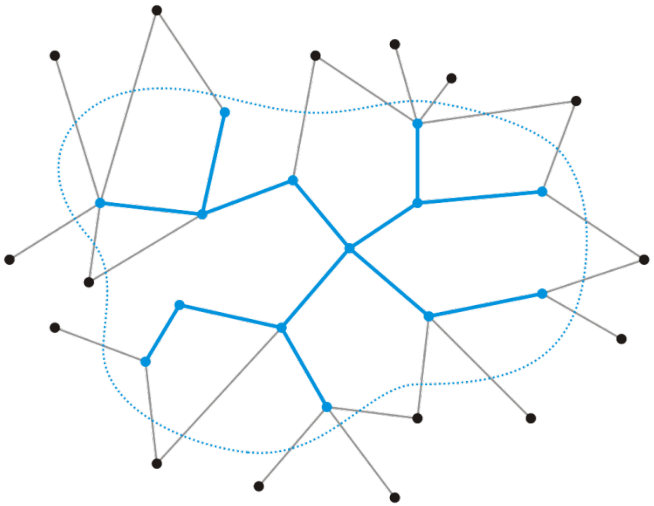
Outline

- Definition and applications
- Prim’s algorithm
- Kruskal’s algorithm

Strategy

Strategy:

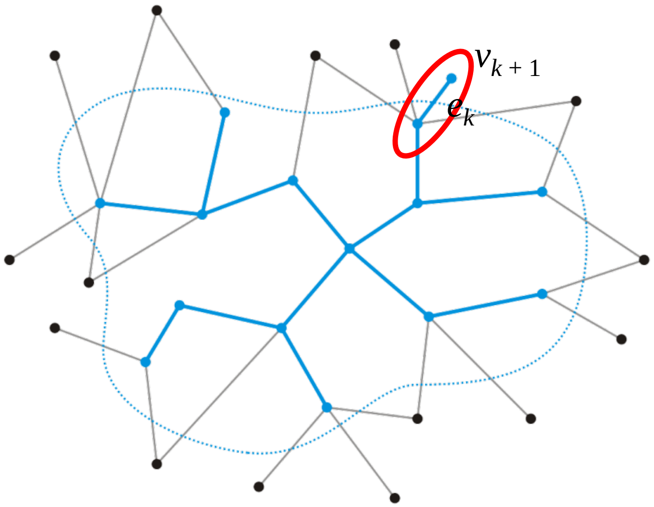
- Suppose we have a known minimum spanning tree on $k < n$ vertices
- How could we extend this minimum spanning tree?



Strategy

Add the edge e_k with least weight that connects this minimum spanning tree to a new vertex v_{k+1}

- This does create a minimum spanning tree on the $k + 1$ nodes—there is no other edge that extends the tree with less weight
- Does the new edge belong to the minimum spanning tree on all n vertices?
  - Yes! The cut property.



Prim’s Algorithm

Prim’s algorithm for finding the minimum spanning tree states:

- Start with an arbitrary vertex to form a minimum spanning tree on one vertex
- At each step, add the edge with least weight that connects the current minimum spanning tree to a new vertex
- Continue until we have $n - 1$ edges and n vertices

Prim's Algorithm

Associate with each vertex three items of data:

- A Boolean flag indicating if the vertex has been visited,
- The minimum distance (weight of a connecting edge) to the partially constructed tree, and
- A pointer to the vertex which will form the parent node in the resulting tree

Implementation:

- Add three member variables to the vertex class
- Or track three tables

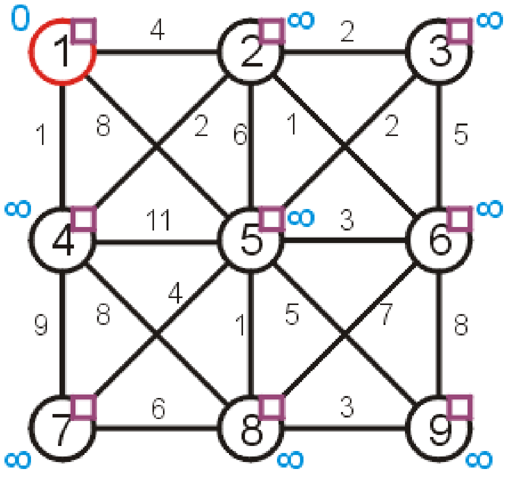
Prim's Algorithm

Initialization:

- Select a root node and set its distance as 0
- Set the distance to all other vertices as ∞
- Set all vertices to being unvisited
- Set the parent pointer of all vertices to 0

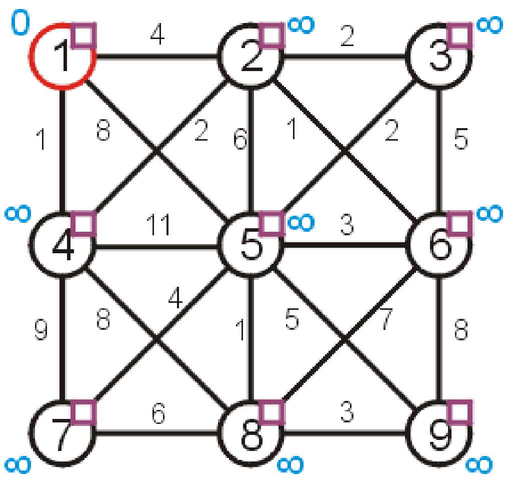
Prim’s Algorithm

Let us find the minimum spanning tree for the following undirected weighted graph



Prim's Algorithm

First we set up the appropriate table and initialize it



		Distance	Parent
1	F	0	0
2	F	∞	0
3	F	∞	0
4	F	∞	0
5	F	∞	0
6	F	∞	0
7	F	∞	0
8	F	∞	0
9	F	∞	0

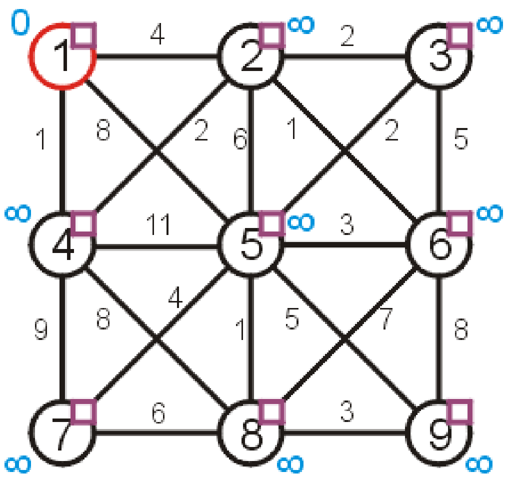
Prim's Algorithm

Iterate while there exists an unvisited vertex with distance < ∞

- Select the unvisited vertex *v* with minimum distance
- Mark *v* as having been visited
- For each unvisited adjacent vertex of *v*, if the weight of the connecting edge is less than the current distance to that vertex:
  - Update the distance to the weight of the edge
  - Set *v* as the parent of the vertex

Prim's Algorithm

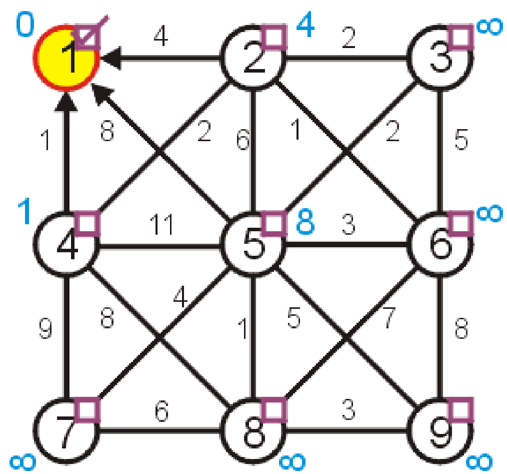
First we set up the appropriate table and initialize it



		Distance	Parent
1	F	0	0
2	F	∞	0
3	F	∞	0
4	F	∞	0
5	F	∞	0
6	F	∞	0
7	F	∞	0
8	F	∞	0
9	F	∞	0

Prim's Algorithm

Visiting vertex 1, we update vertices 2, 4, and 5

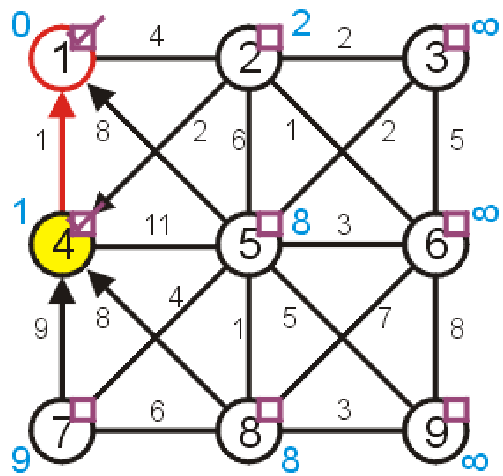


		Distance	Parent
1	T	0	0
2	F	4	1
3	F	∞	0
4	F	1	1
5	F	8	1
6	F	∞	0
7	F	∞	0
8	F	∞	0
9	F	∞	0

