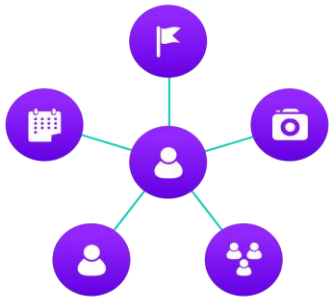
UNIT-V: Graph Data Structure

Syllabus: Graphs: ADT, data structure for graphs, graph traversal, Transitive closure, directed acyclic graph, shortest paths [weighted graphs, Dijkstra's algorithm], minimum spanning trees [Prim's, Kruskal's, disjoint partitions, union-find structures].

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this through an example. On Facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.

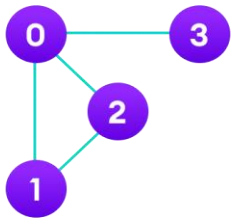


Example of graph data structure

All of Facebook is then a collection of these nodes and edges. This is because Facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E represented as ordered pairs of vertices (u,v)



Vertices and edges

In the graph,
V = {0, 1, 2, 3}
E = {(0,1), (0,2), (0,3), (1,2)}
G = {V, E}

Graph Terminology

Adjacency: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.

Path: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.

Directed Graph: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Graph Representation

Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is

Graph adjacency matrix

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

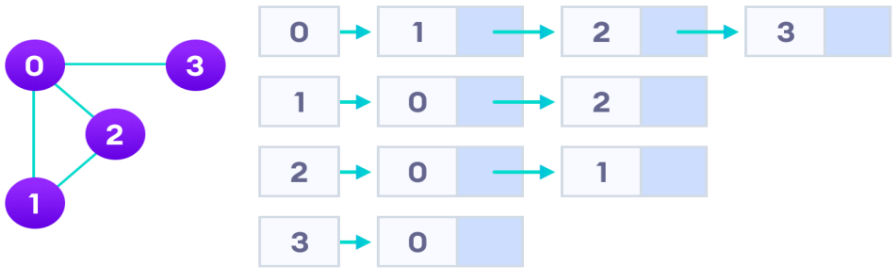
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

2. Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



Adjacency list representation

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

Graph Operations

The most common graph operations are:

- Check if the element is present in the graph
- Graph Traversal
- Add elements(vertex, edges) to graph
- Finding the path from one vertex to another

Graph Traversal

There are two types of graph traversal

1. Depth First Search(DFS)
2. Breadth First Search(BFS)

DFS Algorithm

Traversal means visiting all the nodes of a graph. Depth first traversal or Depth first Search is a recursive algorithm for searching all the vertices of a graph or tree data structure. In this article, you will learn with the help of examples the DFS algorithm, DFS pseudocode, and the code of the depth first search algorithm with implementation in C++, C, Java, and Python programs.

DFS algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

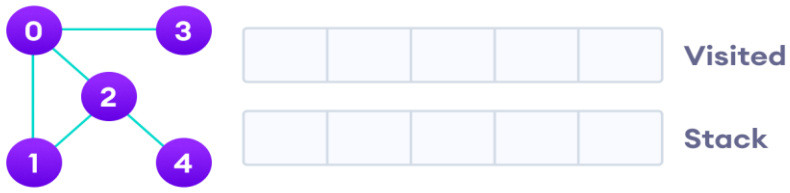
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

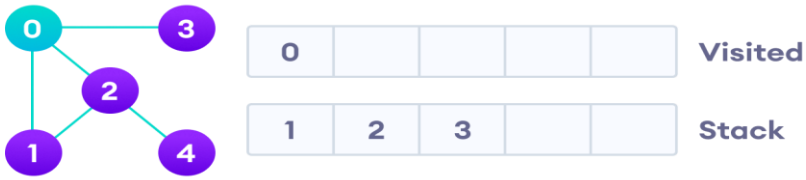
DFS example

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



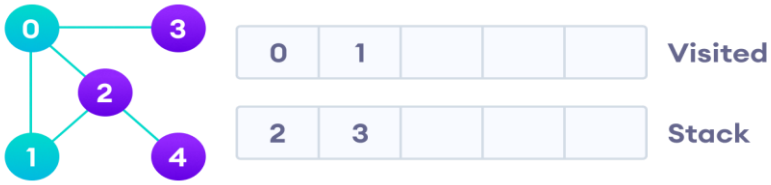
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



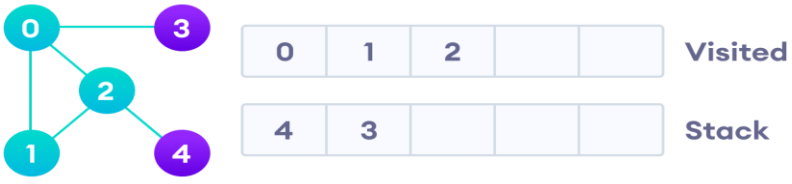
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

