Prim's Algorithm for Minimum Spanning Tree (MST)

Last Updated : 05 Dec, 2024

Introduction to Prim's algorithm:

We have discussed Kruskal's algorithm for Minimum Spanning Tree. Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way.

> *The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.*

A group of edges that connects two sets of vertices in a graph is called cut in graph theory. So, at every step of Prim's algorithm, find a cut, pick the minimum weight edge from the cut, and include this vertex in MST Set (the set that contains already included vertices).

How does Prim's Algorithm Work?

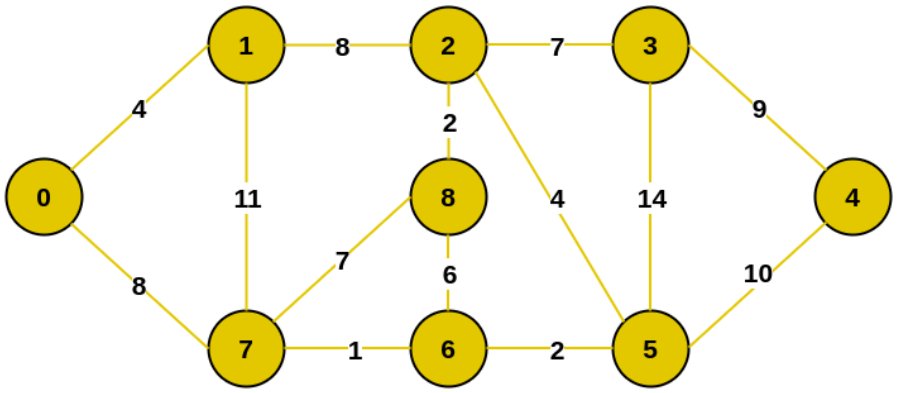
The working of Prim's algorithm can be described by using the following steps:

Step 1: *Determine an arbitrary vertex as the starting vertex of the MST.*
Step 2: *Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).*
Step 3: *Find edges connecting any tree vertex with the fringe vertices.*
Step 4: *Find the minimum among these edges.*
Step 5: *Add the chosen edge to the MST if it does not form any cycle.*
Step 6: *Return the MST and exit*

Note: For determining a cycle, we can divide the vertices into two sets [one set contains the vertices included in MST and the other contains the fringe vertices.]

Illustration of Prim's Algorithm:

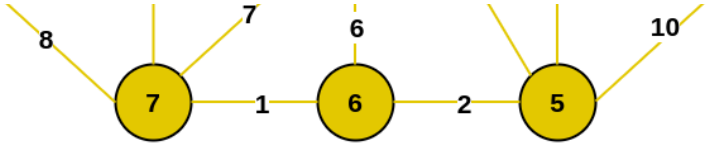
Consider the following graph as an example for which we need to find the Minimum Spanning Tree (MST).



Example of a Graph

Example of a graph

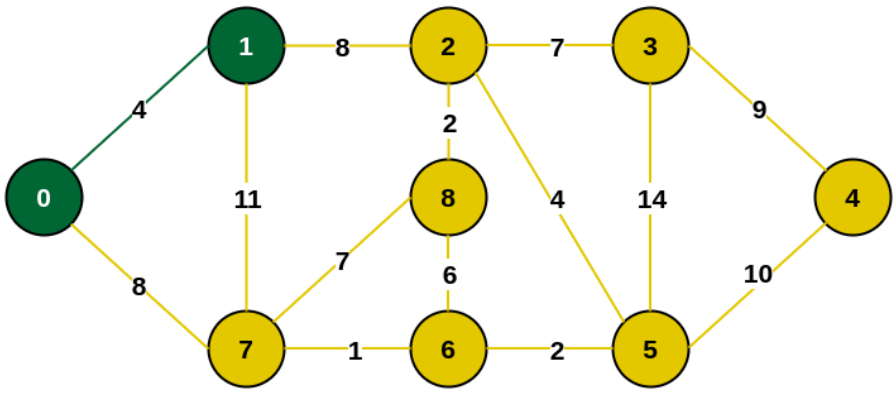
Step 1: *Firstly, we select an arbitrary vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected vertex **0** as the starting vertex.*



