Minimum Spanning Tree (MST)

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- In this topic, we will
  - Define a spanning tree
  - Define the weight of a spanning tree in a weighted graph
  - Define a minimum spanning tree
  - Consider applications
  - List possible algorithms

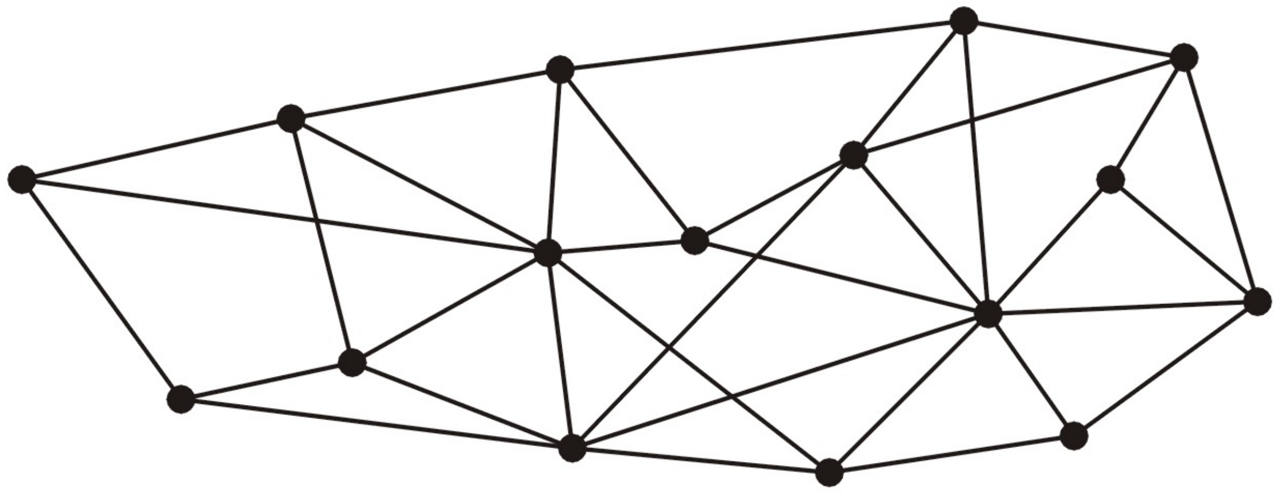
Spanning trees

- 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

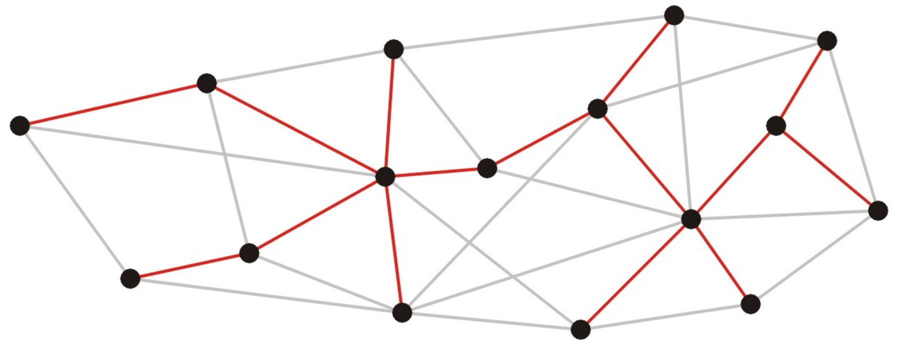
Spanning trees

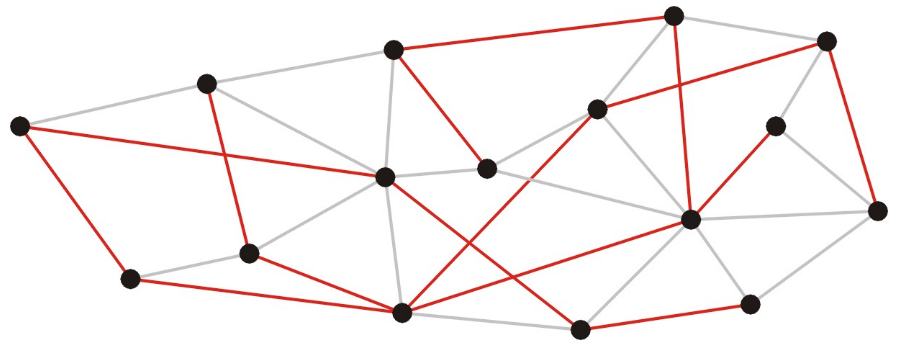
- This graph has 16 vertices and 35 edges



Spanning trees

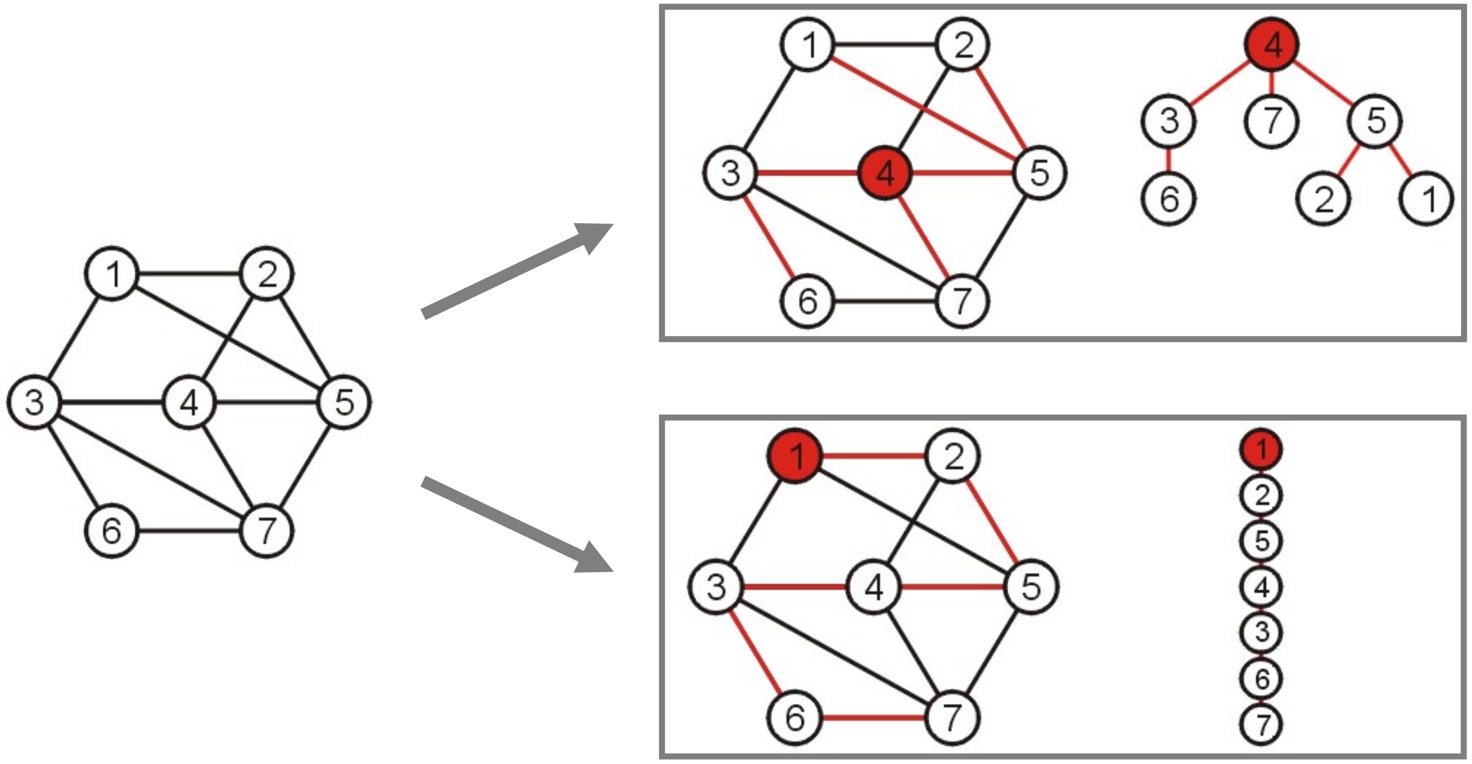
- These 15 edges form a minimum spanning tree





