CSC 212

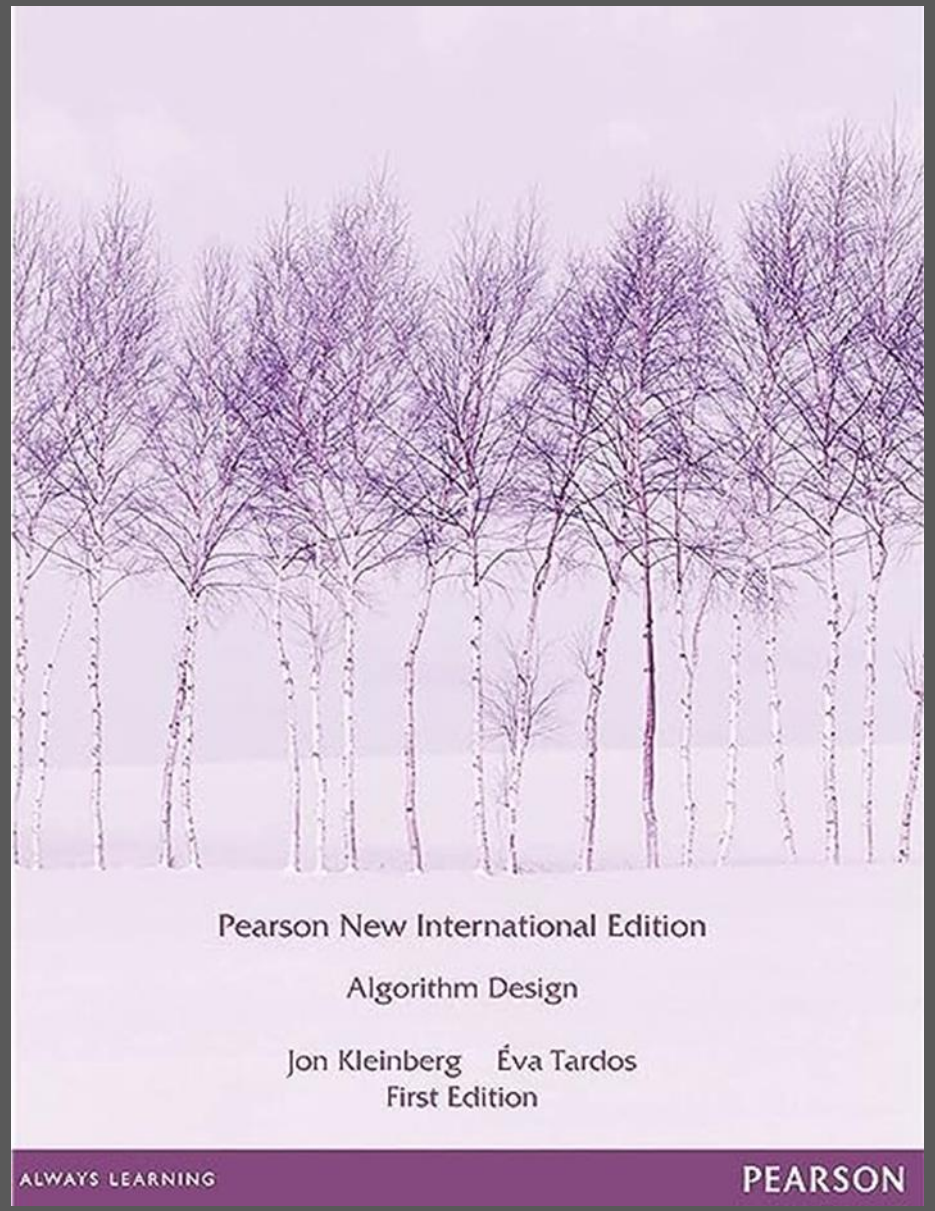
Algorithms & Complexity

Greedy Algorithms

builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion.

Reading

Read Chapter 4
and look through the
exercises



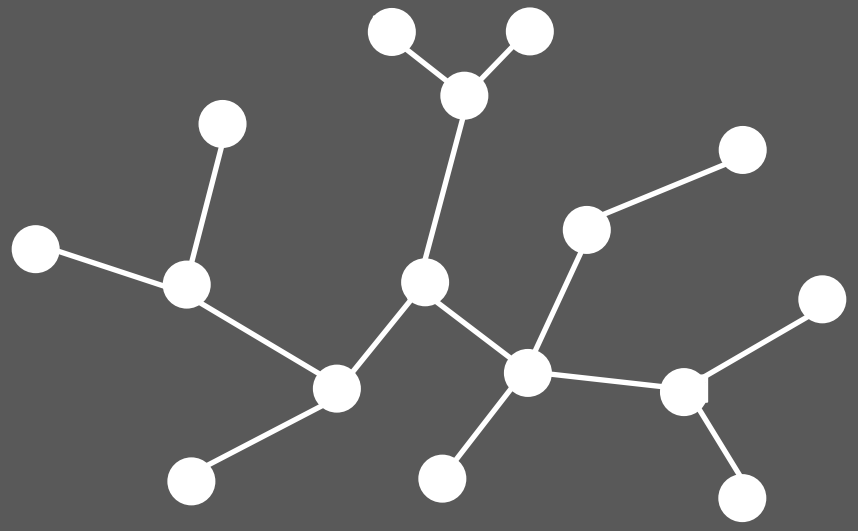
Contents

- Introduction
- Minimum Spanning Tree
- Greedy Algorithms

Note: Review content about Trees from Slide 02_Graphs_Part1

Trees

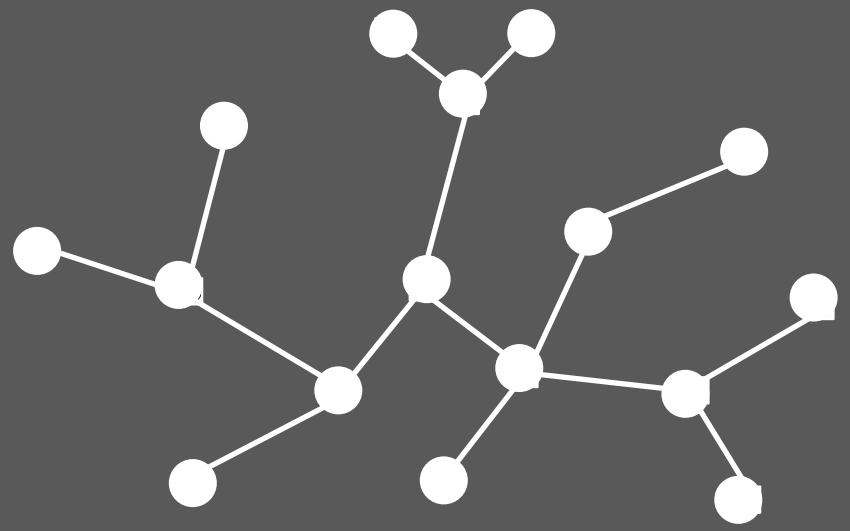
A tree is an undirected graph with exactly one simple path between any pair of nodes



Equivalently: a tree is an undirected graph if it is connected (there is a path between any pair of nodes) and there are no cycles

Properties of Trees

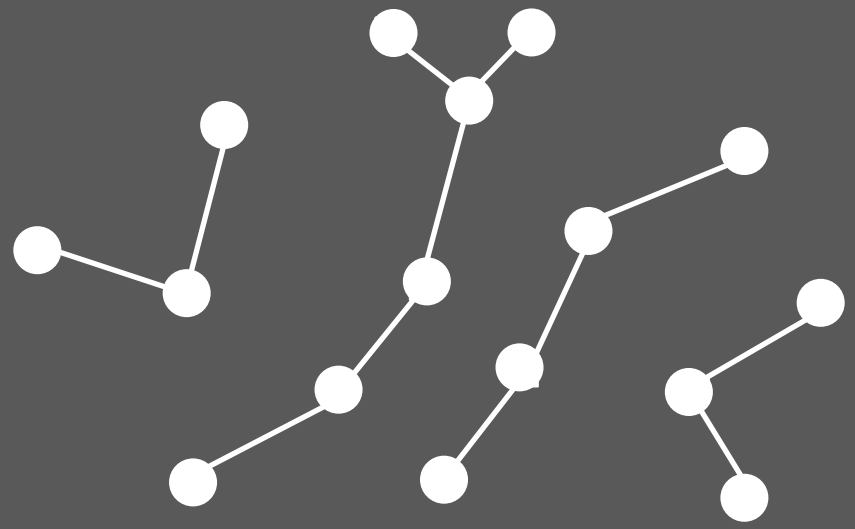
1. $|E| = |V| - 1$
2. connected
3. no cycles



Any two of these properties imply the third, and imply that the graph is a Tree

Forest

An acyclic graph (a graph without cycles) that is not connected is called a Forest.
Each of the connected components in a forest is a Tree.



Spanning Trees

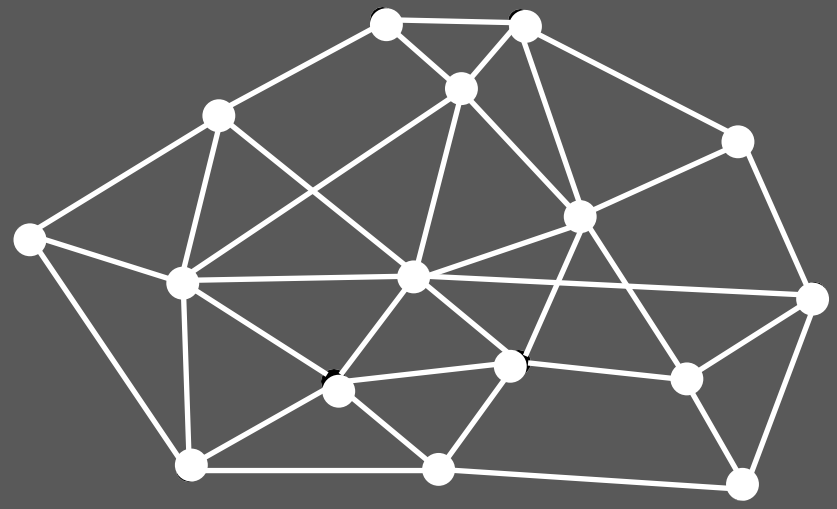
Given an undirected, connected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices of the graph.

Let $G = (V, E)$ be a graph.

A subset $T \subseteq E$ is a spanning tree if (V, T) is a tree

Spanning Trees

Given an undirected, connected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices of the graph.



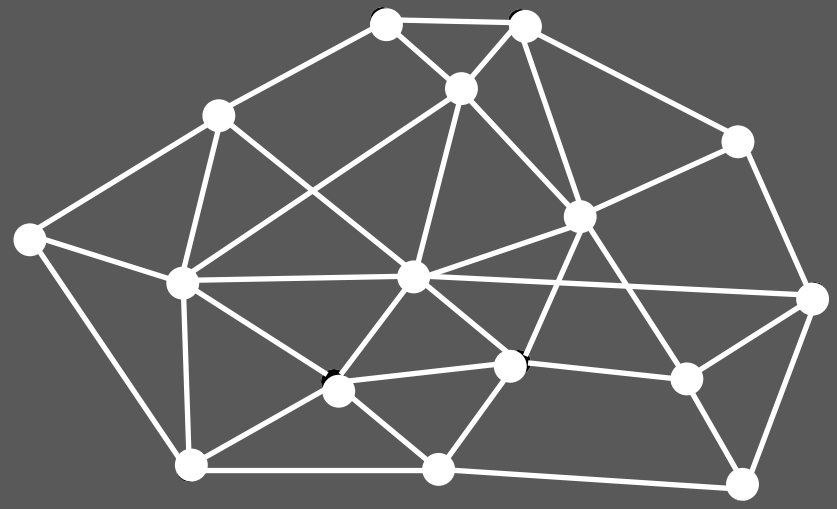
Let $G = (V, E)$ be a graph.

A subset $T \subseteq E$ is a spanning tree

if (V, T) is a tree

Spanning Trees

Given an undirected, connected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices of the graph.

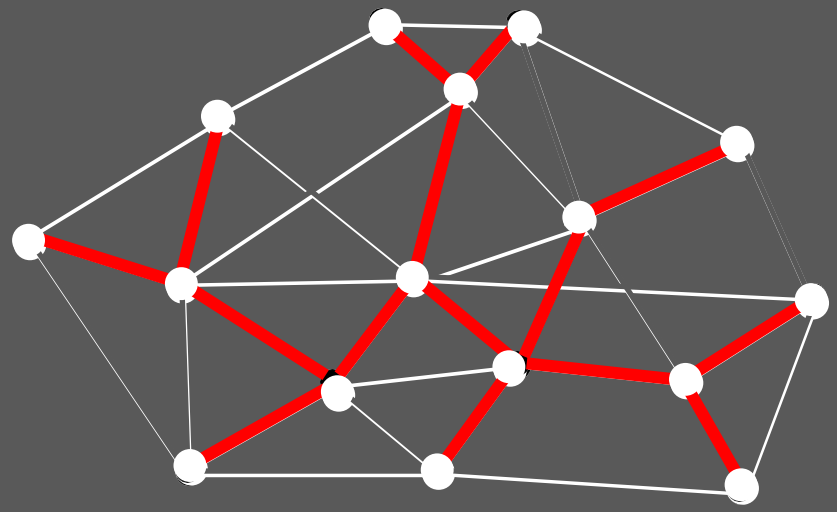


Let $G = (V, E)$ be a graph.

A subset $T \subseteq E$ is a spanning tree if (V, T) is a tree

Spanning Trees

Given an undirected, connected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices of the graph.



Let $G = (V, E)$ be a graph.

A subset $T \subseteq E$ is a spanning tree

if (V, T) is a tree

Same set of vertices V

$T \subseteq E$

(V, T) is a tree

Spanning Trees

Given an undirected, connected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices of the graph.

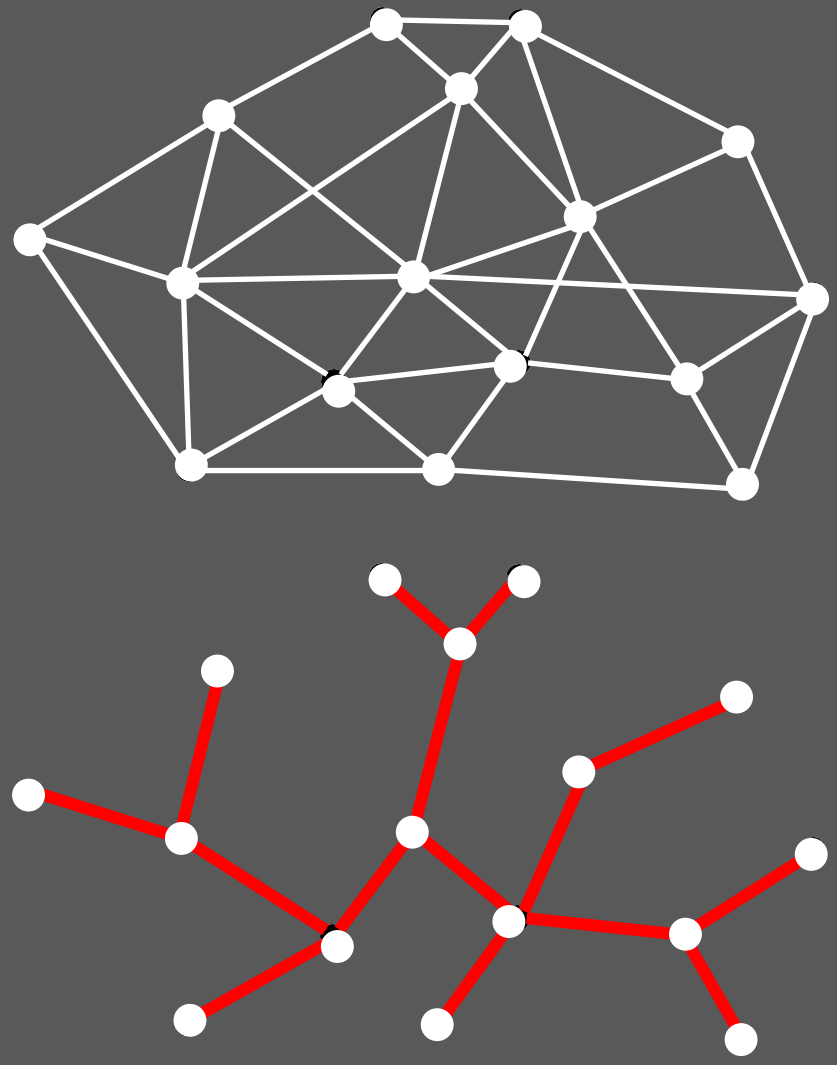
Let $G = (V, E)$ be a graph.

