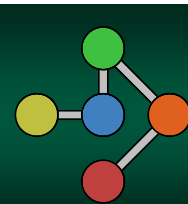
Graph Spanning Trees

Section 4.1 – 4.3

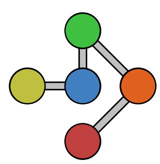
Spanning Trees



Connect the Dots (la la la la)

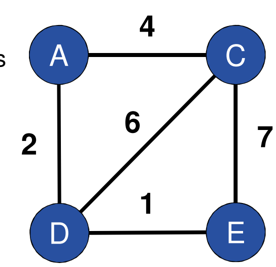
Spanning Trees

- In many cases, we want to make sure that every vertex is connected in a graph
- A *Spanning Tree* includes every vertex of the original graph (an no cycles, obviously)



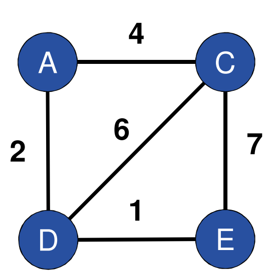
Example Graph

- The graph on the left is quite simple – containing just 4 vertices
- We need to create a tree that contains all 4 vertices

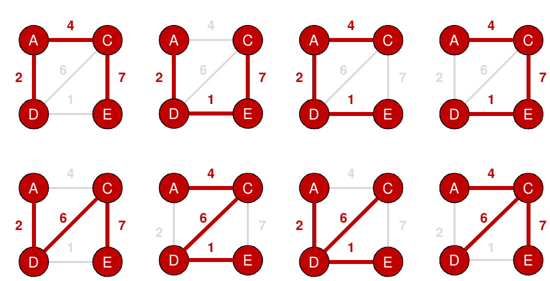


Example Graph

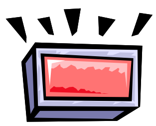
- As you can imagine, most graphs have multiple solutions
- Also note that this graph is weighted
- This can make some solutions better than others



Example: Spanning Trees



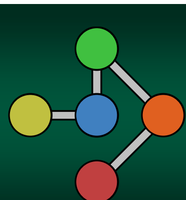
So How Many Spanning Trees?



- Given a complete graph of *N* vertices, there are N^{N-2} possible spanning trees
- Yes, that is N to the power of N – which is even *worst* than exponential growth!

Complete Graph Spanning Trees

Vertices	Total Spanning Trees	Vertices	Total Spanning Trees
1	1	**7**	16,807
2	1	**8**	262,144
3	3	**9**	4,782,969
4	16	**10**	100,000,000
5	125	**11**	2,357,947,691
6	1,296	**12**	61,917,364,224

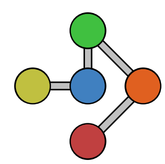


Minimum Spanning Tree

Getting the "Best" Tree

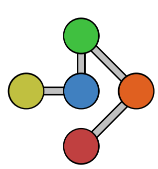
Minimum Spanning Tree

- *Minimum Spanning Tree (MST)* contains all the vertices of a graph with the minimum possible weight
- Naturally, depending on the weights, some solutions are more desirable than others

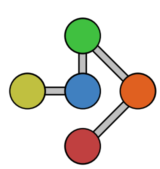


Minimum Spanning Tree

- Some uses (minimize cost)
  - building cable networks
  - building a road that joins cities
- So, creating an algorithm – that computes these trees - is of vital importance

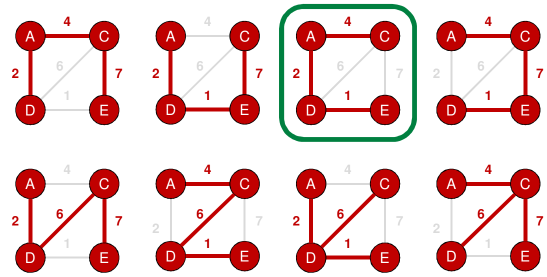


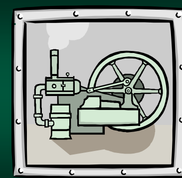
Uniformly Weighted Graph



- If each edge has the same weight then the computation of the minimum spanning tree is trivial
- A simple breadth-first search or depth-first search can be used

Example: Minimal Spanning Tree



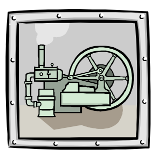


Brute Force

Doing in the hard way
(and impossible)