Select an arbitrary starting vertex. Here we have selected 0

0 is selected as starting vertex

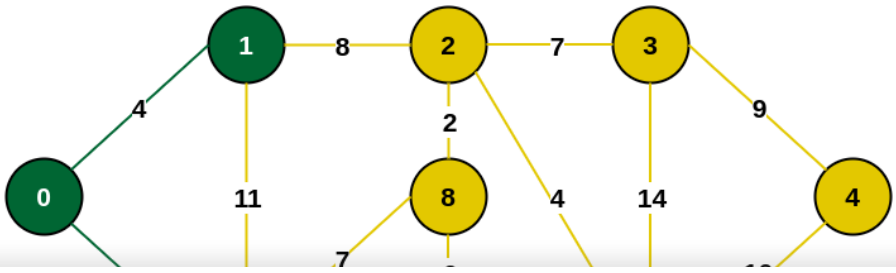
Step 2: *All the edges connecting the incomplete MST and other vertices are the edges {0, 1} and {0, 7}. Between these two the edge with minimum weight is {0, 1}. So include the edge and vertex 1 in the MST.*



Minimum weighted edge from MST to other vertices is 0-1 with weight 4

1 is added to the MST

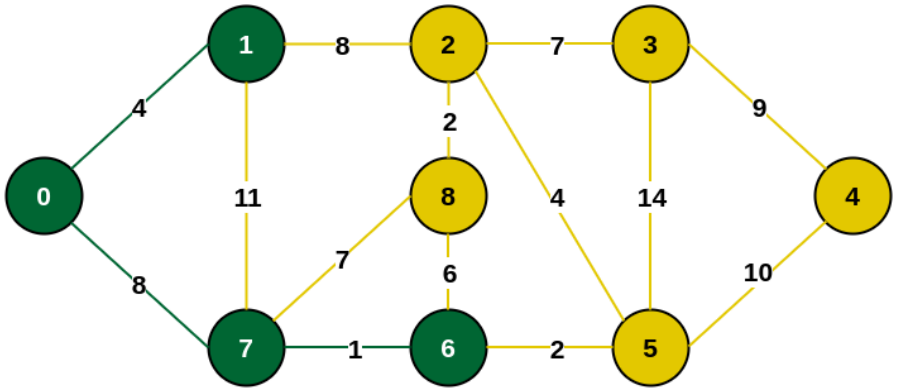
Step 3: *The edges connecting the incomplete MST to other vertices are {0, 7}, {1, 7} and {1, 2}. Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. [We could have also included edge {1, 2} and vertex 2 in the MST].*



Minimum weighted edge from MST to other vertices is 0-7 with weight 8

7 is added in the MST

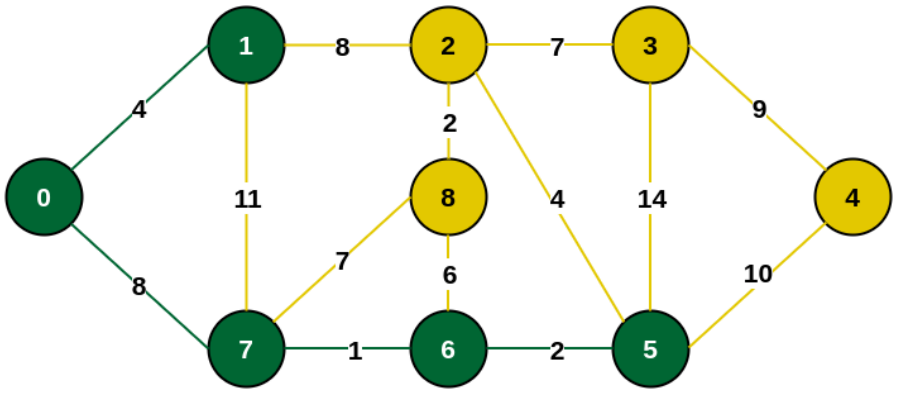
Step 4: *The edges that connect the incomplete MST with the fringe vertices are {1, 2}, {7, 6} and {7, 8}. Add the edge {7, 6} and the vertex 6 in the MST as it has the least weight (i.e., 1).*