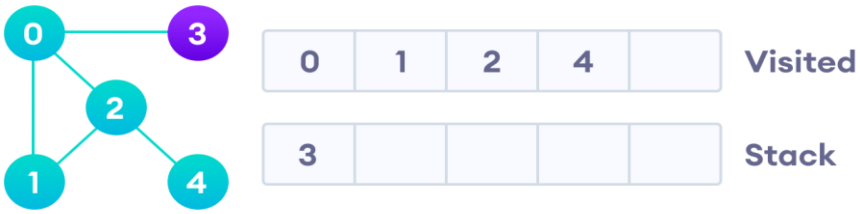


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

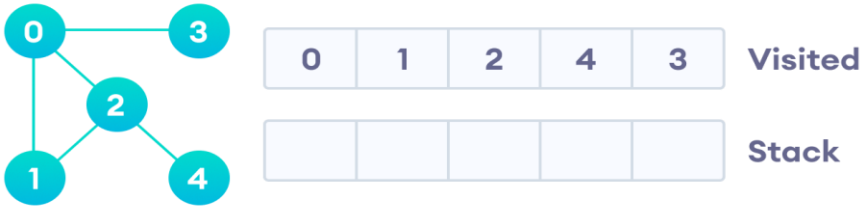


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

DFS pseudocode (recursive implementation)

The pseudocode for DFS is shown below. In the init() function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

```
DFS(G, u)
   u.visited = true
   for each v ∈ G.Adj[u]
      if v.visited == false
         DFS(G,v)

init() {
   For each u ∈ G
      u.visited = false
    For each u ∈ G
     DFS(G, u)
}
```

The code for the Depth First Search Algorithm with an example is shown below.

The code has been simplified so that we can focus on the algorithm rather than other details.

DFS algorithm in Python

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    print(start)

    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited

graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}

dfs(graph, '0')
```

DFS Algorithm Complexity

- The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.
- The space complexity of the algorithm is O(V).

DFS Algorithm Applications

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

Breadth First Search(BFS)

Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

BFS algorithm

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

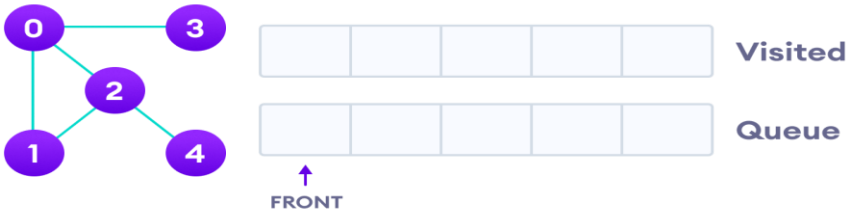
1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.

3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

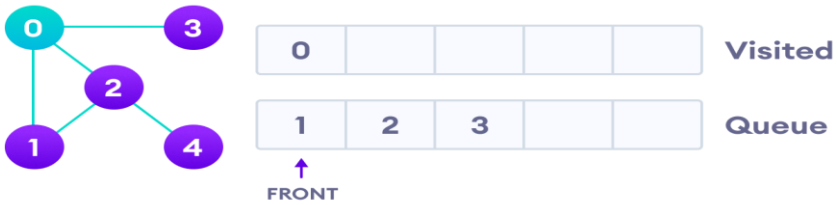
BFS example

Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



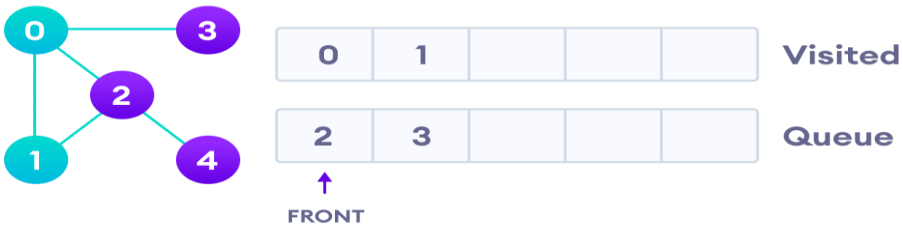
Undirected graph with 5 vertices

We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



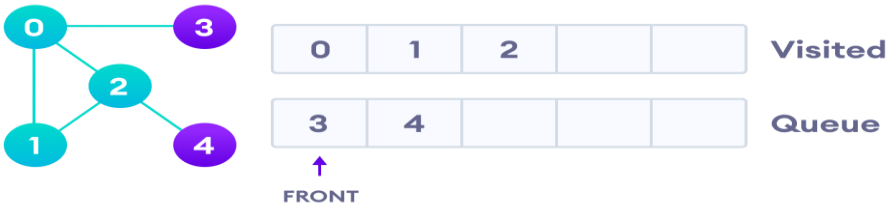
Visit start vertex and add its adjacent vertices to queue

Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

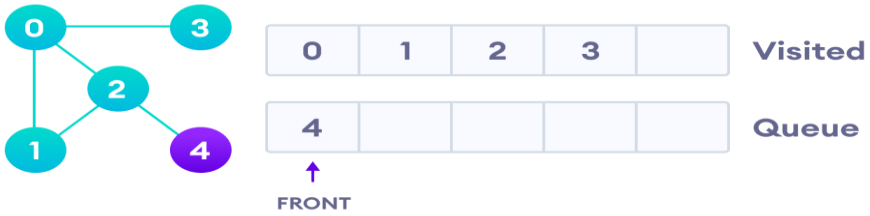


Visit the first neighbour of start node 0, which is 1

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.

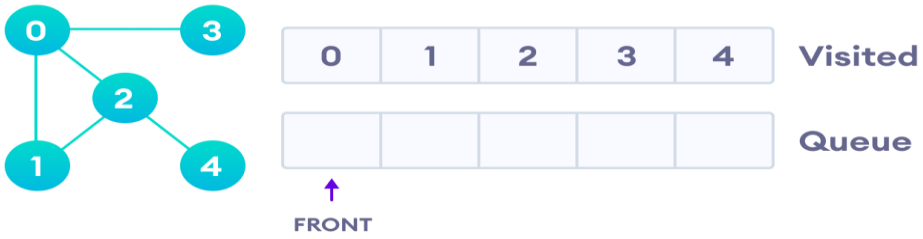


Visit 2 which was added to queue earlier to add its neighbours



4 remains in the queue

Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Visit last remaining item in the stack to check if it has unvisited neighbors

Since the queue is empty, we have completed the Breadth First Traversal of the graph.

BFS pseudocode

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
   remove the head u of Q
   mark and enqueue all (unvisited) neighbours of u
```

