Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Data Structures and Algorithms
★ Graph Advance ★

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io

SCAN ME

Week by week topics (*)

1. Overview, DSA, OOP and Java
2. Arrays
3. Sorting
4. Queue, Stack
5. List
6. Recursion

Mid-Term

7. Advanced Sorting
8. Binary Tree
9. Hash Table
10. Graphs
11. Graphs Adv.

Final-Exam

10 LABS

Objectives

- Spanning Trees
  - Prim algorithm
  - Kruskal algorithm
- Eulerian and Hamilton Graphs

Revisit some definitions

- An acyclic graph is a graph with no cycles.
- A tree is an acyclic connected graph.
- A spanning tree of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree.
- A **minimum spanning tree (MST)** of an edge-weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree.

Minimum Spanning Tree (MST)

- Suppose we wish to connect all the computers in a new office building using the least amount of cable. We can model this problem using an undirected, weighted graph G
- Vertices: V = computers
- Edges: E = all the possible pairs (u,v) of computers
- Weight w(u,v) of edge (u,v) = cable length needed to connect u to v.
- Find a tree T that contains all the vertices of G and has the minimum total weight over all such trees: **Minimum Spanning Tree problem**.

Typical MST applications

Application	Vertex	Edge
Circuit	Component	Wire
Airline	Airport	Flight route
Power distribution	Power plant	Transmission line
Image analysis	Features	Proximity relationship

- Minimum Spanning Trees (MST) are useful in many applications, for example, finding the shortest total connections for a set of edges.
- If we are running cable to the nodes, representing cities, and we wish to minimize cable cost, the MST would be a viable option.

Minimum Spanning Tree (MST)

- Two classical algorithms for computing MSTs
- Prim's algorithm
- Kruskal's algorithm

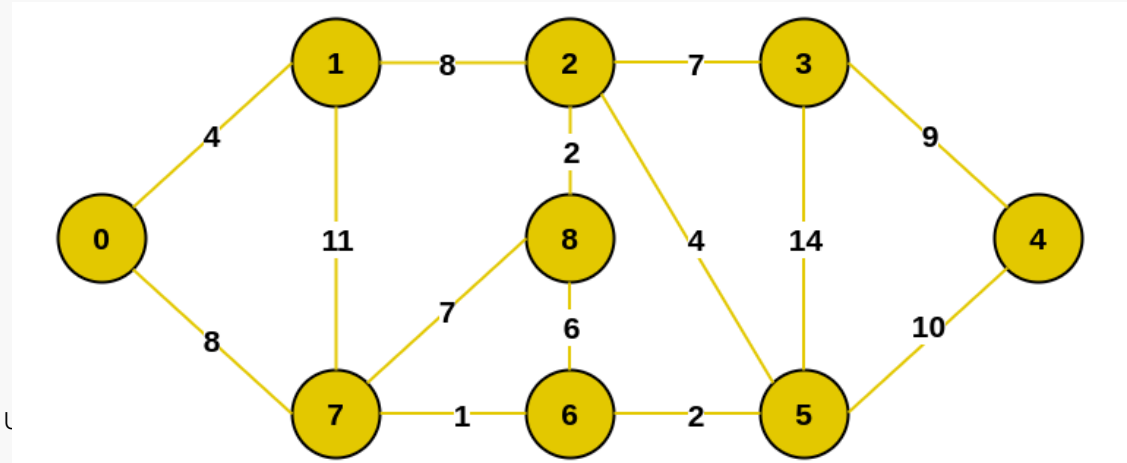
Prim’s algorithm

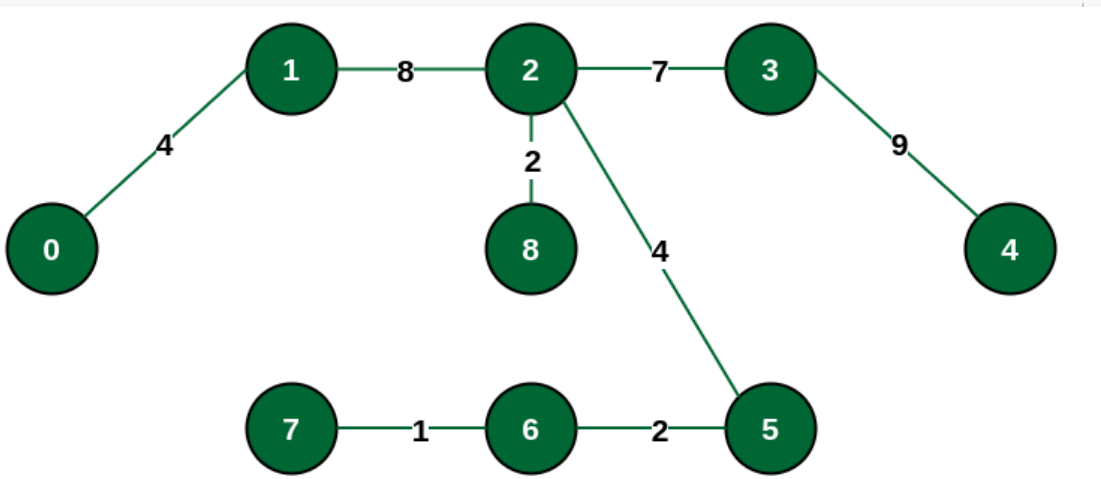
Prim's algorithm - idea

- The algorithm starts with an empty spanning tree.
- Maintain two sets of vertices.
- The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included.
- At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges.
- After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Prim's algorithm - Steps

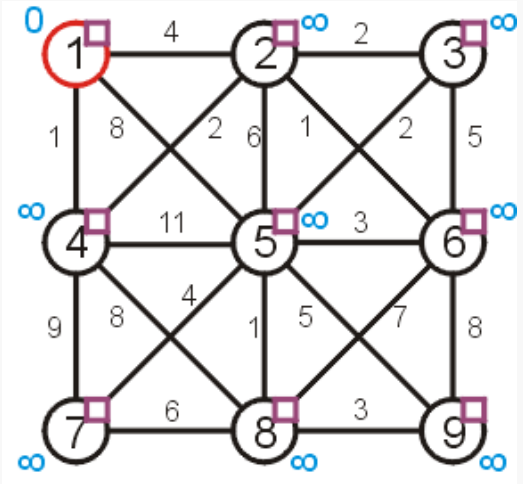
- Step 1: Determine an arbitrary vertex as the starting vertex of the MST.
- Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
- Step 3: Find edges connecting any tree vertex with the fringe vertices.
- Step 4: Find the minimum among these edges.
- Step 5: Add the chosen edge to the MST if it does not form any cycle.
- Step 6: Return the MST and exit





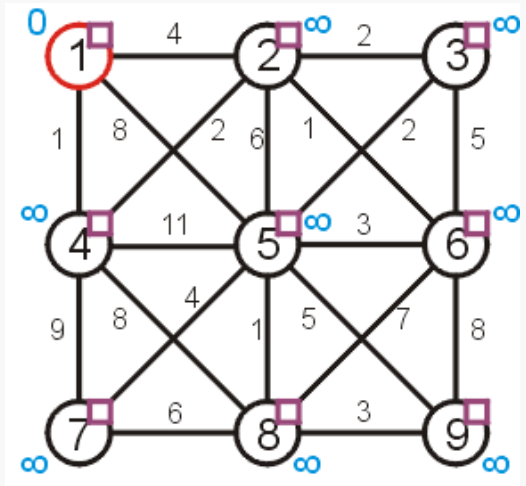
Prim's Algorithm

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



Prim's Algorithm

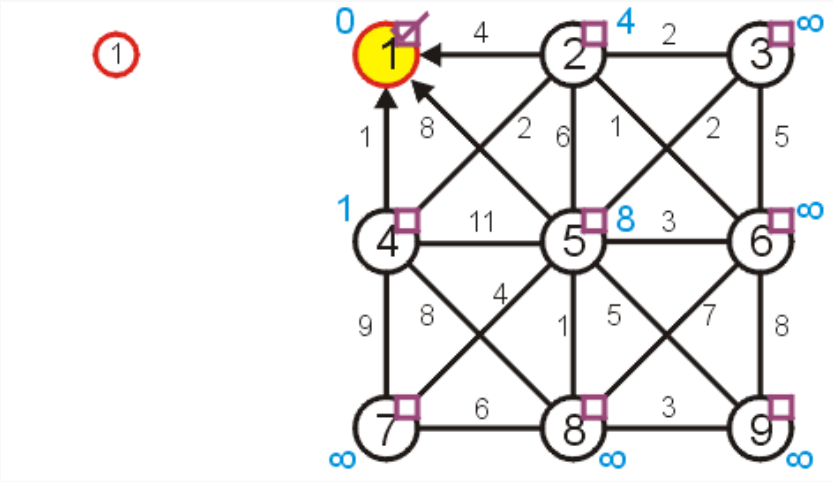
- First we set up the appropriate table and initialize it



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

Prim's Algorithm

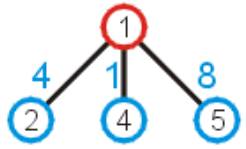
- Visiting vertex 1, we update vertices 2, 4, and 5



		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

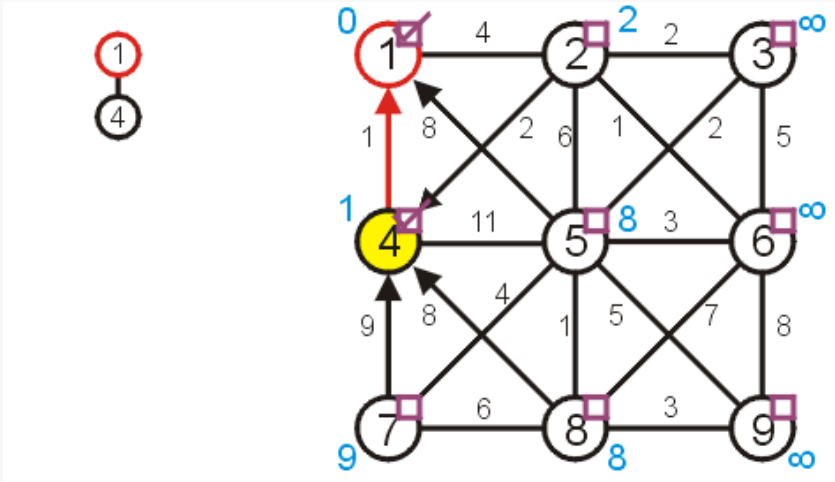
Prim's Algorithm

- 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

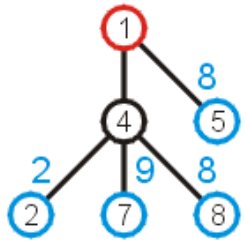
- The next unvisited vertex with minimum distance is vertex 4
  - Update vertices 2, 7, 8
  - Don't update vertex 5