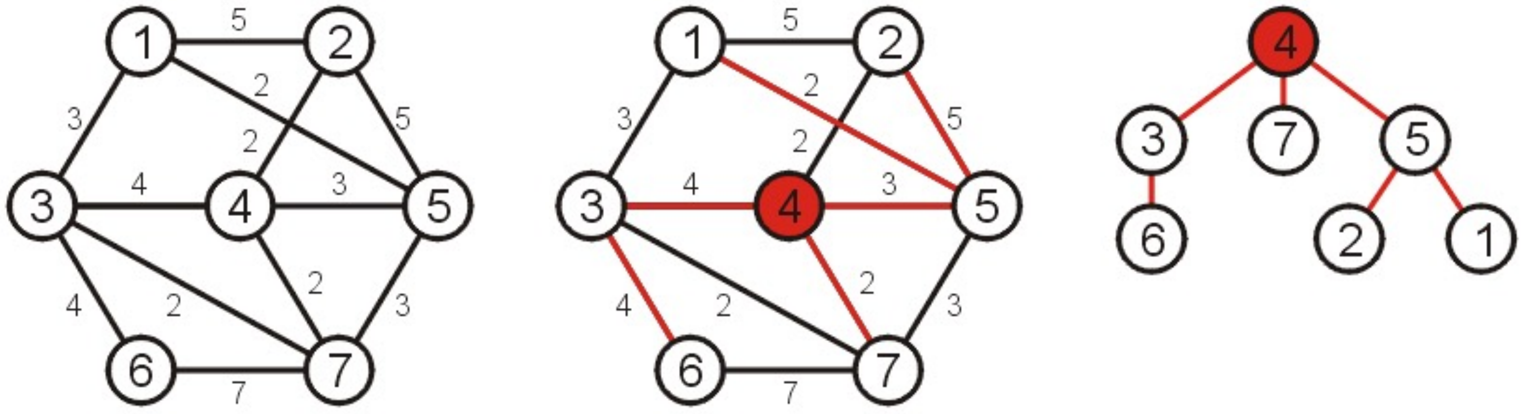
Spanning trees

- 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…



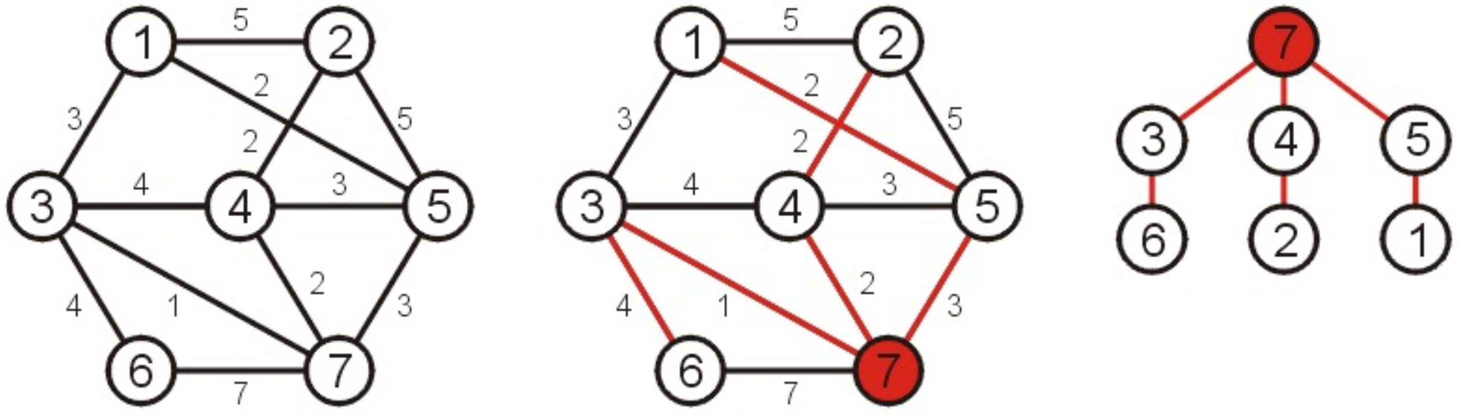
Spanning trees on weighted graphs

- **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



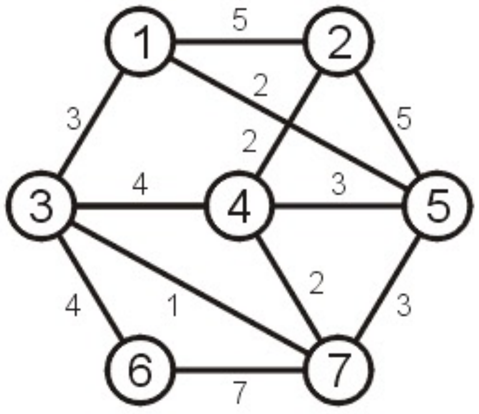
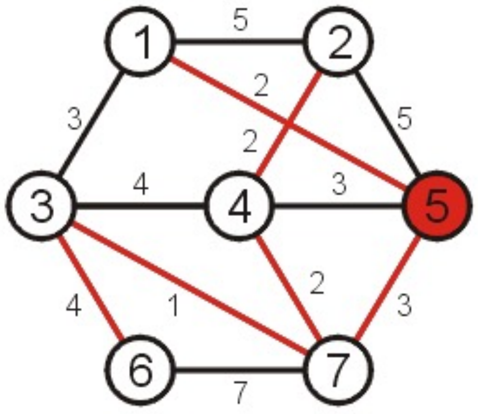
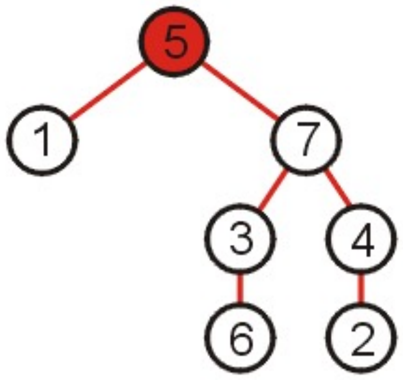
Minimum Spanning Trees

- Which spanning tree minimizes the weight?
  - Such a tree is termed a *minimum spanning tree*
- The weight of this spanning tree is 14



Minimum Spanning Trees

- 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

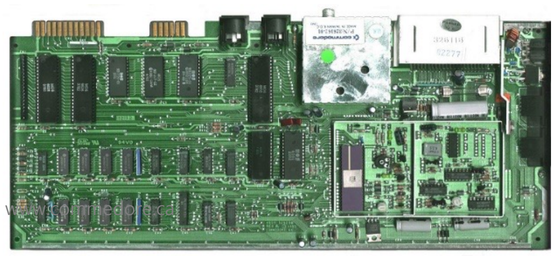
Unweighted graphs

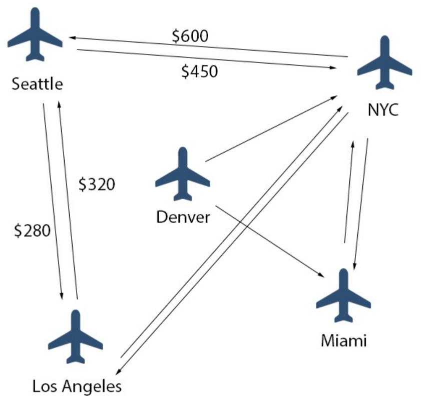
- Observation in unweighted graphs
  - In an unweighted graph, we give each edge a weight of 1
  - Consequently, all minimum spanning trees have weight $|V| - 1$

Application

- Supplying power to all circuit elements on a board
- Supplying power to all rooms in a building
- Flight costs with connection flights

- A minimum spanning tree will give the lowest-cost solution





http://apleroy.com/posts/using-google-maps-to-visually-display-a-minimum-spanning-tree-post-1-of-4

Algorithms

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

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Minimum_spanning_tree

Prim's Algorithm

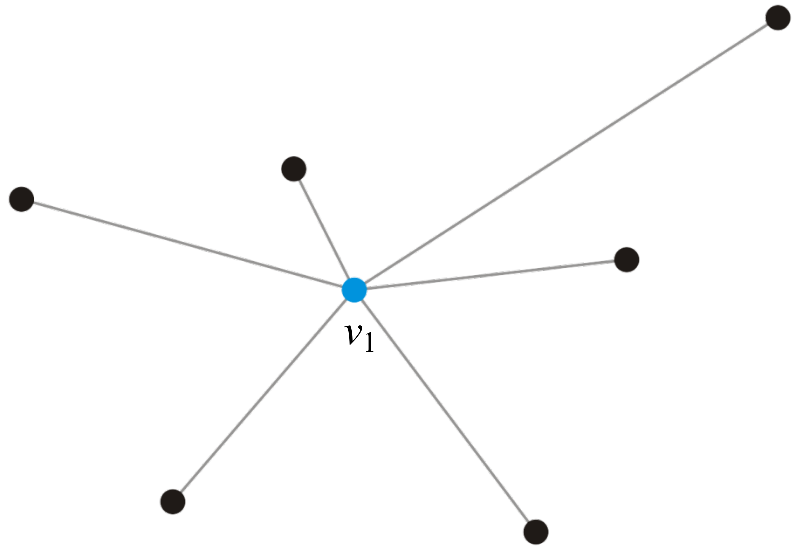
Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- This topic covers Prim's algorithm:
  - Finding a minimum spanning tree (MST)
  - The idea and the algorithm
  - An example

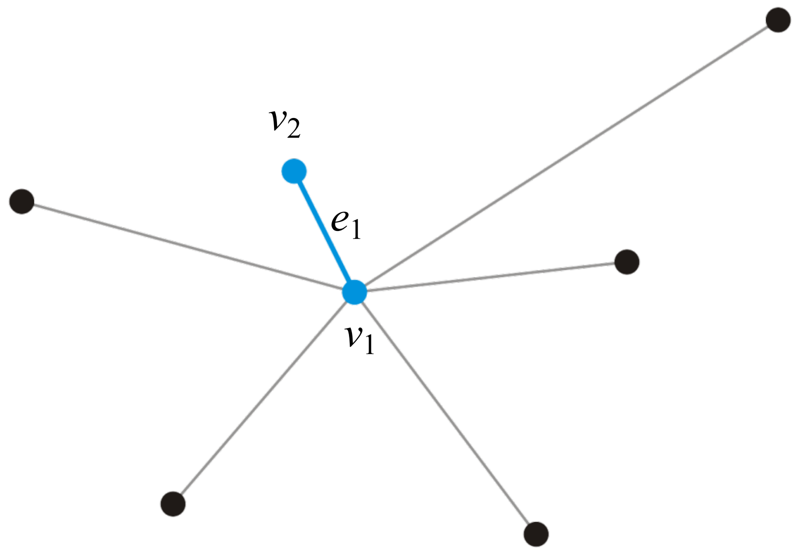
Strategy

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



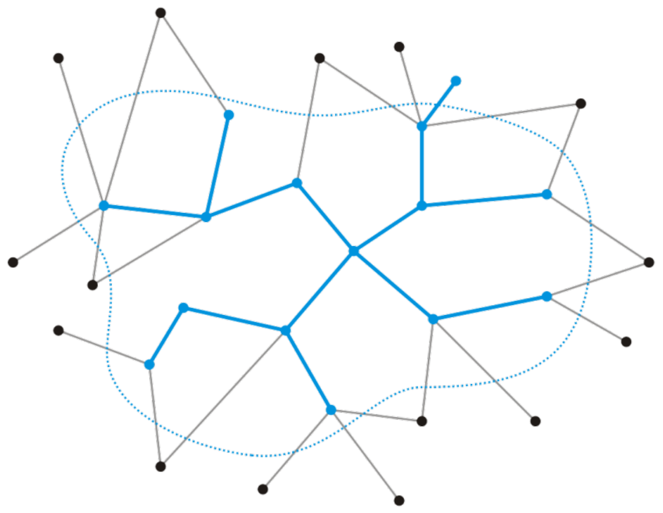
Strategy

- Add the adjacent vertex v_2 that has a connecting edge e_1 of minimum weight
  - This forms an MST on these two vertices



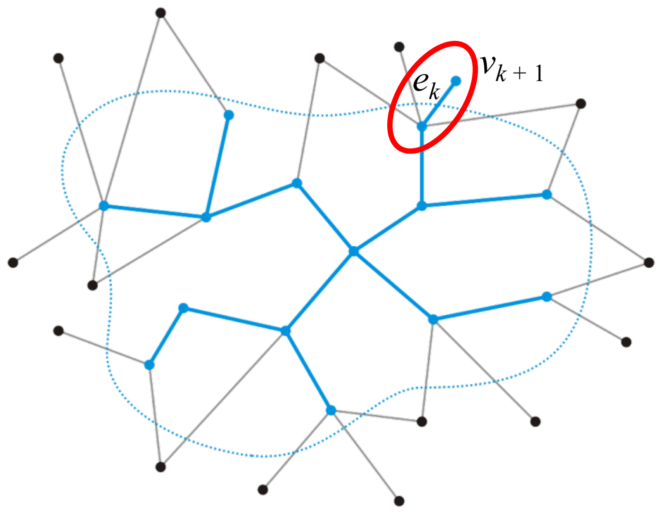
Strategy

- Strategy:
  - Suppose we have a known MST on $k < n$ vertices
  - How could we extend this MST?



Strategy

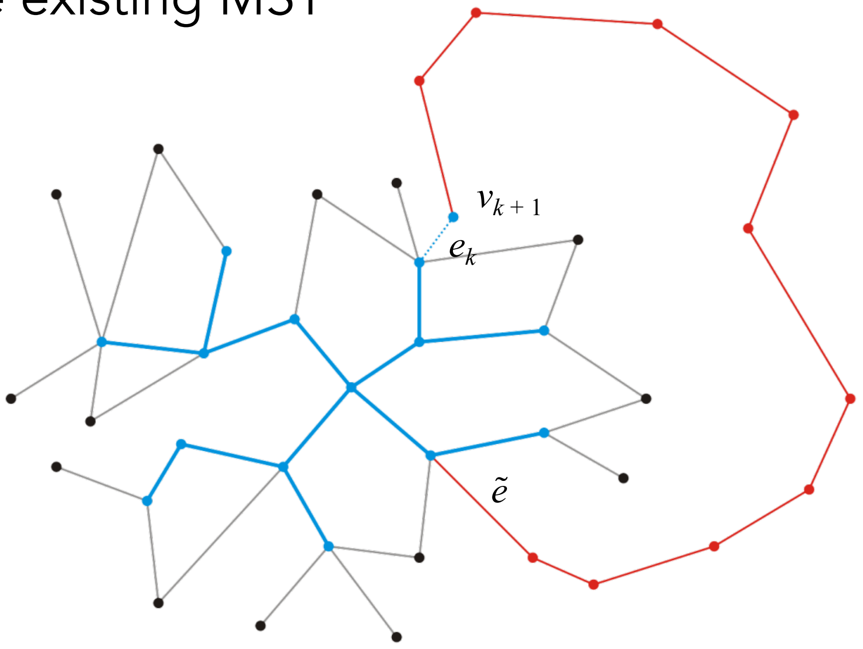
- Suppose you add e_k which has the minimum weight out of all edges connected to this MST
  - Adding e_k does create an MST with $k + 1$ nodes to connect v_{k+1}
    - There is no other edge we could add that would connect this vertex
  - However, can any edge other than e_k be used to connect v_{k+1} in an MST with n nodes later?