Minimum weighted edge from MST to other vertices is 7-6 with weight 1

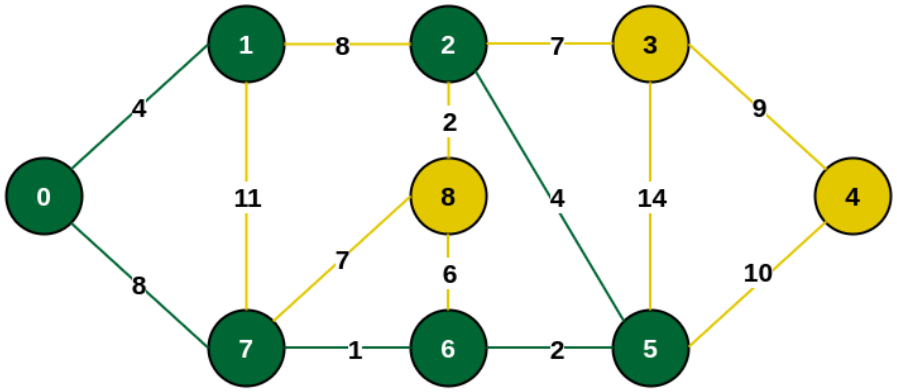
6 is added in the MST

Step 5: *The connecting edges now are {7, 8}, {1, 2}, {6, 8} and {6, 5}. Include edge {6, 5} and vertex 5 in the MST as the edge has the minimum weight (i.e., 2) among them.*



Minimum weighted edge from MST to other vertices is 6-5 with weight 2

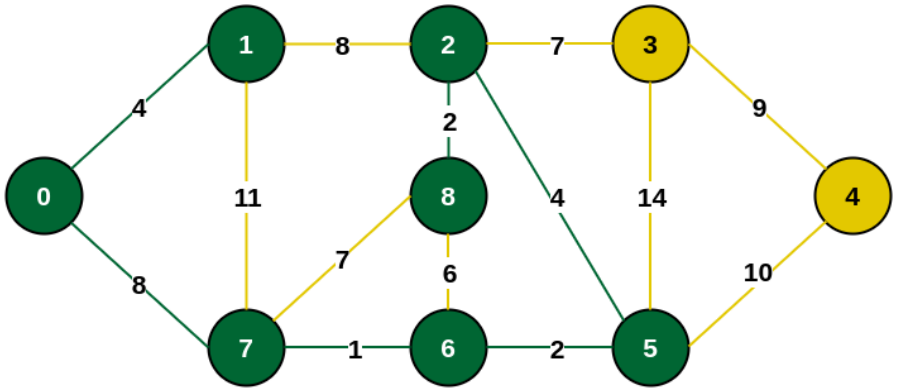
Include vertex 5 in the MST



Minimum weighted edge from MST to other vertices is 5-2 with weight 4

Include vertex 2 in the MST

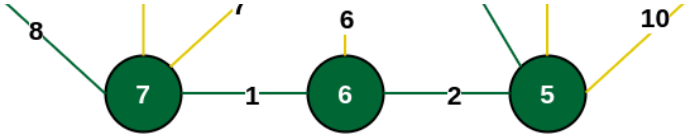
Step 7: *The connecting edges between the incomplete MST and the other edges are {2, 8}, {2, 3}, {5, 3} and {5, 4}. The edge with minimum weight is edge {2, 8} which has weight 2. So include this edge and the vertex 8 in the MST.*