		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

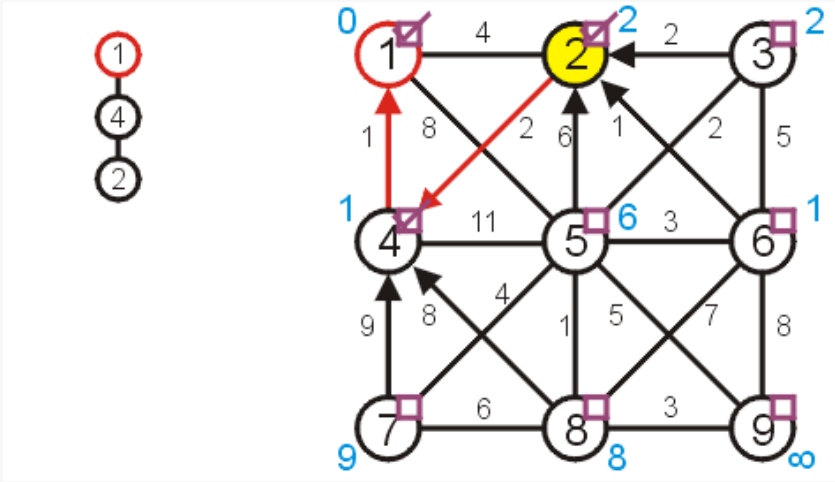
Prim's Algorithm

- 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

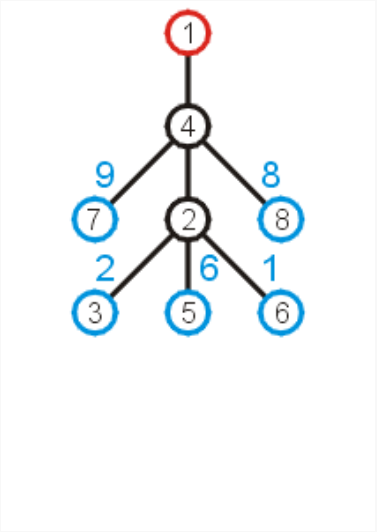
- Next visit vertex 2
- Update 3, 5, and 6



		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

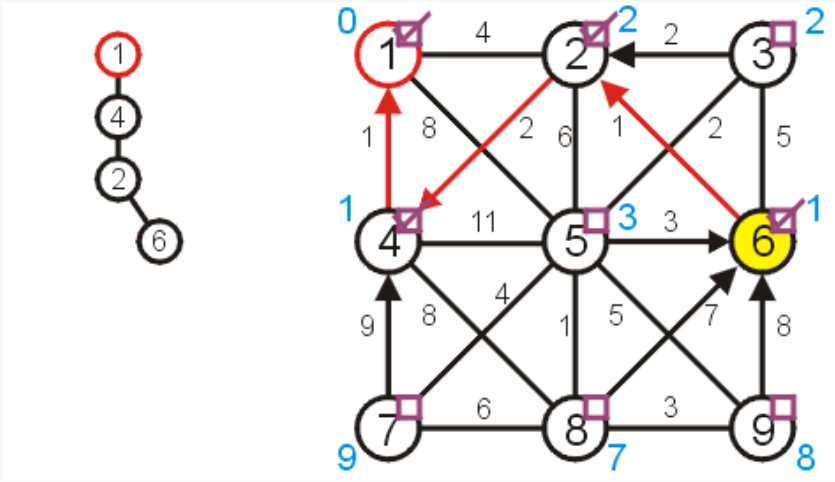
Prim's Algorithm

- 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

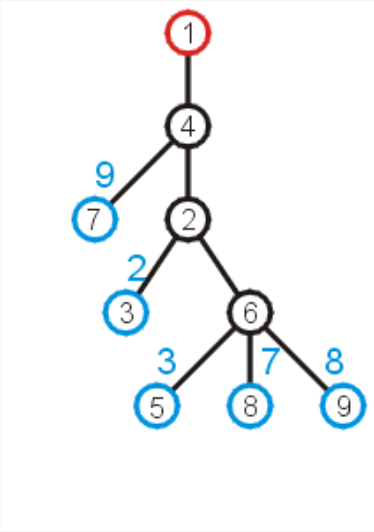
- Next, we visit vertex 6:
- update vertices 5, 8, and 9



		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

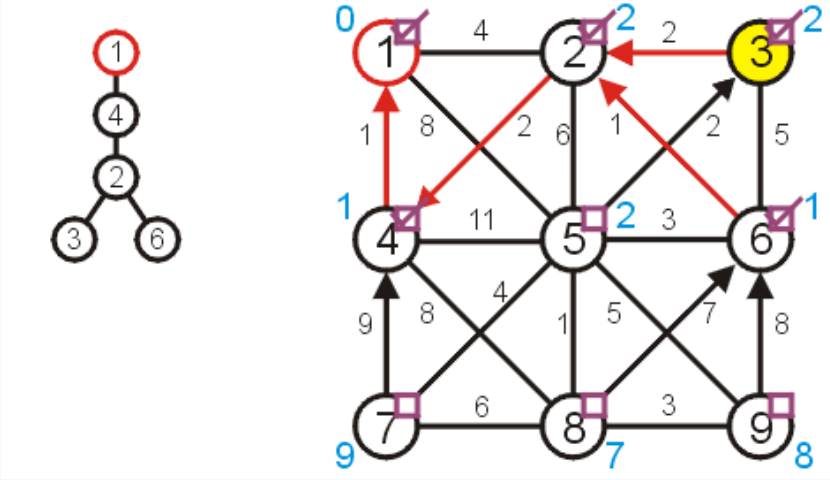
Prim's Algorithm

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



Prim's Algorithm

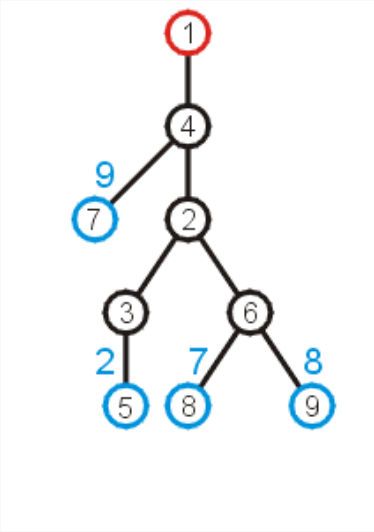
- Next, we visit vertex 3 and update 5



		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

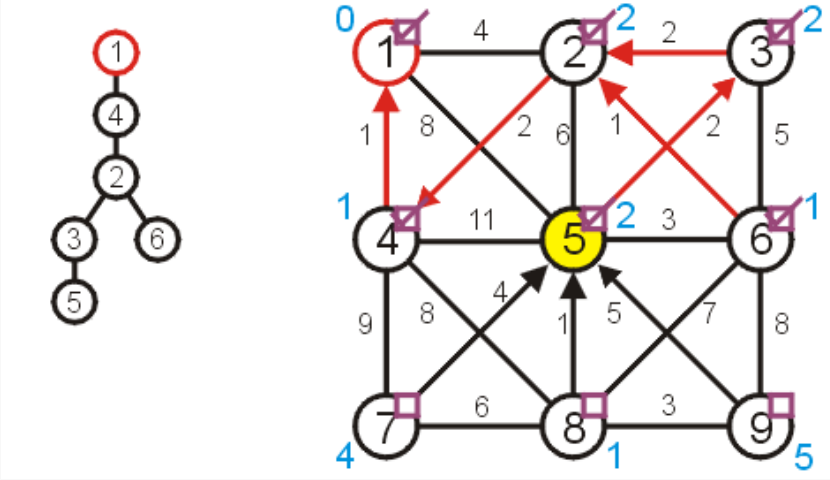
Prim's Algorithm

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



Prim's Algorithm

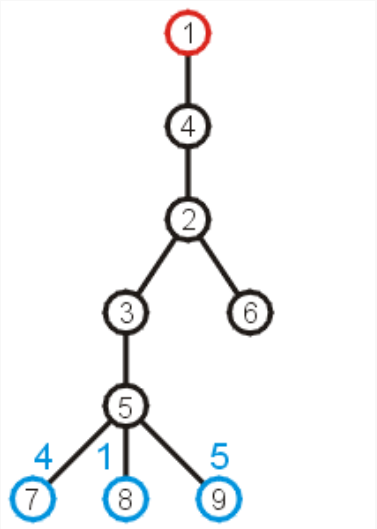
- Visiting vertex 5, we update 7, 8, 9



		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

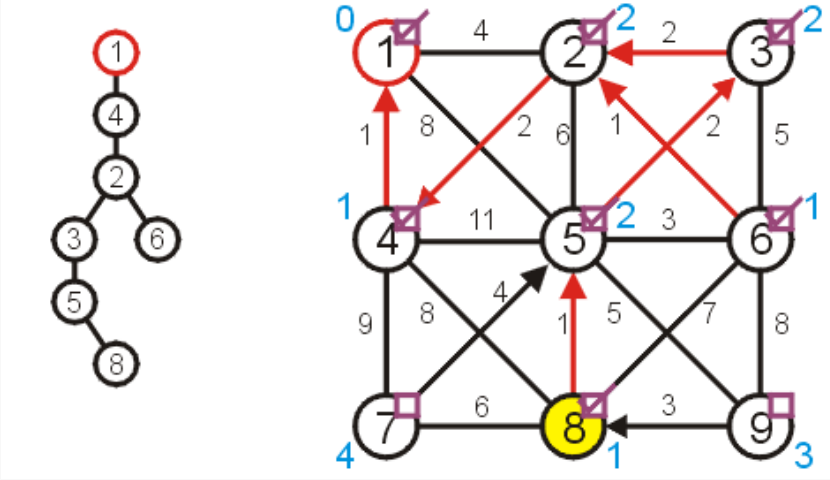
Prim's Algorithm

- 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

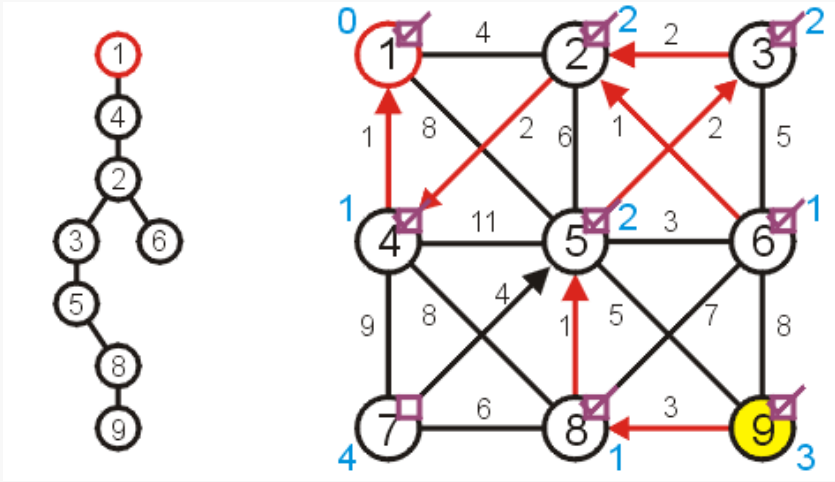
- Visiting vertex 8, we only update vertex 9



		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

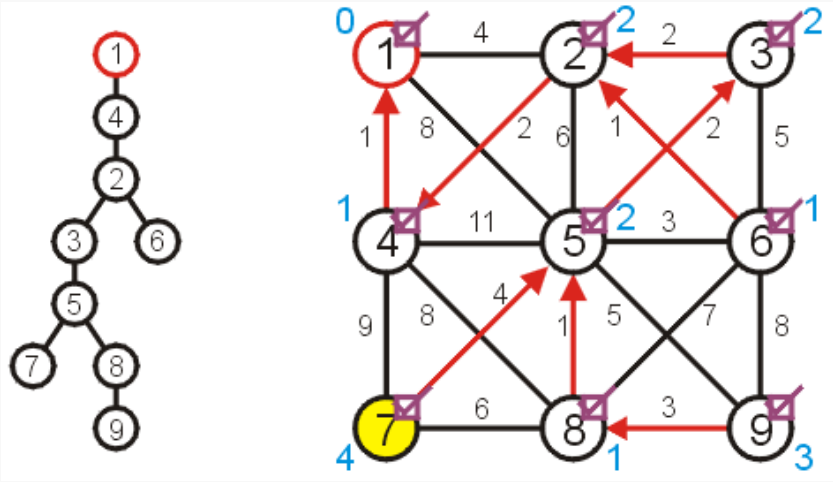
- There are no other vertices to update while visiting vertex 9