Proof by Contradiction

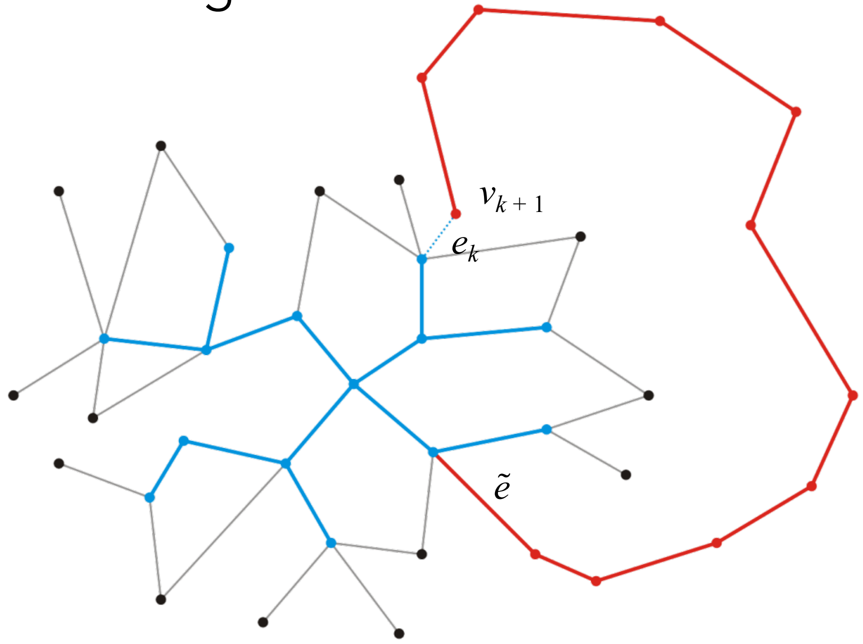
- **Proof by contradiction:**

  Suppose the previous claim is false.
  - Thus, vertex v_{k+1} is connected to the MST via another sequence of edges
  - Out of such sequence of edges, let's call $\tilde{e}$ as the edge out connecting to the existing MST



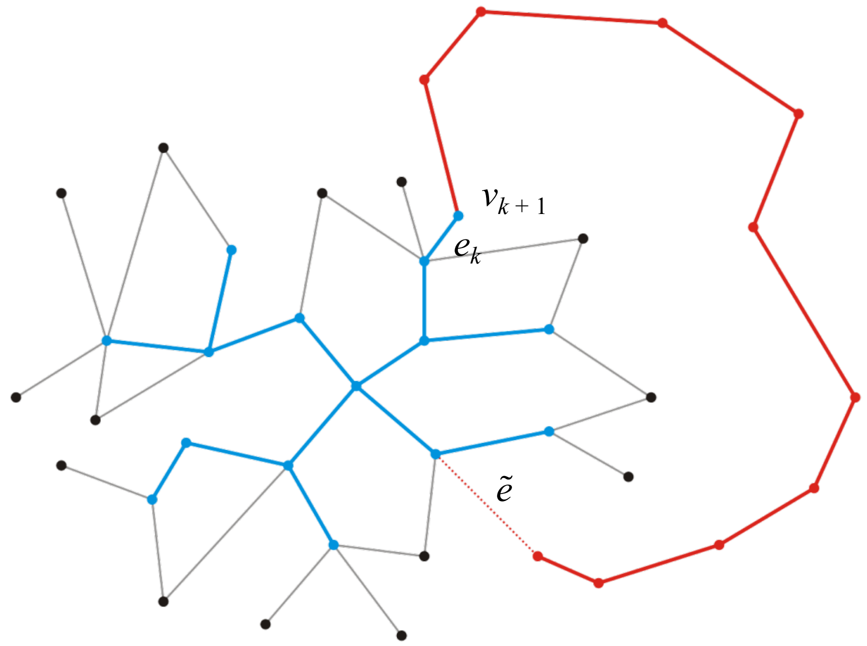
Proof by Contradiction

- Let w be the weight of this MST (with $\tilde{e}$)
  - Recall that we wanted to pick e_k because $|\tilde{e}| > |e_k|$
    - |e| denotes the weight of the edge e
- Suppose we add e_k and exclude $\tilde{e}$ to the MST
  - The result is still a spanning tree, but the weight is now
    - $w + |e_k| - |\tilde{e}| \le w$
  - This contradicts our assumption that the MST with $\tilde{e}$ is minimal
  - Therefore, our MST must contain e_k



Proof by Contradiction: Strategy

- Recall that we did not prescribe the value of *k*, and thus, *k* could be any value, including *k* = 1



Prim’s Algorithm

- Prim’s algorithm for finding the MST states:
  - Start with an arbitrary vertex to form an MST on one vertex
  - At each step, add the vertex *v* not yet in the MST
    - Vertex *v* is connected with an edge with least weight to the existing minimum spanning sub-tree
  - Continue until we have *n* – 1 edges and *n* vertices

- Note: Prim’s algorithm is a greedy algorithm
  - ➔ A greedy algorithm does not always yield the optimal solution
  - ➔ In the case of Prim’s algorithm, however, it does yield the optimal solution

Prim’s Algorithm: Data Structures

- Associate each vertex with:
  - A Boolean flag indicating if the vertex has been visited,
  - The minimum distance to the partially constructed MST, and
  - A pointer to that vertex which will form the parent node in the resulting tree

Prim's Algorithm: Initialization

- 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

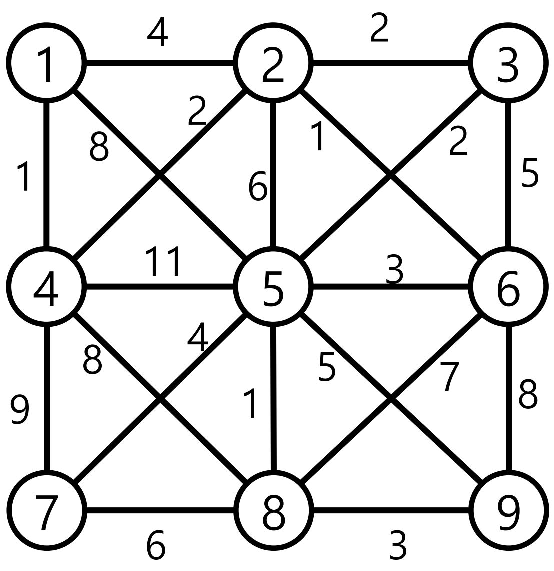
- Iterate while there exists an unvisited vertex with distance < ∞
  - Select that unvisited vertex with minimum distance
  - Mark that vertex as having been visited
  - For each adjacent vertex, if the weight of the connecting edge is less than the current distance to that vertex:
    - Update the distance to be the weight of the connecting 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 $< \infty$
- If all vertices have been visited, we have a spanning tree of the entire graph
- If at any point, all remaining vertices had a distance of ∞, this indicates that the graph is not connected ➔ No MST

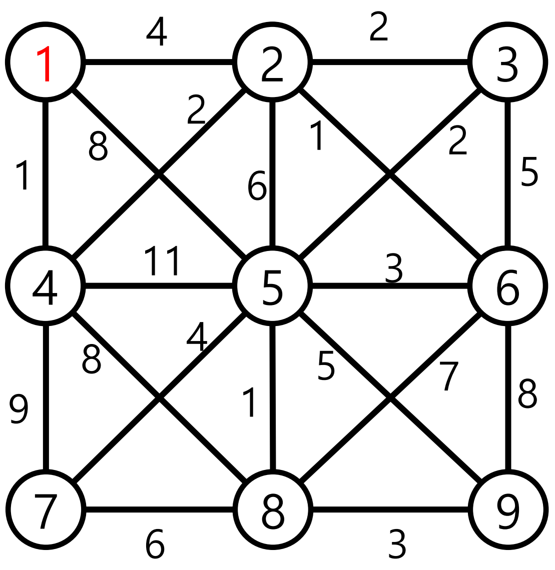
Prim's Algorithm

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



Prim's Algorithm

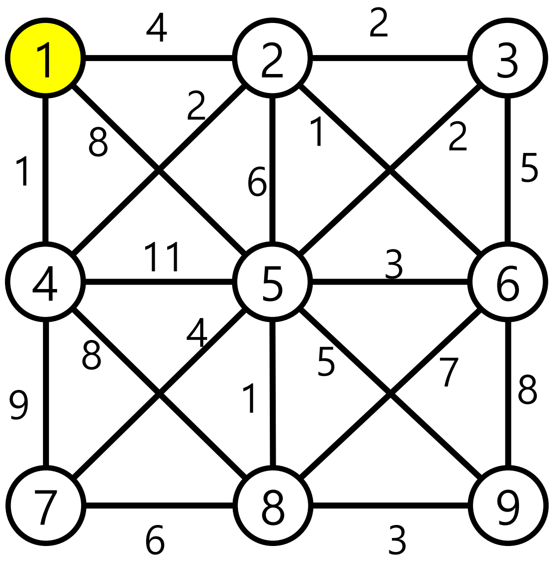
- First, we set up the appropriate table and initialize it
  - Suppose the root is the vertex 1



		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

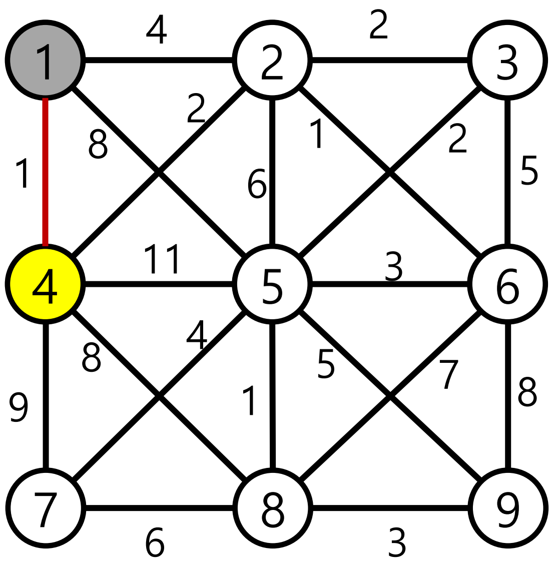
- Visit vertex 1
  - We update vertices 2, 4, and 5
  - MST: {1}



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

Prim’s Algorithm

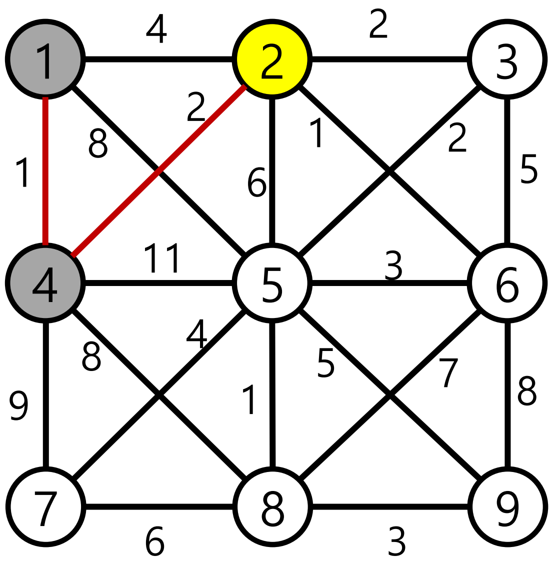
- Visit vertex 4, because vertex 4 has the minimum distance (among unvisited vertices)
  - Update vertices 2, 7, 8
  - Don’t update vertex 5
  - MST: {1,4}



