Greedy Algorithms: Prim’s, Kruskal’s

9.1-9.2

Intro on greedy algorithms

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

It’s applicable to optimization problems

It doesn't worry whether the current best result will bring the overall optimal result.

Intro on greedy algorithms

On each step—and this is the central point of this technique—the choice made must be:

- feasible, i.e., it has to satisfy the problem's constraints
- locally optimal, i.e., it has to be the best local choice among all feasible choices available on that step
- irrevocable, i.e., once made, it cannot be changed on subsequent steps of the algorithm

Intro on greedy algorithms

These requirements explain the technique’s name: on each step, it suggests a “greedy” grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem.

Intro on greedy algorithms

These requirements explain the technique’s name: on each step, it suggests a “greedy” grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem.

As we shall see, there are problems for which a sequence of locally optimal choices does yield an optimal solution for every instance of the problem in question. However, there are others for which this is not the case; for such problems, a greedy algorithm can still be of value if we are interested in or have to be satisfied with an approximate solution.

Intro on greedy algorithms

The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

Intro on greedy algorithms- Prim’s + Kruskal’s

In the first two sections of the chapter, we discuss two classic algorithms for the minimum spanning tree problem: Prim’s algorithm and Kruskal’s algorithm.

What is remarkable about these algorithms is the fact that they solve the same problem by applying the greedy approach in two different ways, and both of them always yield an optimal solution.

Intro on greedy algorithms- Dijkstra’s + Huffman trees

In Section 9.3, we introduce another classic algorithm— Dijkstra’s algorithm for the shortest-path problem in a weighted graph. Section 9.4 is devoted to Huffman trees and their principal application, Huffman codes—an important data compression method that can be interpreted as an application of the greedy technique.

Greedy algorithms are mostly simple

Given an optimization problem, it is usually easy to figure out how to proceed in a greedy manner, possibly after considering a few small instances of the problem.

Greedy algorithms are mostly simple

Given an optimization problem, it is usually easy to figure out how to proceed in a greedy manner, possibly after considering a few small instances of the problem.

It’s more difficult to prove that a greedy algorithm yields an optimal solution (when it does).

Greedy algorithms are mostly simple

Given an optimization problem, it is usually easy to figure out how to proceed in a greedy manner, possibly after considering a few small instances of the problem.

It’s more difficult to prove that a greedy algorithm yields an optimal solution (when it does).

- Use mathematical induction (proof in 9.1)
- show that on each step it does at least as well as any other algorithm could in advancing toward the problem’s goal.

When do we use greedy algorithms

We can determine if the algorithm can be used with any problem if the problem has the following properties:

1. Greedy Choice Property

If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

When do we use greedy algorithms

We can determine if the algorithm can be used with any problem if the problem has the following properties:

2. Optimal Substructure

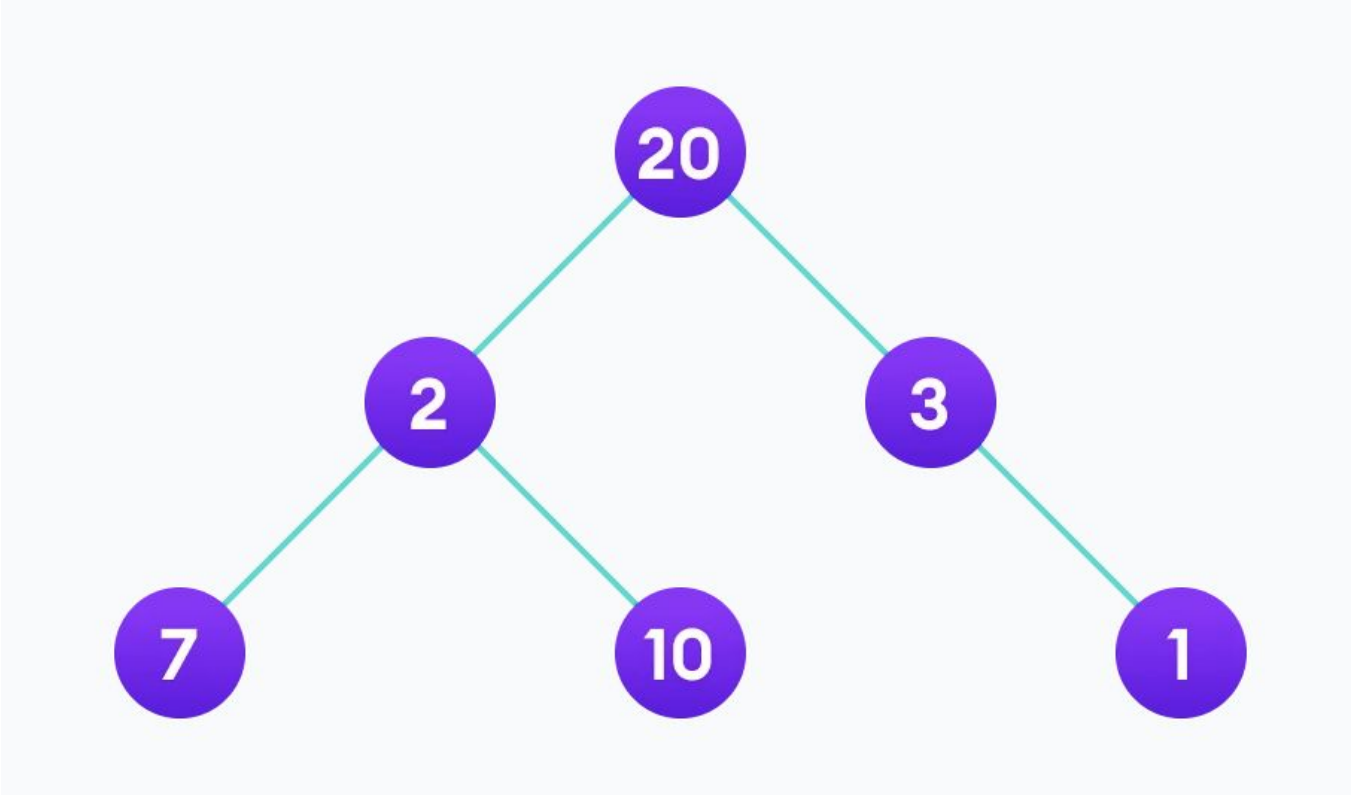
If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

Drawbacks of greedy algorithms

As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm

For example, suppose we want to find the longest path in the graph below from root to leaf. Let's use the greedy algorithm here.

Let’s apply greedy approach to find the longest route

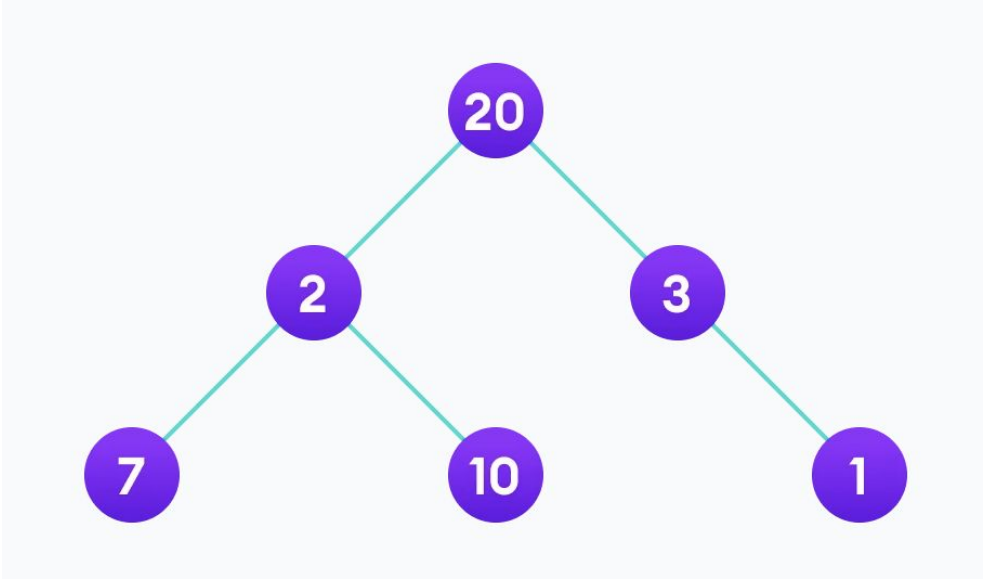


Greedy approach

Let's start with the root node 20.

