What is Greedy Algorithm?

A greedy algorithm is a simple, intuitive algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In other words, a greedy algorithm makes the most advantageous move at each step, without worrying about the consequences of future steps.

One example of a problem that can be solved using a greedy algorithm is the problem of computing a maximal independent set in a matroid. A maximal independent set in a matroid is a set of elements that is not contained in any other independent set and is as large as possible.

To solve this problem using a greedy algorithm, we can start by selecting an element with the highest rank and adding it to the independent set. We can then repeat this process by selecting the next highest ranked element that is not adjacent to any of the elements already in the independent set, and so on. This process is repeated until no more elements can be added to the independent set.

This greedy algorithm has a time complexity of O(n^2), where n is the number of elements in the matroid.

One application of this algorithm is to activity scheduling, where it can be used to select a set of non-conflicting activities that can be performed simultaneously. Another application is in the construction of a minimum spanning tree, where it can be used to select a set of edges that form a tree with the minimum total weight.

What is a Matroid?

INTRODUCTION

One of the primary goals of pure mathematics is to identify common patterns that occur in disparate circumstances, and to create unifying abstractions which identify commonalities and provide a useful framework for further theorems. For example, the pattern of an associative operation with inverses and an identity occurs frequently, and gives rise to the notion of an abstract group. On top of the basic axioms of a group, a vast theoretical framework can be built up, investigating the classification of groups, their internal structure, and the relationships and operations on groups.

Matroid is a mathematical concept in combinatorics and is defined on abstract space.

DEFINITION

A finite Matroid is a pair (S, I) where S is a ground set (all elements), I is a family of subsets (called Independent Sets) of S, such that it follows three properties:

a. Non – emptiness: The empty set is independent. $(\emptyset \in I)$ (Thus, I is not itself empty.)

b. Heredity: Subsets of independent sets are independent such that if $B \in I$ and $A \subseteq B$, then $A \in I$
c. Exchange: For $A, B \in I$, if $|A| < |B|$, there exists $x \in B \setminus A$ such that $A \cup \{x\} \in I$

Independent system: (S, I) is called an Independent System if it satisfies the first two properties: non-emptiness and heredity properties. A matroid is an independent system satisfying the exchange property.

Every X ∈ I is typically called an independent set.

Greedy Algorithm and Matroid

Applying greedy algorithm sometimes generates optimal solutions, but sometimes doesn't. So, we need to identify a common pattern, such that applying greedy algorithm on any problem that follows this pattern, will always lead to the optimal solution.

Matroid turns out that not only it is a sufficient condition, it is also a necessary condition: if a problem can be solved by greedy algorithm optimally, it has to be a Matroid.

Given a matroid $M = (S, \mathcal{I})$, we call an element $x \notin A$ an *extension* of $A \in \mathcal{I}$ if we can add x to A while preserving independence; that is, x is an extension of A if $A \cup \{x\} \in \mathcal{I}$. As an example, consider a graphic matroid M_G. If A is an independent set of edges, then edge e is an extension of A if and only if e is not in A and the addition of e to A does not create a cycle.

If A is an independent subset in a matroid M, we say that A is *maximal* if it has no extensions. That is, A is maximal if it is not contained in any larger independent subset of M.

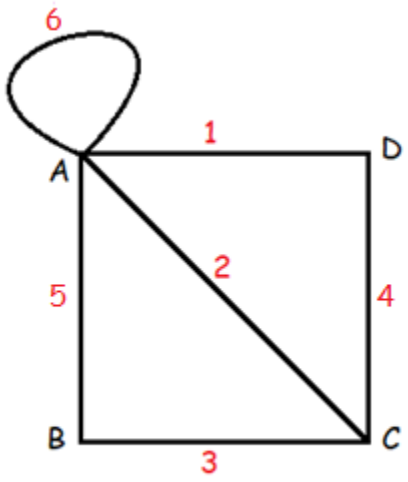
Algorithm to find out maximal independent set of I

Greedy(M):

1. $A = \Phi$
2. for each $x \in S$:
   if $A \cup \{x\} \in I$
   $A \leftarrow A \cup \{x\}$
3. Return A

S = {1,2,3,4,5,6}

I = {{1}, {2}……..{5}, {1,2}, {1,3}……..{ 1,2,3}}



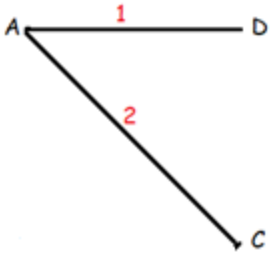
INPUT

Initially taking A = {Φ}

Pass 1: x = 1 that is ∈ S and A ∪ {1} = {1} ∈ I.

