Graph Traversal (cont)

COMP9312_24T2

UNSW
SYDNEY

Minimum spanning tree

Minimum Spanning Tree

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
- Applications
- Solutions

Given 5 vertices, how many undirected edges are needed at least to connect these vertices together?

Given n vertices, how many undirected edges are needed at least to connect these vertices together?

Minimum Spanning Tree

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
- Applications
- Solutions

Given 5 vertices, how many undirected edges are needed at least to connect these vertices together? 4

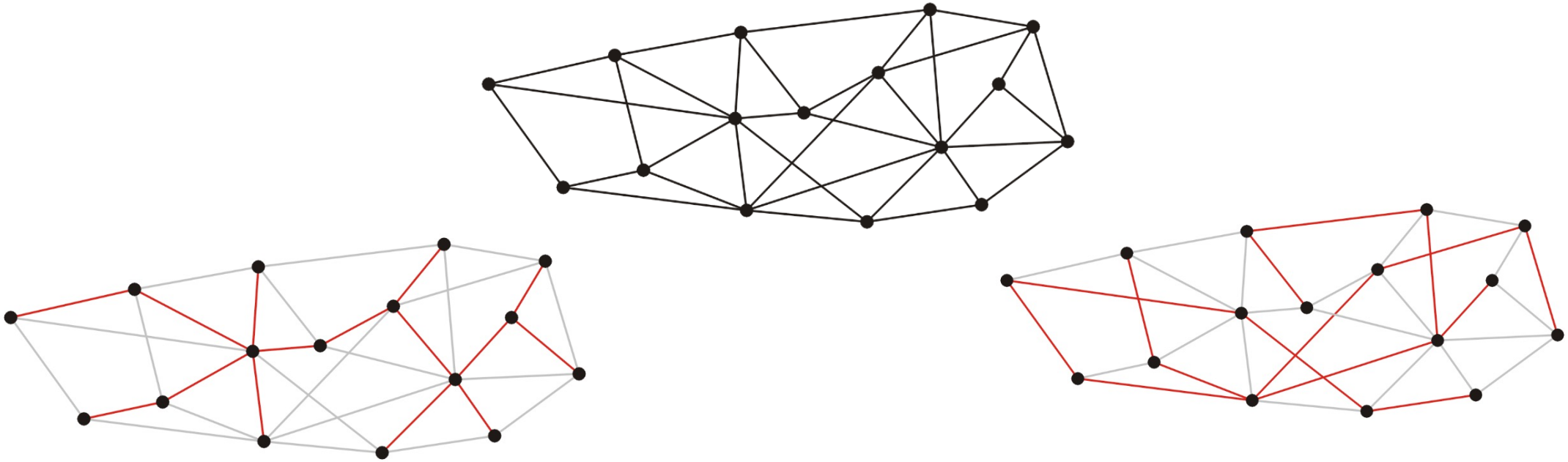
Given n vertices, how many undirected edges are needed at least to connect these vertices together? n-1

Spanning trees

Given a connected graph with n vertices, a spanning tree is defined a collection of $n-1$ edges which connect all n vertices.

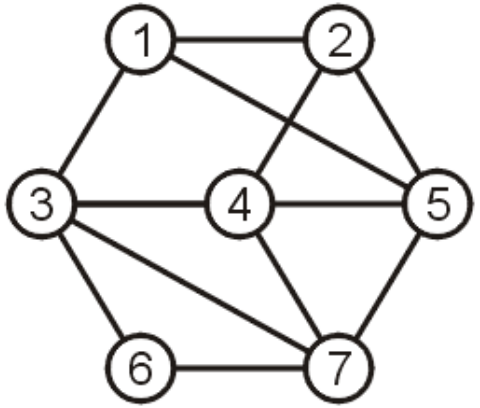
- The n vertices and $n-1$ edges define a connected sub-graph.

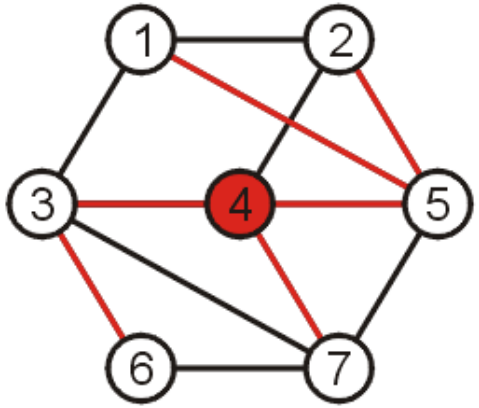
A spanning tree is not necessarily unique.

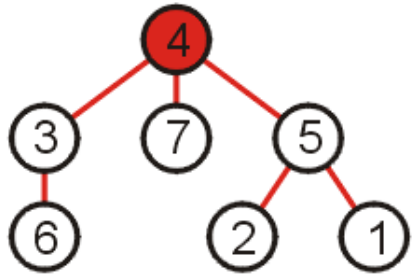


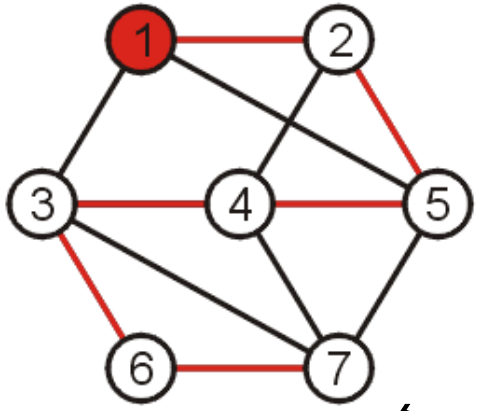
Spanning trees

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.



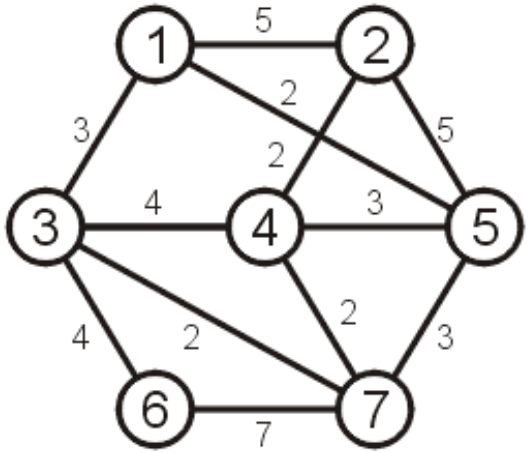


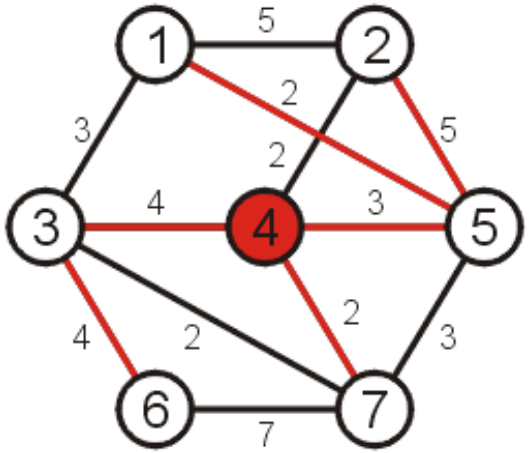




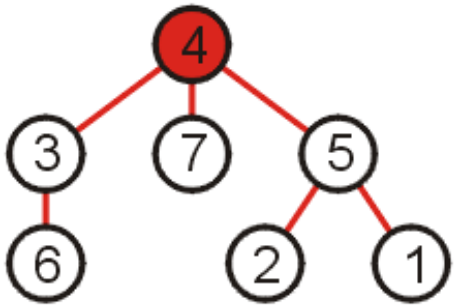
Spanning trees on weighted graphs

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





The weight of the above spanning tree is 20

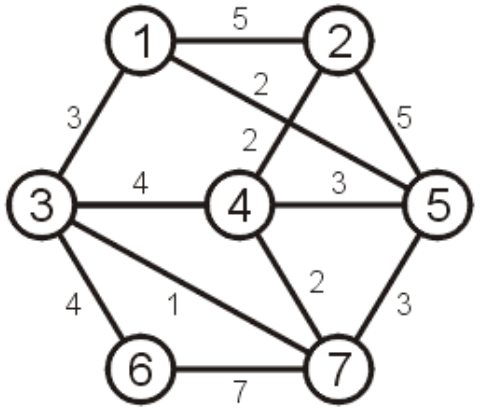


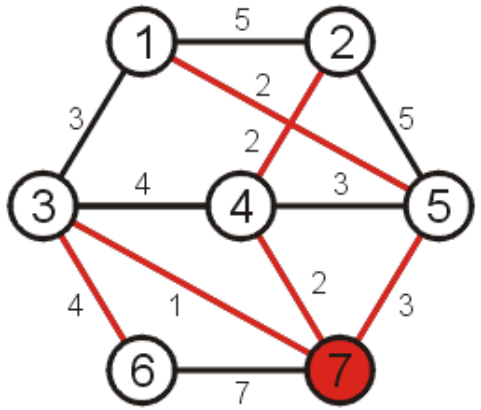
The weight of the above spanning tree is 28

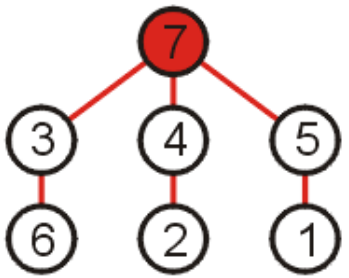
Minimum Spanning Trees

Which spanning tree which minimizes the weight?

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





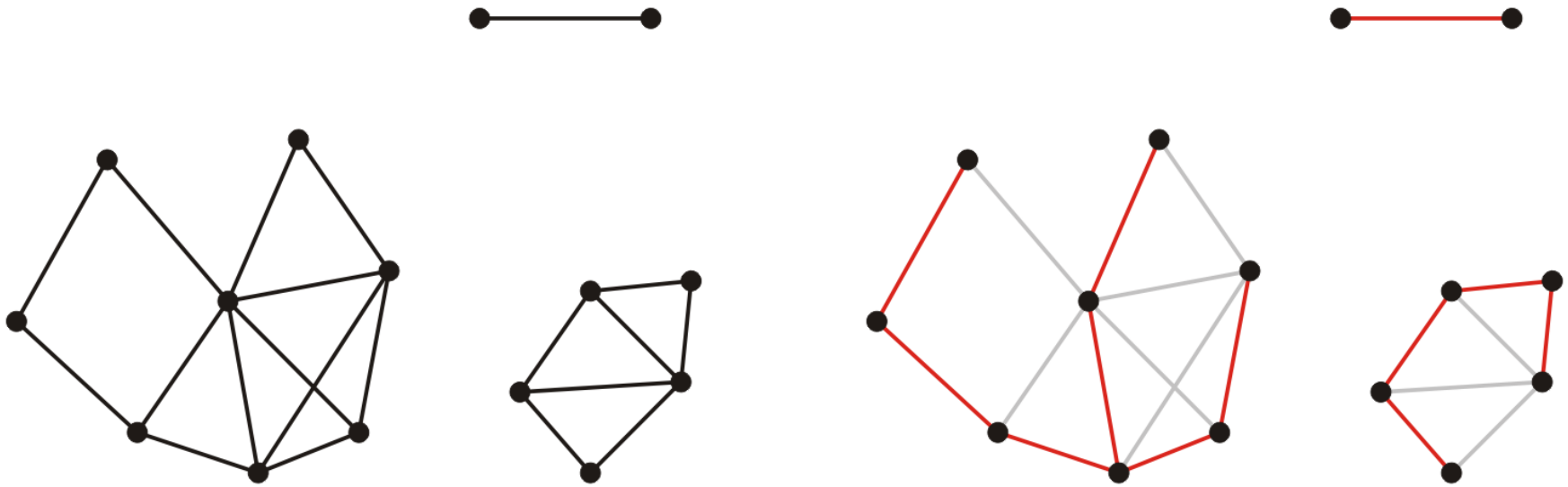


The weight of this spanning tree is 14

Spanning forests

Suppose that a graph is composed of N connected vertex-induced sub-graphs

- In this case, we may define a *spanning forest* as a collection of N spanning trees, one for each connected vertex-induced sub-graph



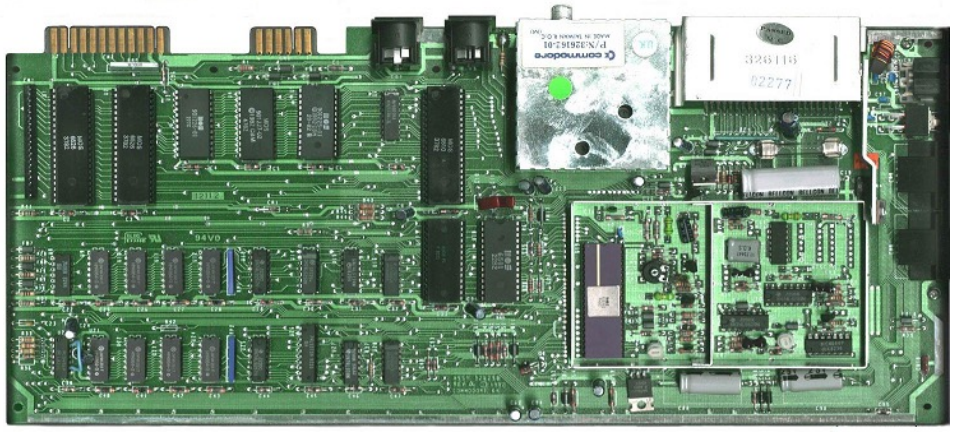
- A *minimum spanning forest* is therefore a collection of N minimum spanning trees, one for each connected vertex-induced sub-graph

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.commedore.ca

www.kpmb.com

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 means of connecting a number of LANs

- Minimize the number of bridges
- Costs not strictly dependant on distances

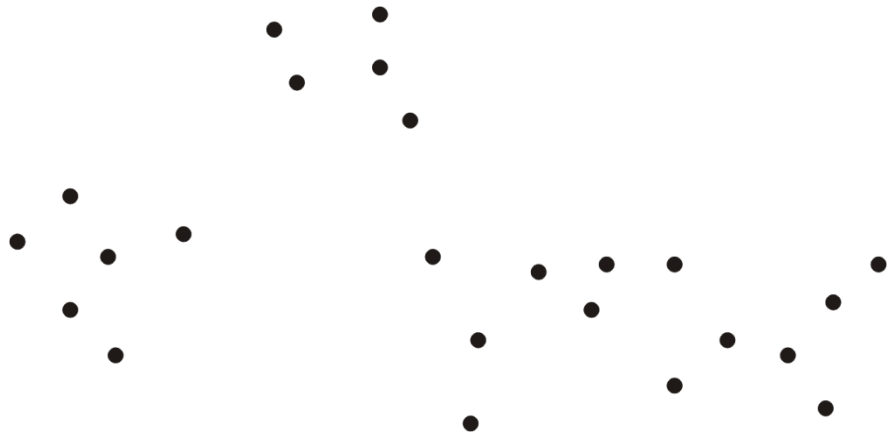
Application

Consider an *ad hoc* wireless network

- Any two terminals can connect with any others

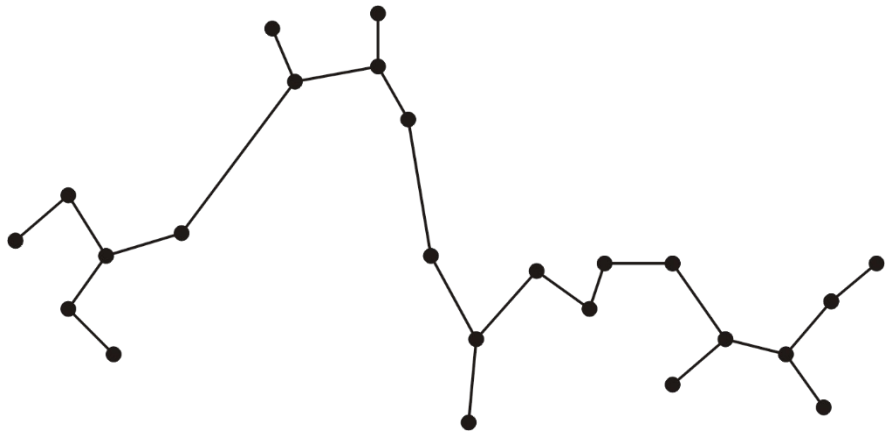
Problem:

- Errors in transmission increase with transmission length
- Can we find clusters of terminals which can communicate safely?



Application

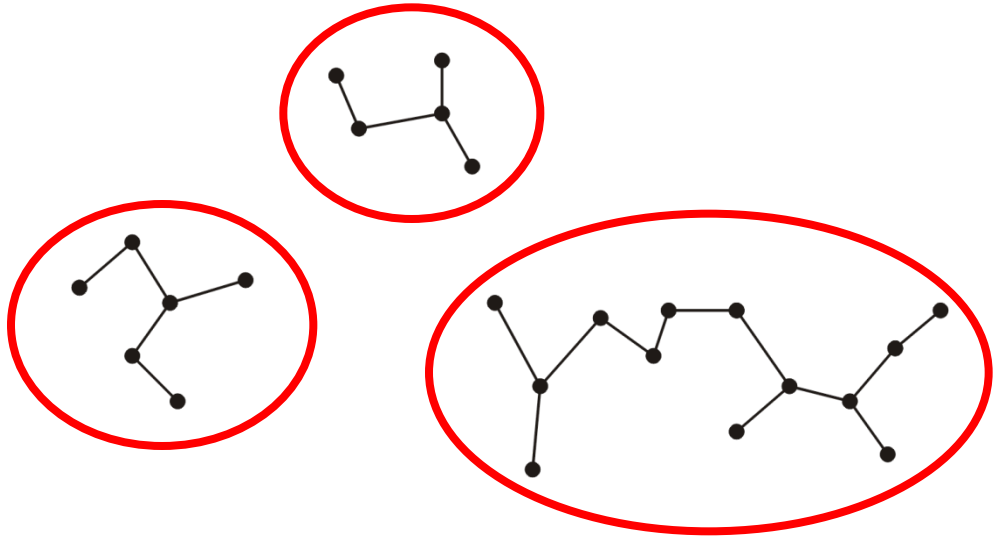
Find a minimum spanning tree

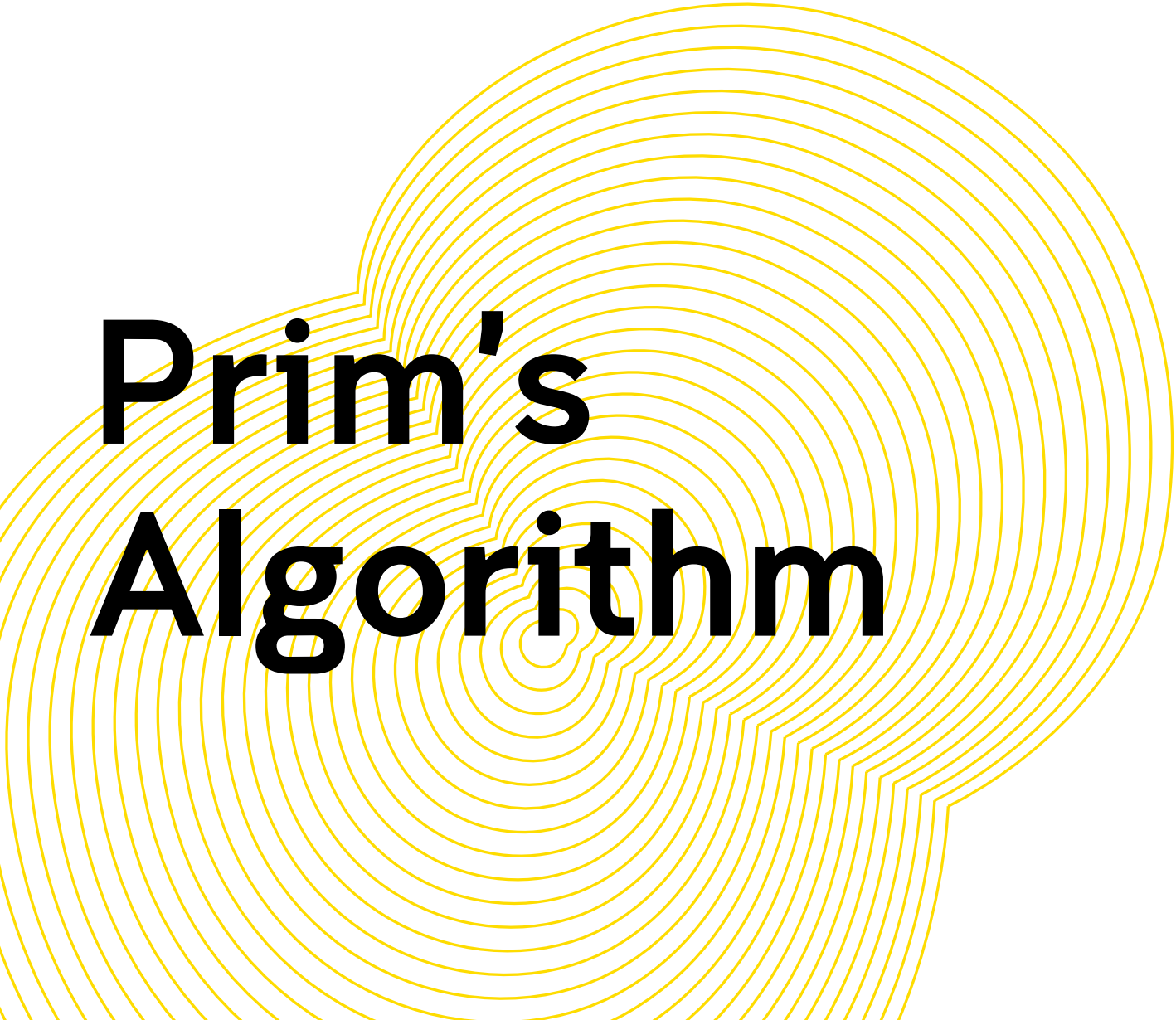


Application

Remove connections which are too long

This *clusters* terminals into smaller and more manageable sub-networks





Prim's Algorithm

Algorithms for finding MST

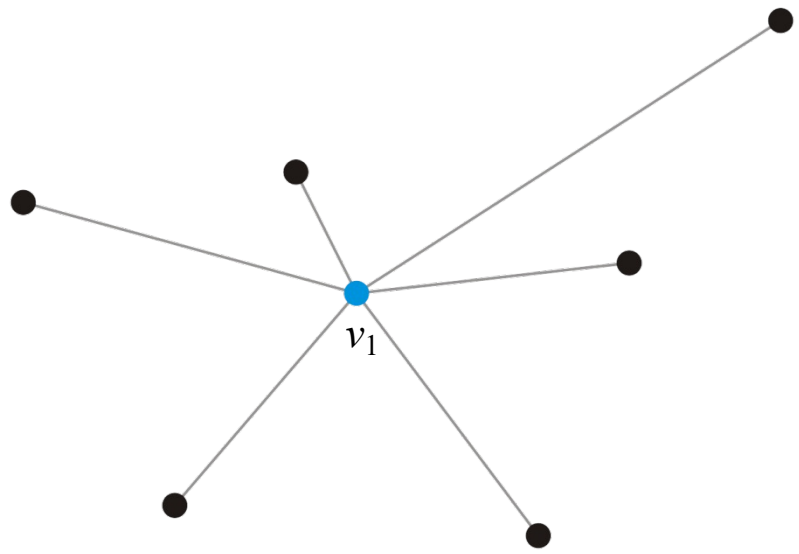
Prim's algorithm for finding the minimum spanning tree states:

- Start with an arbitrary vertex to form a minimum spanning tree on one vertex
- At each step, add that 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

Motivation for Prim's algorithm

Suppose we take a vertex

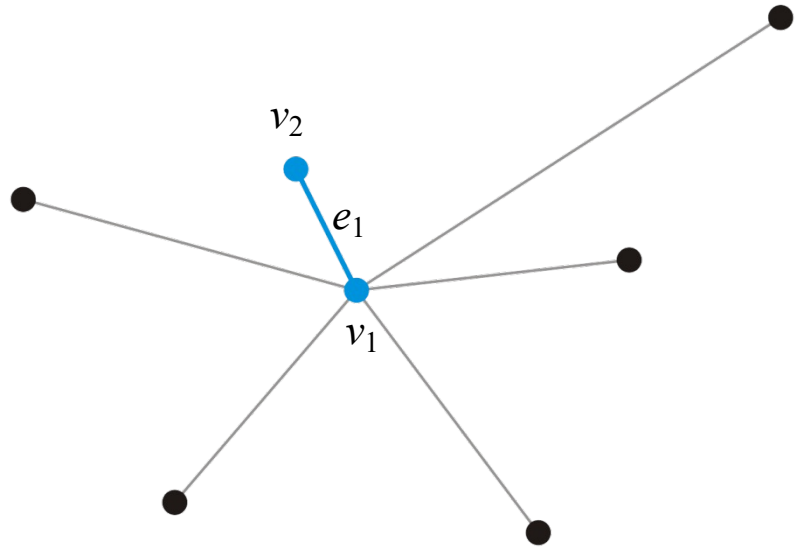
- Given a single vertex e_1, it forms a minimum spanning tree on one vertex



Motivation for Prim's algorithm

Add that adjacent vertex v_2 that has a connecting edge e_1 of minimum weight

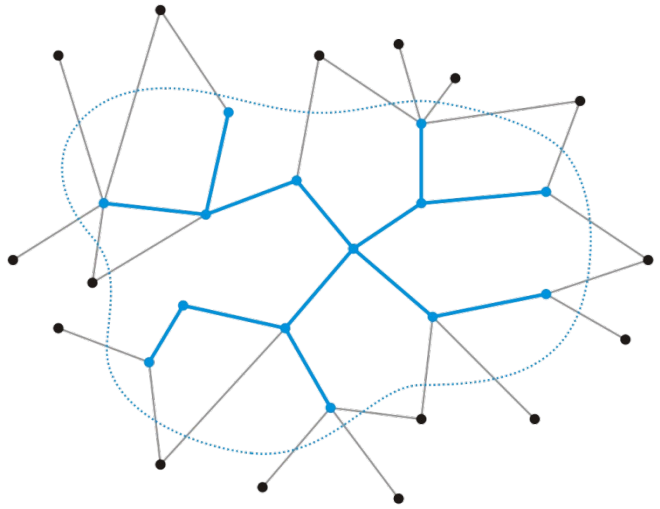
- This forms a minimum spanning tree on our two vertices and e_1 must be in any minimum spanning tree containing the vertices v_1 and v_2



Motivation for Prim's algorithm

Strategy:

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

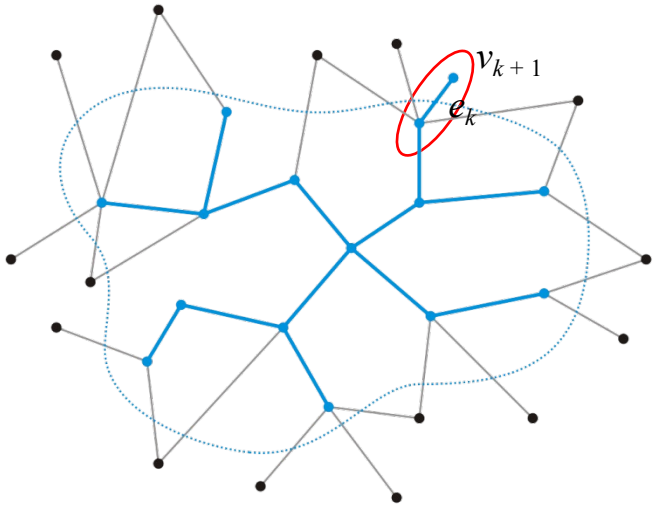


Motivation for Prim's algorithm

Add that 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 $\boldsymbol{k+1}$ **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?

Yes

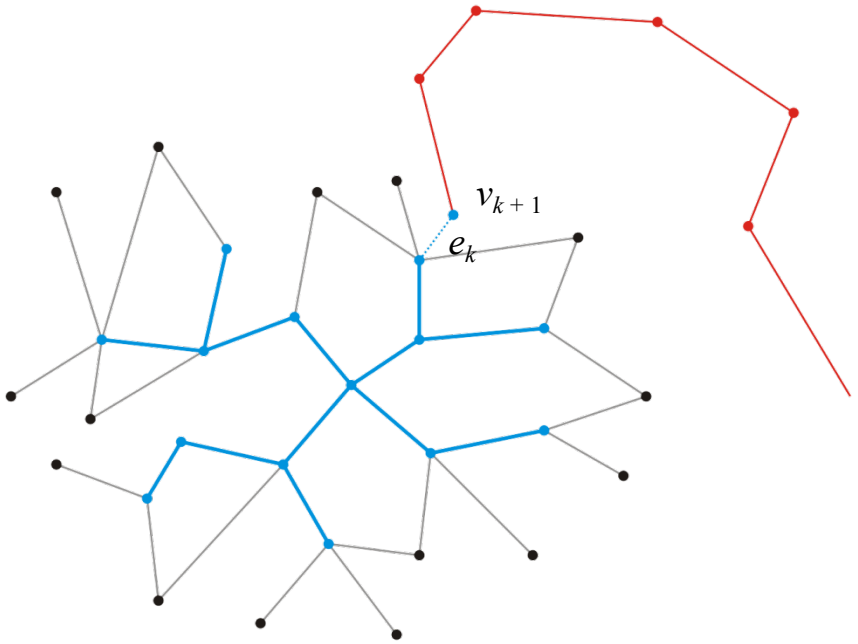


Motivation for Prim's algorithm

Optional

Proof: Suppose it does not

- Thus, vertex v_{k+1} is connected to the minimum spanning tree via another sequence of edges

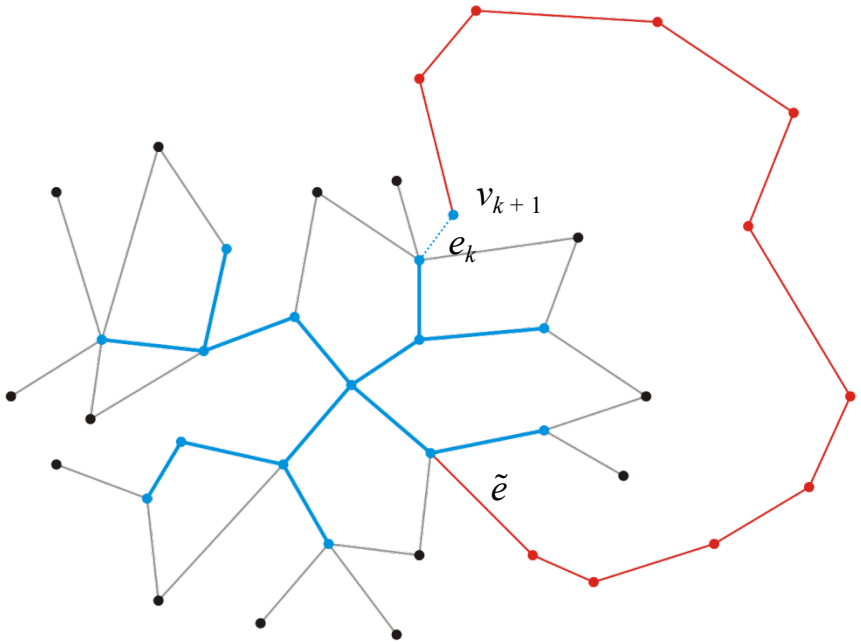


Motivation for Prim's algorithm

Optional

Proof: 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 $\tilde{e}$

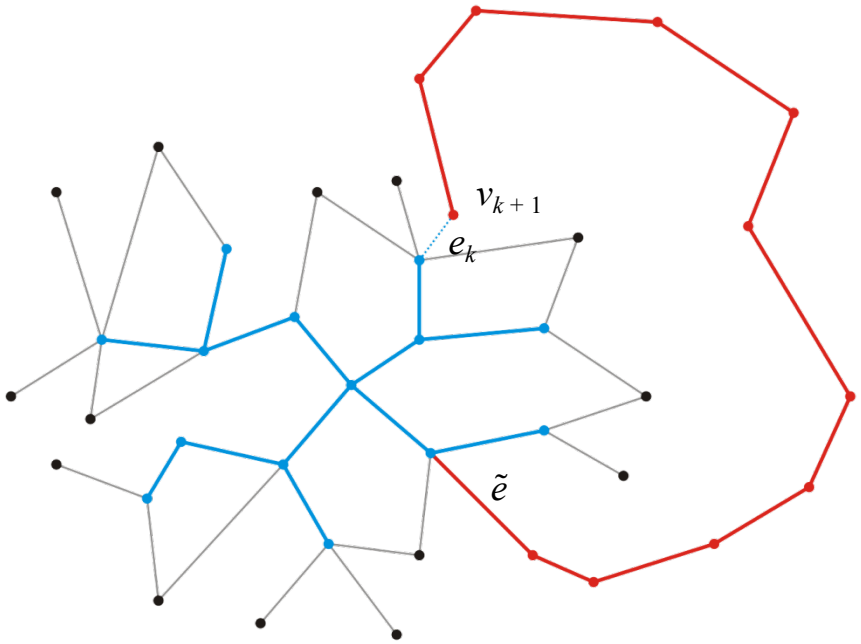


Motivation for Prim's algorithm

Optional

Proof: 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 $|\tilde{e}| > |e_k|$
- Therefore $|e_k| - |\tilde{e}| < 0$ where $|e|$ represents the weight of the edge e

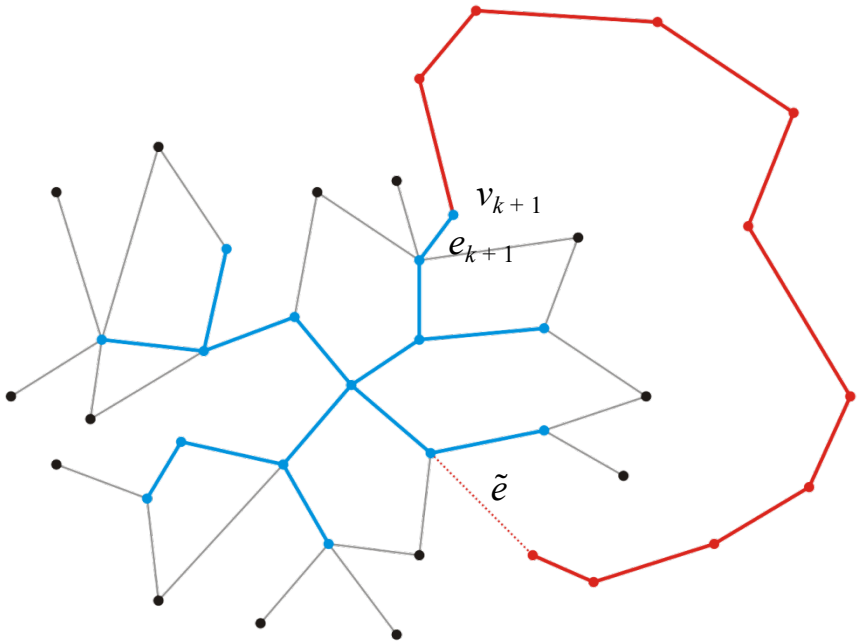


Motivation for Prim's algorithm

Optional

Proof: 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 $w + |e_{k+1}| - |\tilde{e}| \leq w$

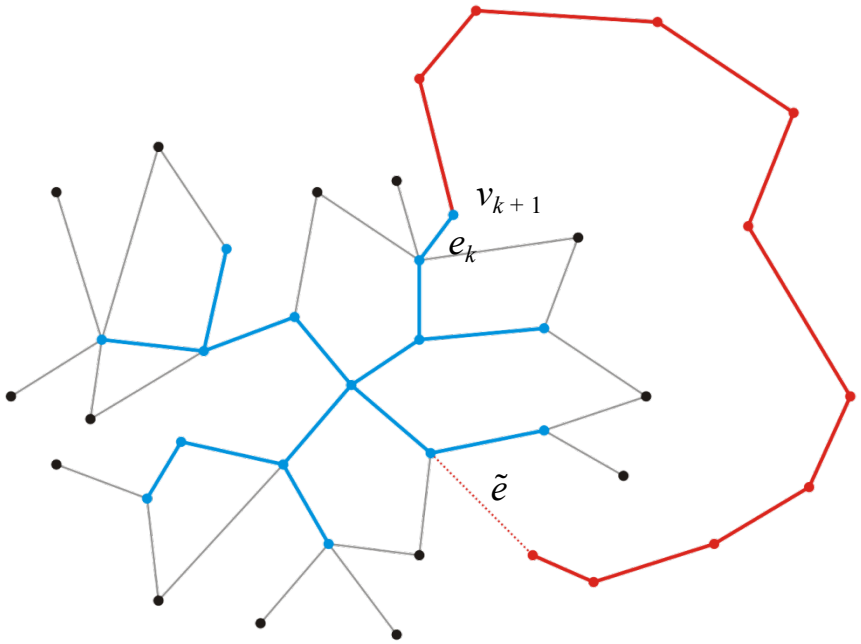


Motivation for Prim's algorithm

Optional

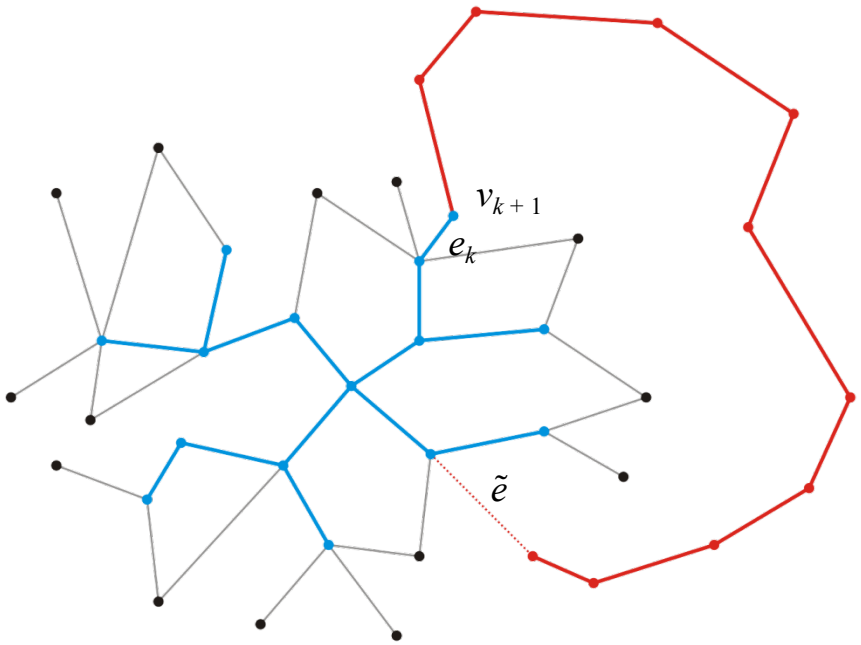
Proof: Thus, by swapping e_k for $\tilde{e}$, we have a spanning tree that has less weight than the so-called minimum spanning tree containing $\tilde{e}$

- This contradicts our assumption that the spanning tree containing $\tilde{e}$ 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 $k = 1$



Prim's Algorithm

Associate with each vertex three items of data:

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

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

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 equal the weight of the edge
  - Set the current vertex as the parent of the adjacent vertex

Prim's Algorithm

Stopping Conditions:

- There are no unvisited vertices which have a distance $< \infty$

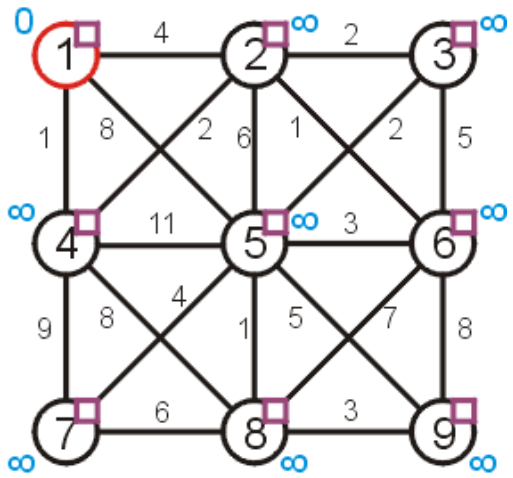
If all vertices have been visited, we have a spanning tree of the entire graph

If 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

Prim's Algorithm

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

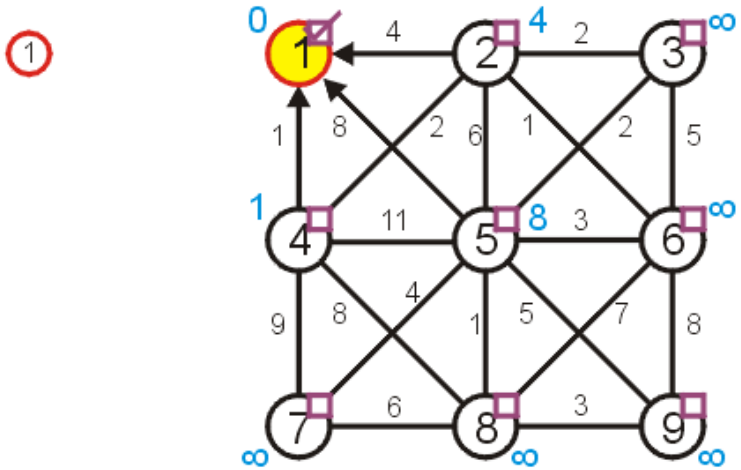
First we set up the appropriate table and initialize it



		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

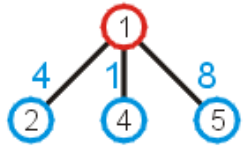
Visiting vertex 1, we update vertices 2, 4, and 5



		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

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:

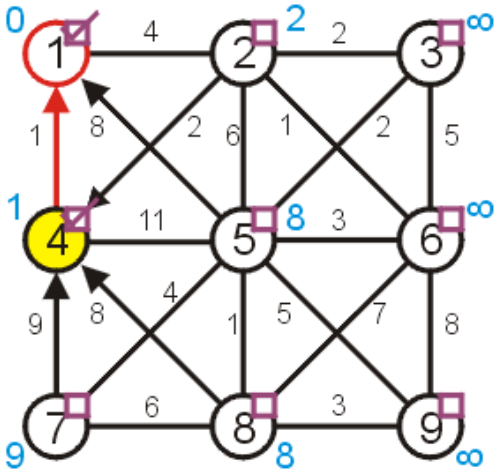


As we wish to find a *minimum* spanning tree, it makes sense we add that vertex with a connecting edge with least weight