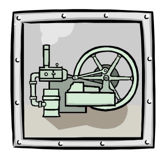
Brute Force

- One way to compute the minimum spanning tree is to simply ***try all of them!***
- Approach:
  - calculate every tree
  - keep track of the tree with the minimum total weight



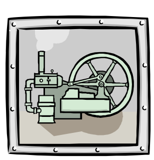
Brute Force

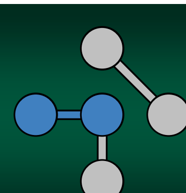
- Calculating just one tree will require O(n) time where *n* is the total number of vertices
- How many trees are there?
- For complete graphs, we know it's n^{n-2} – which is what we must use for Big-O



Brute Force

- So, there is a $n \times n^{n-2}$ computational requirement
- Which is $\mathbf{O(n^{n-1})}$
- Naturally, this is a poor – *and quite impractical* – solution



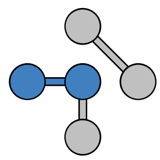


Kruskal's Algorithm

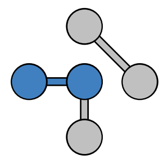
Computing a Minimum Spanning Tree

Kruskal's Algorithm

- *Kruskal's Algorithm* computes a minimum spanning tree
- It was invented in 1956 by *Joseph Kruskal* and published in *Proceedings of the American Mathematical Society*

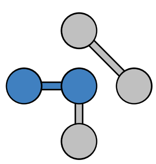


Kruskal's Algorithm



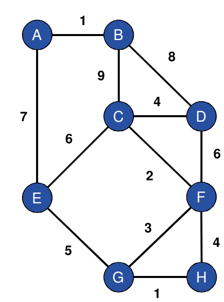
- It conceptually starts with a forest of single vertices
- It then adds edges – with the minimum weight – to connect the two vertices (to connect these subtrees)

Kruskal's Algorithm



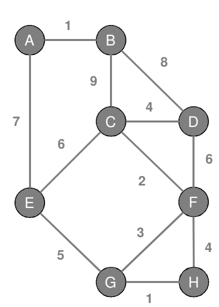
- So, it adds edges in sorted order of weight
- If an edge would cause a cycle, then it is rejected and not part of the solution

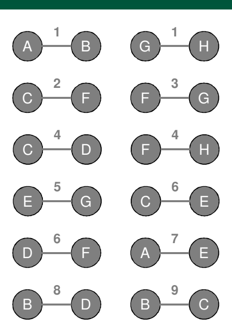
Kruskal's Algorithm Example



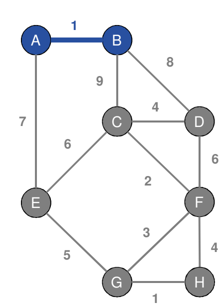
- The following is a graph with multiple weights
- Kruskal's Algorithm will create a list of the edges sorted by weight

