Algorithm

Minimum Spanning Trees

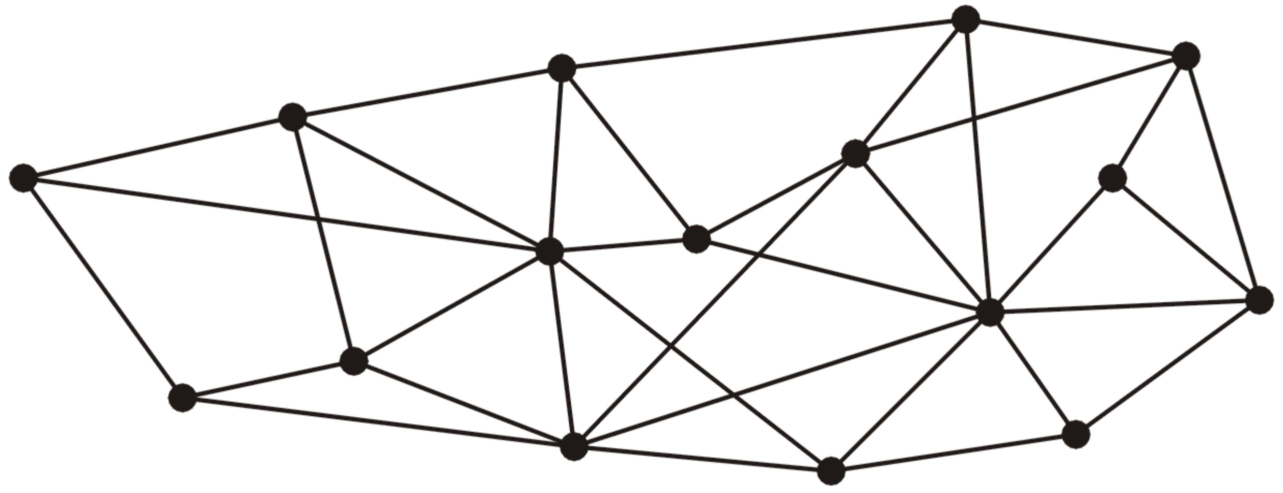
Instructor: Jae-Pil Heo (허재필)

Spanning Trees

- Given a connected graph with $|V| = n$ vertices, a spanning tree is defined a collection of $n - 1$ edges which connect all n vertices
  - The n vertices and $n - 1$ edges define a connected sub-graph
- A spanning tree is not necessarily unique

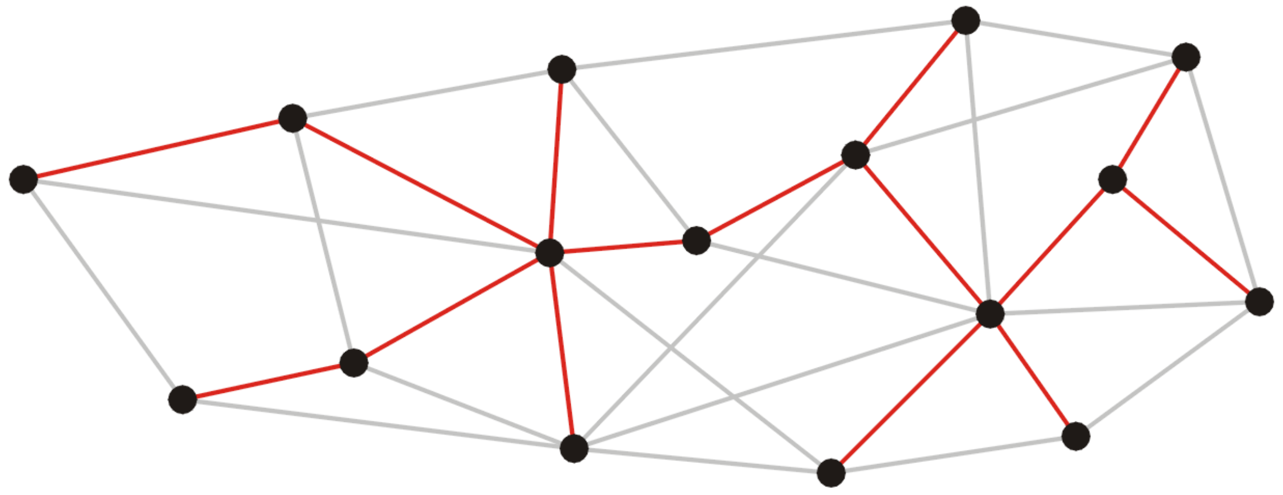
Spanning Trees

- This graph has 16 vertices and 35 edges



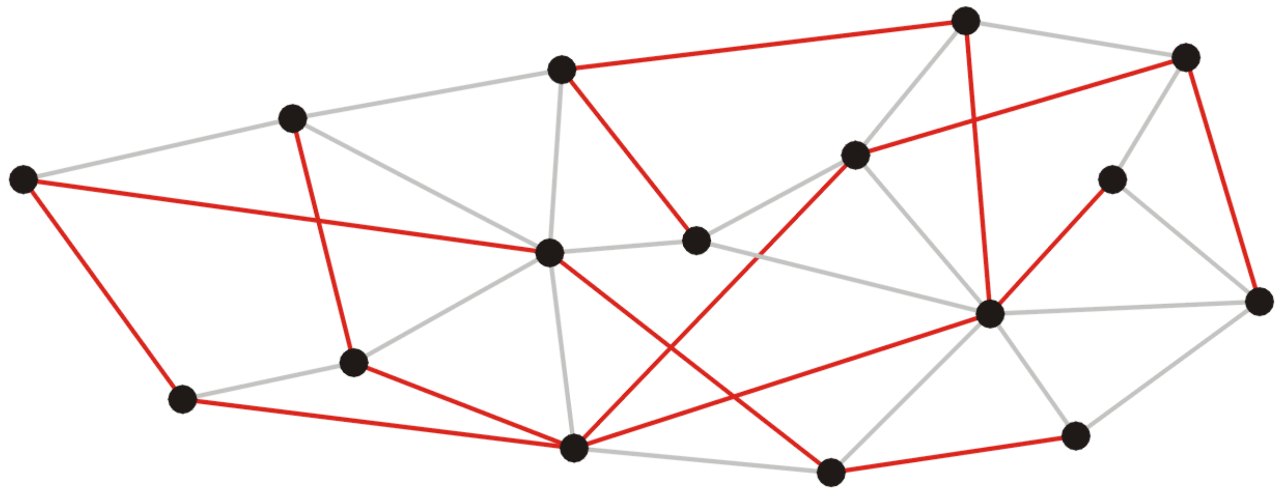
Spanning Trees

- These 15 edges form a minimum spanning tree



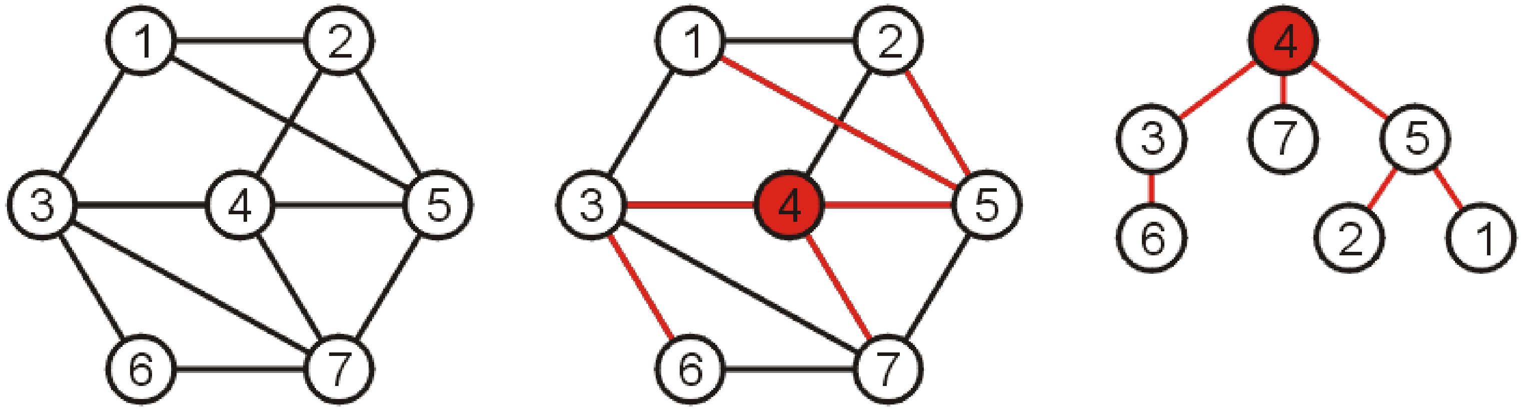
Spanning Trees

- As do these 15 edges:



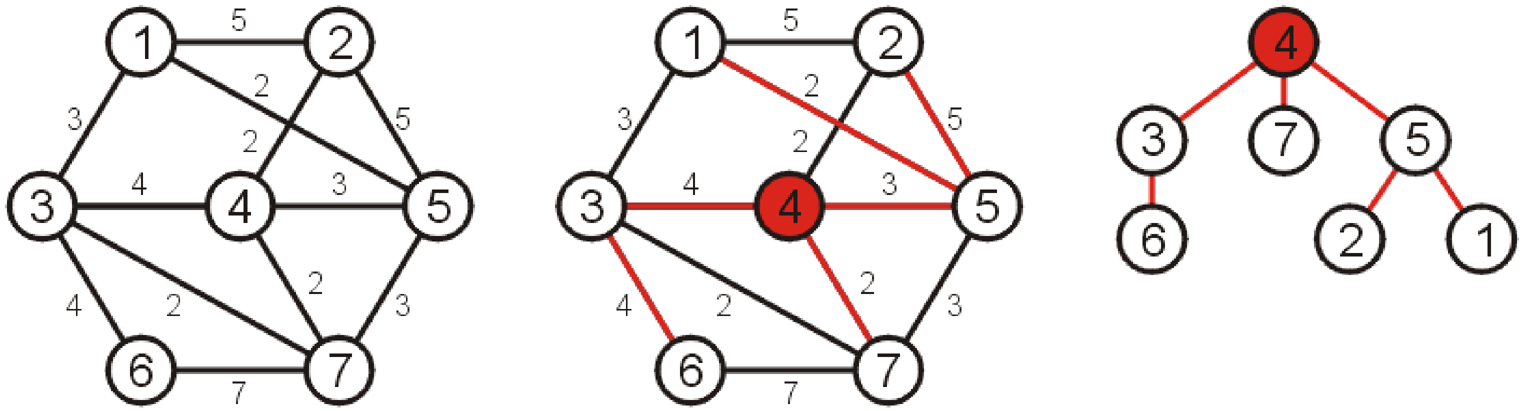
Spanning Trees

- Such a collection of edges is called a *tree* because if any vertex is taken to be the root, we form a tree by treating the adjacent vertices as children, and so on...



Spanning Trees on Weighted Graphs

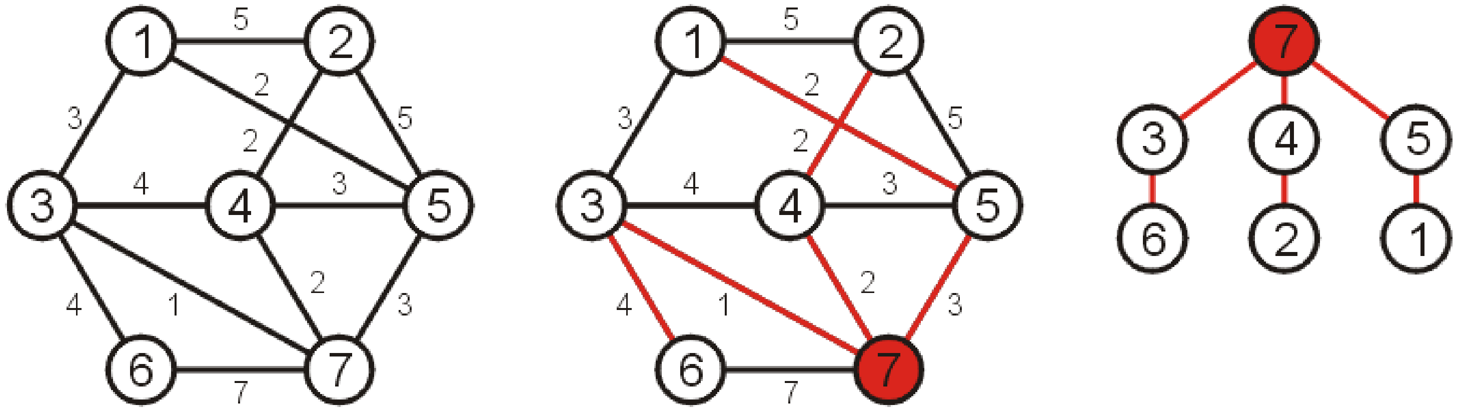
- The weight of a spanning tree is the sum of the weights on all the edges which comprise the spanning tree
  - The weight of this spanning tree is 20



Minimum Spanning Tree

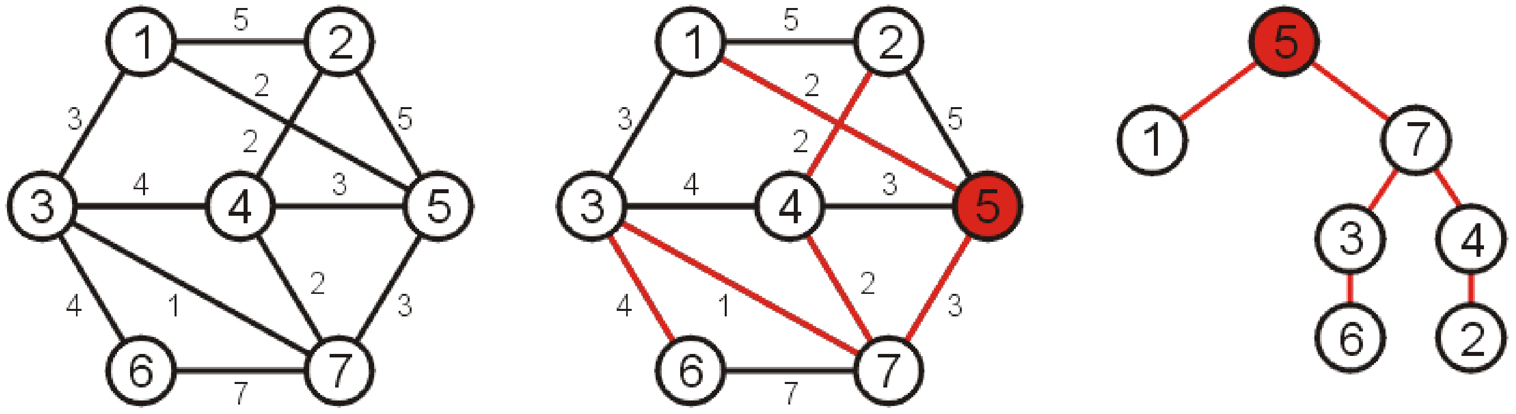
- Which spanning tree which minimizes the weight?
  - Such a tree is termed a *minimum spanning tree*

- The weight of this spanning tree is 14



Minimum Spanning Tree

- If we use a different vertex as the root, we get a different tree, however, this is simply the result of one or more rotations