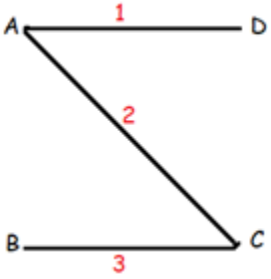
So, A ← A ∪ {1}. Now A = {1}

Pass 2: x = 2 that is ∈ S and A ∪ {2} = {1, 2} ∈ I.



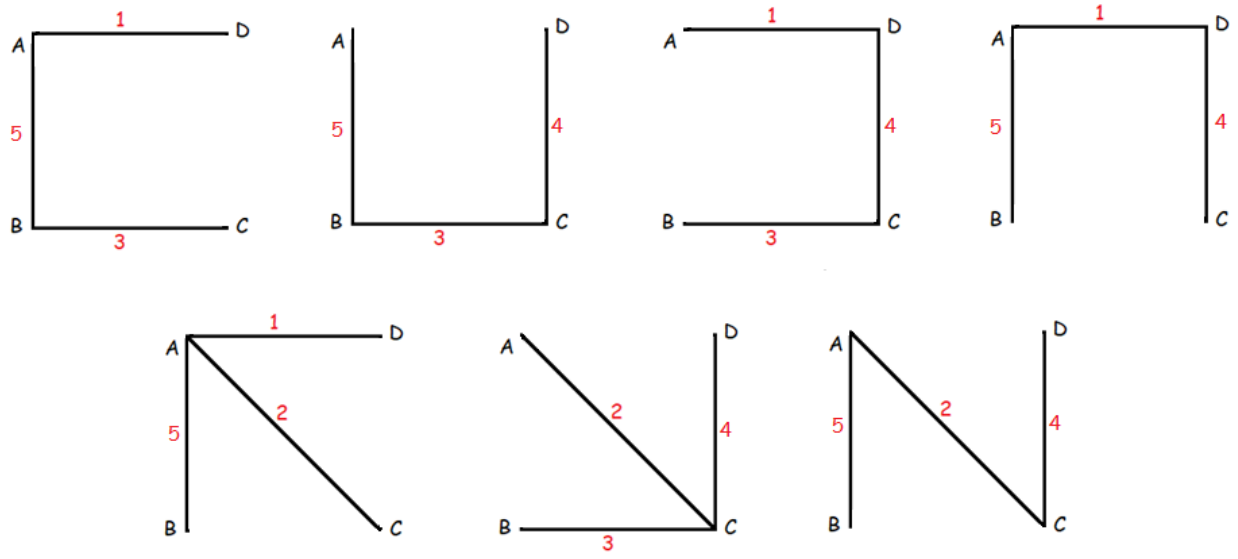
So, A ← A ∪ {2}. Now A = {1, 2}

Pass 3: x = 3 that is ∈ S and A ∪ {3} = {1, 2, 3} ∈ I.



So, A ← A ∪ {3}. Now A = {1, 2, 3}

Now, we can see, no further distinct $\mathbf{x}$ from set $\mathbf{S}$ we can take so that $\mathbf{A \cup \{x\}}$ can be independent. So, we stop the algorithm and the newest A is the maximal independent set depending upon the constraint of the problem that is Acyclicity of edges.



These are all rest of the possible maximal independent sets depending upon the order of choosing each element by x from S every time.

Basis: An independent set X ∈ S is called a basis of the matroid if there doesn't exist Y ∈ S such that X ⊂ Y, i.e., a basis is a maximal independent set of I.