A subset $T \subseteq E$ is a spanning tree if (V, T) is a tree

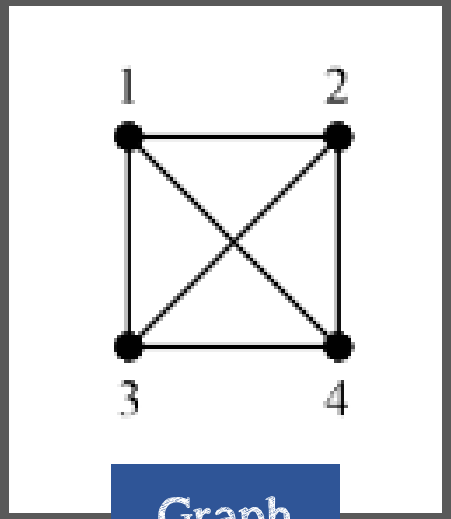
Same set of vertices V

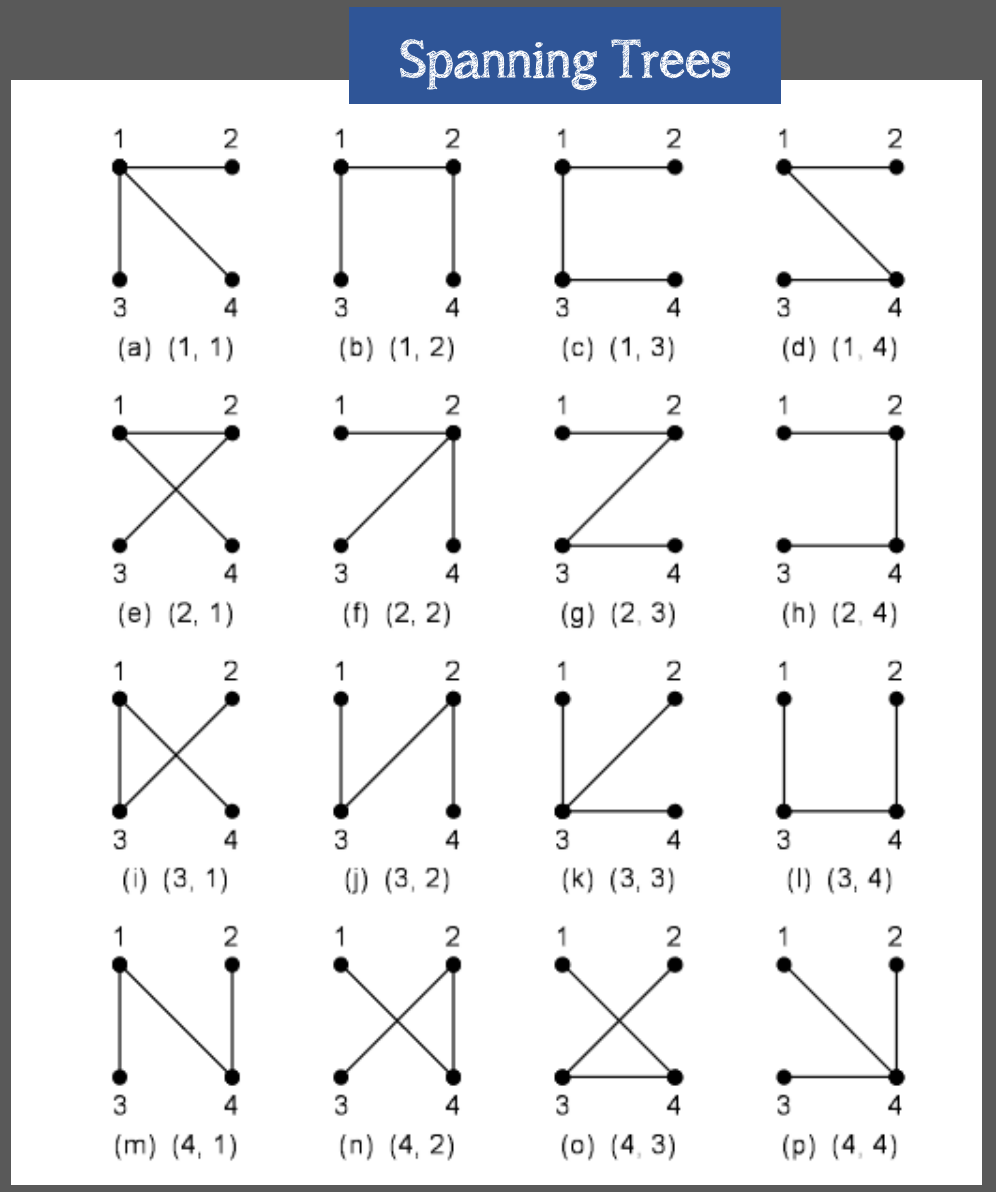
$T \subseteq E$

(V, T) is a tree



Spanning Trees

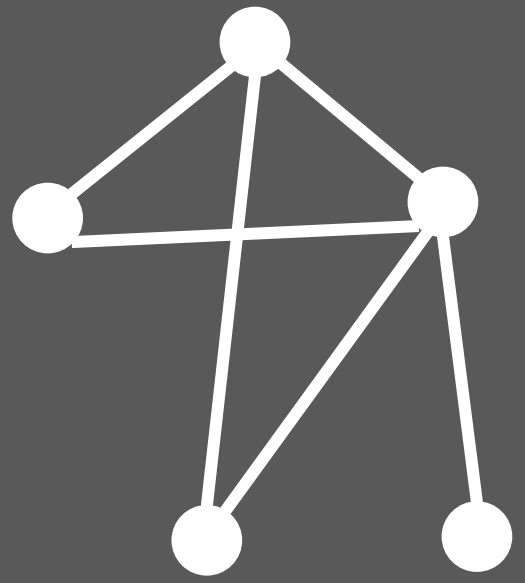




Finding a Spanning Tree

A subtractive method

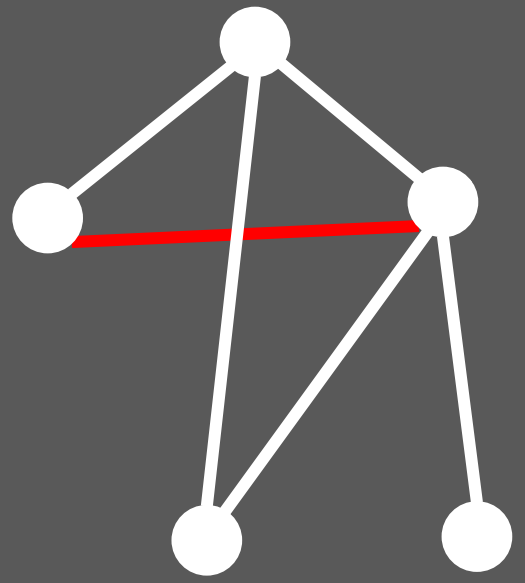
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected
- Repeat until no more cycles



Finding a Spanning Tree

A subtractive method

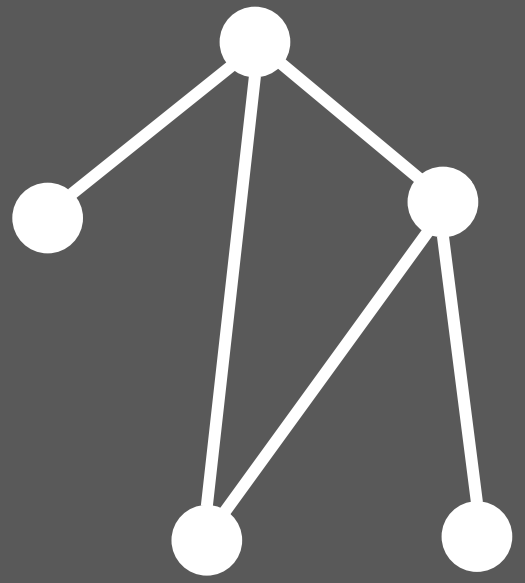
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected
- Repeat until no more cycles



Finding a Spanning Tree

A subtractive method

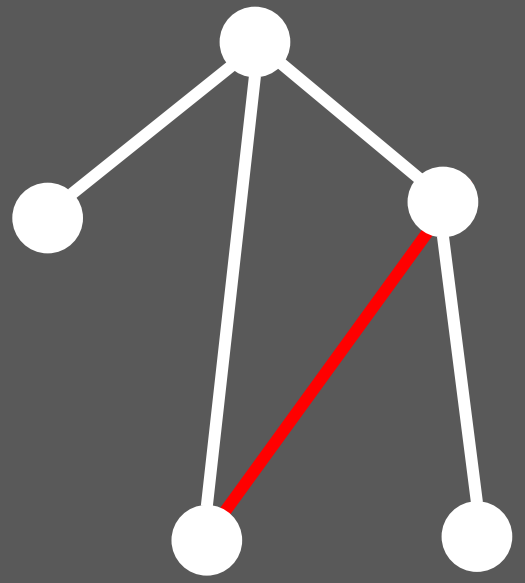
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected
- Repeat until no more cycles



Finding a Spanning Tree

A subtractive method

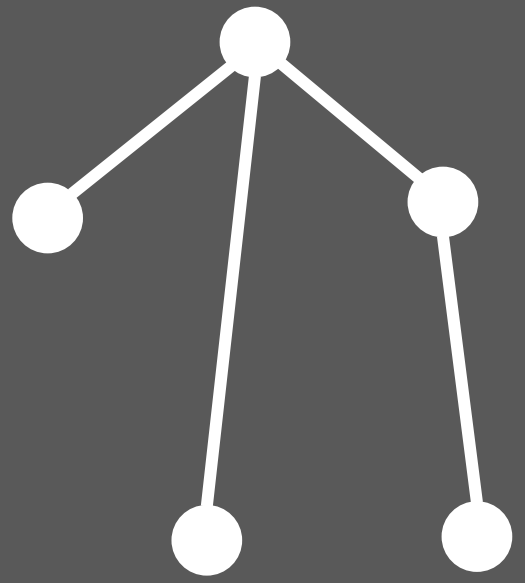
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected
- Repeat until no more cycles



Finding a Spanning Tree

A subtractive method

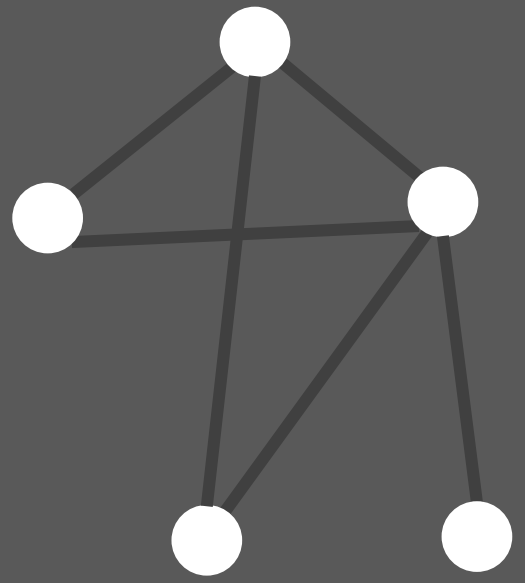
- Start with the whole graph – it is connected
- If there is a cycle, pick an edge on the cycle, throw it out – the graph is still connected
- Repeat until no more cycles



Finding a Spanning Tree

An additive method

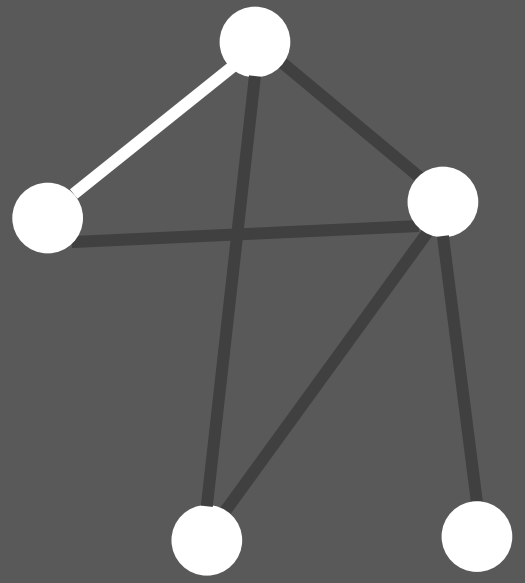
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them
- Repeat until only one component



Finding a Spanning Tree

An additive method

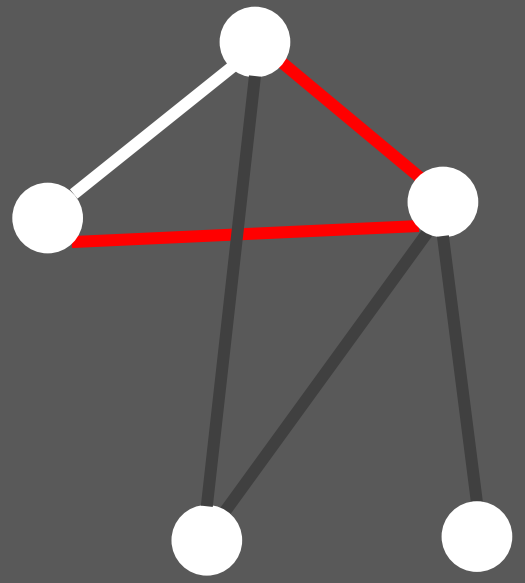
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them
- Repeat until only one component



Finding a Spanning Tree

An additive method

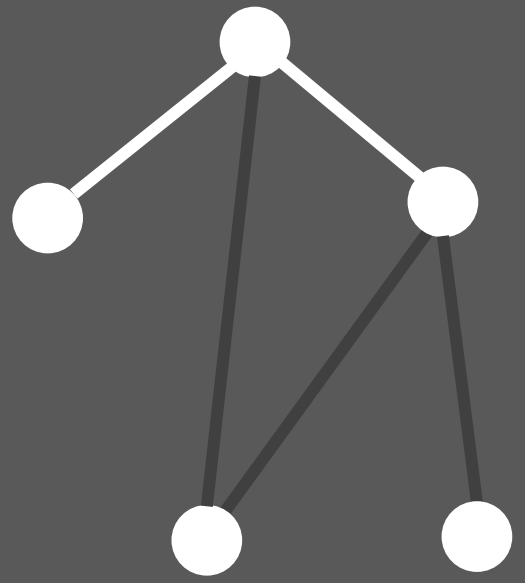
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them
- Repeat until only one component



Finding a Spanning Tree

An additive method

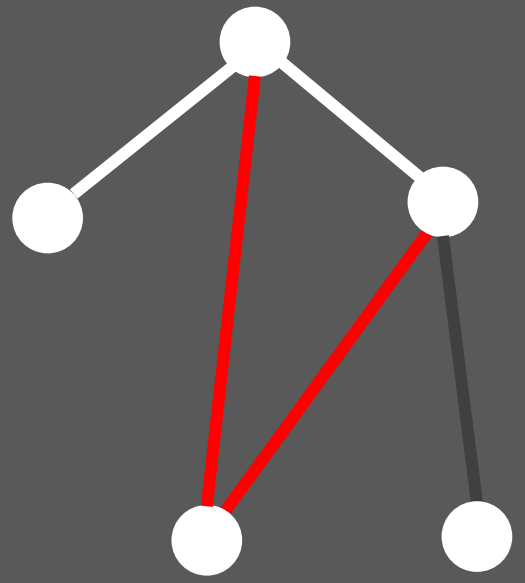
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them
- Repeat until only one component



Finding a Spanning Tree

An additive method

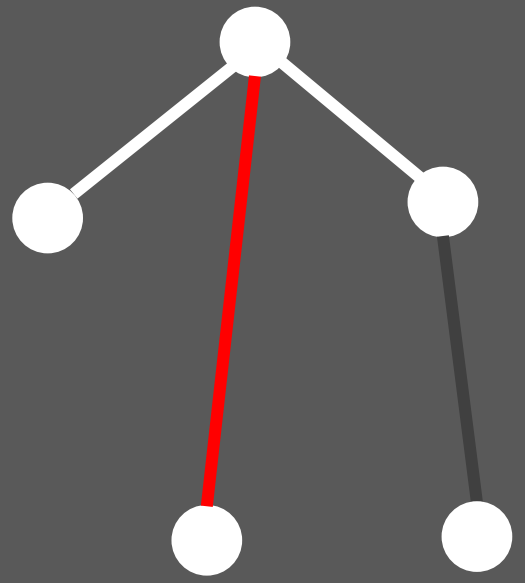
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them
- Repeat until only one component



Finding a Spanning Tree

An additive method

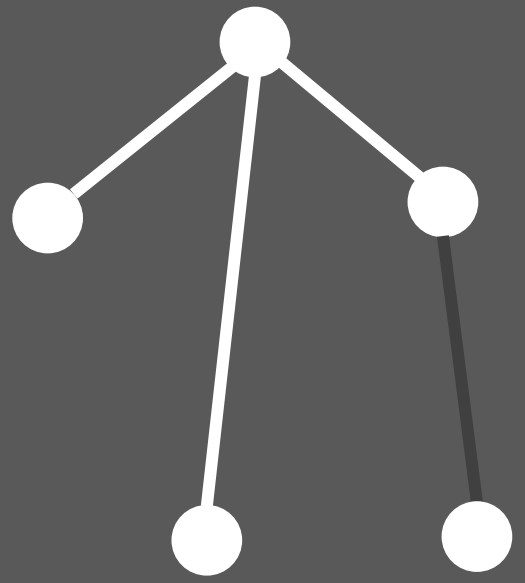
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them
- Repeat until only one component



Finding a Spanning Tree

An additive method

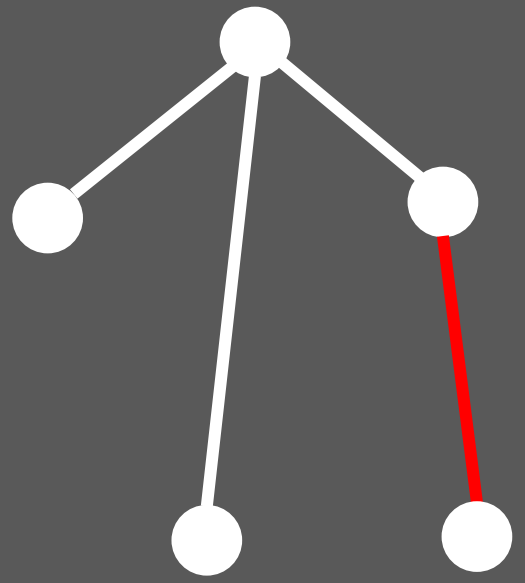
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them
- Repeat until only one component



Finding a Spanning Tree

An additive method

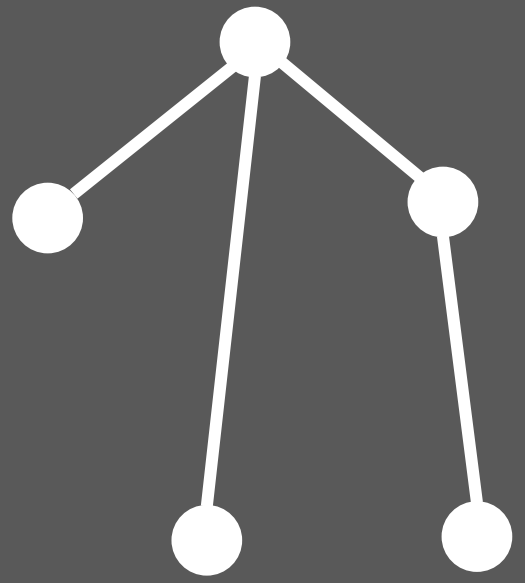
- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them
- Repeat until only one component

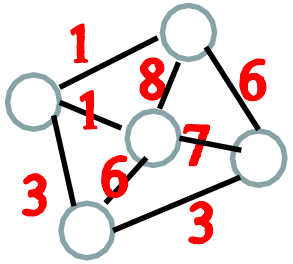


Finding a Spanning Tree

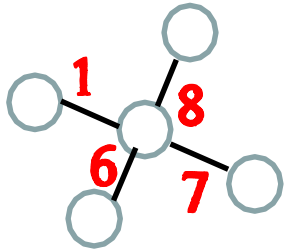
An additive method

- Start with no edges – there are no cycles
- If more than one connected component, insert an edge between them
- Repeat until only one component



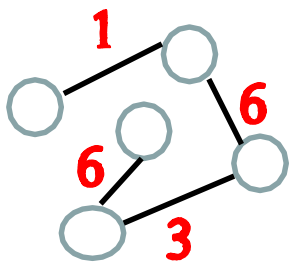


Consider the graph above with 5 nodes and 8 edges



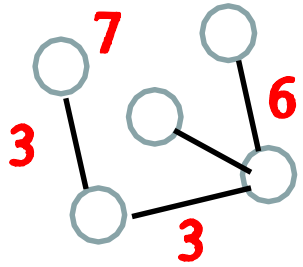
Cost =22

(a)



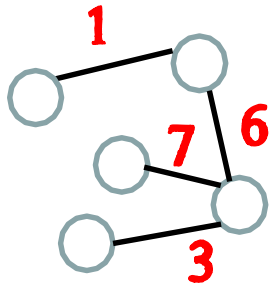
Cost =16

(b)



Cost =19

(c)

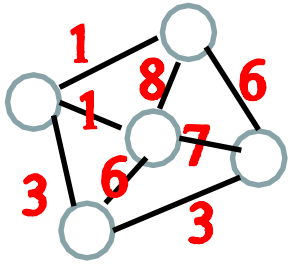


Cost =17

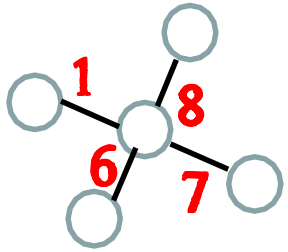
(d)

Minimum Spanning Tree

A minimum spanning tree of a weighted graph is a spanning tree that has the smallest sum of weights of its edges of all the spanning trees for that weighted graph.

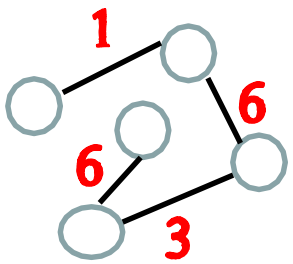


Consider the graph above with 5 nodes and 8 edges



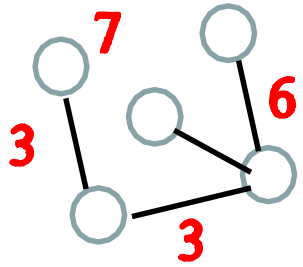
Cost =22

(a)



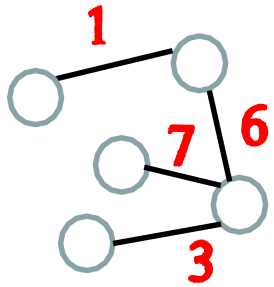
Cost =16

(b)



Cost =19

(c)



Cost =17

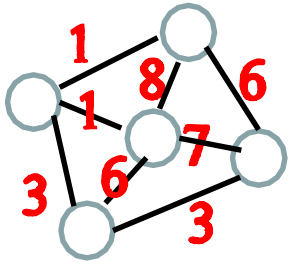
(d)

NB: Tree (b) is the minimum spanning tree (since it has minimum cost)

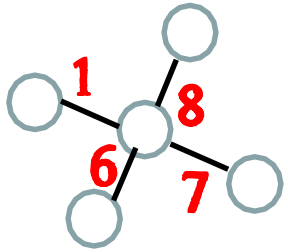
The MST may not be unique -

Minimum Spanning Tree

A minimum spanning tree of a weighted graph is a spanning tree that has the smallest sum of weights of its edges of all the spanning trees for that weighted graph.