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

Prim's Algorithm

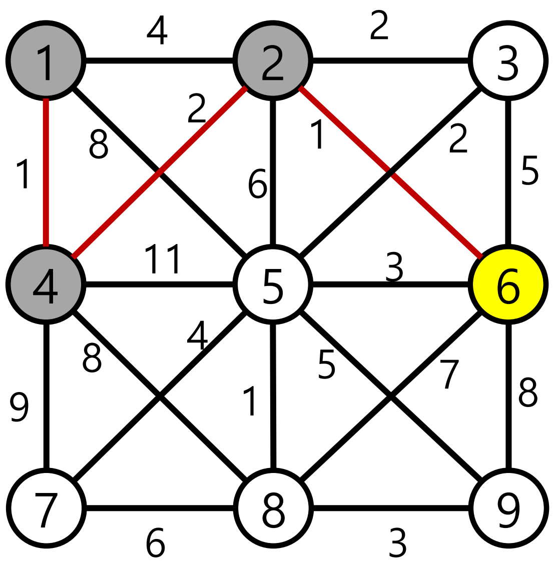
- Visit vertex 2
  - Update 3, 5, and 6
  - MST: {1, 4, 2}



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

Prim’s Algorithm

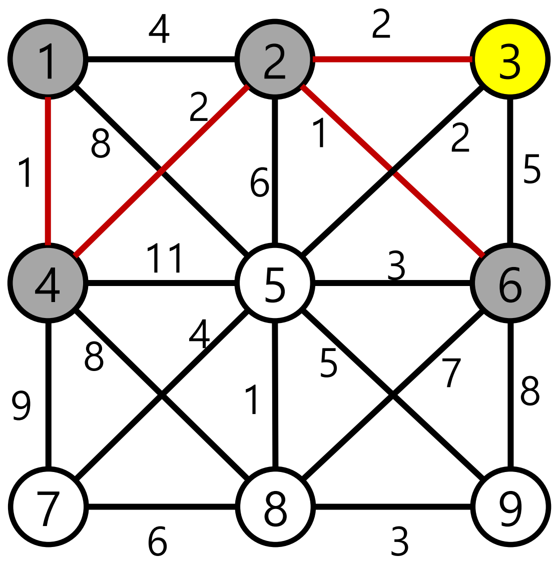
- Next, we visit vertex 6:
  - update vertices 5, 8, and 9
  - MST: {1, 4, 2, 6}



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

Prim's Algorithm

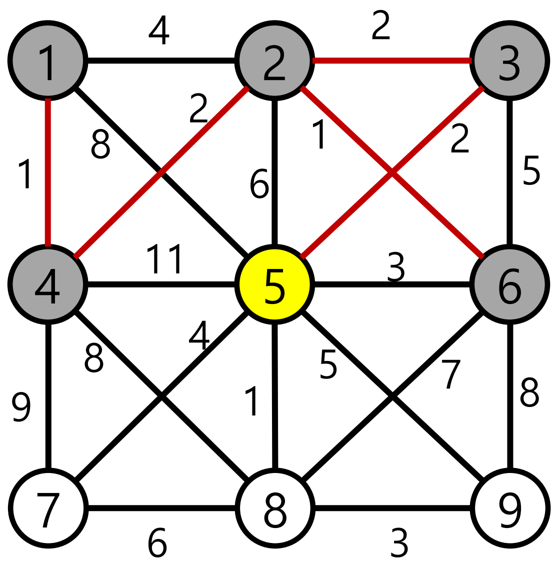
- Visit vertex 3, and update vertex 5
  - MST: {1, 4, 2, 6, 3}



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

Prim's Algorithm

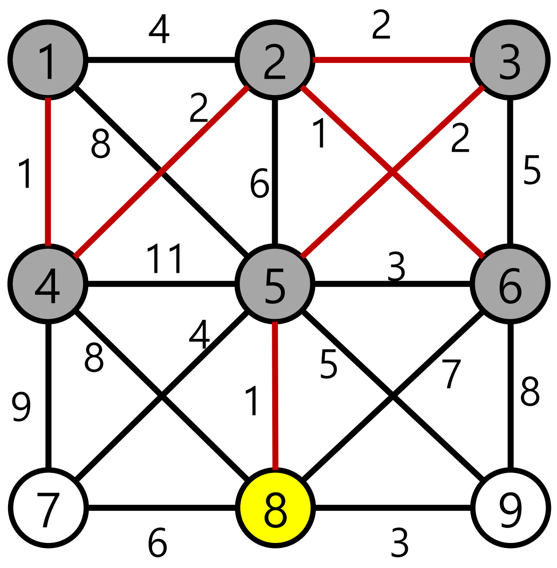
- Visit vertex 5
  - No need to update other vertices
  - MST: {1, 4, 2, 6, 3, 5}



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

Prim's Algorithm

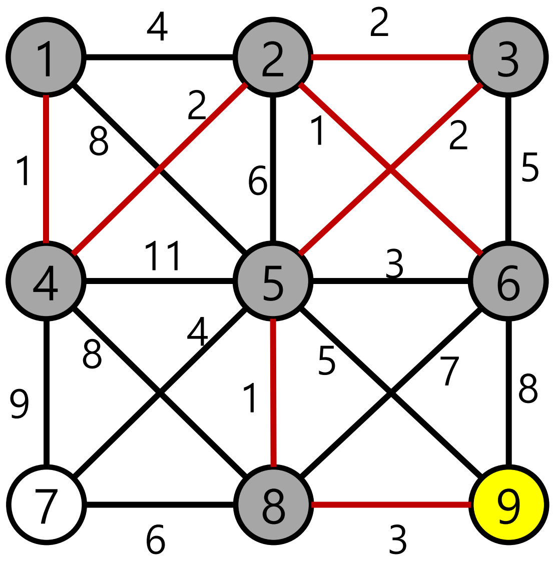
- Visiting vertex 8, we only update vertex 9
  - MST: {1, 4, 2, 6, 3, 5, 8}



		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→T	1	5
9	F	5→3	5→8

Prim’s Algorithm

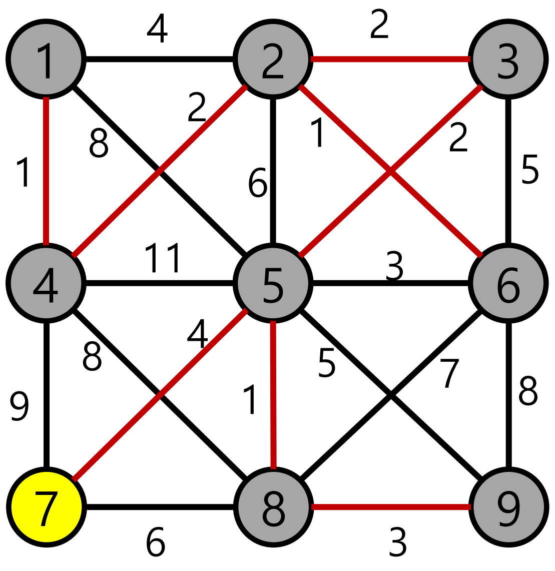
- Visit vertex 9. No need to update other vertices.
  - MST: {1, 4, 2, 6, 3, 5, 8, 9}



		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	T	1	5
9	F➔T	3	8

Prim’s Algorithm

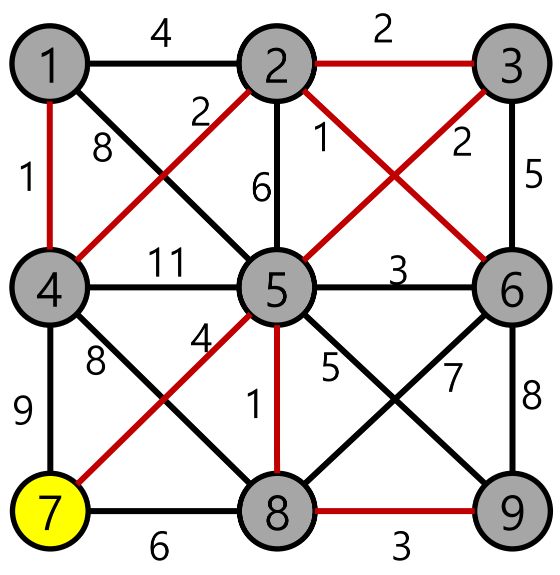
- Visit vertex 7, then done.
  - MST: {1, 4, 2, 6, 3, 5, 8, 9, 7}



		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	F➔T	4	5
8	T	1	5
9	T	3	8

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

Implementation and analysis

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

- Iteration: We iterate $|V| - 1$ times, each time finding the *min. distance* vertex
  - Iterating through the table (to find the min. distance vertex) requires is $\Theta(|V|)$ time
  - Each time we find a min. distance vertex, we must check all of its neighbors (to update distance)

```
for _ in range(|V|): // until visiting all vertices
    // visit the min distance vertex
    v = find_min_dist_vertex(table)
        // update the distance if needed
    for j in get_adj_vertices(v):
        if (graph.get_dist(v, j) < table.get_dist(j))
            table.set_dist(j) = graph.get_dist(v, j)
```

- With an adjacency matrix, the run time is $O(|V|(|V| + |V|)) = O(|V|^2)$
  - Each call of `find_min_dist_vertex()` takes $O(|V|)$
  - Enumerating adj vertices for each vertex take $O(|V|)$

- With an adjacency list, the run time is $O(|V|^2 + |E|) = O(|V|^2)$ as $|E| = O(|V|^2)$
  - Each call of `find_min_dist_vertex()` takes $O(|V|)$
  - Enumerating all adj vertices in the end would take $O(|E|)$

Implementation and analysis

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

Implementation and analysis: Binary Heap

```
    for _ in range(|V|): // until visiting all vertices
        // visit the min distance vertex
        v = find_min_dist_vertex(table)
            // update the distance if needed
        for j in get_adj_vertices(v):
            if (graph.get_dist(v, j) < table.get_dist(j))
                table.set_dist(j) = graph.get_dist(v, j)
```

- The table is maintained with a min heap, where the key is a min. distance associated with vertex
  - `find_min_dist_vertex()` takes $\ln(|V|)$, which is executed $|V|$ times
  - `table.set_dist()` takes $\ln(|V|)$, which is executed $|E|$ times

- Thus, the total run time with binary heap is
  - $O(|V| \ln(|V|) + |E| \ln(|V|)) = O(|E| \ln(|V|))$

Summary

- We have seen an algorithm for finding minimum spanning trees
  - Start with a trivial minimum spanning tree and grow it
  - An alternate algorithm, Kruskal's algorithm, uses a different approach

- Prim's algorithm finds an edge with least weight which grows an already existing tree
  - It solves the problem in O(|*E*| ln(|*V*|)) time

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Minimum_spanning_tree
[2] Wikipedia, http://en.wikipedia.org/wiki/Prim's_algorithm

Kruskal's Algorithm

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- This topic covers Kruskal's algorithm:
  - Finding a minimum spanning tree
  - The idea and the algorithm
  - An example
  - Using a disjoint set data structure