Prim’s Algorithm

The next unvisited vertex with minimum distance is vertex 4

- Update vertices 2, 7, 8
- Don’t update vertex 5

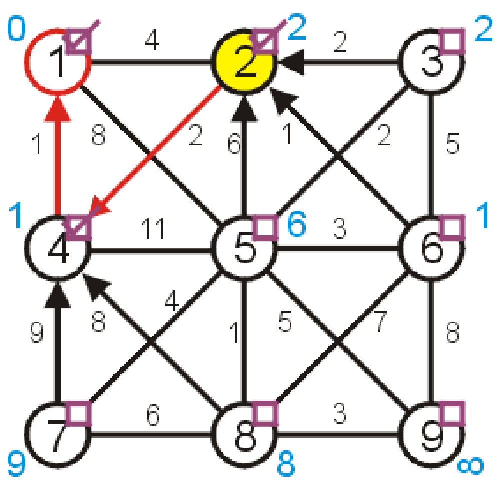


		Distance	Parent
1	T	0	0
2	F	2	4
3	F	∞	0
4	T	1	1
5	F	8	1
6	F	∞	0
7	F	9	4
8	F	8	4
9	F	∞	0

Prim’s Algorithm

Next visit vertex 2

– Update 3, 5, and 6

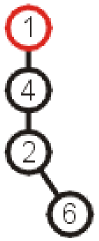


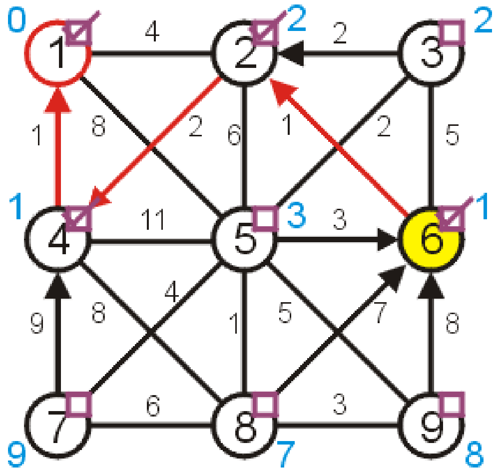
		Distance	Parent
1	T	0	0
2	T	2	4
3	F	2	2
4	T	1	1
5	F	6	2
6	F	1	2
7	F	9	4
8	F	8	4
9	F	∞	0

Prim's Algorithm

Next, we visit vertex 6:

– update vertices 5, 8, and 9

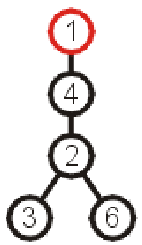


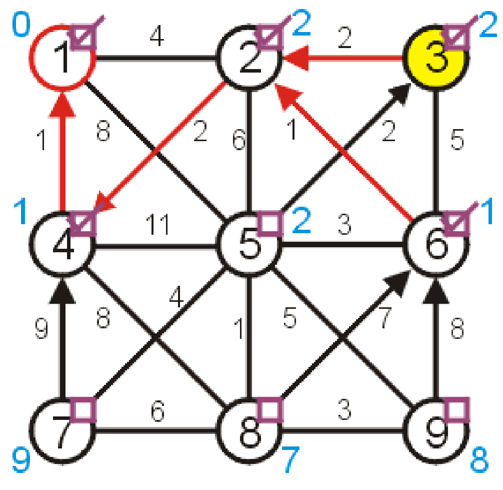


		Distance	Parent
1	T	0	0
2	T	2	4
3	F	2	2
4	T	1	1
5	F	3	6
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

Prim’s Algorithm

Next, we visit vertex 3 and update 5

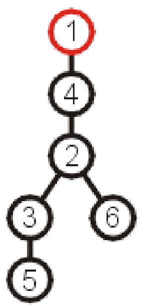


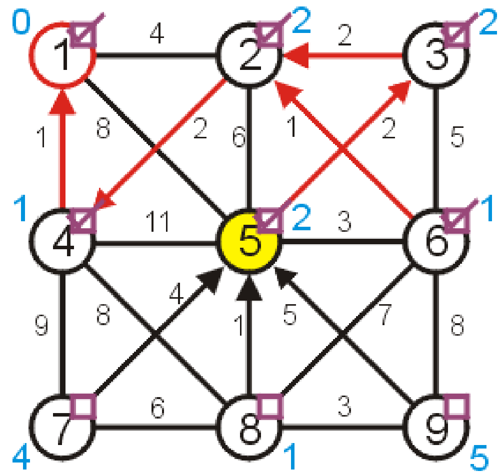


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	F	2	3
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

Prim's Algorithm

Visiting vertex 5, we update 7, 8, 9

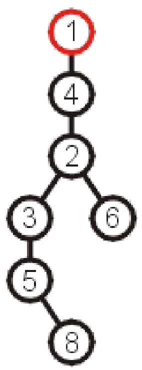


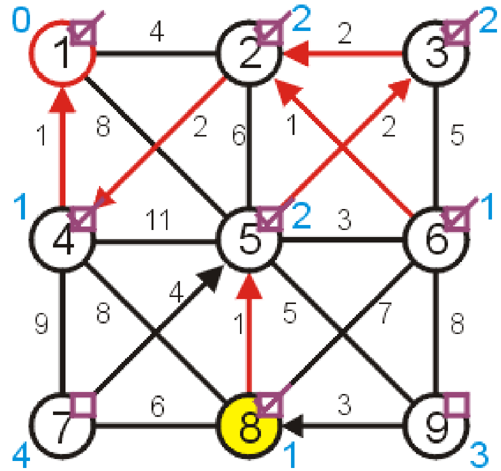


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	F	1	5
9	F	5	5

Prim’s Algorithm

Visiting vertex 8, we only update vertex 9

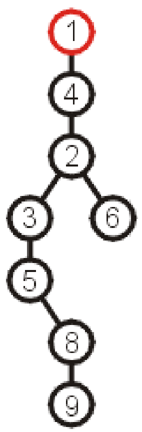


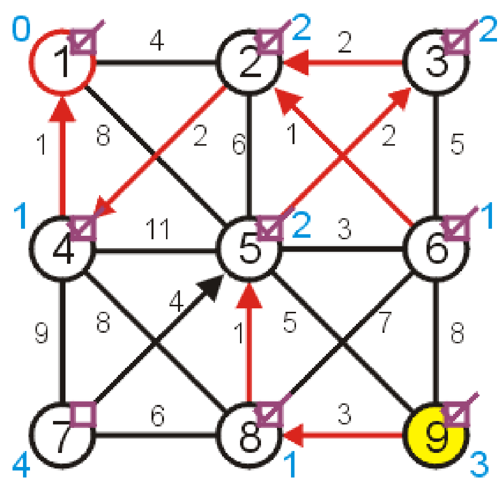


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	T	1	5
9	F	3	8

Prim’s Algorithm

There are no other vertices to update while visiting vertex 9

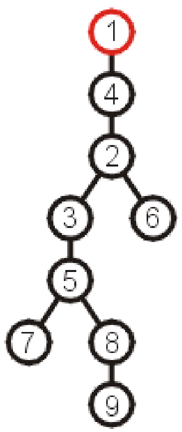


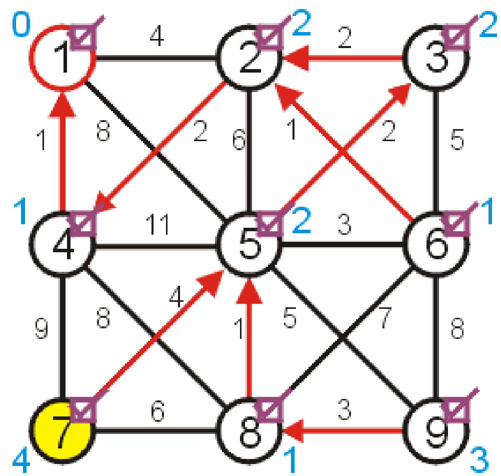


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	T	1	5
9	T	3	8

Prim’s Algorithm

And neither are there any vertices to update when visiting vertex 7





		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	T	4	5
8	T	1	5
9	T	3	8

Prim’s Algorithm

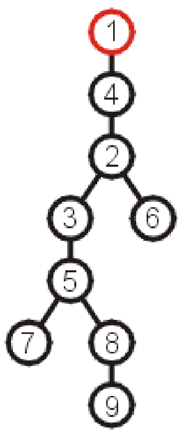
At this point, there are no more unvisited vertices, and therefore we are done

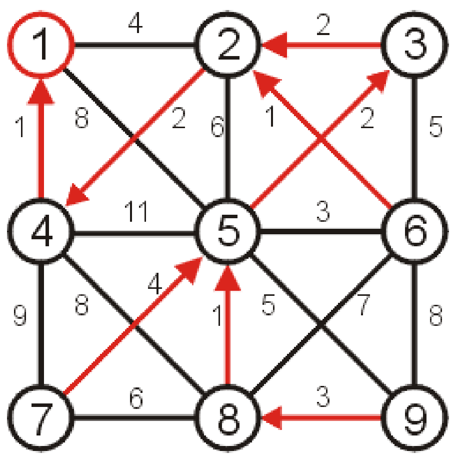
If at any point, all remaining vertices had a distance of ∞, this would indicate that the graph is not connected

- in this case, the minimum spanning tree would only span one connected sub-graph

Prim's Algorithm

Using the parent pointers, we can now construct the minimum spanning tree





		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	T	4	5
8	T	1	5
9	T	3	8

Implementation and analysis

The initialization requires $\Theta(|V|)$ memory and run time

We iterate $|V| - 1$ times, each time finding the *closest* vertex

- Iterating through the table requires is $\Theta(|V|)$ time
- Each time we find a vertex, we must check all of its neighbors

With an adjacency list, the run time is $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$ as $|E| = \mathrm{O}(|V|^2)$

Can we do better?