Minimum weighted edge from MST to other vertices is 2-8 with weight 2

Add vertex 8 in the MST

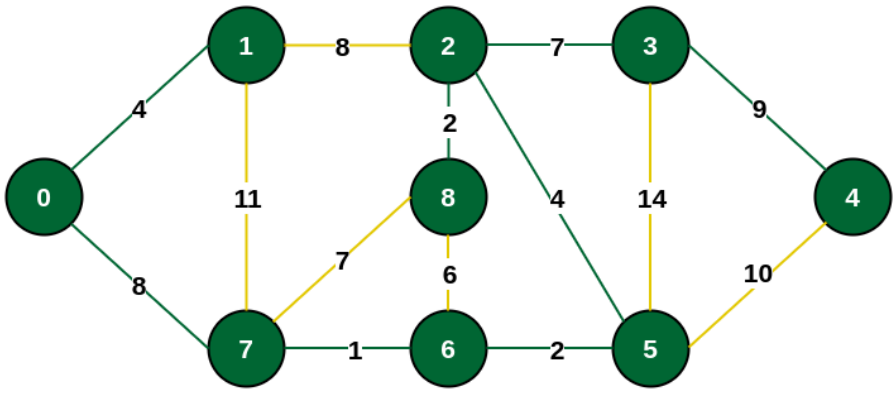
Step 8: *See here that the edges {7, 8} and {2, 3} both have same weight which are minimum. But 7 is already part of MST. So we will consider the edge {2, 3} and include that edge and vertex 3 in the MST.*



Minimum weighted edge from MST to other vertices is 2-3 with weight 7

Include vertex 3 in MST

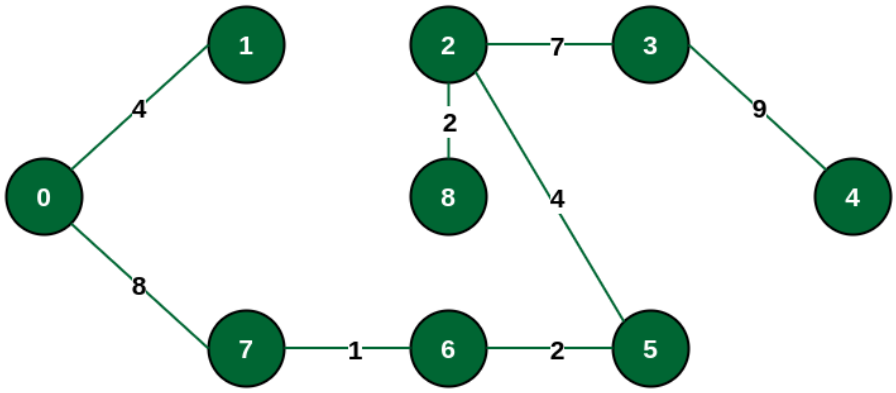
Step 9: *Only the vertex 4 remains to be included. The minimum weighted edge from the incomplete MST to 4 is {3, 4}.*



Minimum weighted edge from MST to other vertices is 3-4 with weight 9

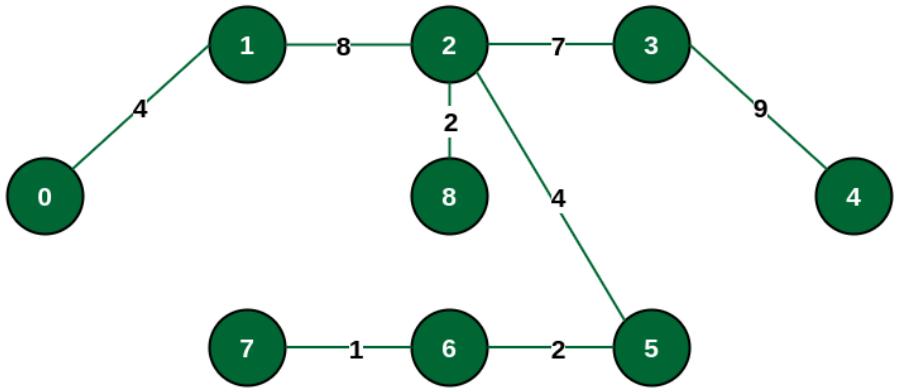
Include vertex 4 in the MST

The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = ***37****.*



The final structure of MST

would look like the following.