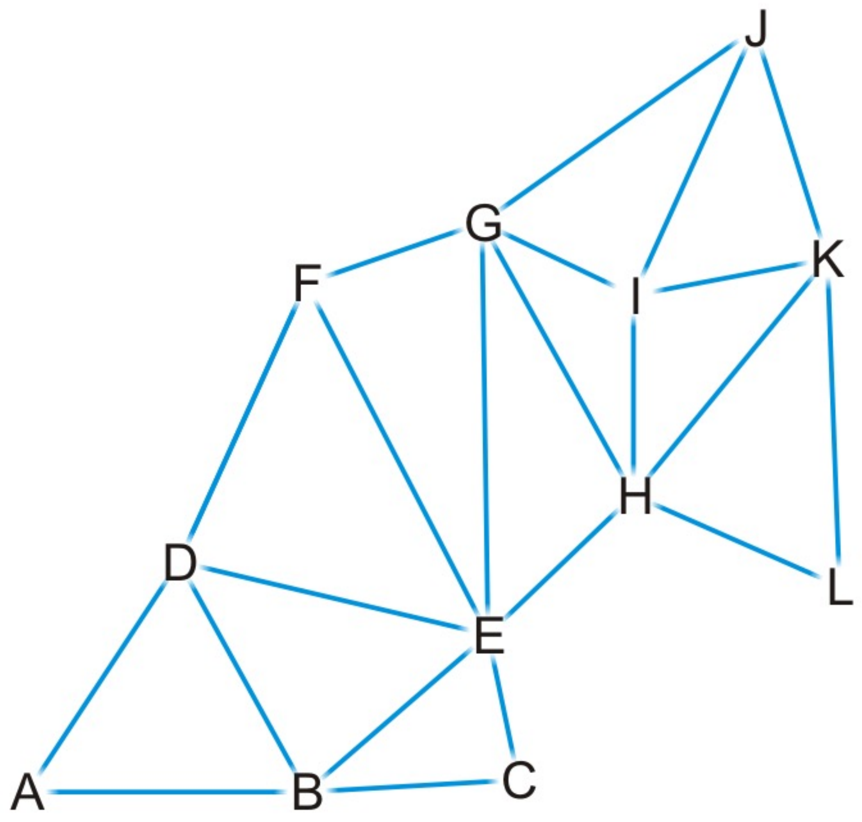
Kruskal’s Algorithm

- 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

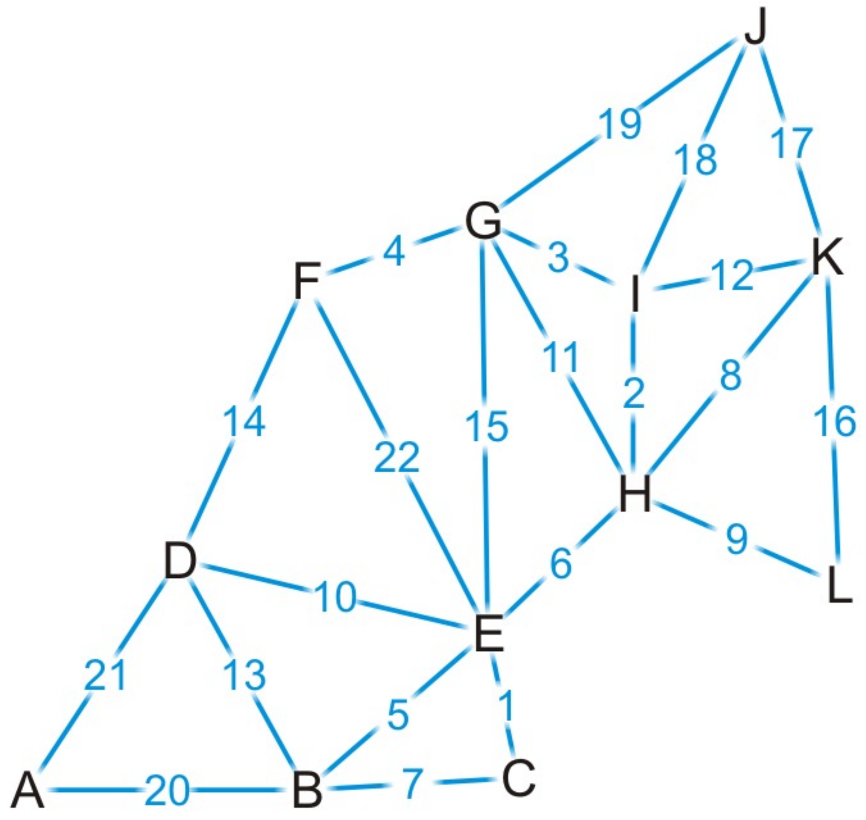
Example

- Here is our abstract representation with 12 nodes



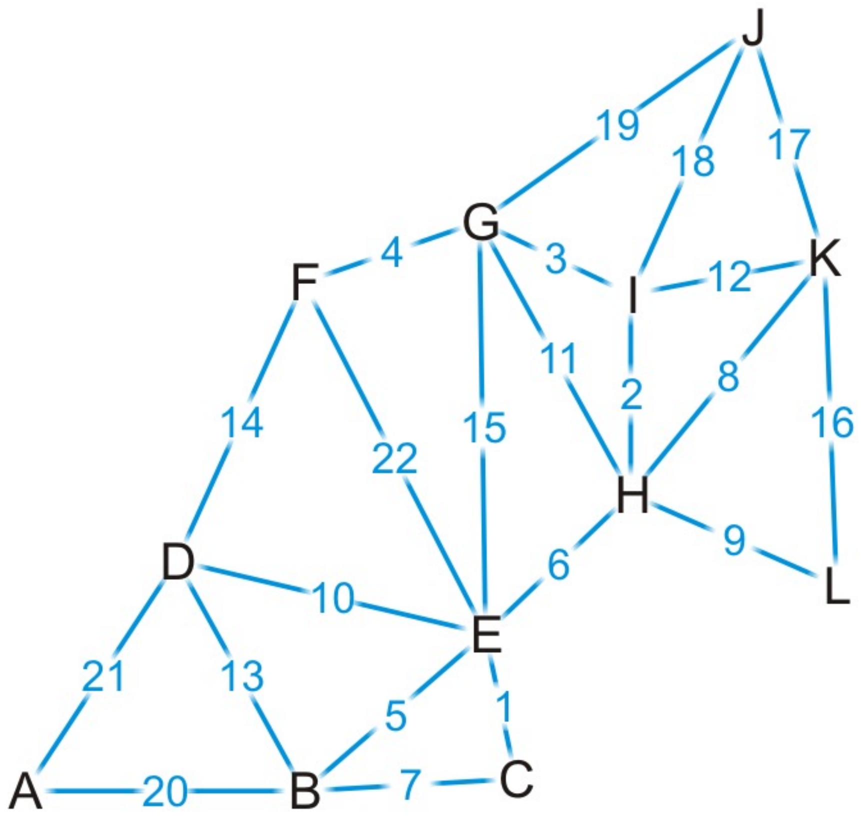
Example

- Let us give a weight to each of the edges



Example

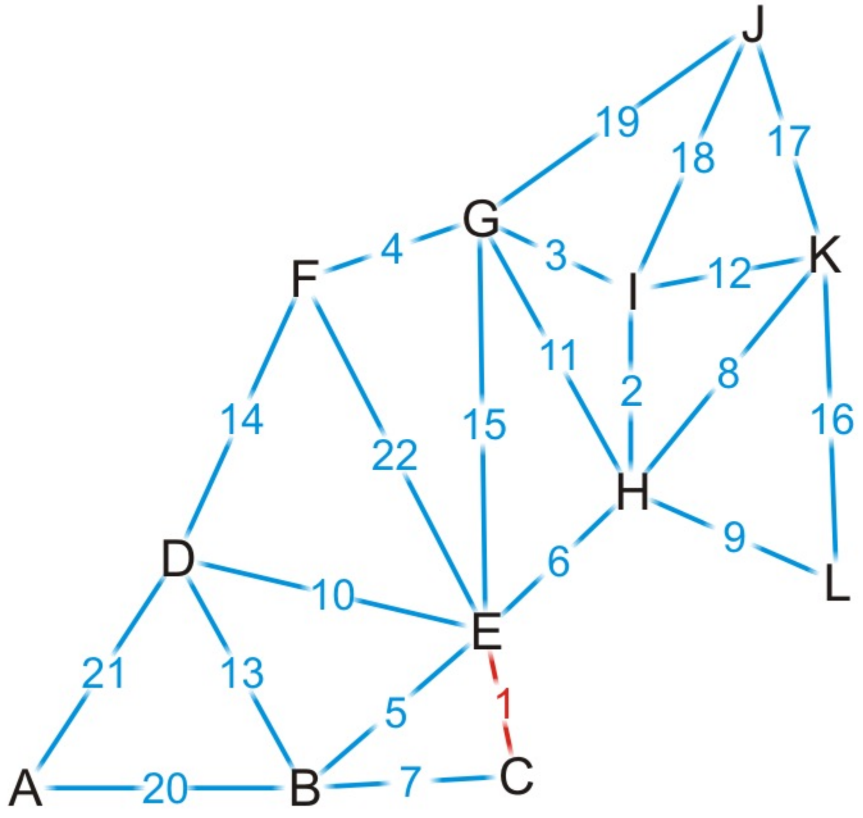
- First, we sort the edges based on weight



