Assignment No. 3

Title	Implement Greedy search algorithm
Aim/Problem Statement	Implement Greedy search algorithm for any of the following application: I. Selection Sort II. Minimum Spanning Tree III. Single-Source Shortest Path Problem IV. Job Scheduling Problem V. Prim's Minimal Spanning Tree Algorithm VI. Kruskal's Minimal Spanning Tree Algorithm VII. Dijkstra's Minimal Spanning Tree Algorithm
CO Mapped	CO1: Design system using different informed search / uninformed search or heuristic approaches
Pre-requisite	
Learning Objective	

Theory:

Greedy strategy:

The greedy method is one of the strategies like Divide and conquer used to solve the problems. This method is used for solving optimization problems. An optimization problem is a problem that demands either maximum or minimum results.

- 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.
- So the problems where choosing locally optimal also leads to global solution are best fit for Greedy.

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

Selection Sort:

The selection sort algorithm sorts a list by repeatedly finding the minimum element from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- The sublist that is already sorted.
- Remaining sublist which is unsorted.

In each iteration of selection sort, the minimum element of the unsorted sublist is picked and moved to the sorted sublist.

A selection sort could indeed be described as a greedy algorithm, in the sense that it:

- tries to choose an output (a permutation of its inputs) that optimizes a certain measure ("sortedness", which could be measured in various ways, e.g. by number of inversions), and
- does so by breaking the task into smaller subproblems (for selection sort, finding the *k*-th element in the output permutation) and picking the locally optimal solution to each subproblem.

Time and Space Complexity of Selection sort:

The time complexity for the selection sort algorithm is O(N2) because it works on a nested for loop which iterates through the list for each separate element. The space complexity is O(1) as no additional memory is required.

When to use or avoid Selection Sort?

When to use Selection Sort:

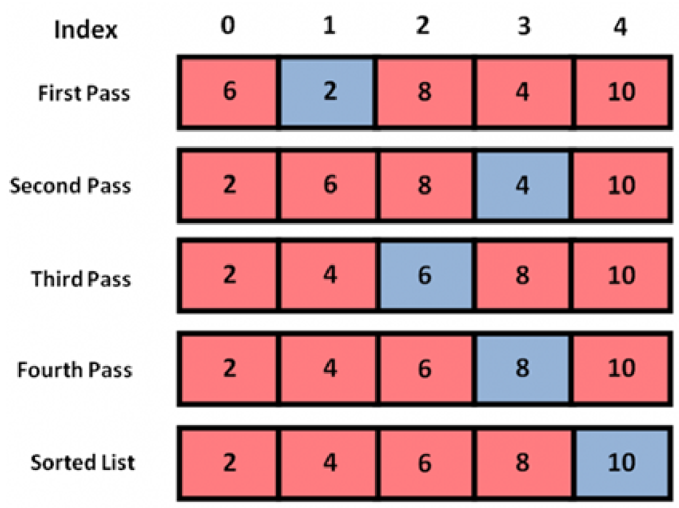
When we have insufficient memory and need a space efficient algorithm.

It is a simple, easy to implement algorithm and can be used for sorting small number of elements.

When to avoid Selection Sort:

The average time complexity for selection sort is quite poor. Thus, we avoid using it for sorting lists of bigger sizes.

Example of Selection sort:

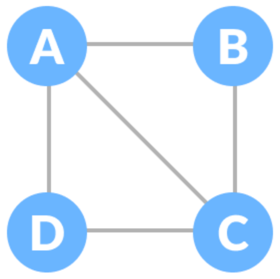


Selection Sort Algorithm

```
selectionSort(array, size)
 repeat (size - 1) times
 set the first unsorted element as the minimum
 for each of the unsorted elements
   if element < currentMinimum
     set element as new minimum
 swap minimum with first unsorted position
end selectionSort
```

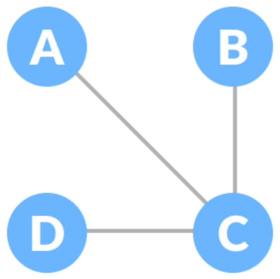
Minimum Spanning Tree

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



Undirected Graph

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



Connected Graph

Spanning tree

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 edges may or may not have weights assigned to them.