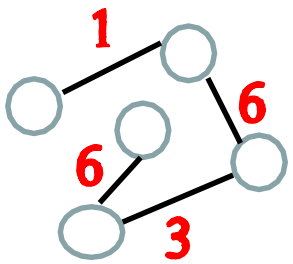


Consider the graph above with 5 nodes and 8 edges



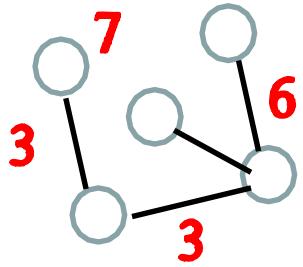
Cost =22

(a)



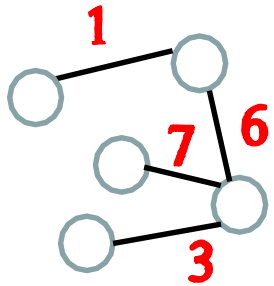
Cost =16

(b)



Cost =19

(c)



Cost =17

(d)

NB: Tree **(b)** is the minimum spanning tree (since it has minimum cost)

The MST may not be unique - more than one spanning tree could have the same minimum cost

Example of an MST problem

In a city there are N houses, each of which is in need of a water supply. It costs r[x] rands to build a well at house x, and it costs c[x, y] rands to build a pipe in between houses x and y. A house can receive water if either there is a well built there or there is some path of pipes to a house with a well. Design an algorithm to find the minimum amount of money needed to supply every house with water.

All the houses need to be connected, but at the LEAST cost.

Greedy Algorithm

Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point.

> Optimization is a process to find the best solution among alternatives

A Greedy algorithm makes short-sighted choices at each step to ensure that the objective function is optimized.

Greedy Algorithm

Greedy choice: the globally optimal solution can be found by selecting a locally optimal choice.

Greedy Algorithm works in an iterative manner, making one locally optimal (greedy) choice after another. The choice made by the algorithm may depend on earlier choices but not on the future.

The Greedy algorithm has only one shot to compute the optimal solution, so it has to make a decision in the moment AND that it never goes back to change the decision.

Greedy Algorithm

PROS

simple and fast

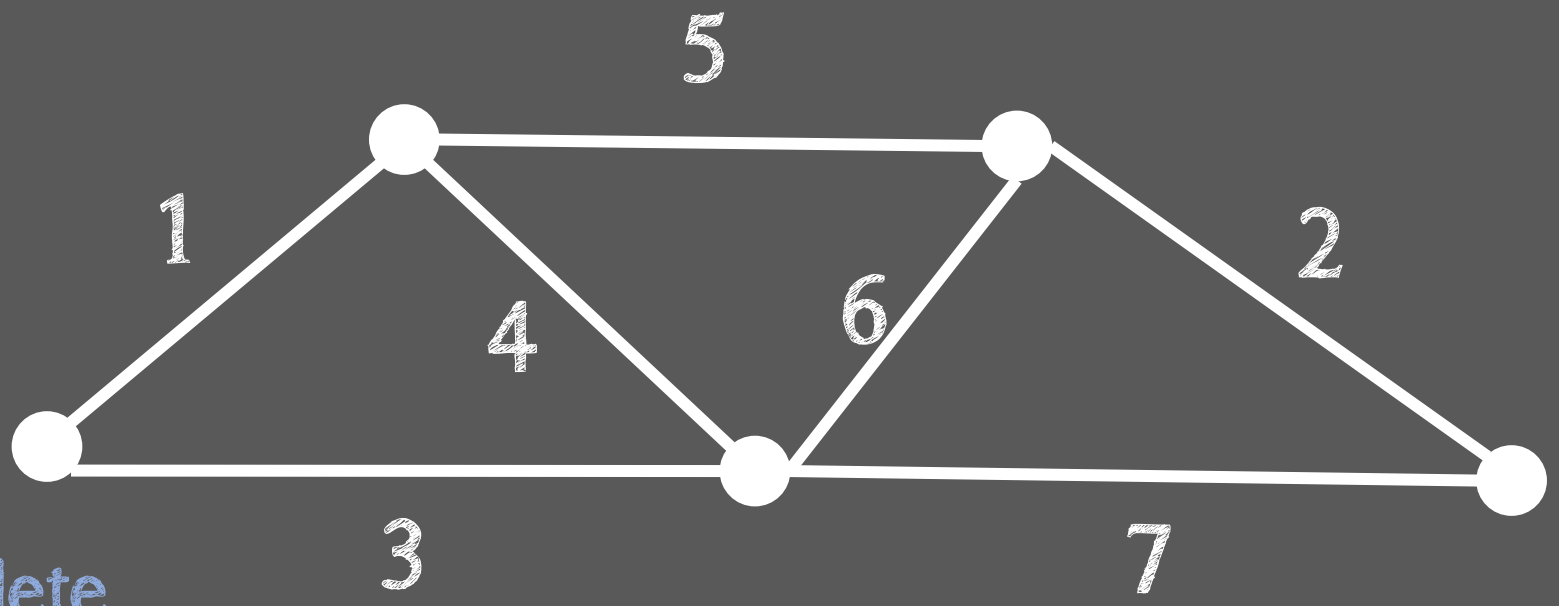
CONS

Doesn't always find a global optimal solution

Usually hard to prove its correctness

Greedy Algorithms for Finding Minimum Spanning Tree

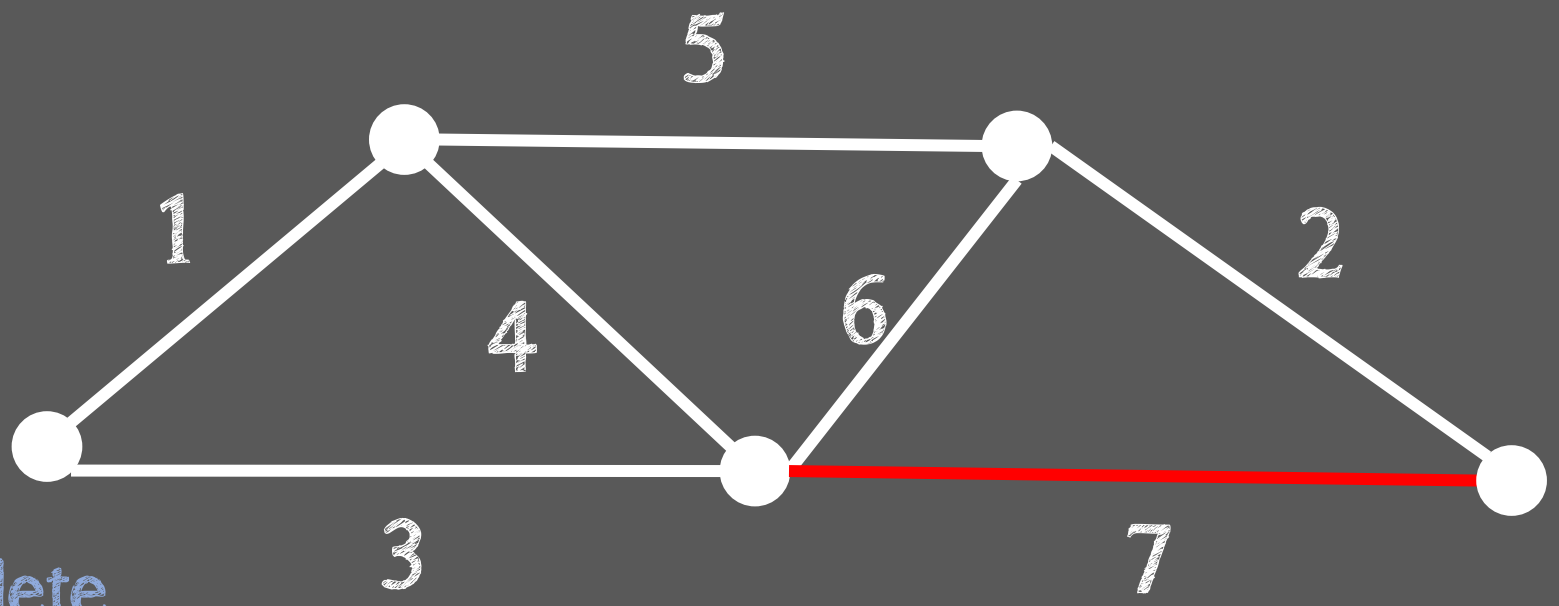
A Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



Reverse-delete Algorithm

Greedy Algorithms for Finding Minimum Spanning Tree

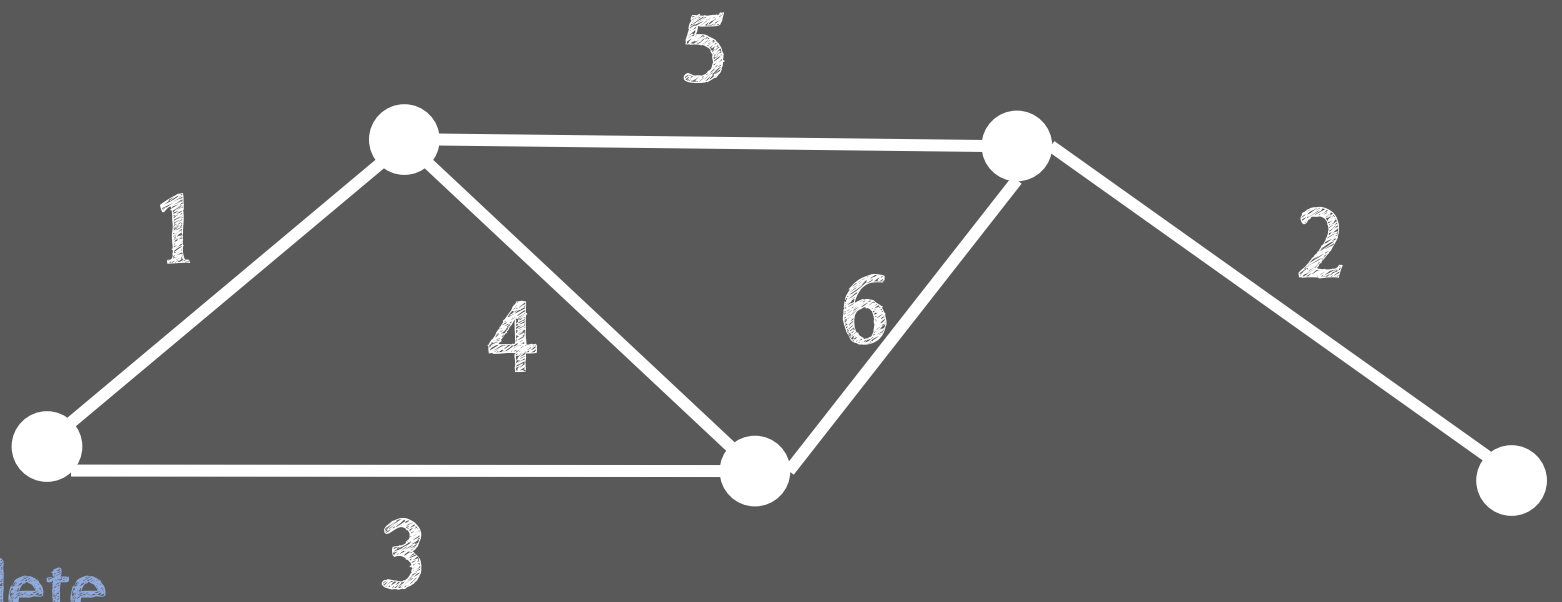
A Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



Reverse-delete Algorithm

Greedy Algorithms for Finding Minimum Spanning Tree

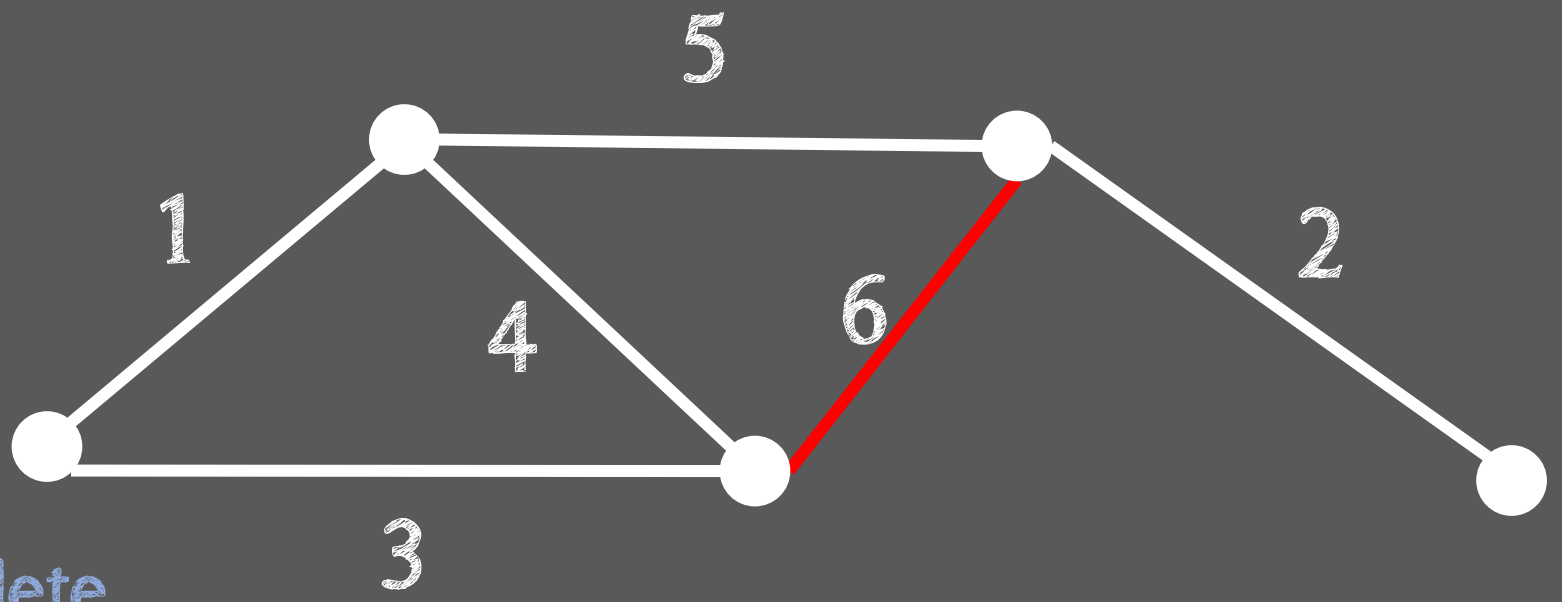
A Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



Reverse-delete Algorithm

Greedy Algorithms for Finding Minimum Spanning Tree

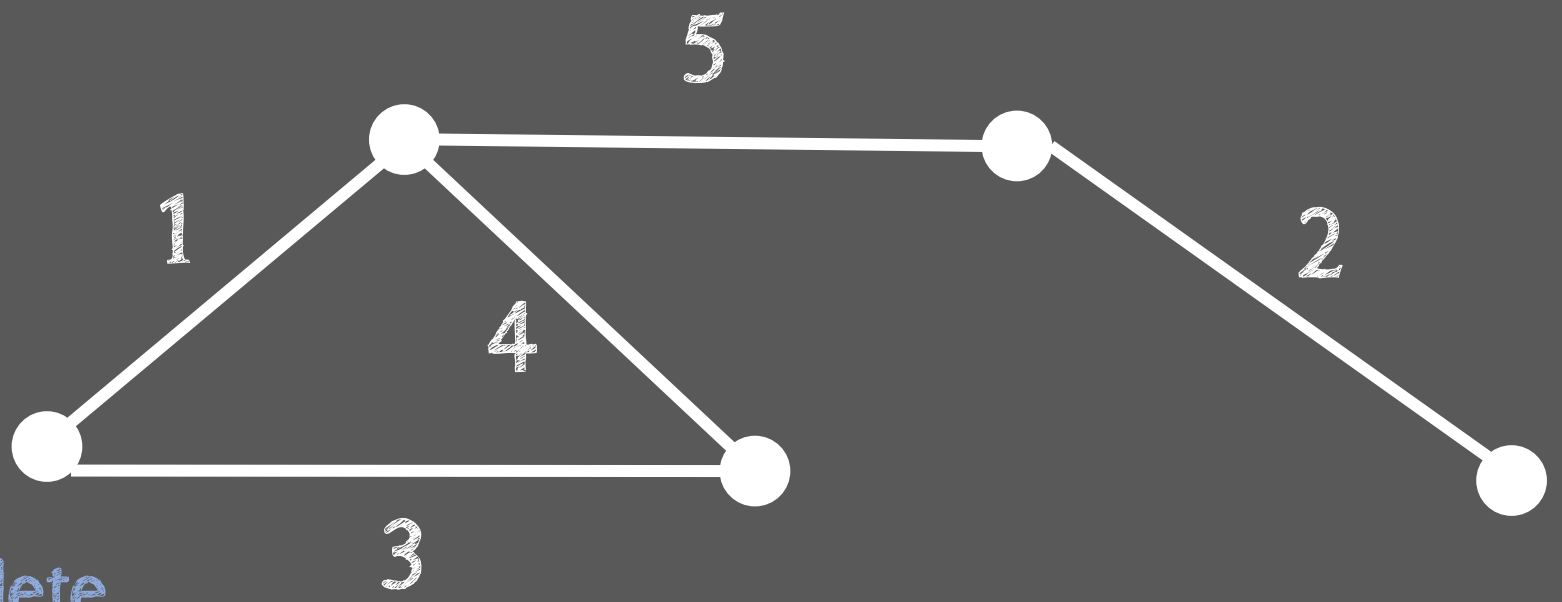
A Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



Reverse-delete Algorithm

Greedy Algorithms for Finding Minimum Spanning Tree

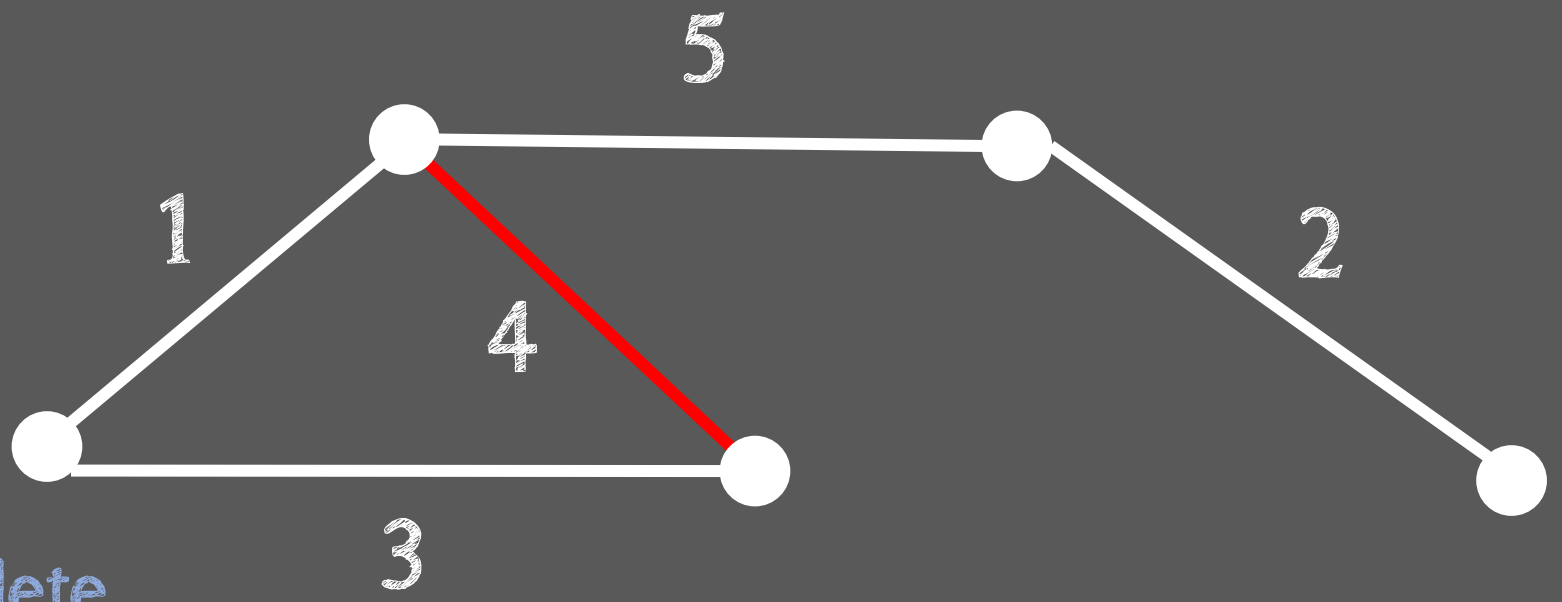
A Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