Prim's Algorithm

The next unvisited vertex with minimum distance is vertex 4

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



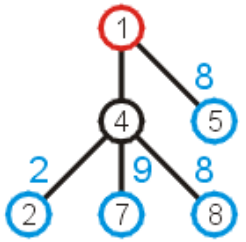
		Distance	Parent
1	T	0	0
2	F	2	4
3	F	∞	0
4	T	1	1
5	F	8	1
6	F	∞	0
7	F	9	4
8	F	8	4
9	F	∞	0

Prim's Algorithm

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)

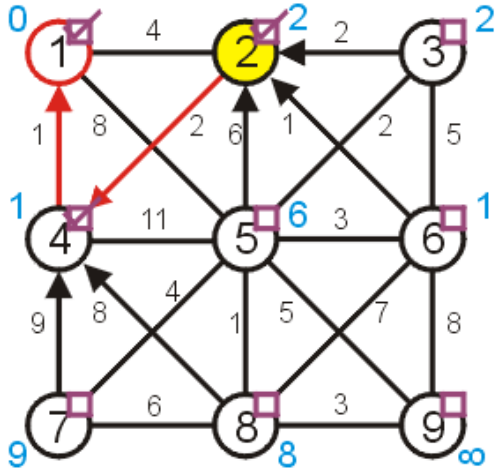
We add that edge with the least weight: (4, 2)



Prim's Algorithm

Next visit vertex 2

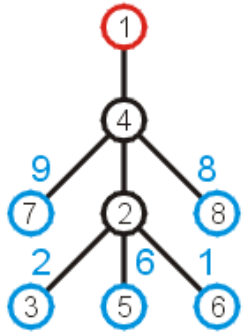
- Update 3, 5, and 6



		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

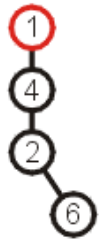
Again looking at the shortest edges to each of the vertices adjacent to the current tree, we note that we can add (2, 6) with the least increase in weight

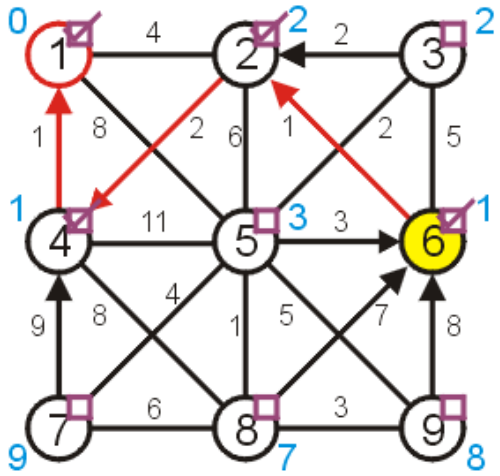


Prim's Algorithm

Next, we visit vertex 6:

- update vertices 5, 8, and 9



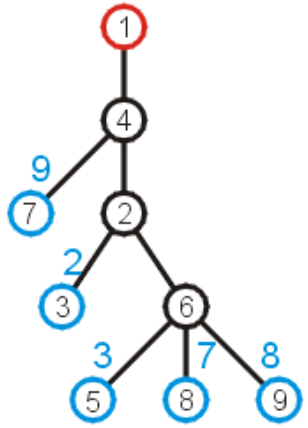


		Distance	Parent
1	T	0	0
2	T	2	4
3	F	2	2
4	T	1	1
5	F	3	6
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

Prim's Algorithm

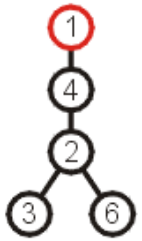
The edge with least weight is (2, 3)

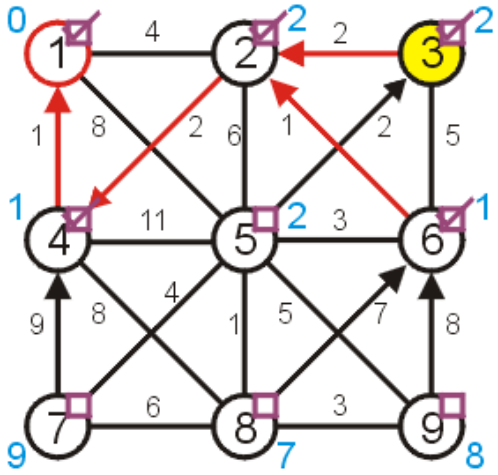
- This adds the weight of 2 to the weight minimum spanning tree



Prim's Algorithm

Next, we visit vertex 3 and update 5

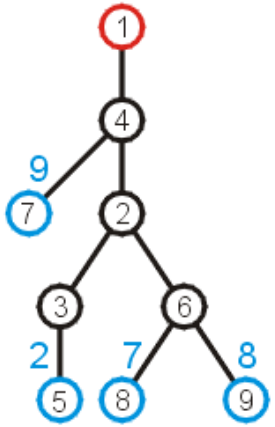




		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	F	2	3
6	T	1	2
7	F	9	4
8	F	7	6
9	F	8	6

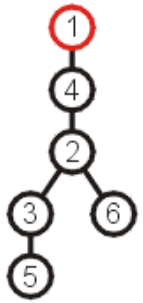
Prim's Algorithm

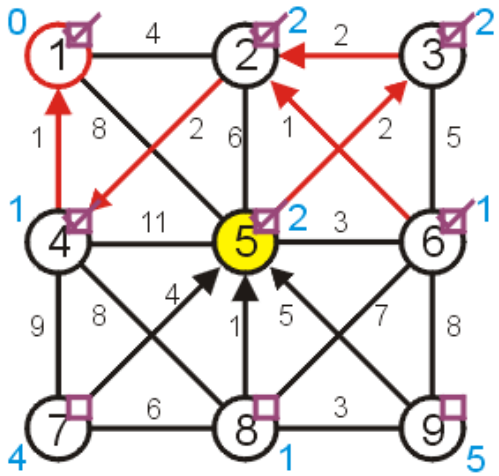
At this point, we can extend the tree by adding the edge (3, 5)



Prim's Algorithm

Visiting vertex 5, we update 7, 8, 9



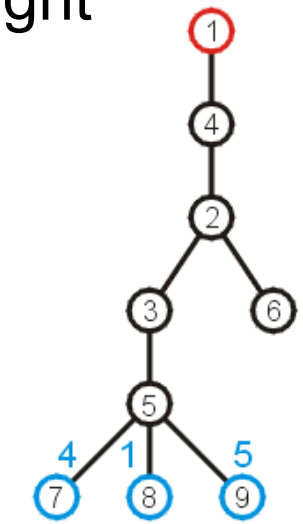


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	F	1	5
9	F	5	5

Prim's Algorithm

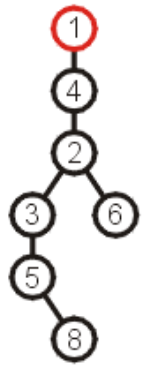
At this point, there are three possible edges which we could include which will extend the tree

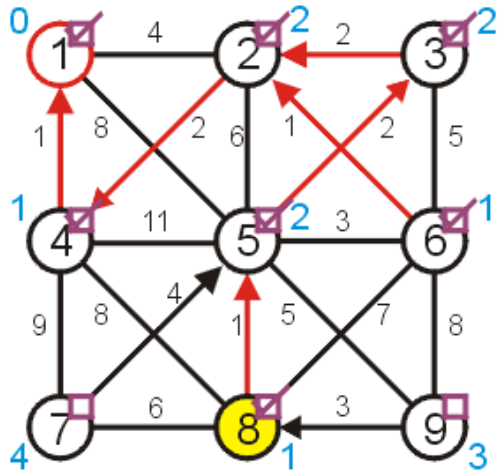
The edge to 8 has the least weight



Prim's Algorithm

Visiting vertex 8, we only update vertex 9

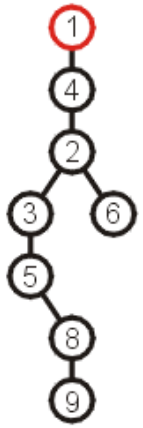


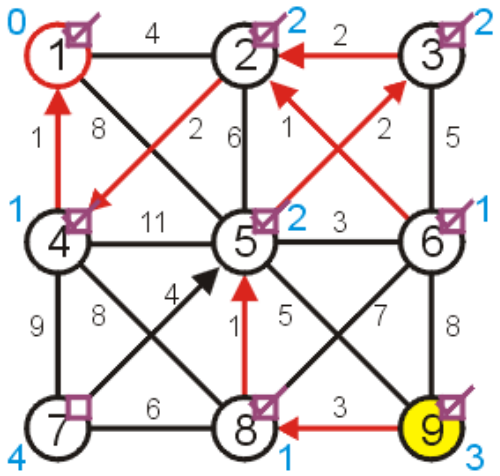


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	T	1	5
9	F	3	8

Prim's Algorithm

There are no other vertices to update while visiting vertex 9

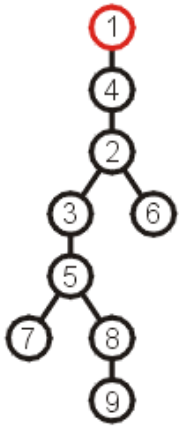


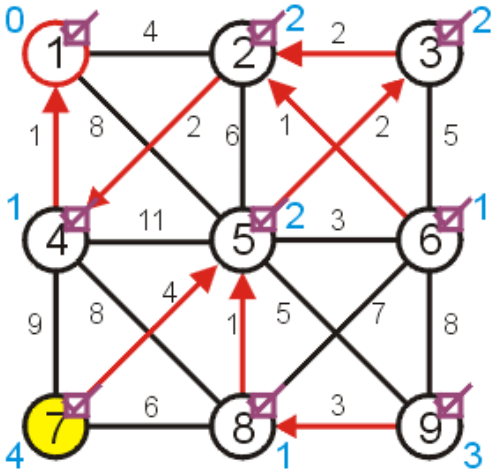


		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	F	4	5
8	T	1	5
9	T	3	8

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

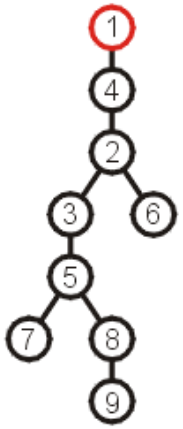
At this point, there are no more unvisited vertices, and therefore we are done

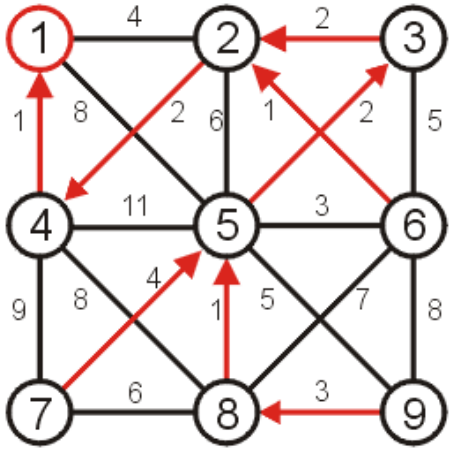
If at any point, all remaining vertices had a distance of ∞, this would indicate that the graph is not connected

- 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

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

Implementation and analysis

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

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

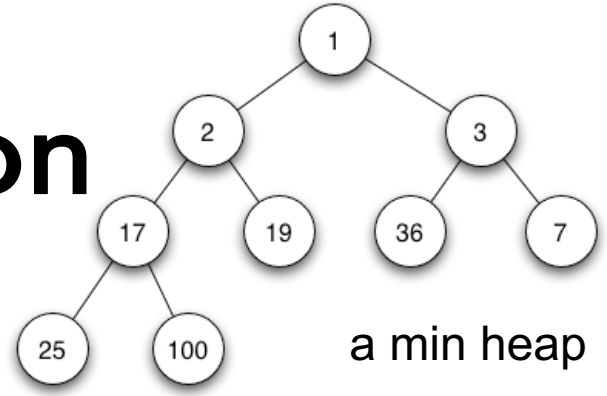
- Iterating through the table requires is $O(|V|)$ time
- Each time we find a vertex, we must check all of its neighbors
- With an adjacency matrix, the run time is $O(|V|(|V| + |V|)) = O(|V|^2)$
- With an adjacency list, the run time is $O(|V|^2 + |E|) = O(|V|^2)$ as $|E| = O(|V|^2)$

Can we do better?

Recall, we only need the vertex with the shortest distance next

How about a min heap?

Min-heap-based optimization



a min heap

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

- The min heap will also require $O(|V|)$ memory which **contains the distance of all the vertices**

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

- Obtain the shortest distance from the min heap O(1), maintain the heap $O(\log(|V|))$
- Thus, the work required for this is $O(|V| \log(|V|))$

Is this all the work that is necessary?

- Recall that for the *closest* vertex in an iteration, **we try to update the distance of all its neighbors**, thus there are O(|E|) updates in total and each update in the heap requires $O(\log(|V|))$.
- Thus, the work required for this is $O(|E| \log(|V|))$

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

Time Complexity

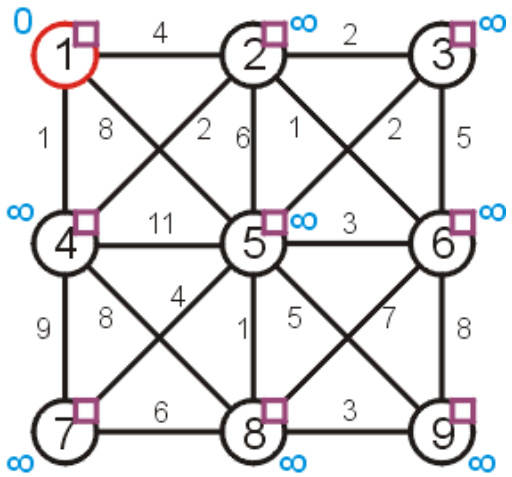
Prim's algorithm: different implementations

- $O(m*log(n))$
- $O(m*log(m))$
- $O(m+n*log(n))$
  - theoretically good, but not practically efficient

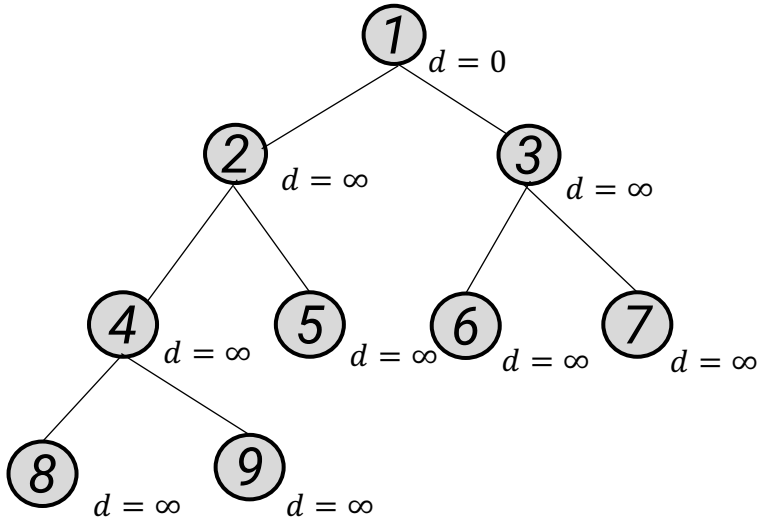
Confused?

We will analyze these results again in the Dijkstra Algorithm.

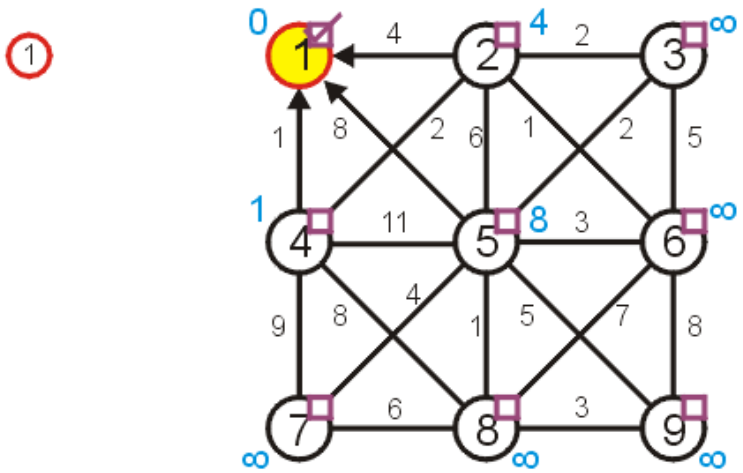
Min-heap-based optimization



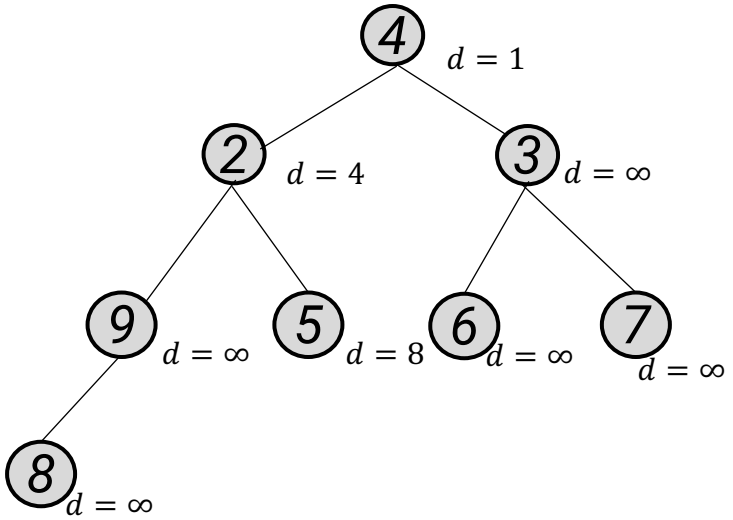
		Distance	Parent
1	F	0	0
2	F	∞	0
3	F	∞	0
4	F	∞	0
5	F	∞	0
6	F	∞	0
7	F	∞	0
8	F	∞	0
9	F	∞	0



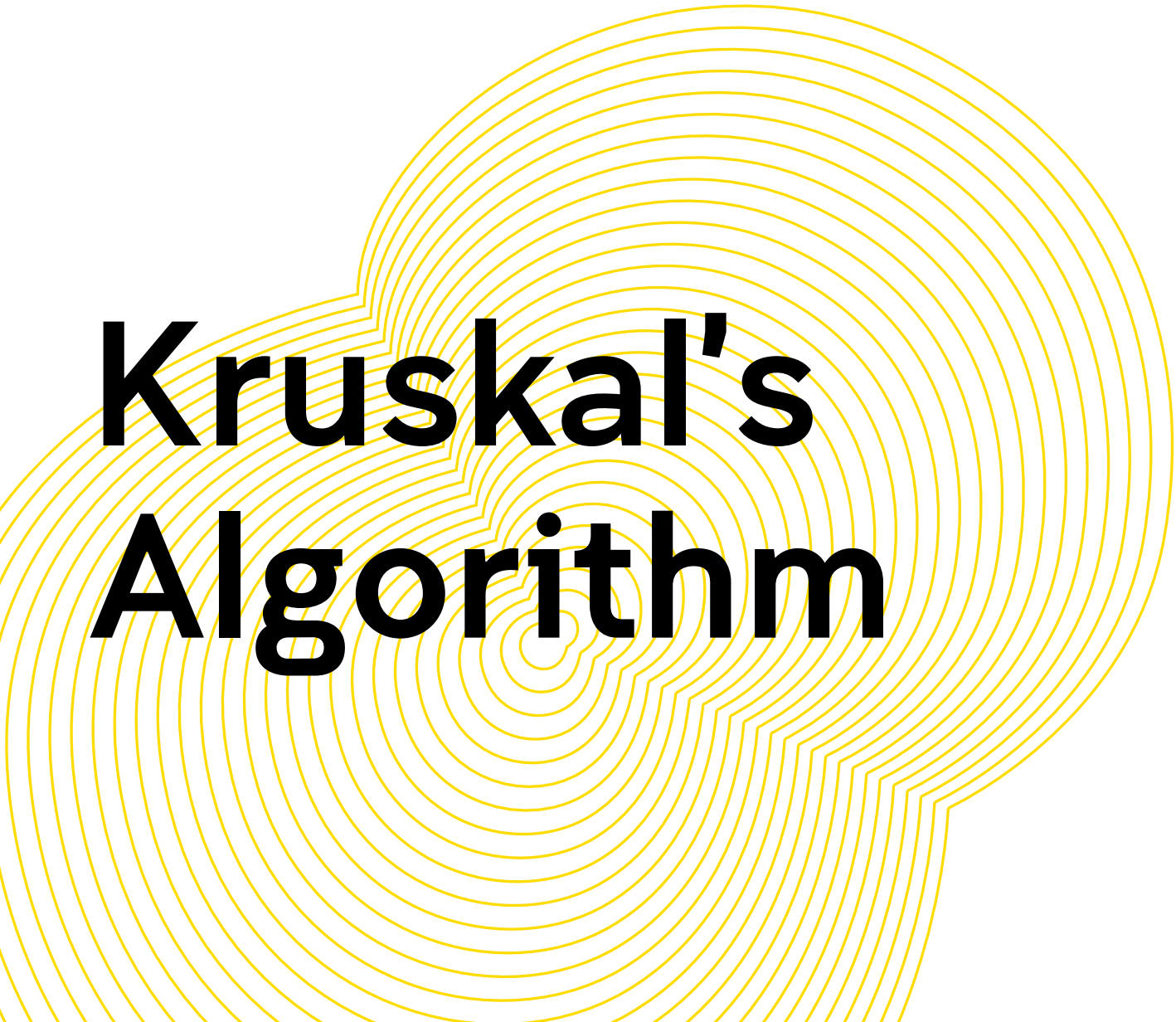
Min-heap-based optimization



		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



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



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

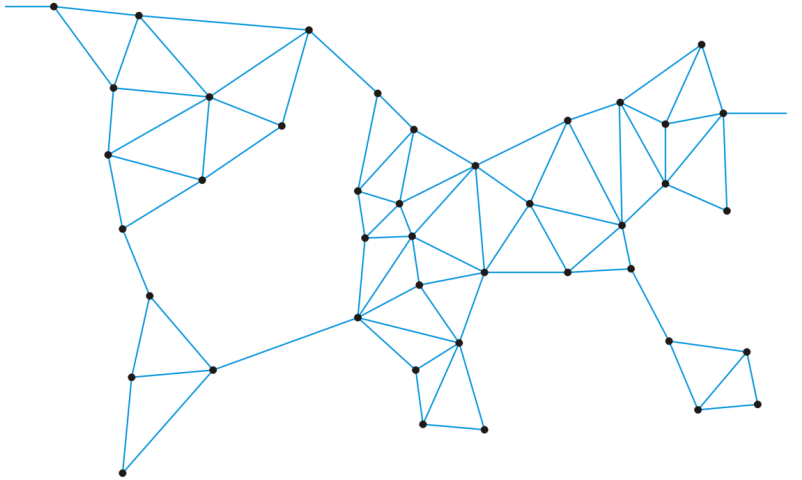
The halting point is:

- 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

Consider the game of *Risk* from Parker Brothers

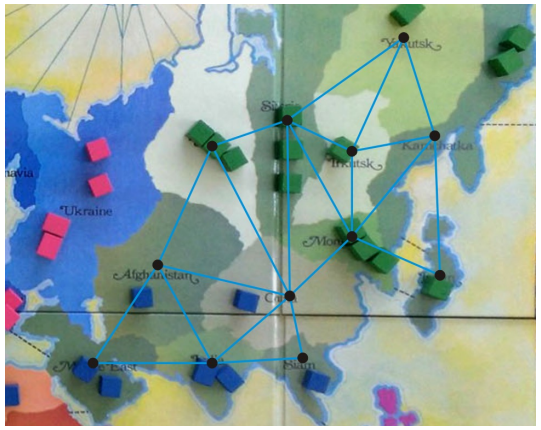
- A game of world domination
- The world is divided into 42 connected regions
- The regions are vertices and edges indicate adjacent regions

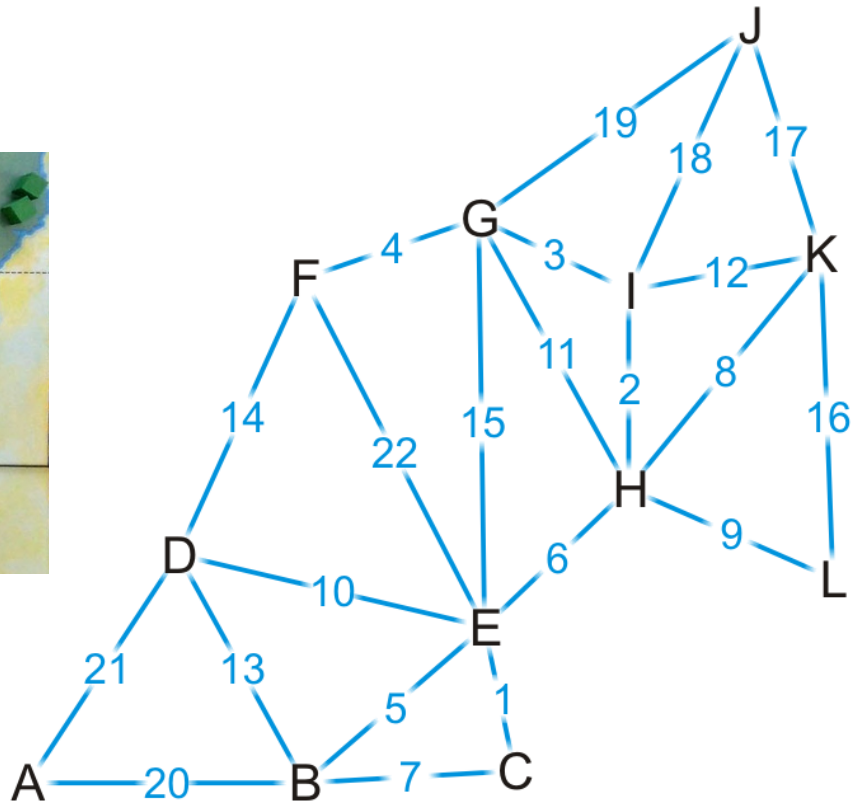


http://thunderbird37.com/tag/parker-brothers/

Example

Here is our abstract representation of Asia. Let us give a weight to each of the edges. First, we sort the edges based on weight.

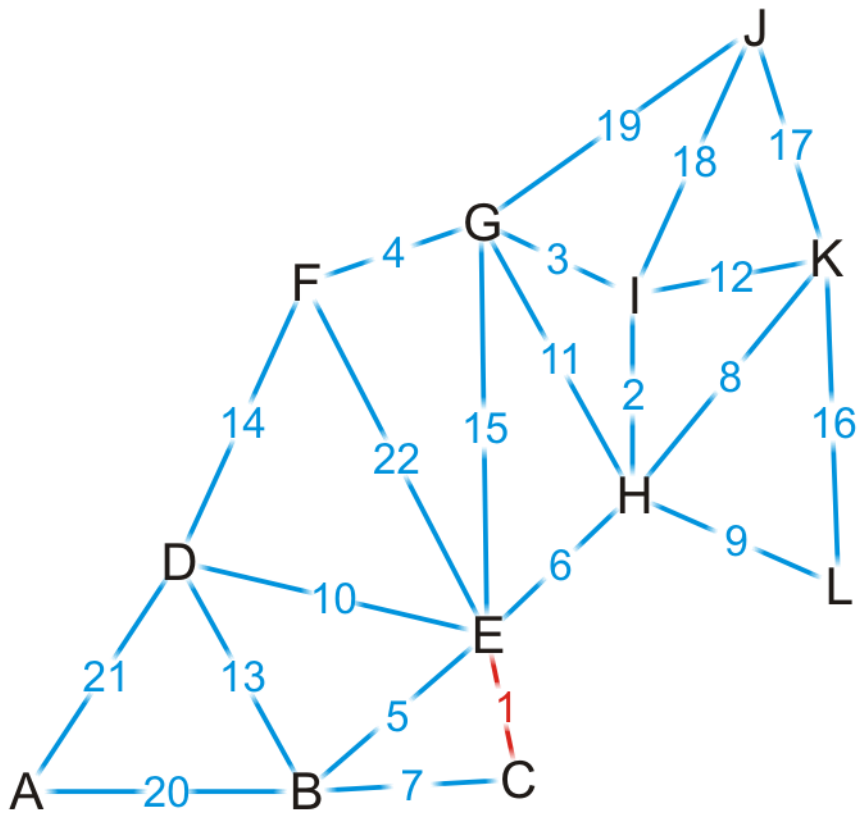




{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

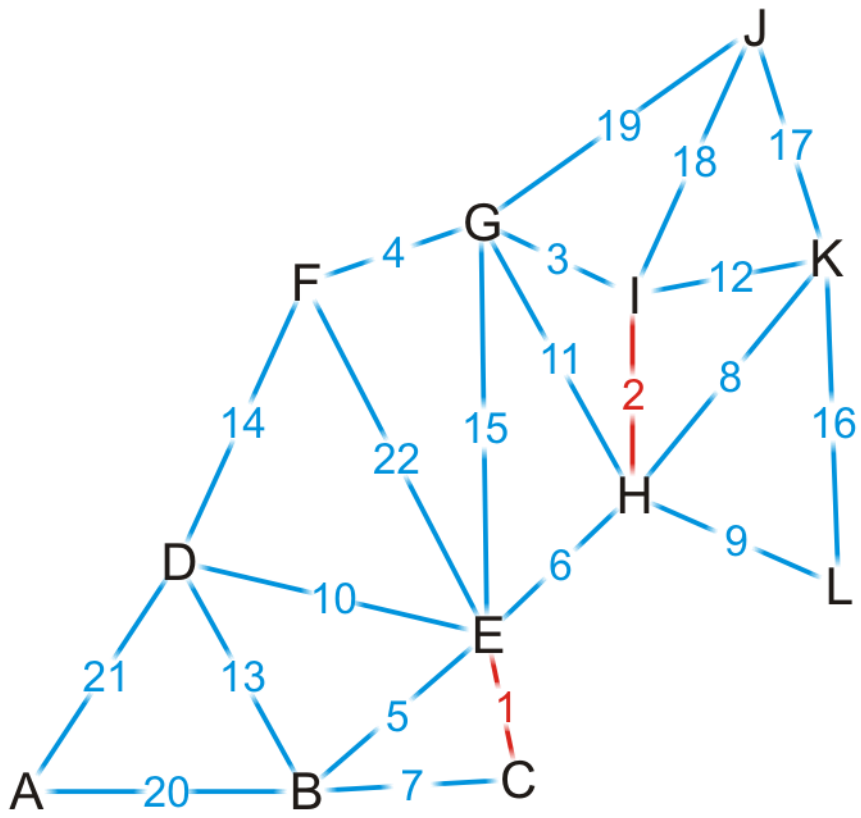
We start by adding edge {C, E}



→ {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

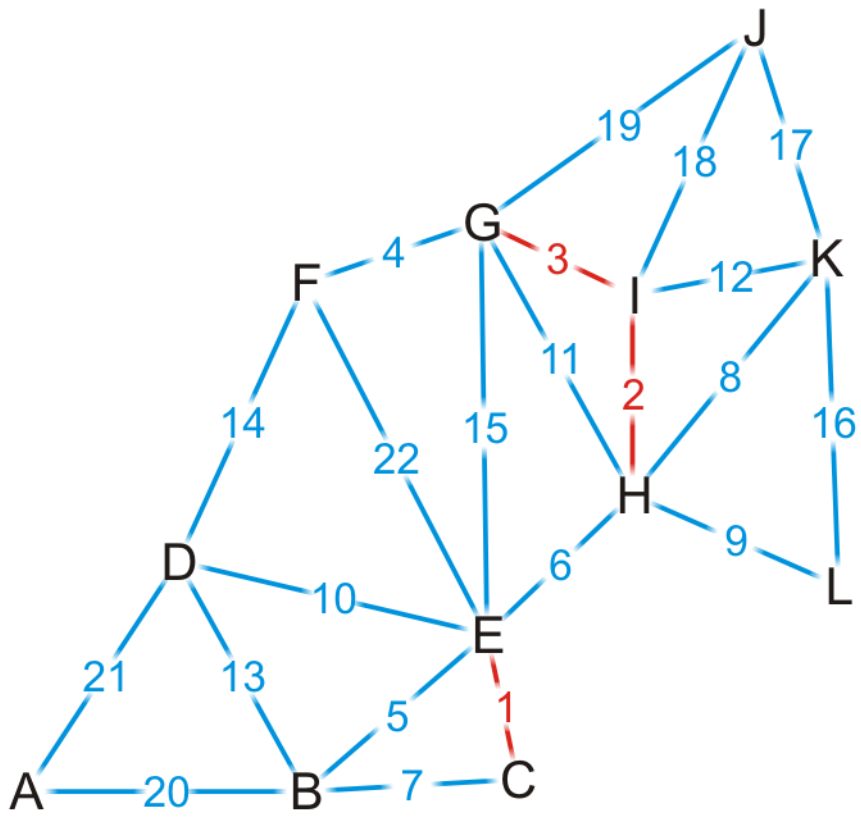
We add edge {H, I}



{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

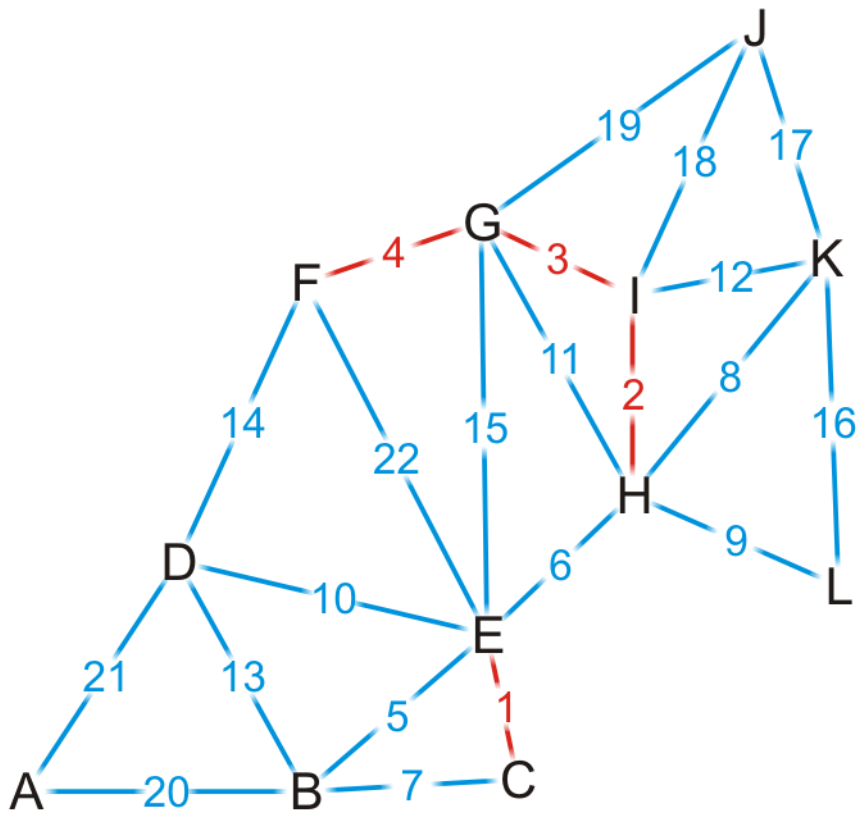
We add edge {G, I}



{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

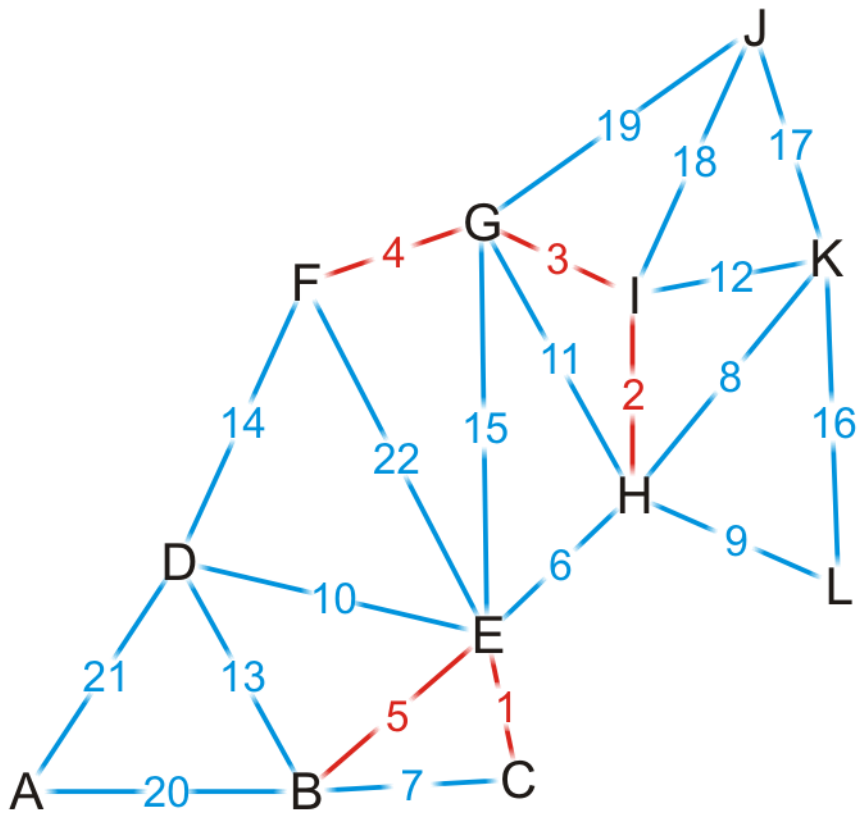
We add edge {F, G}



{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}

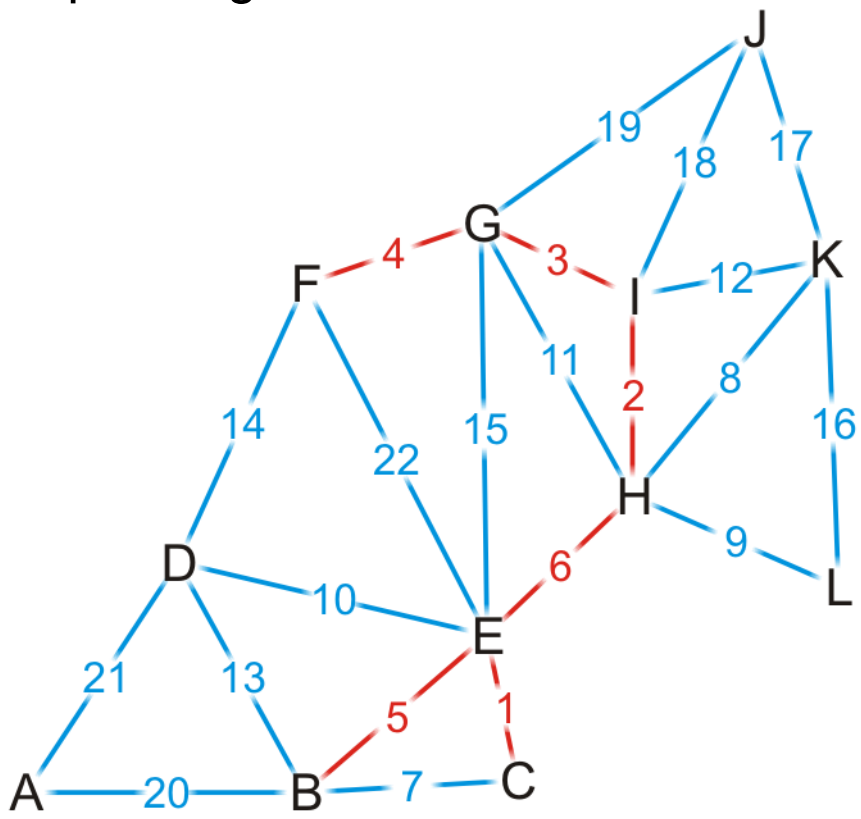


{C, E}
{H, I}
{G, I}
{F, G}
→ {B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

We add edge {E, H}

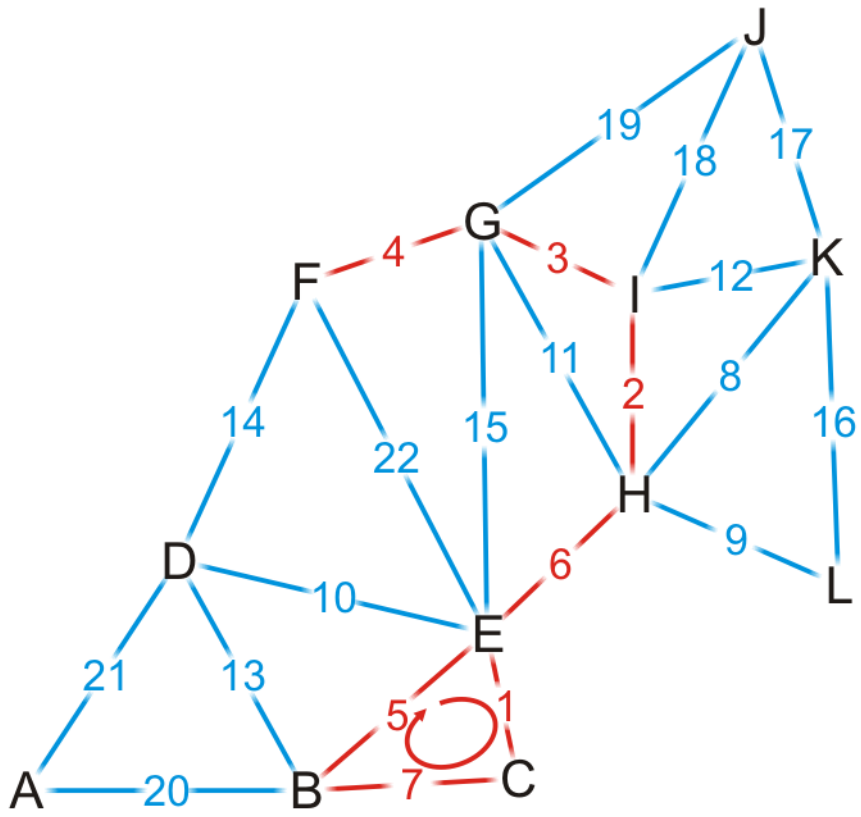
- This coalesces the two spanning sub-trees into one