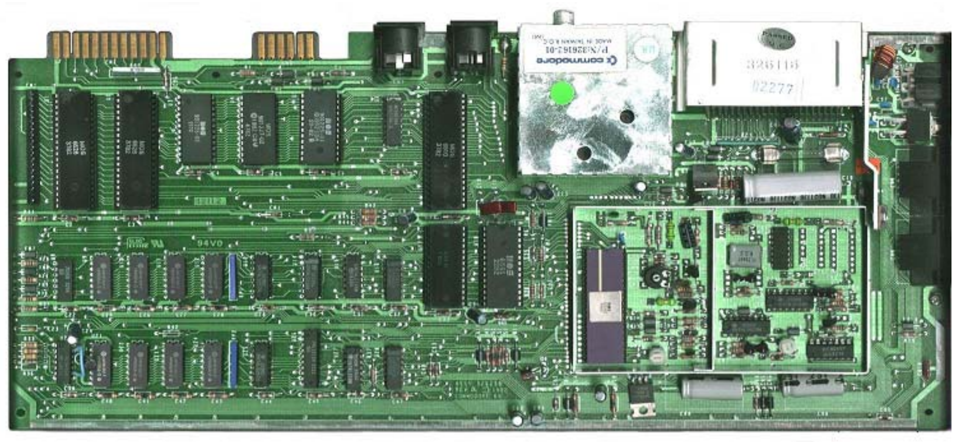


Unweighted Graphs

- Observation
  - In an unweighted graph, we nominally give each edge a weight of 1
  - Consequently, all minimum spanning trees have weight $|V| - 1$

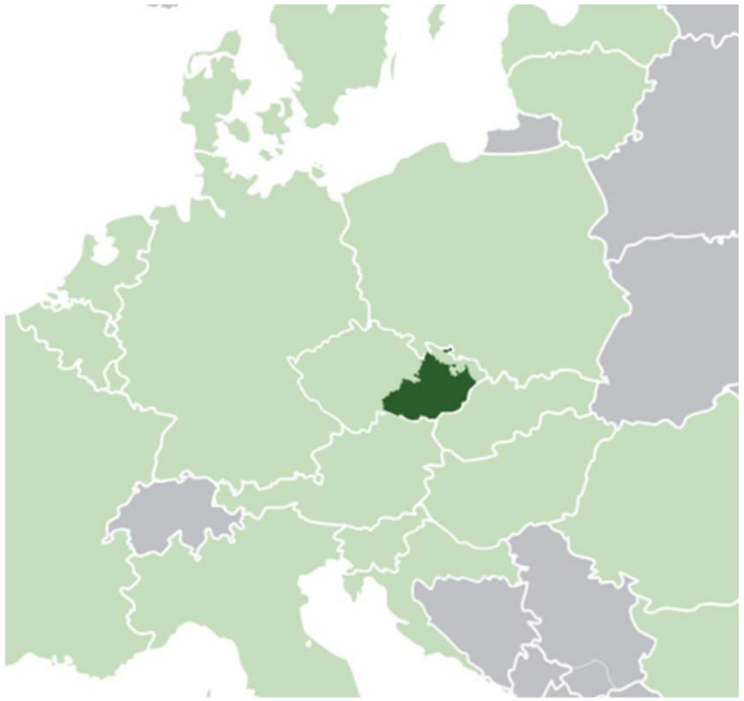
Application

- 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



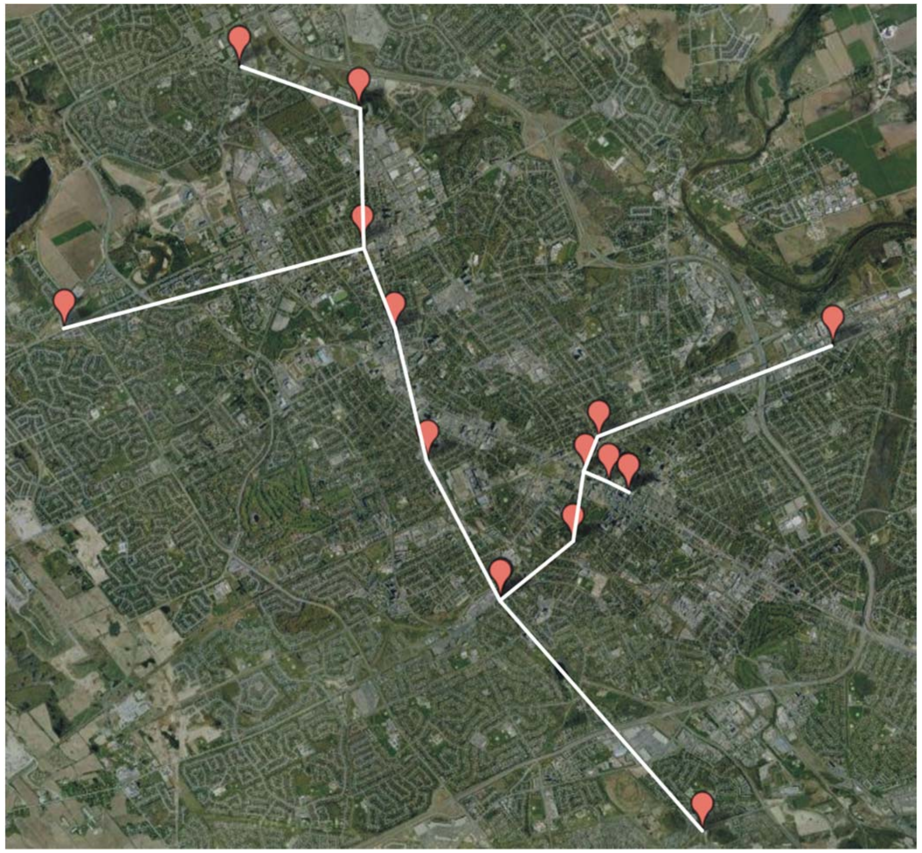
Application

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



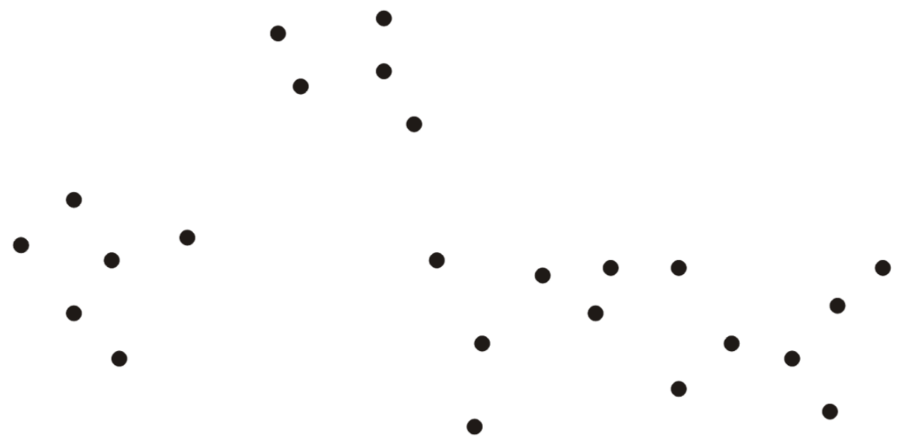
Application

- Consider attempting to find the best way of connecting a number of LANs



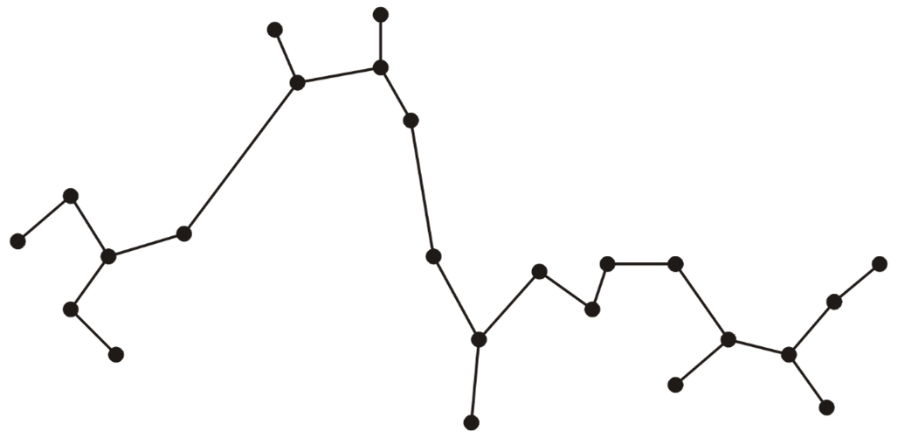
Application

- Consider an *ad hoc* wireless network
  - Any two terminals can connect with any others
- Problem:
  - Errors in transmission increase with transmission length
  - Can we find clusters of terminals which can communicate safely?



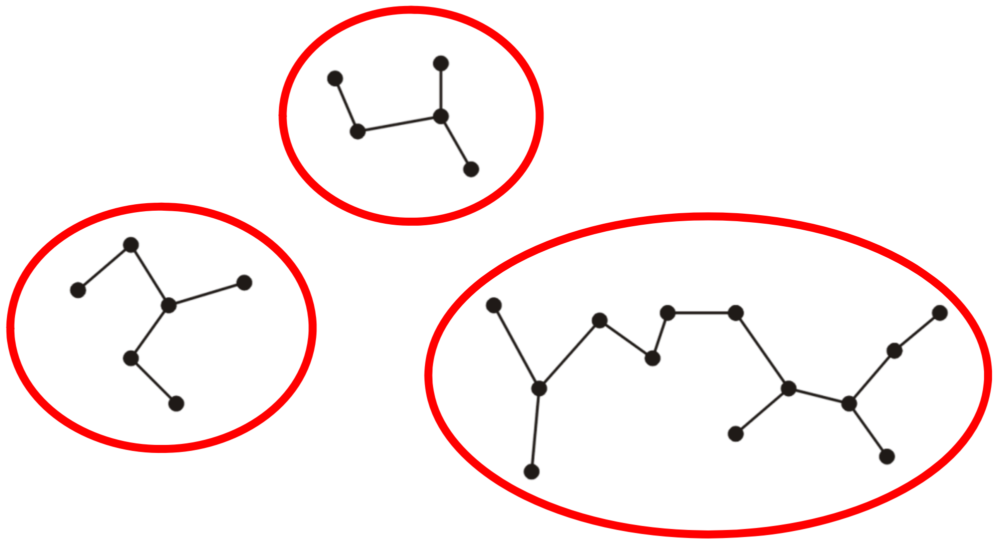
Application

- Find a minimum spanning tree



Application

- Remove connections which are too long
- This *clusters* terminals into smaller and more manageable sub-networks



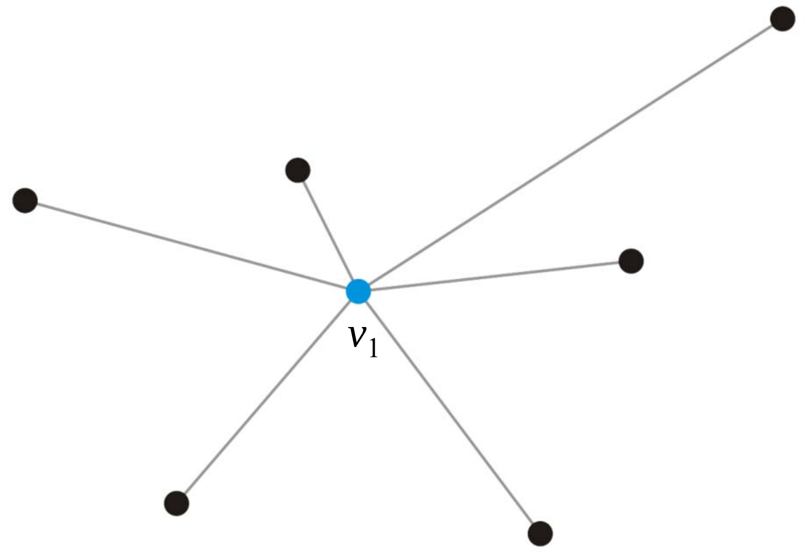
Algorithms

- Two common algorithms for finding minimum spanning trees are:
  - Prim’s algorithm
  - Kruskal’s algorithm

Prim’s Algorithm

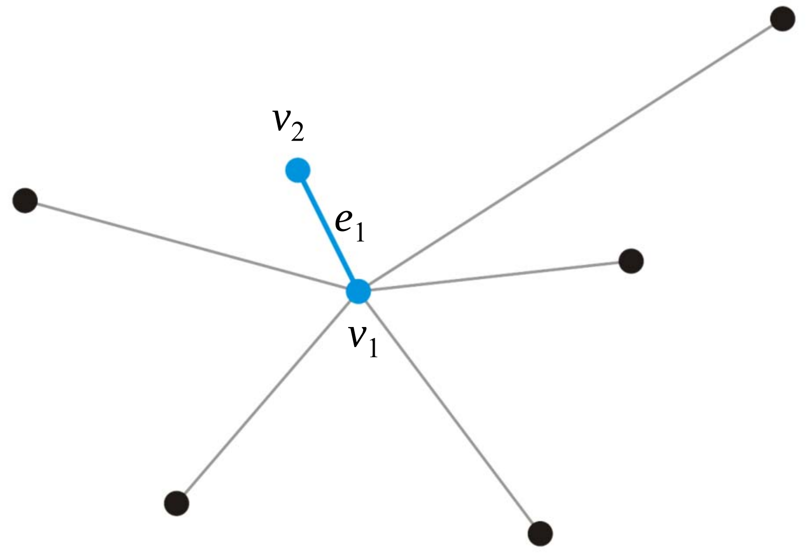
Strategy

- Suppose we take a vertex
  - Given a single vertex v_1, it forms a minimum spanning tree on one vertex



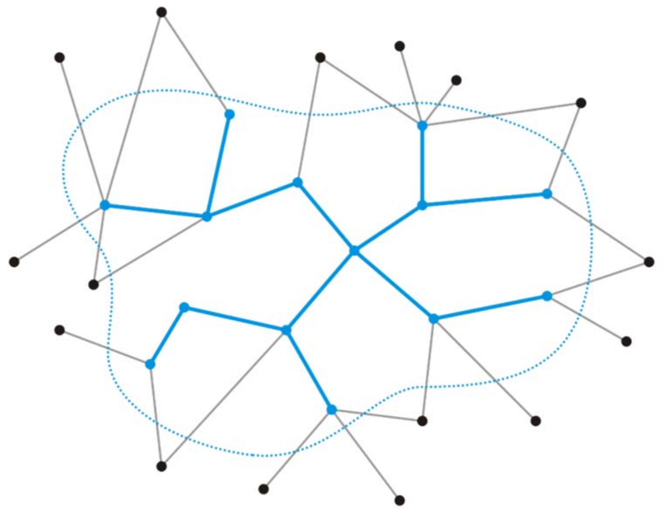
Strategy

- Add that adjacent vertex v_2
  that has a connecting edge e_1 of minimum weight
  - This forms a minimum spanning tree on our two vertices and e_1 must be in any minimum spanning tree containing the vertices v_1 and v_2