Alternative MST structure

Structure of the alternate MST if we had selected edge {1, 2} in the MST

How to implement Prim's Algorithm?

Follow the given steps to utilize the **Prim's Algorithm** mentioned above for finding MST of a graph:

- Create a set **mstSet** that keeps track of vertices already included in MST.
- Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.
- While **mstSet** doesn't include all vertices
    - Pick a vertex **u** that is not there in **mstSet** and has a minimum key value.
    - Include **u** in the **mstSet**.
    - Update the key value of all adjacent vertices of **u**. To update the key values, iterate through all adjacent vertices.
        - For every adjacent vertex **v**, if the weight of edge **u-v** is less than the previous key value of **v**, update the key value as the weight of **u-v**.

cut. The key values are used only for vertices that are not yet included in MST, the key value for these vertices indicates the minimum weight edges connecting them to the set of vertices included in MST.

Below is the implementation of the approach:

C++ C Java Python C# JavaScript

```cpp
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph

#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(vector<int> &key, vector<bool> &mstSet) {

    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(vector<int> &parent, vector<vector<int>>
{
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
```

```
34
35  // Function to construct and print MST for
36  // a graph represented using adjacency
37  // matrix representation
38  void primMST(vector<vector<int>> &graph) {
39
40      // Array to store constructed MST
41      vector<int> parent(V);
42
43      // Key values used to pick minimum weight edge in
44      vector<int> key(V);
45
46      // To represent set of vertices included in MST
47      vector<bool> mstSet(V);
48
49      // Initialize all keys as INFINITE
50      for (int i = 0; i < V; i++)
51          key[i] = INT_MAX, mstSet[i] = false;
52
53      // Always include first 1st vertex in MST.
54      // Make key 0 so that this vertex is picked as fi
55      // vertex.
56      key[0] = 0;
57
58      // First node is always root of MST
59      parent[0] = -1;
60
61      // The MST will have V vertices
62      for (int count = 0; count < V - 1; count++) {
63
64          // Pick the minimum key vertex from the
65          // set of vertices not yet included in MST
66          int u = minKey(key, mstSet);
67
68          // Add the picked vertex to the MST Set
69          mstSet[u] = true;
70
71          // Update key value and parent index of
72          // the adjacent vertices of the picked vertex.
```

```
77                // graph[u][v] is non zero only for adjace
78                // vertices of m mstSet[v] is false for ve
79                // not yet included in MST Update the key
80                // if graph[u][v] is smaller than key[v]
81                if (graph[u][v] && mstSet[v] == false
82                    && graph[u][v] < key[v])
83                    parent[v] = u, key[v] = graph[u][v];
84        }
85
86        // Print the constructed MST
87        printMST(parent, graph);
88    }
89
90    // Driver's code
91    int main() {
92          vector<vector<int>> graph = { { 0, 2, 0, 6, 0 },
93                                        { 2, 0, 3, 8, 5 },
94                                        { 0, 3, 0, 0, 7 },
95                                        { 6, 8, 0, 0, 9 },
96                                        { 0, 5, 7, 9, 0 } };
97
98        // Print the solution
99        primMST(graph);
100
101       return 0;
102   }
103
104   // This code is contributed by rathbhupendra
```

Output

```
Edge     Weight
0 - 1     2
1 - 2     3
0 - 3     6
1 - 4     5
```

Complexity Analysis of Prim's Algorithm:

logV) with the help of a binary heap. In this implementation, we are always considering the spanning tree to start from the root of the graph
Auxiliary Space: O(V)

Optimized Implementation using Adjacency List Representation (of Graph) and Priority Queue