{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

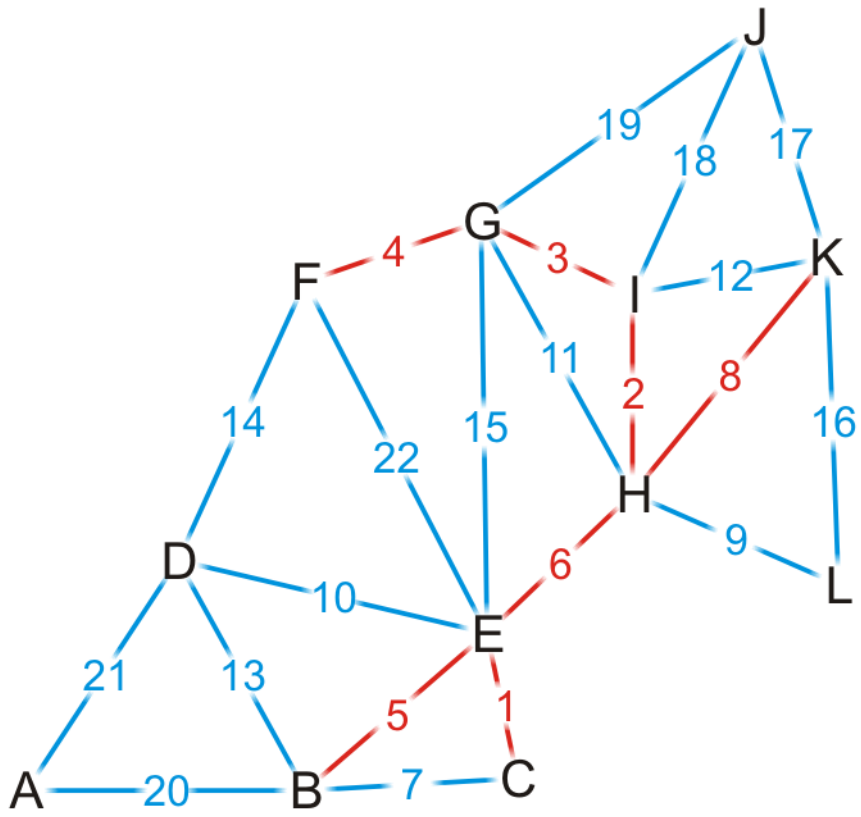
We try adding {B, C}, but it creates a cycle



{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

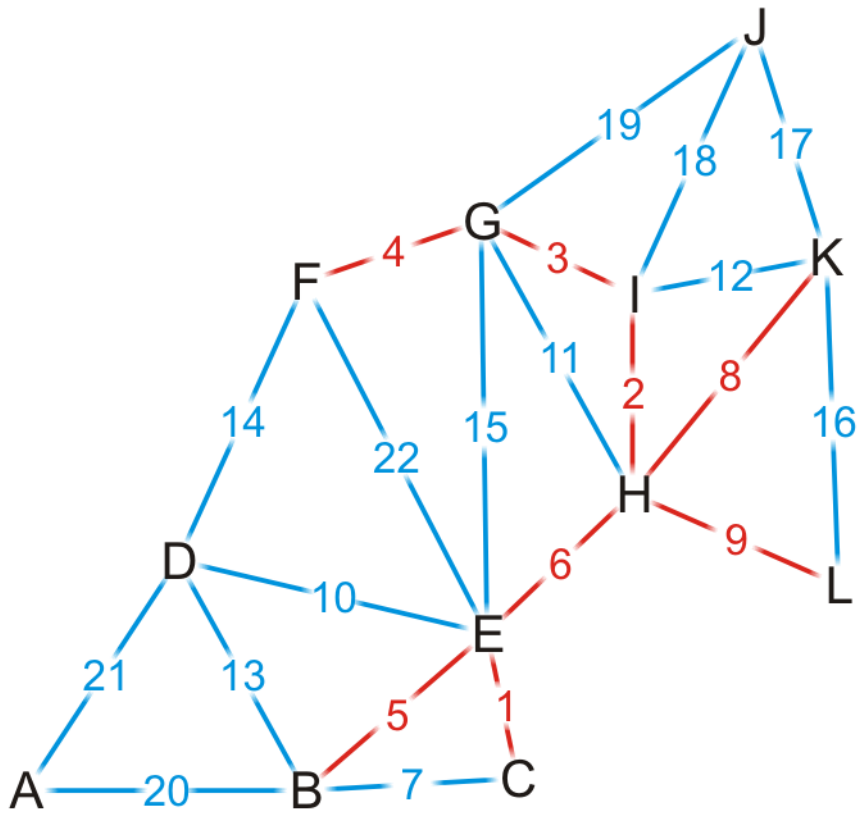
We add edge {H, K}



{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

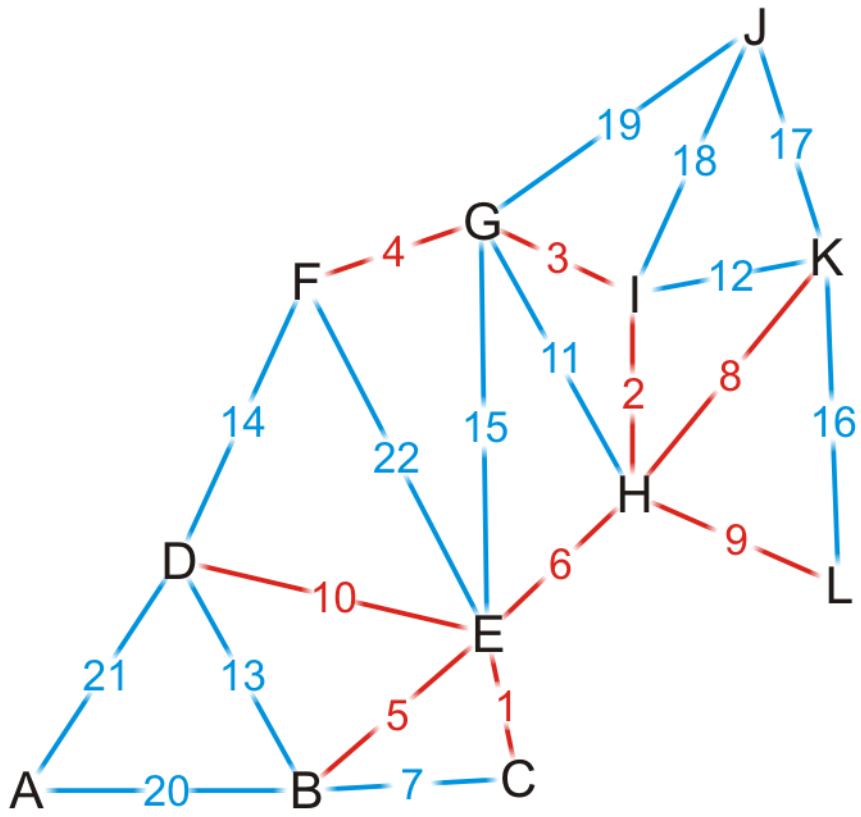
We add edge {H, L}



{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

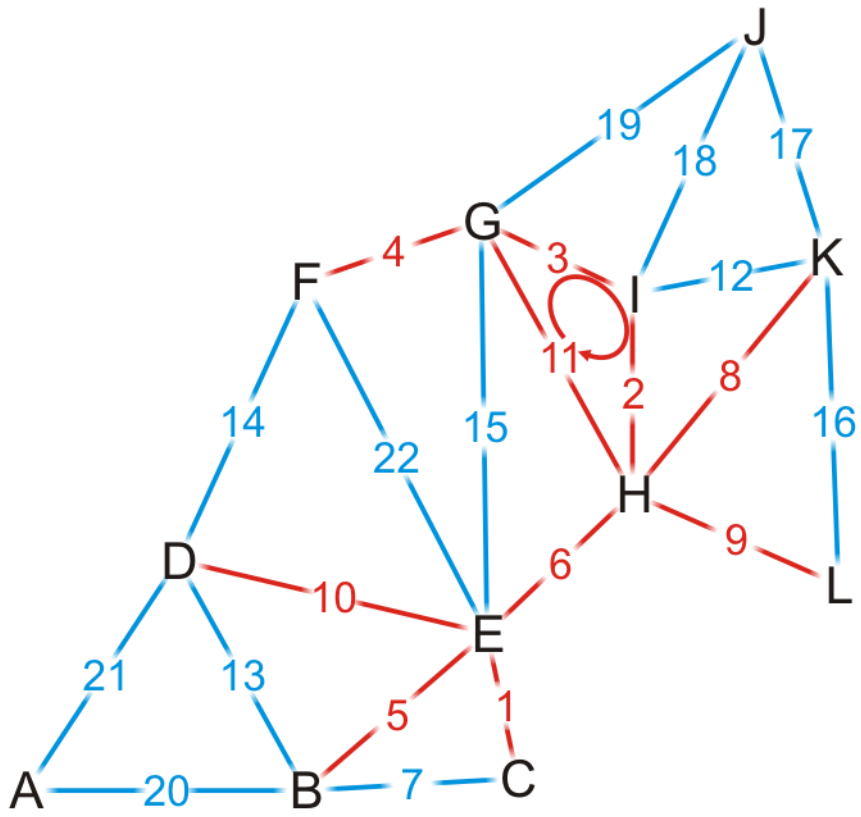
We add edge {D, E}



{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

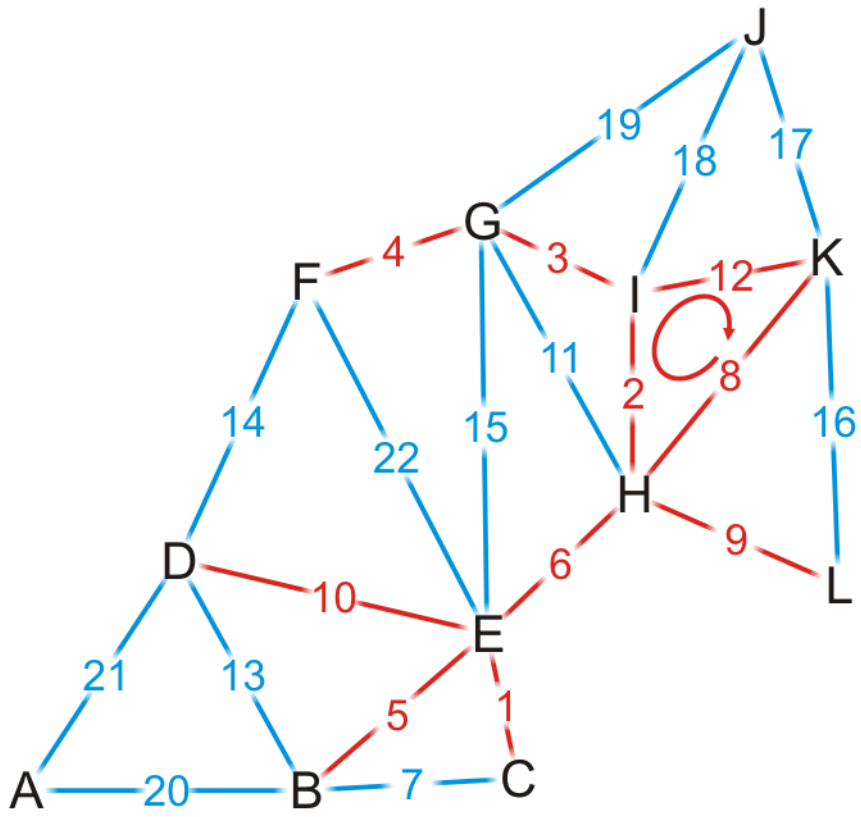
We try adding {G, H}, but it creates a cycle



{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

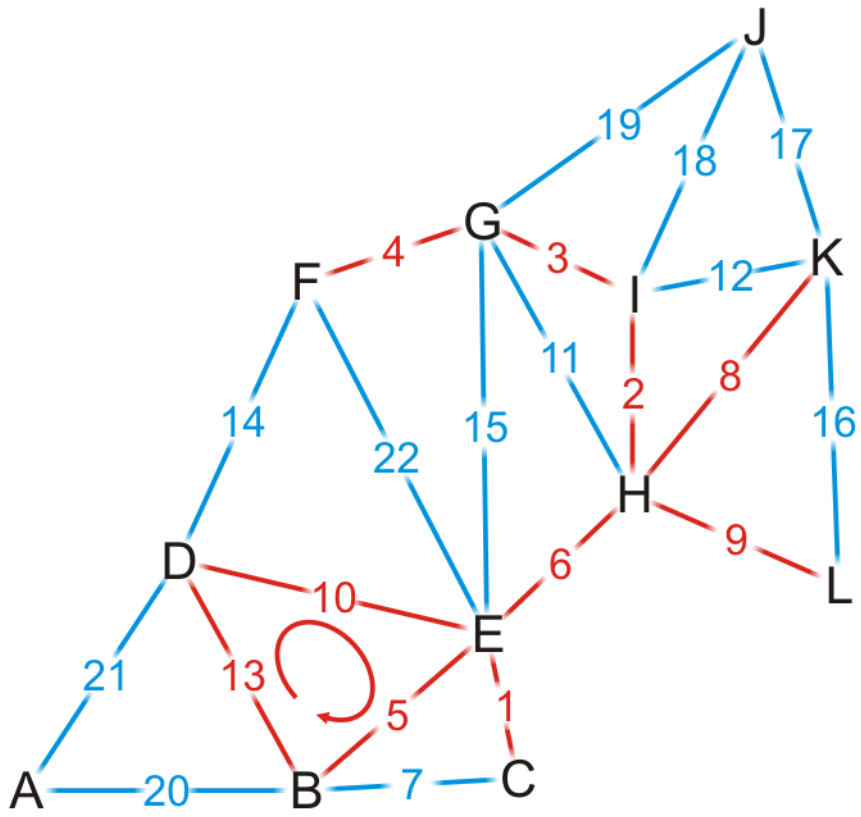
We try adding {I, K}, but it creates a cycle



{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

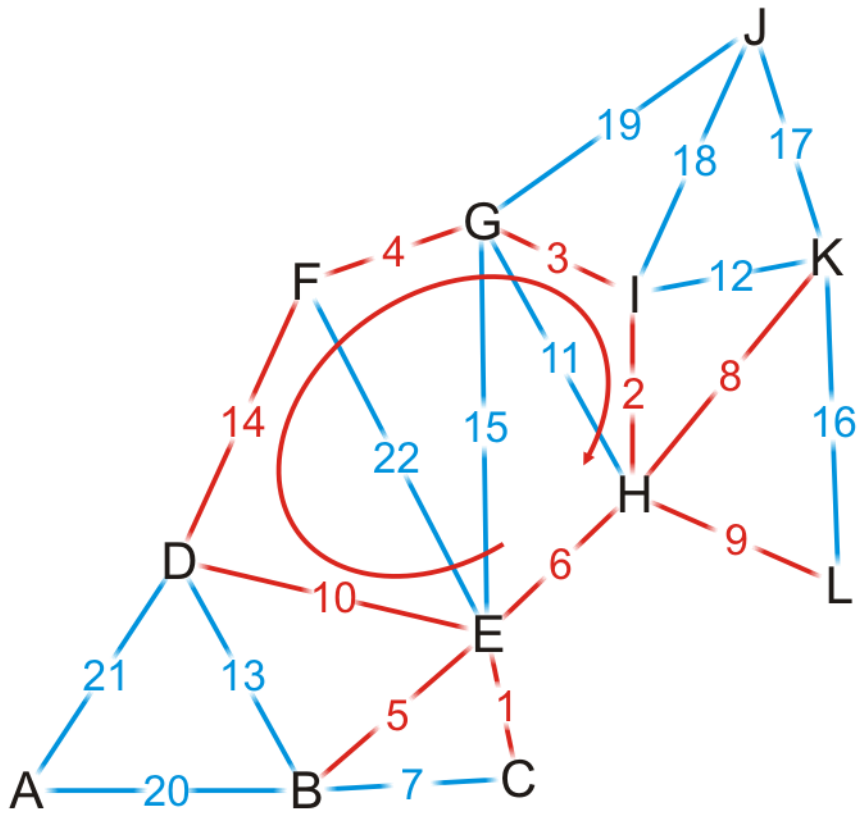
We try adding {B, D}, but it creates a cycle



{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

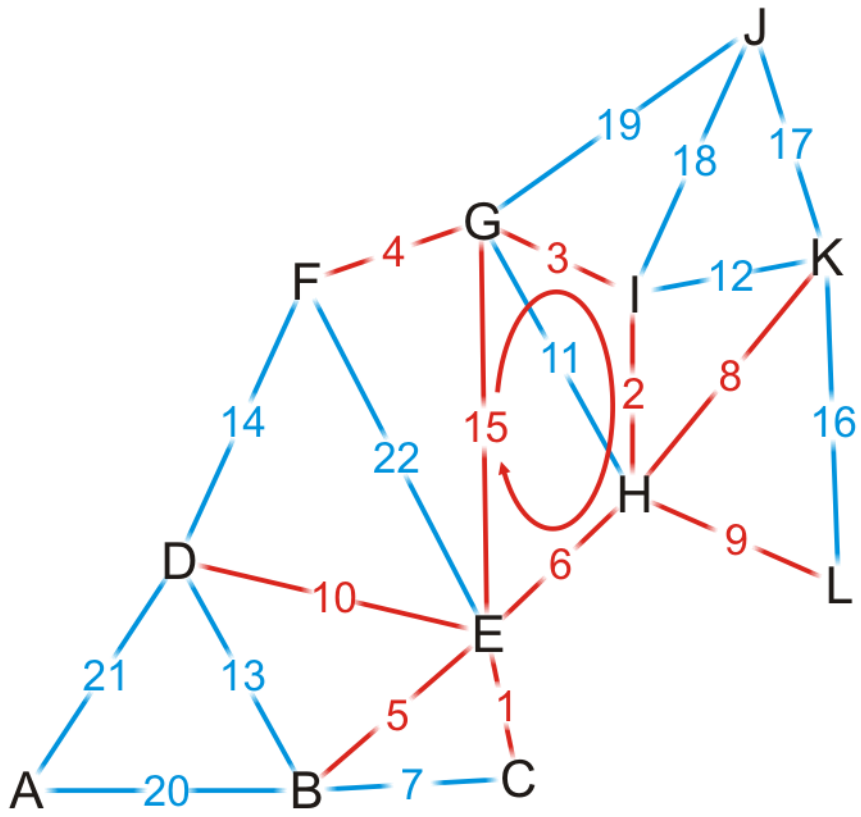
We try adding {D, F}, but it creates a cycle



{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

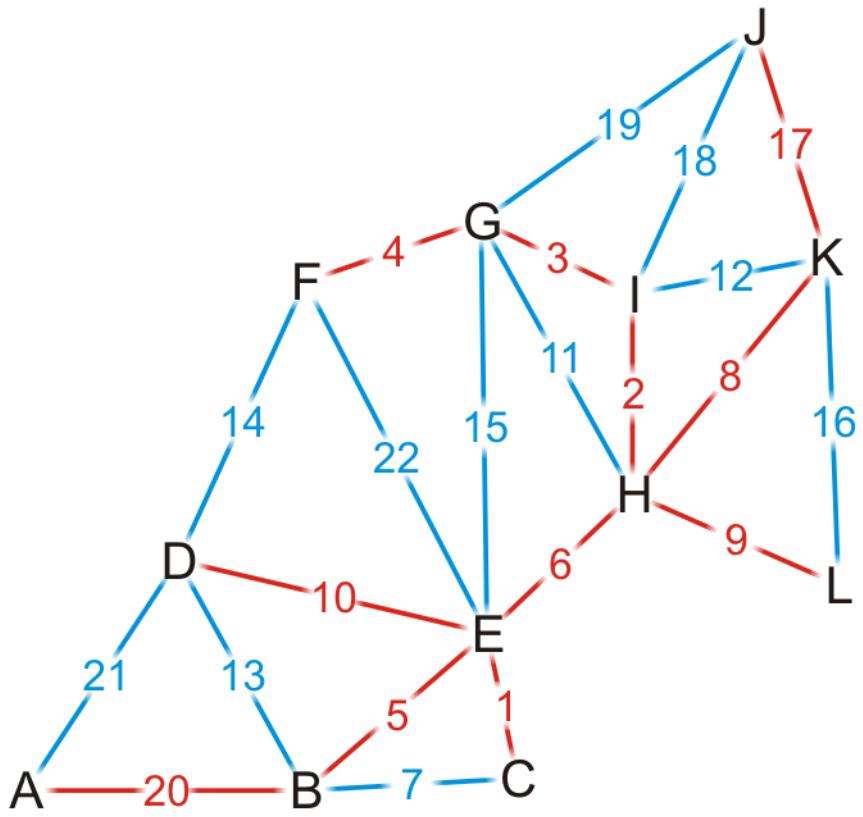
We try adding {E, G}, but it creates a cycle