		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



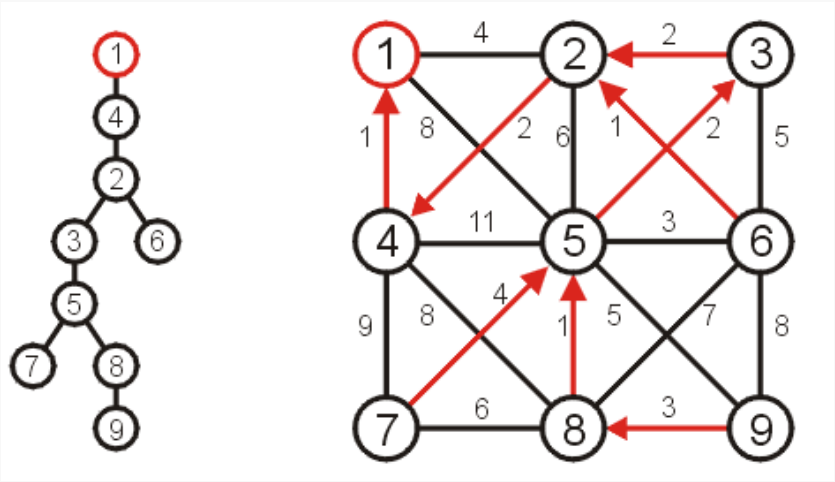
		Distance	Parent
1	T	0	0
2	T	2	4
3	T	2	2
4	T	1	1
5	T	2	3
6	T	1	2
7	T	4	5
8	T	1	5
9	T	3	8

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
- Codes:
  - Textbooks
  - https://www.javatpoint.com/prims-algorithm-java
  - https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/
  - https://www.javatpoint.com/prims-algorithm-java
  - …

Complexity of Prim's algorithm

- 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 neighbours
  - 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 $|E| = O(|V|^2)$

Kruskal's algorithm

Kruskal's algorithm - idea

- Sort all edges of the given graph in increasing order.
- Keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle.
- Picks the minimum weighted edge at first and the maximum weighted edge at last.

Kruskal's Algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
    i. If the cycle is not formed, include this edge.
    ii. Else, discard it.
3. Repeat step #2 until there are (V-1) edges in the spanning tree.

Example

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

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

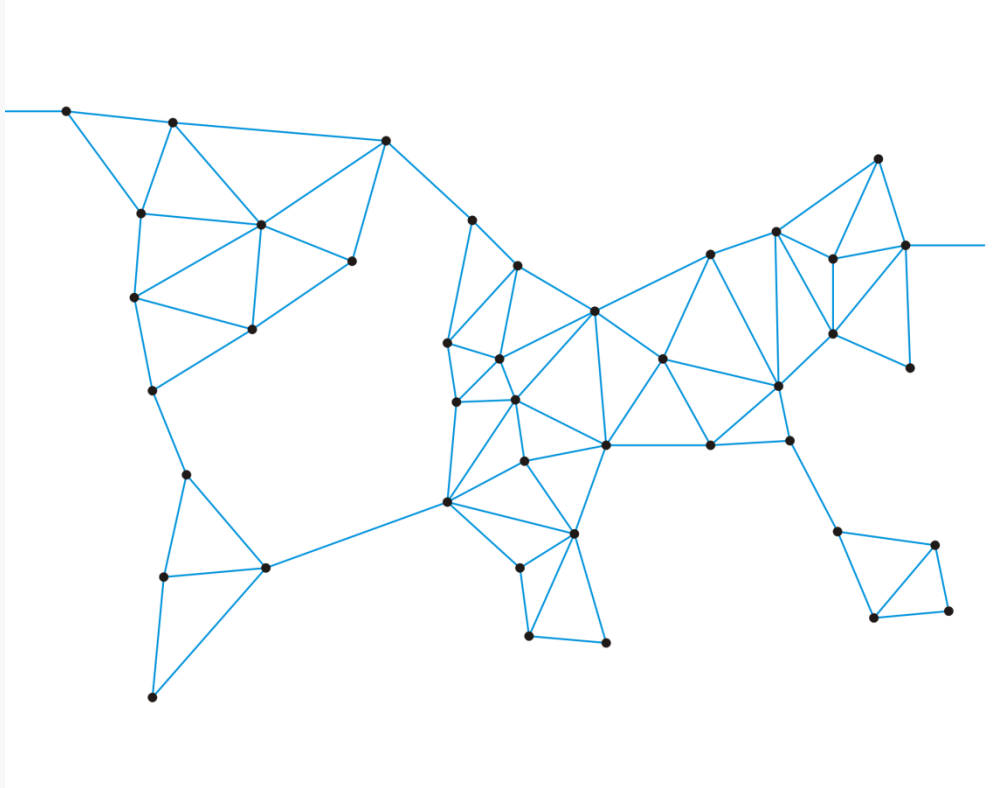
Example

- 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/

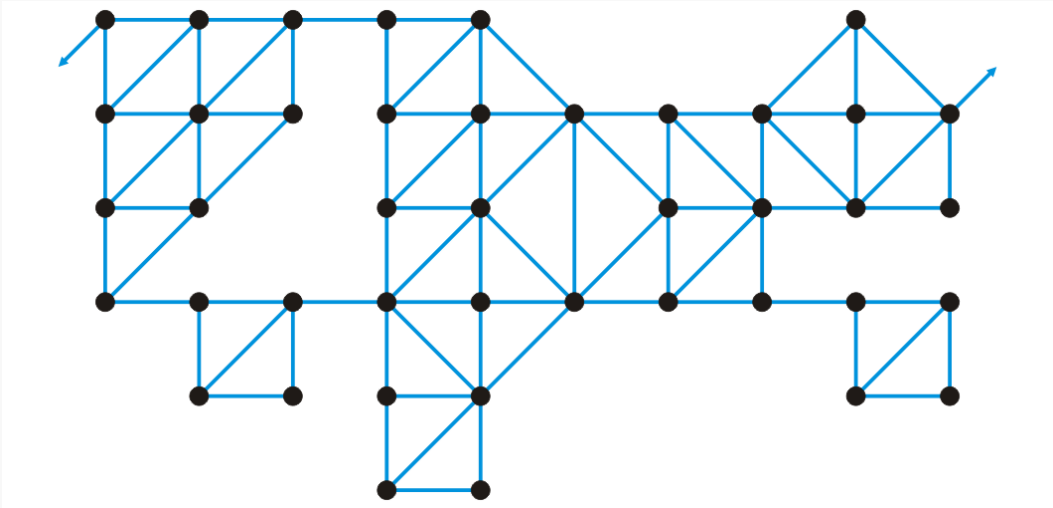
Example

- 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



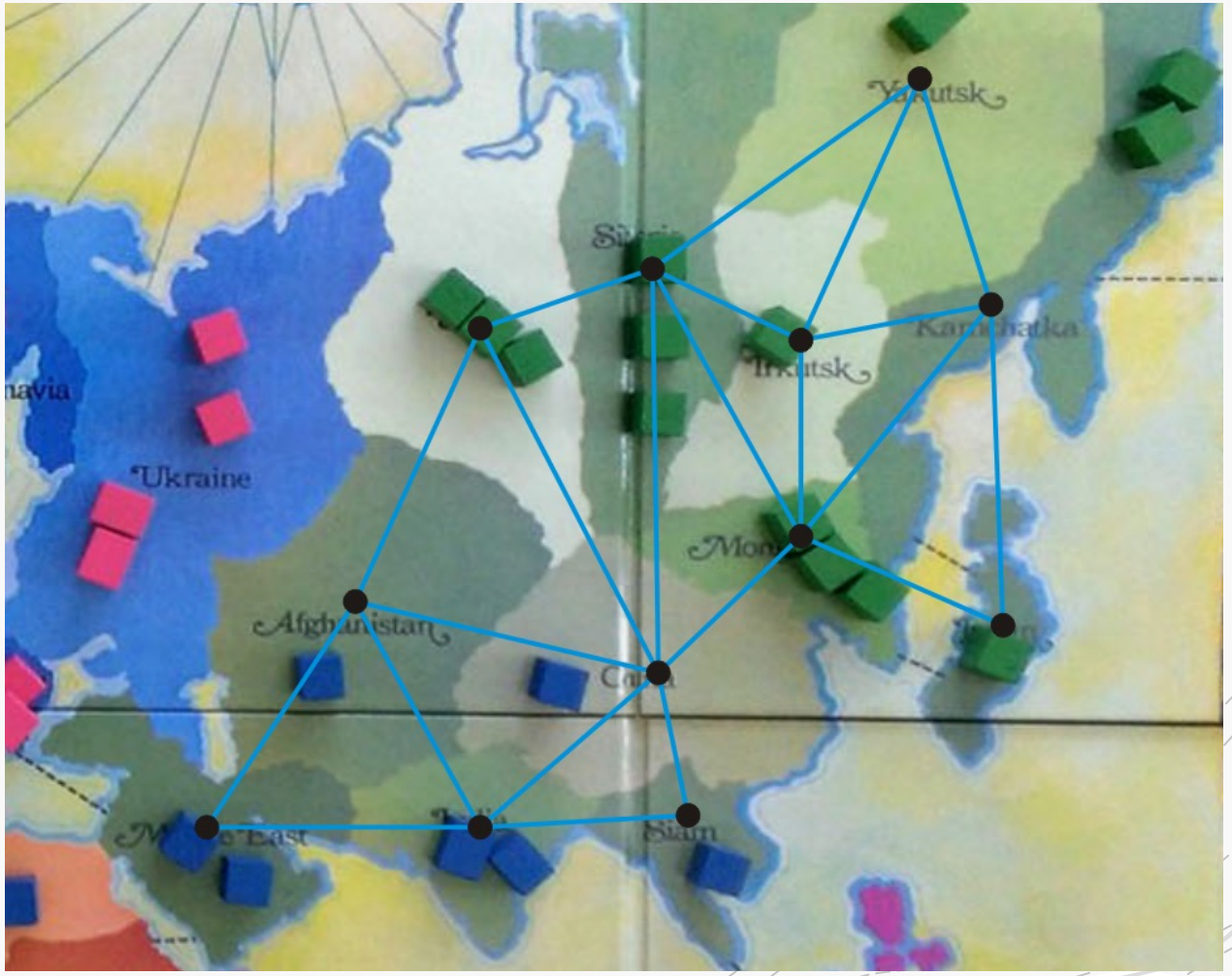
Example

- Consider the game of Risk from Parker Brothers
  - Here is a more abstract representation of the game board
  - Suddenly, it's less interesting: "I've conquered the graph!"