The code for the Breadth First Search Algorithm with an example is shown below.

The code has been simplified so that we can focus on the algorithm rather than other details.

```
# BFS algorithm in Python
import collections

# BFS algorithm
def bfs(graph, root):
```

```
    visited, queue = set(), collections.deque([root])
    visited.add(root)

    while queue:

        # Dequeue a vertex from queue
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")

        # If not visited, mark it as visited, and
        # enqueue it
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)

if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

BFS Algorithm Complexity

- The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.
- The space complexity of the algorithm is O(V).

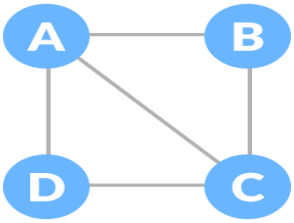
BFS Algorithm Applications

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph
6. In minimum spanning tree

Spanning Tree and Minimum Spanning Tree

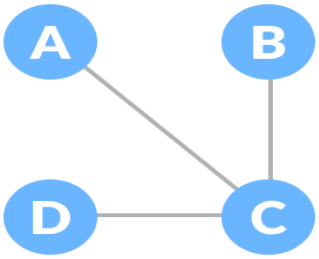
Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.

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



Undirected Graph

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



Connected Graph

Adjacency Matrix

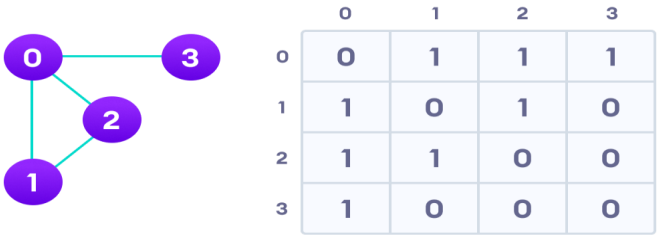
An adjacency matrix is a way of representing a graph G = {V, E} as a matrix of booleans.

Adjacency matrix representation

The size of the matrix is VxV where V is the number of vertices in the graph and the value of an entry Aij is either 1 or 0 depending on whether there is an edge from vertex i to vertex j.

Adjacency Matrix Example

The image below shows a graph and its equivalent adjacency matrix.



Adjacency matrix from a graph

In the case of undirected graphs, the matrix is symmetric about the diagonal because of every edge (i, j), there is also an edge (j, i).

Pros of an adjacency matrix

The basic operations like adding an edge, removing an edge, and checking whether there is an edge from vertex i to vertex j are extremely time-efficient, constant-time operations.

If the graph is dense and the number of edges is large, the adjacency matrix should be the first choice. Even if the graph and the adjacency matrix is sparse, we can represent it using data structures for sparse matrices.

The biggest advantage, however, comes from the use of matrices. The recent advances in hardware enable us to perform even expensive matrix operations on the GPU.

By performing operations on the adjacent matrix, we can get important insights into the nature of the graph and the relationship between its vertices.

Cons of the adjacency matrix

The VxV space requirement of the adjacency matrix makes it a memory hog.

Graphs out in the wild usually don't have too many connections and this is the major reason why adjacency lists are the better choice for most tasks.

While basic operations are easy, operations like inEdges and outEdges are expensive when using the adjacency matrix representation.

If you know how to create two-dimensional arrays, you also know how to create an adjacency matrix.