Reverse-delete Algorithm

Greedy Algorithms for Finding Minimum Spanning Tree

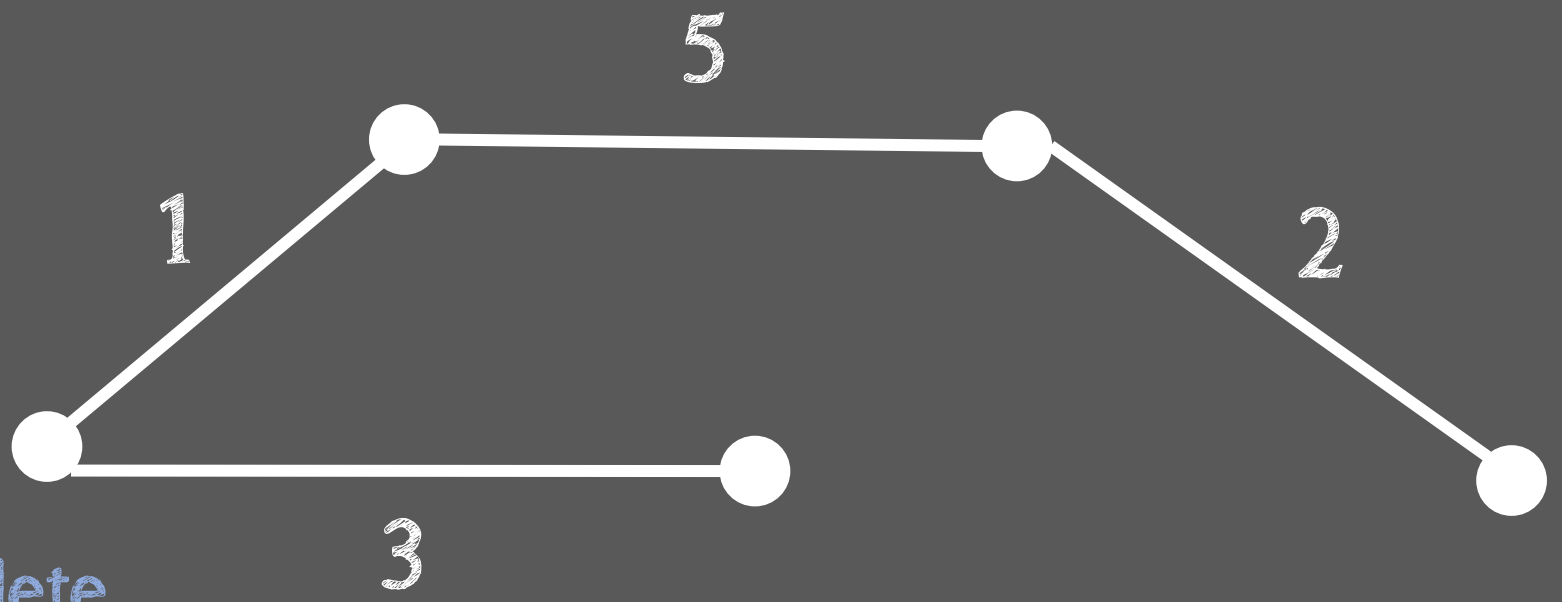
A Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