- At each iteration, we need to find the shortest edge
- How about a priority queue?
    - Assume we are using a binary heap

Implementation and analysis

The initialization still requires $\Theta(|V|)$ memory and run time

– The priority queue will also requires $O(|V|)$ memory

We iterate $|V| - 1$ times, each time finding the *closest* vertex

– The size of the priority queue is $O(|V|)$
– Pop the closest vertex from the priority queue is $O(\ln(|V|))$
– For each of its neighbors, we may update the distance, which is $O(\ln(|V|))$

With an adjacency list, the total run time is $O(|V| \ln(|V|) + |E| \ln(|V|)) = O(|E| \ln(|V|))$

Implementation and analysis

We could use a different heap structure:

- A Fibonacci heap is a node-based heap
- Pop is still $O(\ln(|V|))$, but inserting and moving a key is $\Theta(1)$
- Thus, the overall run-time is $O(|E| + |V| \ln(|V|))$

Implementation and analysis

Thus, we have two run times when using

- A binary heap: $O(|E| \ln(|V|))$
- A Fibonacci heap: $O(|E| + |V| \ln(|V|))$

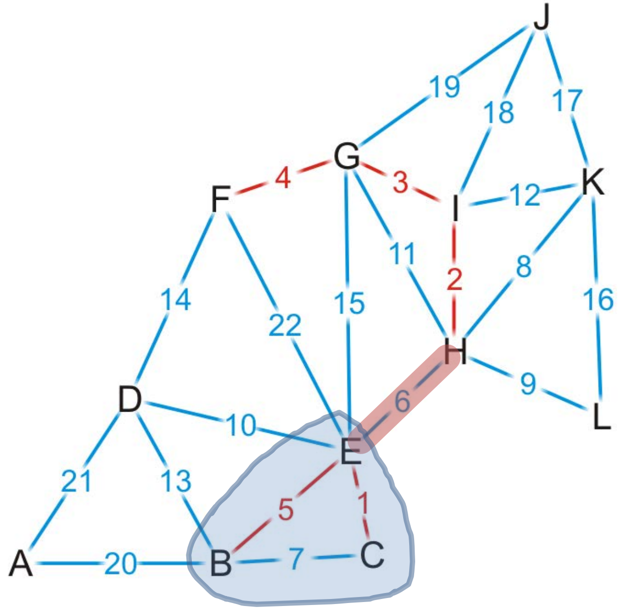
Questions: Which is faster if $|E| = \Theta(|V|)$? How about if $|E| = \Theta(|V|^2)$?

Outline

- Definition and applications
- Prim’s algorithm
- Kruskal’s algorithm

Kruskal's Algorithm

- Sort the edges by weight
- Go through the edges from least weight to greatest weight
  - add the edges to the spanning tree so long as the addition does not create a cycle
  - Does this edge belong to the minimum spanning tree?
    - Yes! The cut property (consider the subtree connected to one end of the edge as the set *S*).



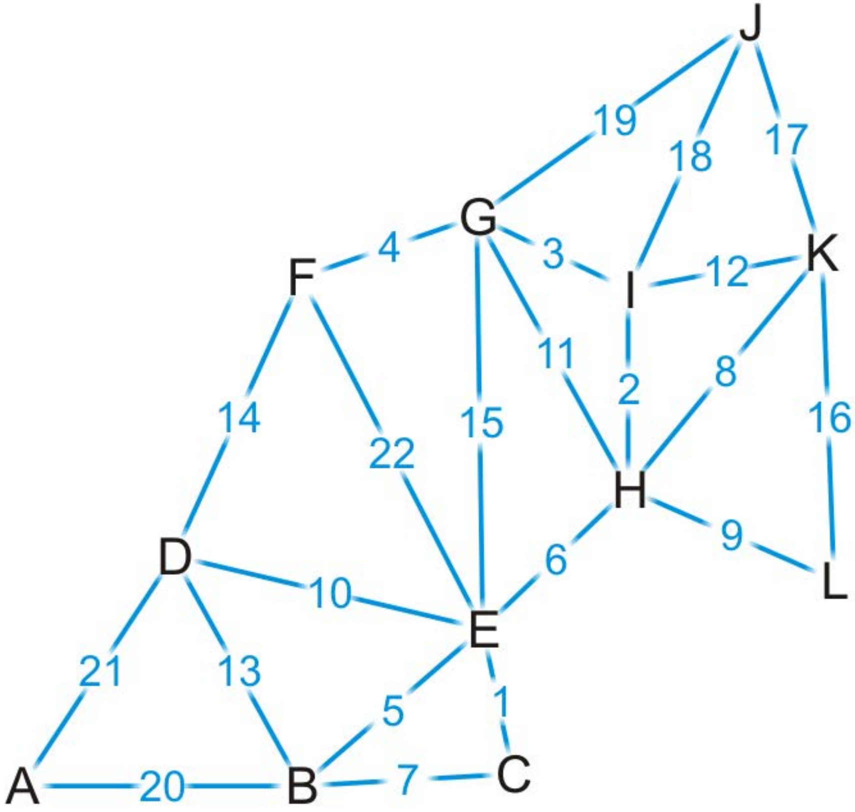
Kruskal's Algorithm

- Sort the edges by weight
- Go through the edges from least weight to greatest weight
  - add the edges to the spanning tree so long as the addition does not create a cycle
  - Does this edge belong to the minimum spanning tree?
    - Yes! The cut property (consider the subtree connected to one end of the edge as the set *S*).

- Repeatedly add more edges until:
  - $|V| - 1$ edges have been added, then we have a minimum spanning tree
  - Otherwise, if we have gone through all the edges, then we have a forest of minimum spanning trees on all connected sub-graphs

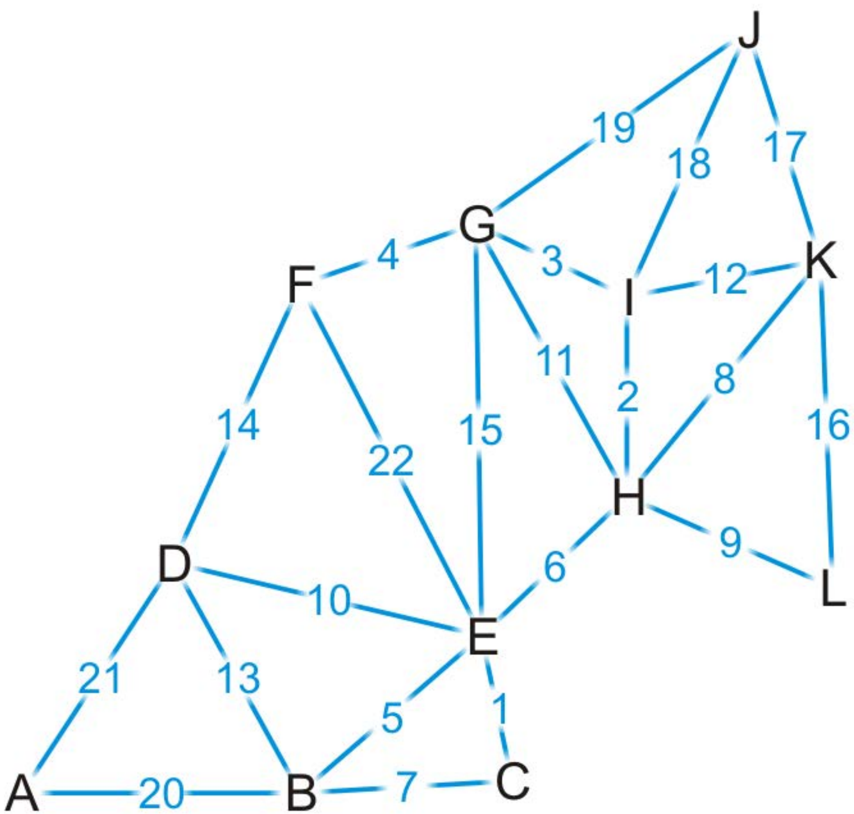
Example

Here is an example graph



Example

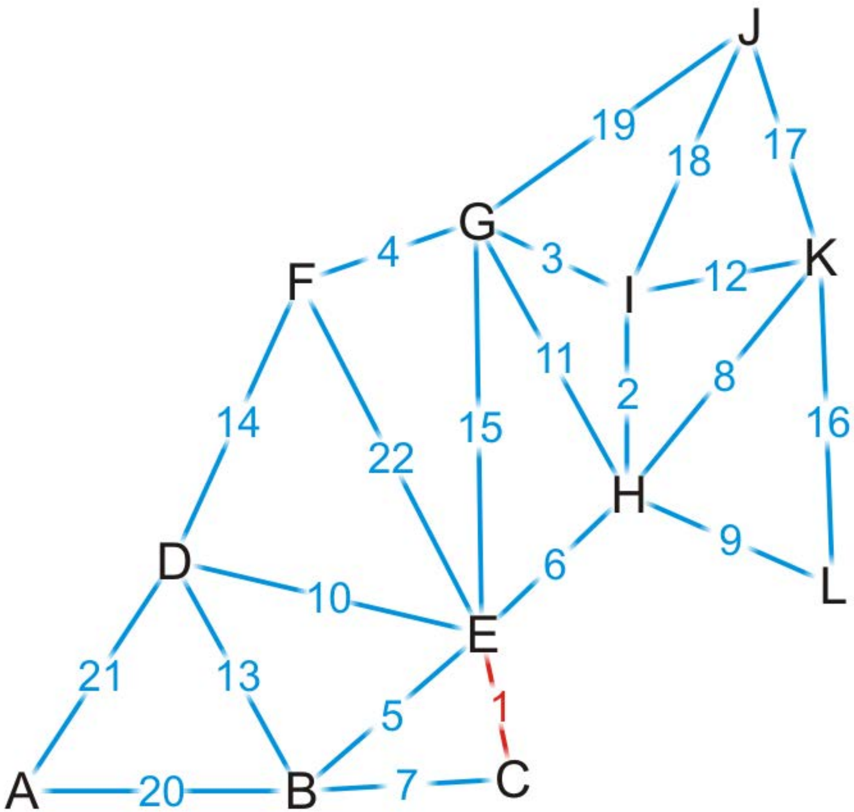
First, we sort the edges based on weight