Example

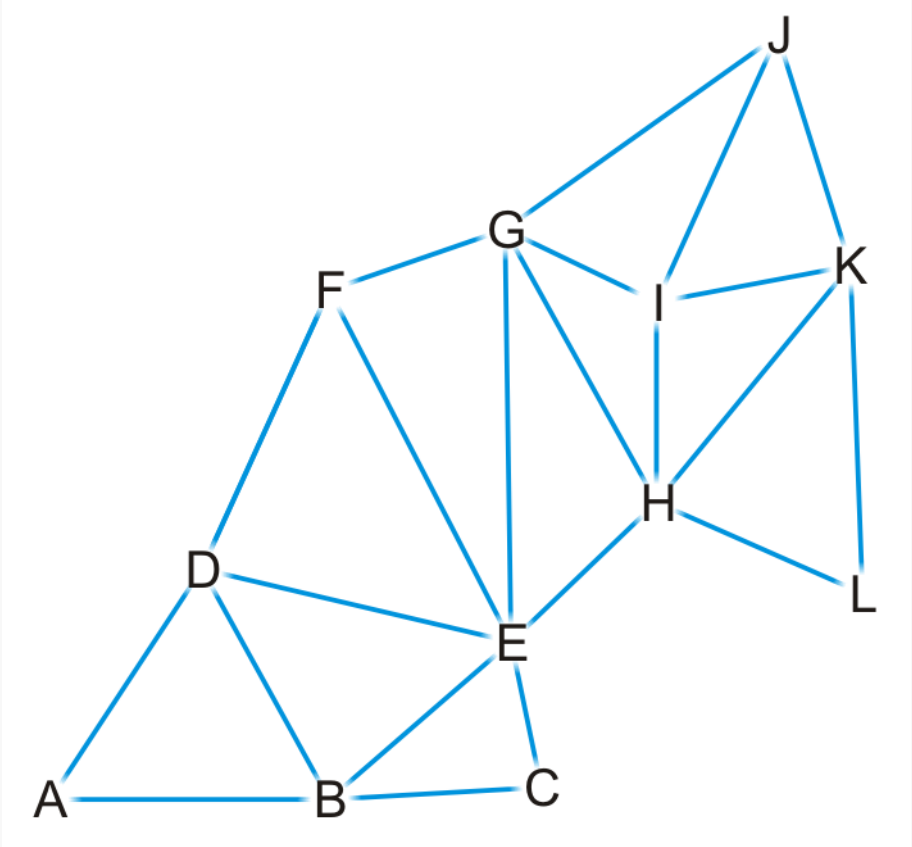
- We'll focus on Asia



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

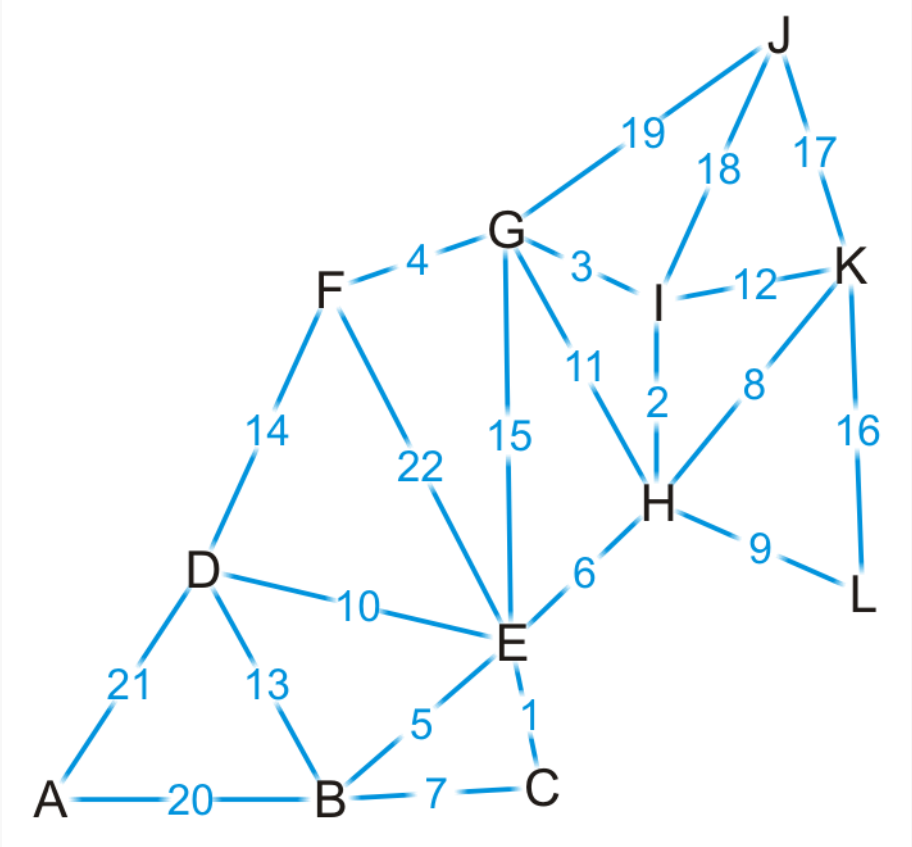
Example

- Here is our abstract representation



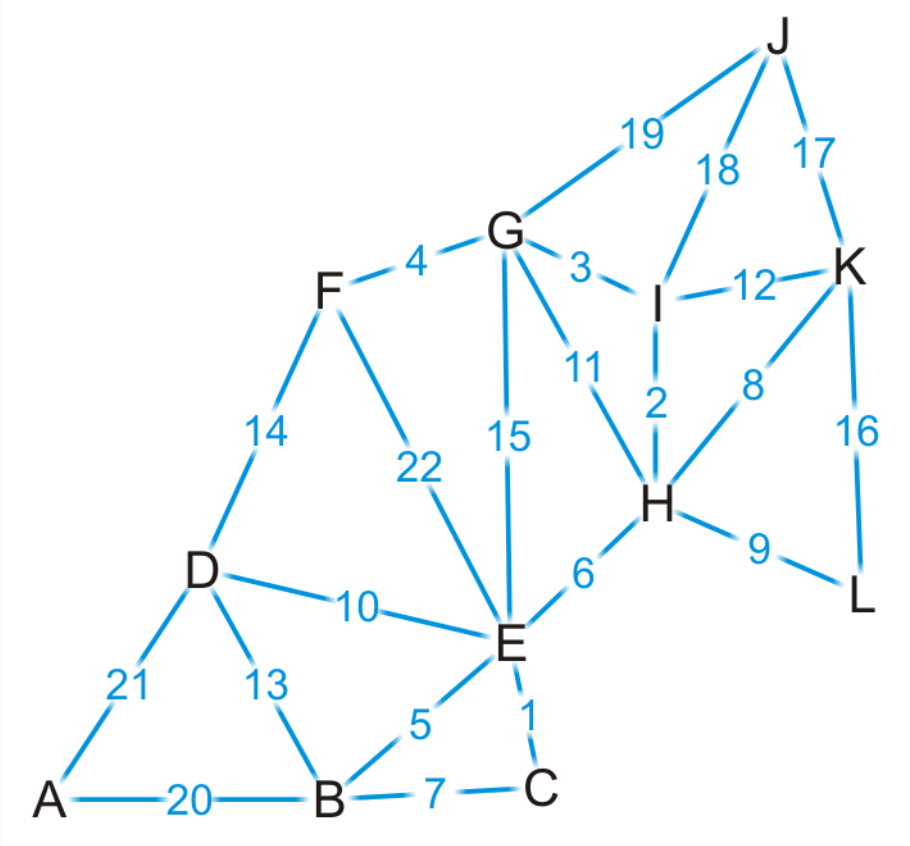
Example

- Let us give a weight to each of the edges



Example

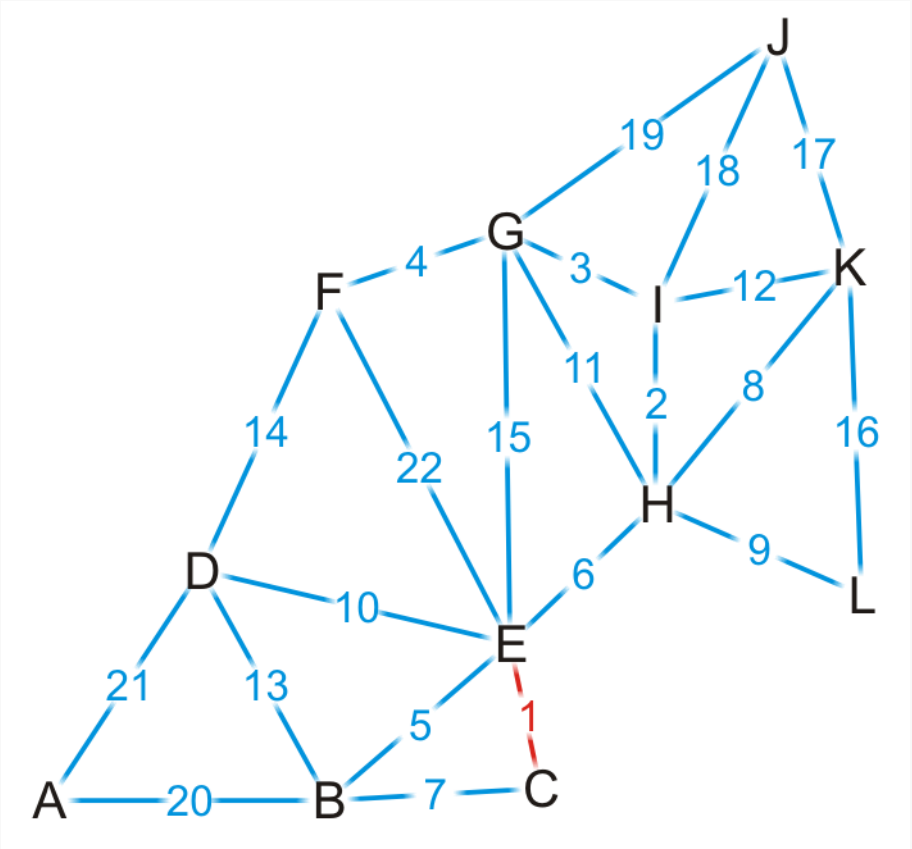
- First, we sort the edges based on weight