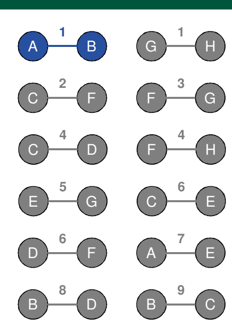
Kruskal's Algorithm Example



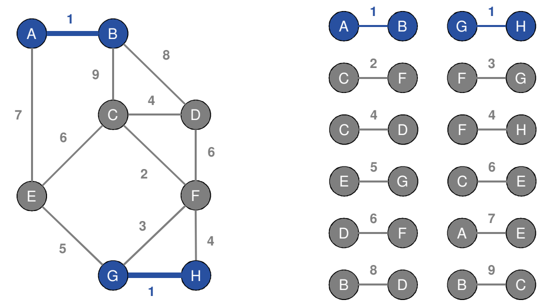


Add Edge

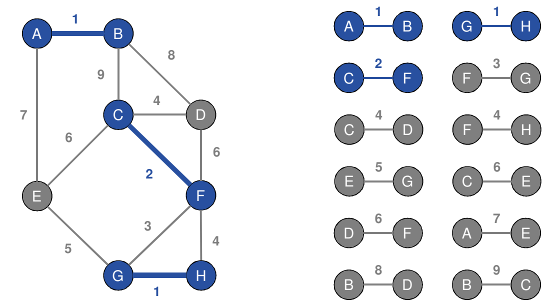




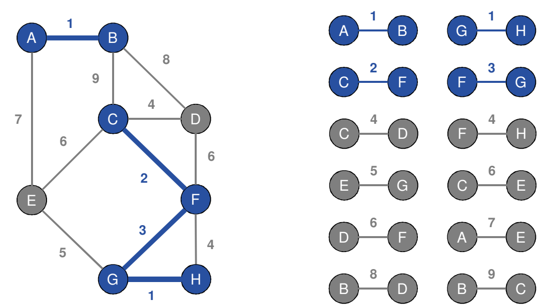
Add Edge



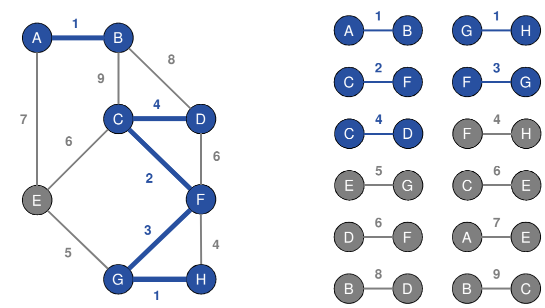
Add Edge



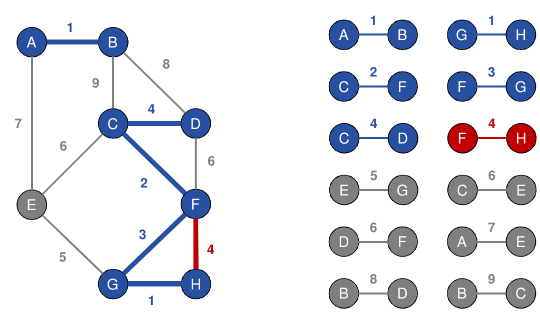
Add Edge



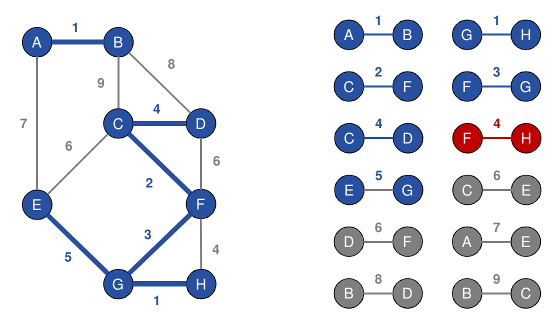
Add Edge



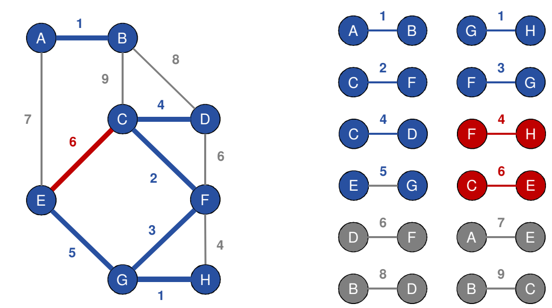
Reject F - H: Would Create a Cycle



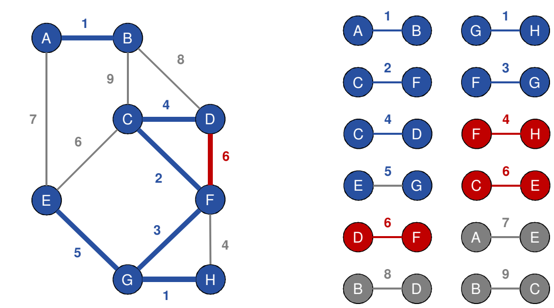
Add Edge



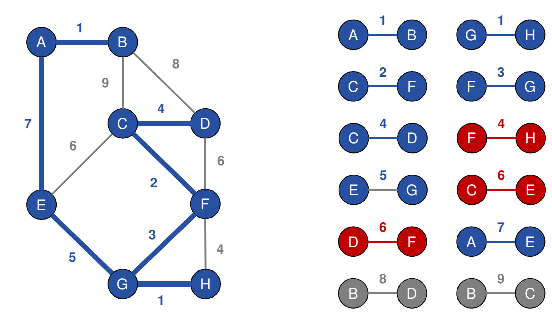
Reject C - E: Would Create a Cycle



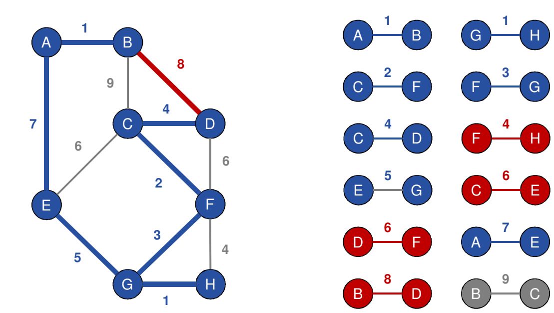
Reject D - F: Would Create a Cycle



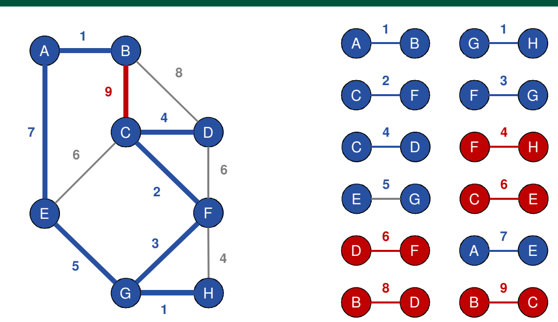
Add Edge



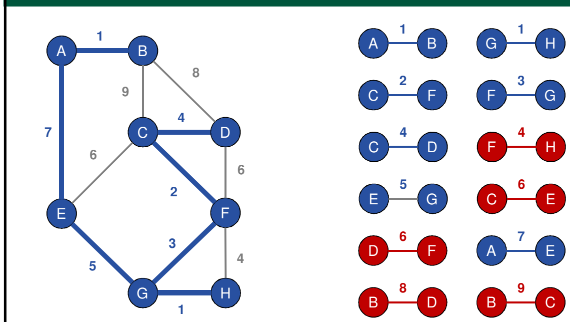
Reject B - D: Would Create a Cycle



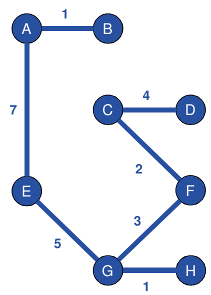
Reject B - C: Would Create a Cycle

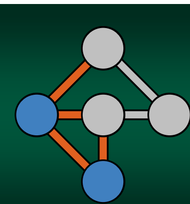


Kruskal’s Algorithm Complete



Resulting Minimum Spanning Tree