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

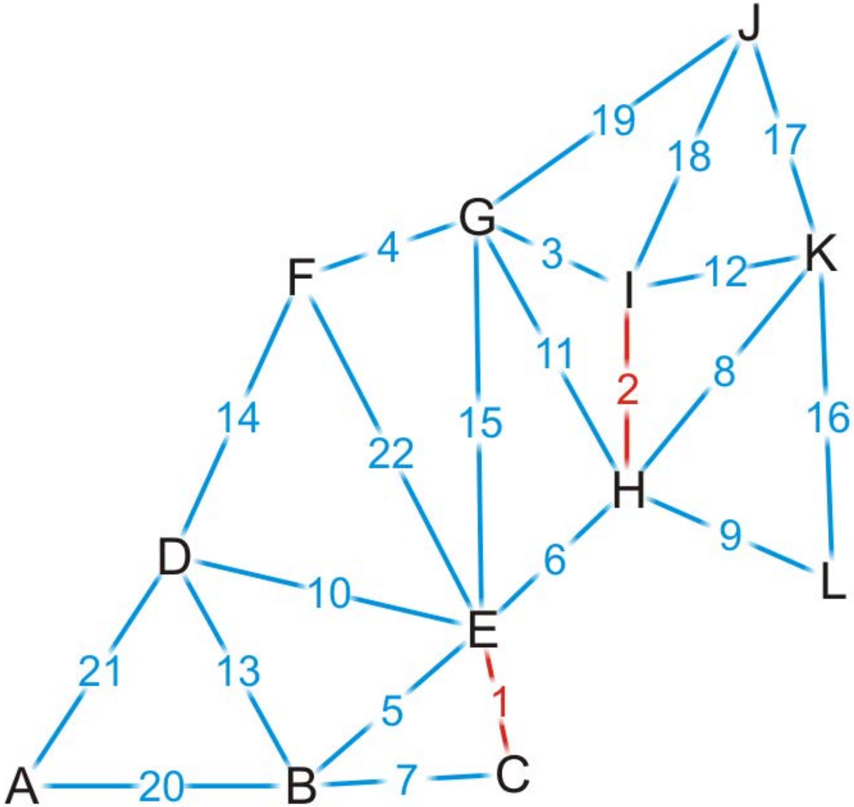
We start by adding edge {C, E}



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

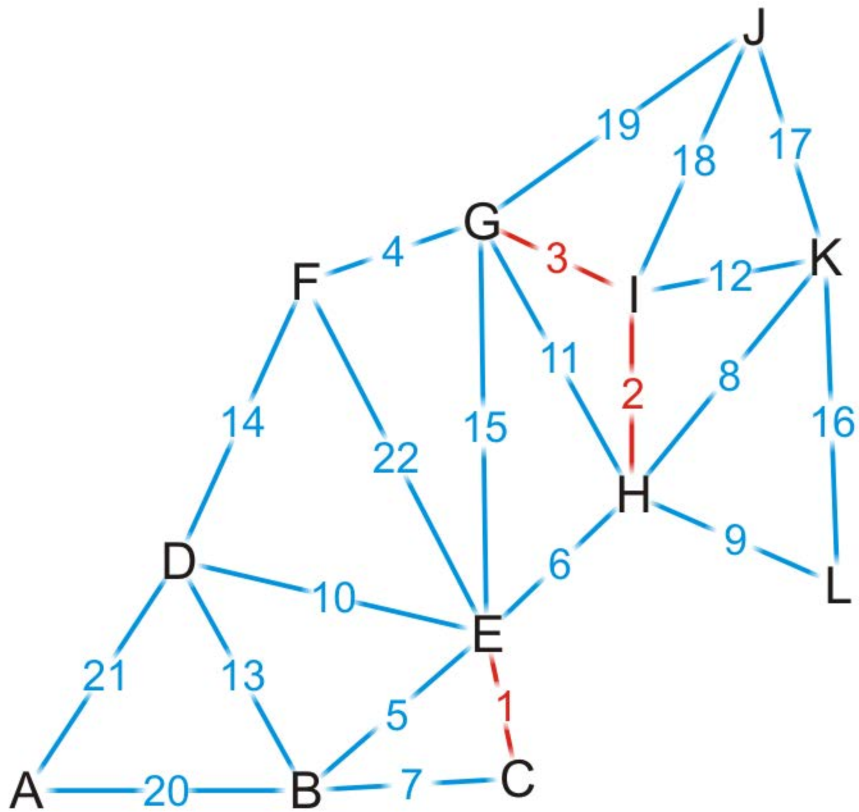
We add edge {H, I}



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

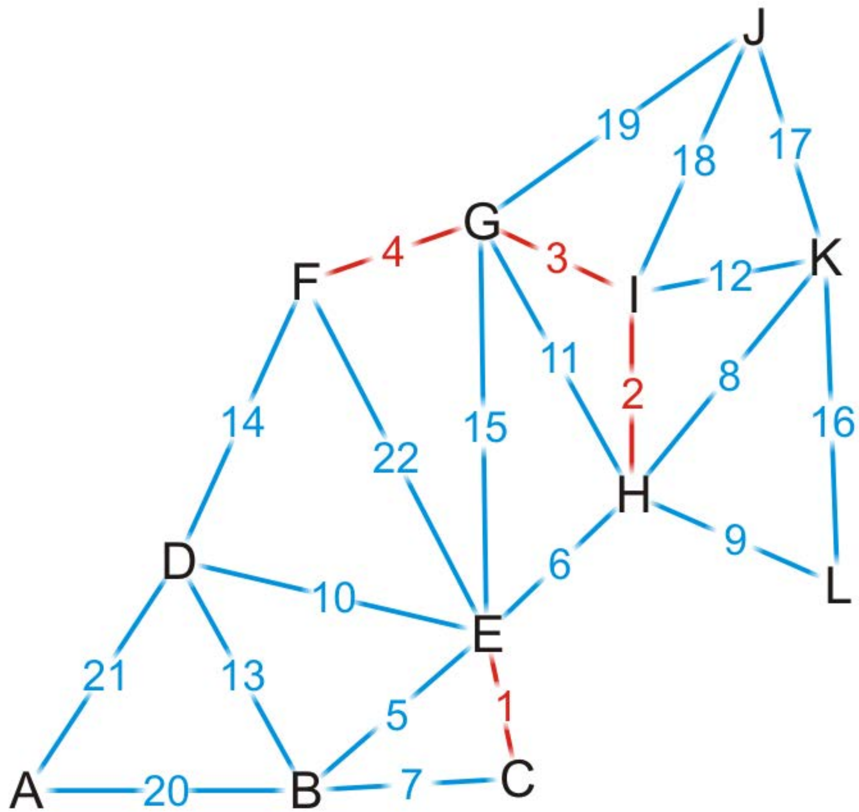
We add edge {G, I}



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

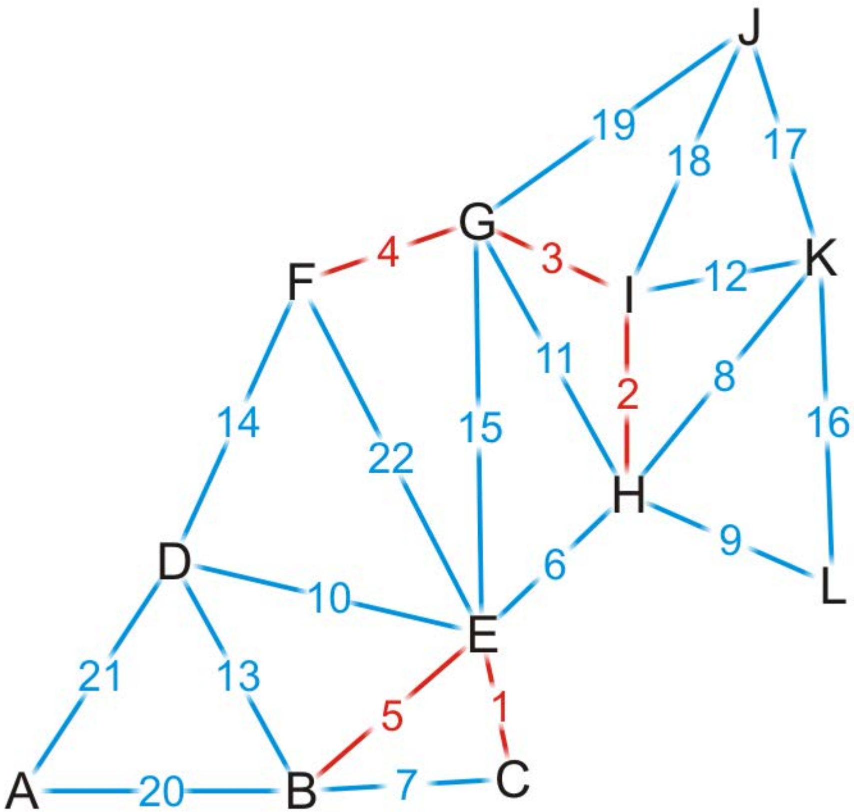