The weight of the right child is 3

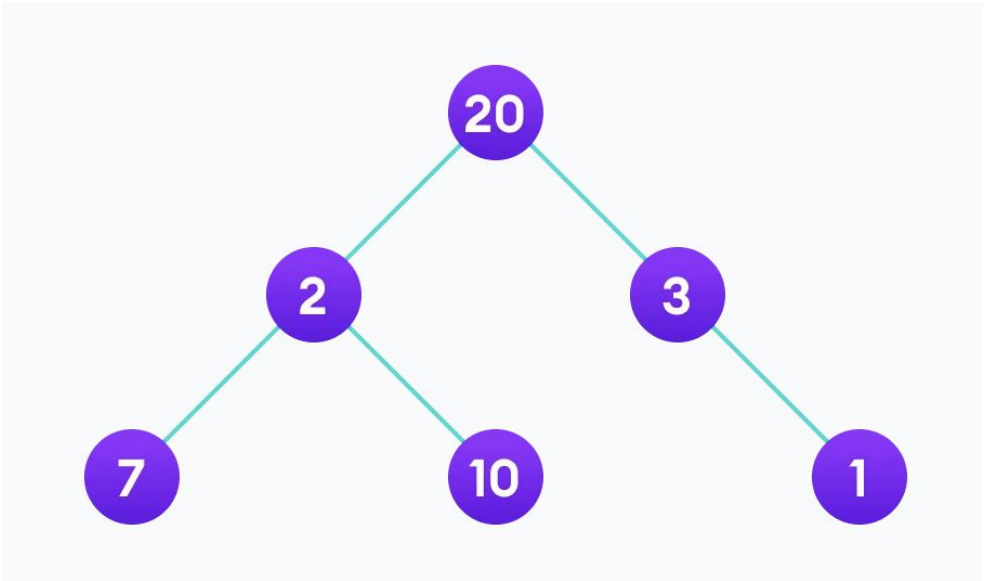
and the weight of the left child is 2.



Greedy approach

Our problem is to find the largest path.

And, the optimal solution at the moment is 3. So, the greedy algorithm will choose 3.

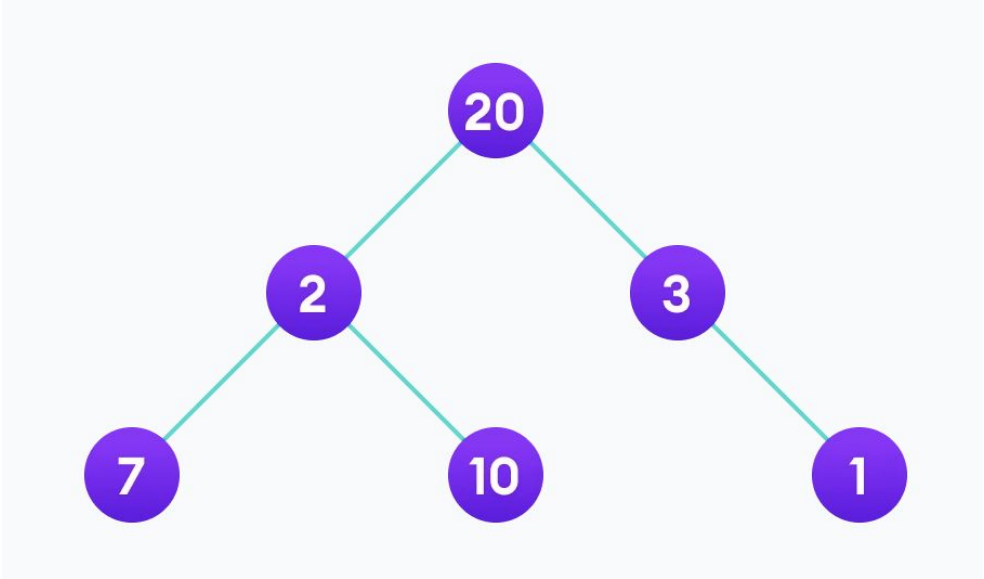


Greedy approach

The weight of the only child of 3 is 1.

This gives us our final result

20 + 3 + 1 = 24.

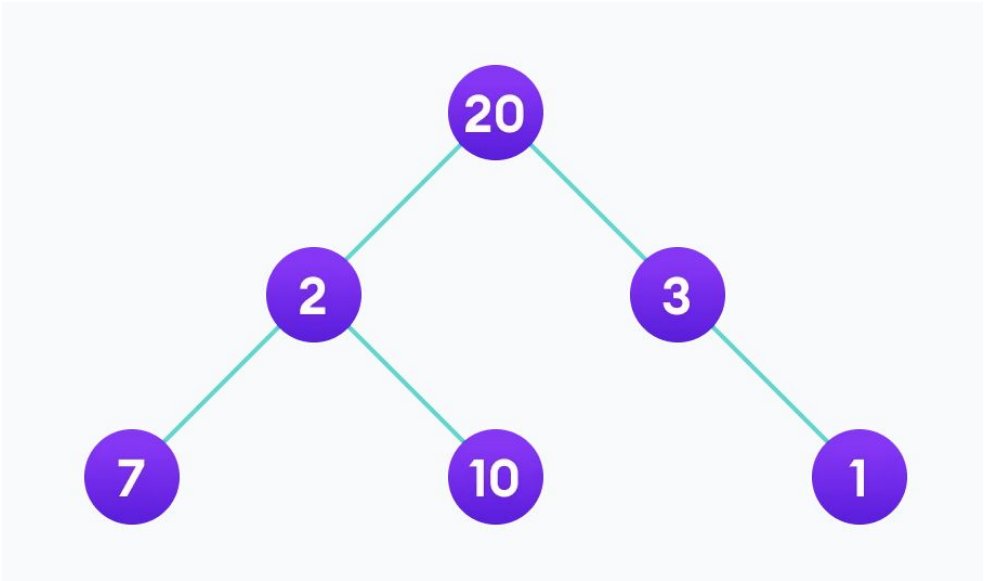


Greedy approach

However, it is not the optimal solution.

There is another path that carries

more weight (20 + 2 + 10 = 32)



Greedy algorithm

1. To begin with, the solution set (containing answers) is empty.
2. At each step, an item is added to the solution set until a solution is reached.
3. If the solution set is feasible, the current item is kept.
4. Else, the item is rejected and never considered again.

Let's now use this algorithm to solve a problem

Greedy algorithm- example

Problem: You have to make a change of an amount using the smallest possible number of coins.

Amount: $18

Available coins are

- $5 coin
- $2 coin
- $1 coin

There is no limit to the number of each coin you can use.

Greedy algorithm- example

1. Create an empty solution-set = { }. Available coins are

Greedy algorithm- example

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}

Greedy algorithm- example

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}
2. We are supposed to find the sum = 18. Let's start with sum = 0.

Greedy algorithm- example

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}
2. We are supposed to find the sum = 18. Let's start with sum = 0.
3. Always select the coin with the largest value (i.e. 5) until the sum > 18. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called greedy choice property.)