{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}

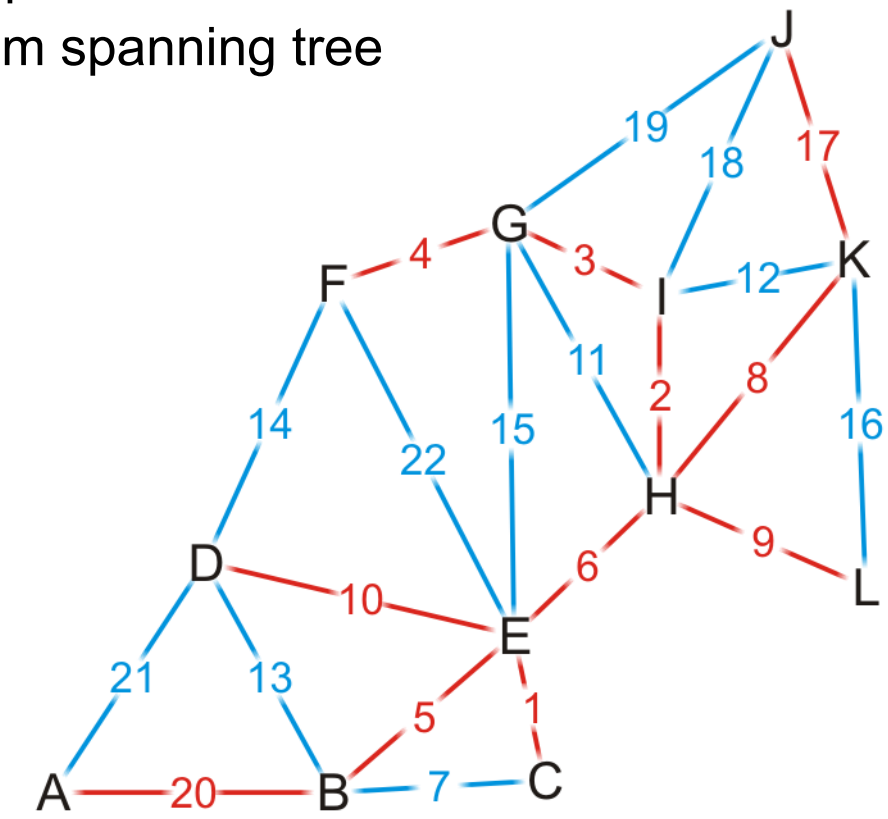


{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
→ {K, L}
→ {J, K}
→ {J, I}
→ {J, G}
→ {A, B}
{A, D}
{E, F}

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 either quicksort or some distribution sort
- To determine if a cycle is created, we could perform a BFS traversal
  - A run-time of $O(|V|)$
- Consequently, the run-time would be $O(|E| \log(|E|) + |E| \cdot |V|)$
- However, $|E| = O(|V|^2)$, so $\log(E) = O(\log(|V|^2)) = O(2 \log(|V|)) = O(\log(|V|))$
- Consequently, the run-time would be $O(|E| \log(|E|) + |E||V|) = O(|E| \cdot |V|)$

The critical operation is determining if two vertices are connected

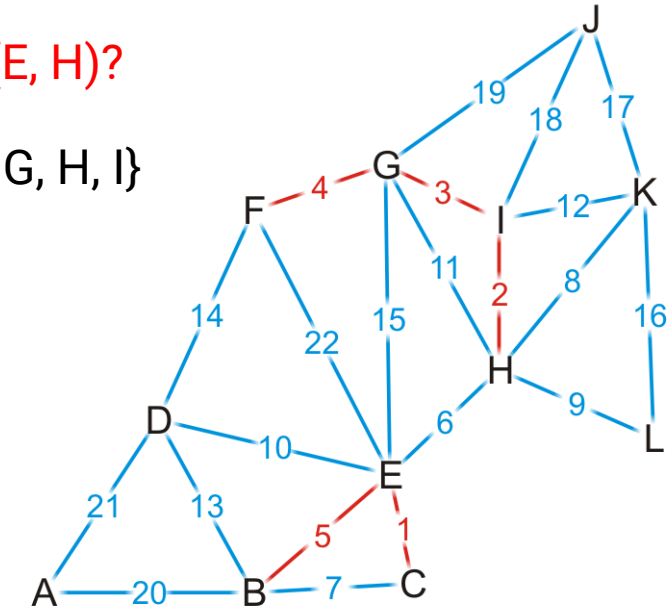
Analysis

Instead, we could use **disjoint sets**

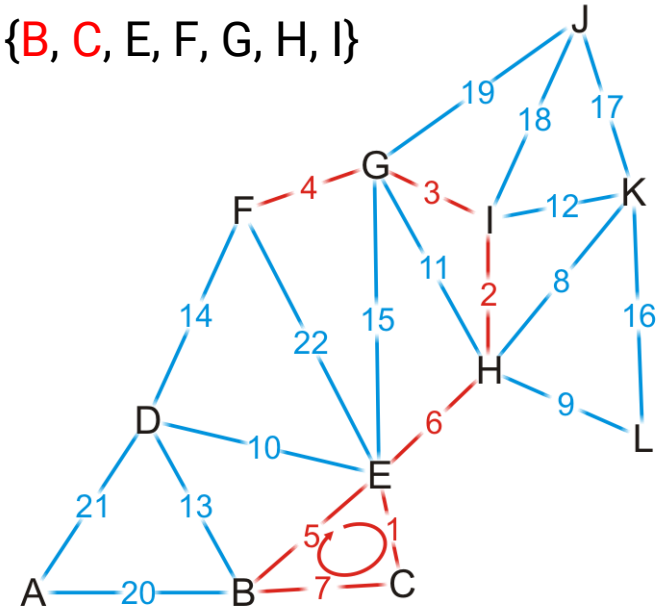
- Consider edges in the same connected sub-graph as forming a set
- If the vertices of the next edge are in different sets, take the union of the two sets
- Do not add an edge if both vertices are in the same set

Add edge (E, H)?

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



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



Analysis

The disjoint set data structure has the following average run-times:

- Checking if two vertices are in the same set is **almost linear**
- Taking the union of two disjoint sets is **almost linear**

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

The dominant time is now the time required to sort the edges:

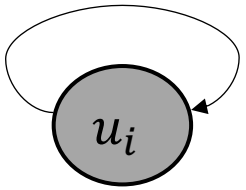
- Using quicksort, the run-time is $O(|E| \log(|E|))$

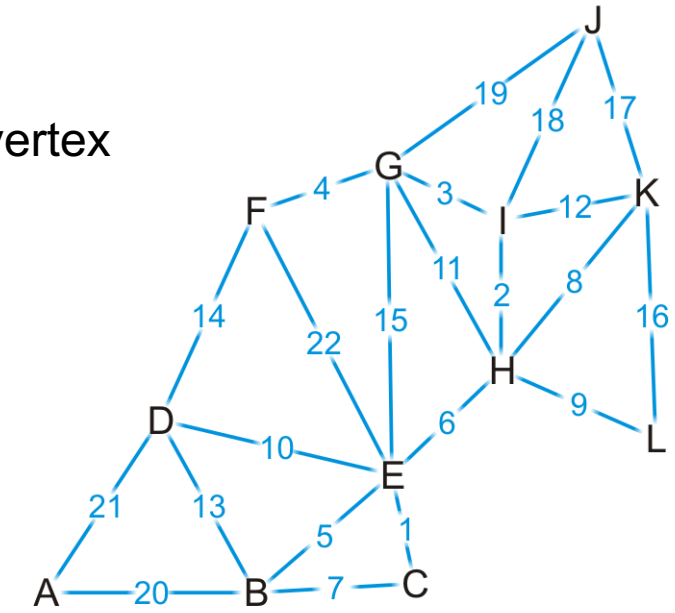
Example

Going through the example again with disjoint sets. We start with twelve singletons

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

Initialization:
For each vertex $u_i \in [A, L]$, each vertex direct to themselves.



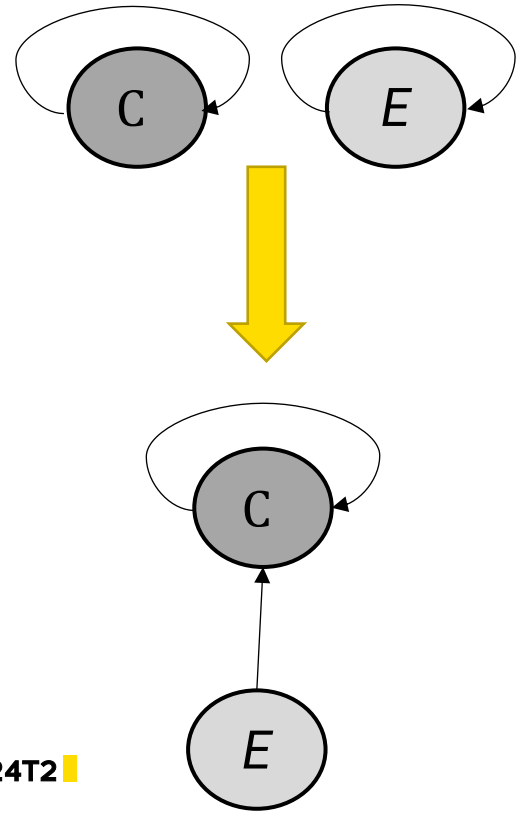


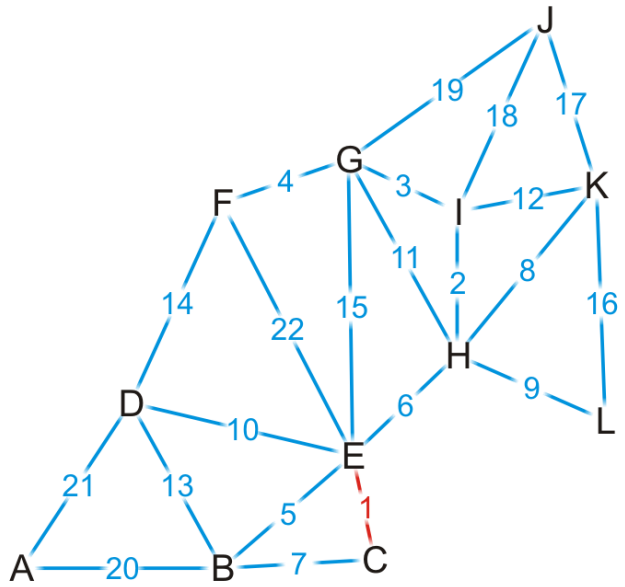
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

We start by adding edge {C, E}

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



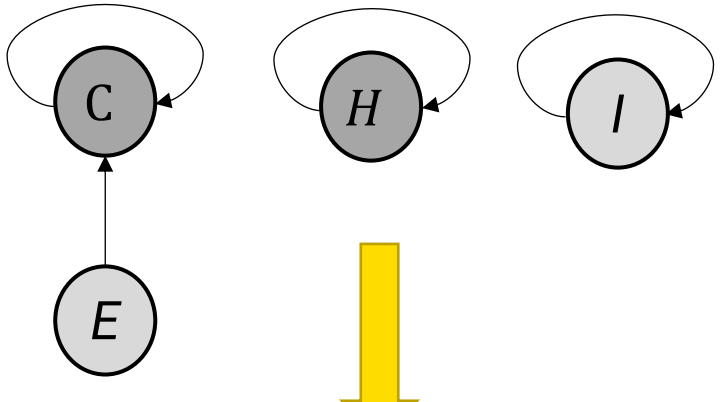


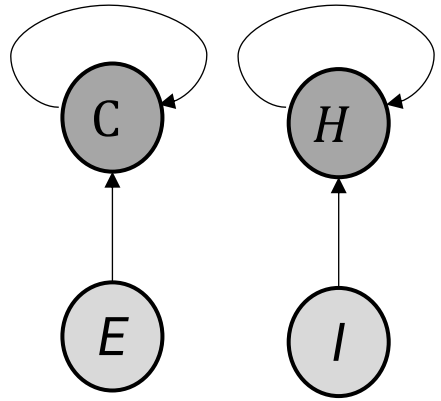
→ {C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

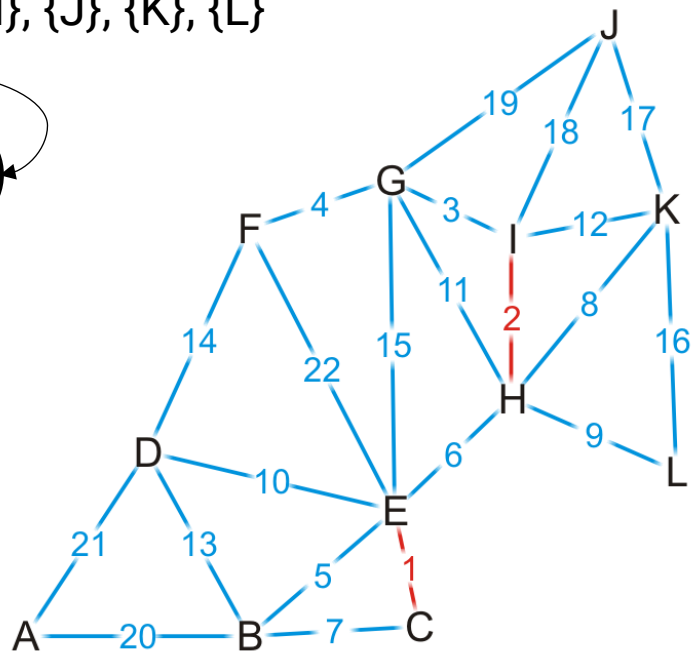
Example

We add edge {H, I}

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







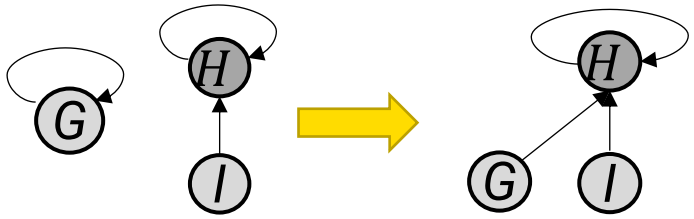
{C, E}
→ {H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

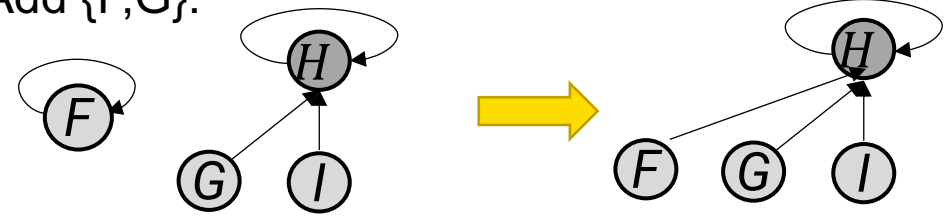
{A}, {B, C, E}, {D}, {F, G, H, I}, {J}, {K}, {L}

Similarly, we add {G, I}, {F, G}, {B, E}

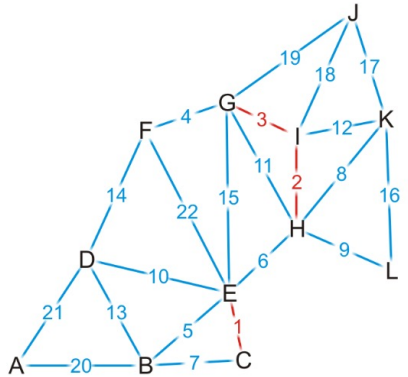
Add {G,I}: According to the rule of Union by Size, make smaller tree point to bigger one's root.



Add {F,G}:



Add {B,E} (Union by Size):



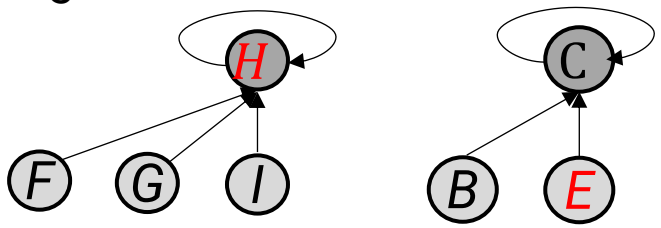
{C, E}
{H, I}
→ {G, I}
→ {F, G}
→ {B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

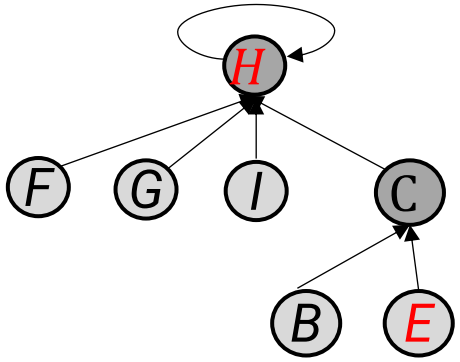
The vertices of {E, H} are in different sets

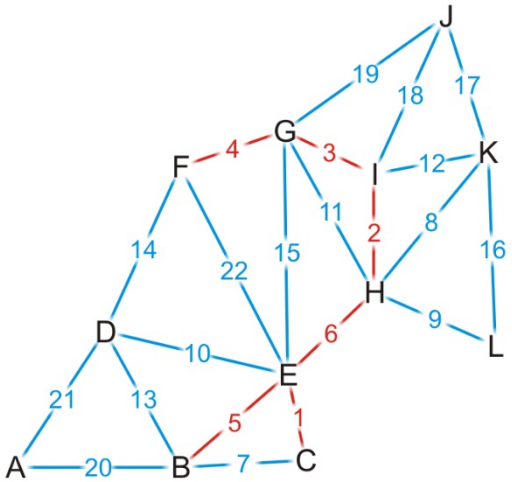
{A}, {B, C, E}, {D}, {F, G, H, I}, {J}, {K}, {L}

Merge these two trees:



Make smaller tree point to bigger one's root (C points to H).



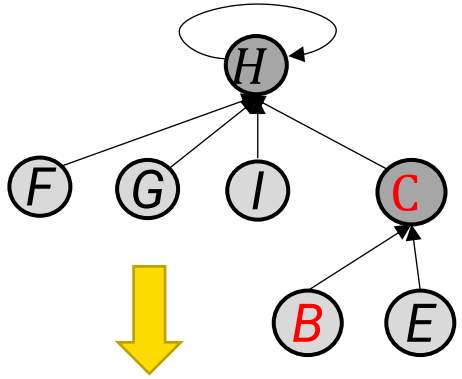


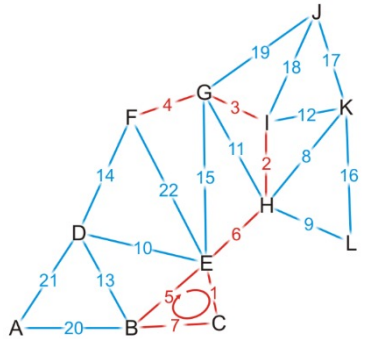
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
→ {E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

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

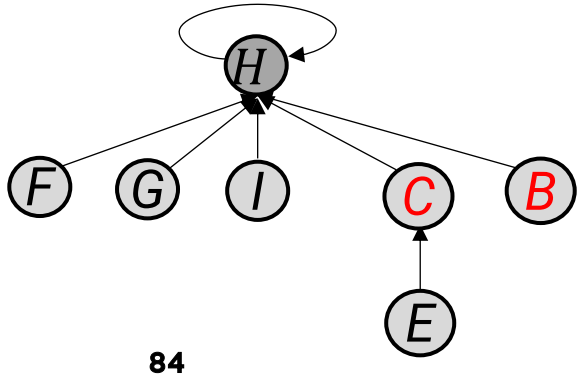
{A}, {B, C, E, F, G, H, I}, {D}, {J}, {K}, {L}





{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}

According to the rule of Path Compression, while finding B, direct B to the root H. Because C is already point to root, the position of C would not change.