We add edge {F, G}



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

We add edge {B, E}

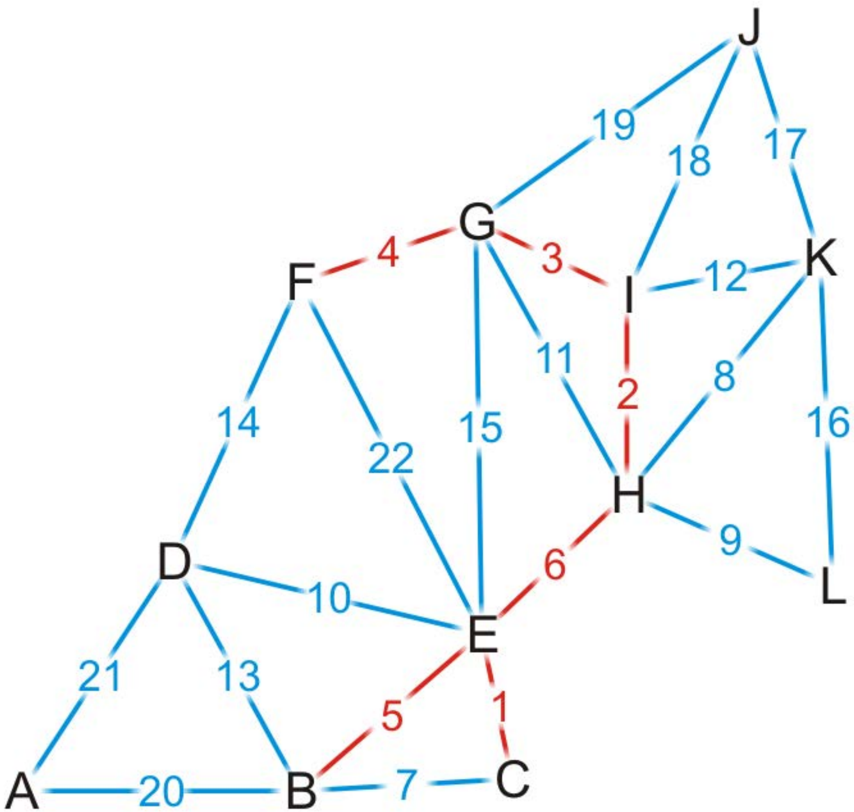


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

We add edge {E, H}

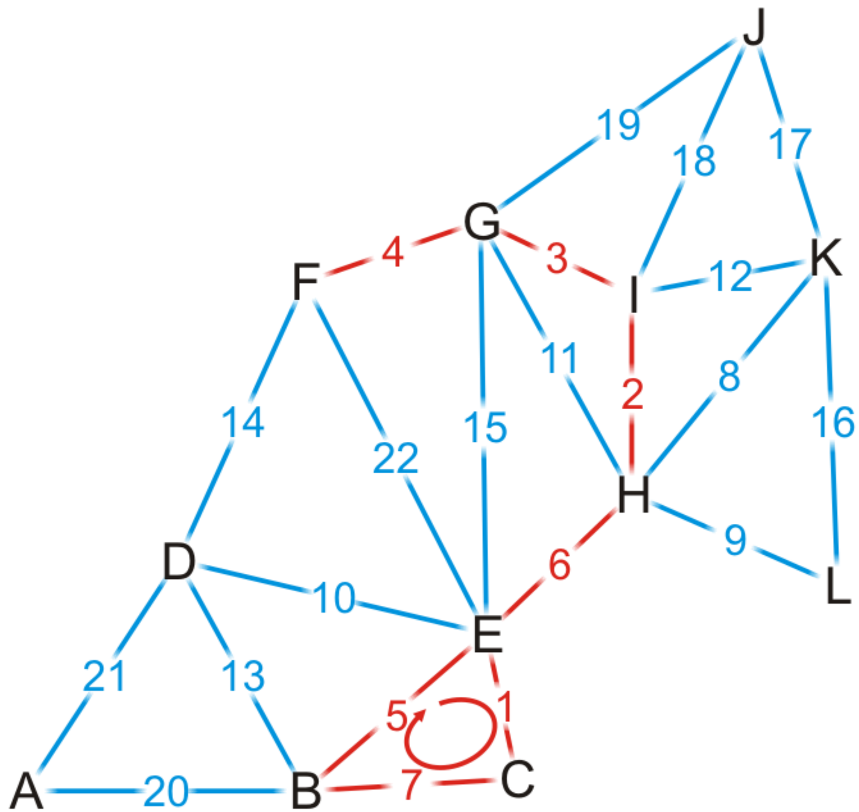
– This coalesces the two spanning sub-trees into one



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

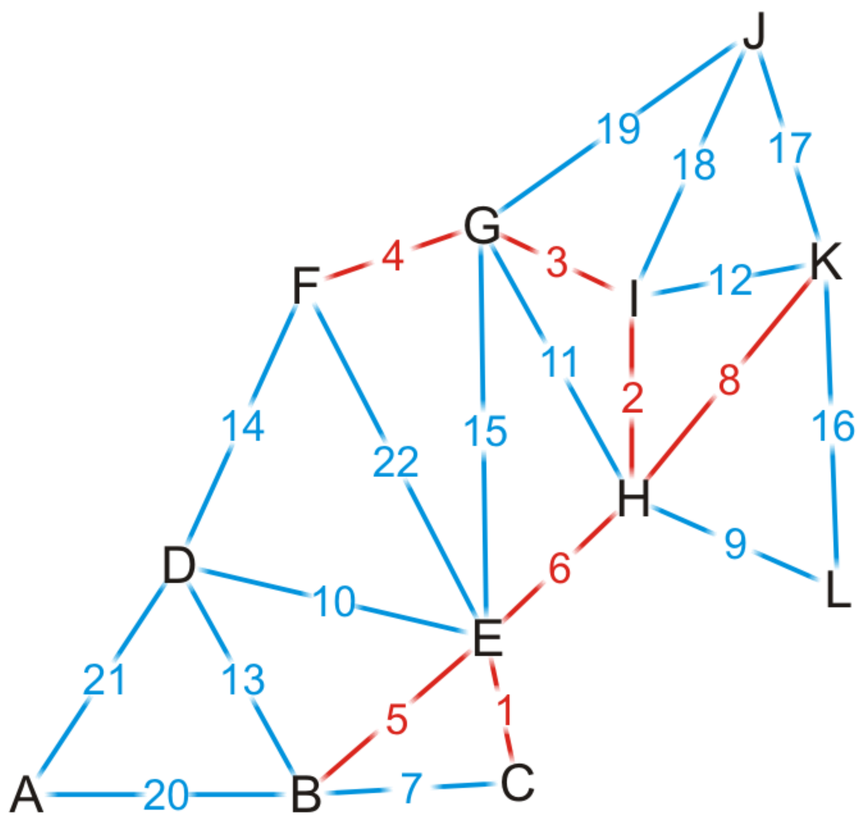
We try adding {B, C}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