Reverse-delete Algorithm

Greedy Algorithms for Finding Minimum Spanning Tree

A Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it



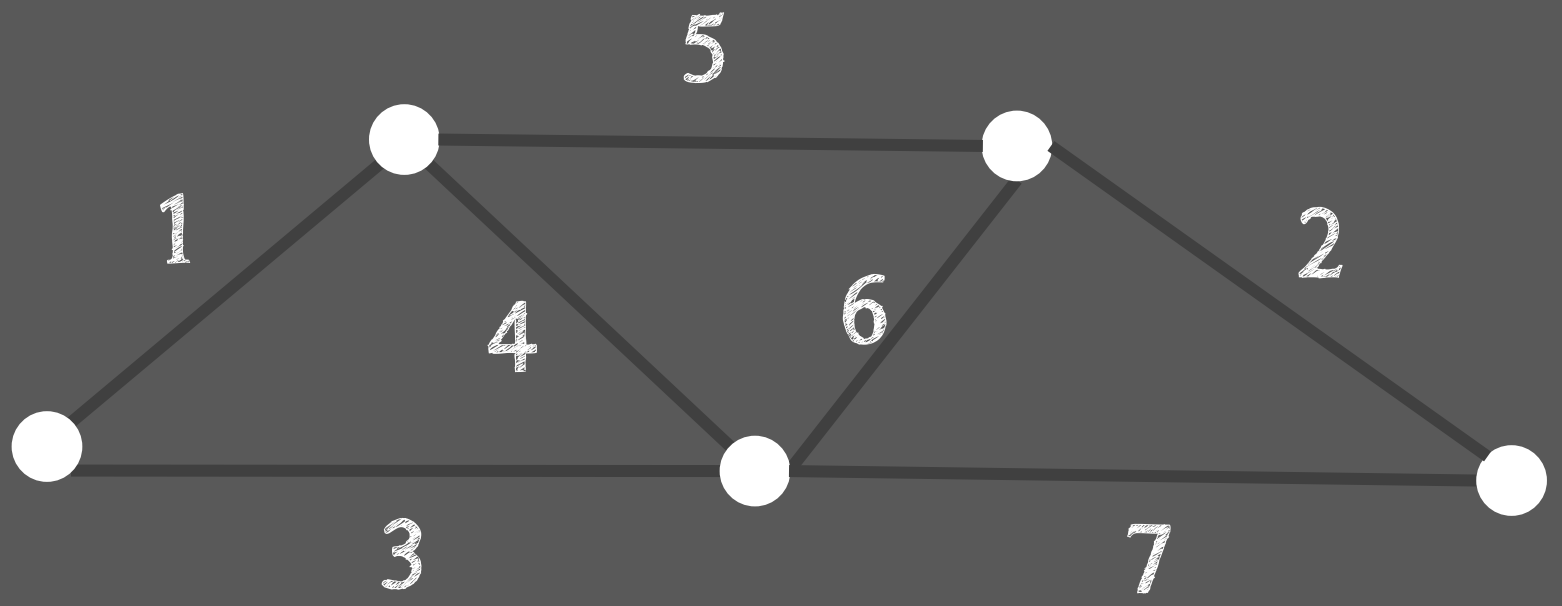
Reverse-delete Algorithm

Greedy Algorithms for Finding Minimum Spanning Tree

B

Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's Algorithm

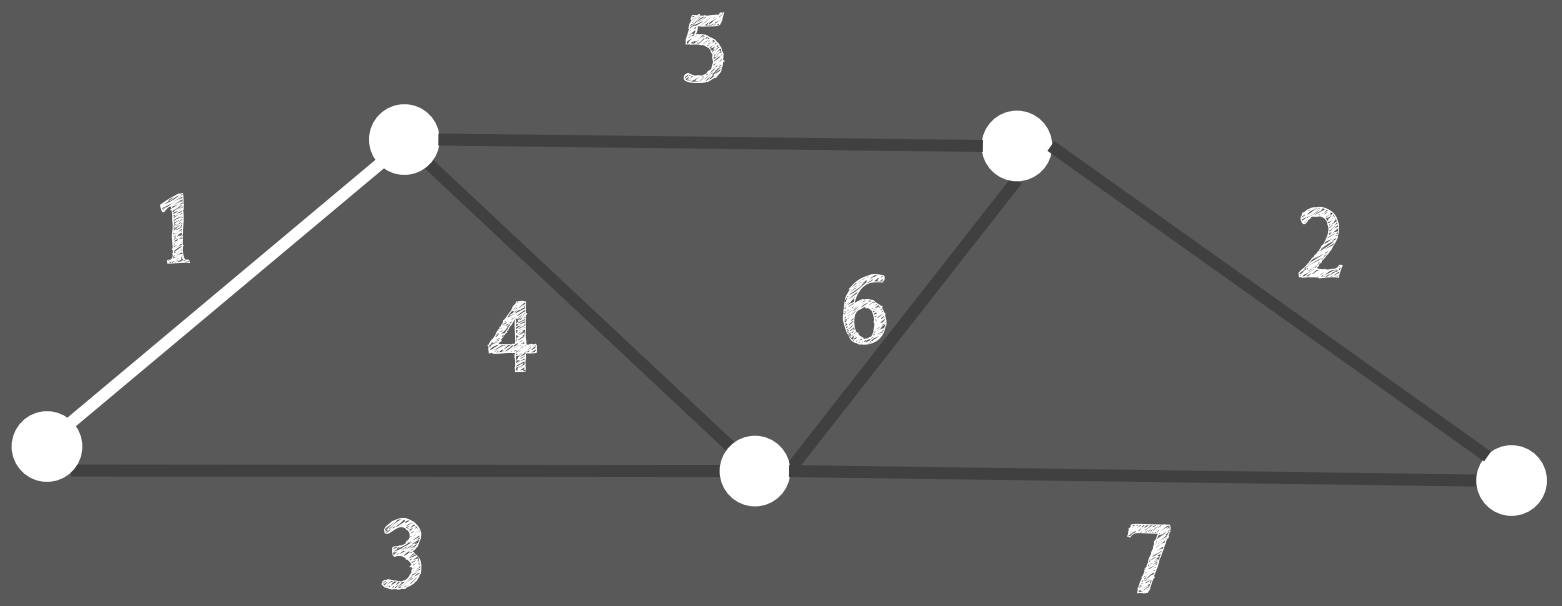


Greedy Algorithms for Finding Minimum Spanning Tree

B

Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

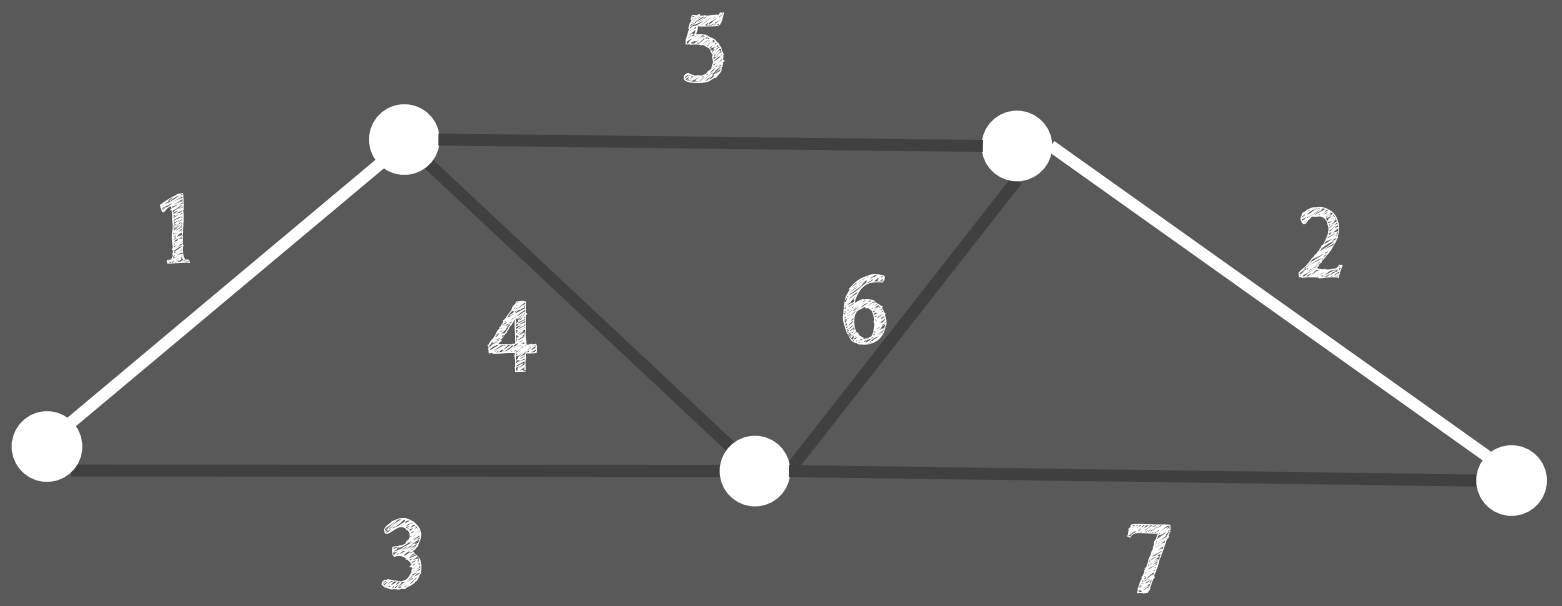
Kruskal's Algorithm



Greedy Algorithms for Finding Minimum Spanning Tree

B Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's Algorithm

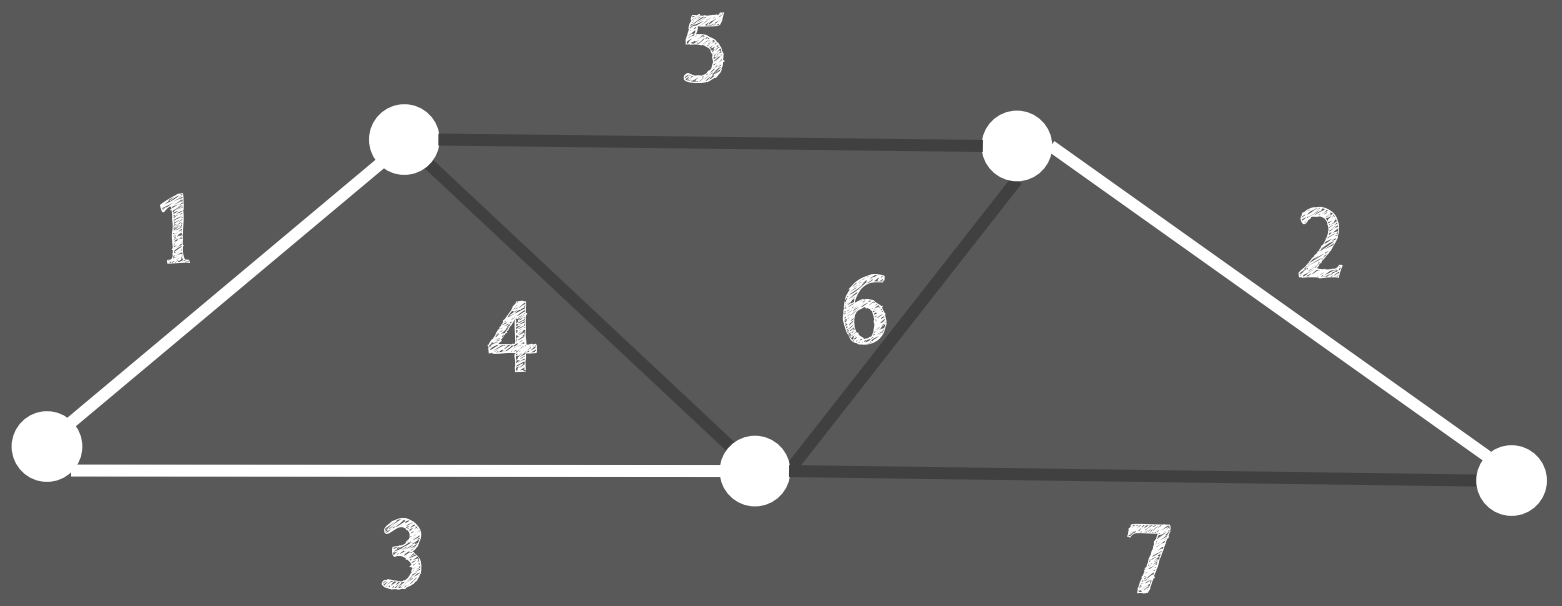


Greedy Algorithms for Finding Minimum Spanning Tree

B

Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's Algorithm

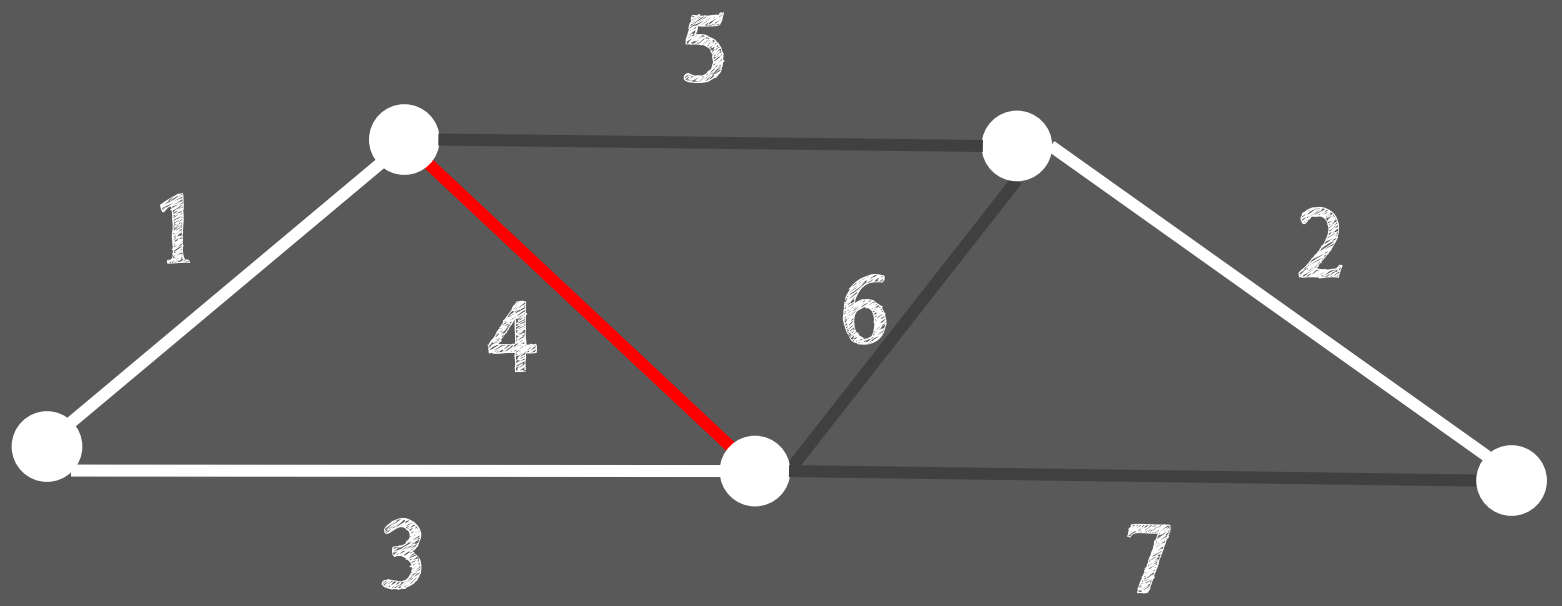


Greedy Algorithms for Finding Minimum Spanning Tree

B

Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's Algorithm

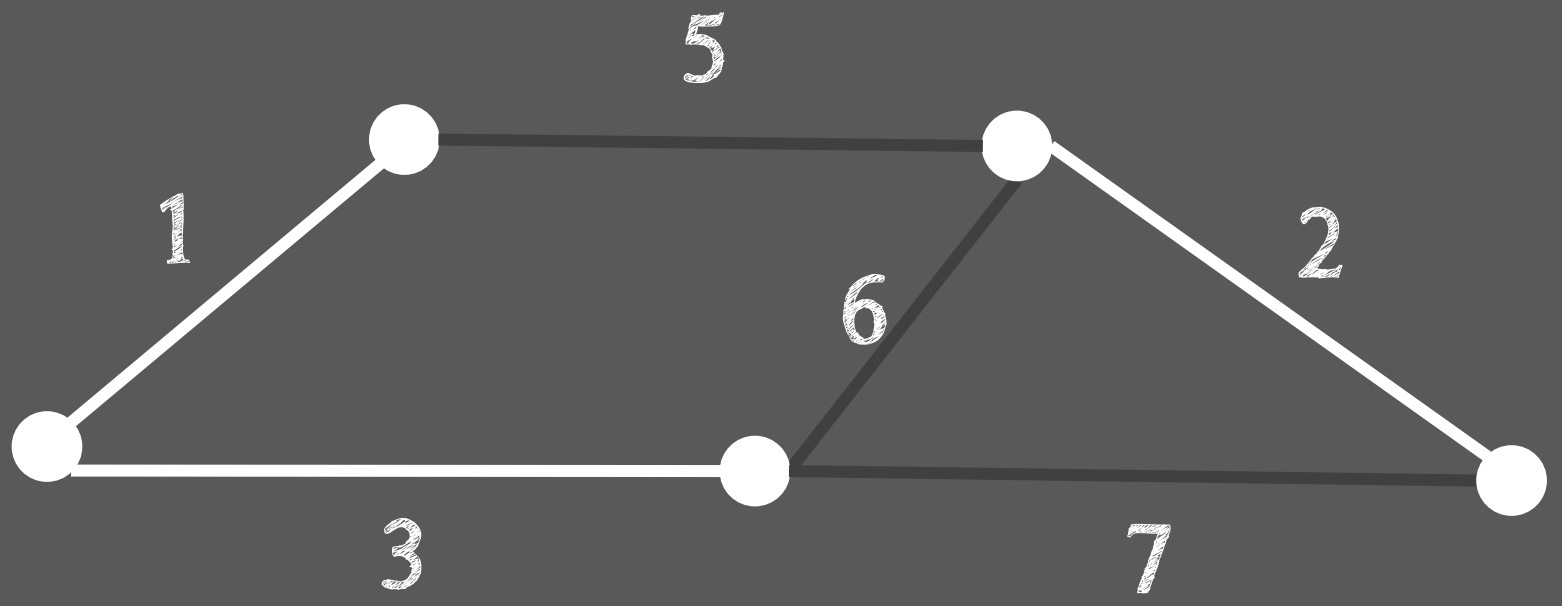


Greedy Algorithms for Finding Minimum Spanning Tree

B

Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's Algorithm

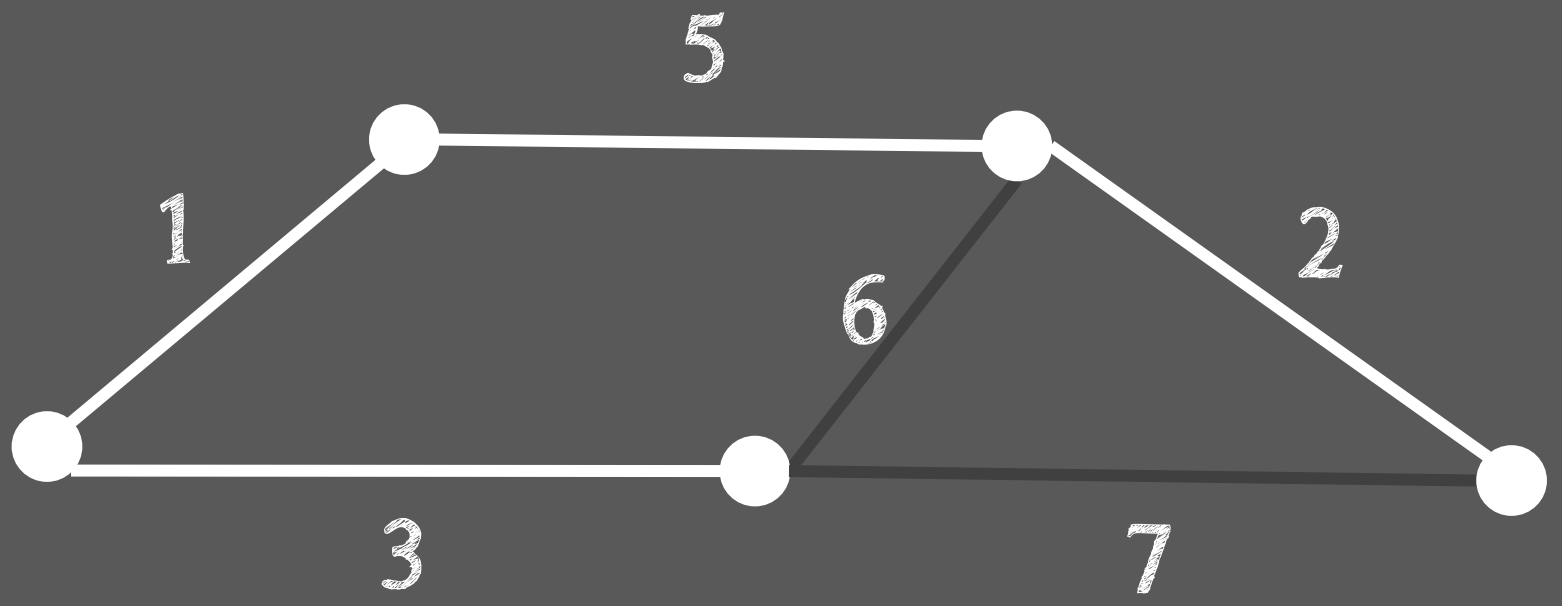


Greedy Algorithms for Finding Minimum Spanning Tree

B

Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's Algorithm

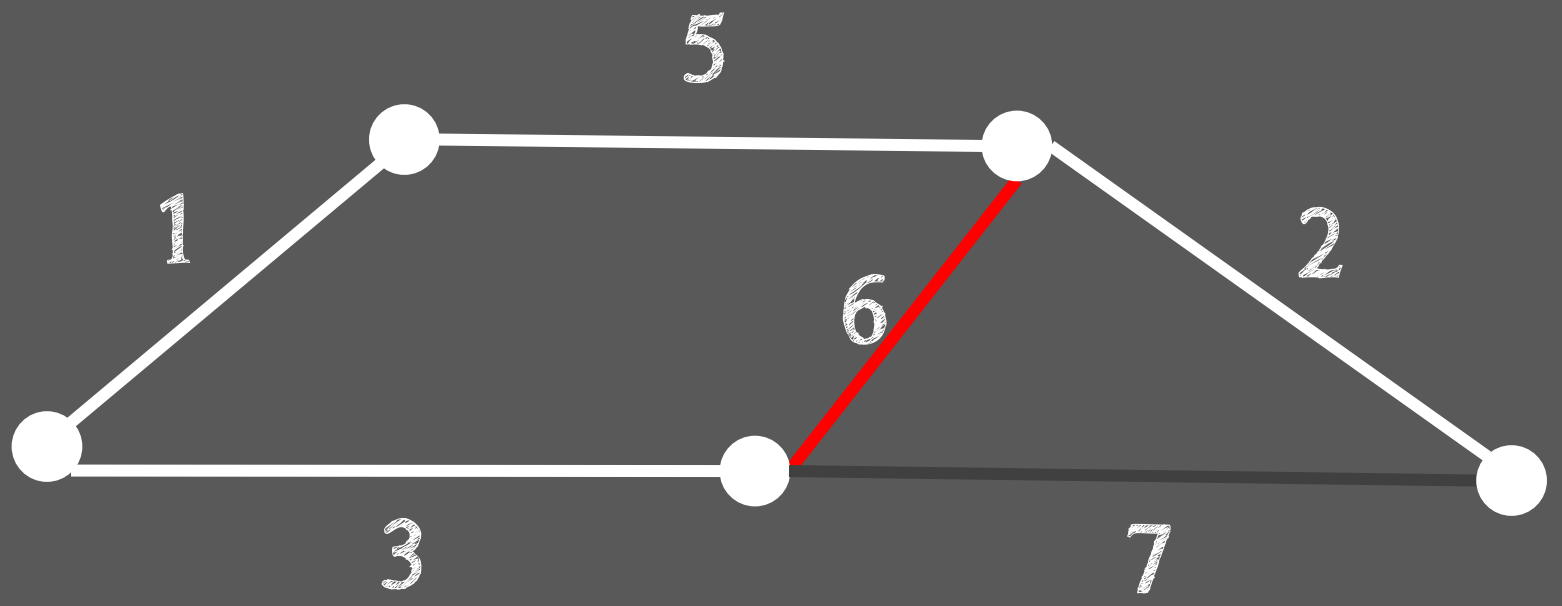


Greedy Algorithms for Finding Minimum Spanning Tree

B

Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's Algorithm

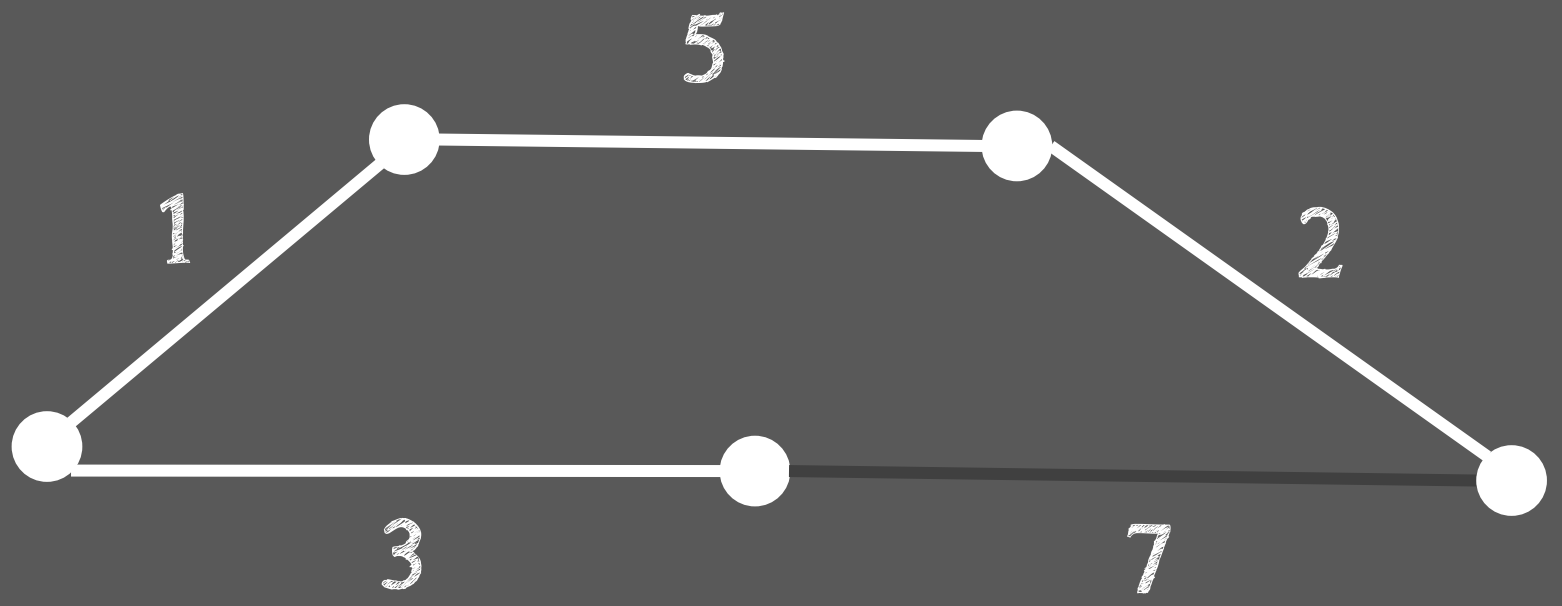


Greedy Algorithms for Finding Minimum Spanning Tree

B

Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's Algorithm

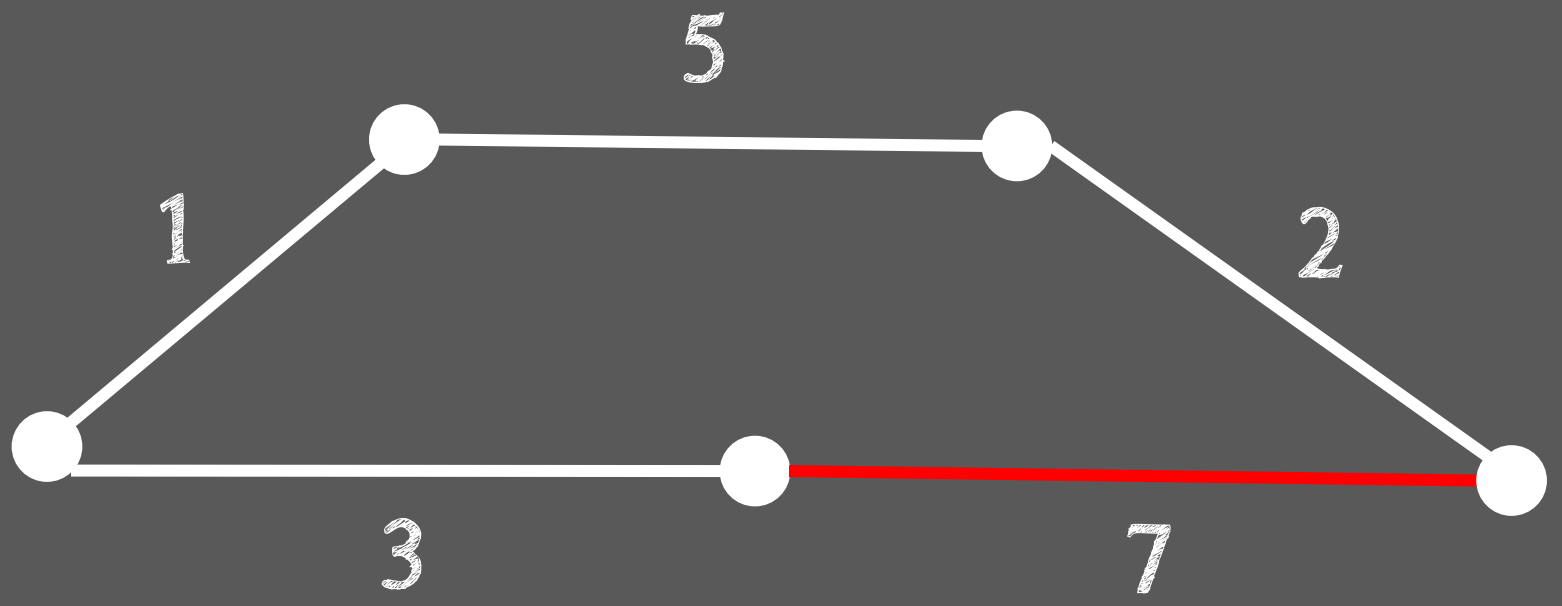


Greedy Algorithms for Finding Minimum Spanning Tree

B

Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's Algorithm

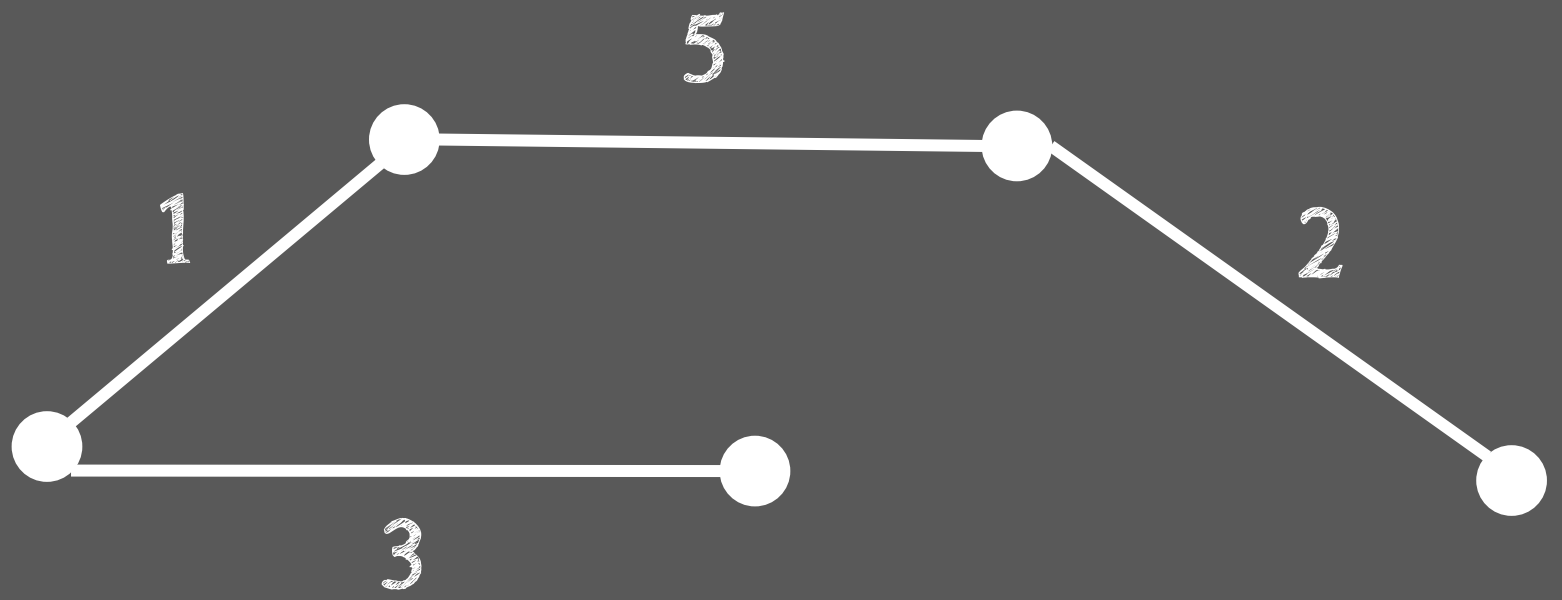


Greedy Algorithms for Finding Minimum Spanning Tree

B

Find a min weight edge
- if it forms a cycle with edges already taken, throw it out, otherwise keep it