{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

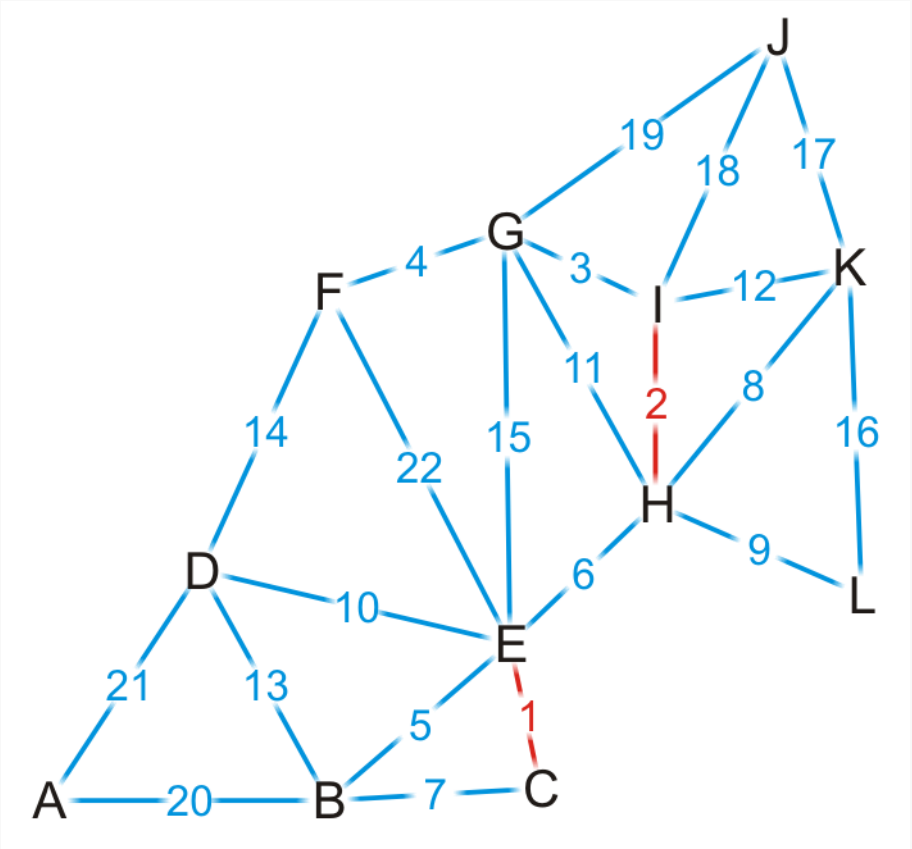
- We start by adding edge {C, E}



→ {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

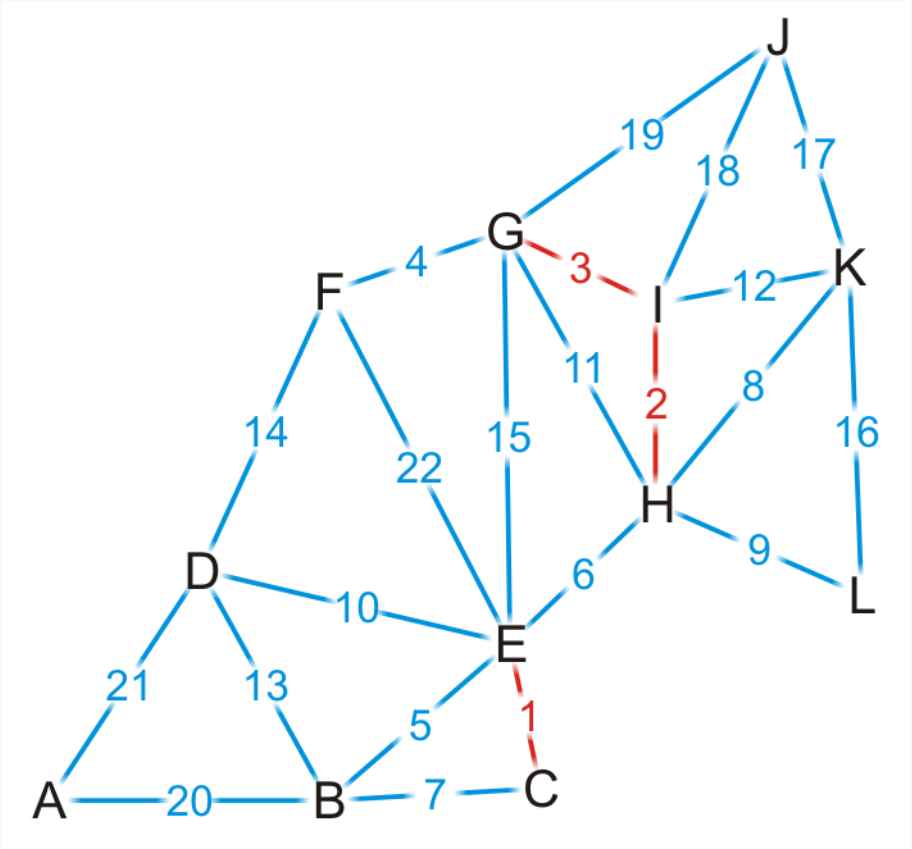
- We add edge {H, I}



{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

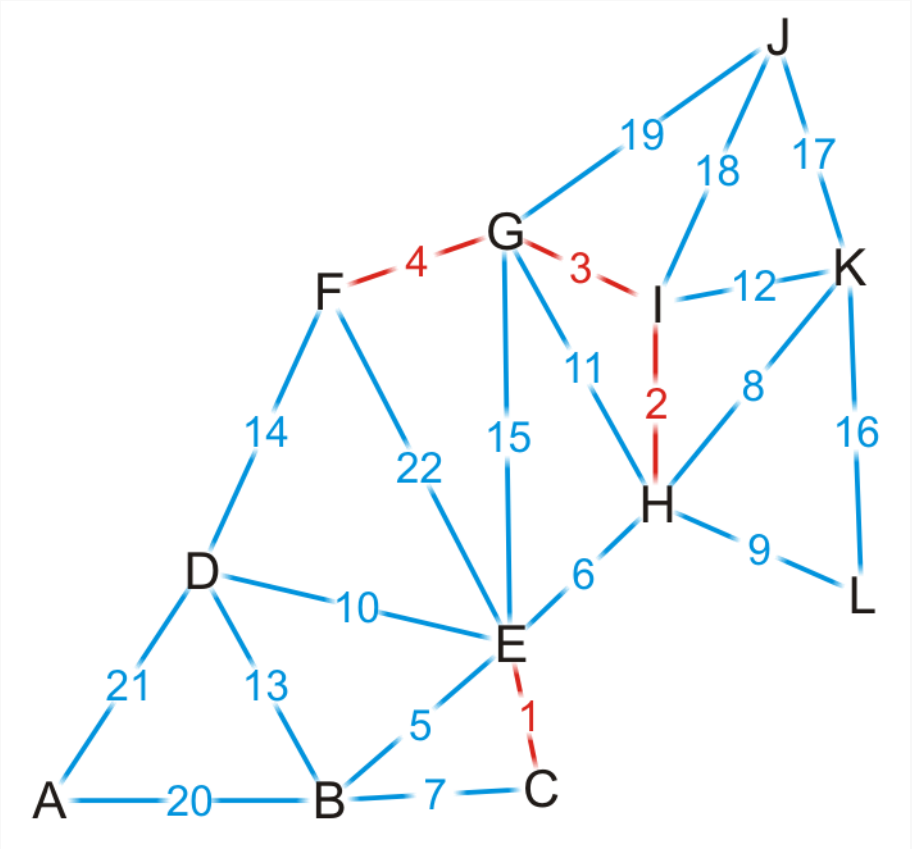
- We add edge {G, I}



{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

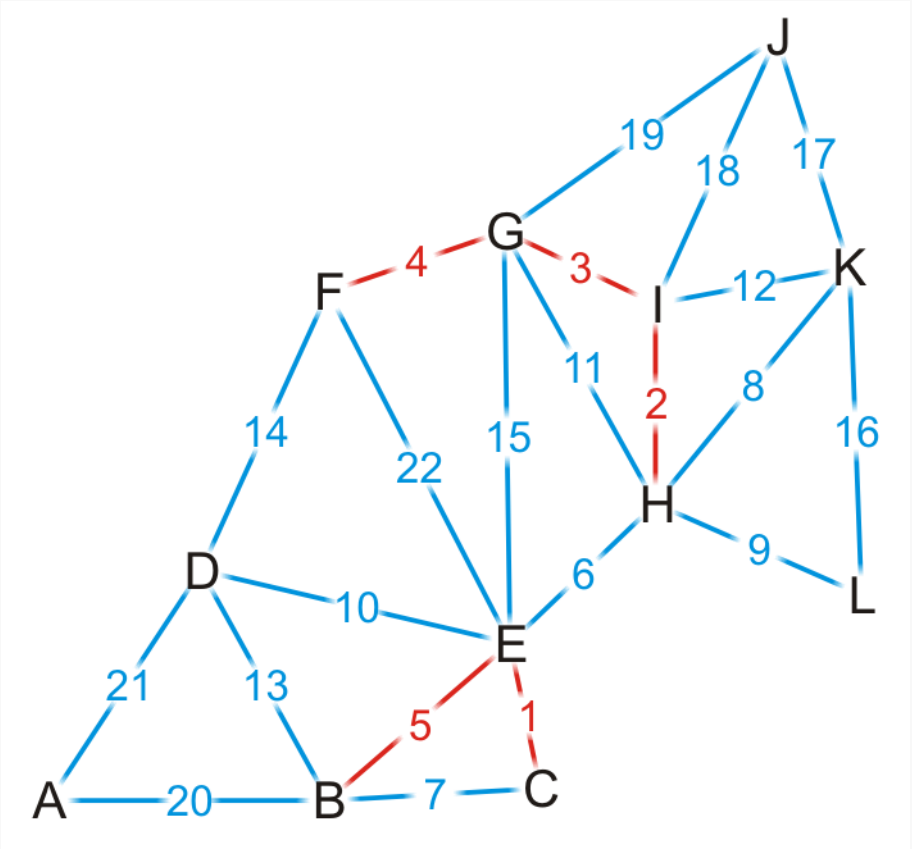
- We add edge {F, G}



{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

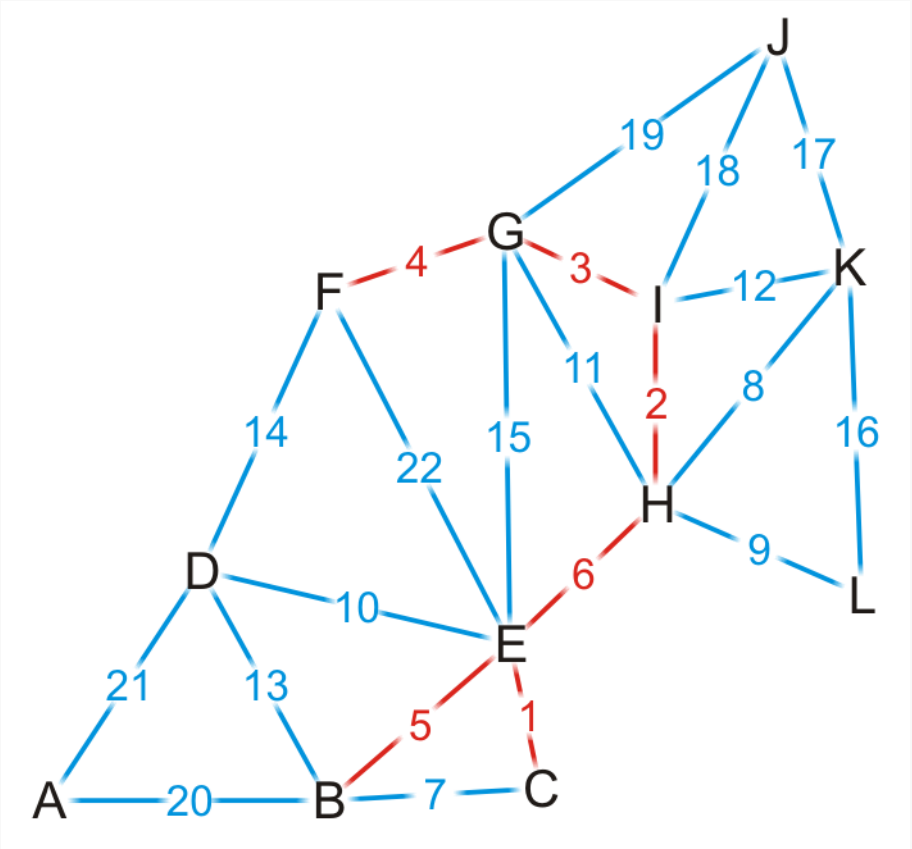
- We add edge {B, E}



{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

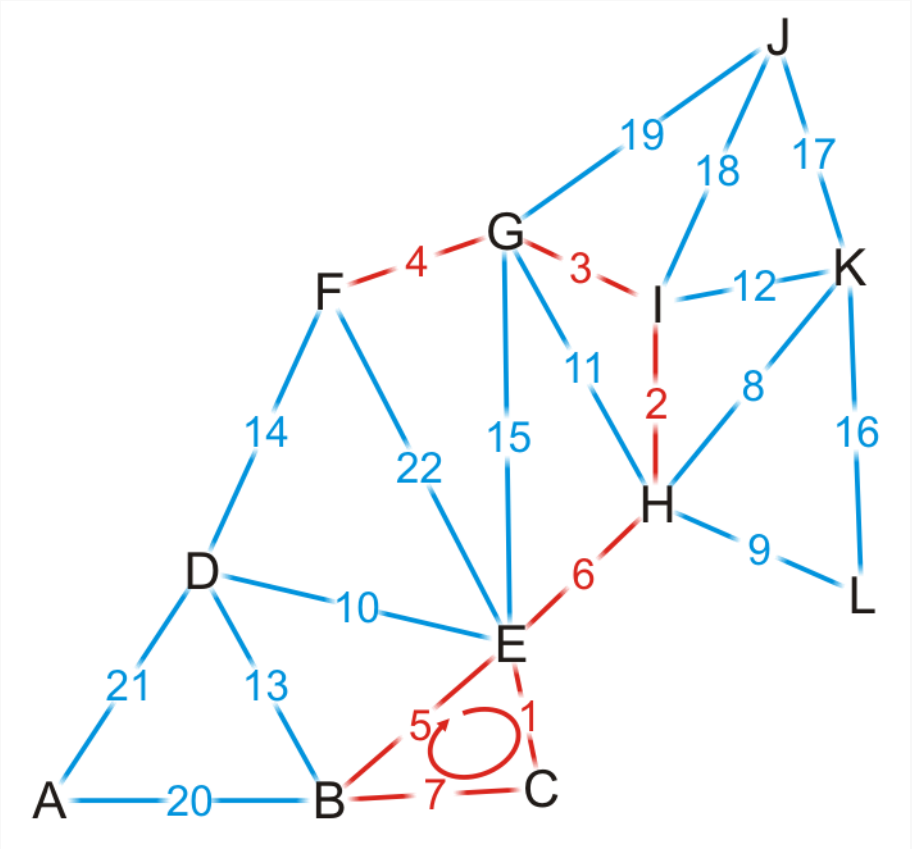
- We add edge {E, H}
- This combines the two spanning sub-trees into one