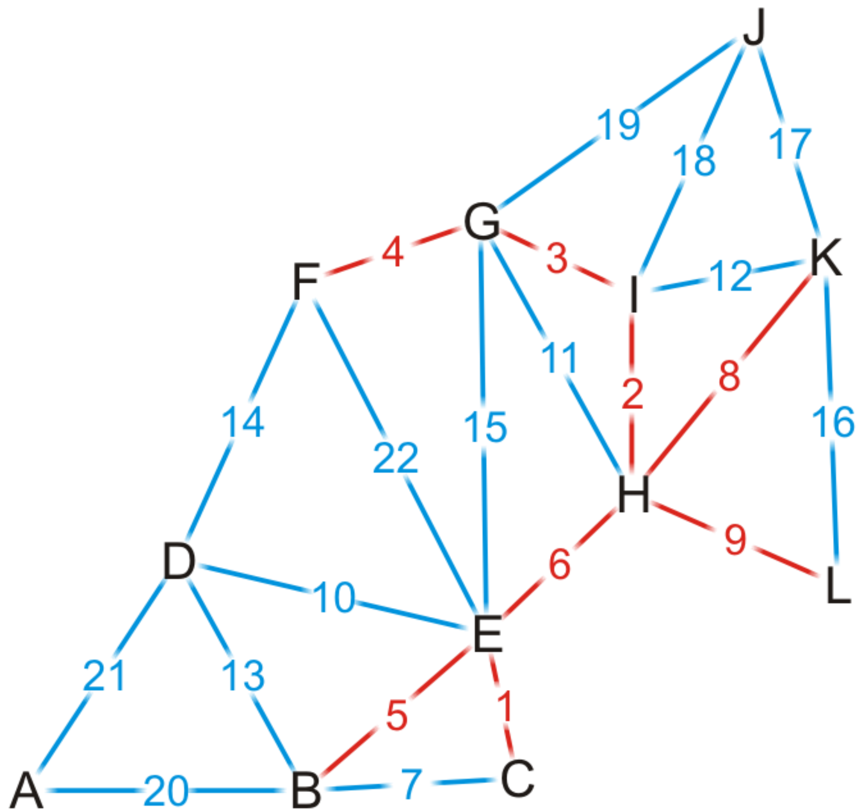
We add edge {H, K}



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

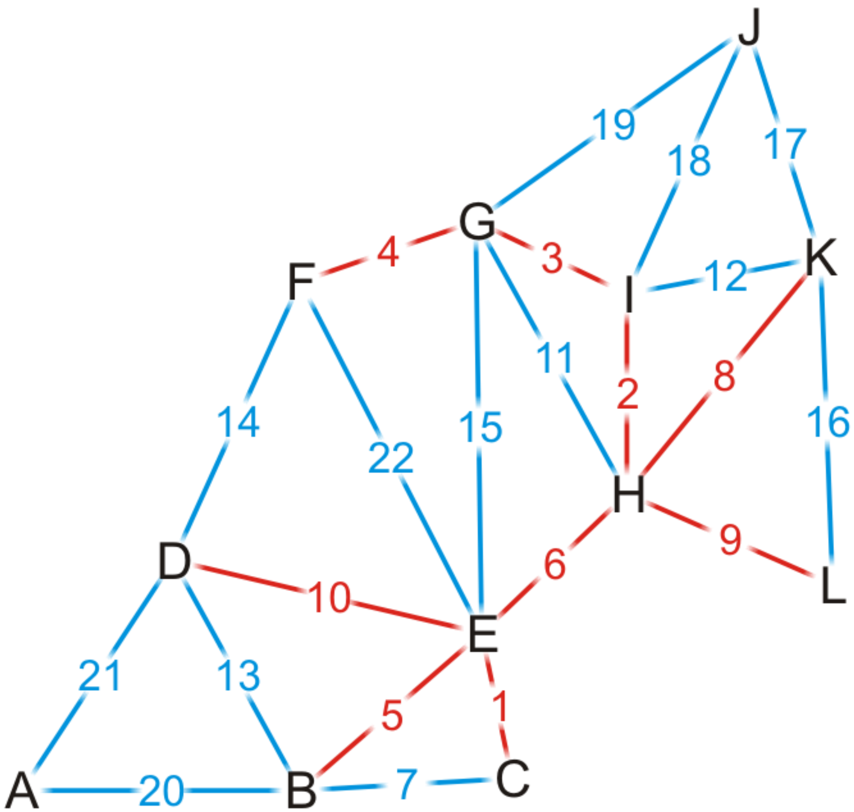
We add edge {H, L}



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

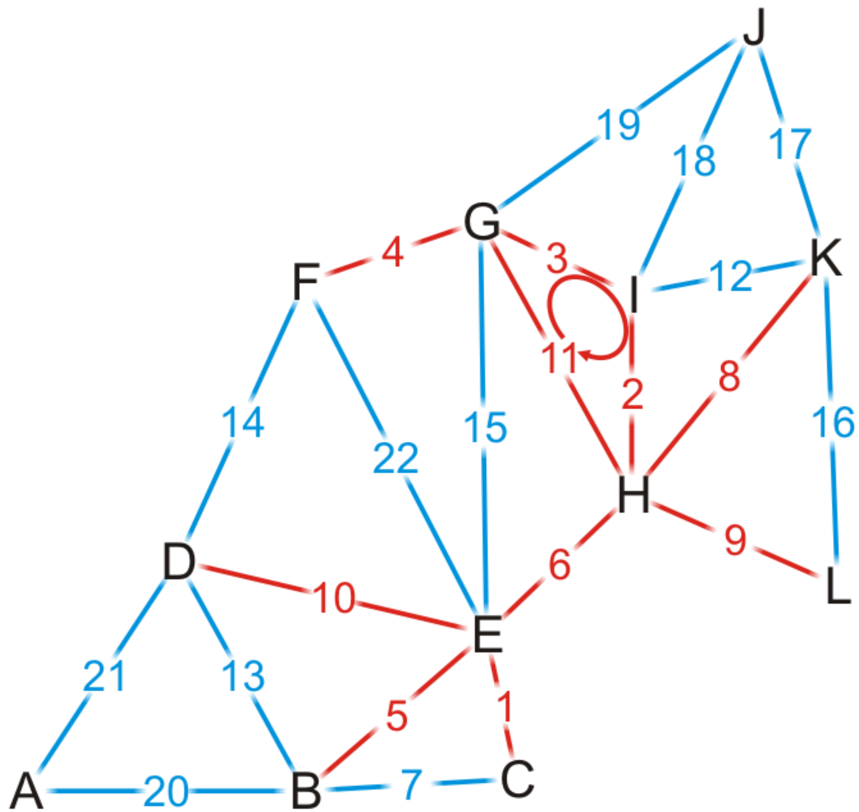
We add edge {D, E}



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

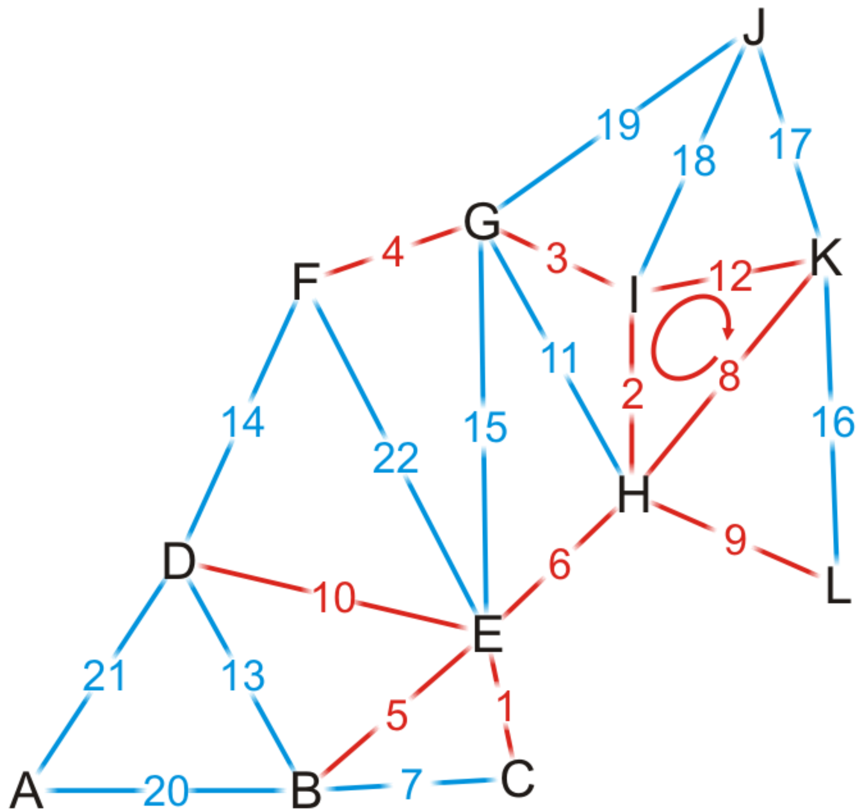
We try adding {G, H}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

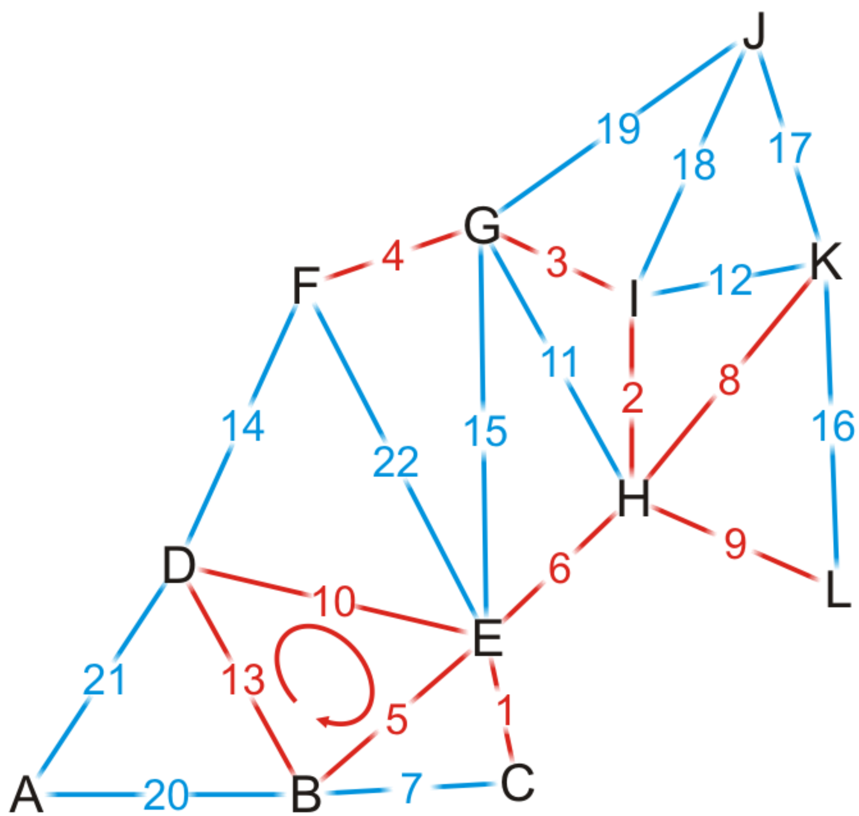
We try adding {I, K}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