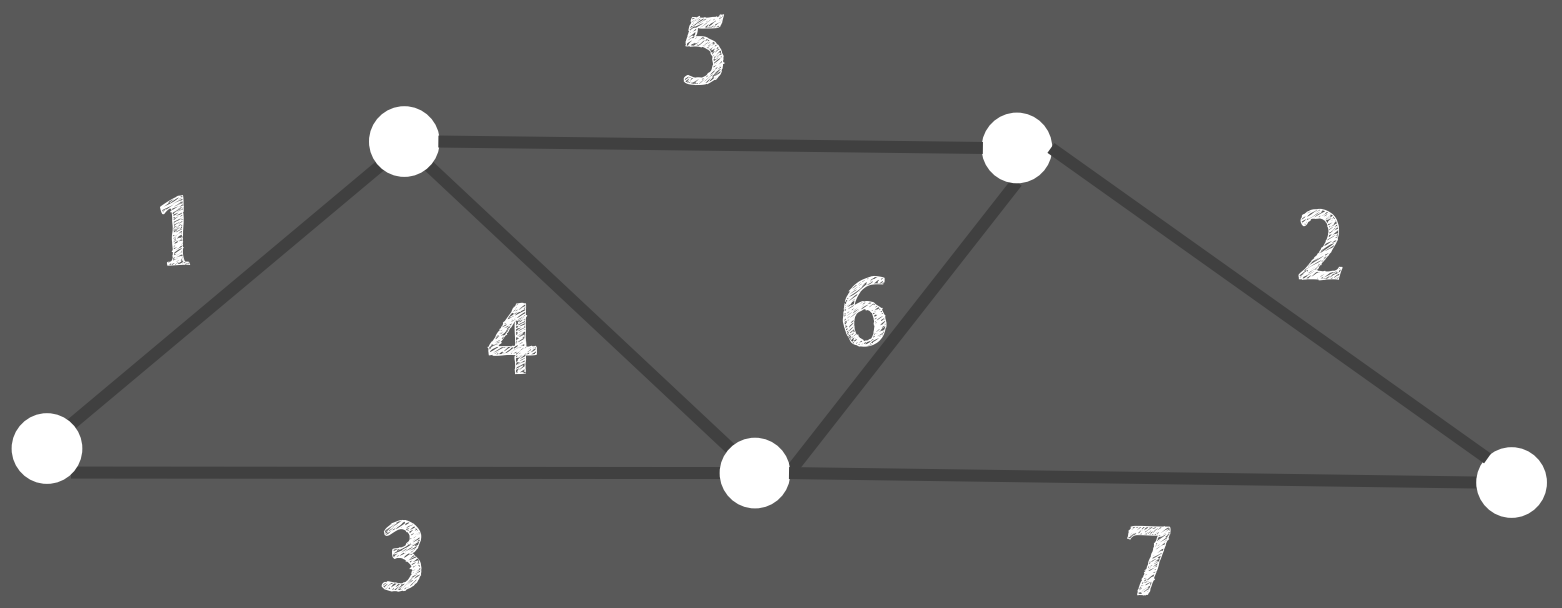
Kruskal's Algorithm



Greedy Algorithms for Finding Minimum Spanning Tree

C Start with any vertex, add min weight edge extending that connected component that does not form a cycle

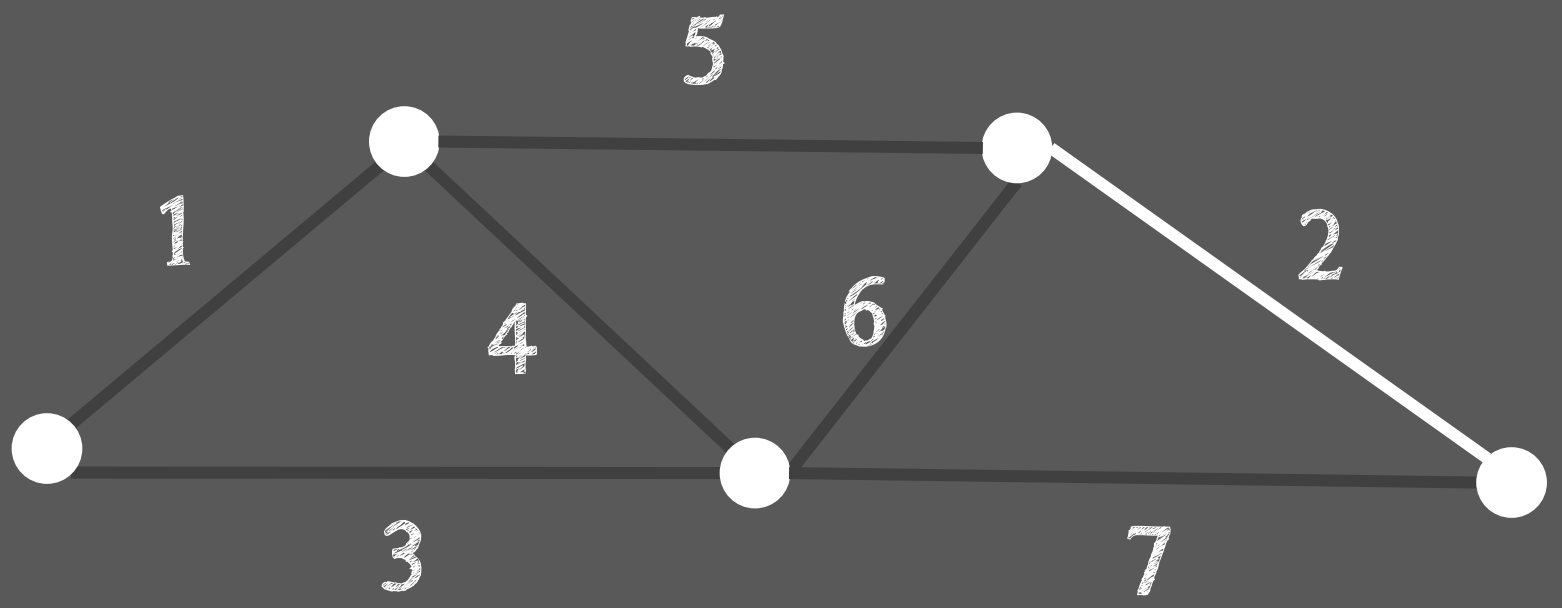
Prim's Algorithm



Greedy Algorithms for Finding Minimum Spanning Tree

C Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's Algorithm

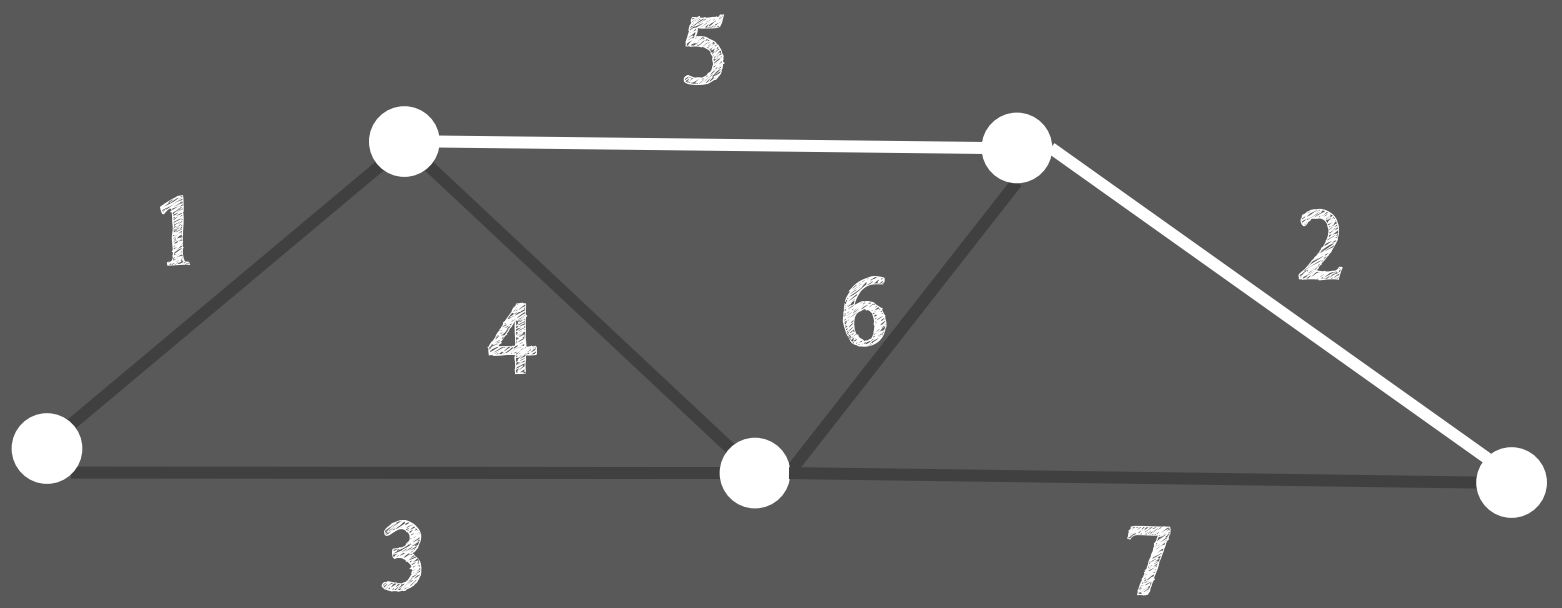


Greedy Algorithms for Finding Minimum Spanning Tree

C

Start with any vertex, add min weight edge extending that connected component that does not form a cycle

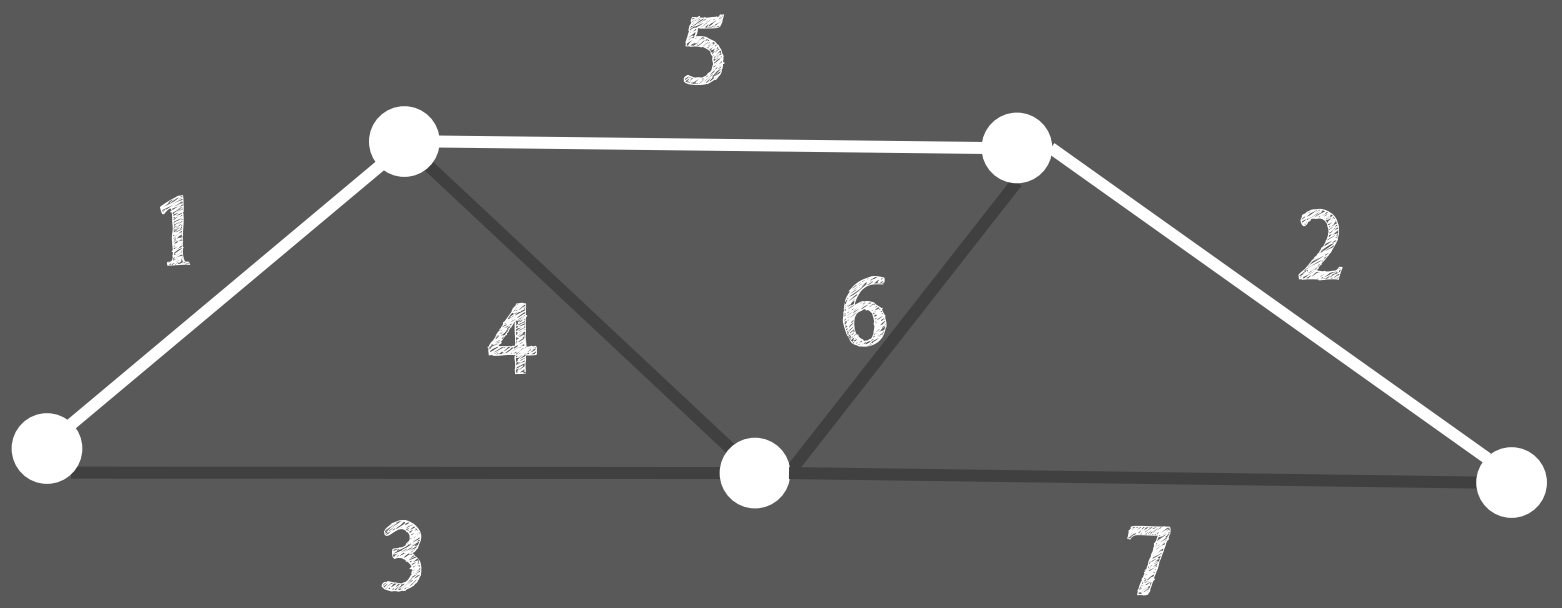
Prim's Algorithm



Greedy Algorithms for Finding Minimum Spanning Tree

C Start with any vertex, add min weight edge extending that connected component that does not form a cycle

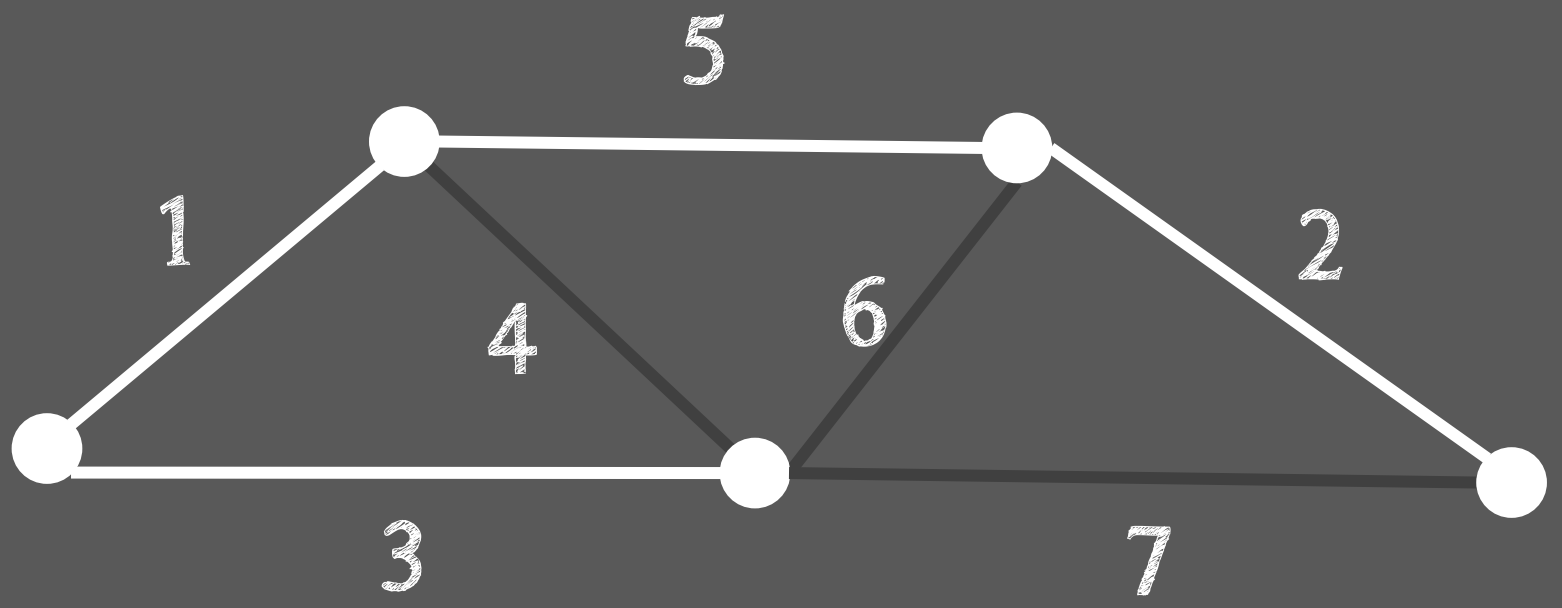
Prim's Algorithm



Greedy Algorithms for Finding Minimum Spanning Tree

C Start with any vertex, add min weight edge extending that connected component that does not form a cycle

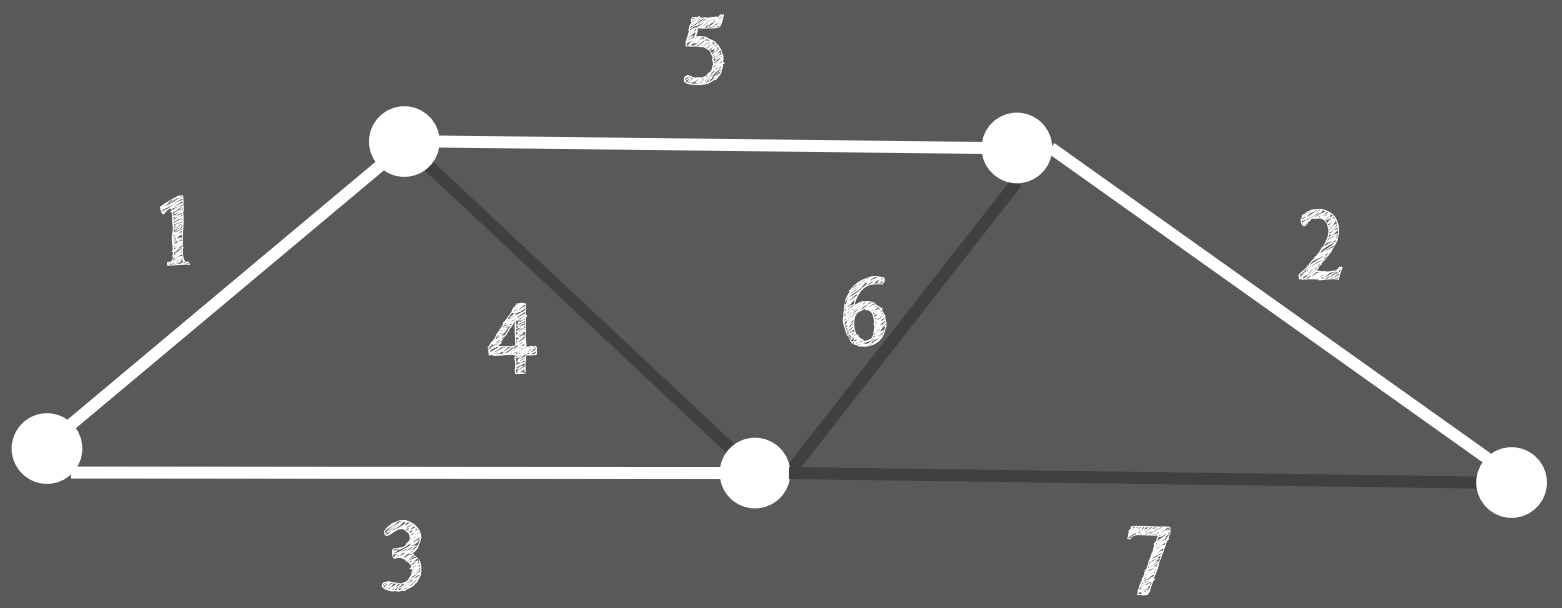
Prim's Algorithm



Greedy Algorithms for Finding Minimum Spanning Tree

C Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's Algorithm



Data Structure

PRIORITY QUEUE

A priority queue data structure keeps track of the smallest edge connecting T to each vertex not yet in T.