{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

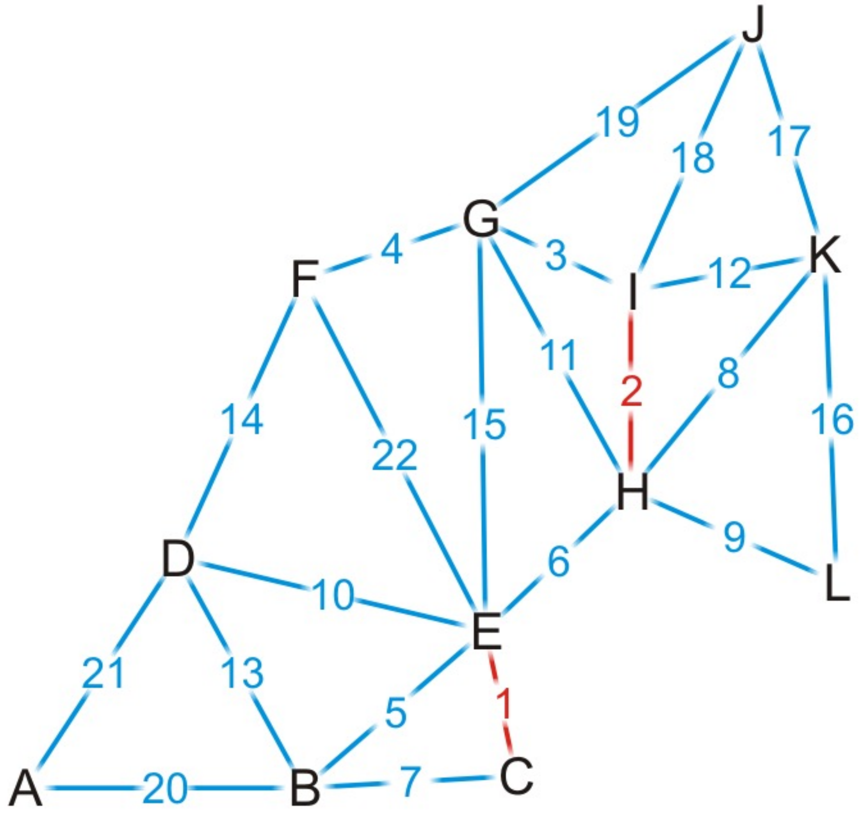
- We start by adding edge {C, E}



→ {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

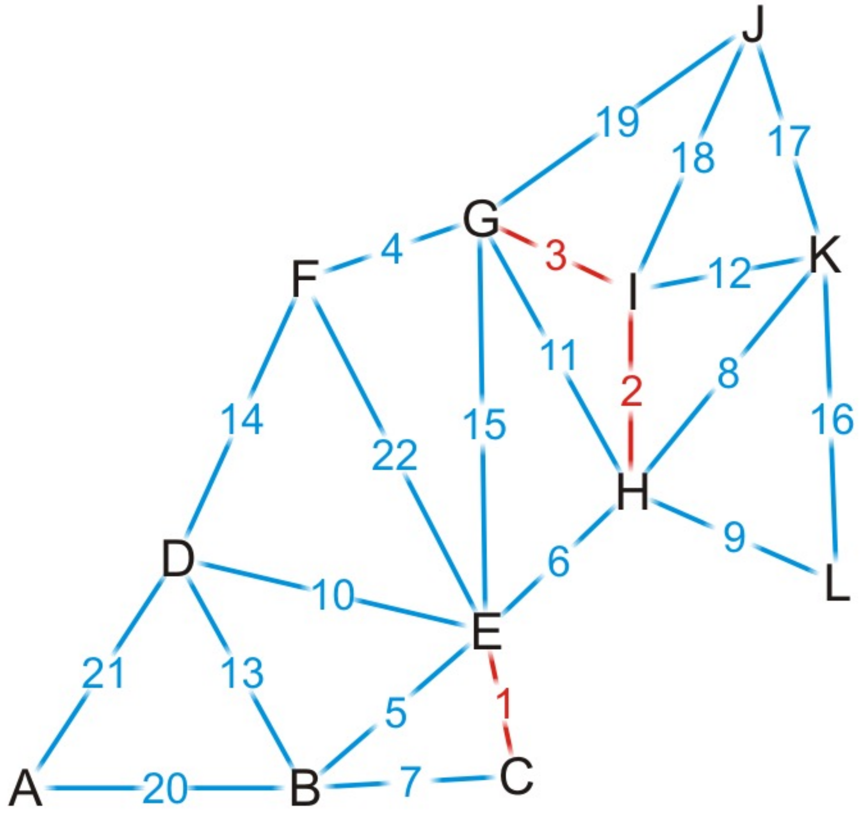
- We add edge {H, I}



{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

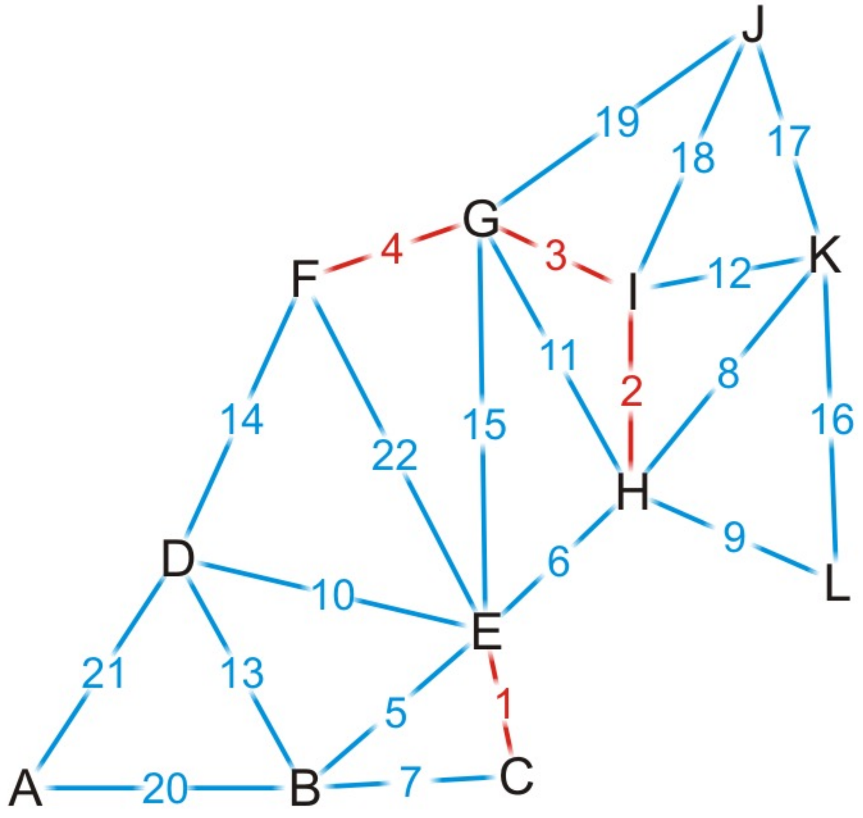
- We add edge {G, I}



{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

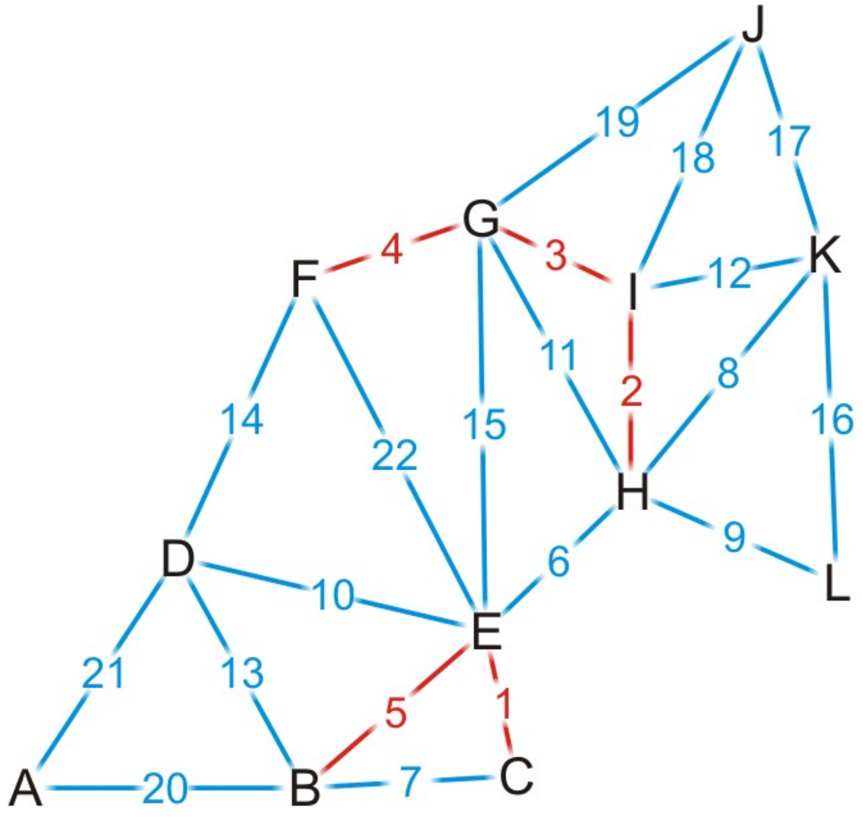
- We add edge {F, G}



{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

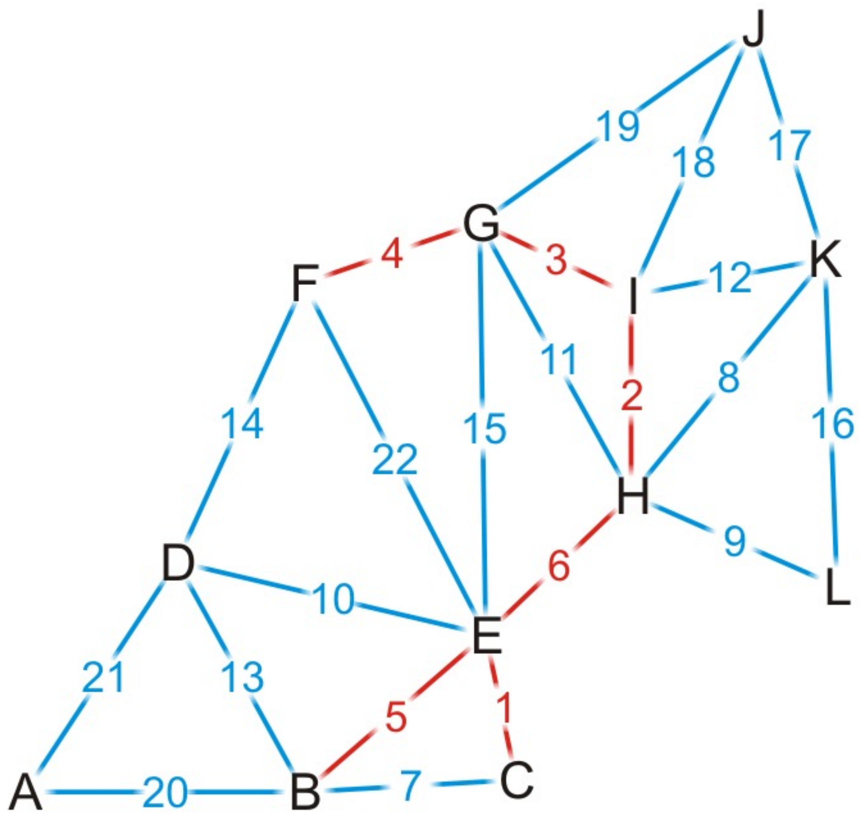
- We add edge {B, E}



{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

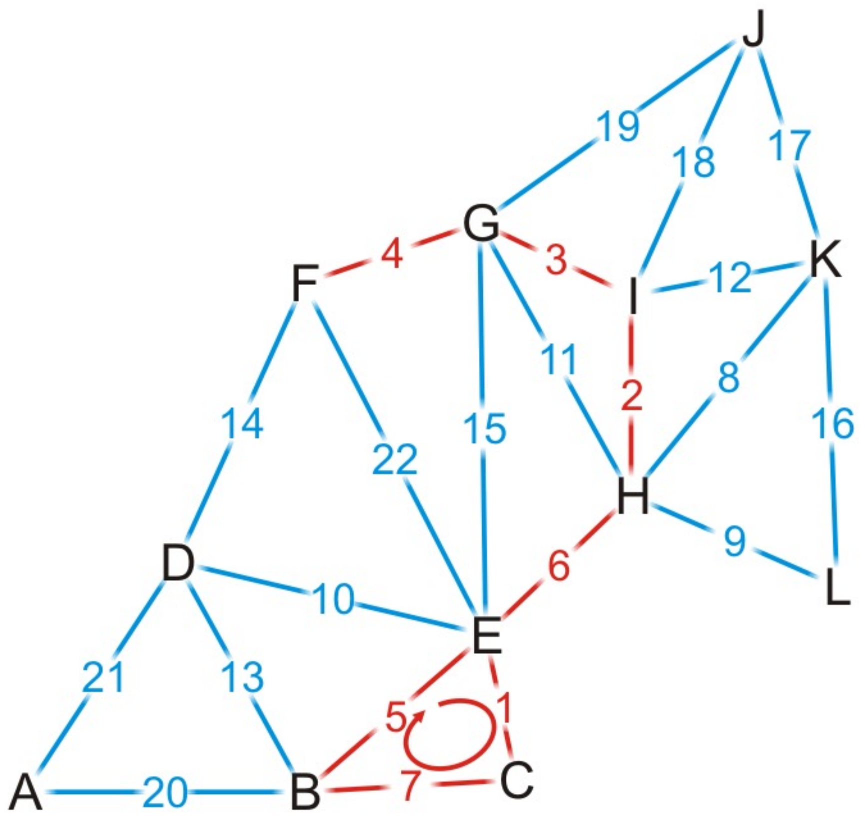
- We add edge {E, H}
  - This coalesces the two spanning sub-trees into one



{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

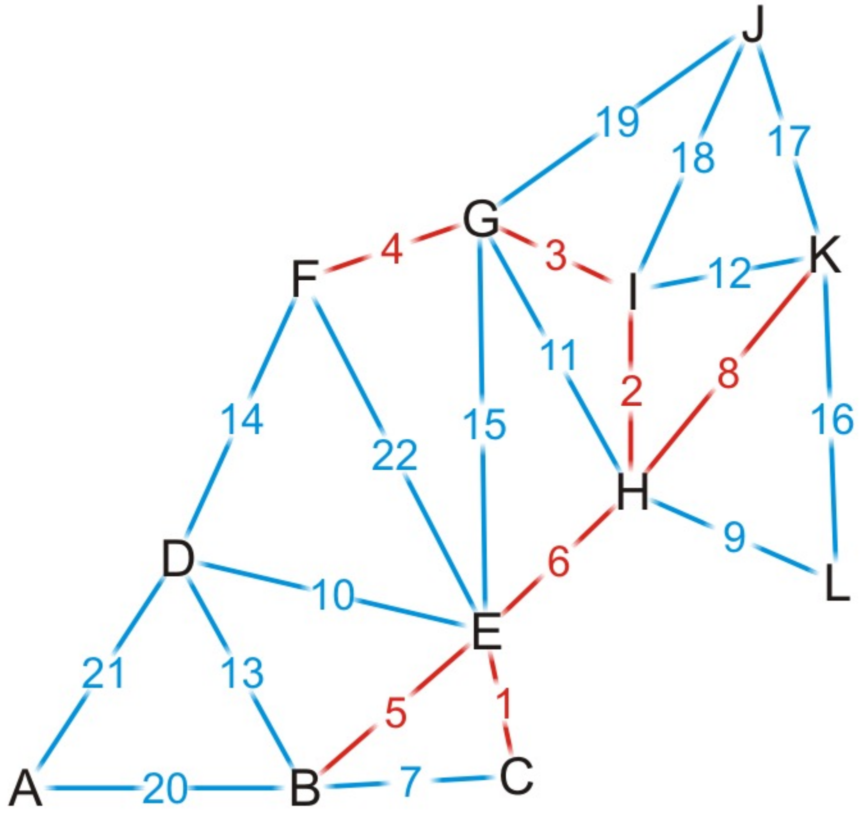
- We try adding {B, C}, but it creates a cycle