{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

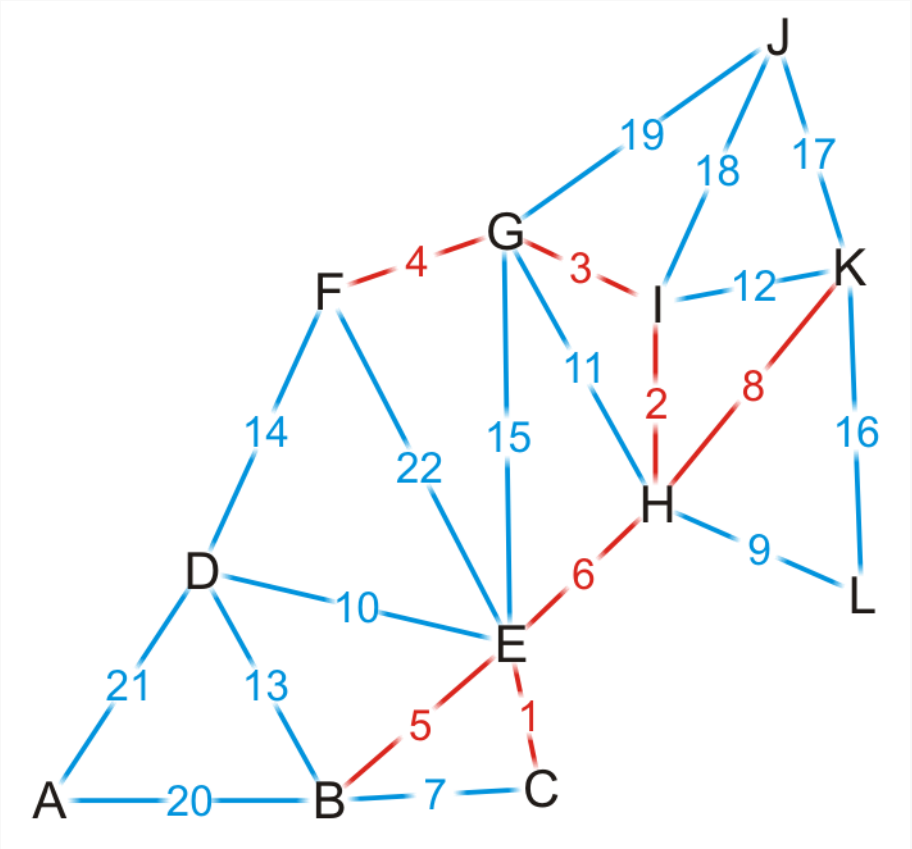
- We try adding {B, C}, but it creates a cycle



{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

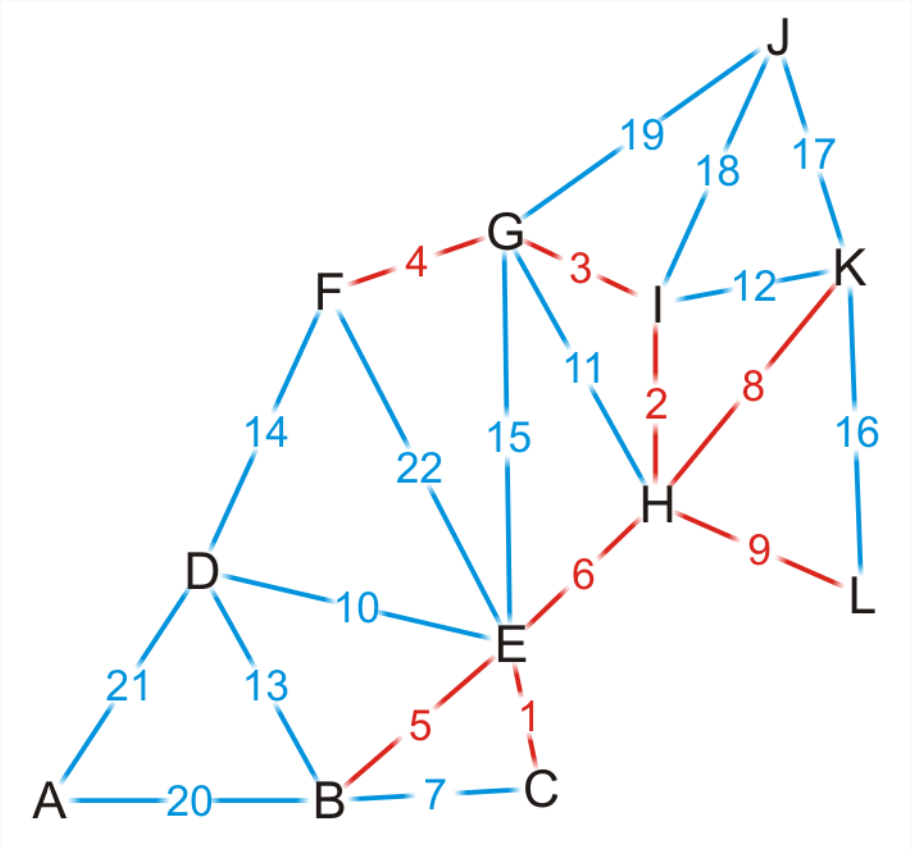
- We add edge {H, K}



{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

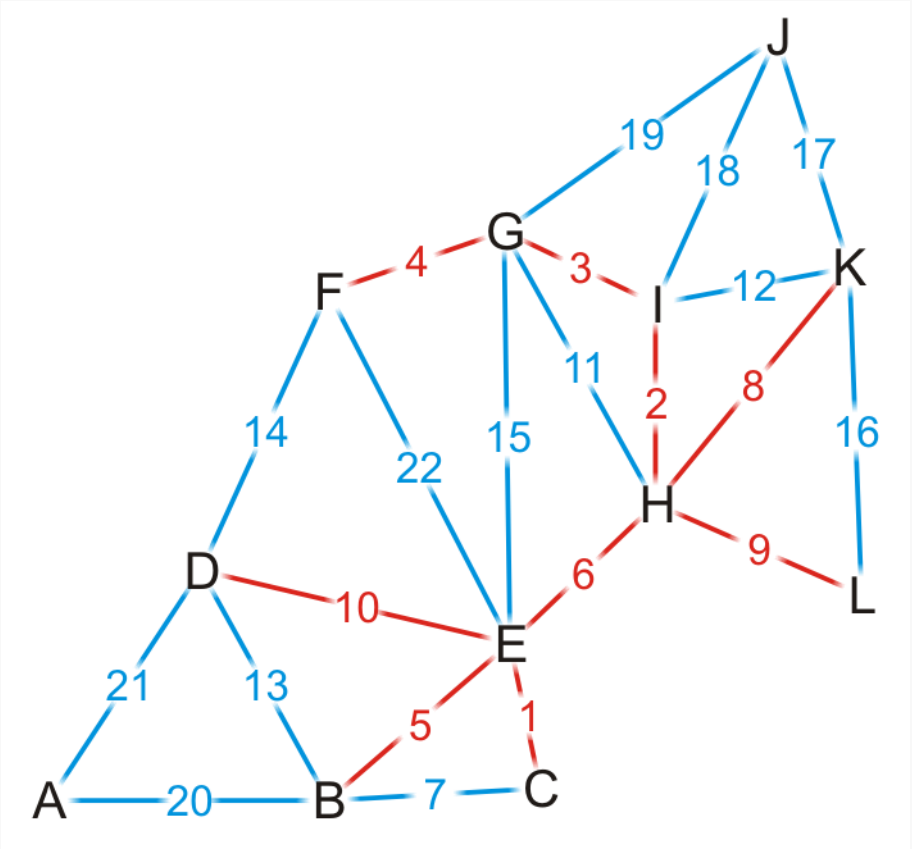
- We add edge {H, L}



{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

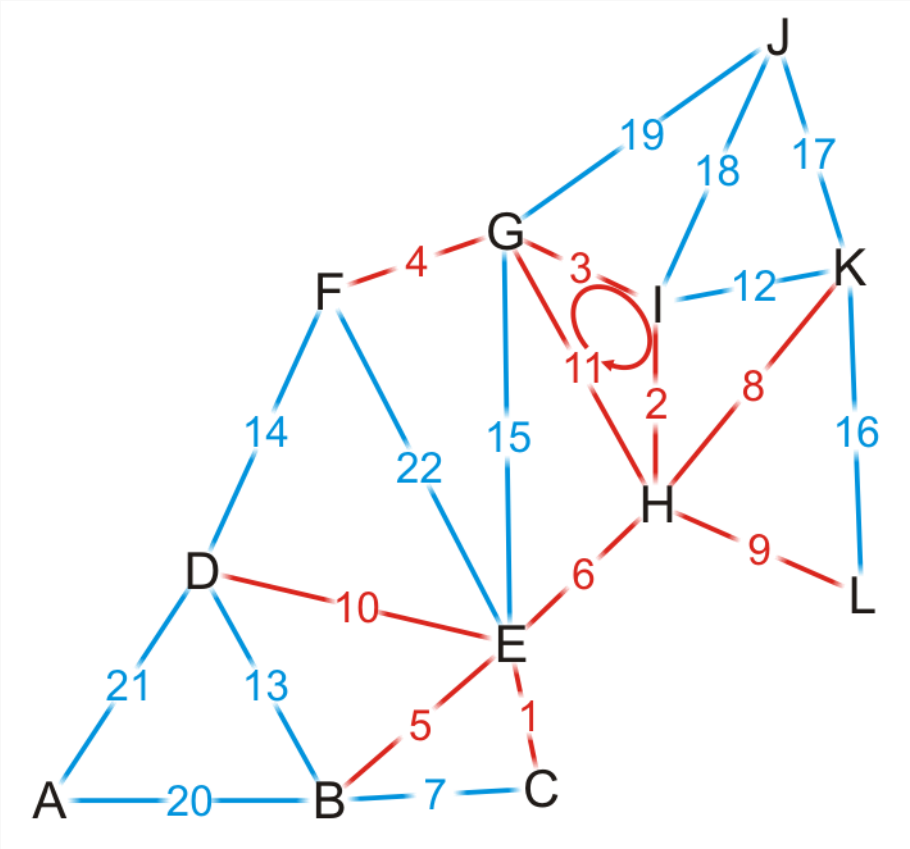
- We add edge {D, E}



{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

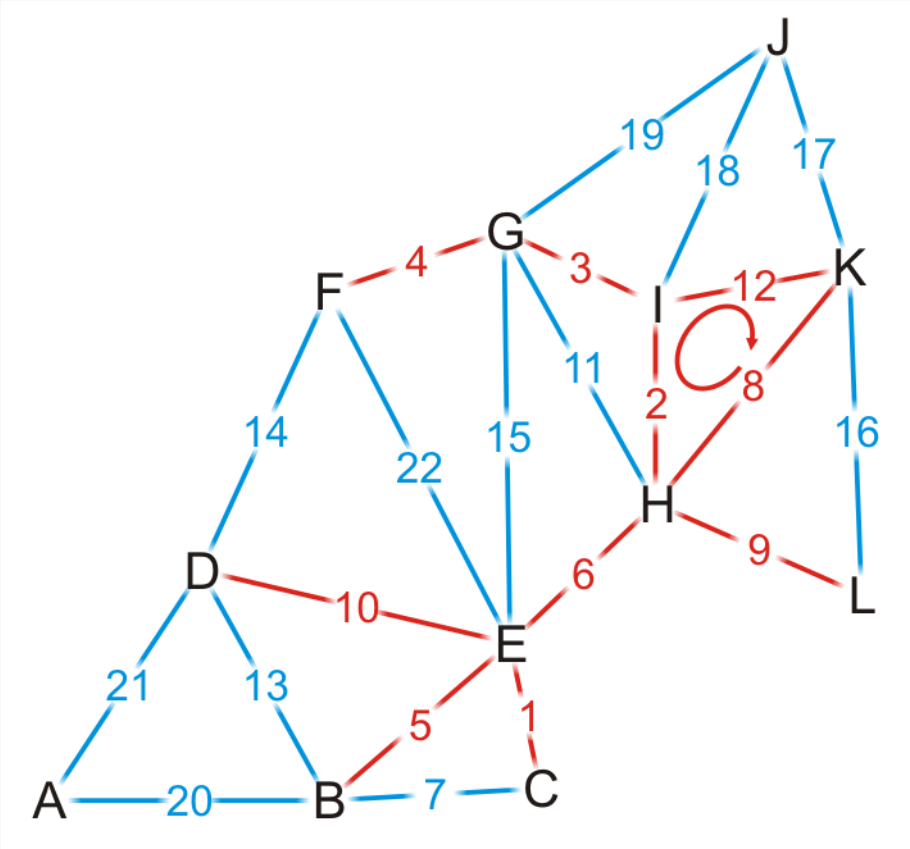
- We try adding {G, H}, but it creates a cycle