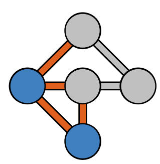


Prim's Algorithm

Computing a Minimum Spanning Tree

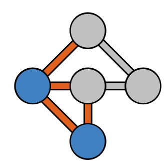
Prim's Algorithm

- *Prim's Algorithm* computes a minimum spanning tree
- It was invented in 1956 by computer scientist *Robert C. Prim*

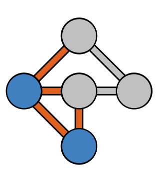


Prim's Algorithm

- However, it was discovered that computer scientist *Vojtěch Jarník* had also invented it in 1930
- So, it is sometimes called the *Prim-Jarník Algorithm*

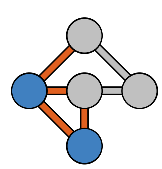


Prim's Algorithm



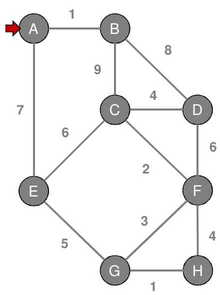
- This algorithm starts with a single (arbitrarily chosen) vertex
- The algorithm then adds the minimal edge – that the tree can currently "see"

Prim's Algorithm

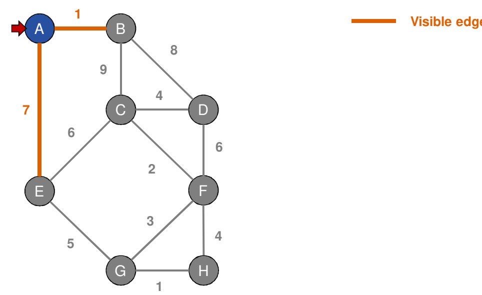


- These are edges connect to unvisited vertices (not in the tree)
- The algorithm is *greedy* – always following the local optimal path – to grow the tree