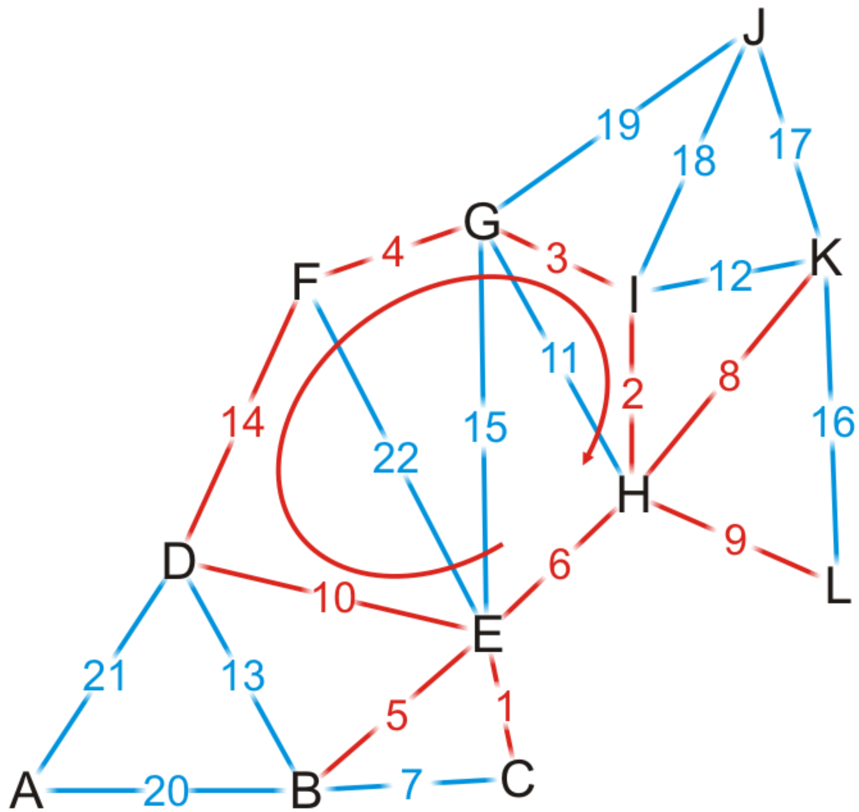
We try adding {B, D}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

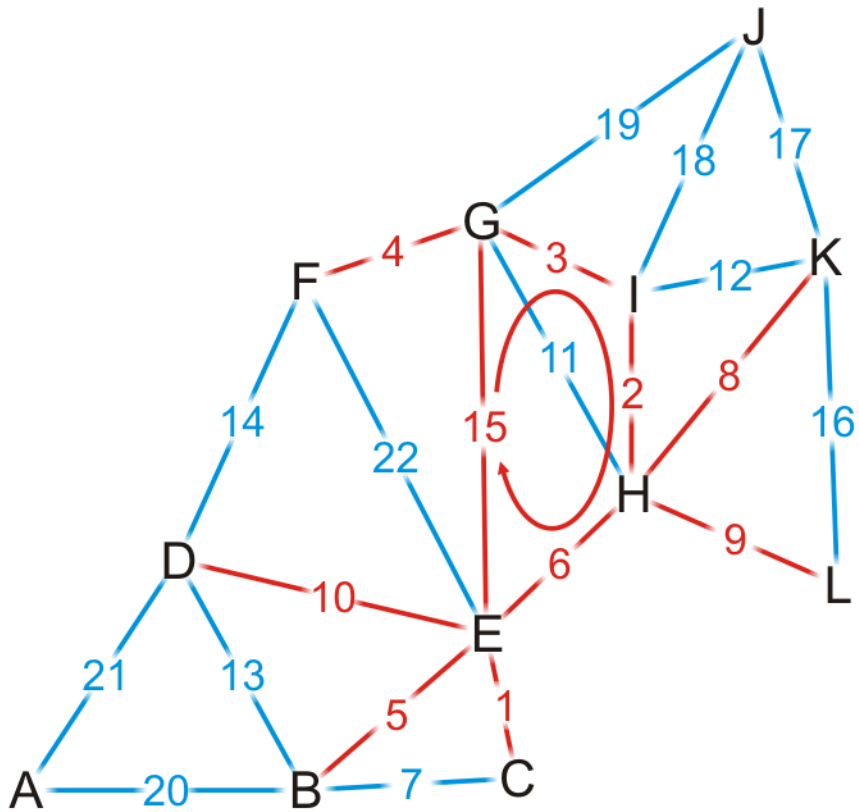
We try adding {D, F}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

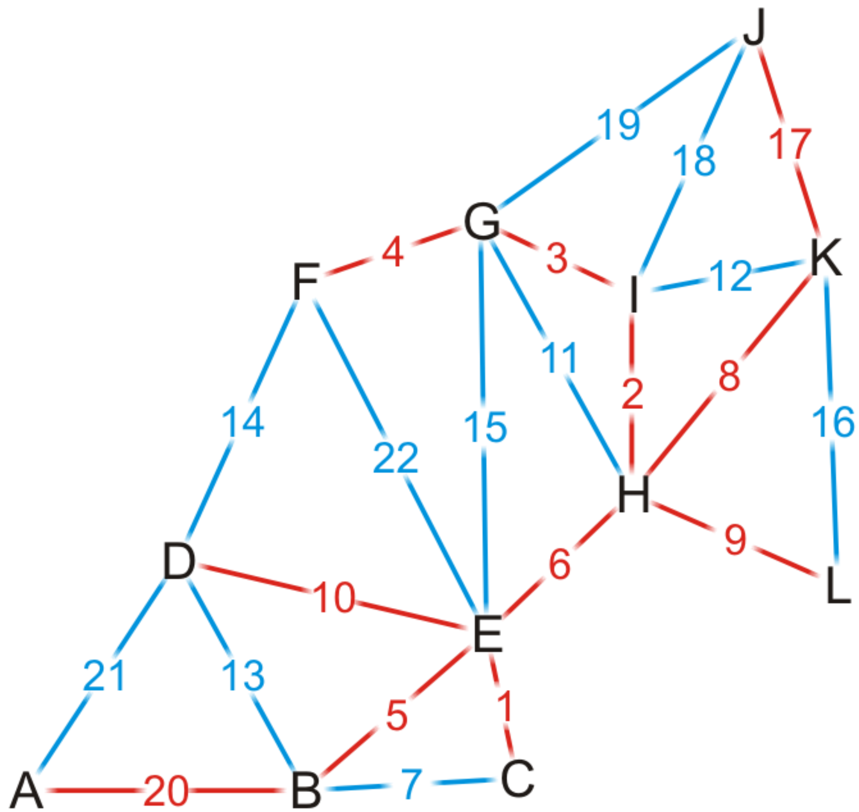
We try adding {E, G}, but it creates a cycle



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

By observation, we can still add edges {J, K} and {A, B}

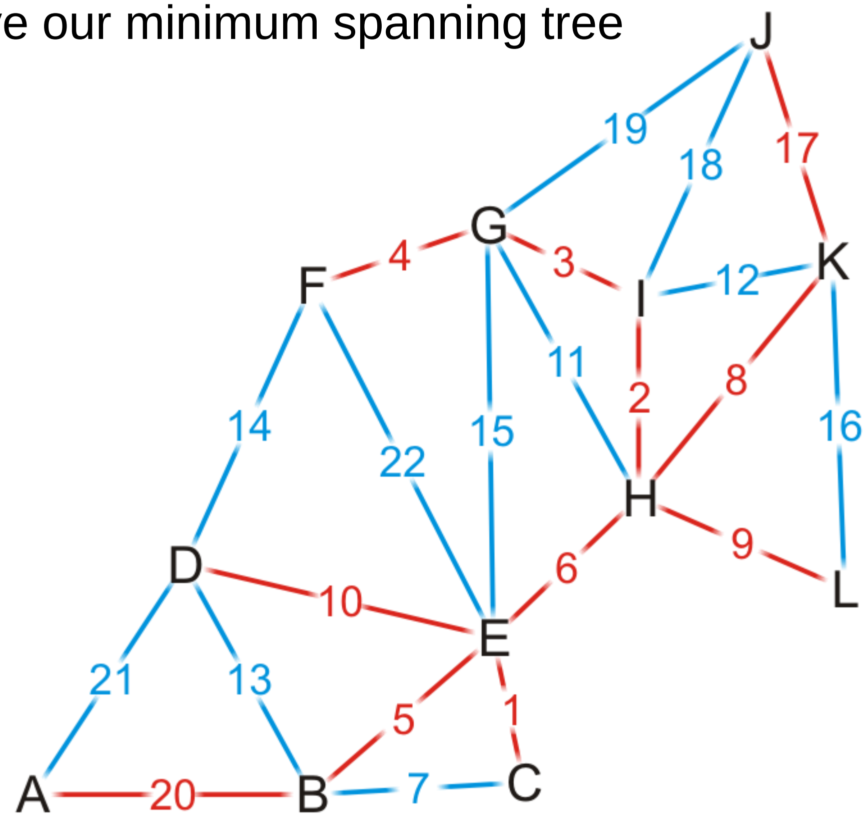


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

Having added {A, B}, we now have 11 edges

- We terminate the loop
- We have our minimum spanning tree



{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Analysis

Implementation

- We would store the edges and their weights in an array
- We would sort the edges using some sorting algorithm: $O(|E| \ln(|E|))$
- For each edge, add it if no cycle is created.
  - How do we determine if a cycle would be created?
  - Check if the two vertices of the edge are already connected by the added edges.