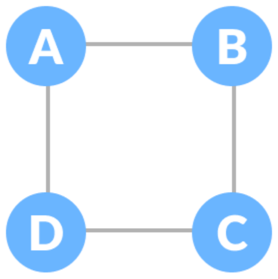
> The total number of spanning trees with `n` vertices that can be created from a complete graph is equal to $n^{(n-2)}$.

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.

Example of a Spanning Tree

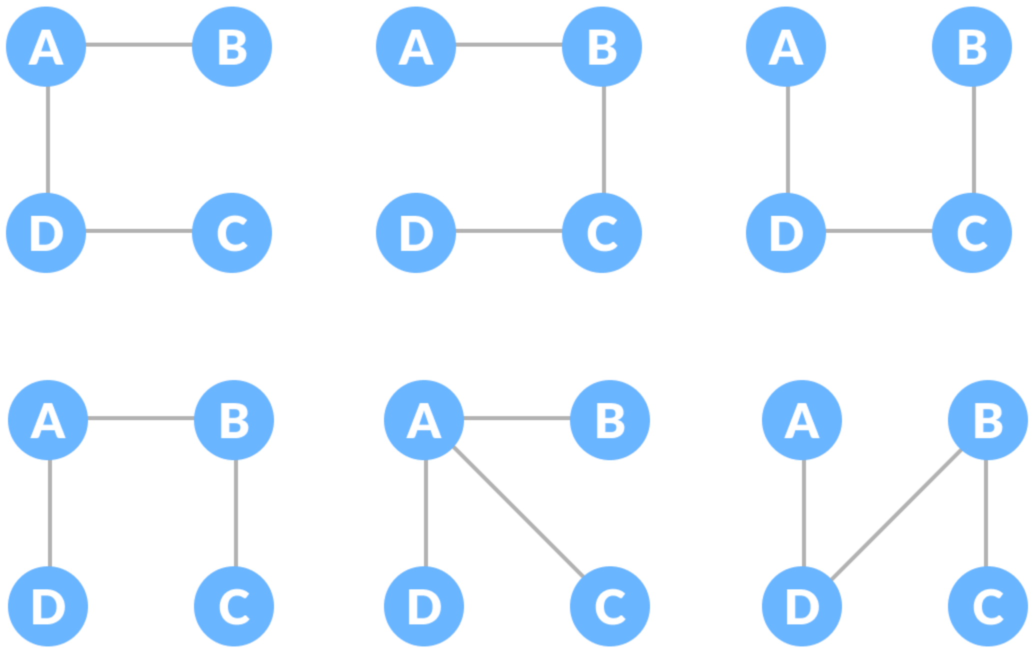
Let's understand the spanning tree with examples below:

Let the original graph be:



Normal graph

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



Examples of spanning tree

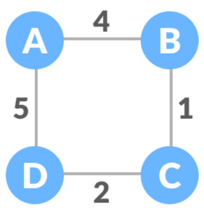
Minimum Spanning Tree

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

Example of a Spanning Tree

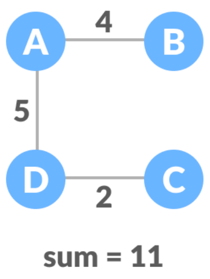
Let's understand the above definition with the help of the example below.

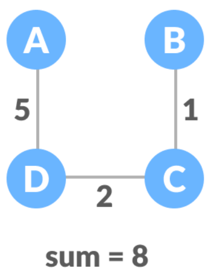
The initial graph is:



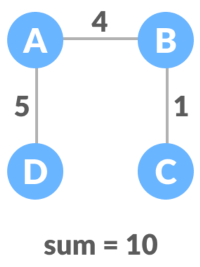
Weighted graph

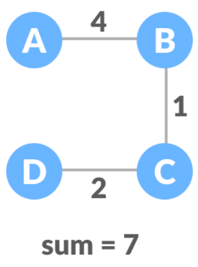
The possible spanning trees from the above graph are:





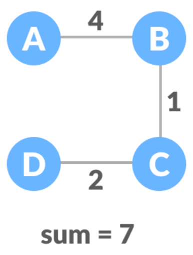
Minimum spanning tree – 1 Minimum spanning tree – 2





Minimum spanning tree – 3 Minimum spanning tree - 4

The minimum spanning tree from the above spanning trees is:



Minimum spanning tree

Minimum Spanning tree Applications

- To find paths in the map
- To design networks like telecommunication networks, water supply networks, and electrical grids.