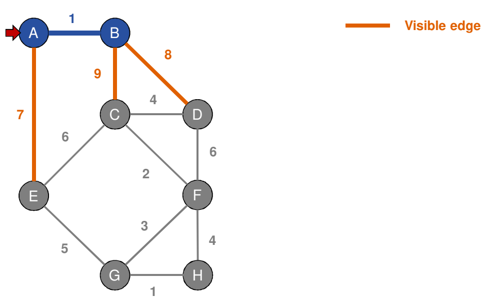
Prim's Algorithm Example



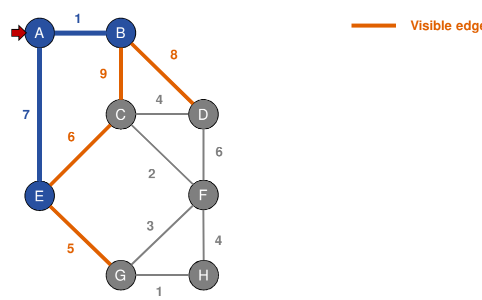
Prim's Algorithm Example



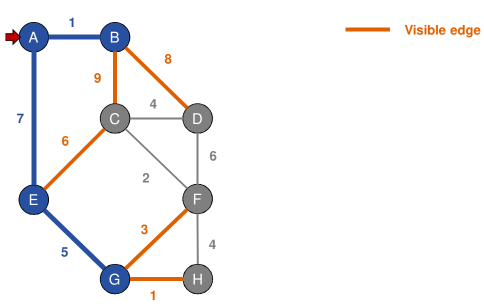
Prim's Algorithm Example



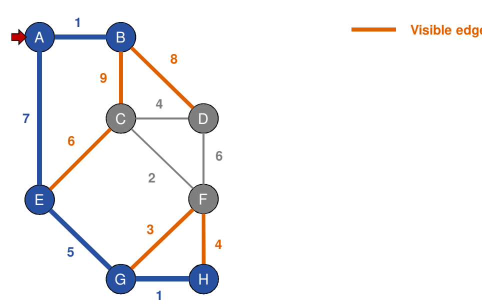
Prim's Algorithm Example



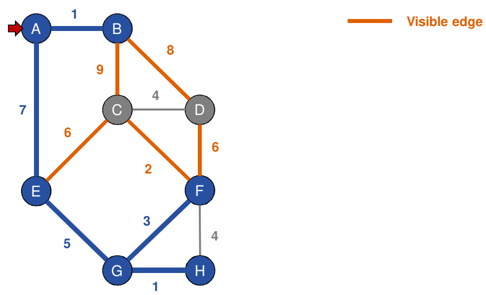
Prim's Algorithm Example



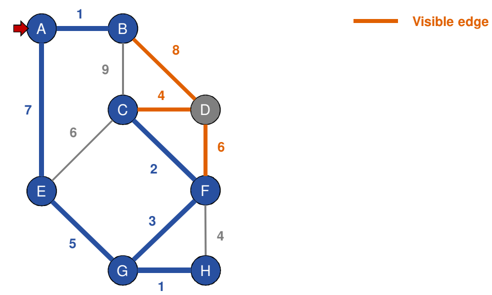
Prim's Algorithm Example



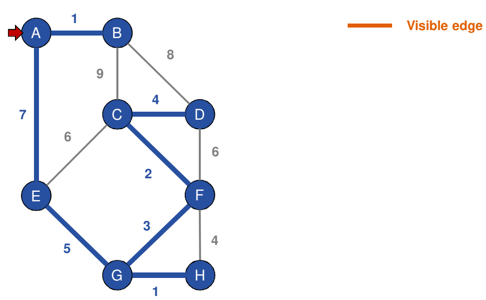
Prim's Algorithm Example



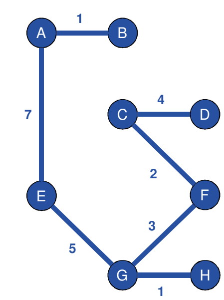
Prim's Algorithm Example



Prim's Algorithm Complete



Resulting Minimum Spanning Tree



Kruskal vs. Prim Algorithm

- Both algorithms find the minimum spanning tree – *though not necessarily identical*
- Both are O(|E| log |V|) where |E| is the number of edges and |V| is the number of vertices
- Kruskal is far easier to conceptualize, but Prim is far easier to implement