```
# Adjacency Matrix representation in Python
class Graph(object):

    # Initialize the matrix
    def __init__(self, size):
        self.adjMatrix = []
        for i in range(size):
            self.adjMatrix.append([0 for i in range(size)])
        self.size = size

    # Add edges
    def add_edge(self, v1, v2):
        if v1 == v2:
            print("Same vertex %d and %d" % (v1, v2))
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1

    # Remove edges
    def remove_edge(self, v1, v2):
        if self.adjMatrix[v1][v2] == 0:
            print("No edge between %d and %d" % (v1, v2))
            return
        self.adjMatrix[v1][v2] = 0
        self.adjMatrix[v2][v1] = 0
```

```
    def __len__(self):
        return self.size

    # Print the matrix
    def print_matrix(self):
        for row in self.adjMatrix:
            for val in row:
                print('{:4}'.format(val)),
            print

def main():
    g = Graph(5)
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(2, 3)

    g.print_matrix()

if __name__ == '__main__':
    main()
```

Adjacency Matrix Applications

- Creating a routing table in networks
- Navigation tasks

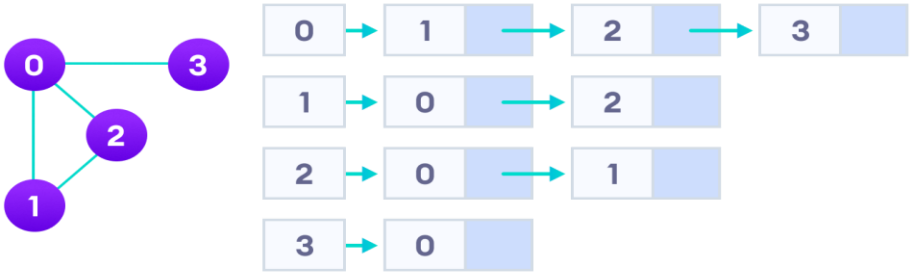
Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

Adjacency List representation

A graph and its equivalent adjacency list representation are shown below.



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a sparse graph with millions of vertices and edges, this can mean a lot of saved space.

Adjacency List Structure

The simplest adjacency list needs a node data structure to store a vertex and a graph data structure to organize the nodes.

We stay close to the basic definition of a graph - a collection of vertices and edges {V, E}. For simplicity, we use an unlabeled graph as opposed to a labeled one i.e. the vertices are identified by their indices 0,1,2,3.

Let's dig into the data structures at play here.

```
struct node
{
   int vertex;
   struct node* next;
};

struct Graph
{
   int numVertices;
   struct node** adjLists;
};
```

Don't let the struct node** adjLists overwhelm you.

All we are saying is we want to store a pointer to struct node*. This is because we don't know how many vertices the graph will have and so we cannot create an array of Linked Lists at compile time.

Adjacency List C++

It is the same structure but by using the in-built list STL data structures of C++, we make the structure a bit cleaner. We are also able to abstract the details of the implementation.

```
class Graph
{
   int numVertices;
   list<int> *adjLists;

public:
   Graph(int V);
   void addEdge(int src, int dest);
};
```

Adjacency List Java

We use Java Collections to store the Array of Linked Lists.

```
class Graph
{
   private int numVertices;
   private LinkedList<integer> adjLists[];
}
```

The type of LinkedList is determined by what data you want to store in it. For a labeled graph, you could store a dictionary instead of an Integer

Adjacency List Python

There is a reason Python gets so much love. A simple dictionary of vertices and their edges is a sufficient representation of a graph. You can make the vertex itself as complex as you want.