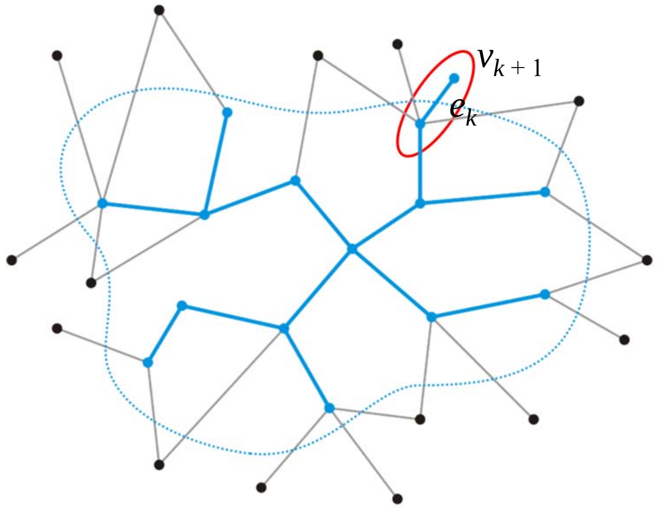
Strategy

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



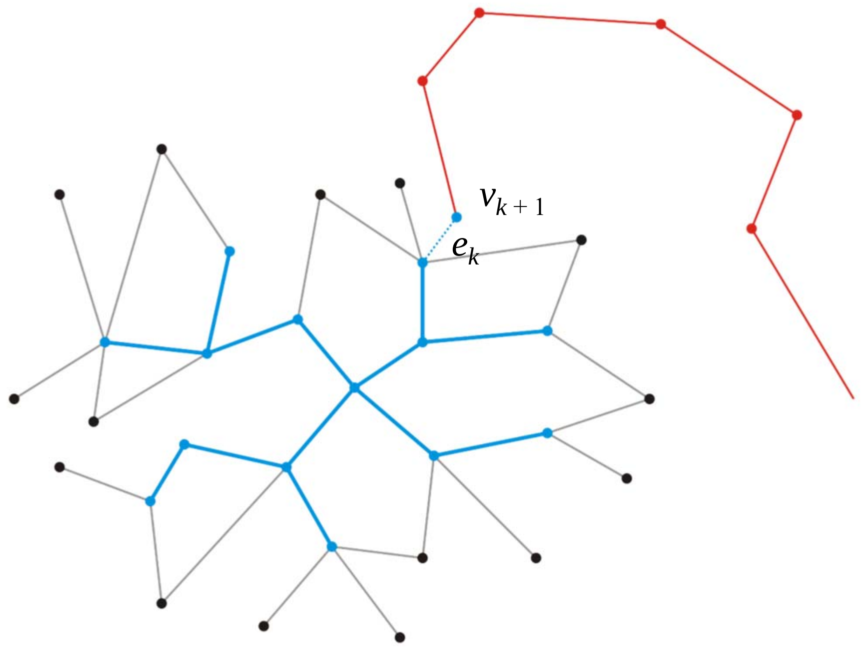
Strategy

- Add that edge e_k with least weight that connects this minimum spanning tree to a new vertex v_{k+1}
  - This creates a minimum spanning tree on $k + 1$ nodes—
    there is no other edge we could add that would connect this vertex
  - Does the new edge, however, belong to the minimum spanning tree on all n vertices?



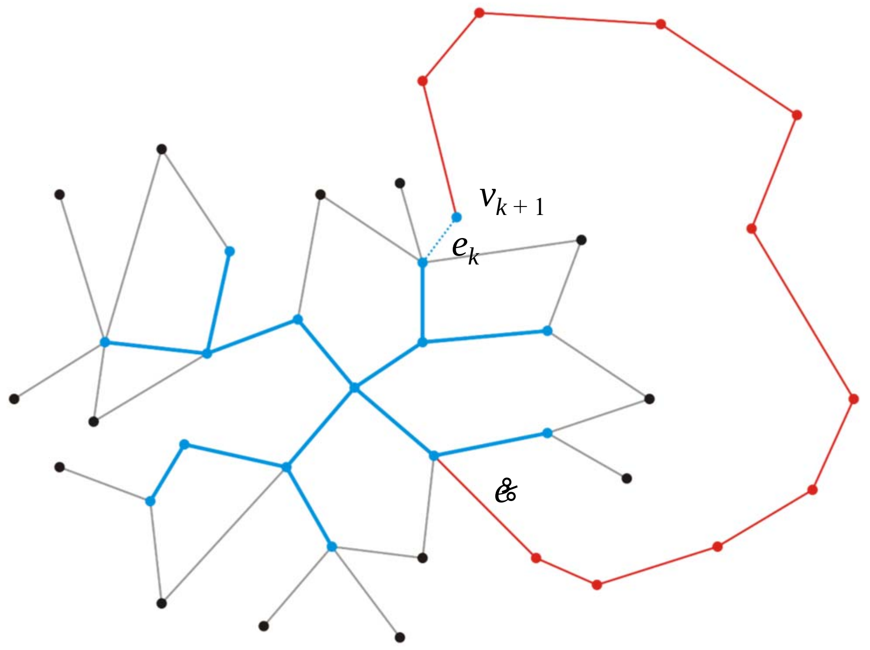
Strategy

- Suppose it does not
  - Thus, vertex v_{k+1} is connected to the minimum spanning tree via another sequence of edges



Strategy

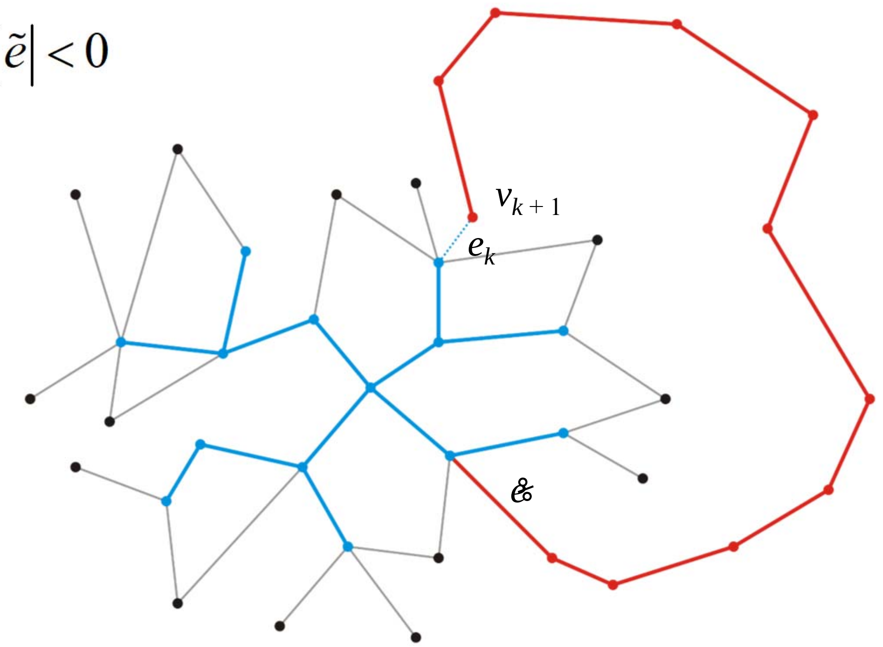
- Because a minimum spanning tree is connected, there must be a path from vertex v_{k+1} back to our existing minimum spanning tree
  - It must be connected along some edge &



Strategy

- Let w be the weight of this minimum spanning tree
  - Recall, however, that when we chose to add v_{k+1}, it was because e_k was the edge connecting an adjacent vertex with least weight
  - Therefore $|\tilde{e}| > |e_k|$ where $|e|$ represents the weight of the edge e

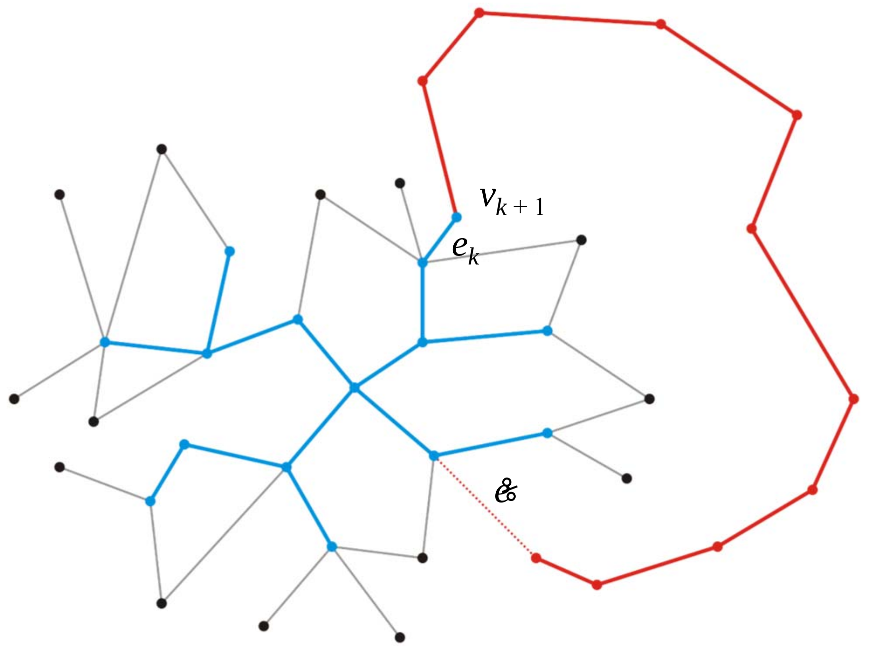
$$|e_k| - |\tilde{e}| < 0$$



Strategy

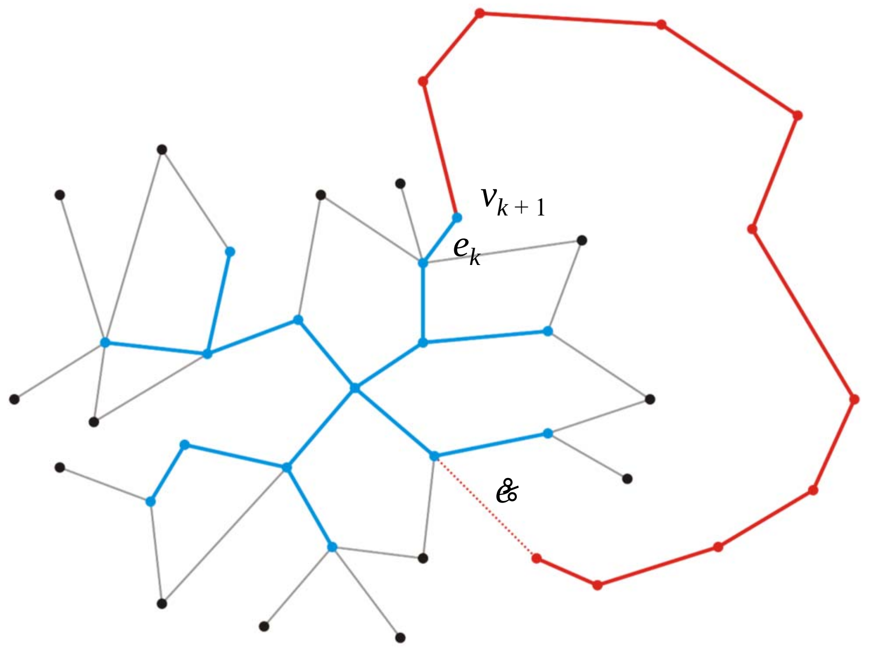
- Consider, however, suppose we swap edges and instead choose to include e_k and exclude $\tilde{e}$
  - The result is still a minimum spanning tree, but the weight is now

$$w + |e_k| - |\tilde{e}| \leq w$$



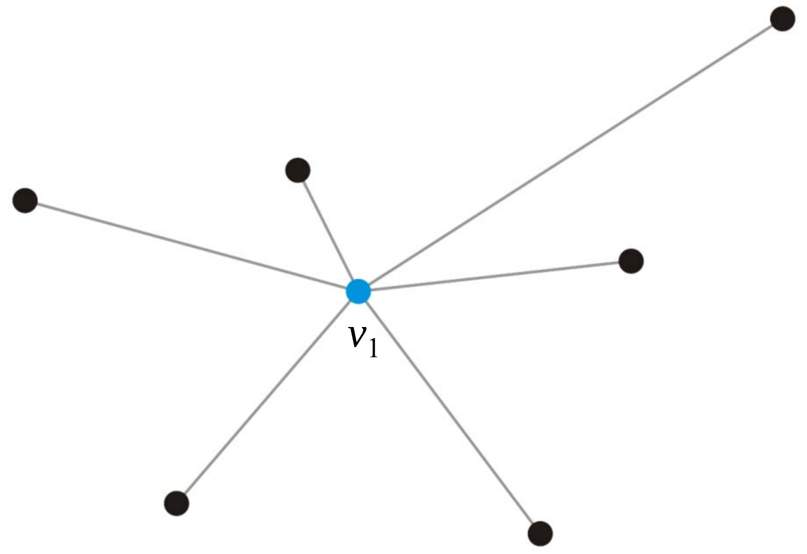
Strategy

- Thus, by swapping e_k for &, we have a spanning tree that has less weight than the so-called minimum spanning tree containing &
  - This contradicts our assumption that the spanning tree containing & was minimal
  - Therefore, our minimum spanning tree must contain e_k



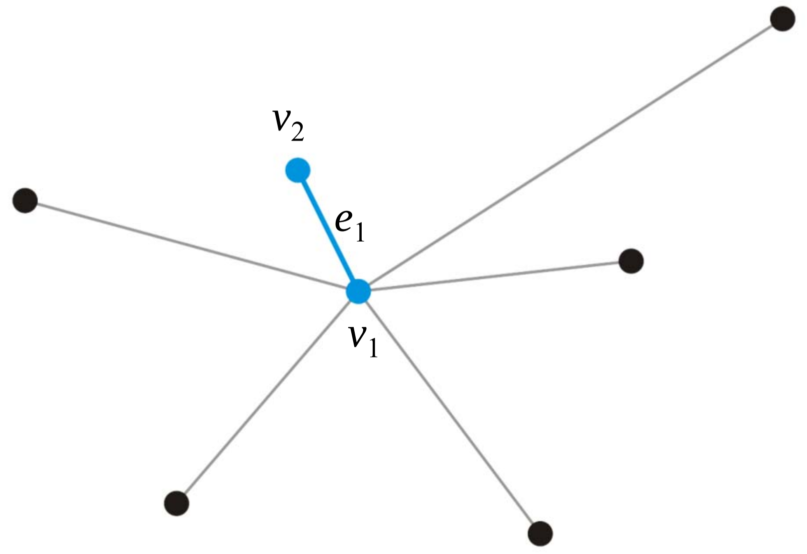
Strategy

- Recall that we did not prescribe the value of k, and thus, k could be any value, including $k = 1$
  - Given a single vertex v_1, it forms a minimum spanning tree on one vertex



Strategy

- Add that adjacent vertex v_2 that has a connecting edge e_1 of minimum weight
  - This forms a minimum spanning tree on our two vertices and e_1 must be in any minimum spanning tree containing the vertices v_1 and v_2



Prim’s Algorithm

- Start with an arbitrary vertex to form a minimum spanning tree on one vertex
- At each step, add a vertex v not yet in the minimum spanning tree that has an edge with least weight that connects v to the existing minimum spanning sub-tree
- Continue until we have $n - 1$ edges and n vertices

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

- Iterate while there exists an unvisited vertex with distance < ∞
  - Select an unvisited vertex with minimum distance
  - Mark that vertex as having been visited
  - For each adjacent vertex, if the weight of the connecting edge is less than the current distance to that vertex:
    - Update the distance to equal the weight of the edge
    - Set the current vertex as the parent of the adjacent vertex