Greedy algorithm- example

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}
2. We are supposed to find the sum = 18. Let's start with sum = 0.
3. Always select the coin with the largest value (i.e. 5) until the sum > 18. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called greedy choice property.)
4. In the first iteration, solution-set = {5} and sum = 5

Greedy algorithm- example

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}
2. We are supposed to find the sum = 18. Let's start with sum = 0.
3. Always select the coin with the largest value (i.e. 5) until the sum > 18. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called greedy choice property.)
4. In the first iteration, solution-set = {5} and sum = 5
5. In the second iteration, solution-set = {5, 5} and sum = 10

Greedy algorithm- example

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}
2. We are supposed to find the sum = 18. Let's start with sum = 0.
3. Always select the coin with the largest value (i.e. 5) until the sum > 18. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called greedy choice property.)
4. In the first iteration, solution-set = {5} and sum = 5
5. In the second iteration, solution-set = {5, 5} and sum = 10
6. In the third iteration, solution-set = {5, 5, 5} and sum = 15.

Greedy algorithm- example

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}
2. We are supposed to find the sum = 18. Let's start with sum = 0.
3. Always select the coin with the largest value (i.e. 5) until the sum > 18. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called greedy choice property.)
4. In the first iteration, solution-set = {5} and sum = 5
5. In the second iteration, solution-set = {5, 5} and sum = 10
6. In the third iteration, solution-set = {5, 5, 5} and sum = 15.
7. In the fourth iteration, solution-set = {5, 5, 5, 2} and sum = 17. (We cannot select 5 here because if we do so, sum = 20 which is greater than 18. So, we select the 2nd largest item which is 2.)

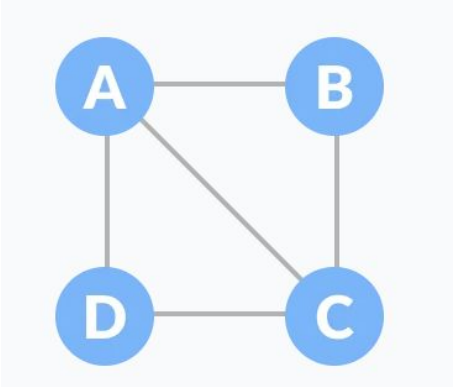
Greedy algorithm- example

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}
2. We are supposed to find the sum = 18. Let's start with sum = 0.
3. Always select the coin with the largest value (i.e. 5) until the sum > 18. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called greedy choice property.)
4. In the first iteration, solution-set = {5} and sum = 5
5. In the second iteration, solution-set = {5, 5} and sum = 10
6. In the third iteration, solution-set = {5, 5, 5} and sum = 15.
7. In the fourth iteration, solution-set = {5, 5, 5, 2} and sum = 17. (We cannot select 5 here because if we do so, sum = 20 which is greater than 18. So, we select the 2nd largest item which is 2.)
8. Similarly, in the fifth iteration, select 1. Now sum = 18 and solution-set = {5, 5, 5, 2, 1}.

Spanning trees

Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.

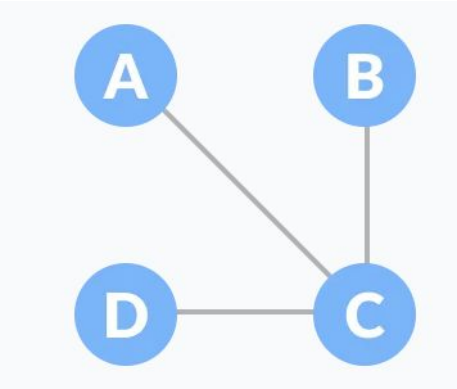
An undirected graph is a graph in which the edges do not point in any direction (ie. the edges are bidirectional).



Spanning trees

Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.

A connected graph is a graph in which there is always a path from a vertex to any other vertex.



Spanning trees

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

Spanning trees

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n^{(n-2)}$

Spanning trees

A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

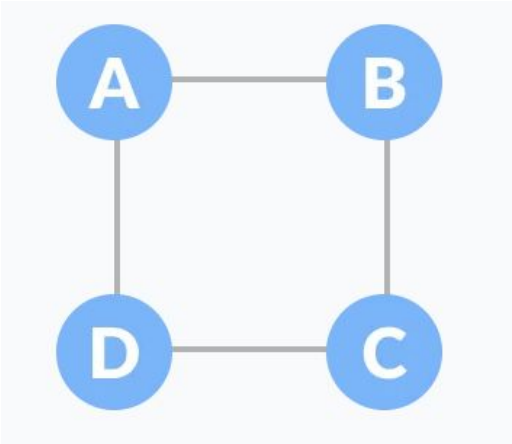
The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n^{(n-2)}$

If we have n = 4, the maximum number of possible spanning trees is equal to $4^{(4-2)}$ = 16. Thus, 16 spanning trees can be formed from a complete graph with 4 vertices.

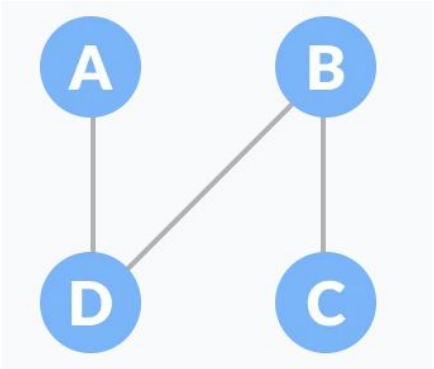
Spanning tree- example

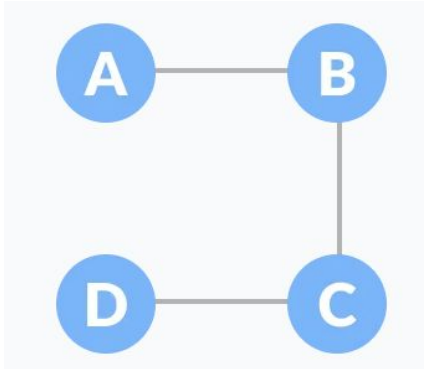
Let's understand the spanning tree with examples below:

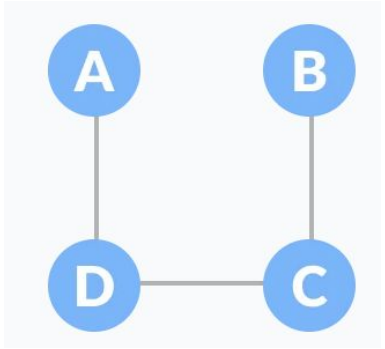
Let the original graph be:

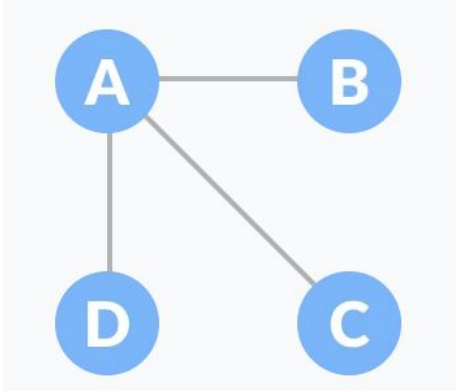


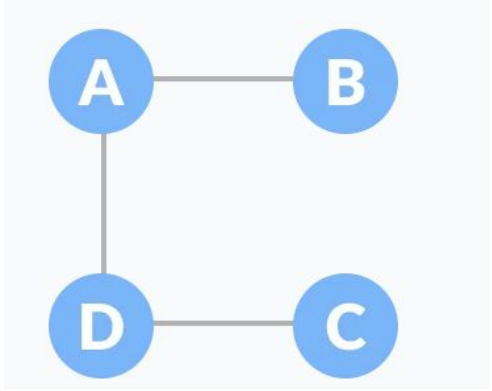
Some of the possible spanning trees that can be created from the above graph are:

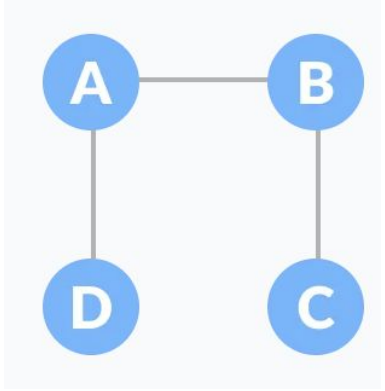












Minimum spanning tree

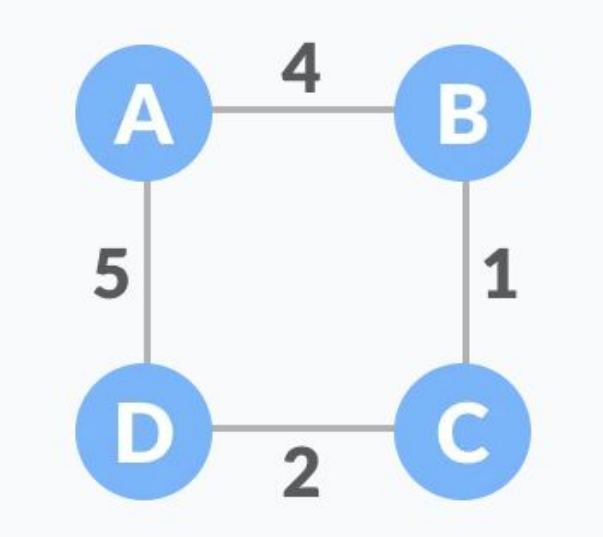
A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

Minimum spanning tree- example

A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

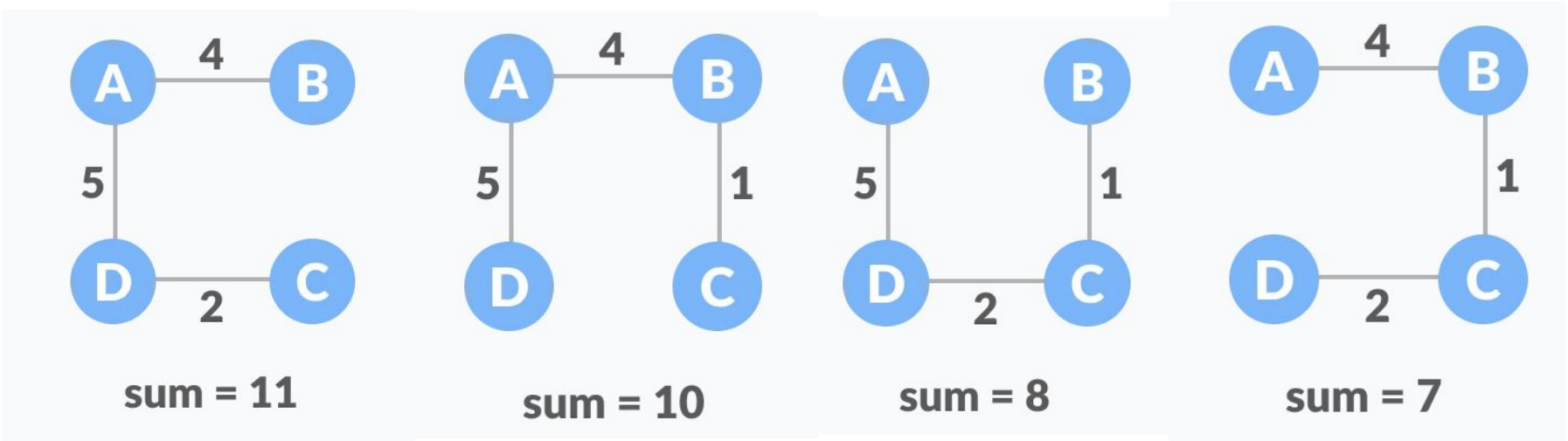
Let's understand the above definition with the help of an example.

The initial graph is:



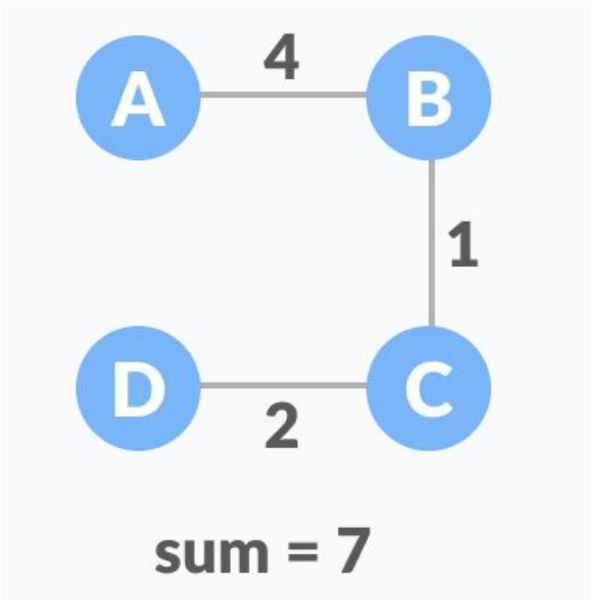
Minimum spanning tree- example

The possible spanning trees from the above graph are:



Minimum spanning tree- example

The minimum spanning tree from the above spanning trees is:

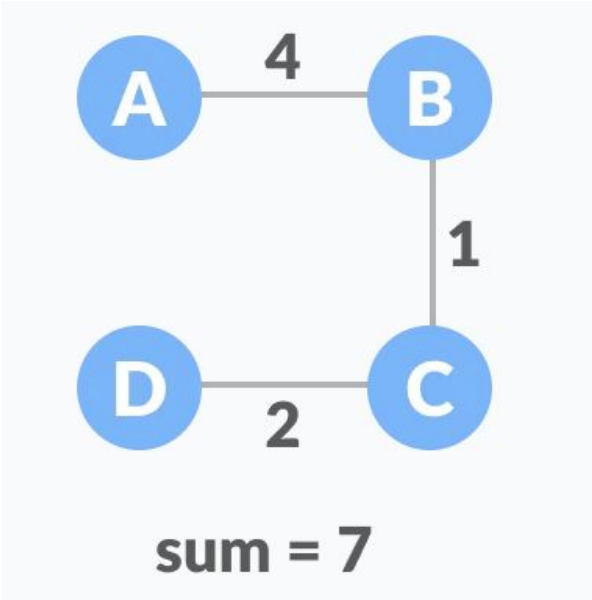


Minimum spanning tree- example

The minimum spanning tree from the above spanning trees is:

The minimum spanning tree from a graph is found using the following algorithms:

1. Prim's Algorithm
2. Kruskal's Algorithm



Prim’s algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

How Prim’s algorithm works

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

How Prim’s algorithm works

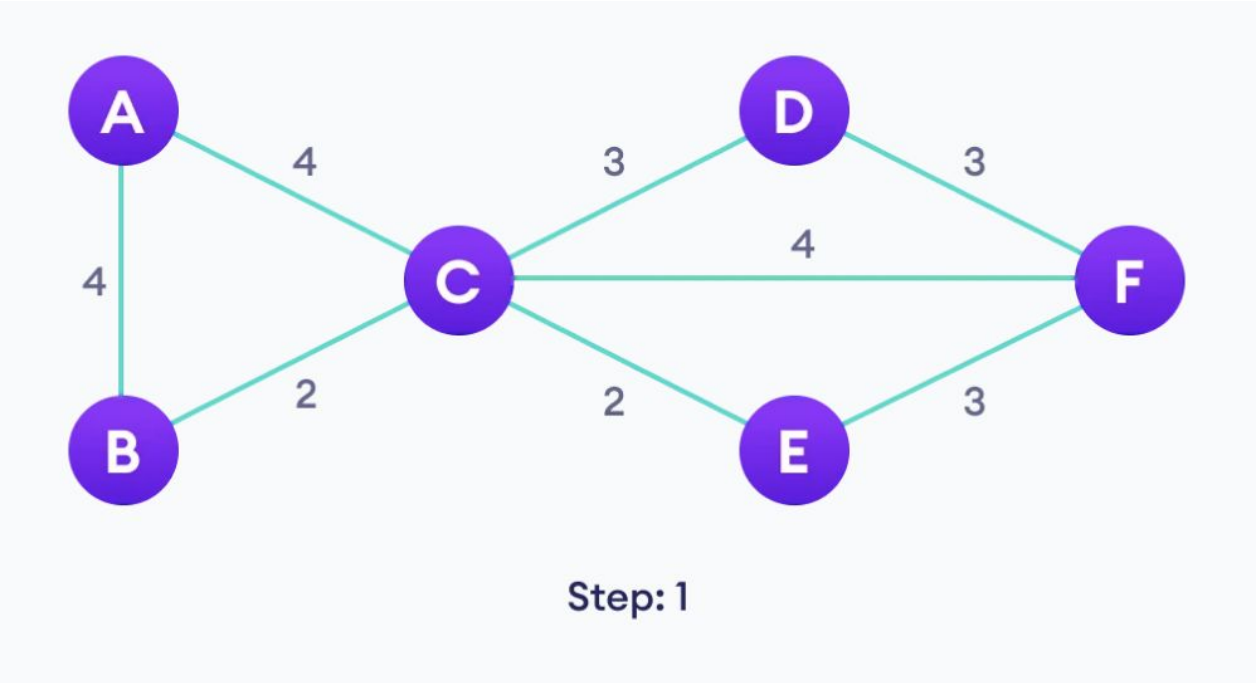
It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the minimum spanning tree (MST), the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and 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.

How Prim’s algorithm works

The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed in the previous slide) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Prim’s algorithm- example