```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}

# Adjacency List representation in Python

class AdjNode:
   def __init__(self, value):
      self.vertex = value
      self.next = None


class Graph:
   def __init__(self, num):
      self.V = num
      self.graph = [None] * self.V

   # Add edges
   def add_edge(self, s, d):
      node = AdjNode(d)
      node.next = self.graph[s]
      self.graph[s] = node

      node = AdjNode(s)
      node.next = self.graph[d]
      self.graph[d] = node
```

```
    # Print the graph
    def print_agraph(self):
        for i in range(self.V):
            print("Vertex " + str(i) + ":", end="")
            temp = self.graph[i]
            while temp:
                print(" -> {}".format(temp.vertex), end="")
                temp = temp.next
            print(" \n")

if __name__ == "__main__":
    V = 5

    # Create graph and edges
    graph = Graph(V)
    graph.add_edge(0, 1)
    graph.add_edge(0, 2)
    graph.add_edge(0, 3)
    graph.add_edge(1, 2)

    graph.print_agraph()
```

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.

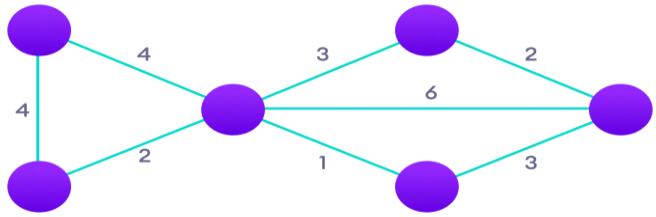
Each subpath is the shortest path

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.

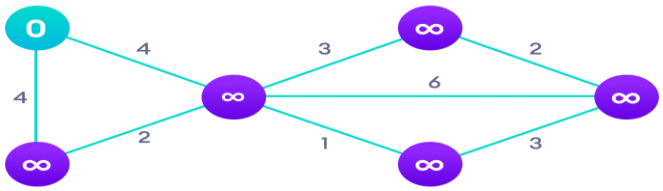
Example of Dijkstra's algorithm

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



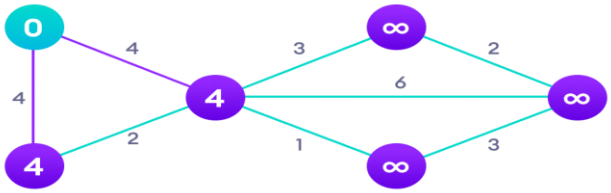
Step: 1

Start with a weighted graph



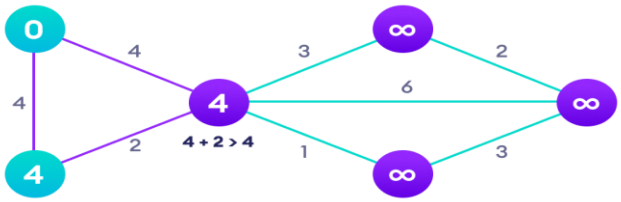
Step: 2

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



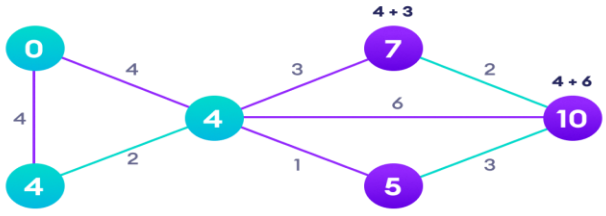
Step: 3

Go to each vertex and update its path length

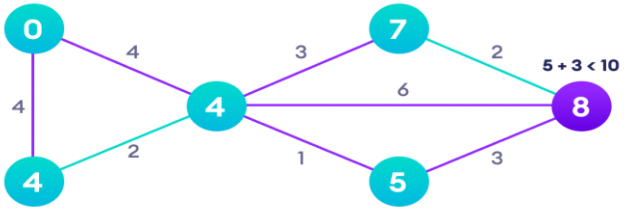


Step: 4

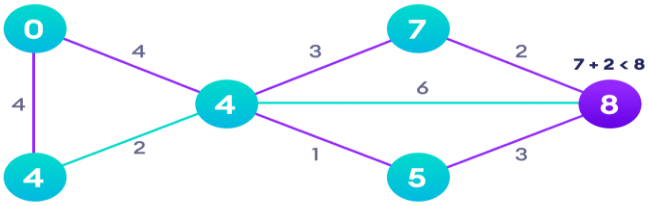
If the path length of the adjacent vertex is lesser than new path length, don't update it



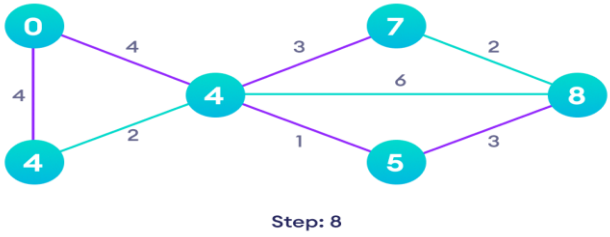
Avoid updating path lengths of already visited vertices



After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Notice how the rightmost vertex has its path length updated twice



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[]
```