{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

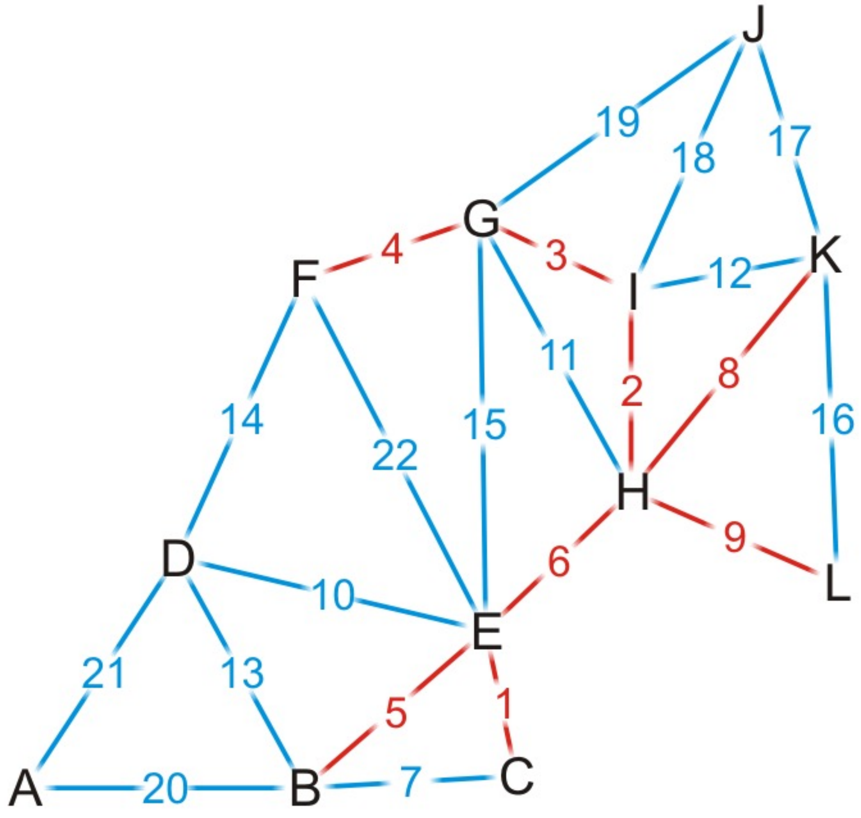
- We add edge {H, K}



{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

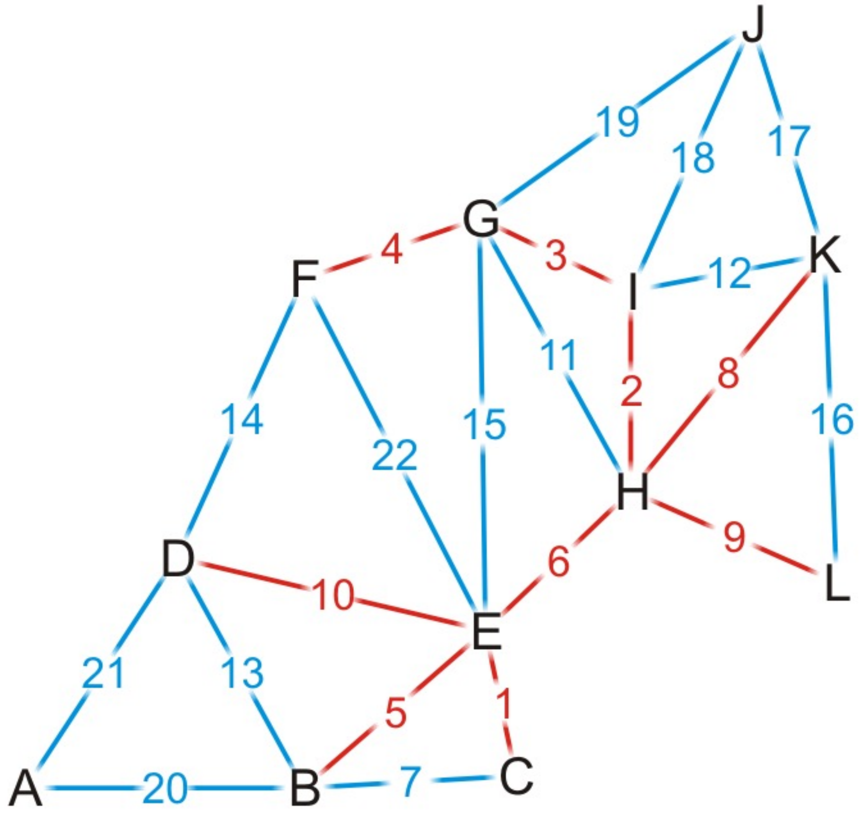
- We add edge {H, L}



{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

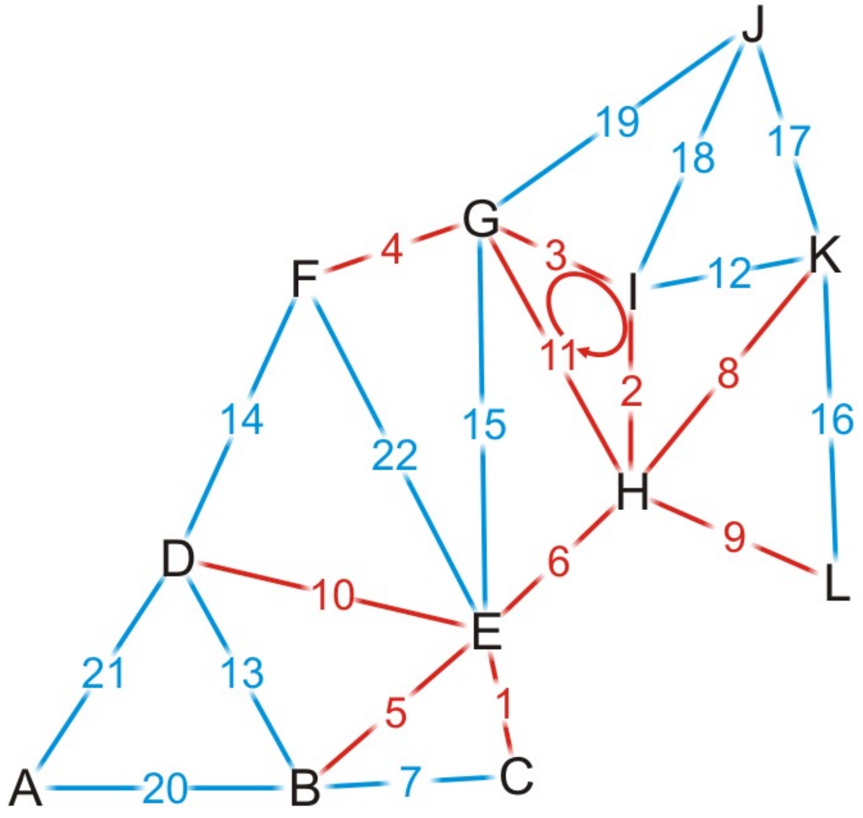
- We add edge {D, E}



{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

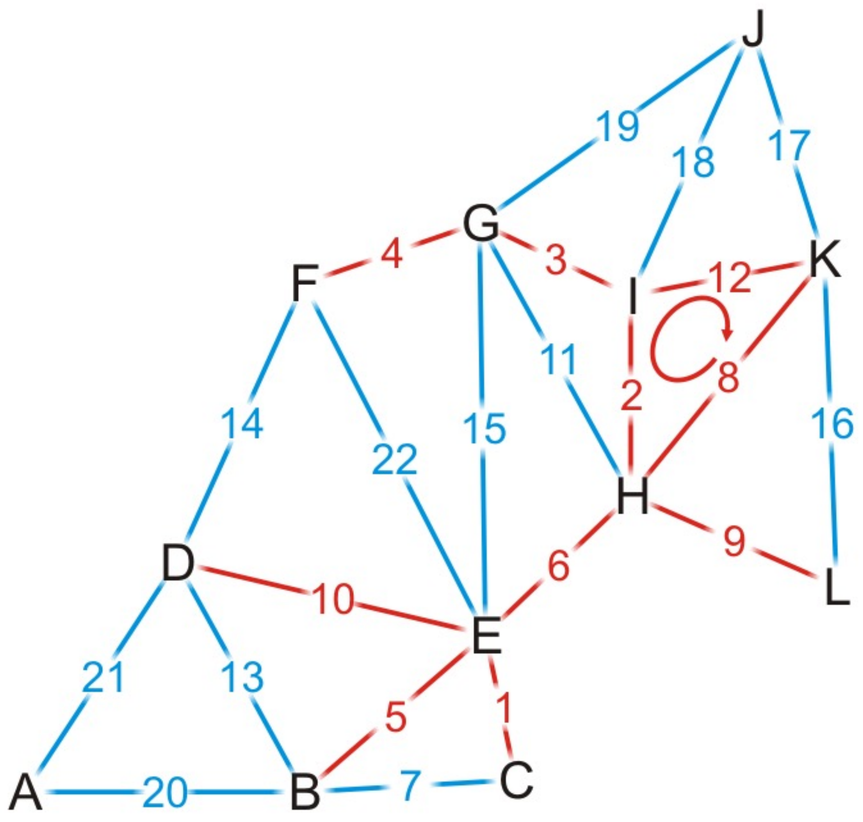
- We try adding {G, H}, but it creates a cycle



{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

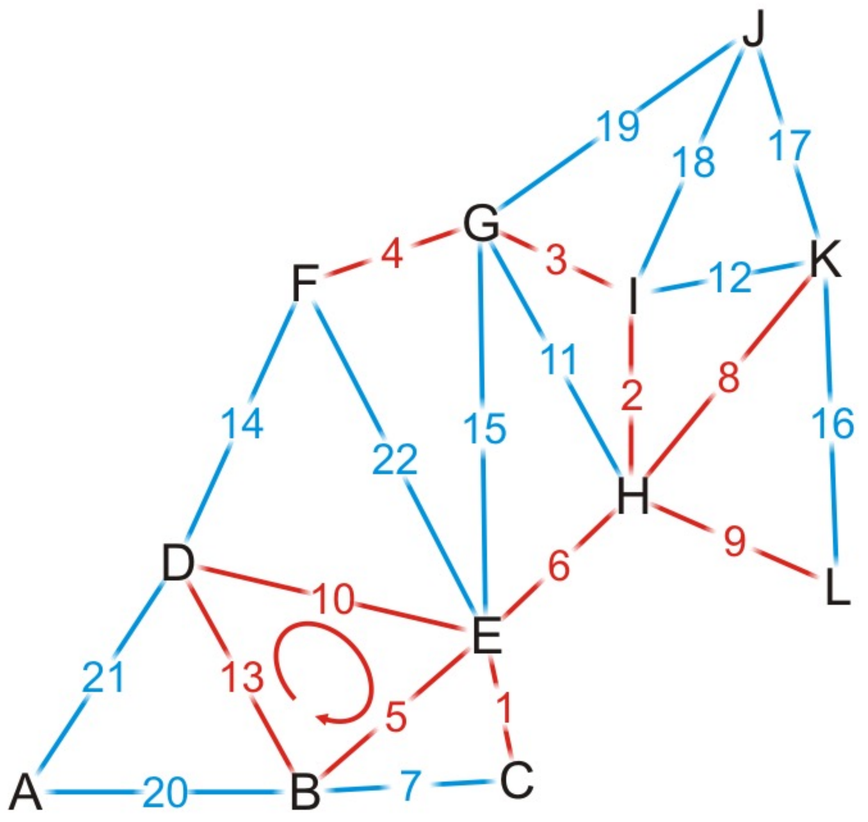
- We try adding {I, K}, but it creates a cycle