Single-Source Shortest Path Problem

In a **shortest- paths problem**, we are given a weighted, directed graphs G = (V, E), with weight function **w: E → R** mapping edges to real-valued weights. The weight of path p = $(v_0, v_1, \ldots\ldots v_k)$ is the total of the weights of its constituent edges:

$$w(P) = \sum_{i=1}^{k} w(v_{i-1} v_i)$$

We define the shortest - path weight from u to v by δ(u,v) = min (w (p): u→v), if there is a path from u to v, and δ(u,v)= ∞, otherwise.

The **shortest path** from vertex s to vertex t is then defined as any path p with weight w (p) = δ(s,t).

The **breadth-first- search algorithm** is the shortest path algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight.

In a **Single Source Shortest Paths Problem**, we are given a Graph G = (V, E), we want to find the shortest path from a given source vertex s ∈ V to every vertex v ∈ V.

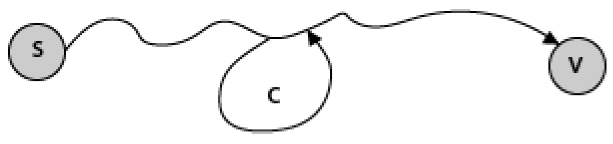
Variants:

There are some variants of the shortest path problem.

- **Single- destination shortest - paths problem:** Find the shortest path to a given destination vertex t from every vertex v. By shift the direction of each edge in the graph, we can shorten this problem to a single - source problem.
- **Single - pair shortest - path problem:** Find the shortest path from u to v for given vertices u and v. If we determine the single - source problem with source vertex u, we clarify this problem also. Furthermore, no algorithms for this problem are known that run asymptotically faster than the best single - source algorithms in the worst case.
- **All - pairs shortest - paths problem:** Find the shortest path from u to v for every pair of vertices u and v. Running a single - source algorithm once from each vertex can clarify this problem; but it can generally be solved faster, and its structure is of interest in the own right.

Shortest Path: Existence:

If some path from s to v contains a negative cost cycle then, there does not exist the shortest path. Otherwise, there exists a shortest s - v that is simple.



Cost of C < 0

Job Scheduling Problem

Job scheduling is the problem of scheduling jobs out of a set of N jobs on a single processor which maximizes profit as much as possible. Consider N jobs, each taking unit time for execution. Each job is having some profit and deadline associated with it. Profit earned only if the job is completed on or before its deadline. Otherwise, we have to pay a profit as a penalty. Each job has deadline $d_i \geq 1$ and profit $p_i \geq 0$. At a time, only one job can be active on the processor.

The job is feasible only if it can be finished on or before its deadline. A feasible solution is a subset of N jobs such that each job can be completed on or before its deadline. An optimal solution is a solution with maximum profit. The simple and inefficient solution is to generate all subsets of the given set of jobs and find the feasible set that maximizes the profit. For N jobs, there exist 2^N schedules, so this brute force approach runs in $O(2^N)$ time.

However, the greedy approach produces an optimal result in fairly less time. As each job takes the same amount of time, we can think of the schedule S consisting of a sequence of job slots 1, 2, 3, …, N, where S(t) indicates job scheduled in slot t. Slot t has a span of (t – 1) to t. S(t) = 0 implies no job is scheduled in slot t.

Schedule S is an array of slots S(t), S(t) ∈ {1, 2, 3, …, N} for each t ∈ {1, 2, 3, …, N}

Schedule S is *feasible* if,

- S(t) = i, then $t \le d_i$ (Scheduled job must meet its deadline)
- Each job can be scheduled at max once.

Our goal is to find a feasible schedule S which maximizes the profit of scheduled job. The goal can be achieved as follow: Sort all jobs in decreasing order of profit. Start with the empty schedule, select one job at a time and if it is feasible then schedule it in the *latest possible slot*.