Intuition

1. *We transform the adjacency matrix into adjacency list using **ArrayList<ArrayList<Integer>>**. in Java, list of list in Python and array of vectors in C++.*
2. *Then we create a Pair class to store the vertex and its weight .*
3. *We sort the list on the basis of lowest weight.*
4. *We create priority queue and push the first vertex and its weight in the queue*
5. *Then we just traverse through its edges and store the least weight in a variable called **ans.***
6. *At last after all the vertex we return the ans.*

Implementation

C++ Java Python C# JavaScript

```
#include<bits/stdc++.h>
using namespace std;

// Function to find sum of weights of edges of the
Minimum Spanning Tree.
int spanningTree(int V, int E, vector<vector<int>>
&edges) {

    // Create an adjacency list representation of
```

```
10      // Fill the adjacency list with edges and their
    weights
11      for (int i = 0; i < E; i++) {
12          int u = edges[i][0];
13          int v = edges[i][1];
14          int wt = edges[i][2];
15          adj[u].push_back({v, wt});
16          adj[v].push_back({u, wt});
17      }
18
19      // Create a priority queue to store edges with
    their weights
20      priority_queue<pair<int,int>,
    vector<pair<int,int>>, greater<pair<int,int>>> pq;
21
22      // Create a visited array to keep track of
    visited vertices
23      vector<bool> visited(V, false);
24
25      // Variable to store the result (sum of edge
    weights)
26      int res = 0;
27
28      // Start with vertex 0
29      pq.push({0, 0});
30
31      // Perform Prim's algorithm to find the Minimum
    Spanning Tree
32      while(!pq.empty()){
33          auto p = pq.top();
34          pq.pop();
35
36          int wt = p.first;  // Weight of the edge
37          int u = p.second;  // Vertex connected to
    the edge
38
39          if(visited[u] == true){
40              continue;  // Skip if the vertex is
    already visited
41          }
42
```

```
    visited
45
46          // Explore the adjacent vertices
47          for(auto v : adj[u]){
48              // v[0] represents the vertex and v[1]
    represents the edge weight
49              if(visited[v[0]] == false){
50                  pq.push({v[1], v[0]});  // Add the
    adjacent edge to the priority queue
51              }
52          }
53      }
54
55      return res;  // Return the sum of edge weights
    of the Minimum Spanning Tree
56  }
57
58  int main() {
59      vector<vector<int>> graph = {{0, 1, 5},
60                                   {1, 2, 3},
61                                   {0, 2, 1}};
62
63      cout << spanningTree(3, 3, graph) << endl;
64
65      return 0;
66  }
```

Output

```
4
```

Complexity Analysis of Prim's Algorithm:

Time Complexity: O(E*log(E)) where E is the number of edges

Auxiliary Space: O(V^2) where V is the number of vertex

Prim's algorithm for finding the minimum spanning tree (MST):

graph.
2. It has a time complexity of O(E log V) using a binary heap or Fibonacci heap, where E is the number of edges and V is the number of vertices.
3. It is a relatively simple algorithm to understand and implement compared to some other MST algorithms.

Disadvantages:

1. Like Kruskal's algorithm, Prim's algorithm can be slow on dense graphs with many edges, as it requires iterating over all edges at least once.
2. Prim's algorithm relies on a priority queue, which can take up extra memory and slow down the algorithm on very large graphs.
3. The choice of starting node can affect the MST output, which may not be desirable in some applications.

Other Implementations of Prim's Algorithm:

Given below are some other implementations of Prim's Algorithm

- Prim's Algorithm for Adjacency Matrix Representation – In this article we have discussed the method of implementing Prim's Algorithm if the graph is represented by an adjacency matrix.
- Prim's Algorithm for Adjacency List Representation – In this article Prim's Algorithm implementation is described for graphs represented by an adjacency list.
- Prim's Algorithm using Priority Queue: In this article, we have discussed a time-efficient approach to implement Prim's algorithm.