It is equipped with (at least) three operations:

insert(*elem, key*) inserts the given element-key pair into the queue

extractmin() removes the element with the minimum key from the priority queue, and returns the (*elem, key*) pair.

decreasekey(*elem, newkey*) changes the key of the element *elem* from its current key to min(*originalkey*, *newkey*),

It is usually used in implementing Dijkstra and Prim's algorithm

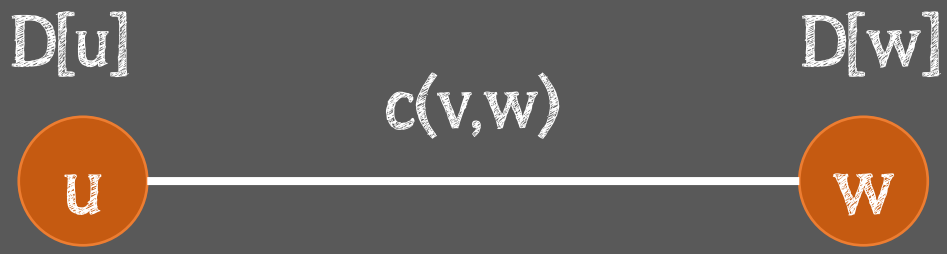
Data Structure

PRIORITY QUEUE

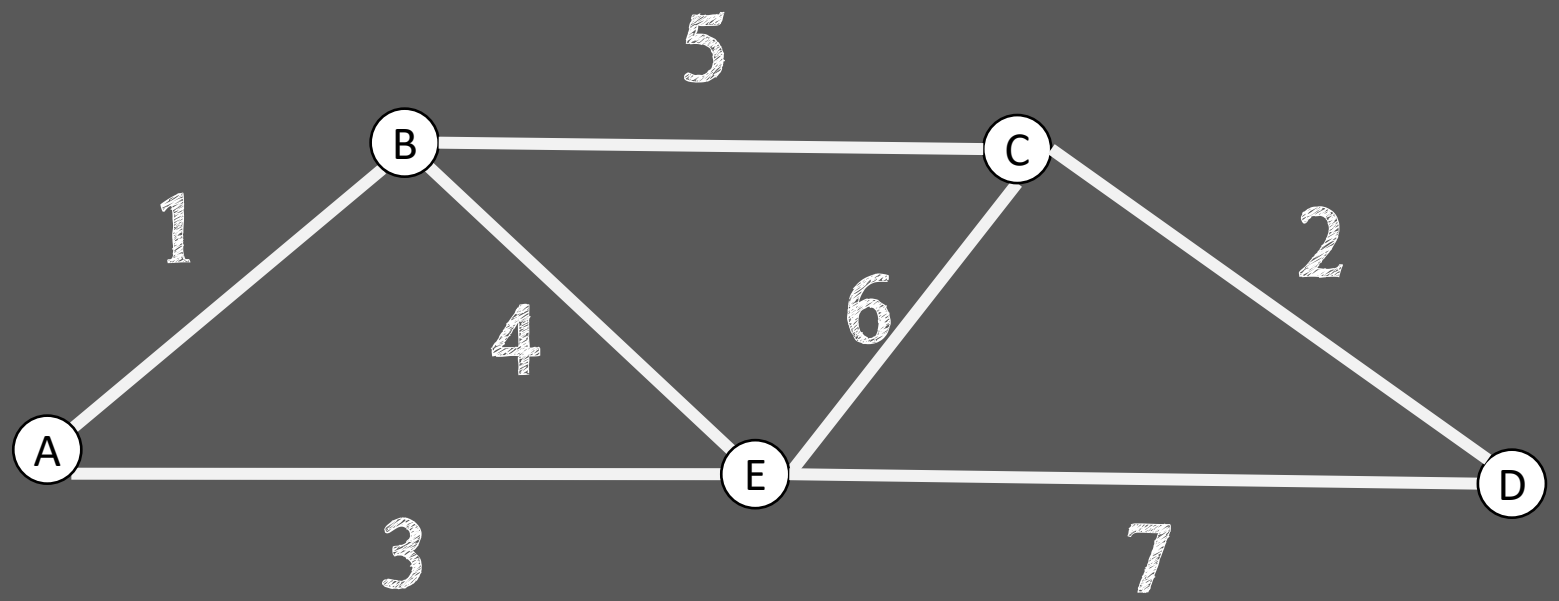
Similar pseudocode structure

```
while (some vertices unmarked) {
    u = best of unmarked vertices;
    mark u;
    for (each v adj to u) {
        update v;
    }
}
```

- Dijkstra
  - best: next in PQ
  - update: D[w] = min (D[w], D[u]+c(v,w))
- Prim
  - best: next in PQ
  - update: D[w] = min (D[w], c(v,w))



```
while (some vertices unmarked) {
   u = best of unmarked vertices;
   mark u;
   for (each v adj to u) {
      update v;
   }
}
```



DIJKSTRA (shortest path)

update: D[w] = min(D[w], D[v]+c(v,w))

S={}

PQ= {A(0), B(∞), C(∞), D(∞), E(∞)}

S={A}

PQ= {B(1), C(∞), D(∞), E(3)}

S={A, B}

PQ= {C(6), D(∞), E(3)}

PRIM (minimum spanning tree

update: D[w] = min(D[w], c(v,w))

S={}

PQ= {A(0), B(∞), C(∞), D(∞), E(∞)}

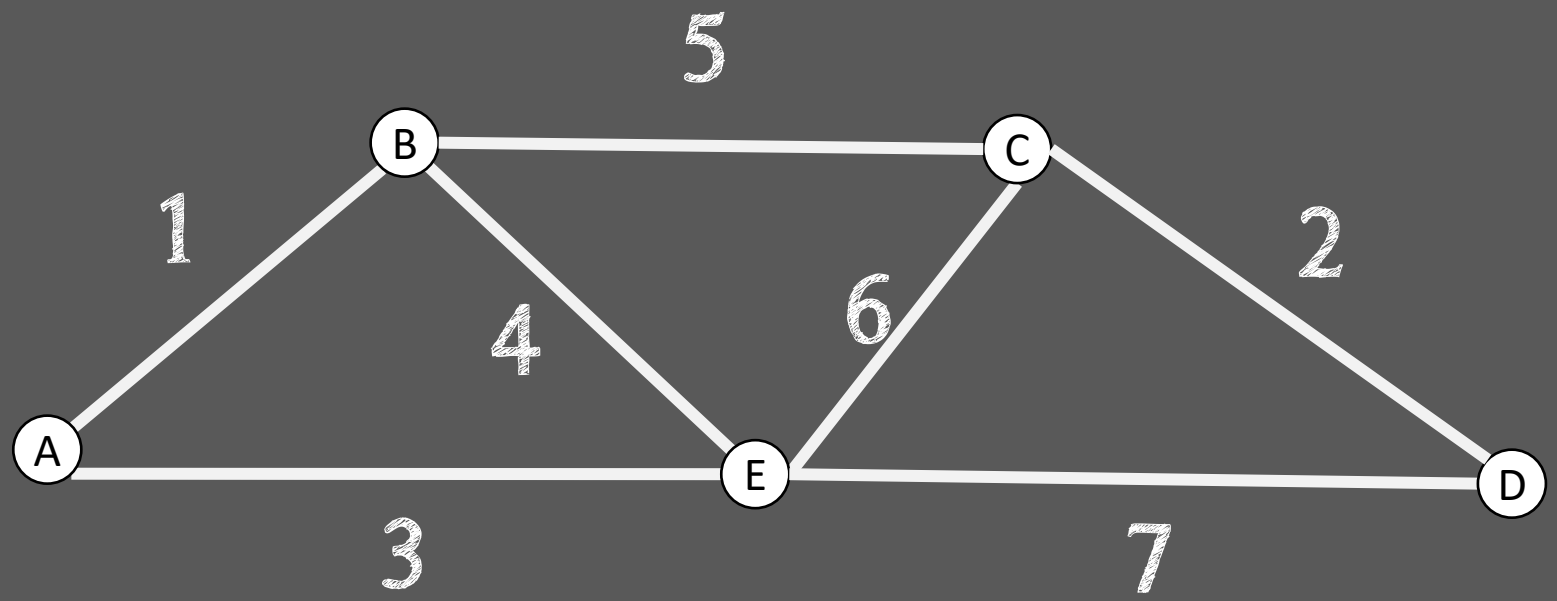
S={A}

PQ= {B(1), C(∞), D(∞), E(3)}

S={A, B}

PQ= {C(5), D(∞), E(3)}

```
while (some vertices unmarked) {
   u = best of unmarked vertices;
   mark u;
   for (each v adj to u) {
      update v;
   }
}
```



DIJKSTRA (shortest path)

update: D[w] = min(D[w], D[v]+c(v,w))

S={A, B, E}
PQ= {C(6), D(10)}

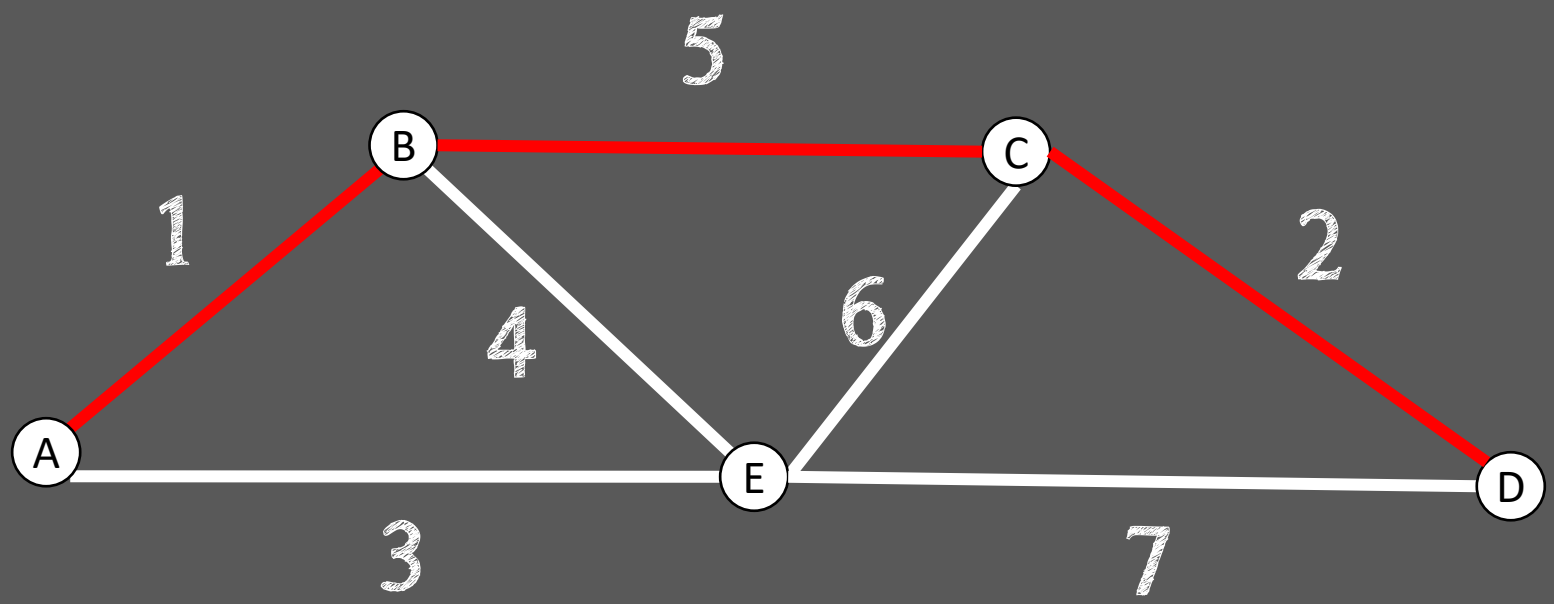
S={A, B, E, C}
PQ= {D(8)}

PRIM (minimum spanning tree

update: D[w] = min(D[w], c(v,w))

S={A, B, E}
PQ= {C(5), D(7))}

S={A, B, E, C}
PQ= {D(2))}

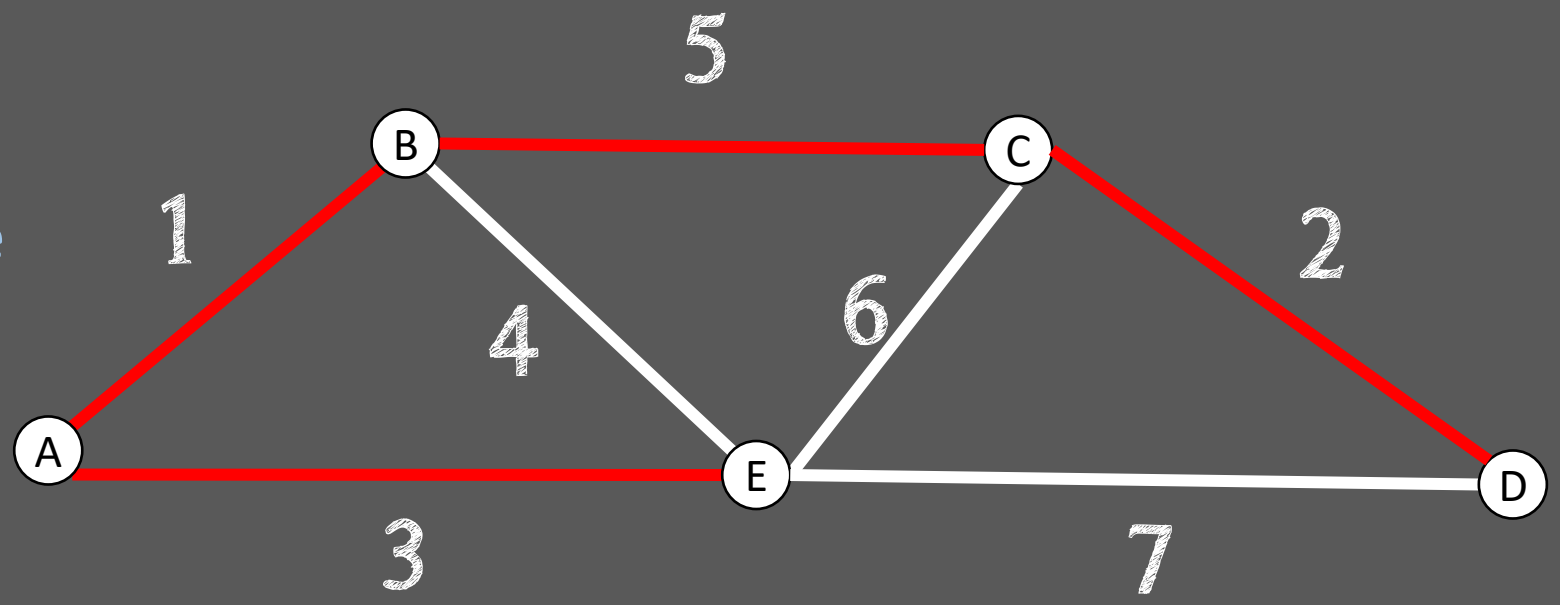


DIJKSTRA
(shortest path from A to D
marked in red)

Cost of
shortest path
is 8

PRIM (minimum spanning tree
marked in red)

Cost of
minimum
spanning tree
is 10



Data Structure

Union-Find

A disjoint Union-Find data structure keeps track of which vertex is in which component.

It is equipped with (at least) three operations:

makeset(*elem*), which takes an element *elem* and creates a new singleton set for it

find(*elem*), which finds the canonical representative for the set containing the element

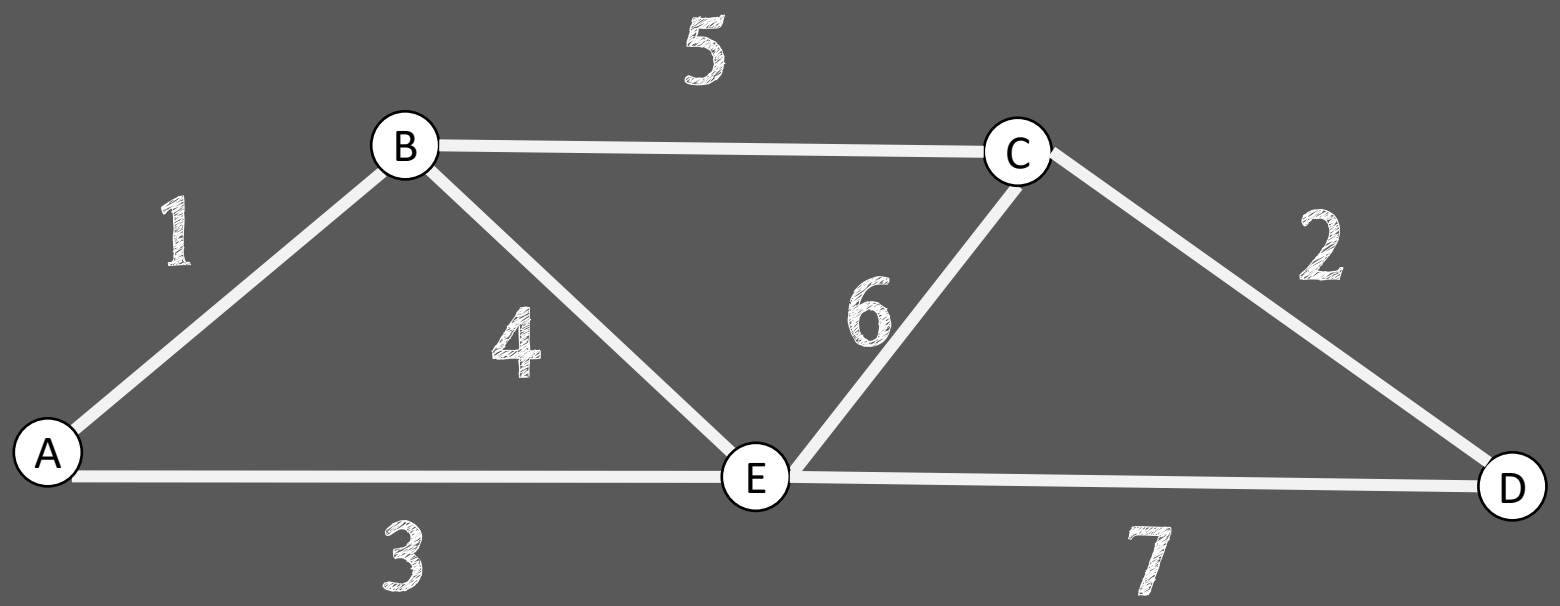
union(*elem1*,*elem2*), which merges the two sets that *elem1* and *elem2* are in

It is usually used in implementing Kruskal's algorithm

KRUSKAL (minimum spanning tree)

First, sort the edges in ascending order

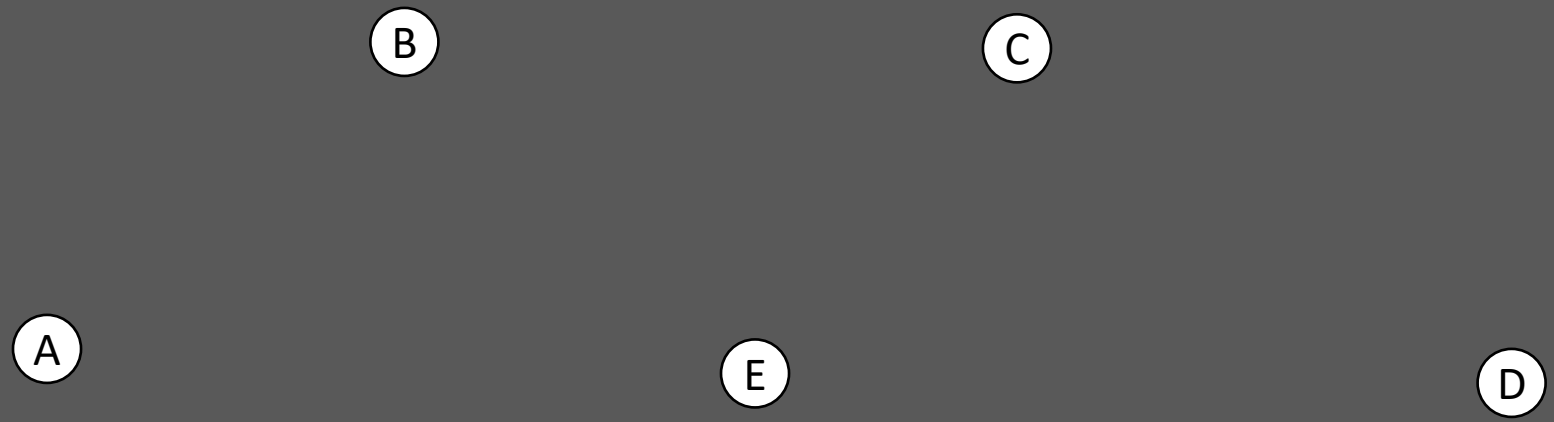
Edge	Weight
A-B	1
C-D	2
A-E	3
B-E	4
B-C	5
E-C	6
E-D	7



KRUSKAL (minimum spanning tree)