Prim's Algorithm

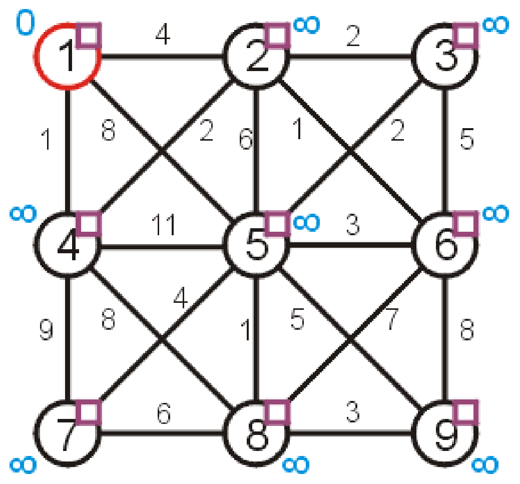
- Halting Conditions
  - There are no unvisited vertices which have a distance < ∞

- If all vertices have been visited, we have a spanning tree of the entire graph

- If there are vertices with distance ∞, then the graph is not connected and we only have a minimum spanning tree of the connected sub-graph containing the root

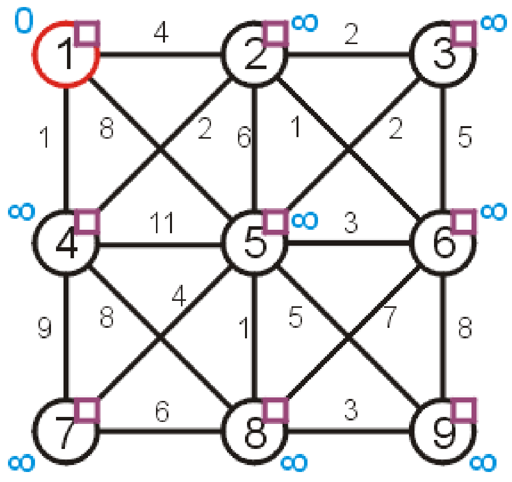
Example: Prim’s Algorithm

- Let us find the minimum spanning tree for the following un directed weighted graph



Example: Prim’s Algorithm

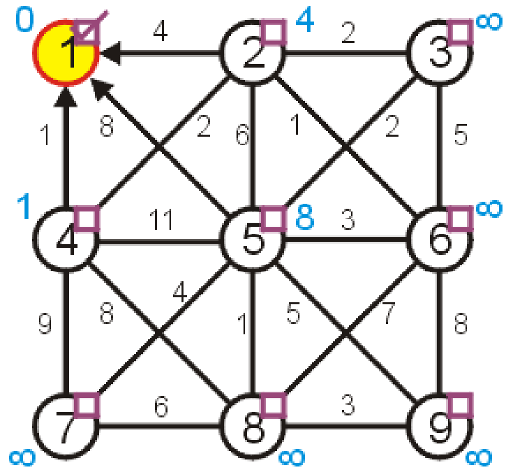
- First we set up the appropriate table and initialize it



		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

Example: Prim's Algorithm

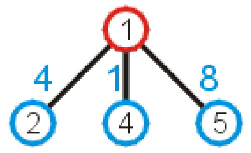
- Visiting vertex 1, we update vertices 2, 4, and 5



		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

Example: Prim’s Algorithm

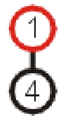
- What these numbers really mean is that at this point, we could extend the trivial tree containing just the root node by one of the three possible children:

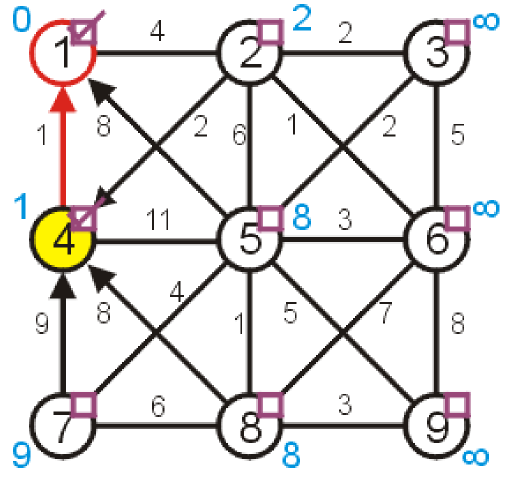


- As we wish to find a *minimum spanning tree*, it makes sense we add that vertex with a connecting edge with least weight

Example: Prim's Algorithm

- The next unvisited vertex with minimum distance is vertex 4
  - Update vertices 2, 7, 8
  - Don't update vertex 5





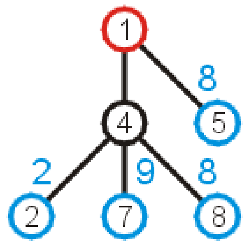
		Distance	Parent
1	T	0	0
2	F	2	4
3	F	∞	0
4	T	1	1
5	F	8	1
6	F	∞	0
7	F	9	4
8	F	8	4
9	F	∞	0

Example: Prim’s Algorithm

- Now that we have updated all vertices adjacent to vertex 4, we can extend the tree by adding one of the edges

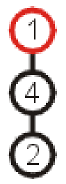
  (1, 5), (4, 2), (4, 7), or (4, 8)

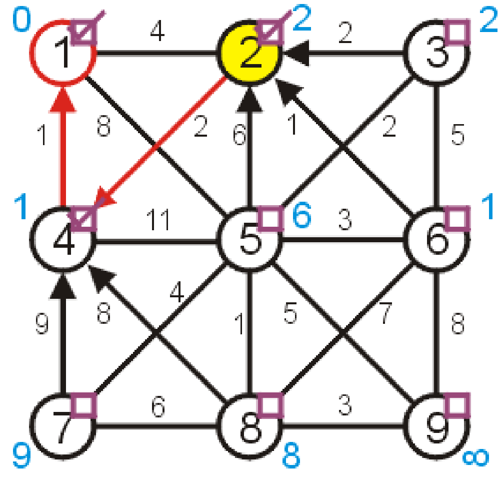
- We add that edge with the least weight: (4, 2)



Example: Prim's Algorithm

- Next visit vertex 2
  - Update 3, 5, and 6

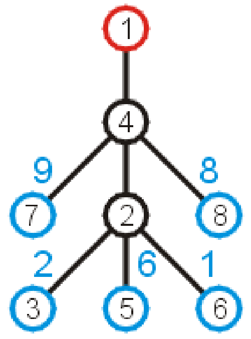




		Distance	Parent
1	T	0	0
2	T	2	4
3	F	2	2
4	T	1	1
5	F	6	2
6	F	1	2
7	F	9	4
8	F	8	4
9	F	∞	0

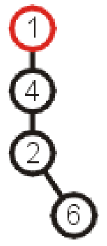
Example: Prim's Algorithm

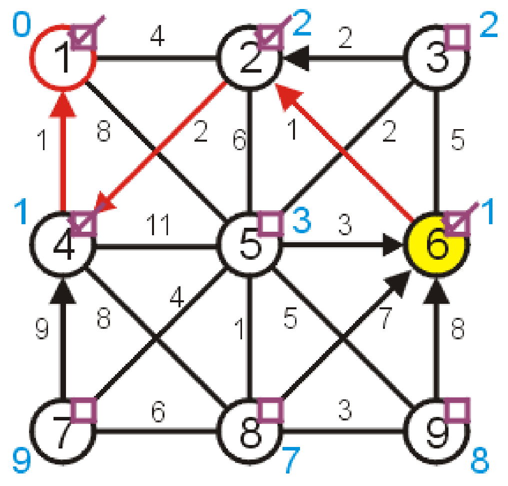
- Again looking at the shortest edges to each of the vertices adjacent to the current tree, we note that we can add (2, 6) with the least increase in weight



Example: Prim's Algorithm

- Next, we visit vertex 6:
  - update vertices 5, 8, and 9

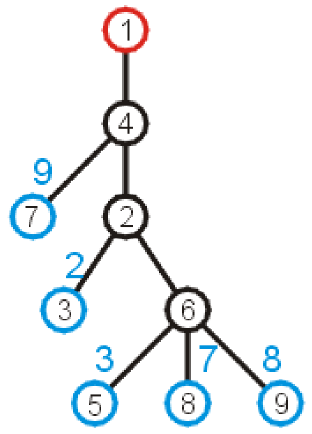




		Distance	Parent
1	T	0	0
2	T	2	4
3	F	2	2
4	T	1	1
5	F	3	6
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

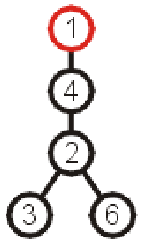
Example: Prim’s Algorithm

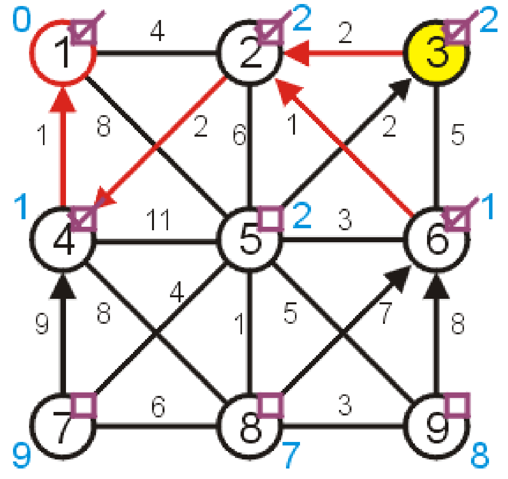
- The edge with least weight is (2, 3)
  - This adds the weight of 2 to the weight minimum spanning tree



Example: Prim's Algorithm

- Next, we visit vertex 3 and update 5

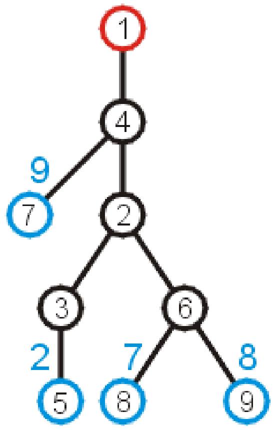




		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	F	2	3
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

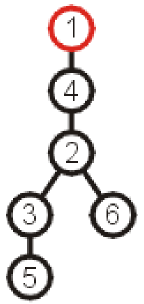
Example: Prim's Algorithm

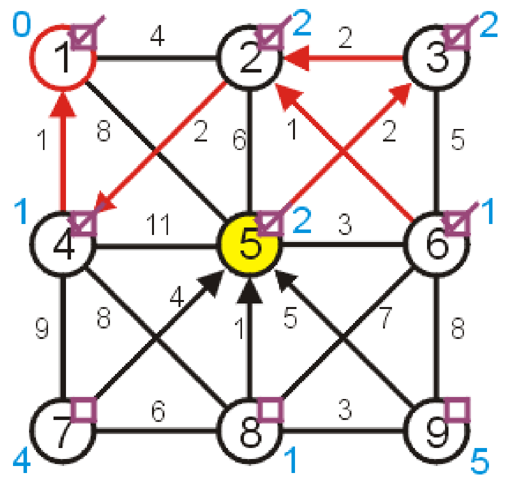
- At this point, we can extend the tree by adding the edge (3, 5)



Example: Prim's Algorithm

- Visiting vertex 5, we update 7, 8, 9

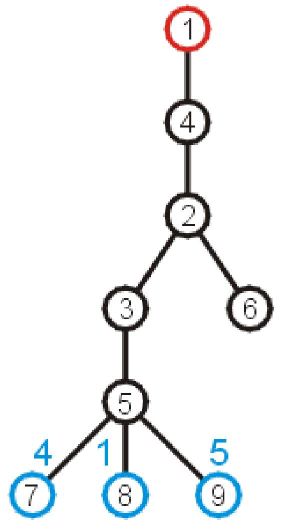




		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	F	1	5
9	F	5	5

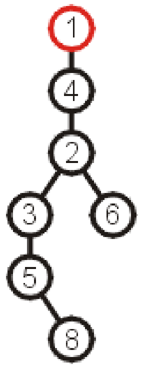
Example: Prim’s Algorithm

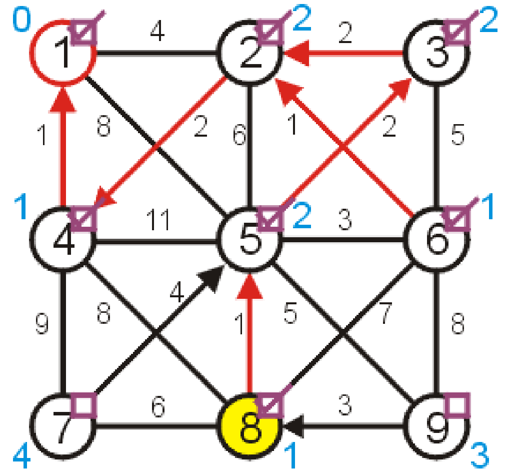
- At this point, there are three possible edges which we could include which will extend the tree
- The edge to 8 has the least weight



Example: Prim's Algorithm

- Visiting vertex 8, we only update vertex 9

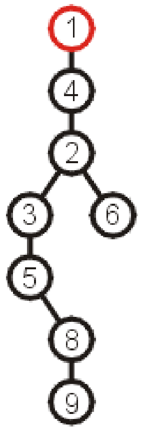


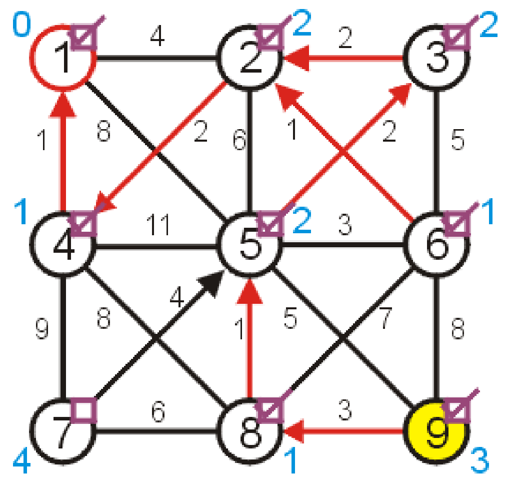


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	T	1	5
9	F	3	8

Example: Prim's Algorithm

- There are no other vertices to update while visiting vertex 9

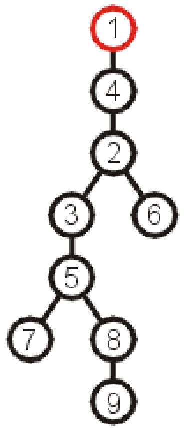


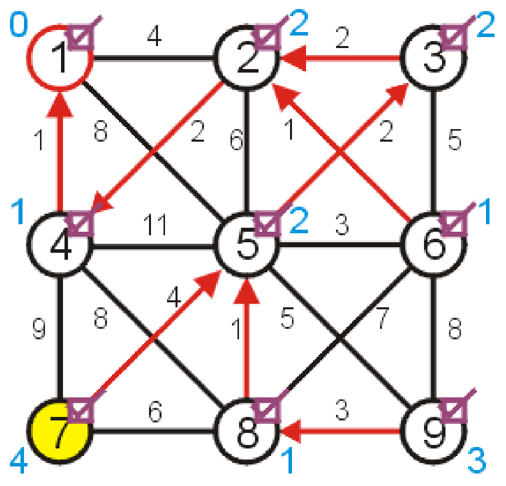


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	T	1	5
9	T	3	8

Example: 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

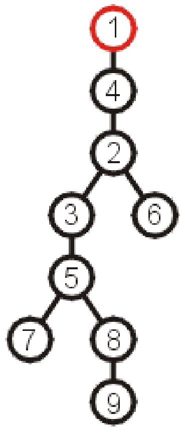
Example: Prim’s Algorithm

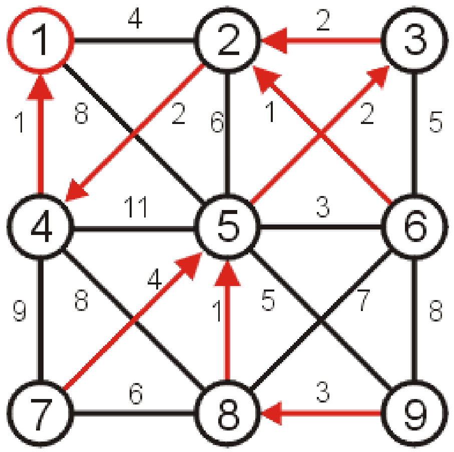
- At this point, there are no more unvisited vertices, and therefore we are done