Start with a weighted graph

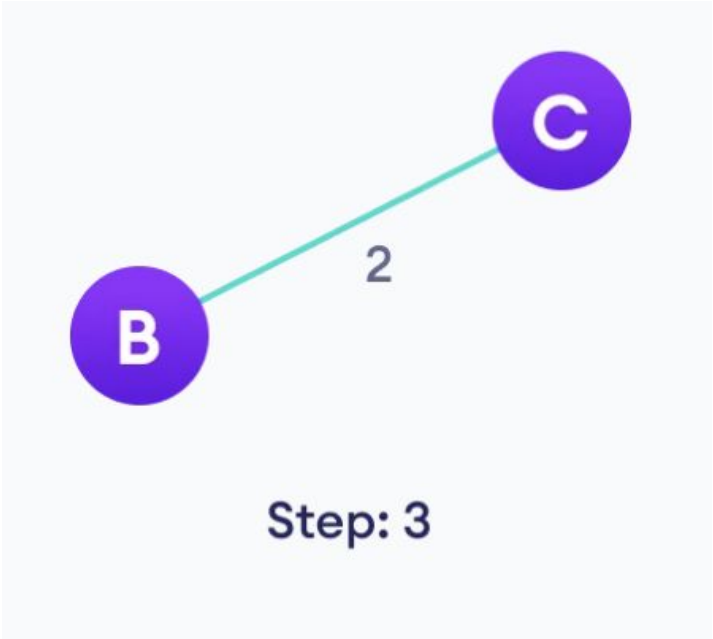


Prim’s algorithm- example

Choose a vertex

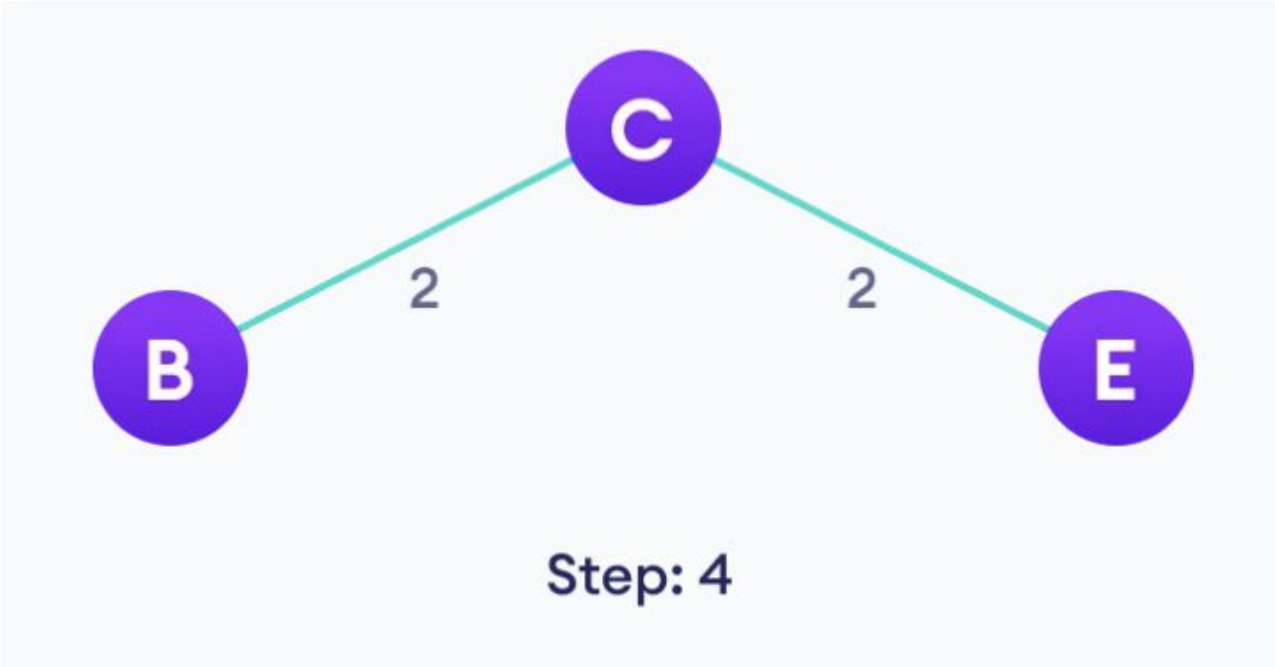
Prim’s algorithm- example

Choose the shortest edge from this vertex and add it



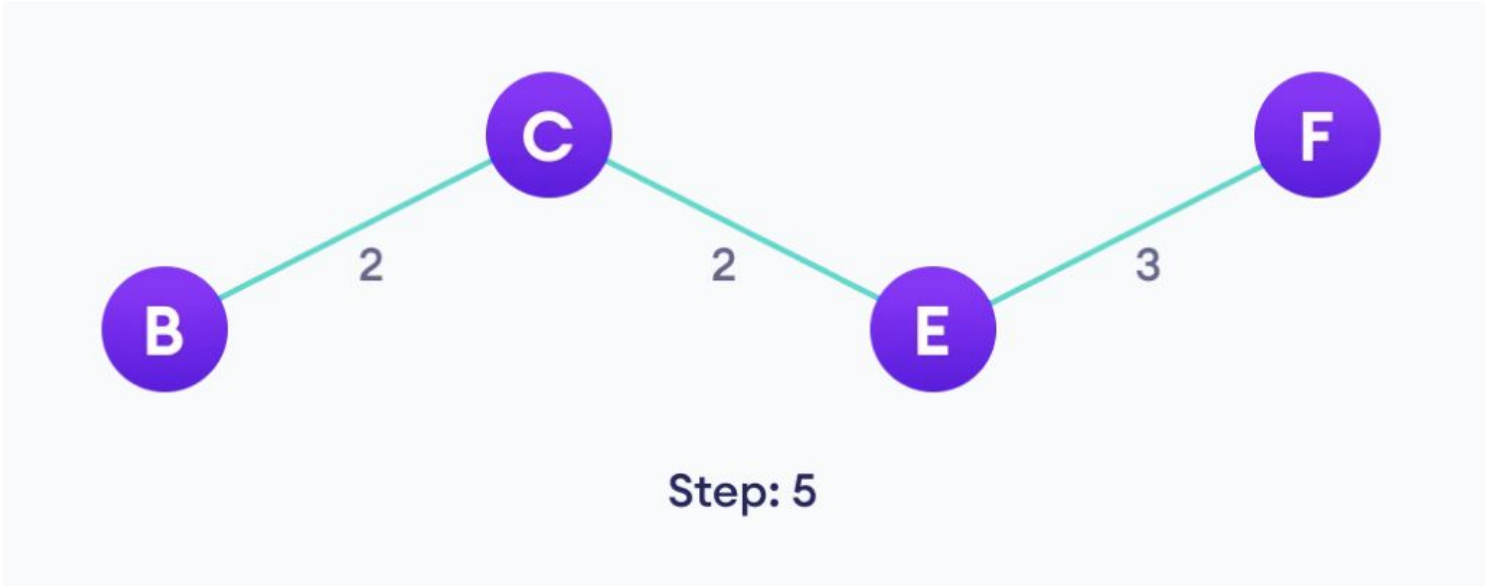
Prim’s algorithm- example

Choose the nearest vertex not yet in the solution



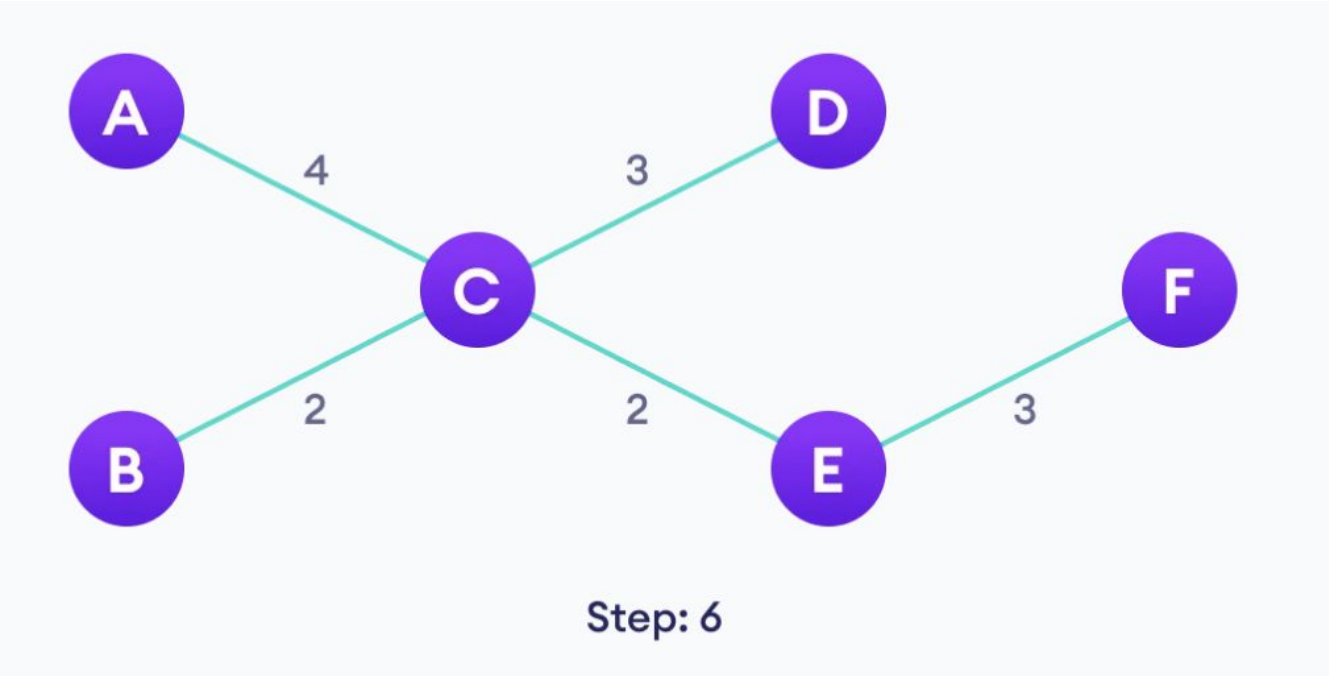
Prim’s algorithm- example

Choose the nearest edge not yet in the solution If there are multiple, choose one at random.



Prim’s algorithm- example

Repeat until you have a spanning tree



Prim’s algorithm- pseudocode

The pseudocode for prim's algorithm shows how we create two sets of vertices U and V-U. U contains the list of vertices that have been visited and V-U the list of vertices that haven't. One by one, we move vertices from set V-U to set U by connecting the least weight edge.

Prim’s algorithm- pseudocode

T = ∅; (null set/empty set)

U = { 1 };

while (U ≠ V)

    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;

    T = T ∪ {(u, v)}

      U = U ∪ {v}

Prim’s algorithm- pseudocode

1) Create a set mstSet that keeps track of vertices already included in minimum spanning tree (MST).

2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

3) While mstSet doesn’t include all vertices

….a) Pick a vertex u which is not there in mstSet and has minimum key value.

….b) Include u to mstSet.

….c) Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

(A cut is a partition of the vertices of a graph into two disjoint subsets)

Prim’s algorithm- video

https://youtu.be/eB61LXLZVqs

Prim’s algorithm-time complexity

Time Complexity of the code I will send out is O(V^2).

If the input graph is represented using adjacency list, then the time complexity of Prim’s algorithm can be reduced to O(E log V) with the help of binary heap.

Prim’s algorithm- applications

- Laying cables of electrical wiring
- In network designed
- To make protocols in network cycles
- Network of roads and railroad tracks connecting cities
- Irrigation channels

Kruskal’s algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

How Kruskal’s algorithm works

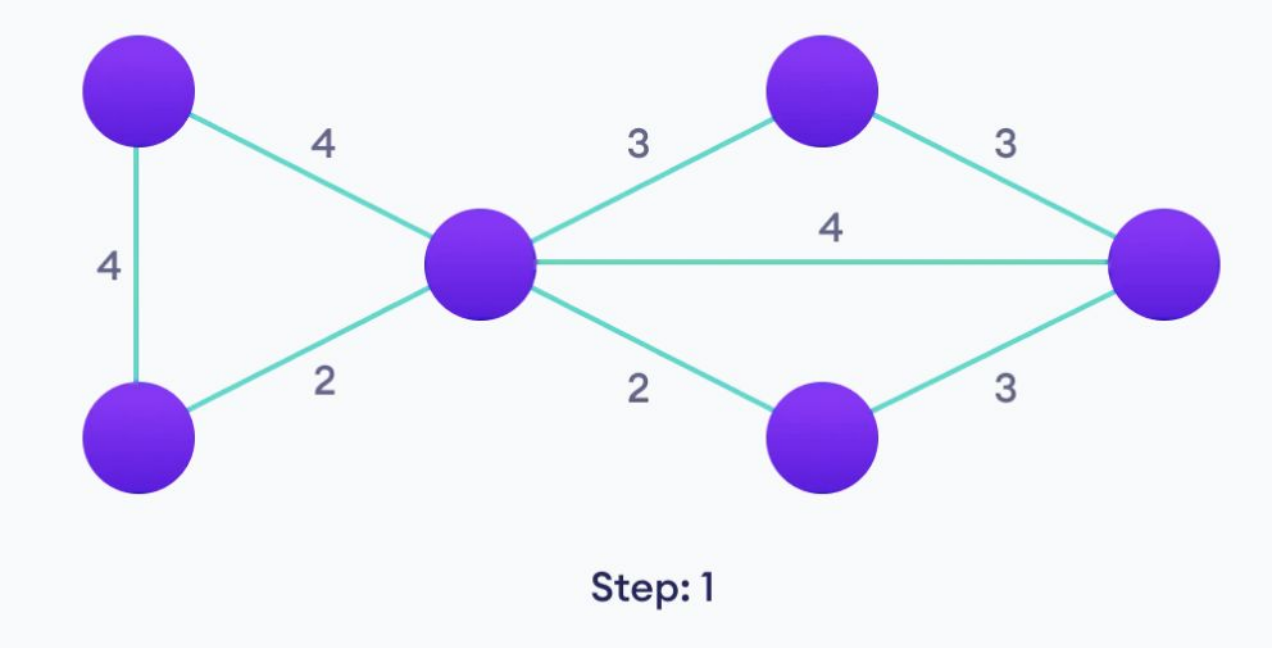
We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