Every basis of a matroid (S, I) has the same size, and we denote this size as the rank of this matroid.

It easy to use the exchange property to prove the uniqueness of the size of bases in a matroid by contradiction:

Suppose a matroid (S, I) has two maximal independent sets X and Y with sizes |X| > |Y |, then there must exist some element x ∈ X\Y such that Y ∪ x ∈ I. Then Y is not a basis of (S, I). A contradiction.

Types of Matroids:

i. **Linear:** Let M be an arbitrary $\mathrm{n \times m}$ matrix; S be set of all columns in M. Let I be a collection of linearly independent subsets of S. Then (S, I) is a Matroid.

(We say a subset $X \subseteq S$ is linearly independent subset if all the columns in X are linearly independent (non-zero) columns.)

ii. Uniform: A subset $X \subseteq \{1, 2, 3, ..., n\}$ is independent if and only if $|X| \leq k$, Any subset of $\{1, 2, 3, ..., n\}$ of size k is a basis of $U_{k,n}$

iii. Graphic:

As another example of matroids, consider the ***graphic matroid*** $M_G = (S_G, \mathcal{I}_G)$ defined in terms of a given undirected graph $G = (V, E)$ as follows:

- The set S_G is defined to be E, the set of edges of G.
- If A is a subset of E, then $A \in \mathcal{I}_G$ if and only if A is acyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a forest.

Greedy on Weighted Matroid

We are given a weighted matroid M = (S, I) and we wish to find an independent set $A \in I$, such that w(A) is maximized. The weight w(x) of any element $x \in S$ is positive. The algorithm takes as input a weighted matroid M = (S, I) with an associated positive weight function w, and it returns an optimal subset A. In our pseudocode, we denote components of M by E and I and the weight function by w. The algorithm is greedy because it considers in turn each element $x \in S$, in order of monotonically decreasing weight, and immediately adds it to the set A being accumulated if $A \cup \{x\}$ is independent.

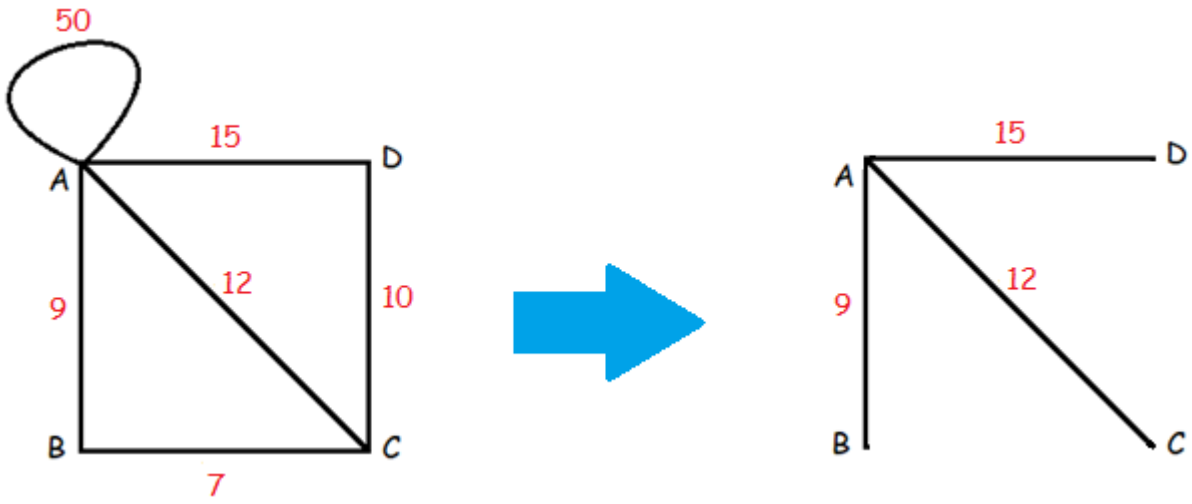
Greedy(M,w):

1. Arrange the elements in $S = \{1, 2, ..., n\}$ so that their weights are in monotonically decreasing order by weight w.