- 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

Example: 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

Prim’s Algorithm

- To summarize:
  - we begin with a vertex which represents the root
  - starting with this trivial tree and iteration, we find the shortest edge which we can add to this already existing tree to expand it

- This is a reasonably efficient algorithm: the number of visits to vertices is kept to a minimum

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

- Can we do better?
  - Recall, we only need the shortest edge next
  - How about a priority queue?
    - Assume we are using a binary heap
    - We will have to update the heap structure—this requires additional work

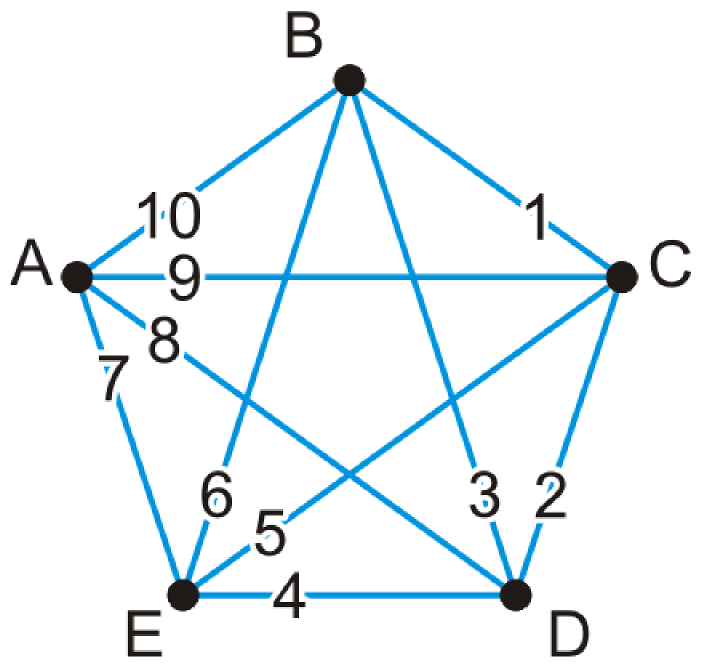
Analysis when using Priority Queue

- The initialization 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
  - Place the shortest distances into a priority queue
  - The size of the priority queue is $O(|V|)$
  - Thus, the work required for this is $O(|V| \lg(|V|))$
- Is this all the work that is necessary?
  - Recall that each edge visited may result in a new edge being pushed to the very top of the heap
  - Thus, the work required for this is $O(|E| \lg(|V|))$

- Thus, the total run time is
  $O(|V| \lg(|V|) + |E| \lg(|V|)) = O(|E| \lg(|V|))$

Analysis

- Here is a worst-case graph if we were to start with Vertex A
  - Assume that the adjacency lists are in order
  - Each time, the edge is percolated to the top of the heap



Kruskal’s Algorithm

Kruskal’s Algorithm

- Kruskal’s algorithm sorts the edges by weight and goes through the edges from least weight to greatest weight adding the edges to an empty graph so long as the addition does not create a cycle

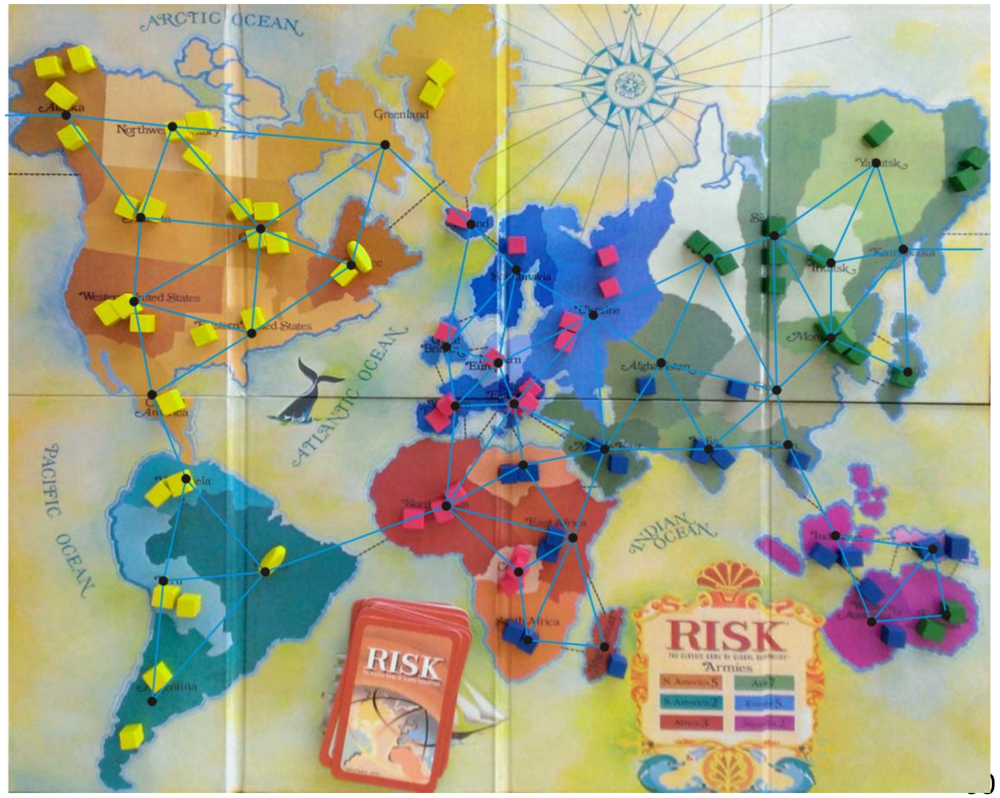
- The halting point is:
  - When $|V| - 1$ edges have been added
    - In this case we have a minimum spanning tree
  - We have gone through all edges, in which case, we have a forest of minimum spanning trees on all connected sub-graphs

Example: Kruskal’s Algorithm

- Consider the game of *Risk* from Parker Brothers
  - A game of world domination
  - The world is divided into 42 connected regions

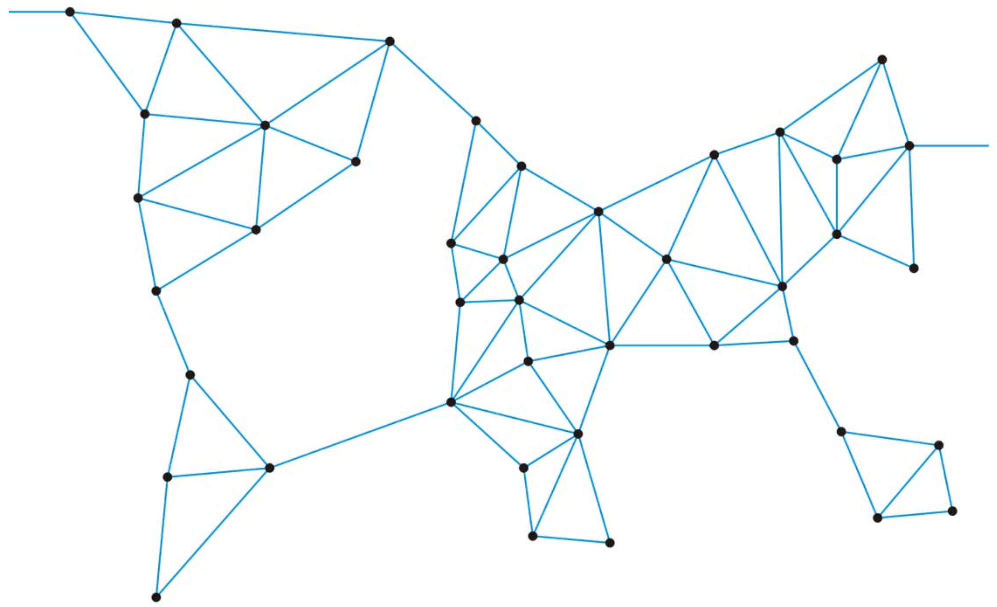
Example: Kruskal's Algorithm

- Consider the game of *Risk* from Parker Brothers
  - A game of world domination
  - The world is divided into 42 connected regions
  - The regions are vertices and edges indicate adjacent regions



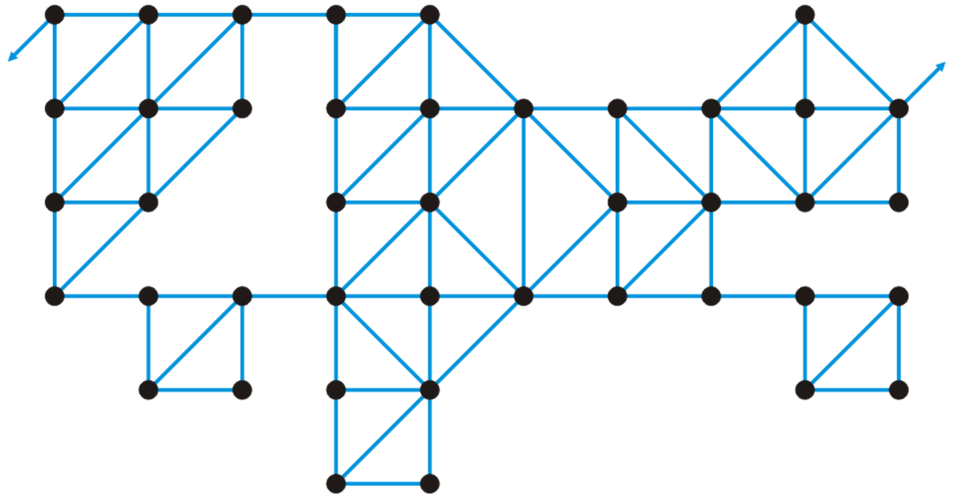
Example: Kruskal's Algorithm

- Consider the game of *Risk* from Parker Brothers
  - A game of world domination
  - The world is divided into 42 connected regions
  - The regions are vertices and edges indicate adjacent regions
  - The graph is sufficient to describe the game



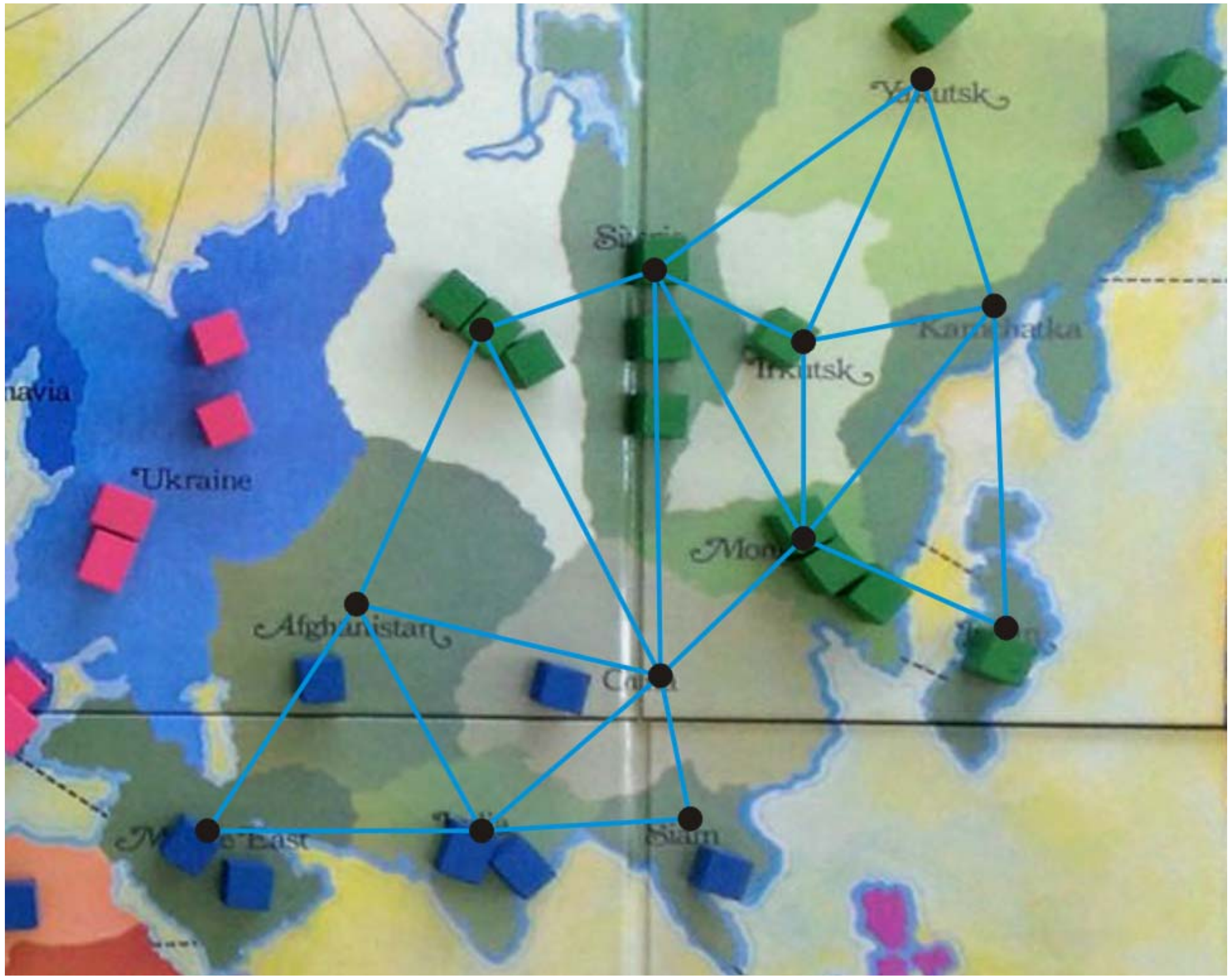
Example: Kruskal's Algorithm

- Consider the game of *Risk* from Parker Brothers
  - Here is a more abstract representation of the game board