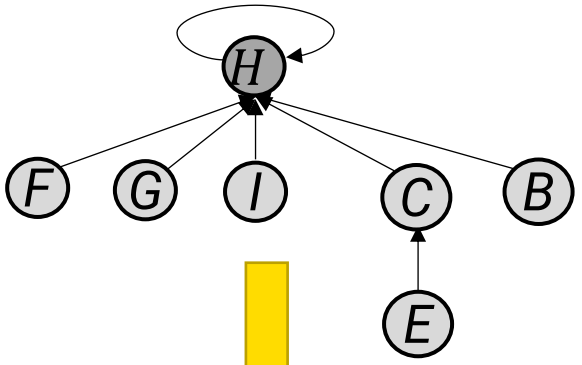


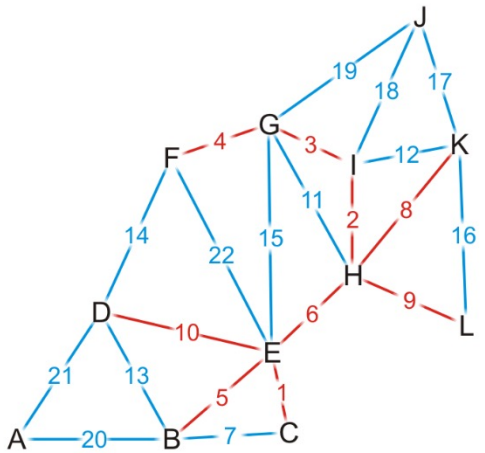
Example

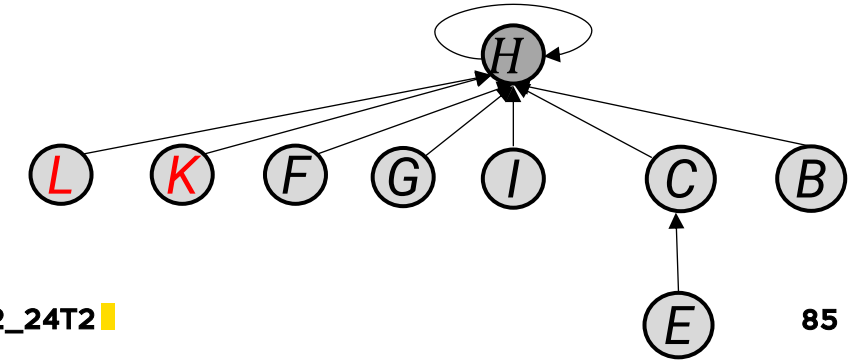
{A}, {B, C, E, F, G, H, I, K, L}, {D}, {J}

We add edge {H, K}, {H, L}

Add edge {H, K}, {H, L}:







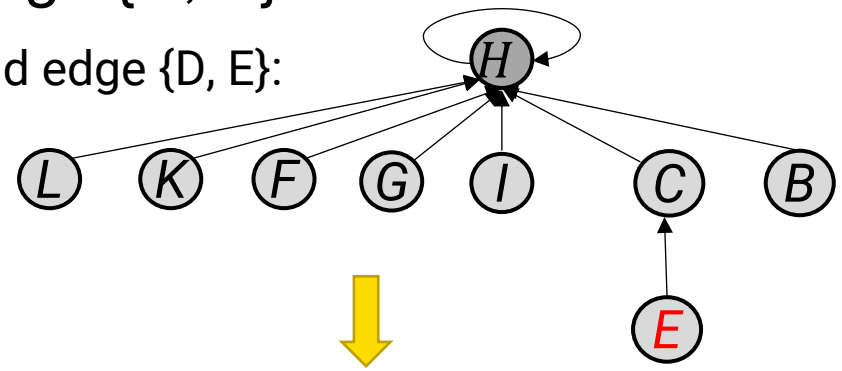
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

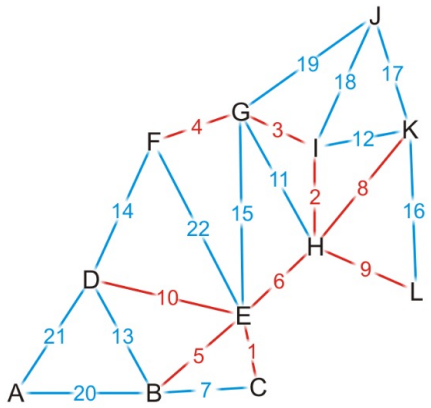
Example

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

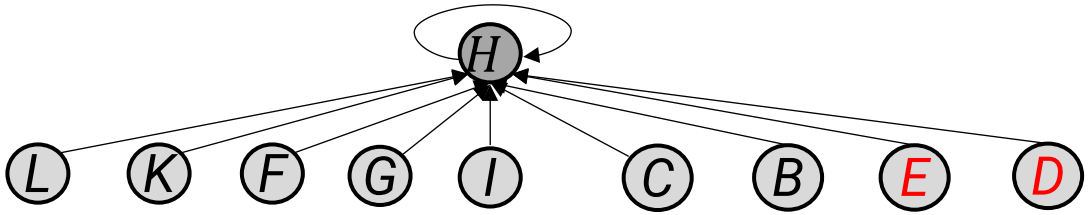
We add edge {D, E}

Add edge {D, E}:





While adding D, according to the rule of Union by Size, make smaller tree D point to the bigger one's root H.
While finding E, according to the rule of Path Compression, direct E to the root H.

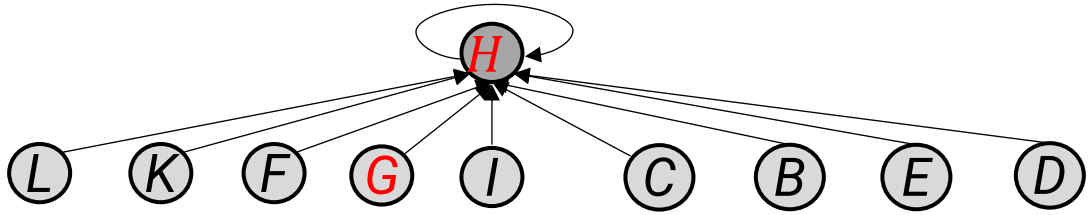


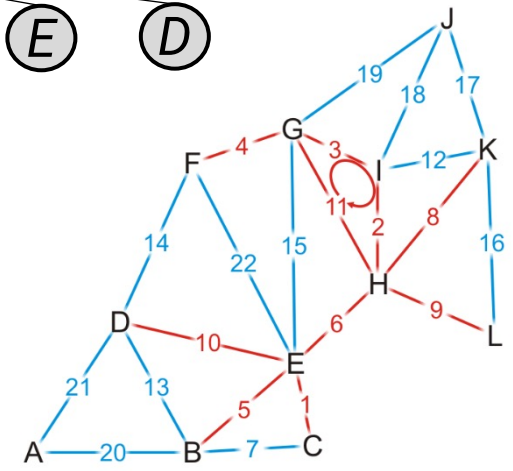
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

Both G and H are in the same set

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



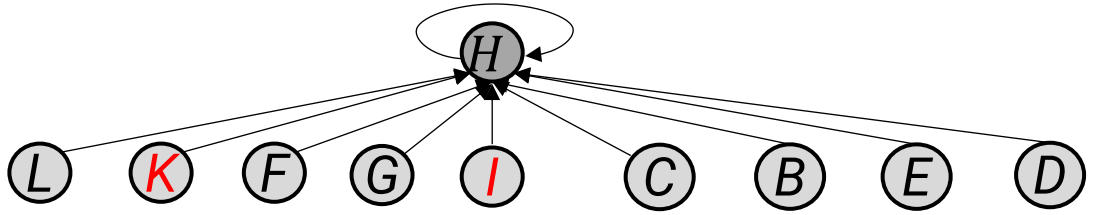


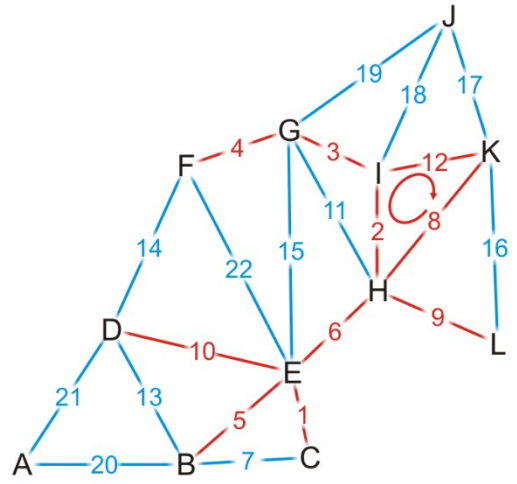
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
→ {G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

Both {I, K} are in the same set

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



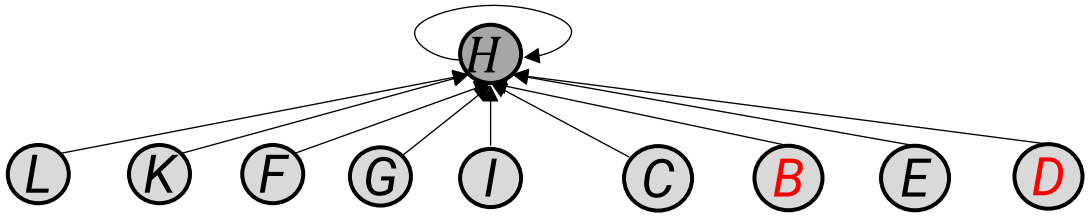


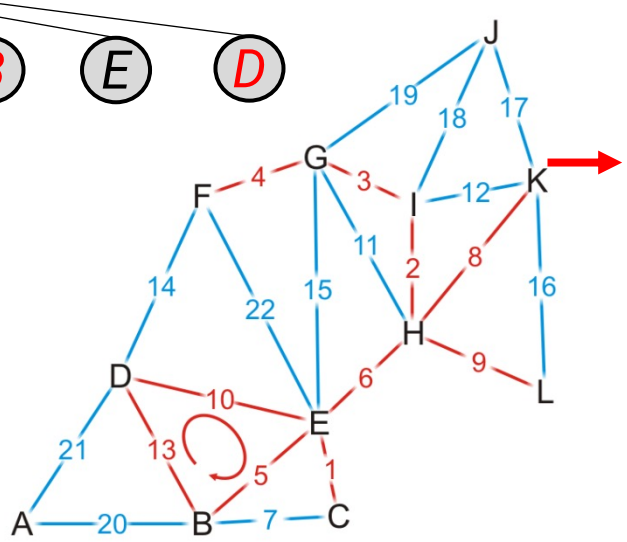
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
→ {I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

Both {B, D} are in the same set

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



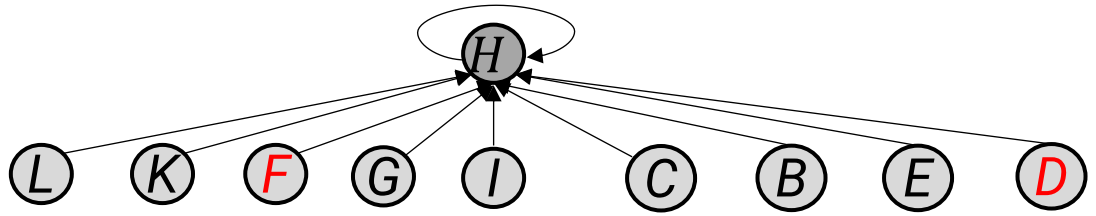


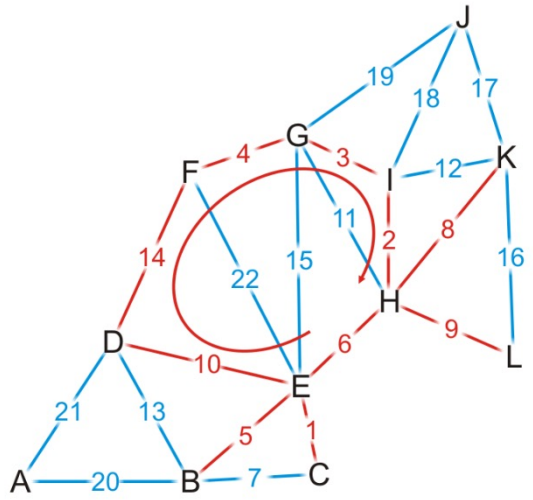
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

Both {D, F} are in the same set

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



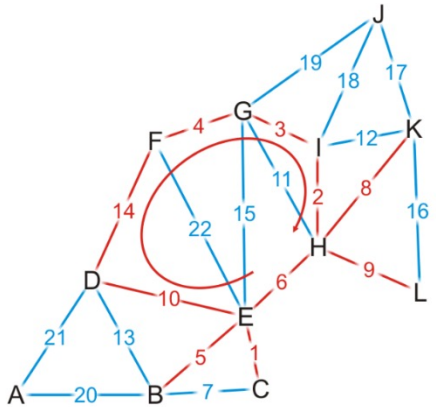


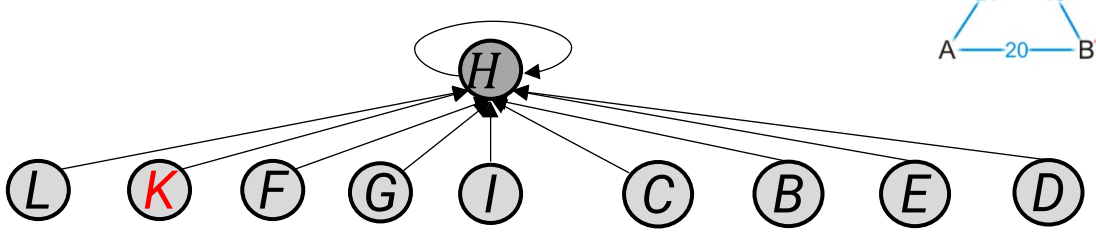
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
{J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

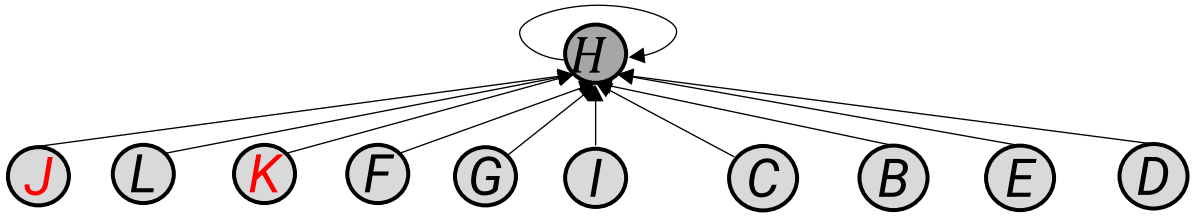
Example

Until adding {J,K}

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



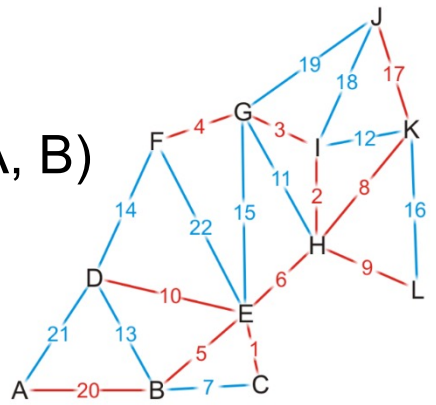




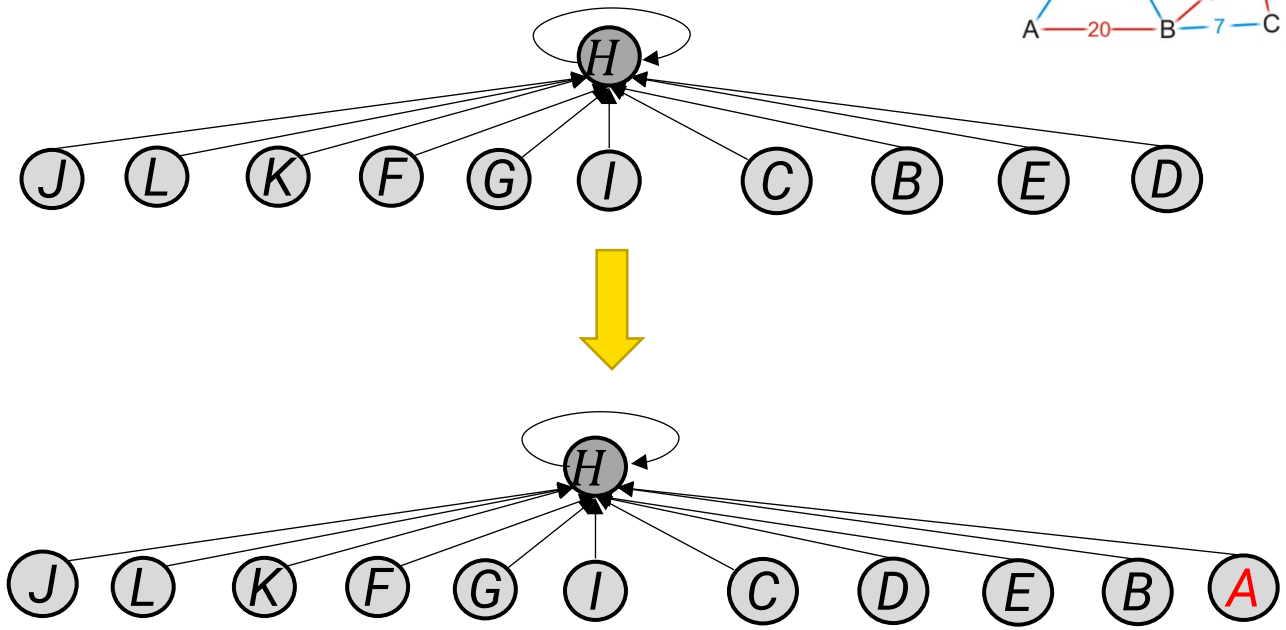
{C, E}
{H, I}
{G, I}
{F, G}
{B, E}
{E, H}
{B, C}
{H, K}
{H, L}
{D, E}
{G, H}
{I, K}
{B, D}
{D, F}
{E, G}
{K, L}
→ {J, K}
{J, I}
{J, G}
{A, B}
{A, D}
{E, F}

Example

We end when there is only one set, having added (A, B)



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



{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}

Summary

This topic has covered Kruskal's algorithm

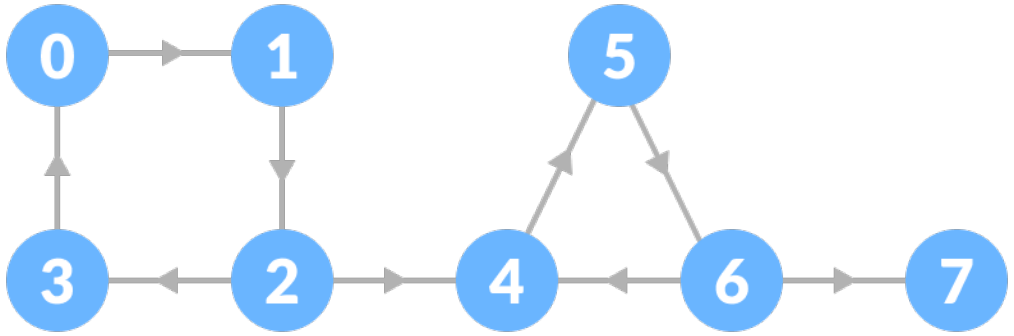
- Sort the edges by weight
- Create a disjoint set of the vertices
- Begin adding the edges one-by-one checking to ensure no cycles are introduced
- The result is a minimum spanning tree
- The run time is $\mathrm{O}(|E| \log(|E|))$

Strongly Connected Component

Strongly Connected Component (SCC)

Given a directed graph, a strongly connected component is a (maximal) subgraph in which there is a path from each vertex to another vertex.

There are three strongly connected components in this example.



Example credit: https://www.programiz.com/dsa/strongly-connected-components

Computing SCC

Optional

- Kosaraju's Algorithm
  - Two DFS

- **Tarjans's Algorithm**
  - One DFS

Motivation

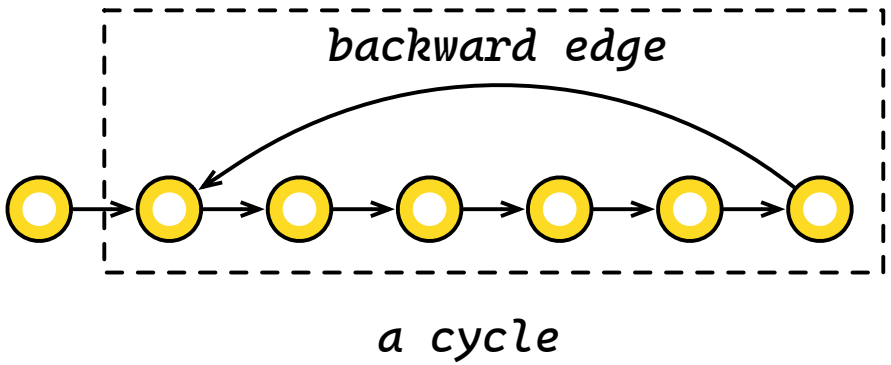
Optional

Cycle <=> SCC

Search in a graph

Identify a backward edge => Find a cycle

Mark certain ancestors in the cycle

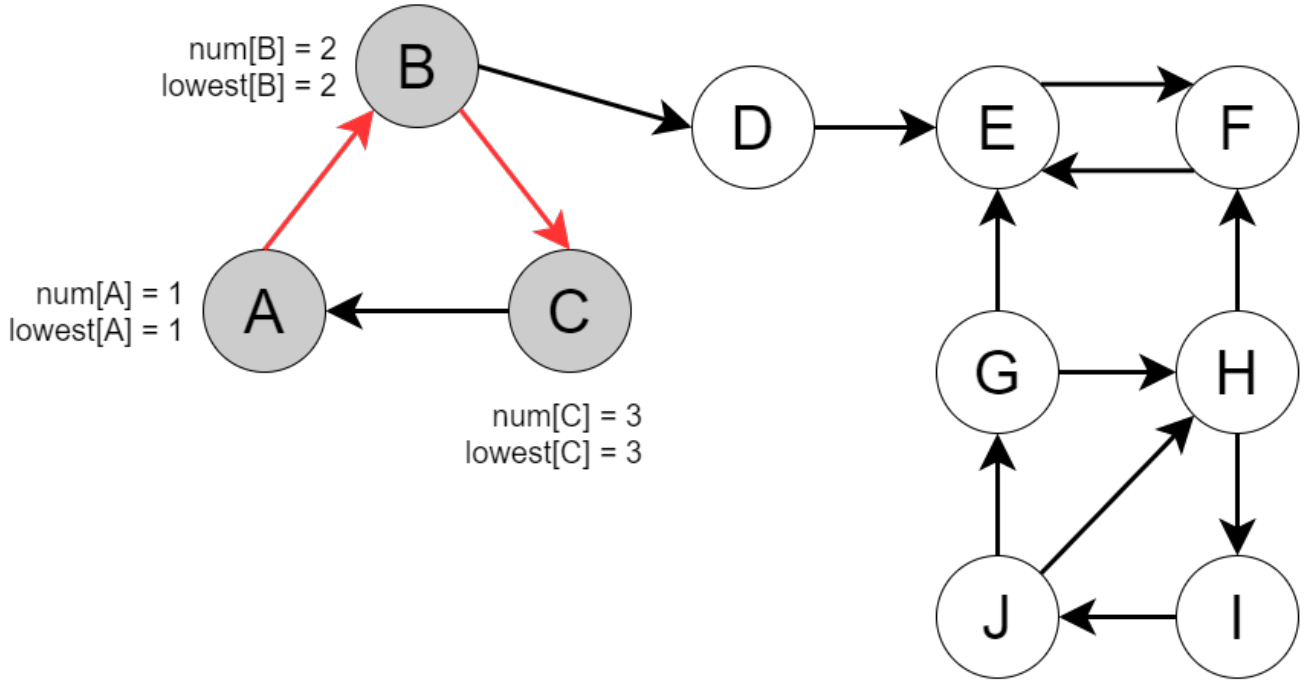


Running Example (1)