1. $A = \emptyset$
2. for each $x \in S$, taken in monotonically decreasing order by w
3. if $A \cup \{x\} \in I$ then:
4. $A \leftarrow A \cup \{x\}$
5. return A

Line 4 checks whether adding each element x to A would maintain A as an independent set. If A would remain independent, then line 5 adds x to A. Otherwise, x is discarded. Since the empty set is independent, and since each iteration of the **for** loop maintains A's independence, the subset A is always independent, by induction. Therefore, GREEDY always returns an independent subset A. We shall see in a moment that A is a subset of maximum possible weight, so that A is an optimal subset.

The running time of GREEDY is easy to analyze. Let n denote $|S|$. The sorting phase of GREEDY takes time $O(n \lg n)$. Line 4 executes exactly n times, once for each element of S. Each execution of line 4 requires a check on whether or not the set $A \cup \{x\}$ is independent. If each such check takes time $O(f(n))$, the entire algorithm runs in time $O(n \lg n + nf(n))$.



NOTE: If the weights are distinct, this algorithm will generate **maximum** weighted independent set.

Application

Activity or Task scheduling using greedy approach

What is a task?

A task or an activity is process assigned to a processor to be executed.

In case of single task, the processor will start executing the task right away. In case of multiple tasks assigned to a single processor simultaneously, there is multiple ways of sorting the tasks into the order of executing. In this algorithm we will use greedy algorithm on unit time processes.

In this algorithm, the problem is to schedule unit time tasks with deadlines and profit/penalties for a single processor. The problem has following inputs

a) A set S = {t1, t2, t3…tN} for N unit time tasks.
b) An integer set D = {d1, d2, d3…dN} for N deadlines.
c) An integer set W = {w1, w2, w3…wN} for N weights.

For t1 a task which has deadline of d1 and weight of w1.

Deadline represents a time instance, it Is expected for a task to end before it's corresponding deadline.

Weight represents profit/penalty. If the task completes before the deadline, the weight is considered as profit and if the task is considered after the deadline, the weight is considered as the penalty.

The greedy approach prioritizes to minimize the penalty and maximize the profit.

Ex:-

S = {A, B, C, D, E}

D = {2, 1, 2, 1, 3}

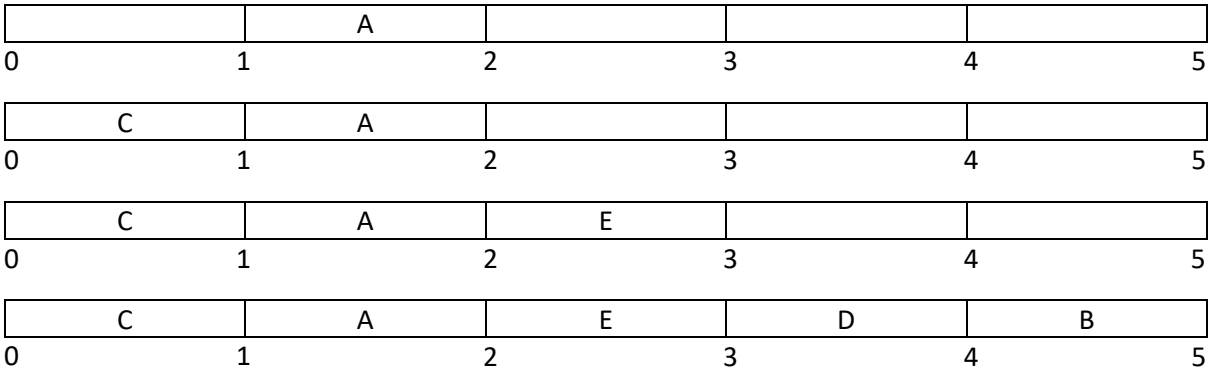
W = {100, 19, 27, 25, 15}

	A	B	C	D	E
Deadline	2	1	2	1	3
Weight	100	19	27	25	15

Sort the jobs based on order of penalty

	A	C	D	B	E
Deadline	2	2	1	1	3
Weight	100	27	25	19	15

Schedule the unit jobs with high penalty list



Profit = 100 + 27 + 15 = 142

Penalty = 25 + 19 = 44

In the greedy algorithm, the sequence is first sorted in decreasing order based on the weight. Then for each task from the sequence, starting from highest weight is put into the result sequence in such a way, if there is some free time before the time instance of the task's corresponding deadline, the task will be assigned to that time slot, else the next task will be checked. After traversing the whole sequence, the tasks that were not assigned to the result sequence will be added in any order.