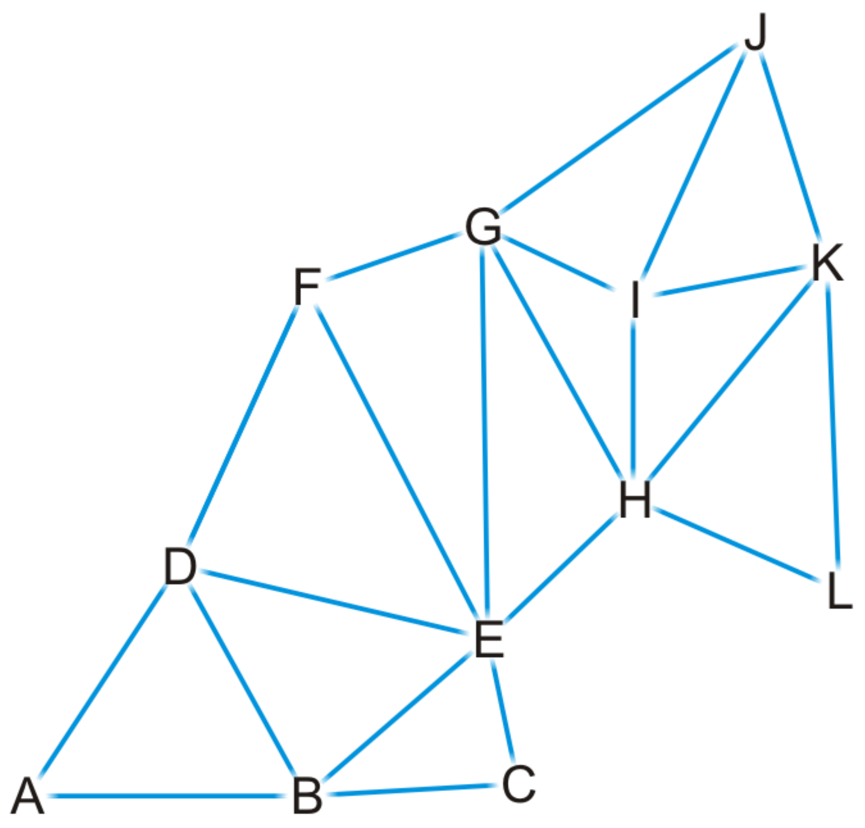
Example: Kruskal's Algorithm

- We'll focus on Asia



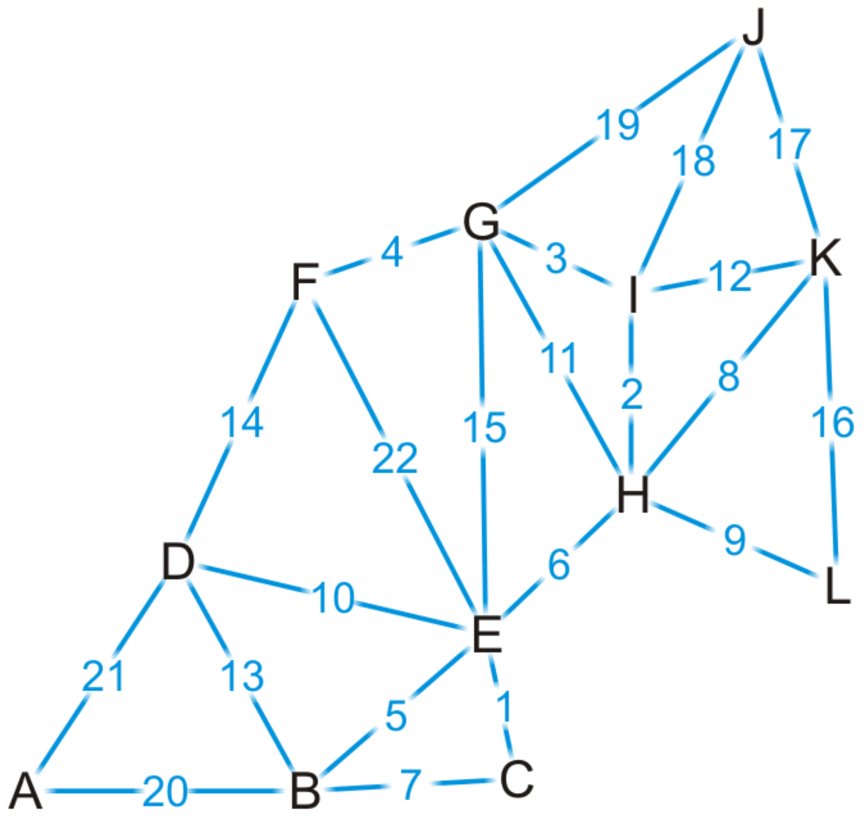
Example: Kruskal's Algorithm

- Here is our abstract representation



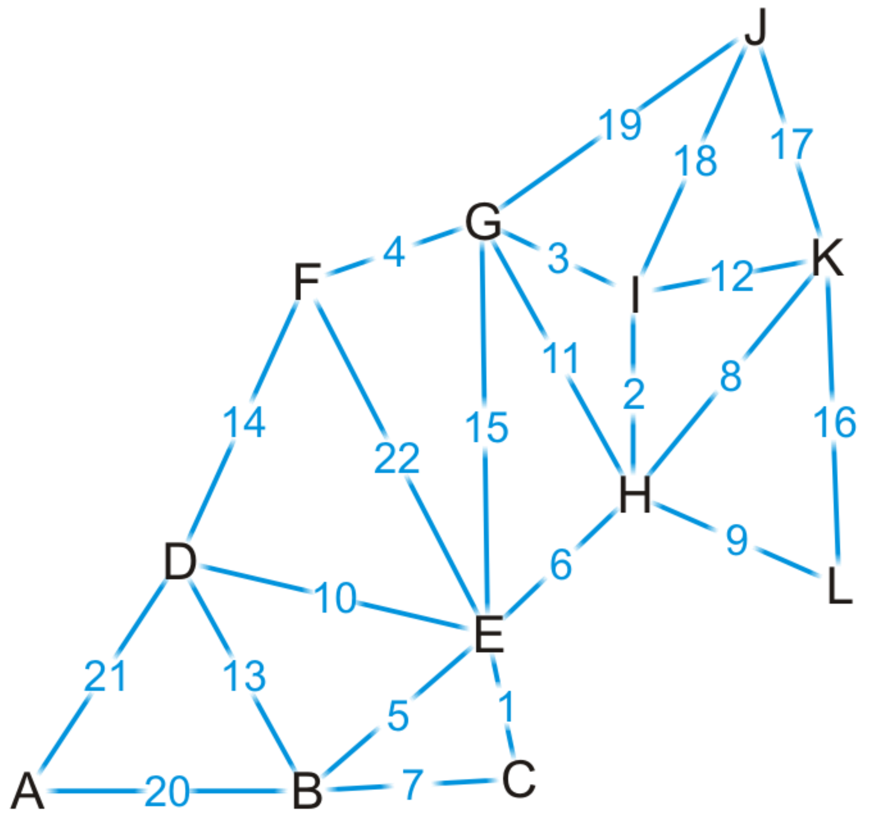
Example: Kruskal's Algorithm

- Let us give a weight to each of the edges



Example: Kruskal's Algorithm

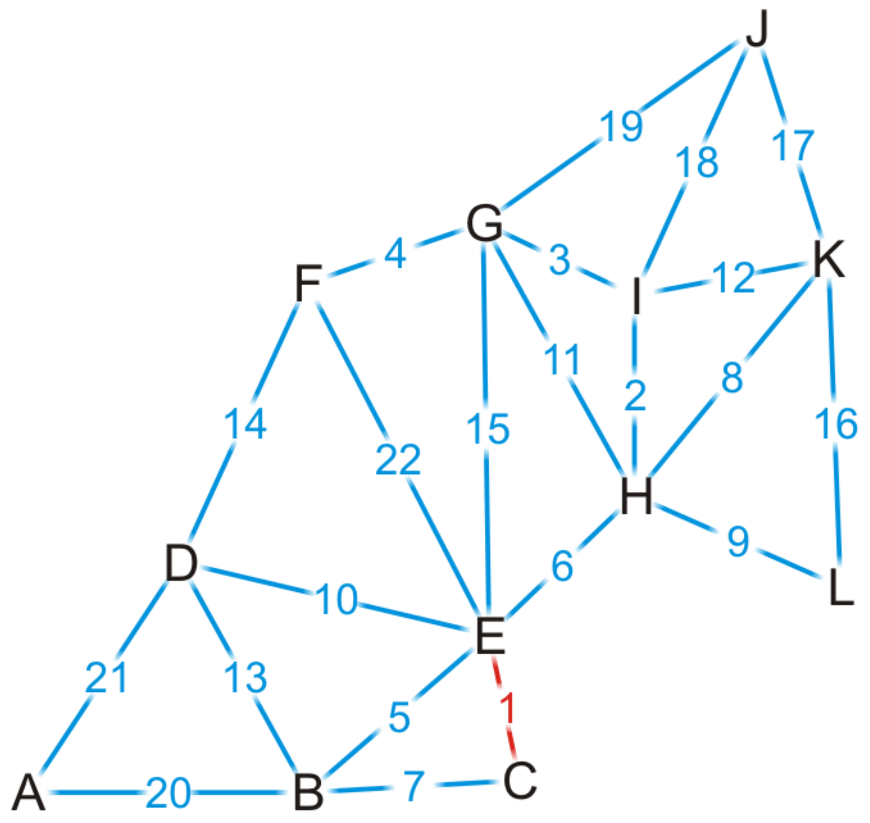
- First, we sort the edges based on weight