{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

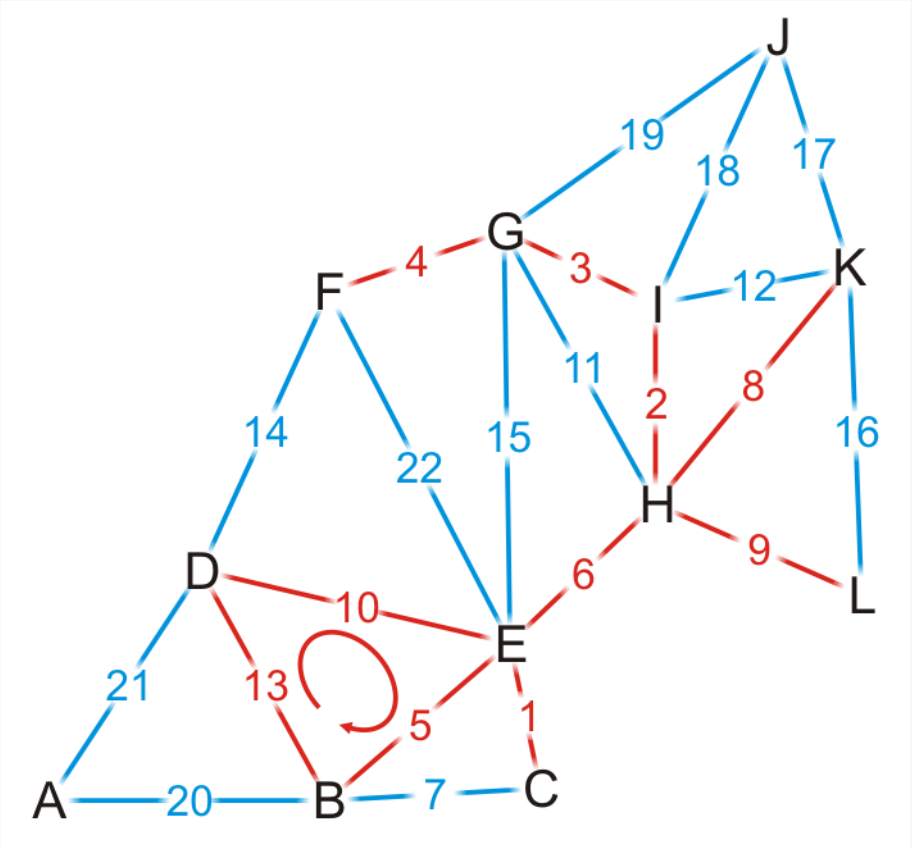
- We try adding {I, K}, but it creates a cycle



{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

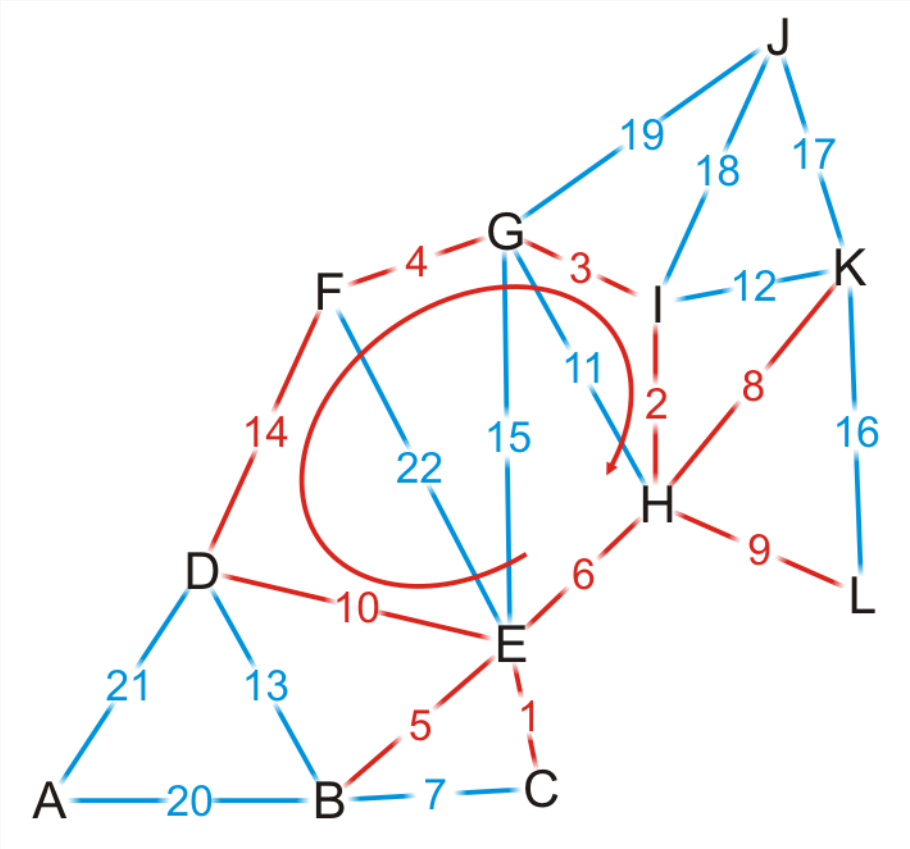
- We try adding {B, D}, but it creates a cycle



{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

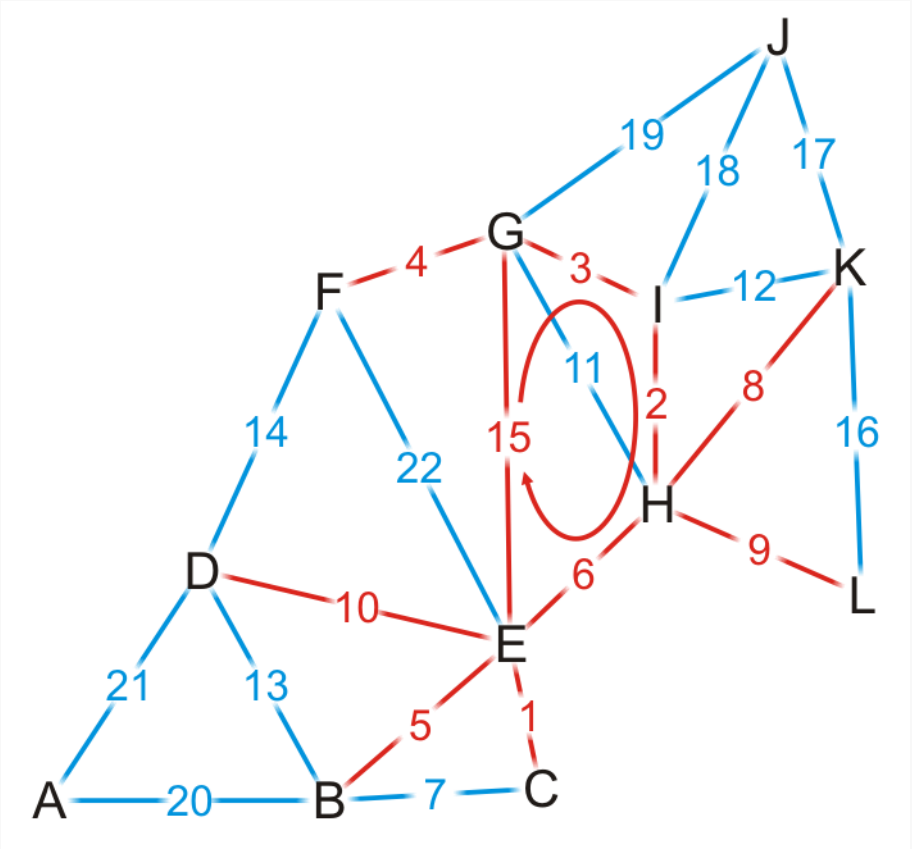
- We try adding {D, F}, but it creates a cycle



{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

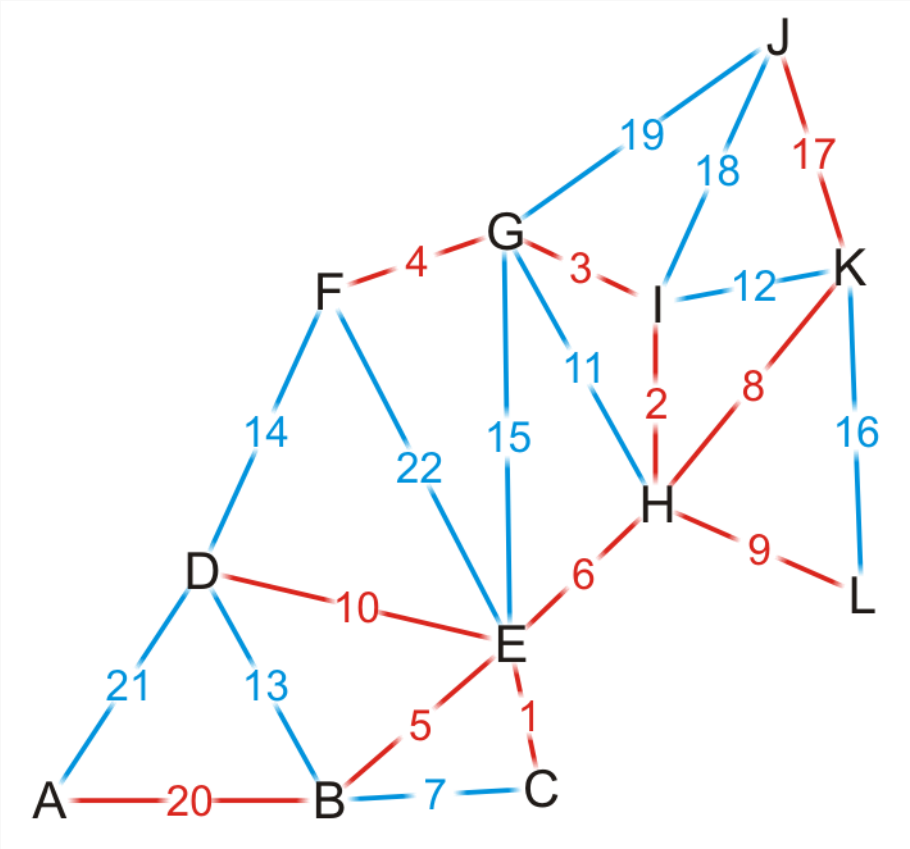
- We try adding {E, G}, but it creates a cycle