Algorithm for Job Scheduling

```
**Algorithm** JOB_SCHEDULING( J, D, P )

// Description : Schedule the jobs using the greedy approach which maximizes the profit

// Input :

J: Array of N jobs

D: Array of the deadline for each job

P: Array of profit associated with each job

// Output : Set of scheduled job which gives maximum profit

Sort all jobs in J in decreasing order of profit

S ← Φ // S is set of scheduled jobs, initially it is empty

SP ← 0 // Sum is the profit earned

for i ← 1 to N do

    if Job J[i] is feasible then

            Schedule the job in the latest possible free slot meeting its deadline.
```

```
        S ← S ∪ J[i]

        SP ← SP + P[i]

    end

end
```

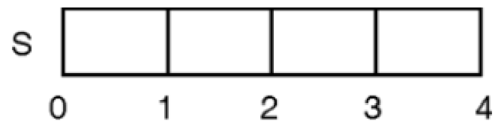
Example of Job scheduling problem

Problem: Solve the following job scheduling with deadlines problem using the greedy method. Number of jobs N = 4. Profits associated with Jobs : (P_1, P_2, P_3, P_4) = (100, 10, 15, 27). Deadlines associated with jobs (d_1, d_2, d_3, d_4) = (2, 1, 2, 1)

Solution:

Sort all jobs in descending order of profit.

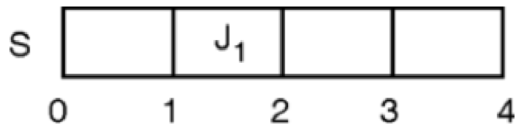
So, P = (100, 27, 15, 10), J = (J_1, J_4, J_3, J_2) and D = (2, 1, 2, 1). We shall select one by one job from the list of sorted jobs, and check if it satisfies the deadline. If so, schedule the job in the latest free slot. If no such slot is found, skip the current job and process the next one. Initially,



Profit of scheduled jobs, SP = 0

Iteration 1:

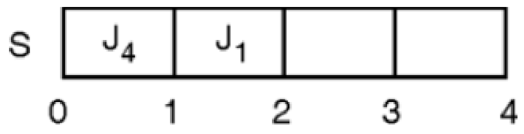
Deadline for job J_1 is 2. Slot 2 (t = 1 to t = 2) is free, so schedule it in slot 2. Solution set S= {J_1}, and Profit SP = {100}



Iteration 2:

Deadline for job J_4 is 1. Slot 1 (t = 0 to t = 1) is free, so schedule it in slot 1.

Solution set S = {J_1,J_4}, and Profit SP = {100, 27}



Iteration 3:

Job J_3 is not feasible because first two slots are already occupied and if we schedule J_3 any time later t = 2, it cannot be finished before its deadline 2. So job J_3 is discarded,

Solution set S = $\{J_1, J_4\}$, and Profit SP = {100, 27}

Iteration 4:

Job J_2 is not feasible because first two slots are already occupied and if we schedule J_2 any time later t = 2, it cannot be finished before its deadline 1. So job J_2 is discarded,

Solution set S = $\{J_1, J_4\}$, and Profit SP = {100, 27}

With the greedy approach, we will be able to schedule two jobs $\{J_1, J_4\}$, which gives a profit of 100 + 27 = 127 units.

Complexity Analysis of Job Scheduling

Simple greedy algorithm spends most of the time looking for the latest slot a job can use. On average, N jobs search N/2 slots. This would take $O(N^2)$ time.

However, with the use of set data structure (find and union), the algorithm runs nearly in O(N) time.

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

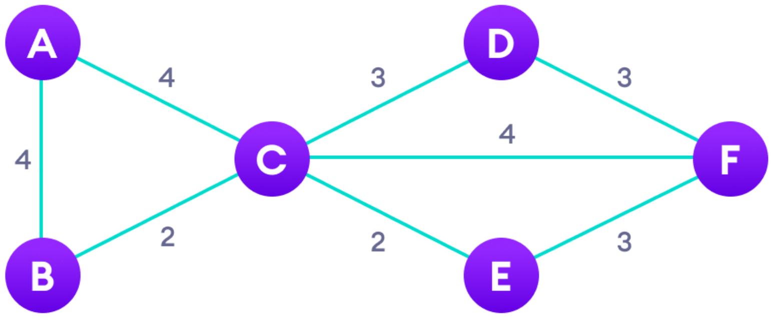
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

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:

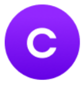
1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