{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: Kruskal’s Algorithm

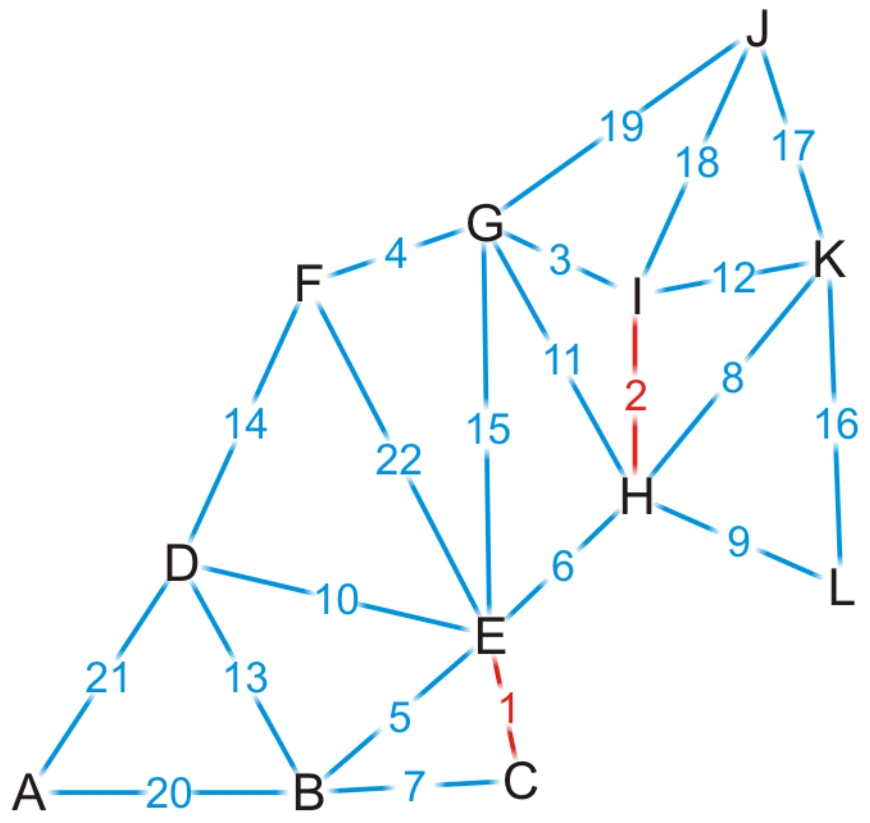
- We start by adding edge {C, E}



→ {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: Kruskal's Algorithm

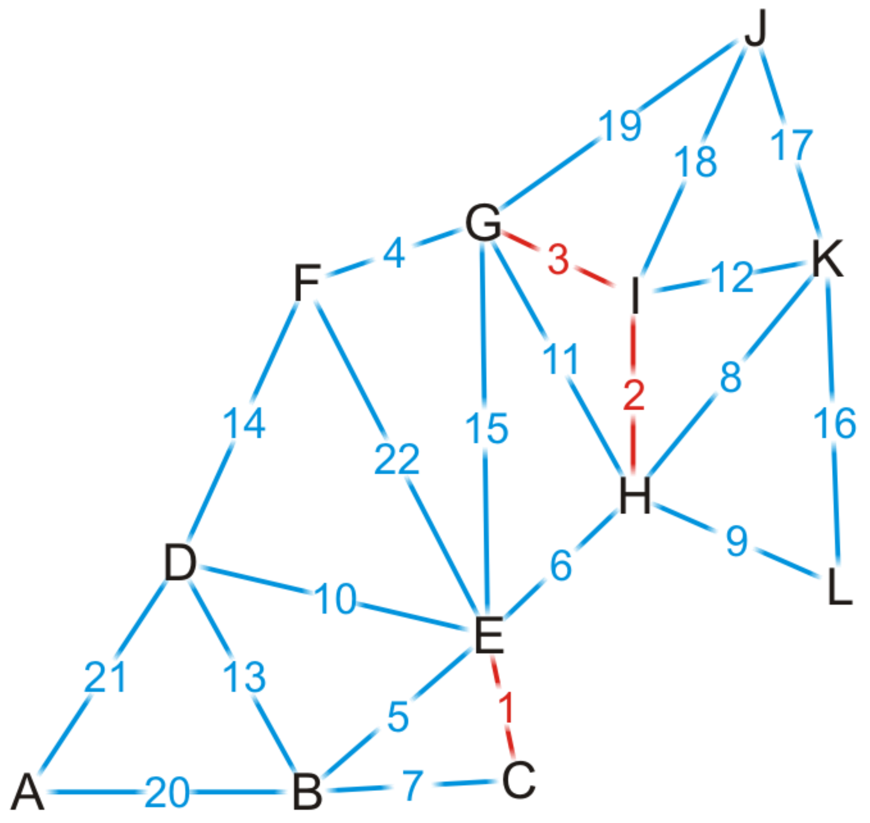
- We add edge {H, I}



{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: Kruskal's Algorithm

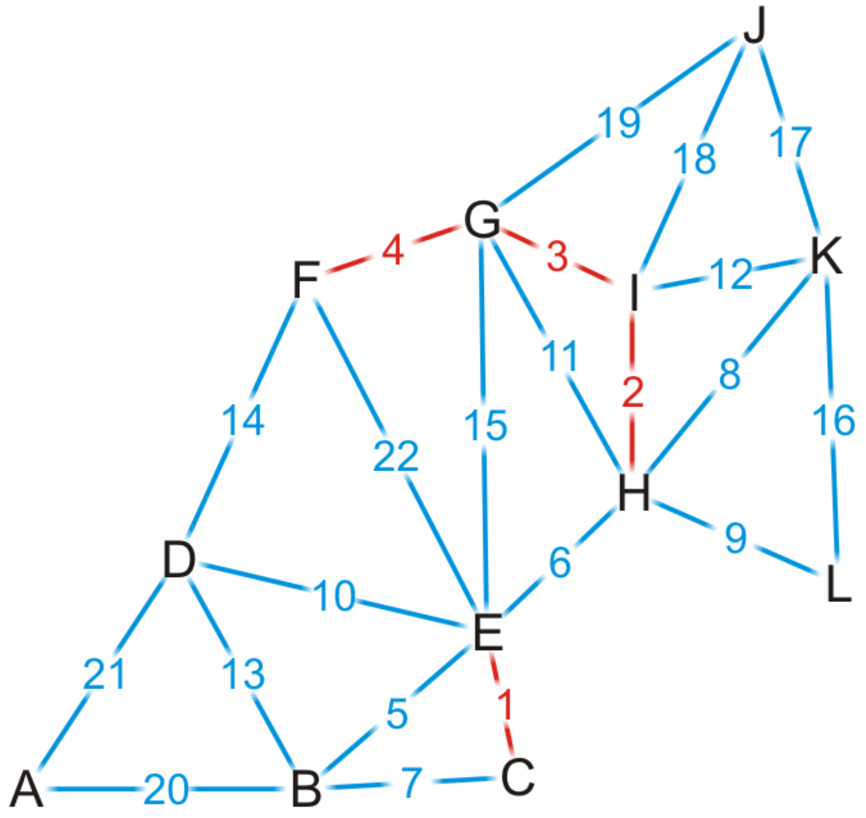
- We add edge {G, I}



{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: Kruskal's Algorithm

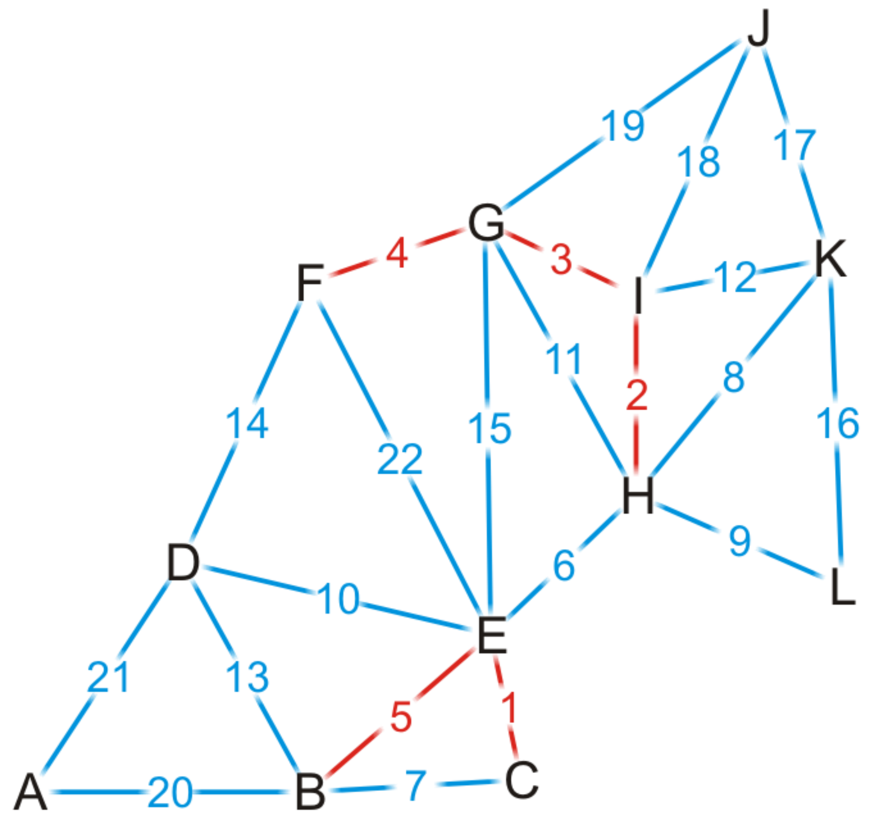
- We add edge {F, G}



{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: Kruskal's Algorithm

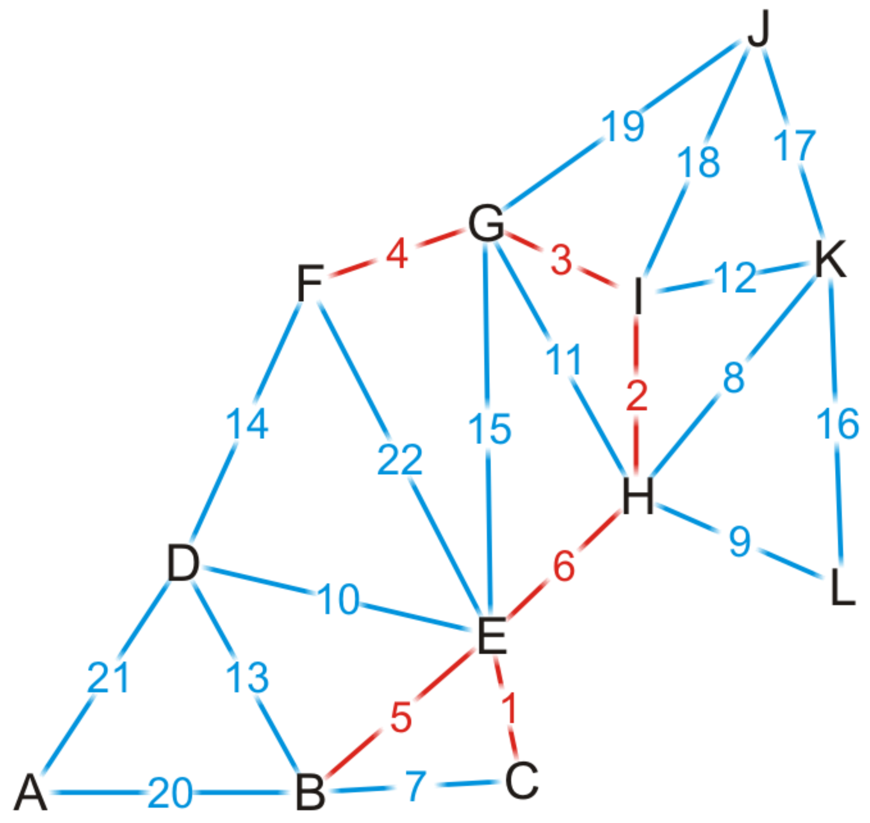
- We add edge {B, E}



{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: Kruskal’s Algorithm

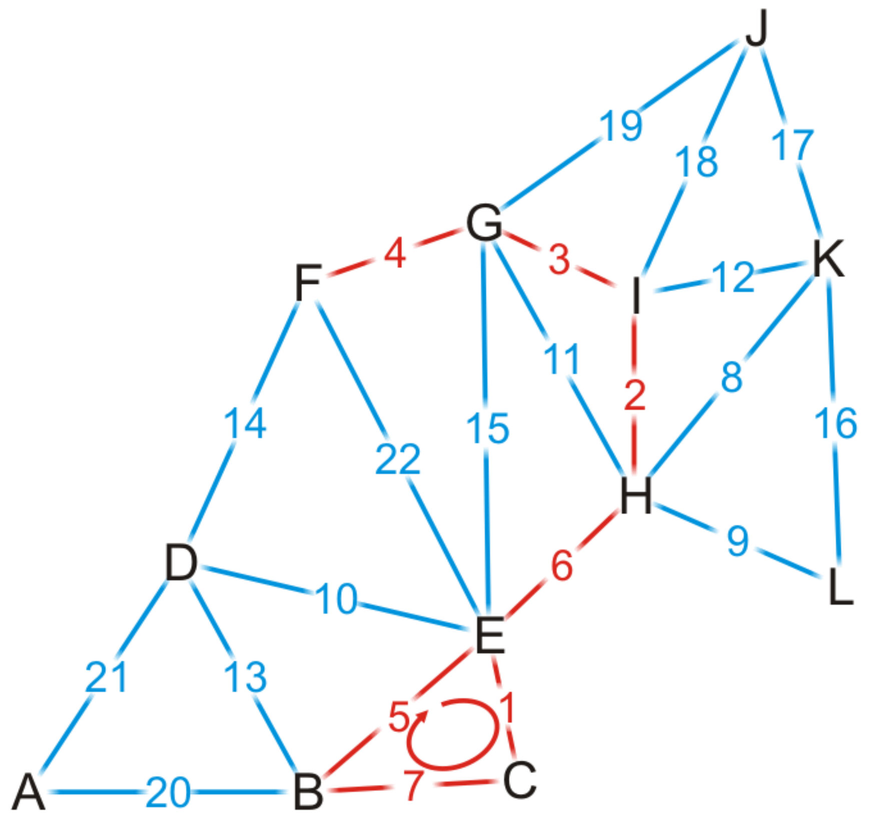
- We add edge {E, H}
  - This coalesces the two spanning sub-trees into one



{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: Kruskal’s Algorithm

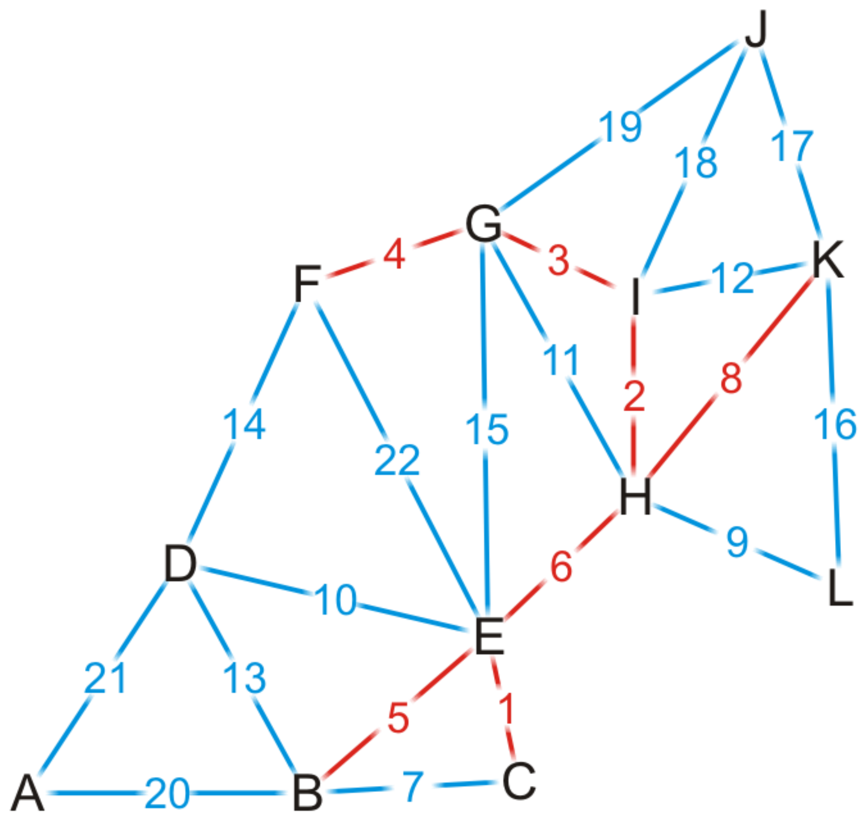
- We try adding {B, C}, but it creates a cycle