Code for Dijkstra's Algorithm

The implementation of Dijkstra's Algorithm in C++ is given below. The complexity of the code can be improved, but the abstractions are convenient to relate the code with the algorithm.

```
# Dijkstra's Algorithm in Python
import sys

# Providing the graph
vertices = [[0, 0, 1, 1, 0, 0, 0],
          [0, 0, 1, 0, 0, 1, 0],
          [1, 1, 0, 1, 1, 0, 0],
          [1, 0, 1, 0, 0, 0, 1],
          [0, 0, 1, 0, 0, 1, 0],
          [0, 1, 0, 0, 1, 0, 1],
          [0, 0, 0, 1, 0, 1, 0]]

edges = [[0, 0, 1, 2, 0, 0, 0],
        [0, 0, 2, 0, 0, 3, 0],
        [1, 2, 0, 1, 3, 0, 0],
        [2, 0, 1, 0, 0, 0, 1],
        [0, 0, 3, 0, 0, 2, 0],
        [0, 3, 0, 0, 2, 0, 1],
        [0, 0, 0, 1, 0, 1, 0]]
```

```
# Find which vertex is to be visited next
def to_be_visited():
    global visited_and_distance
    v = -10
    for index in range(num_of_vertices):
        if visited_and_distance[index][0] == 0 \
            and (v < 0 or visited_and_distance[index][1] <=
                visited_and_distance[v][1]):
            v = index
    return v

num_of_vertices = len(vertices[0])

visited_and_distance = [[0, 0]]
for i in range(num_of_vertices-1):
    visited_and_distance.append([0, sys.maxsize])

for vertex in range(num_of_vertices):

    # Find next vertex to be visited
    to_visit = to_be_visited()
    for neighbor_index in range(num_of_vertices):

        # Updating new distances
        if vertices[to_visit][neighbor_index] == 1 and \
                visited_and_distance[neighbor_index][0] == 0:
            new_distance = visited_and_distance[to_visit][1] \
                + edges[to_visit][neighbor_index]
            if visited_and_distance[neighbor_index][1] > new_distance:
                visited_and_distance[neighbor_index][1] = new_distance

        visited_and_distance[to_visit][0] = 1

i = 0

# Printing the distance
for distance in visited_and_distance:
    print("Distance of ", chr(ord('a') + i),
          " from source vertex: ", distance[1])
    i = i + 1
```

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)

Dijkstra's Algorithm Applications

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map

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.

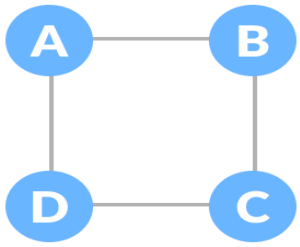
The total number of spanning trees with n vertices that can be created from a complete graph is equal to $\mathbf{n^{(n-2)}}$.

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

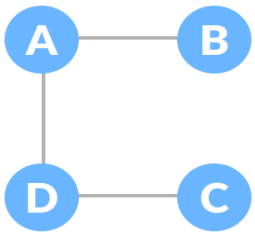
Let's understand the spanning tree with examples below:

Let the original graph be:

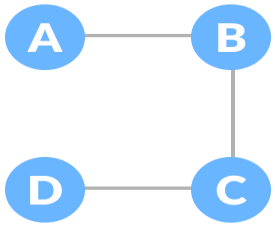


Normal graph

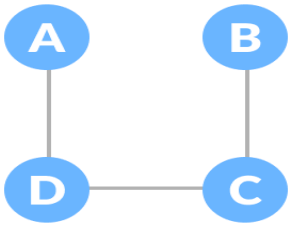
Some of the possible spanning trees that can be created from the above graph are:



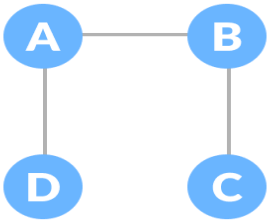
A spanning tree



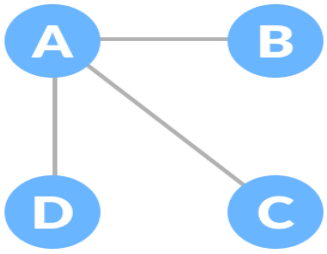
A spanning tree



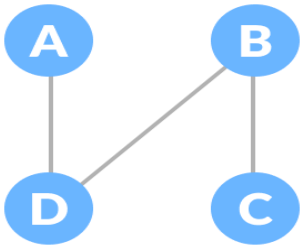
A spanning tree



A spanning tree



A spanning tree



A spanning tree

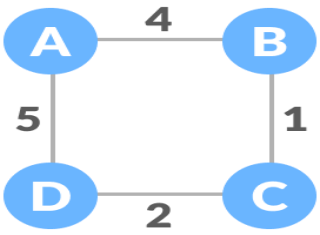
Minimum Spanning Tree

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

Example of a Spanning Tree

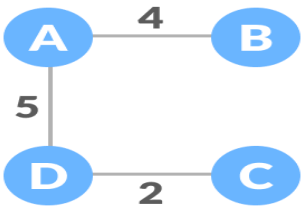
Let's understand the above definition with the help of the example below.