{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

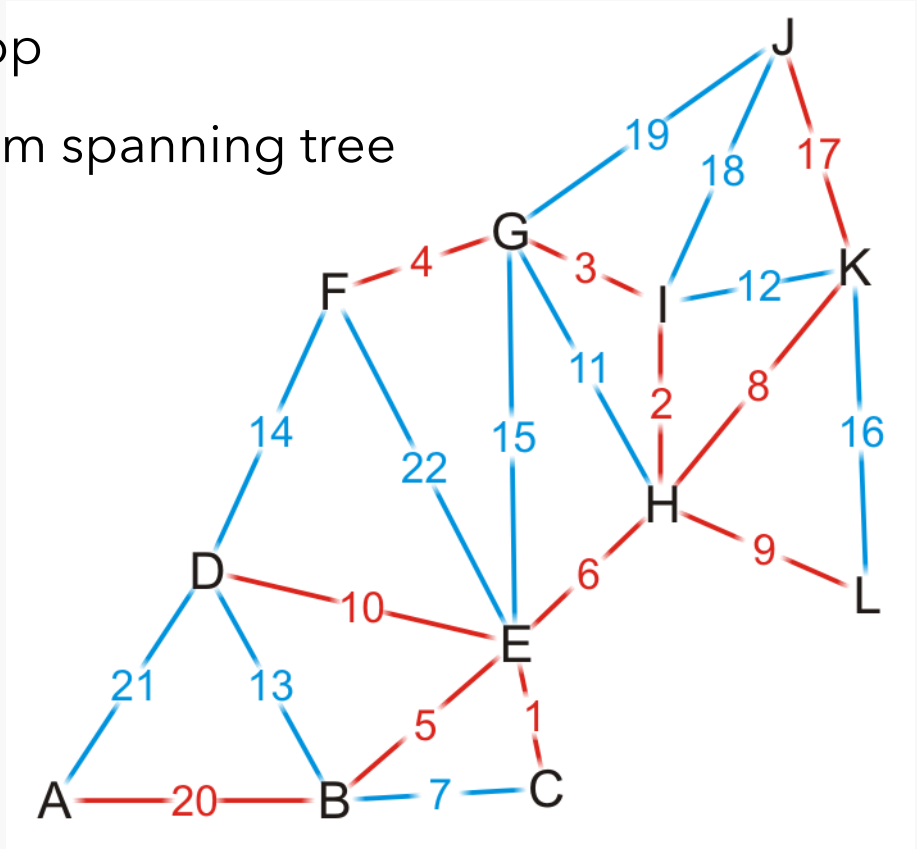
- By observation, we can still add edges {J, K} and {A, B}



{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 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 $O(|E| \ln(|V|) + |E||V|) =$ **$O(|E| \cdot |V|)$**

The critical operation is determining if two vertices are connected

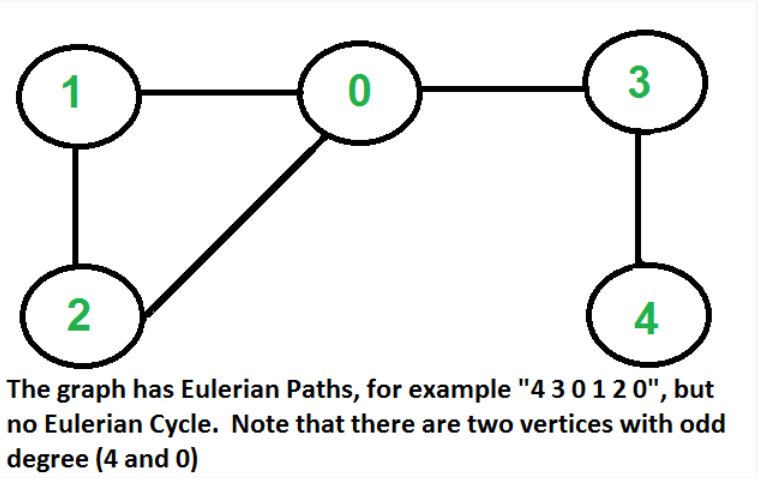
Euler path and circle

Eulerian Path & Cycle

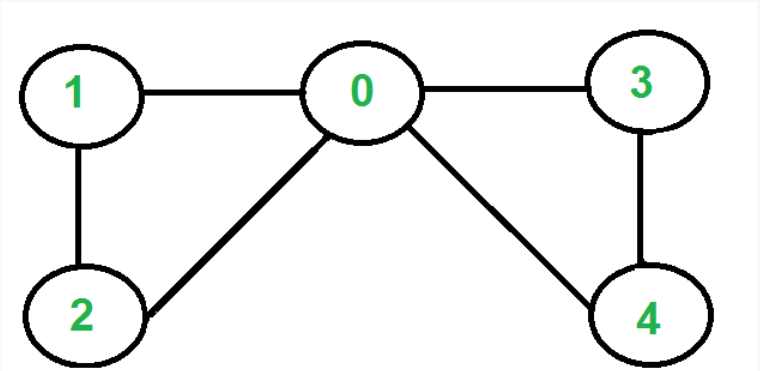
- An **Euler path (or trail)** is a path that uses every edge of a graph exactly once.
  - An Euler path starts and ends at **different** vertices.
- An **Euler circuit (or cycle)** is a circuit that uses every edge of a graph exactly once.
  - An Euler circuit starts and ends at **the same** vertex

Example

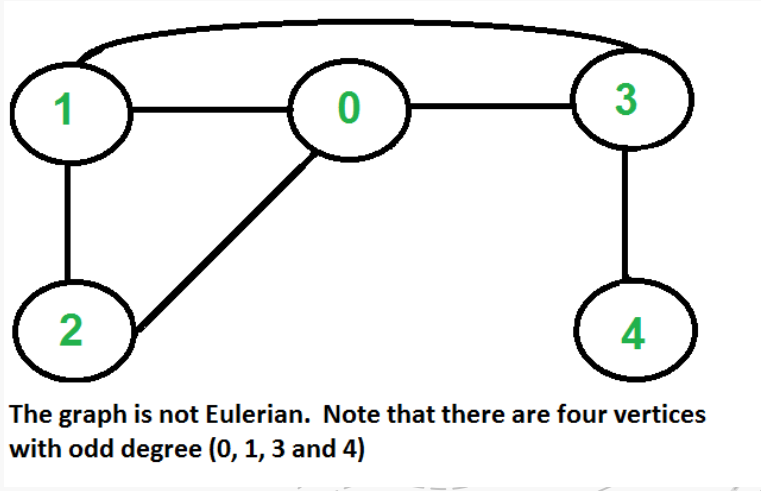
- *"Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once?"*



The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)



The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2" Note that all vertices have even degree



The graph is not Eulerian. Note that there are four vertices with odd degree (0, 1, 3 and 4)

Eulerian Cycle

- **Theorem**: *If a graph G has an Euler circuit, then all of its vertices must be even vertices.*
- Or
- *If the number of odd vertices in G is anything other than 0, then G cannot have an Euler circuit.*

Eulerian Path

- **Theorem**: *If a graph G has an Euler path, then it must have exactly two odd vertices.*
- Or
- *If the number of odd vertices in G is anything other than 2, then G cannot have an Euler path.*

A procedure for constructing an Euler cycle

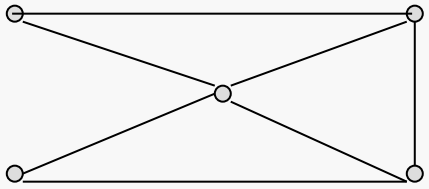
```
Algorithm  Euler(G)
//Input: Connected graph G with all vertices having even degrees
//Output: Euler cycle

Construct a cycle in G
Remove all the edges of cycle from G to get subgraph H
while H has edges
       find a non-isolated vertex v that is both in cycle and in H
       //the existence of such a vertex is guaranteed by G's connectivity
       construct subcycle in H
       splice subcycle into cycle at v
       remove all the edges of subcycle from H
return cycle
```

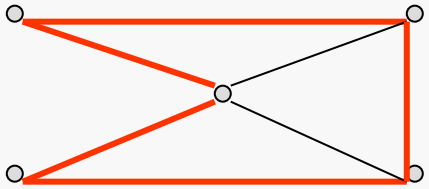
Algorithm for finding an Euler cycle from the vertex X using stack

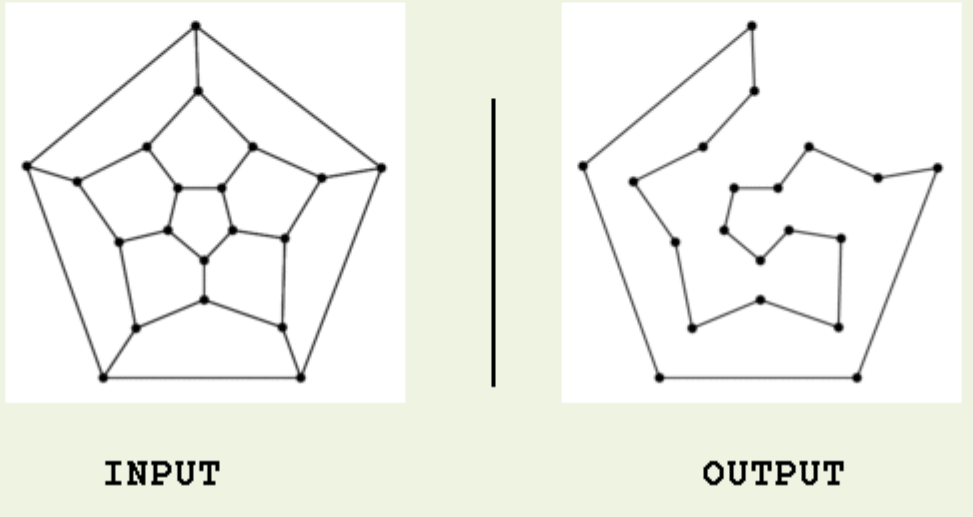
```
Algorithm  Euler(G)
//Input: Connected graph G with all vertices having even degrees
//Output: Euler cycle

declare a stack S of characters
declare empty array E (which will contain Euler cycle)
push the vertex X to S
while(S is not empty)
 {ch = top element of the stack S
  if ch is isolated then remove it from the stack and put it to E
   else
   select the first vertex Y (by alphabet order), which is adjacent
   to ch,push  Y  to S and remove the edge (ch,Y) from the graph
 }
 the last array E obtained is an Euler cycle of the graph
```

Hamilton paths and cycles

- **Hamilton(ian) Cycle or Circuit** in a graph G is a cycle that visits every vertex of G exactly once and returns to the starting vertex.
- **Hamilton(ian) Path** in a graph G is a path that visits every vertex of G exactly once and Hamilton(ian) Path doesn't have to return to the starting vertex. It's an open path.







Hamilton paths and cycles

- **Applications**
- The Hamiltonian Cycle problem has practical applications in various fields, such as logistics, network design, and computer science.
- Hamiltonian Paths have applications in various fields, such as finding optimal routes in transportation networks, circuit design, and graph theory research.

Hamiltonian Cycle using Backtracking Algorithm

- Create an empty path array and add vertex 0 to it.
- Add other vertices, starting from the vertex 1.
- Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added.
  - If we find such a vertex, we add the vertex as part of the solution.
  - If we do not find a vertex, then we return false.

Summary

- Spanning Trees
  - Prim algorithm
  - Kruskal algorithm
- Eulerian and Hamilton Graphs

Homework & Further Reading

Java code for

- Spanning Trees
  - Prim algorithm
  - Kruskal algorithm
- Eulerian and Hamilton Graphs

Vietnam National University of HCMC
International University
School of Computer Science and Engineering

THANK YOU

Dr Vi Chi Thanh - vcthanh@hcmiu.edu.vn
https://vichithanh.github.io