{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: Kruskal's Algorithm

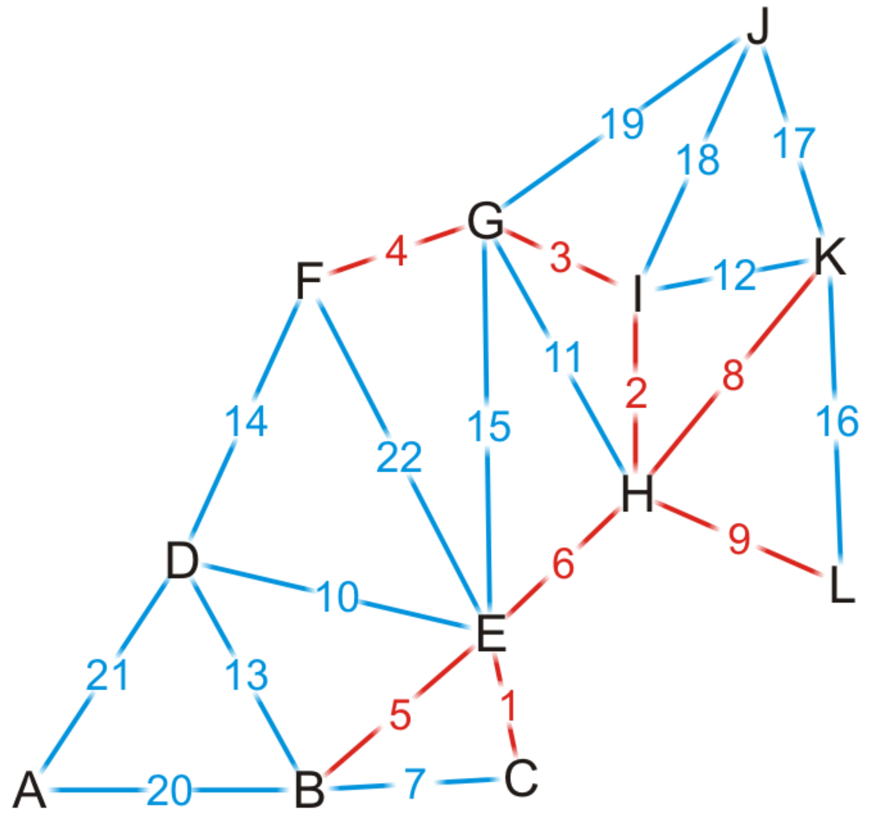
- We add edge {H, K}



{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: Kruskal's Algorithm

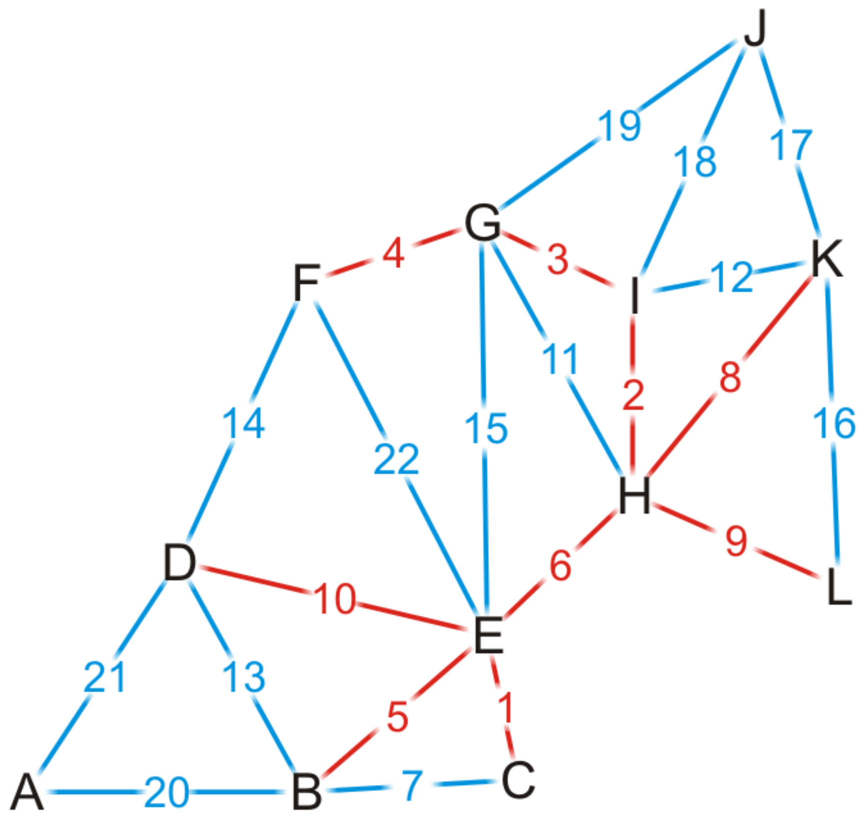
- We add edge {H, L}



{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: Kruskal's Algorithm

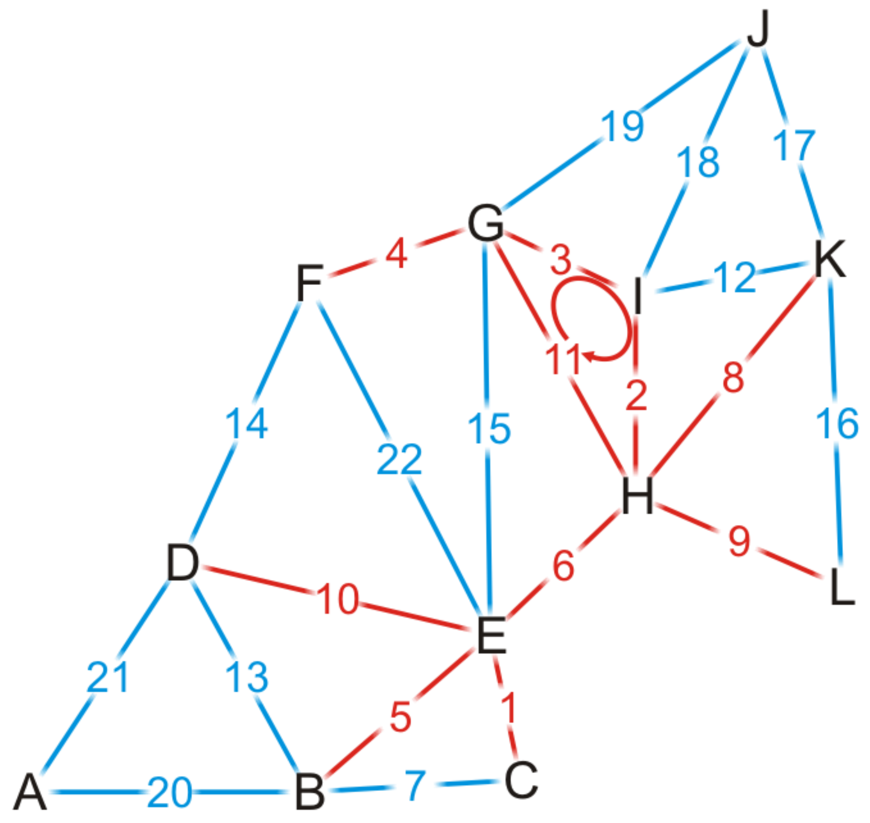
- We add edge {D, E}



{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: Kruskal’s Algorithm

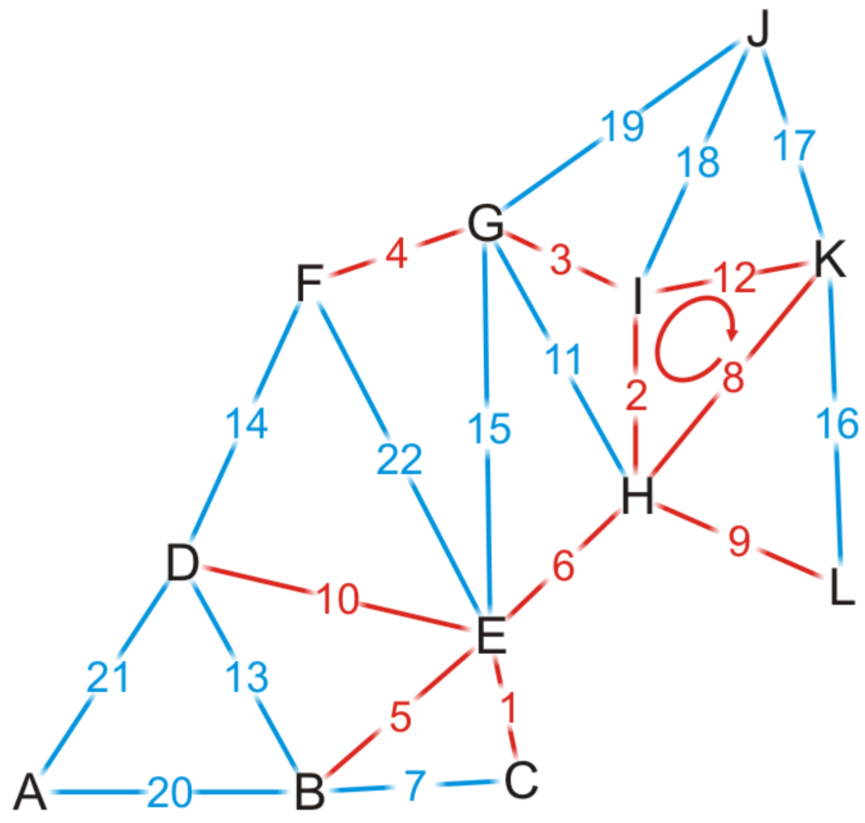
- We try adding {G, H}, but it creates a cycle



{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: Kruskal's Algorithm

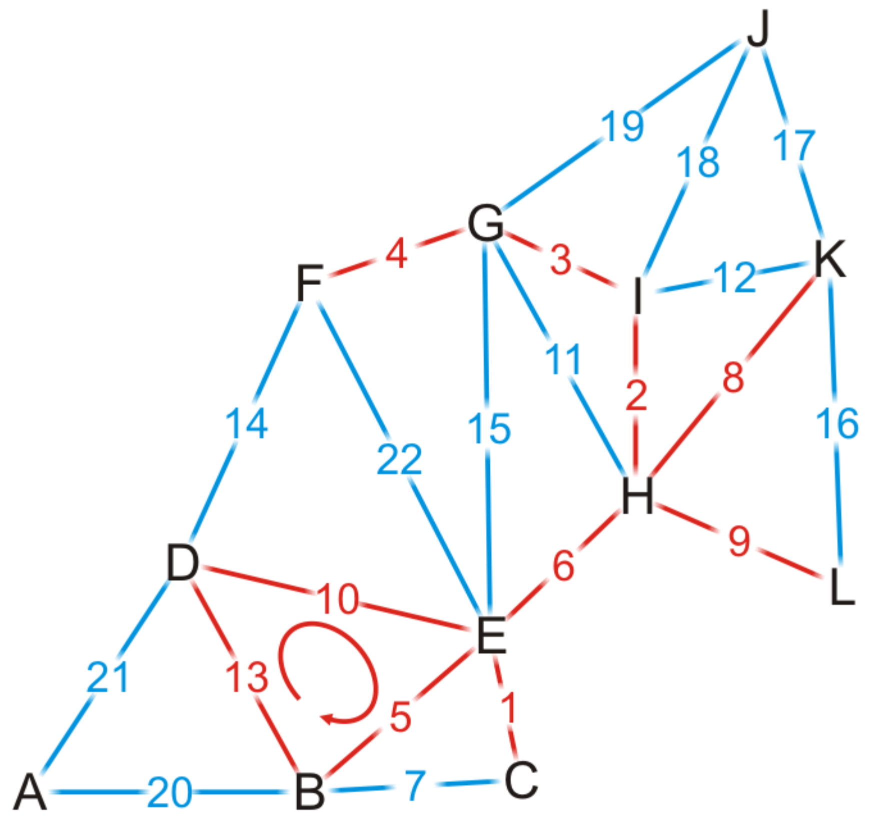
- We try adding {I, K}, but it creates a cycle



{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: Kruskal's Algorithm

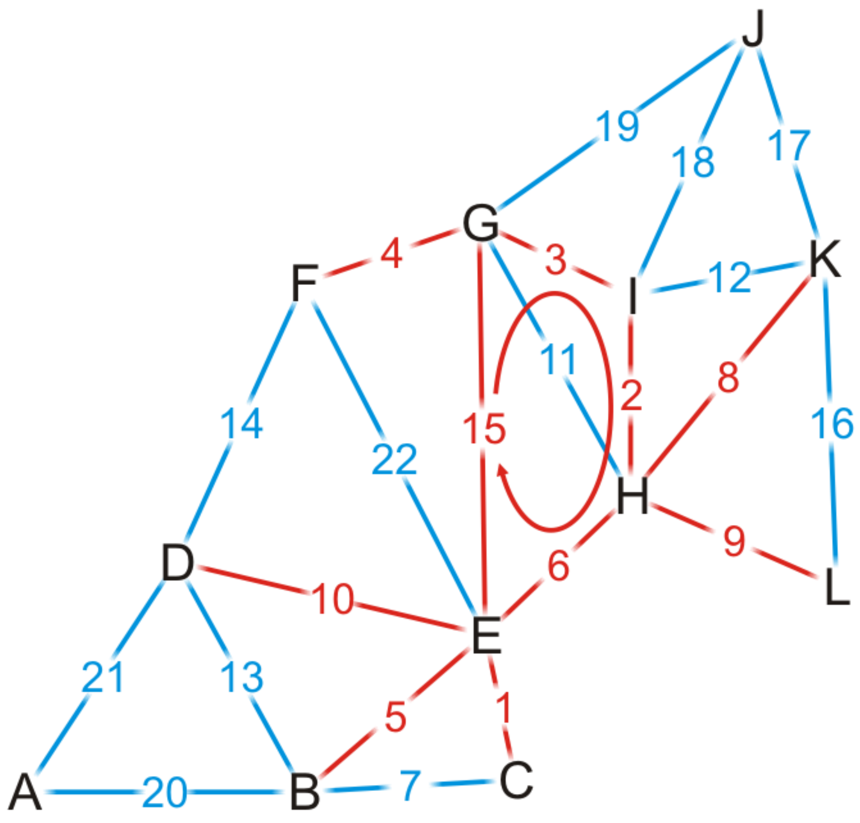
- We try adding {B, D}, but it creates a cycle



{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: Kruskal's Algorithm

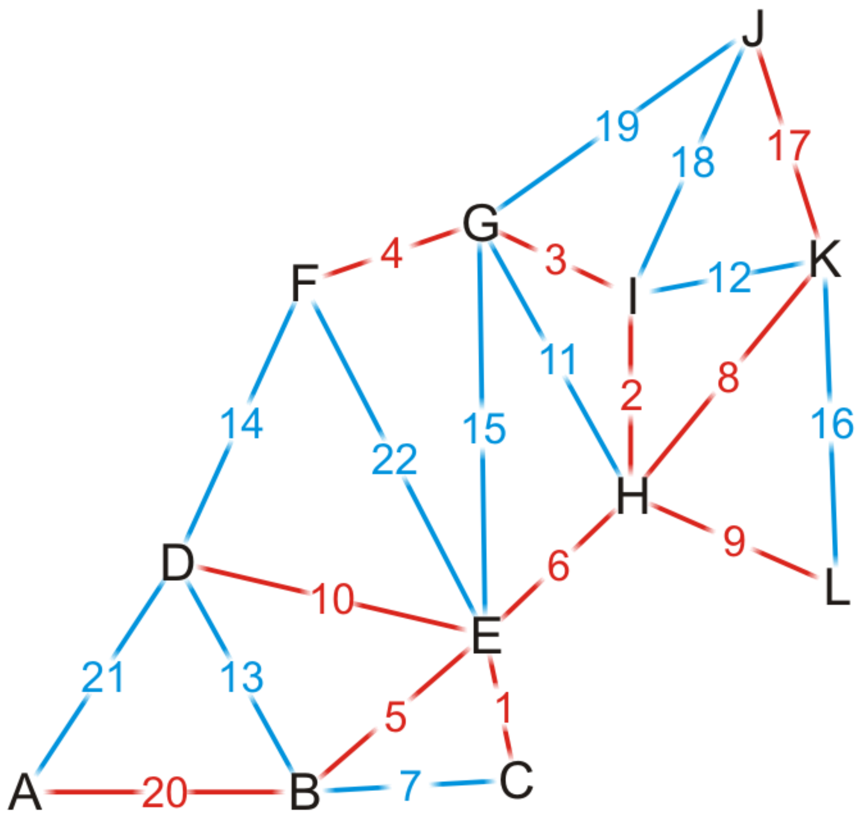
- We try adding {E, G}, but it creates a cycle



{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: Kruskal’s Algorithm

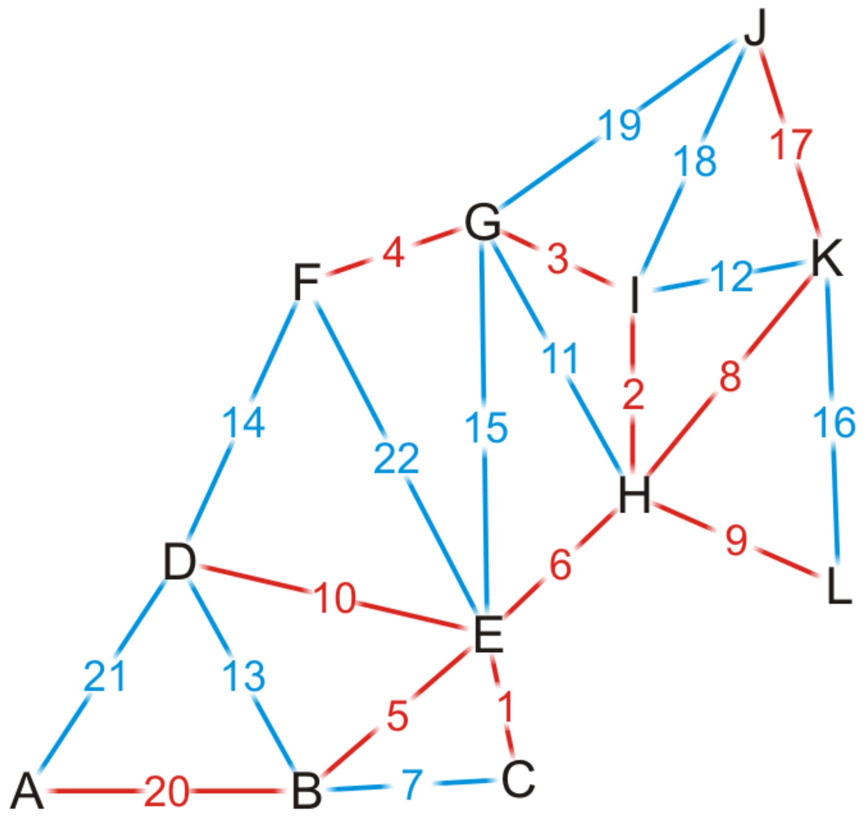
- By observation,
  we can still add edges {J, K} and {A, B}



{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: Kruskal’s Algorithm

- 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

- We would store the edges and their weights in an array
- We would sort the edges
- To determine if a cycle is created, we could perform a traversal
  - A run-time of $\mathrm{O}(|V|)$
- Consequently, the run-time would be $\mathrm{O}(|E| \lg(|E|) + |E|\cdot|V|)$
- However, $|E| = \mathrm{O}(|V|^2)$, so $\lg(E) = \mathrm{O}(\lg(|V|^2)) = \mathrm{O}(2\lg(|V|)) = \mathrm{O}(\ln(|V|))$
- Consequently, the run-time would be $\mathrm{O}(|E| \lg(|V|) + |E||V|) = \mathrm{O}(|E|\cdot|V|)$

- The complexity can be reduced by using “disjoint sets”.
  We will discuss about it later.

Any Question?