Example of Prim's algorithm



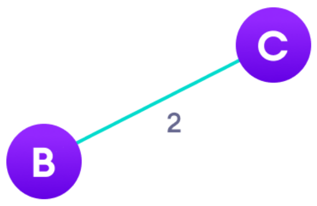
Step: 1

Start with a weighted graph



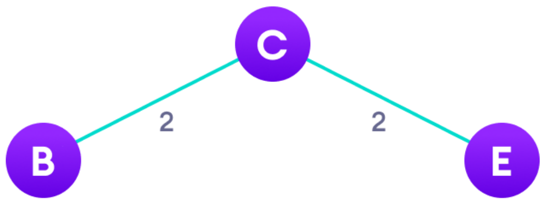
Step: 2

Choose a vertex



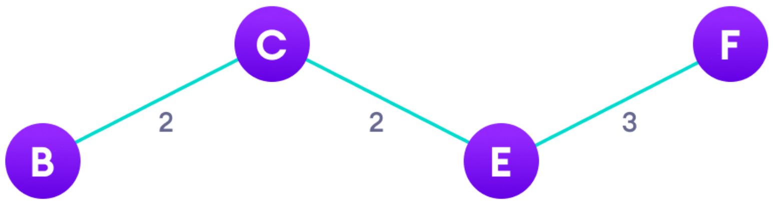
Step: 3

Choose the shortest edge from this vertex and add it



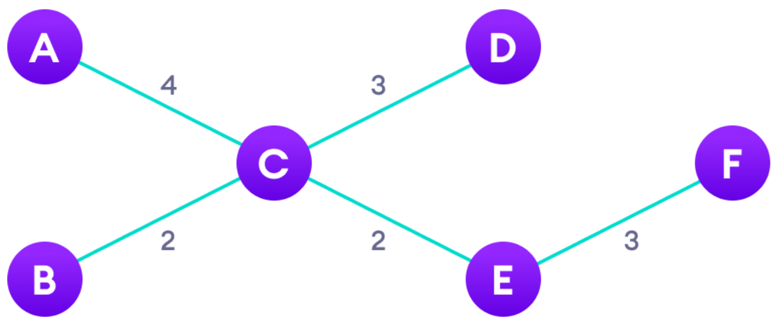
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

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



Step: 6

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.

```
T = ∅;

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 Complexity

The time complexity of Prim's algorithm is `O(E log V)`.

Prim's Algorithm Application

- Laying cables of electrical wiring
- In network designed
- To make protocols in network cycles

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

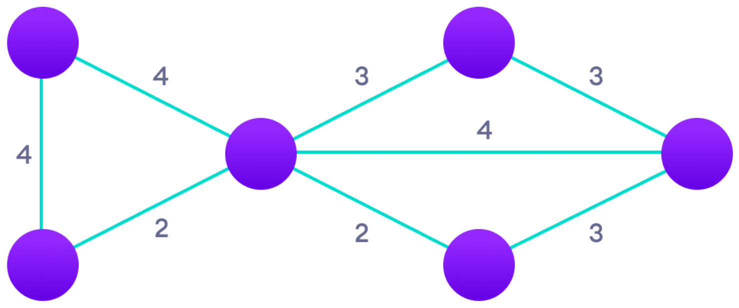
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

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:

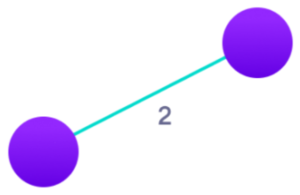
1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Example of Kruskal's algorithm



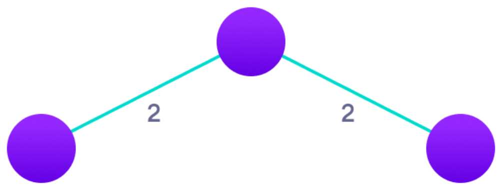
Step: 1

Start with a weighted graph



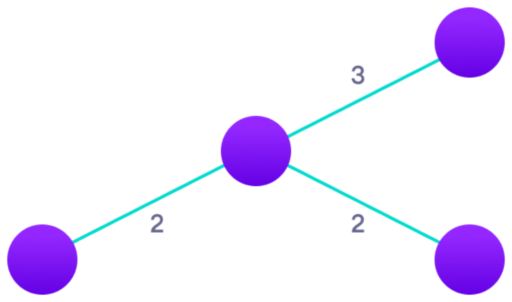
Step: 2

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



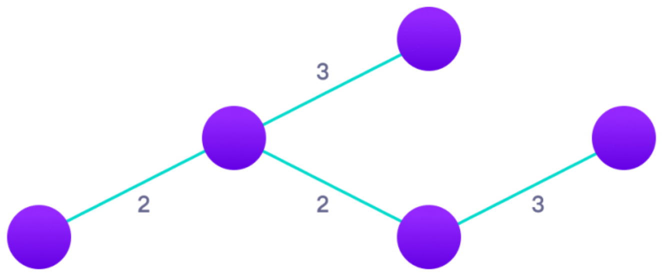
Step: 3

Choose the next shortest edge and add it



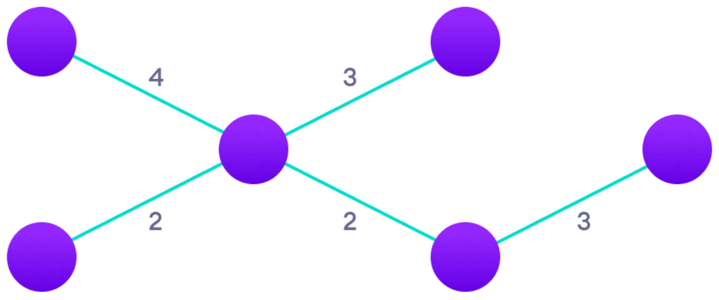
Step: 4

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



Step: 5

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



Step: 6

Repeat until you have a spanning tree

Kruskal Algorithm Pseudocode

Any minimum spanning tree algorithm revolves around checking if adding an edge creates a loop or not.

The most common way to find this out is an algorithm called Union Find. 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.