Algorithm:-

Input: The input sequence containing n tasks and their corresponding deadline and weights.

Output: the output sequence contain n time instances.

1) Sort the input sequence in decreasing order based on the weights.
2) For task 0 to n-1
   a) Check the locations backwards from output sequence before the deadline of the task
   b) If there is any empty location before the time instance in output sequence
      Assign the task to that location.
      Else
      Assign the task to missed list
3) If there is any item in the missed list, append them to the empty locations of the output sequence in any order.

The algorithm takes O(n) time for creating the result sequence, and the time depends on the sort used to sort the sequence.

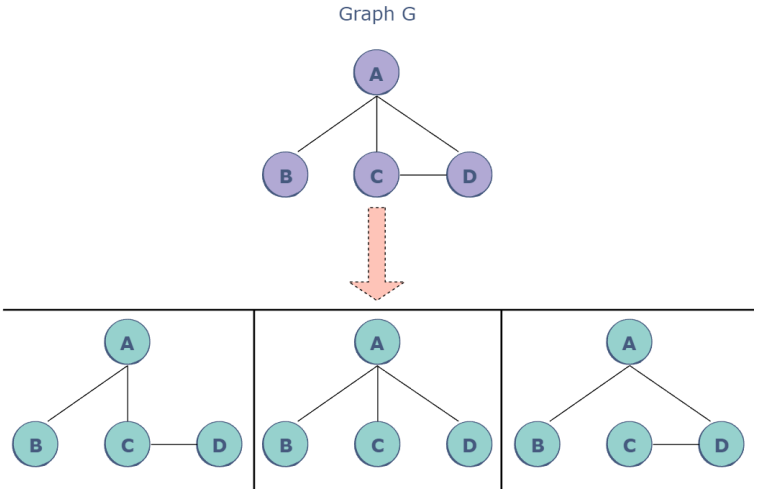
Application

What is a spanning tree?

Given an undirected and connected graph G = (V, E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

Some properties of a spanning tree can be deduced from this definition:

1. Since "a spanning tree covers all of the vertices", it cannot be disconnected.
2. A spanning tree cannot have any cycles *and* consist of (n-1) edges (where *n* is the number of vertices of the graph) because "it uses the minimum number of edges".

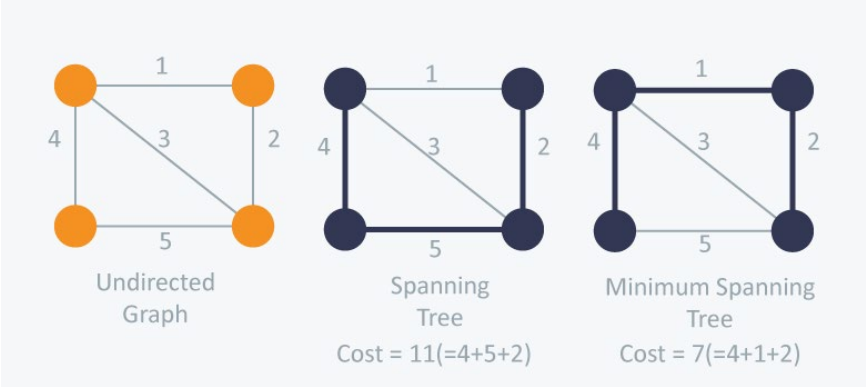


3.No of spanning tree can be construct: $^{|E|}C_{|V|-1}$ -- no. of cycles

#Weighted graphs are the graph data structures in which the edges are given some weight or value based on the type of graph we are representing.