Second, make each node a separate tree

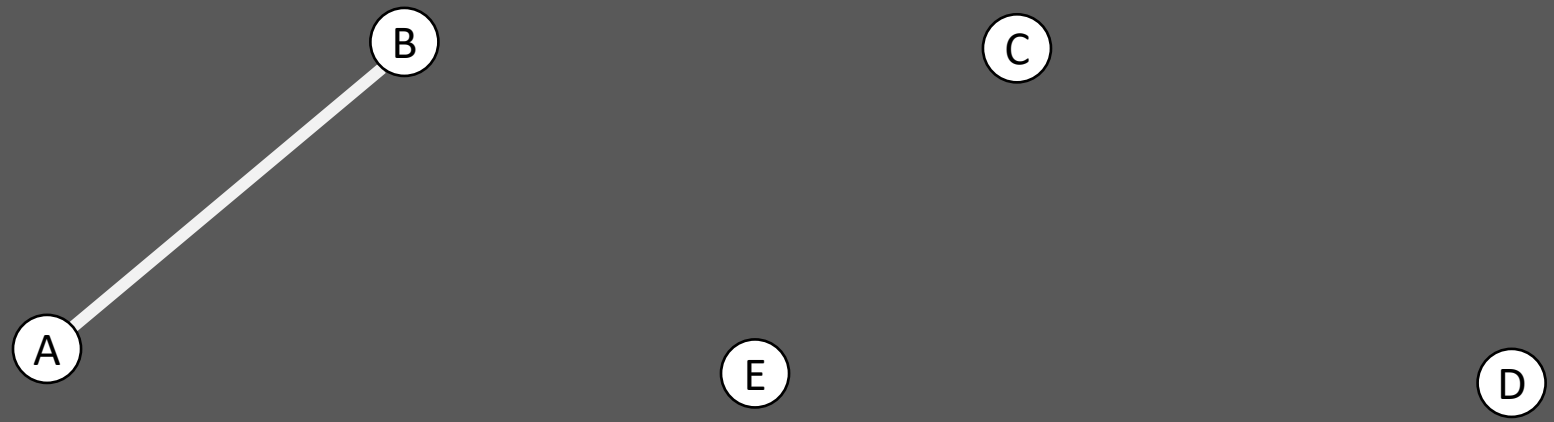
Edge	Weight
A-B	1
C-D	2
A-E	3
B-E	4
B-C	5
E-C	6
E-D	7



KRUSKAL (minimum spanning tree)

Next, form new trees starting from the minimum edge weight

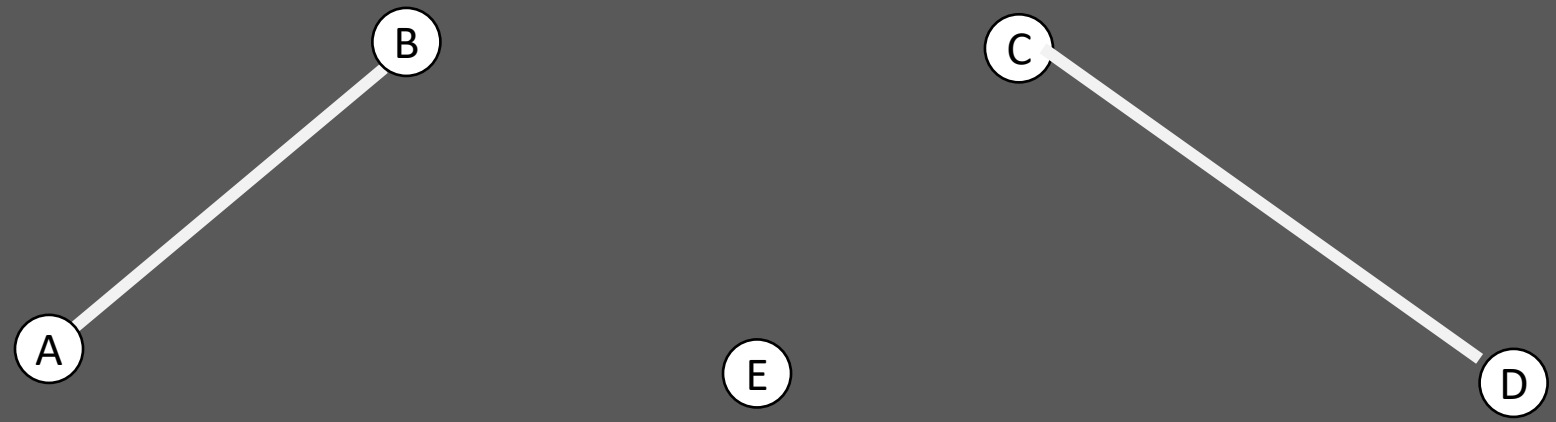
Edge	Weight
A-B	1
C-D	2
A-E	3
B-E	4
B-C	5
E-C	6
E-D	7



KRUSKAL (minimum spanning tree)

Next, form new trees starting from the minimum edge weight

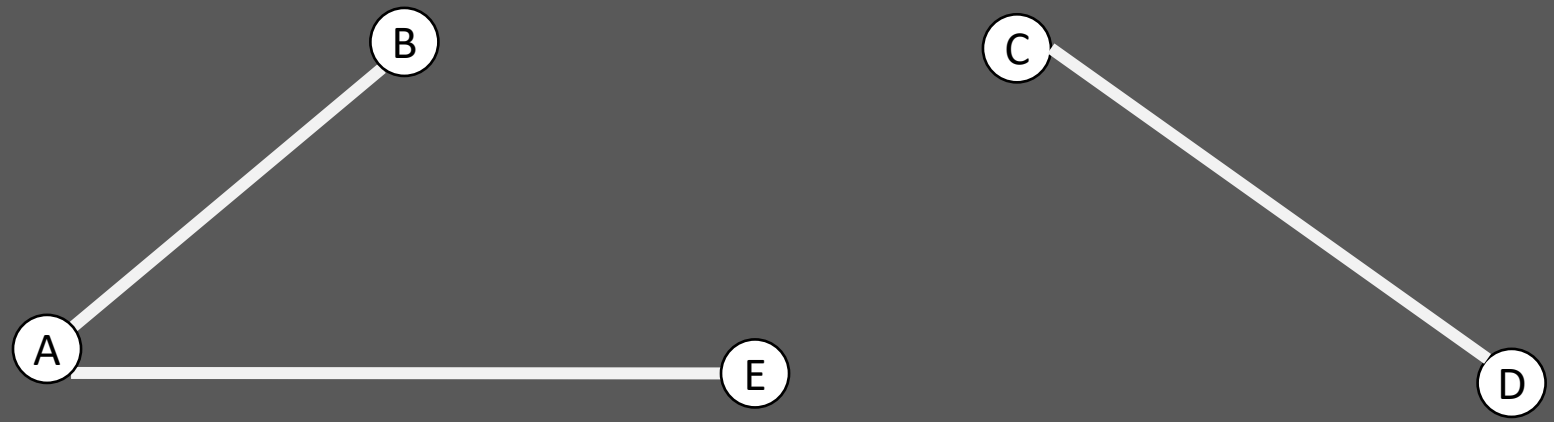
Edge	Weight
A-B	1
C-D	2
A-E	3
B-E	4
B-C	5
E-C	6
E-D	7



KRUSKAL (minimum spanning tree)

Next, form new trees starting from the minimum edge weight

Edge	Weight
A-B	1
C-D	2
A-E	3
B-E	4
B-C	5
E-C	6
E-D	7

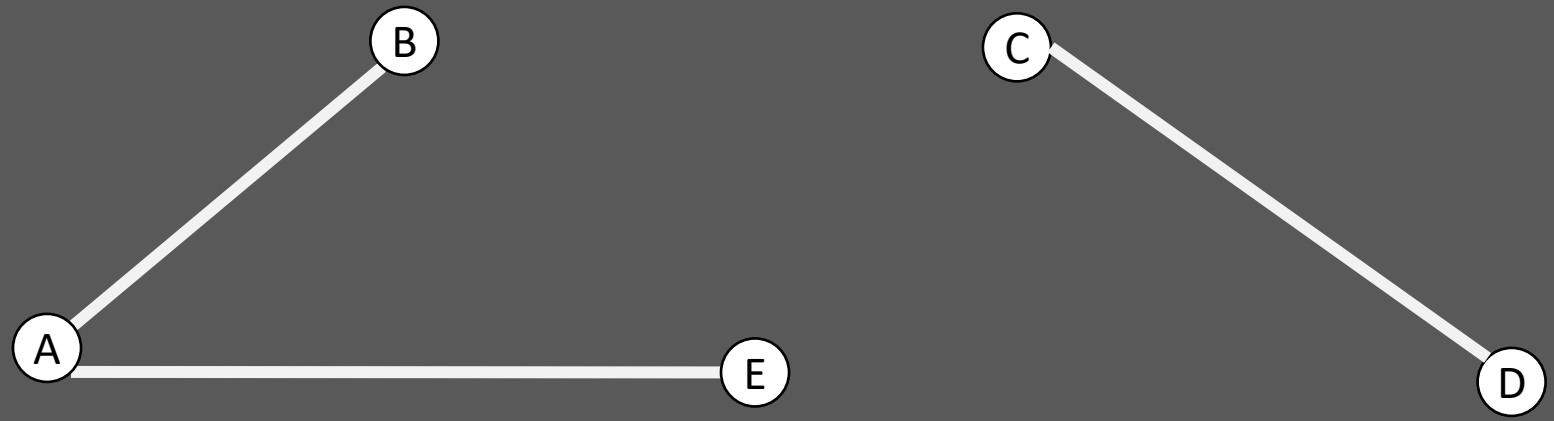


KRUSKAL (minimum spanning tree)

Next, form new trees starting from the minimum edge weight

B-E is the next edge to be added, but it is a tree belonging to A-B-E, so we don't include it

Edge	Weight
A-B	1
C-D	2
A-E	3
B-E	4
B-C	5
E-C	6
E-D	7

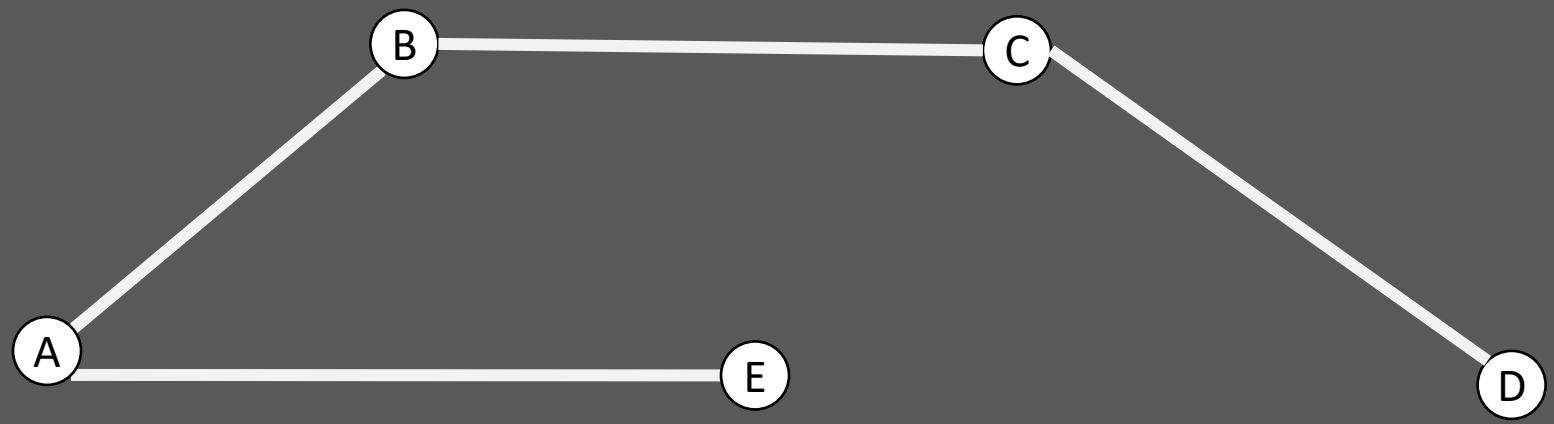


We can also say that it is because B-E forms a cycle A-B-E-A, so we don't include it

KRUSKAL (minimum spanning tree)

Next, form new trees starting from the minimum edge weight

Edge	Weight
A-B	1
C-D	2
A-E	3
B-E	4
B-C	5
E-C	6
E-D	7

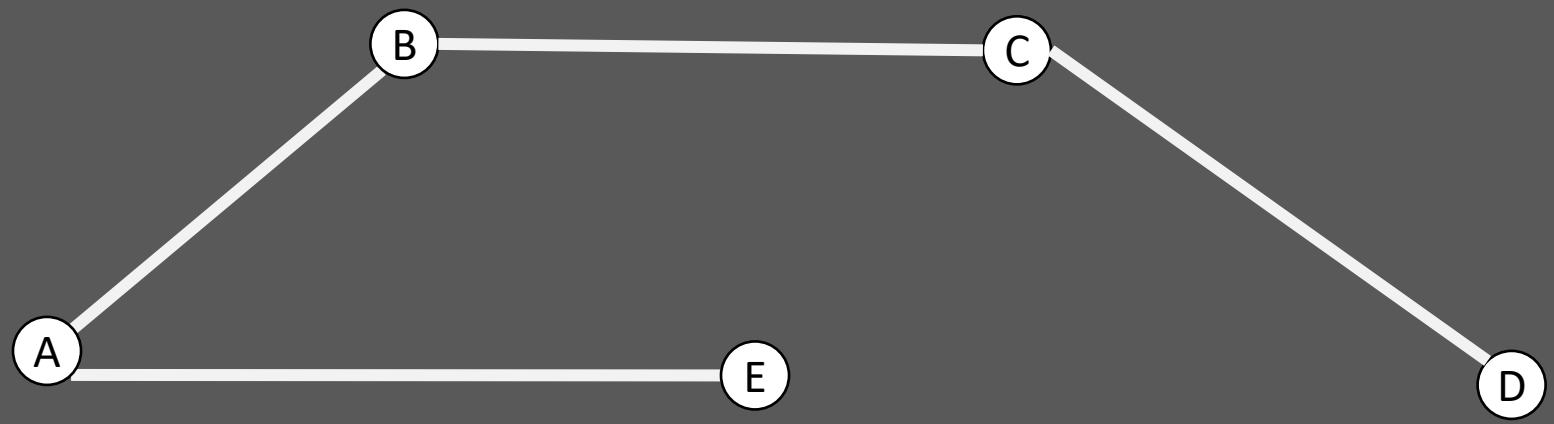


KRUSKAL (minimum spanning tree)

Next, form new trees starting from the minimum edge weight

E-C is the next edge to be added, but it is a tree belonging to A-B-C-E, so we don't include it

Edge	Weight
A-B	1
C-D	2
A-E	3
B-E	4
B-C	5
E-C	6
E-D	7



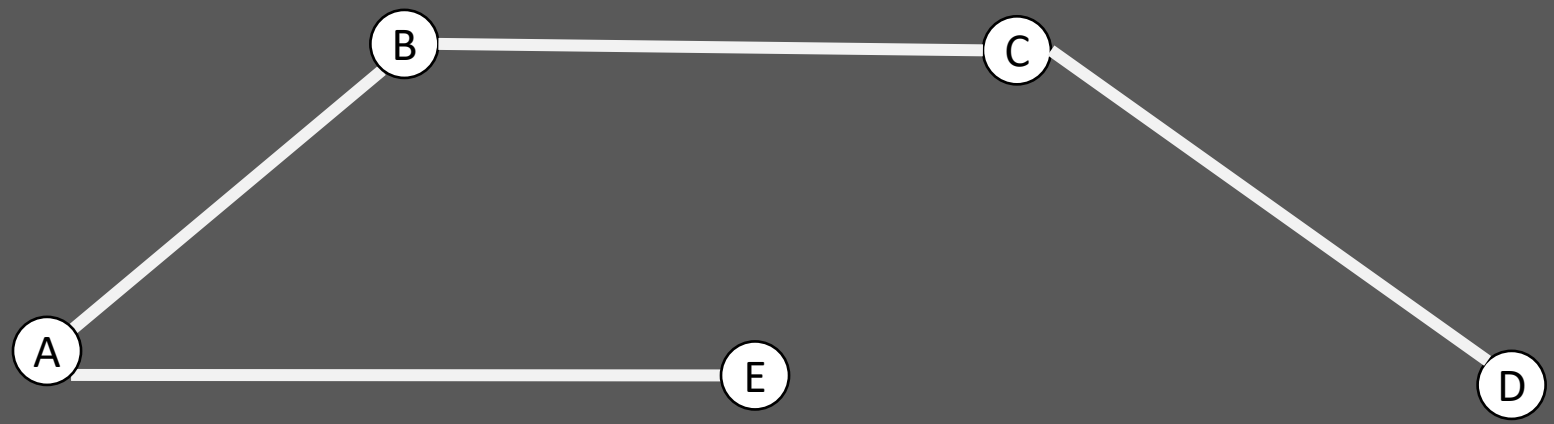
We can also say that it is because E-C forms a cycle A-B-C-E-A, so we don't include it

KRUSKAL (minimum spanning tree)

Next, form new trees starting from the minimum edge weight

E-C is the next edge to be added, but it is a tree belonging to A-B-C-D-E, so we don't include it

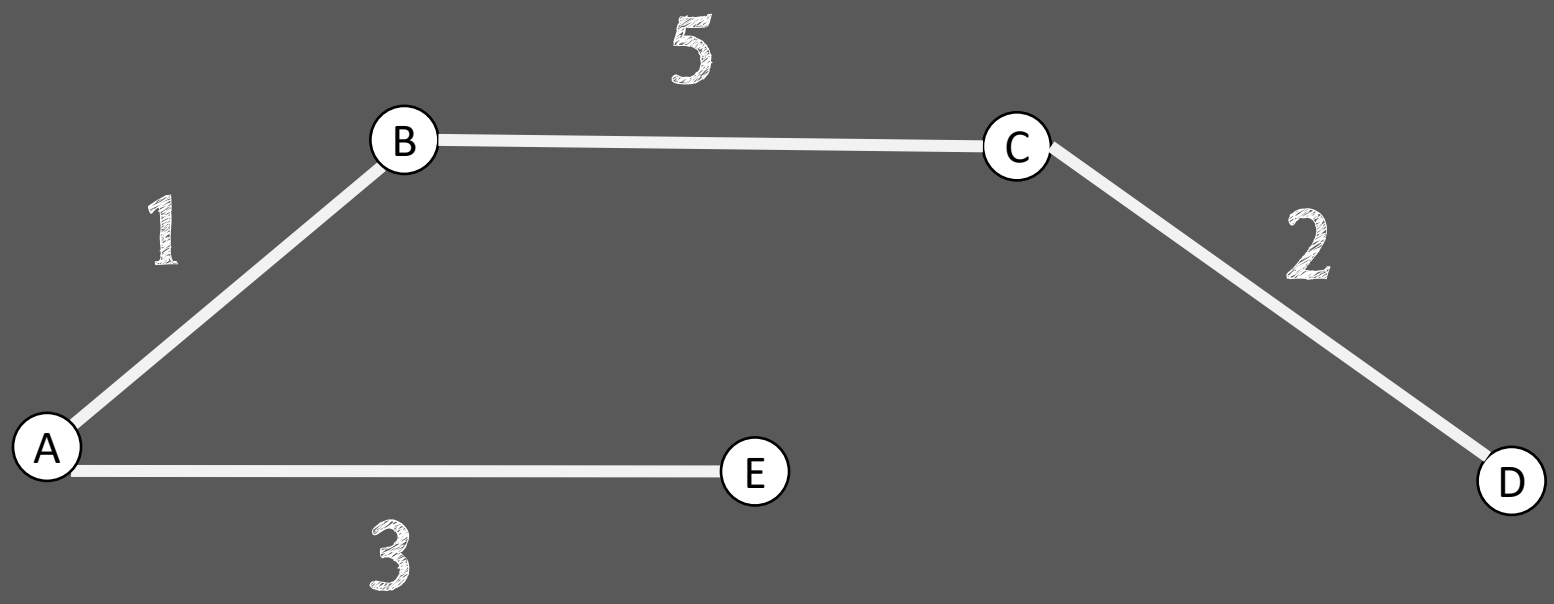
Edge	Weight
A-B	1
C-D	2
A-E	3
B-E	4
B-C	5
E-C	6
E-D	7



We can also say that it is because E-C forms a cycle A-B-C-D-E-A, so we don't include it

KRUSKAL (minimum spanning tree)

Cost of minimum spanning tree is 10



Comparing Prim and Kruskal's algorithm for minimum spanning tree

PRIM'S ALGORITHM	KRUSKAL'S ALGORITHM
It starts to build the Minimum SpanningTree from any vertex in the graph.	It starts to build the Minimum SpanningTree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices andcan be improved up to O(E + log V) using Fibonacci heaps.	Kruskal's algorithm's time complexity isO(E log V), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest (disconnected components) and it can work on disconnected graphs
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.