The critical operation is determining if two vertices are connected

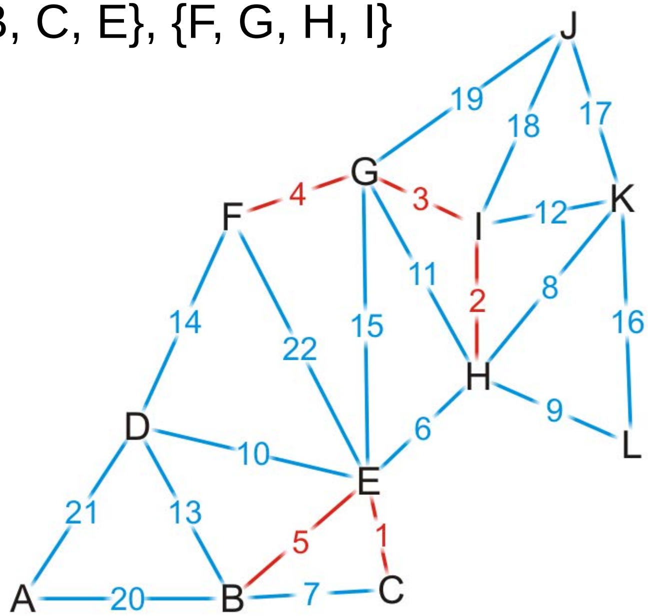
- If we perform a traversal on the added edges, it is $O(|V|)$. Consequently, the total run-time would be $O(|E| \ln(|E|) + |E|\cdot|V|) = O(|E|\cdot|V|)$
- Better solution?

Analysis

Instead, we could use disjoint sets

- Consider edges in the same connected sub-graph as forming a set

{B, C, E}, {F, G, H, I}



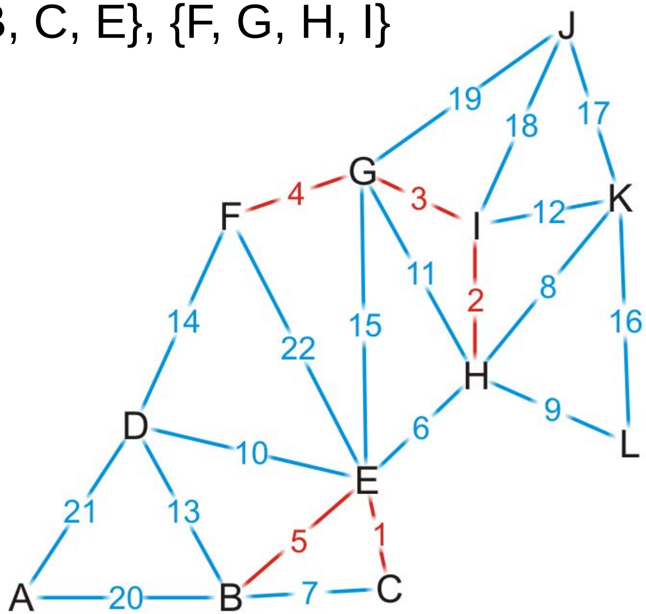
Analysis

Instead, we could use disjoint sets

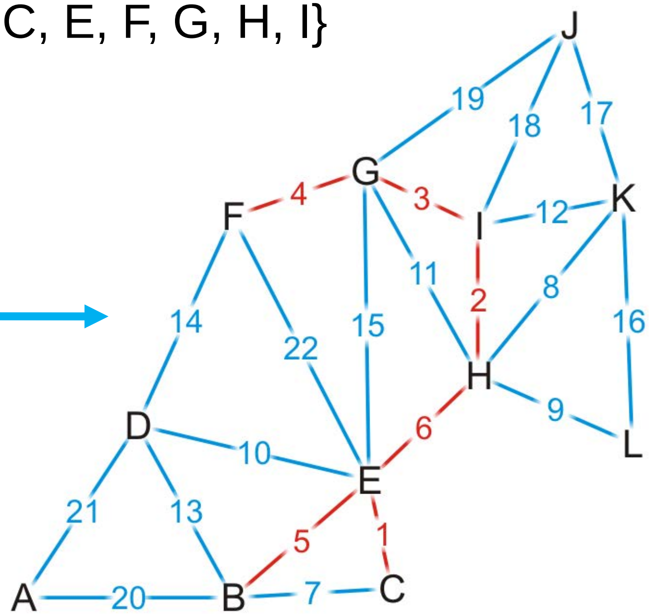
- Consider edges in the same connected sub-graph as forming a set
- If the vertices of the next edge are in different sets, take the union of the two sets

Add edge (E, H)?

{B, C, E}, {F, G, H, I}



{B, C, E, F, G, H, I}



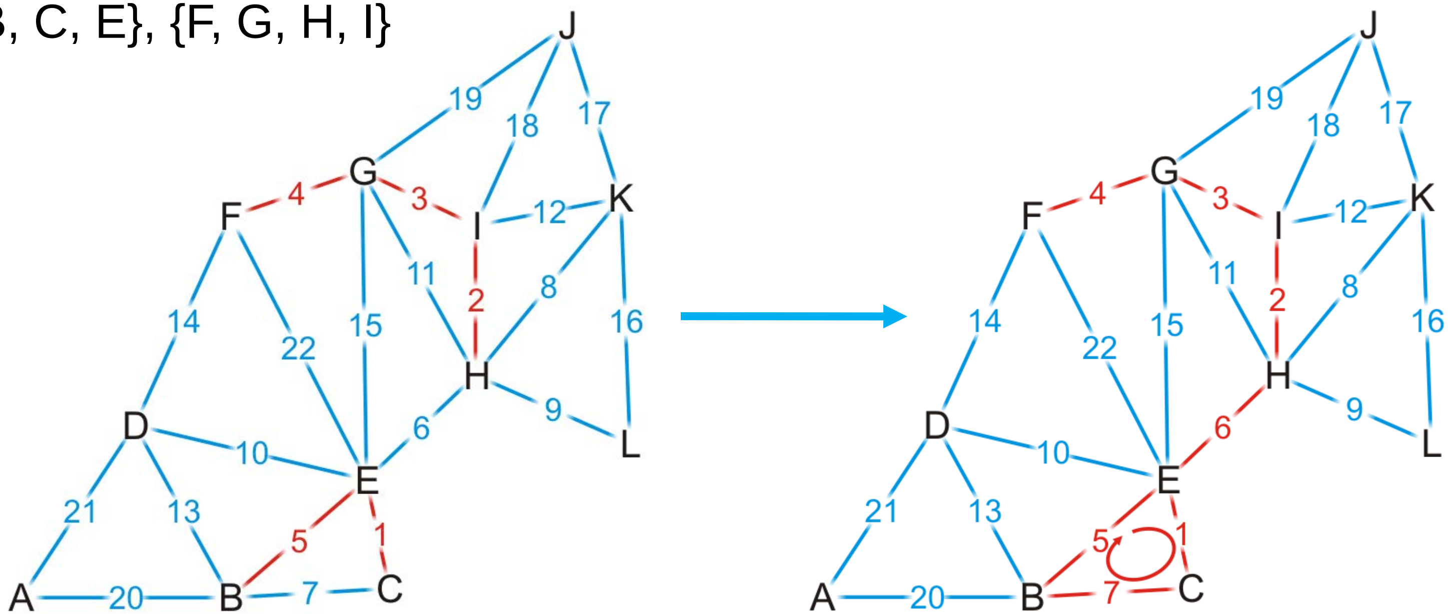
Analysis

Instead, we could use disjoint sets

- Consider edges in the same connected sub-graph as forming a set
- If the vertices of the next edge are in different sets, take the union of the two sets
- Do not add an edge if both vertices are in the same set

Add edge (B, C)?

{B, C, E}, {F, G, H, I}



Analysis

The disjoint set data structure has run-time $\mathbf{O}(\alpha(n))$, which is effectively a constant

Thus, checking and building the minimum spanning tree is now $\mathrm{O}(|E|)$

The dominant time is now the time required to sort the edges, which is $\mathrm{O}(|E| \ln(|E|)) = \mathrm{O}(|E| \ln(|V|))$

– If there is an efficient $\Theta(|E|)$ sorting algorithm, the run-time is then $\Theta(|E|)$

Summary

- Definition and applications
- Prim’s algorithm
  - Start with a trivial minimum spanning tree and grow it by adding edges with least weight
- Kruskal’s algorithm
  - Go through the edges from least weight to greatest weight, adding an edge if it does not create a cycle