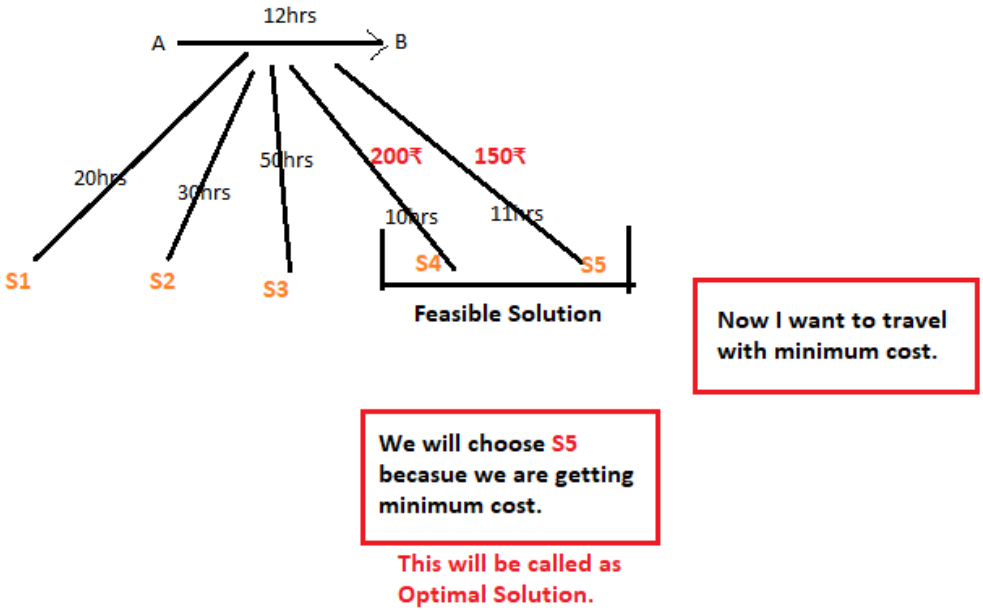
What is minimum spanning tree?

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees.



Greedy Algorithm:

A greedy algorithm is an algorithmic strategy that makes the best optimal choice(Solution looking for either minimum result or maximum result) at each small stage with the goal of this eventually leading to a globally optimum solution. This means that the algorithm picks the best solution at the moment without regard for consequences. It picks the best immediate output, but does not consider the big picture, hence it is considered greedy.



 # In a problem we can have more than 1 feasible solution but can't have more than 1 Optimal Solution.

To solve minimum spanning tree we have two algorithm:

1) Kruskal's Algorithm
2)Prim's Algorithm

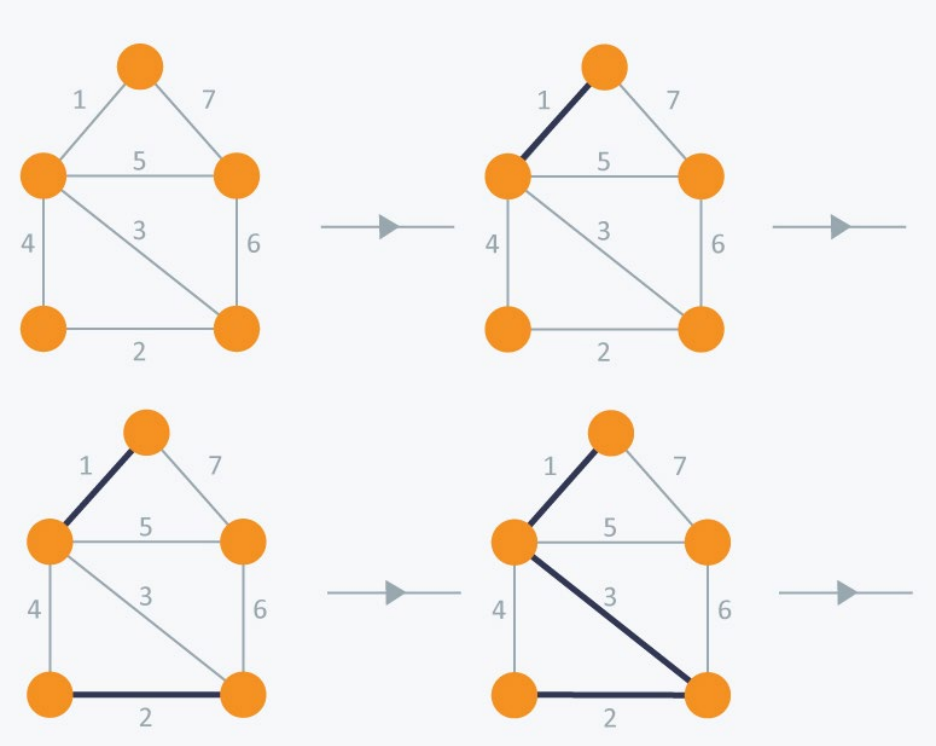
1) Kruskal's Algorithm

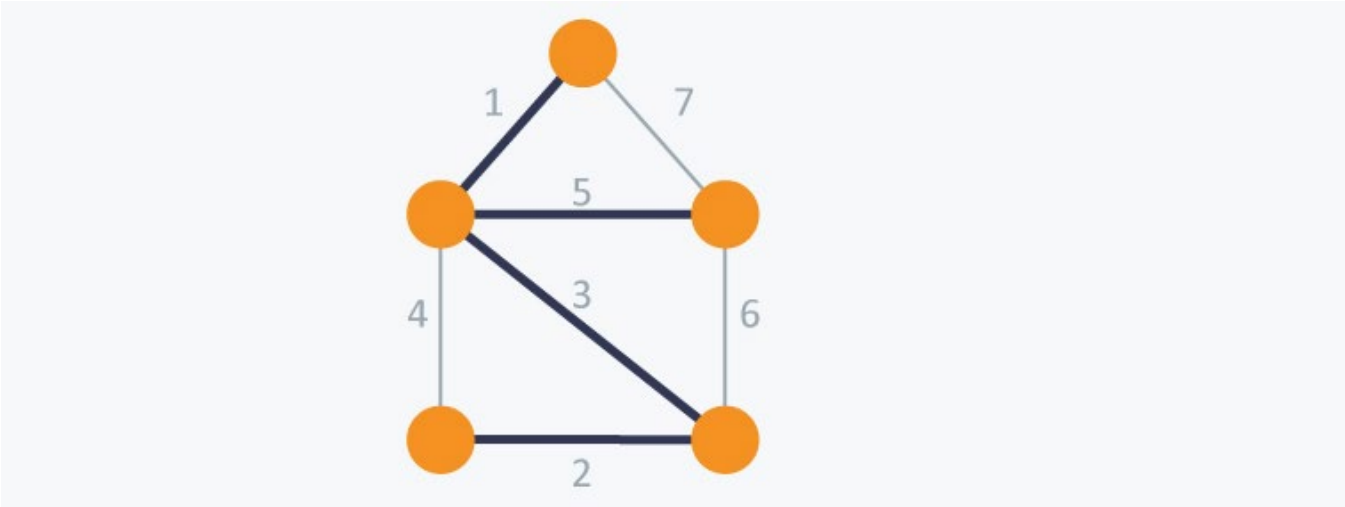
Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?
Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.





In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 (= 1 + 2 + 3 + 5).