The initial graph is:



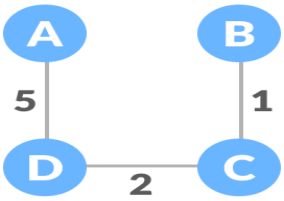
Weighted graph

The possible spanning trees from the above graph are:



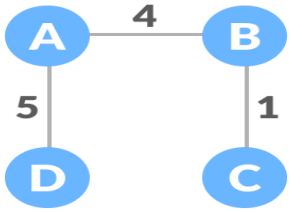
sum = 11

Minimum spanning tree - 1



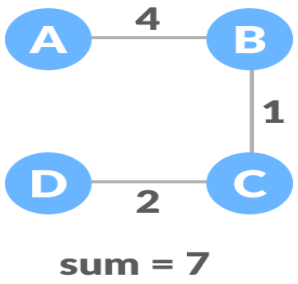
sum = 8

Minimum spanning tree - 2



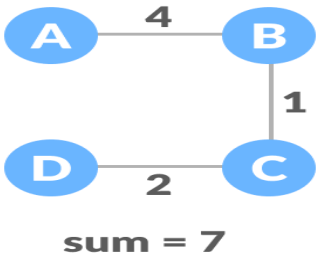
sum = 10

Minimum spanning tree - 3



Minimum spanning tree - 4

The minimum spanning tree from the above spanning trees is:



Minimum spanning tree

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

- Prim's Algorithm
- Kruskal's Algorithm

Spanning Tree Applications

- Computer Network Routing Protocol
- Cluster Analysis
- Civil Network Planning

Minimum Spanning tree Applications

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

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

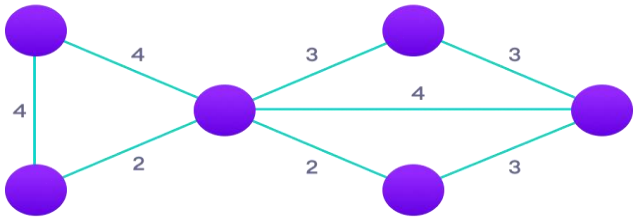
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

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:

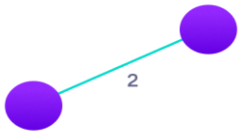
- Sort all the edges from low weight to high
- Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
- Keep adding edges until we reach all vertices.