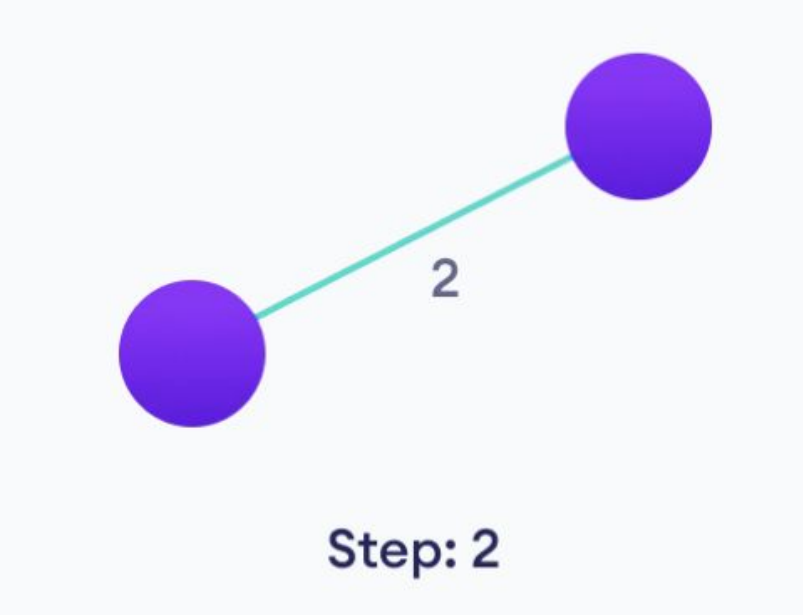
Kruskal’s algorithm- example

Start with a weighted graph



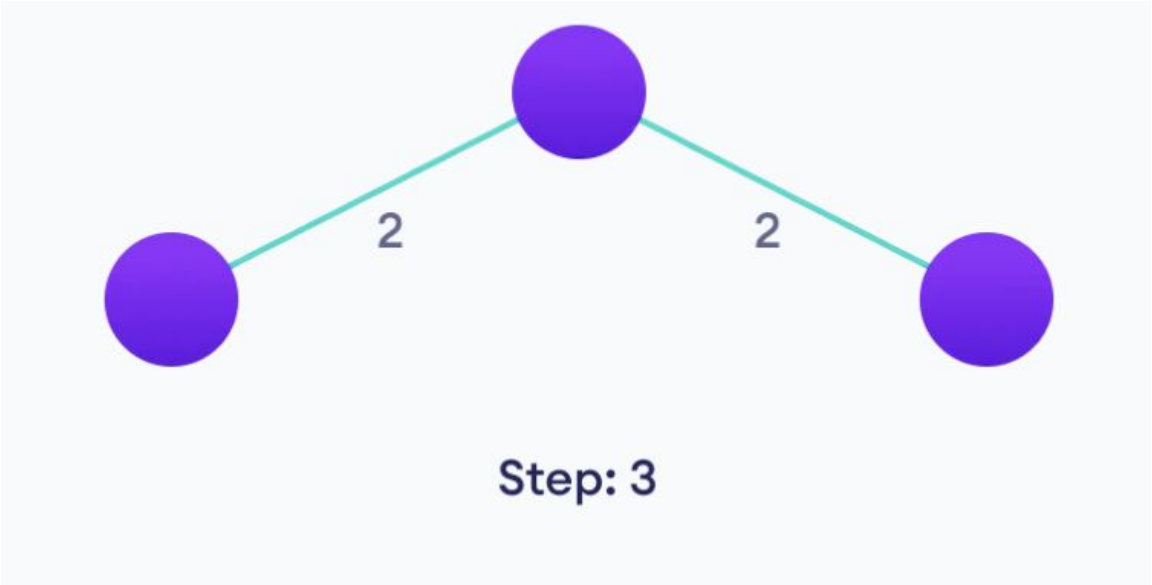
Kruskal’s algorithm- example

Choose the edge with the least weight, if there are more than 1, choose any



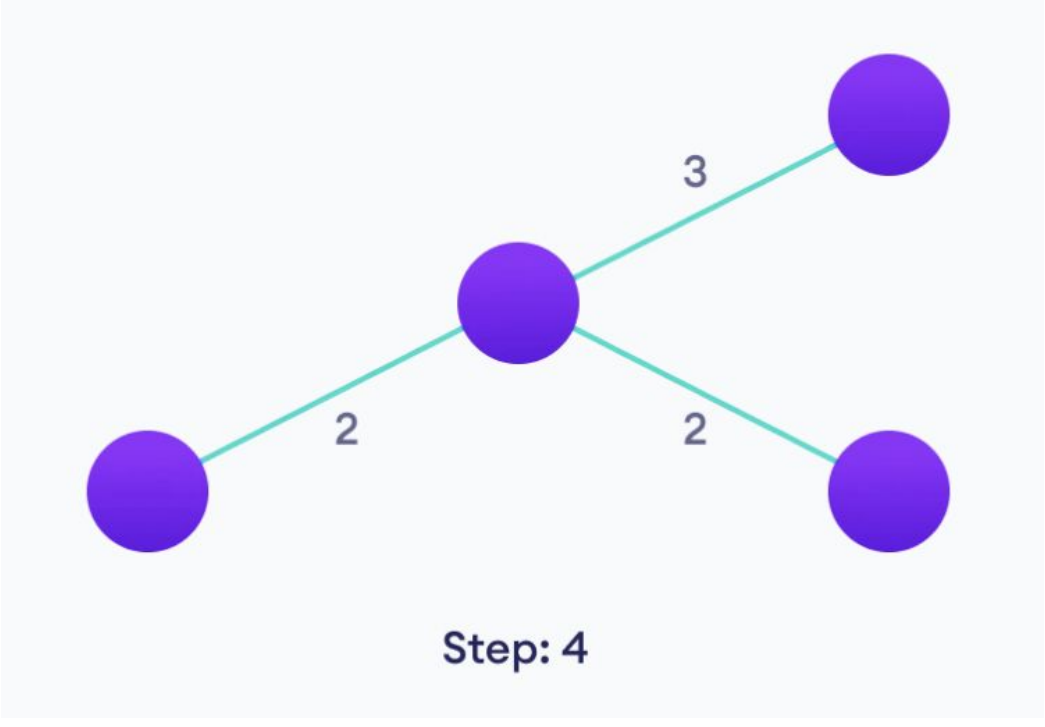
Kruskal’s algorithm- example

Choose the next shortest edge and add it



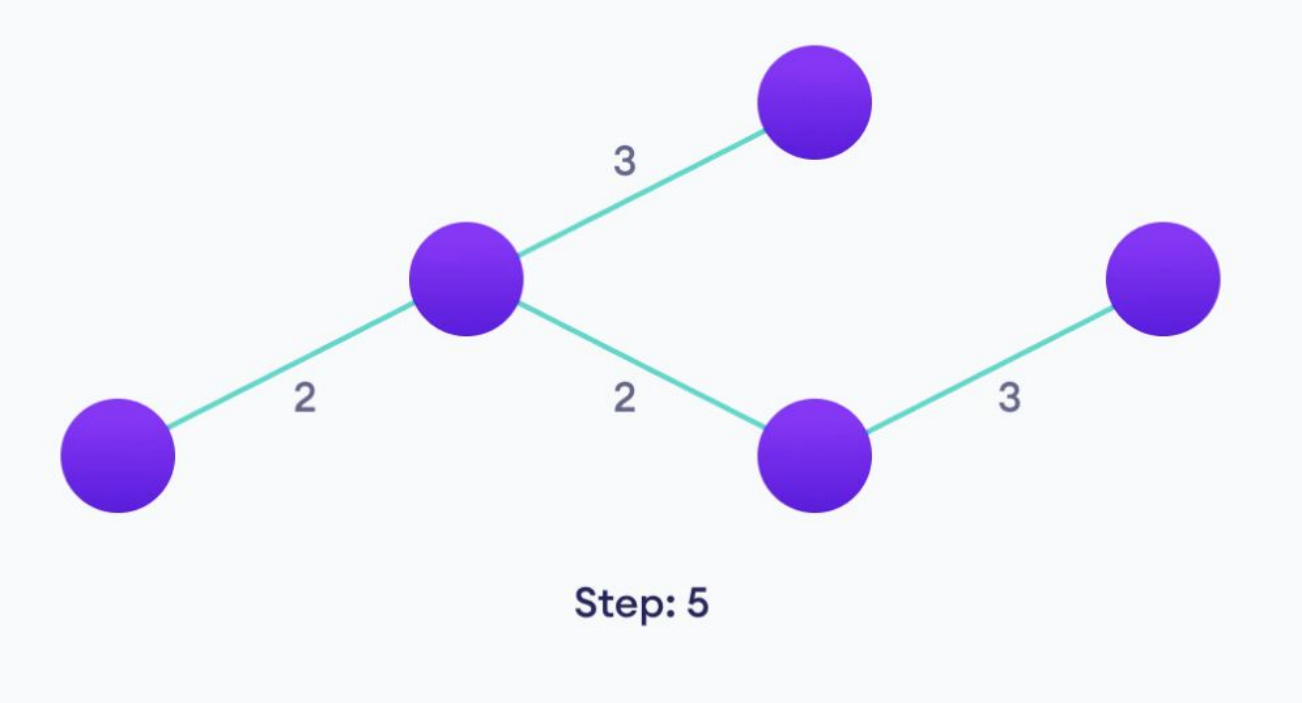
Kruskal’s algorithm- example

Choose the next shortest edge and add it



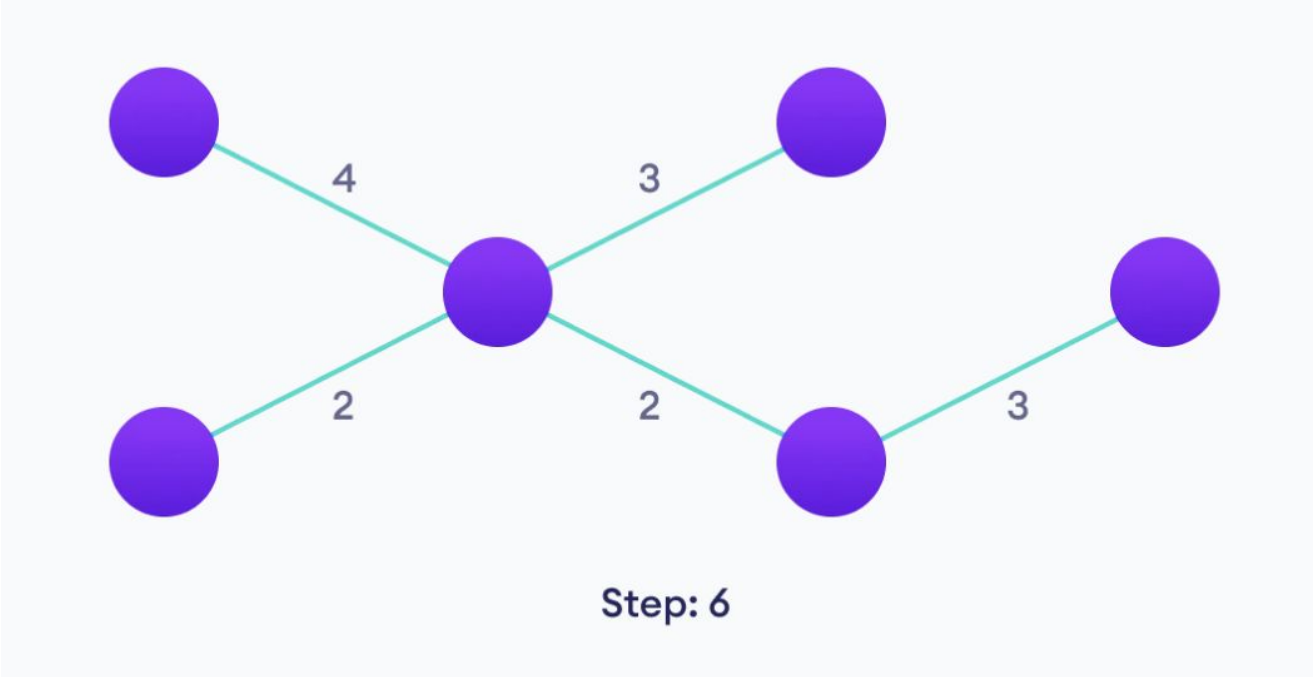
Kruskal’s algorithm- example

Choose the next shortest edge that doesn't create a cycle and add it



Kruskal’s algorithm- example

Repeat until you have a spanning tree



Kruskal’s algorithm- pseudocode

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
    MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v):
    A = A ∪ {(u, v)}
    UNION(u, v)
  return A
```

Kruskal’s algorithm- pseudocode

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. If cycle is not formed, include this edge. Else, discard it.

3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Step 2 uses the union-find algorithm to detect cycles

Union find algorithm

The Union-Find algorithm divides the vertices into clusters and allows us to check if two vertices belong to the same cluster or not and hence decide whether adding an edge creates a cycle.

https://courses.cs.duke.edu/cps100e/fall09/notes/UnionFind.pdf

Kruskal’s algorithm- video

https://youtu.be/3rrNH_AizMA

Kruskal’s algorithm-time complexity

Time complexity is O(ElogE) or O(ElogV)

Sorting of edges takes O(ELogE) time.

After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(LogV) time.

So overall complexity is O(ELogE + ELogV) time. The value of E can be at most O(V2), so O(LogV) is O(LogE) the same. Therefore, the overall time complexity is O(ElogE) or O(ElogV)