```
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 Complexity

The time complexity Of Kruskal's Algorithm is: O(E log E).

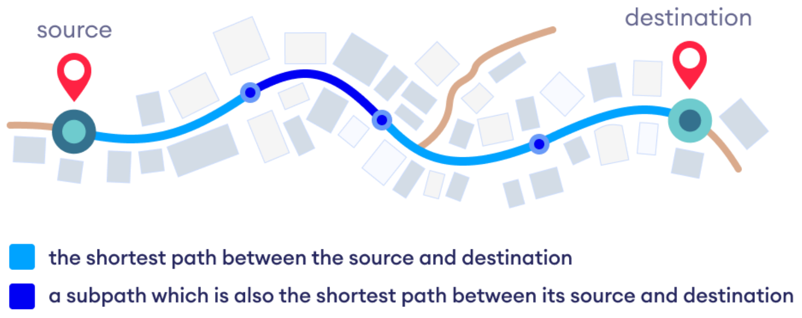
Dijkstra's Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

How Dijkstra's Algorithm works

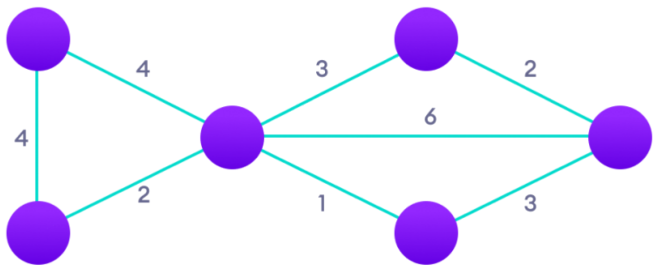
Dijkstra's Algorithm works on the basis that any subpath `B -> D` of the shortest path `A -> D` between vertices A and D is also the shortest path between vertices B and D.


source
destination
the shortest path between the source and destination
a subpath which is also the shortest path between its source and destination

Each subpath is the shortest path. Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors. The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

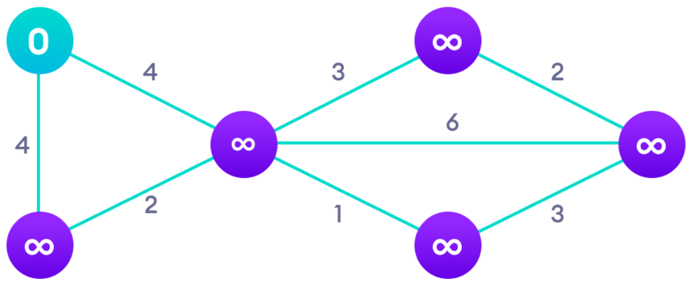
Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.