{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

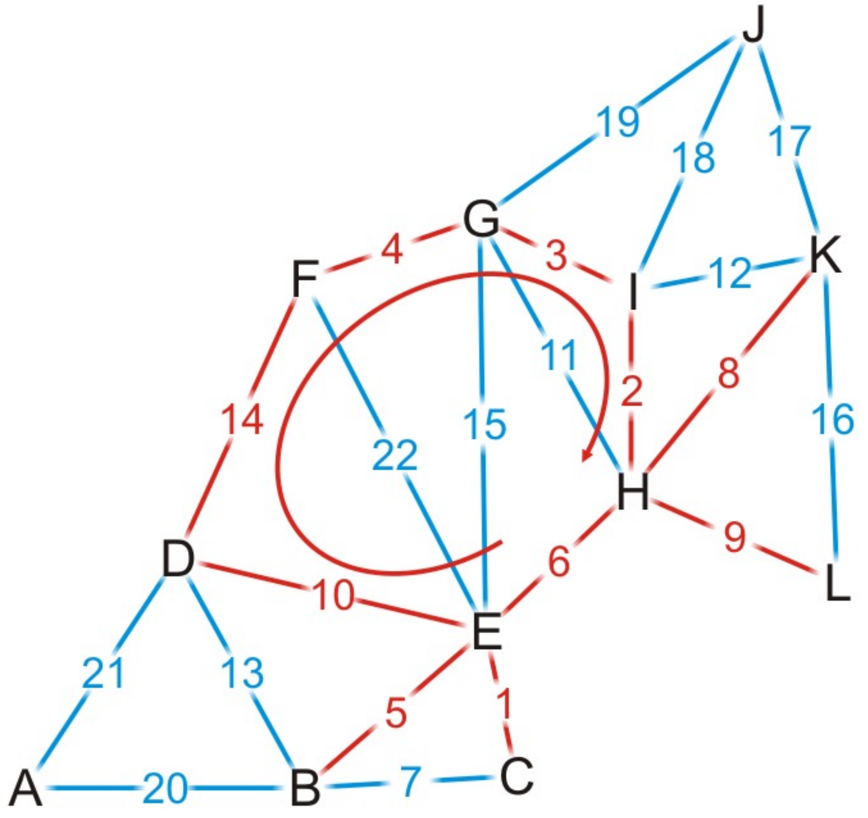
- We try adding {B, D}, but it creates a cycle



{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

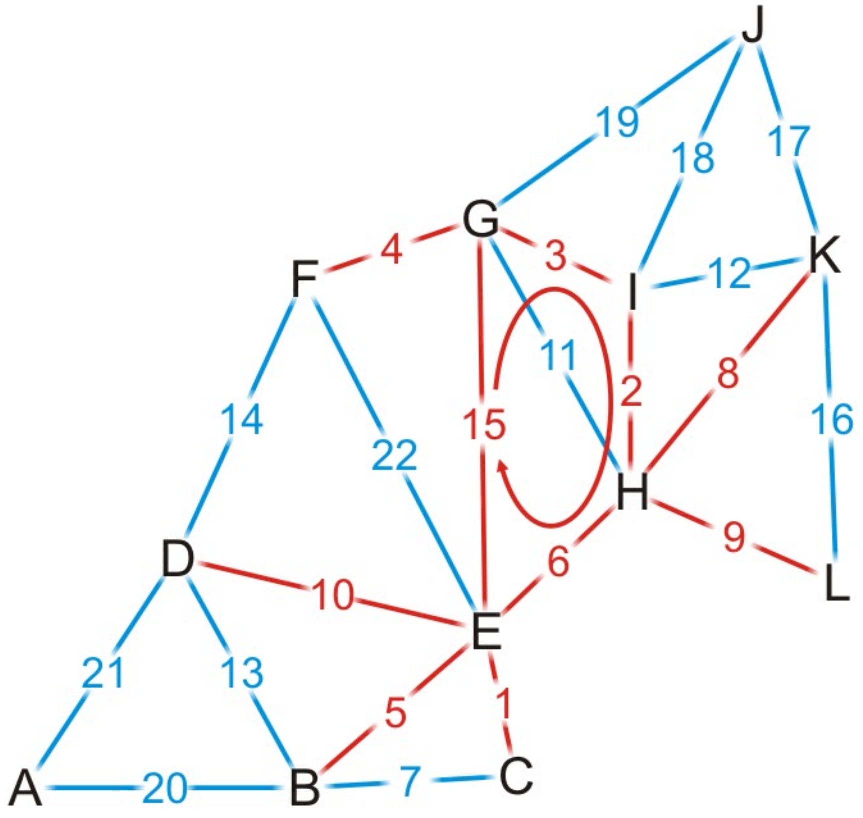
- We try adding {D, F}, but it creates a cycle



{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

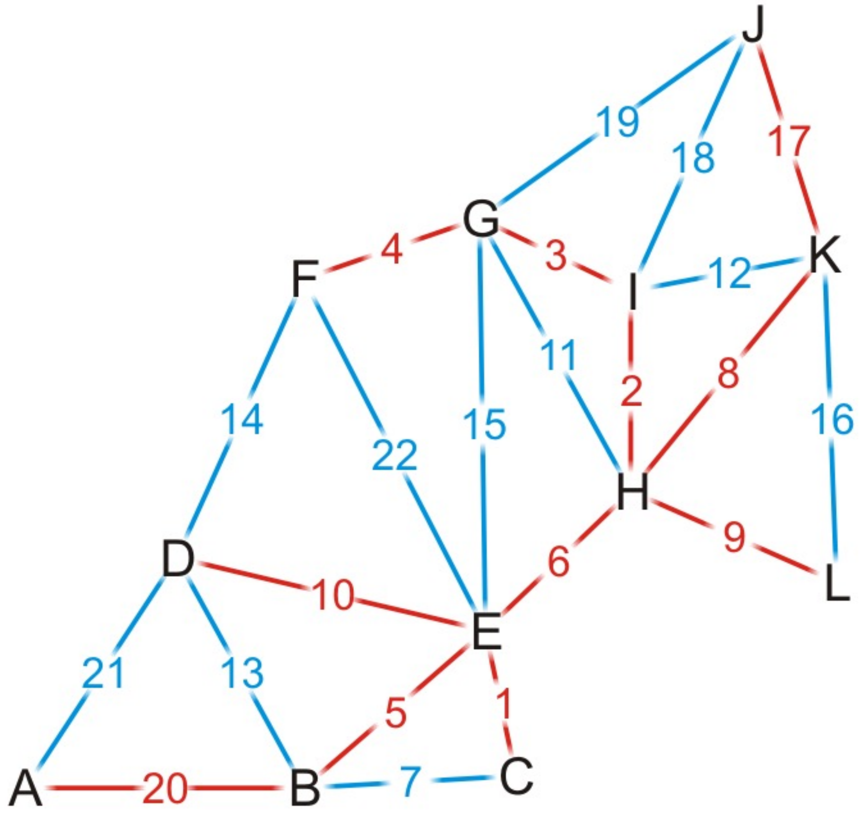
- We try adding {E, G}, but it creates a cycle



{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

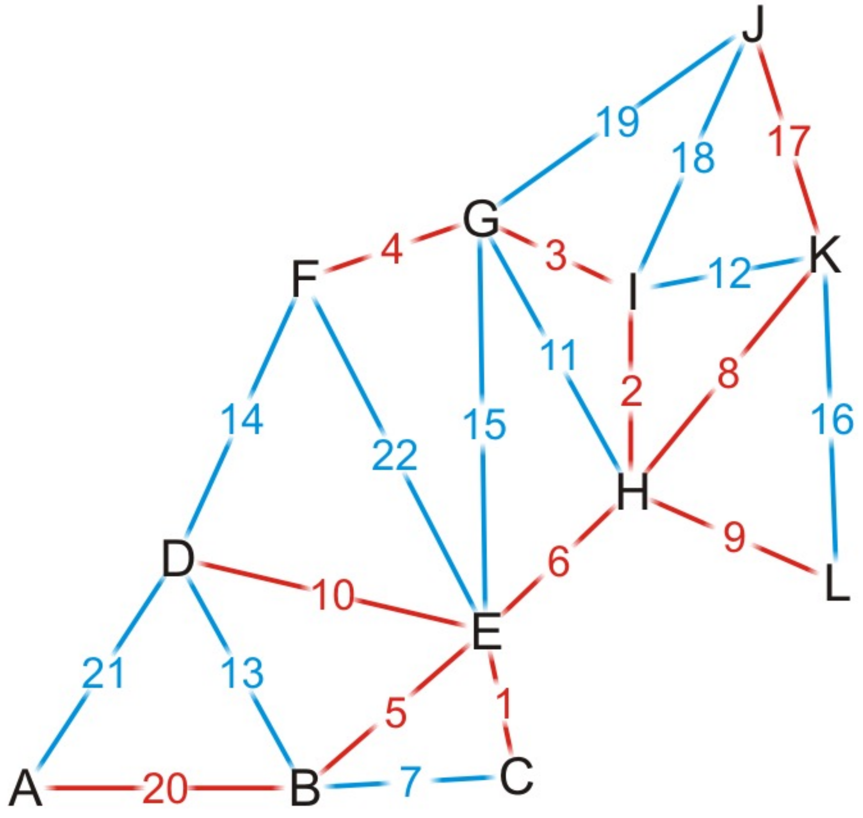
- By observation, we can still add edges {J, K} and {A, B}



{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}

Kruskal’s Algorithm (baseline)

```
1    (Sort the edges in an increasing order)
2    A:={}
3    while E is not empty do {
3        take an edge (u, v) that is shortest in E
         and delete it from E
4        if  u and v are in different components then
                 add (u, v) to A
         }
```

Note: each time a shortest edge in E is considered.

Analysis

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

Kruskal’s Algorithm (efficient)

```
A:={}
for each vertex v in V                                  } O(|V|)
        do MAKE_SET(v)                                  }
sort the edges of E by nondecreasing weight w           } O(|E| ln|E|)
for each edge (u,v) in E, in order by nondecreasing weight   ← O(|E|)
        do if FIND_SET(u) != FIND_SET(v)                } O(ln|V|)
                then    A:= A ∪ {(u,v)}                 }
                        UNION(u,v)                      }
return A
```

- Consequently, the run-time would be $O(|V| + |E| \ln(|E|) + |E| \ln|V|) = O(|E| \cdot \ln|V|)$

Disjoint Sets Data Structure (Chapter 8)

- A disjoint-set is a collection of distinct dynamic sets: $\{S_1, S_2, \ldots, S_k\}$
- Each set is identified by a member of the set, called *representative (e.g., root of the set formed as a tree)*
- Disjoint set operations:
  - MAKE-SET(*x*): create a new set with an only member *x*. (assume *x* is not already in some other set.)
  - UNION(*x*,*y*): combine the two sets containing *x* and *y* into one new set. A new representative is selected.
  - FIND-SET(*x*): return the representative of the set containing *x*.

Summary

- This topic has covered Kruskal's algorithm
  - Sort the edges by weight
  - Create a disjoint set of the vertices
  - Begin adding the edges one-by-one checking to ensure no cycles are introduced
  - The result is a minimum spanning tree
  - The run time is O(|*E*| ln(|*V*|)

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Minimum_spanning_tree
[2] Wikipedia, http://en.wikipedia.org/wiki/Kruskal's_algorithm