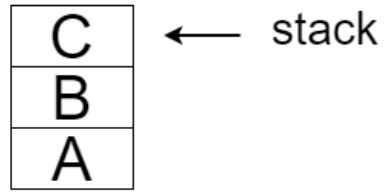
Optional

num: the vertex counter

lowest: the lowest num that a vertex can reach



Process back edge (C,A), update lowest[C], backtrack and update lowest[B]

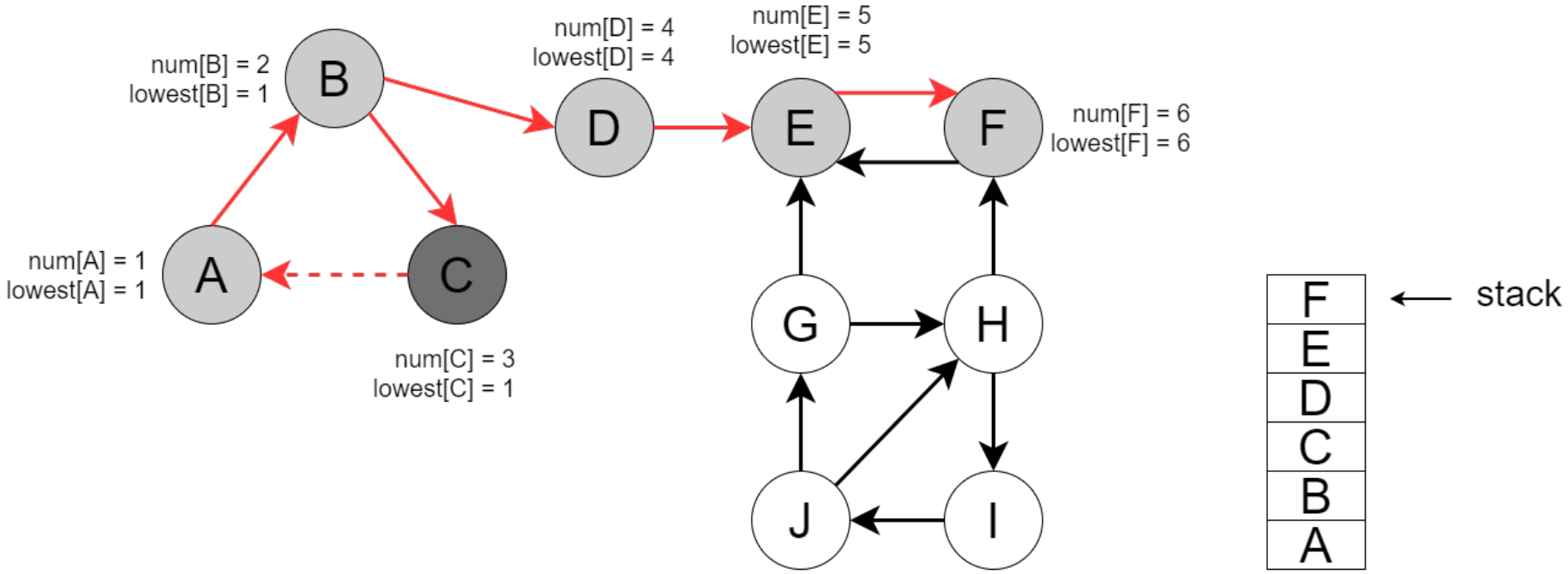


Example credit: https://www.baeldung.com/cs/scc-tarjans-algorithm

Running Example (2)

Optional

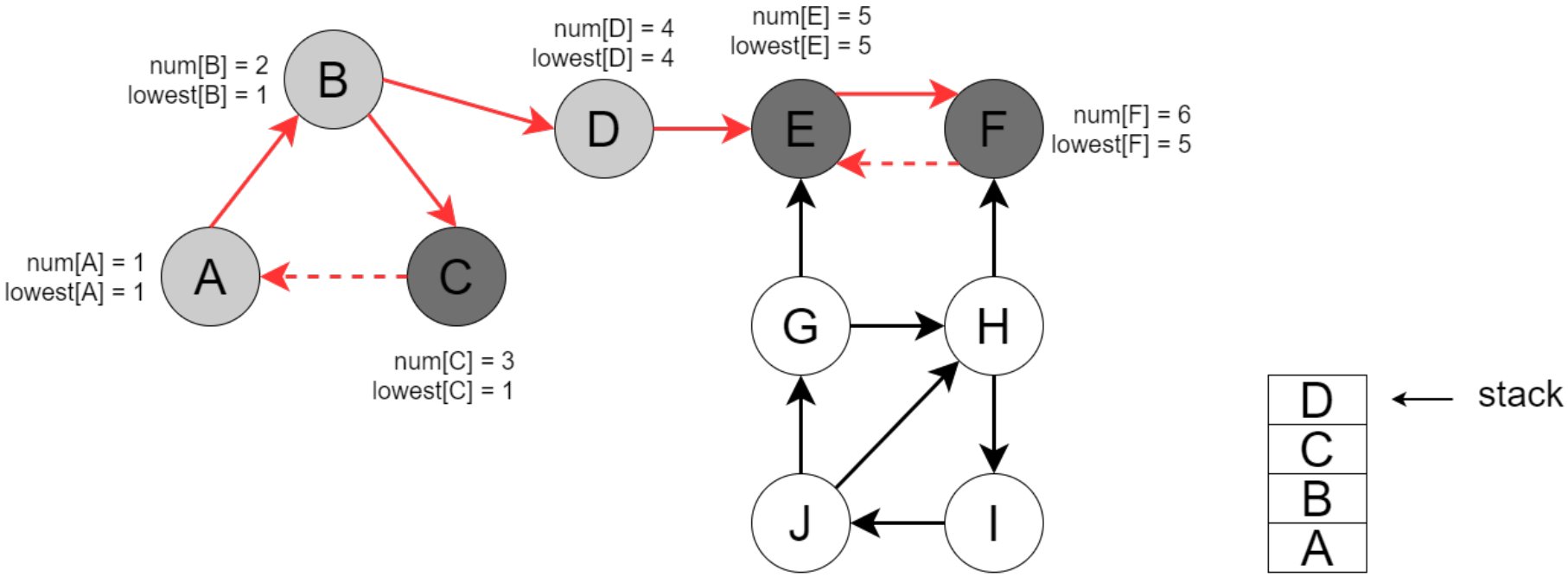
- Continue DFS search
- Process the back edge (F,E), identify an SCC {E,F}



Running Example (3)

Optional

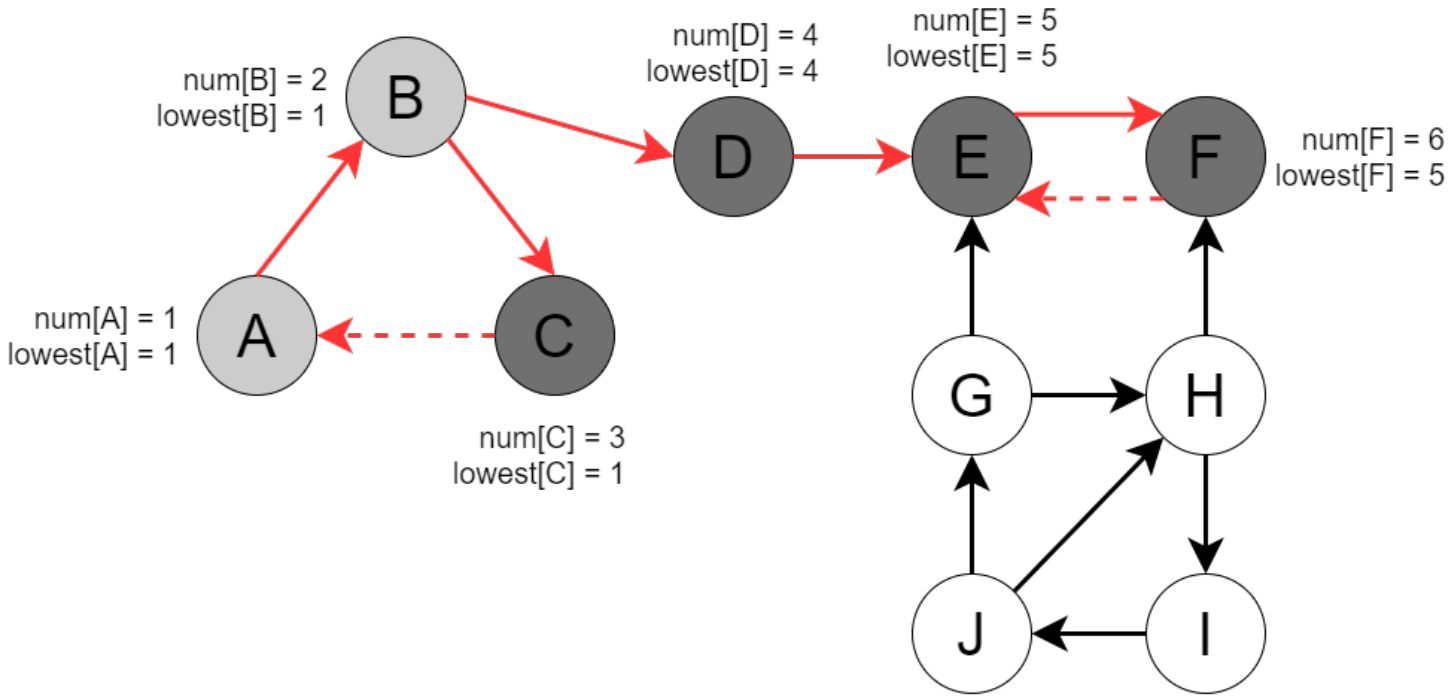
DFS backtracks to D, and an SCC {D} is found

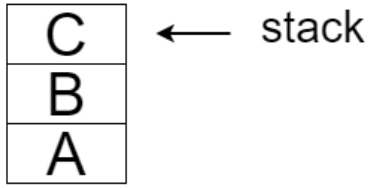


Running Example (4)

Optional

DFS backtracks to B, then to A, find the SCC {A,B,C}

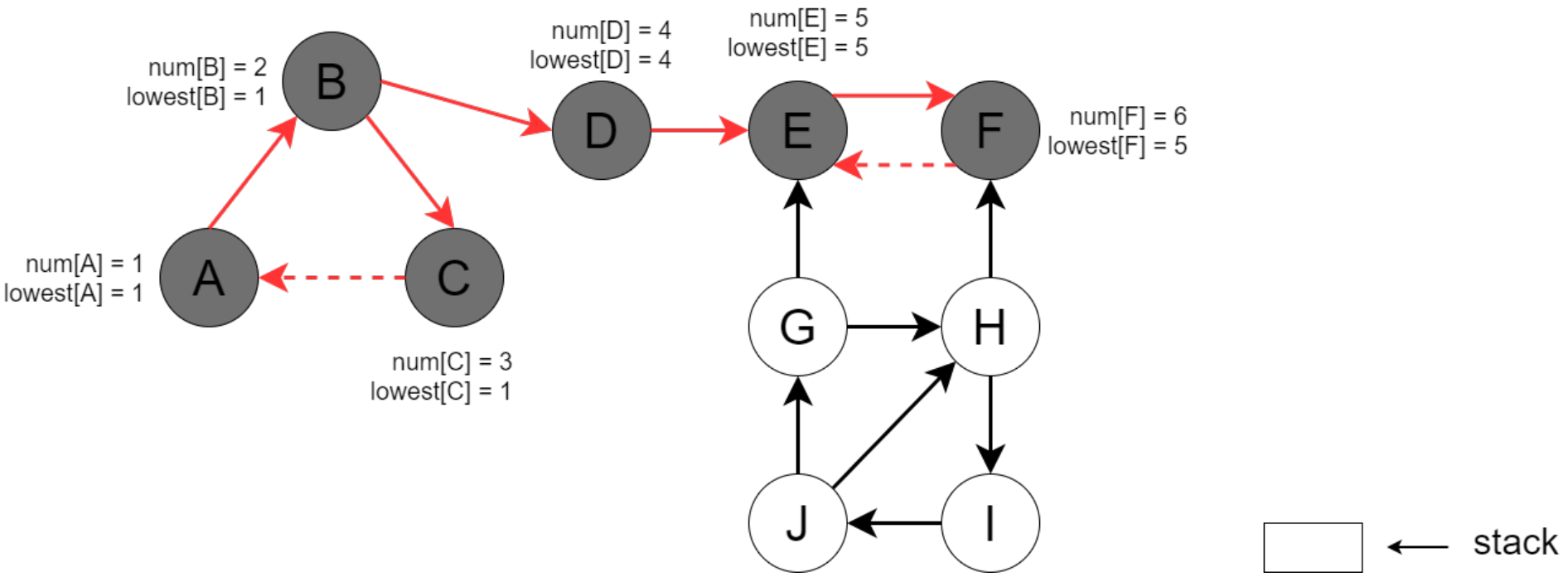




Running Example (5)

Optional

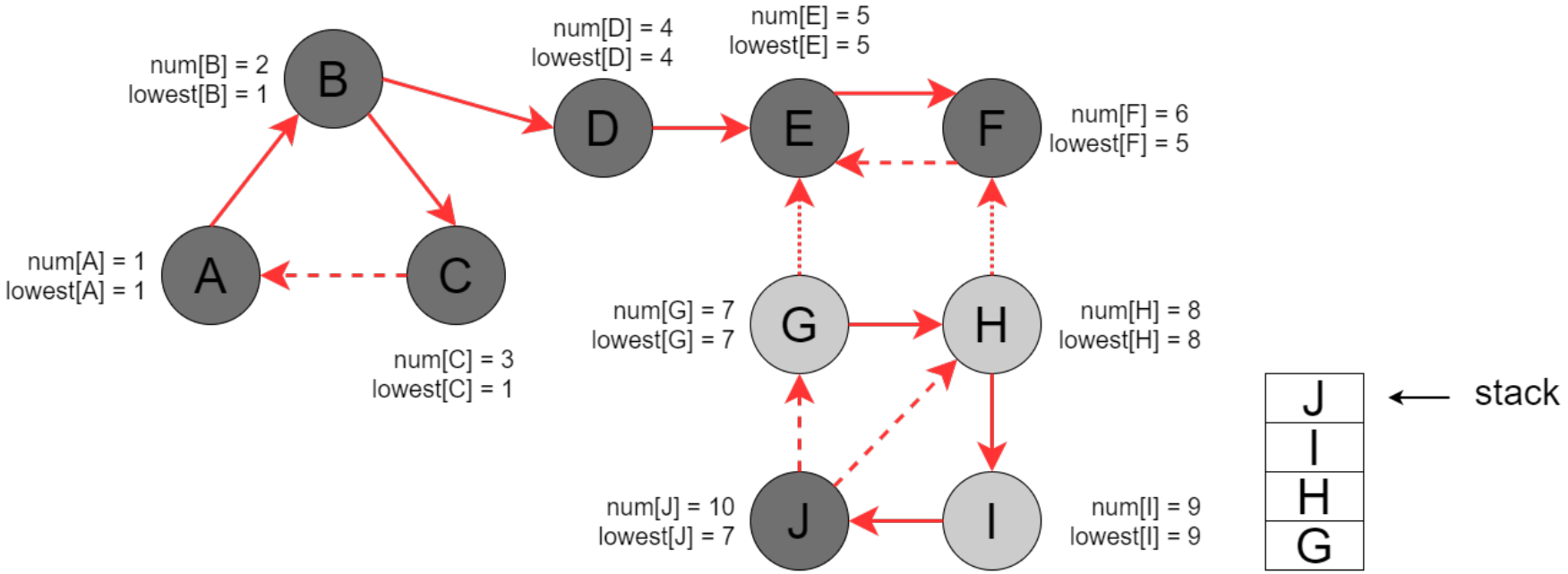
No reachable vertices are left



Running Example (6)

Optional

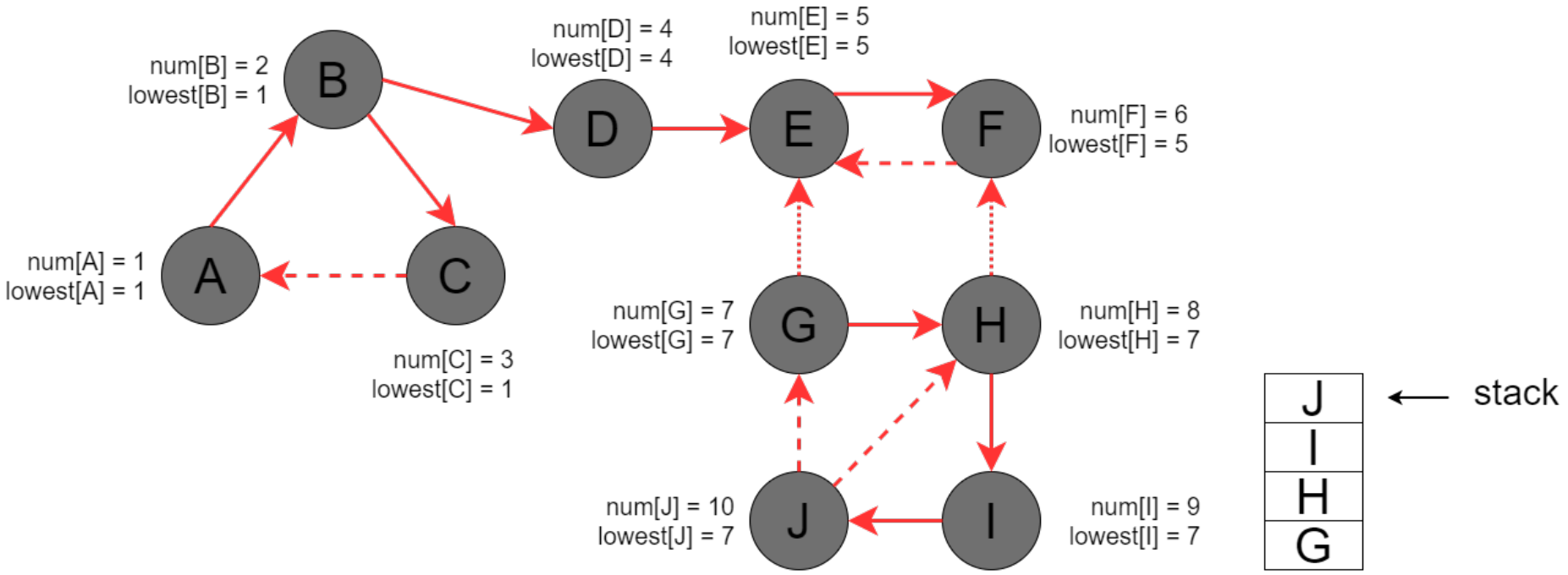
Run DFS from an unvisited vertex G



Running Example (7)

Optional

Process back edge from J, update lowest[J] and backtrack.

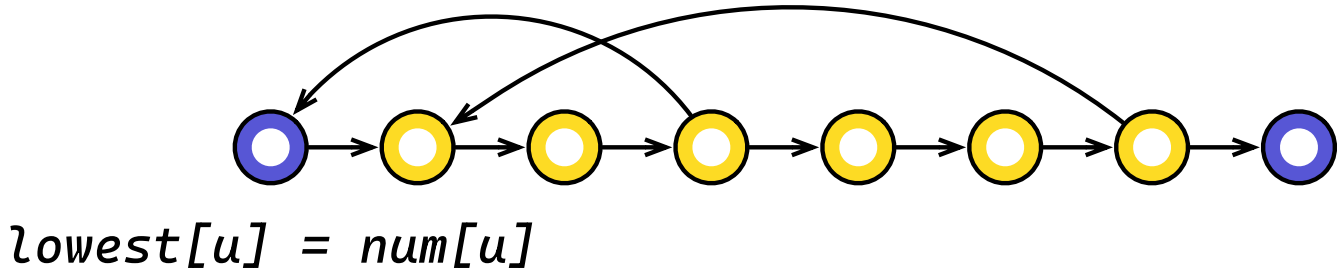


Code Review

Optional

Key step: pop stack when lowest[u] = num[u]

See the provided code for details



Learning Outcome

- Know the definition of spanning tree and minimum spanning tree

- Understand the algorithms to compute the minimum spanning tree (Prim's algorithm, Kruskal's algorithm with the Union-Find structure)