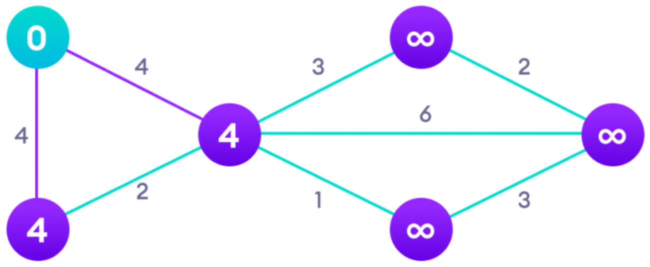
Step: 1

Start with a weighted graph



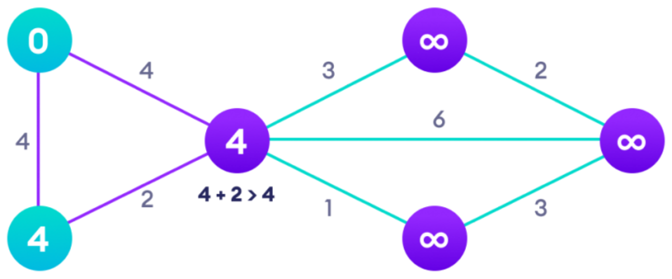
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



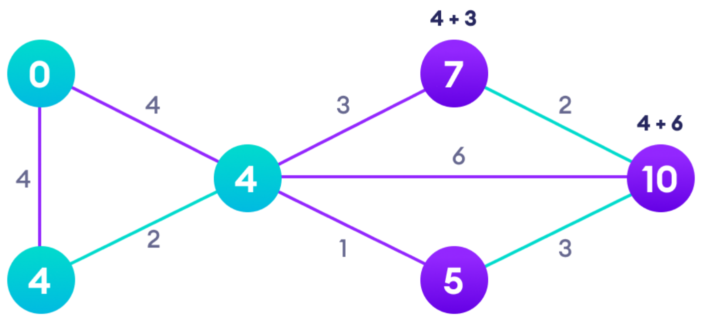
Step: 3

Go to each vertex and update its path length



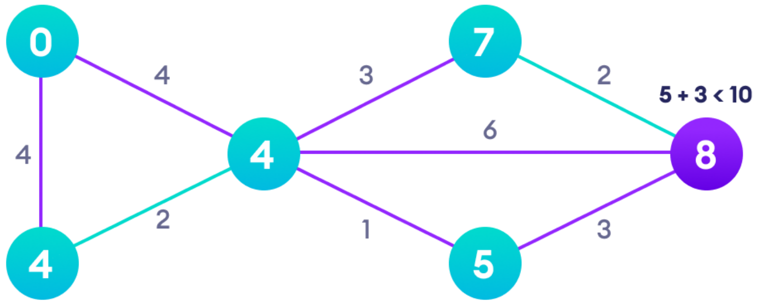
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



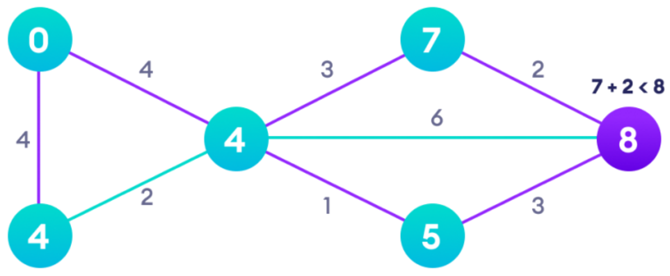
Step: 5

Avoid updating path lengths of already visited vertices



Step: 6

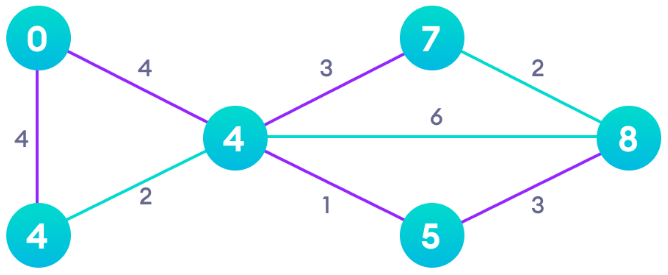
After each iteration, we pick the unvisited vertex with the least path length. So we choose 5



Step: 7

before 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Djikstra's algorithm pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size *v*, where *v* is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Dijkstra's Algorithm Complexity

Time Complexity: `O(E Log V)`

where, E is the number of edges and V is the number of vertices.

Space Complexity: `O(V)`

Questions:

1. **What are the advantages and disadvantages of greedy method?**
2. **What are the characteristics of Greedy algorithm?**