Kruskal’s algorithm- applications

- Laying cables of electrical wiring
- In network designed
- To make protocols in network cycles
- Network of roads and railroad tracks connecting cities
- Irrigation channels

Prim’s vs Kruskal’s

Prim’s	Kruskal’s
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree 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(V2), V being the number of vertices and can be improved up to O(E log V) using Fibonacci heaps.	Kruskal’s algorithm has a time complexity is O(E log V), V being the number of vertices.
Prim’s algorithm runs faster in dense graphs.	Kruskal’s algorithm runs faster in sparse graphs.
Prim’s algorithm uses List Data Structure.	Kruskal’s algorithm uses Heap Data Structure.

Summary

- The greedy technique suggests constructing a solution to an optimization problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step, the choice made must be feasible, locally optimal, and irrevocable.
- Prim’s algorithm is a greedy algorithm for constructing a minimum spanning tree of a weighted connected graph. It works by attaching to a previously constructed subtree a vertex closest to the vertices already in the tree.
- Kruskal’s algorithm is another greedy algorithm for the minimum spanning tree problem. It constructs a minimum spanning tree by selecting edges in nondecreasing order of their weights provided that the inclusion does not create a cycle. Checking the latter condition efficiently requires an application of one of the so-called union-find algorithms.

Homework problems!

Please complete exercise 9.1 #9, 11

Please complete exercise 9.2 #1, 4

** If you believe that the algorithms work correctly on graphs with negative weights, prove this assertion; it you believe this is not to be the case, give a counterexample for each algorithm.