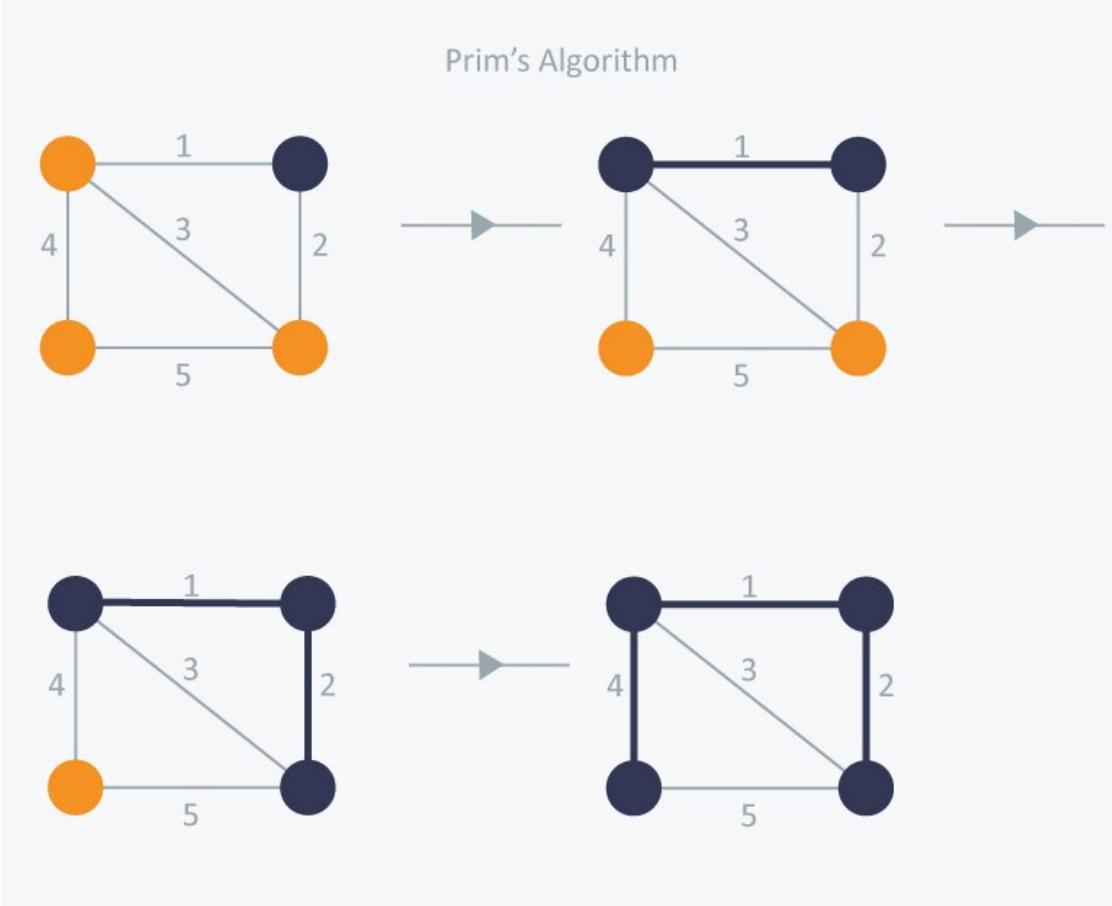
Time Complexity:
In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(VlogE) for best case and O(ElogE) for worst case, which is the overall Time Complexity of the algorithm. The LogE part denotes the heap building time to find the least weighted edge, which occurs V-1 times for best case, and E time for the worst case. In best case, No edge retrieved from the heap forms a cycle, causing the minimum number of edges to be picked from mean heap. In the worst case, all edges need to be traversed before reaching the optimal tree, which is E.

2)Prim's Algorithm:

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.



In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

Time Complexity:

The time complexity of the Prim's Algorithm is $O((V^2 \log V)$ for worst case and $O(V \log V)$. LogV time is required for heap building for all adjacent vertices of already selected vertice(s), which can be up to V-1. The thing occurs for V-1 times, since the heap is built each time a new vertex is selected. In best case, the vertices chosen does not form a cycle, which limits the heap building once for each vertex added, causing the complexity be $O(V \log V)$. In worst case, the heap is recreated v-1 times for each vertex, causing the complexity be $O(V^2 \log V)$.

Difference between Prim's and Kruskal's Algorithm:

S.No.	Prim's Algorithm	Kruskal's Algorithm
1	This algorithm begins to construct the shortest spanning tree from any vertex in the graph.	This algorithm begins to construct the shortest spanning tree from the vertex having the lowest weight in the graph.
2	To obtain the minimum distance, it traverses one node more than one time.	It crosses one node only one time.
3	The time complexity of Prim's algorithm is O(V2).	The time complexity of Kruskal's algorithm is O(E log V).
4	In Prim's algorithm, all the graph elements must be connected.	Kruskal's algorithm may have disconnected graphs.
5	When it comes to dense graphs, the Prim's algorithm runs faster.	When it comes to sparse graphs, Kruskal's algorithm runs faster.
6	It prefers list data structure.	It prefers the heap data structure.