Example of Kruskal's algorithm



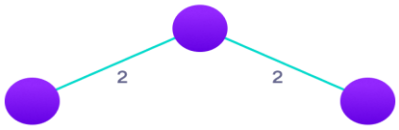
Step: 1

Start with a weighted graph



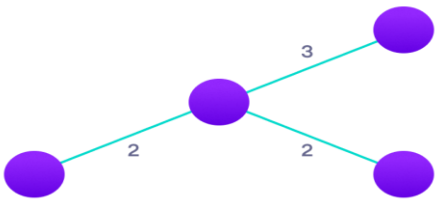
Step: 2

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



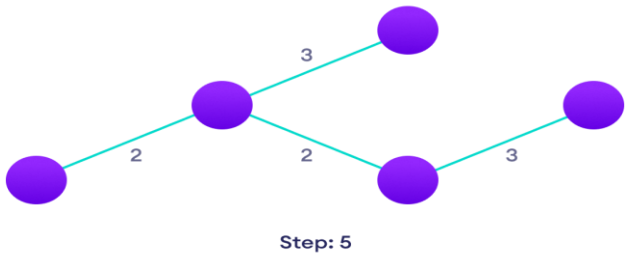
Step: 3

Choose the next shortest edge and add it



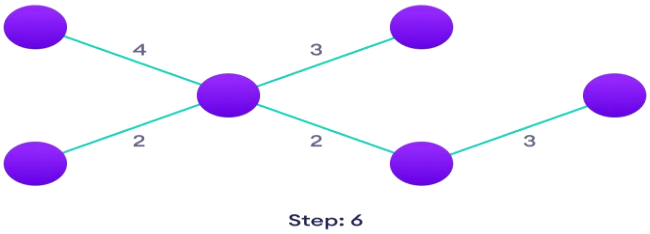
Step: 4

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 in Python

```
class Graph:
   def __init__(self, vertices):
      self.V = vertices
      self.graph = []

   def add_edge(self, u, v, w):
      self.graph.append([u, v, w])
```

```
    # Search function

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def apply_union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    #  Applying Kruskal algorithm
    def kruskal_algo(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print("%d - %d: %d" % (u, v, weight))

g = Graph(6)
g.add_edge(0, 1, 4)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 2)
g.add_edge(1, 0, 4)
```

```
g.add_edge(2, 0, 4)
g.add_edge(2, 1, 2)
g.add_edge(2, 3, 3)
g.add_edge(2, 5, 2)
g.add_edge(2, 4, 4)
g.add_edge(3, 2, 3)
g.add_edge(3, 4, 3)
g.add_edge(4, 2, 4)
g.add_edge(4, 3, 3)
g.add_edge(5, 2, 2)
g.add_edge(5, 4, 3)
g.kruskal_algo()
```

Kruskal's vs Prim's Algorithm

Prim's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from an edge, Prim's algorithm starts from a vertex and keeps adding lowest-weight edges which aren't in the tree, until all vertices have been covered.

Kruskal's Algorithm Complexity

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

Kruskal's Algorithm Applications

- In order to layout electrical wiring
- In computer network (LAN connection)

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

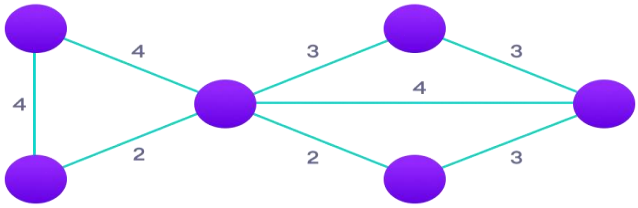
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

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

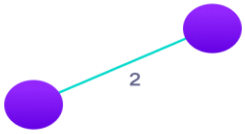


Step: 1

Start with a weighted graph

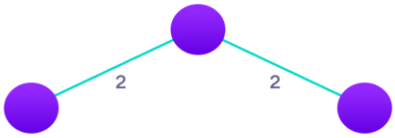
Step: 2

Choose a vertex



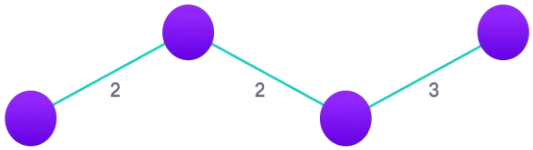
Step: 3

Choose the shortest edge from this vertex and add it



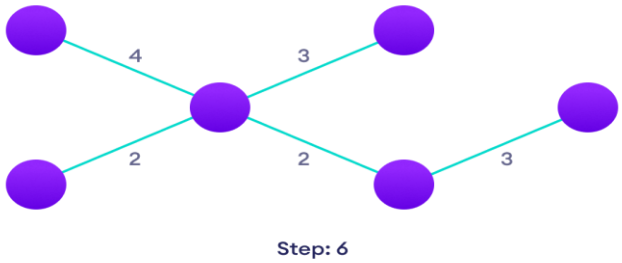
Step: 4

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}
```

Although adjacency matrix representation of graphs is used, this algorithm can also be implemented using Adjacency List to improve its efficiency.

```
# Prim's Algorithm in Python

INF = 9999999
# number of vertices in graph
V = 5
# create a 2d array of size 5x5
# for adjacency matrix to represent graph
G = [[0, 9, 75, 0, 0],
     [9, 0, 95, 19, 42],
     [75, 95, 0, 51, 66],
     [0, 19, 51, 0, 31],
     [0, 42, 66, 31, 0]]
# create a array to track selected vertex
# selected will become true otherwise false
selected = [0, 0, 0, 0, 0]
# set number of edge to 0
no_edge = 0
# the number of edge in minimum spanning tree will be
# always less than(V - 1), where V is number of vertices in
# graph
# choose 0th vertex and make it true
selected[0] = True
```

```
# print for edge and weight
print("Edge : Weight\n")
while (no_edge < V - 1):
    # For every vertex in the set S, find the all adjacent vertices
    #, calculate the distance from the vertex selected at step 1.
    # if the vertex is already in the set S, discard it otherwise
    # choose another vertex nearest to the selected vertex  at step 1.
    minimum = INF
    x = 0
    y = 0
    for i in range(V):
        if selected[i]:
            for j in range(V):
                if ((not selected[j]) and G[i][j]):
                    # not in selected and there is an edge
                    if minimum > G[i][j]:
                        minimum = G[i][j]
                        x = i
                        y = j
    print(str(x) + "-" + str(y) + ":" + str(G[x][y]))
    selected[y] = True
    no_edge += 1
```

Prim's vs Kruskal's Algorithm

Kruskal's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from a vertex, Kruskal's algorithm sorts all the edges from low weight to high and keeps adding the lowest edges, ignoring those edges that create a cycle.

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