This is taken from geeksforgeeks.org . This is not my original copy of my project. I do not have access to my projects now. This is overview of the Project.

Analysis & Implementation of Graph Searching Algorithms using C

Key Technologies Used: C programming language, Data Structures, Graph Searching Algorithms

Description.

- **Shortest Spanning Tree Algorithms**

 PRIMS and Kruskal's Algorithm : Designed and implemented both algorithms in C language. Compared the actual time of execution and space used during execution with theoretical Asymptotic time complexity and space complexity.

- **Single Source Shortest path Algorithms**

 Dijkstra's algorithm and The Bellman-Ford algorithm: Designed and implemented both algorithms in C language. Compared the actual time of execution and space used during execution with theoretical Asymptotic time complexity and space complexity.

PRIMS Algorithm

1) Create a set *mstSet* that keeps track of vertices already included in MST.

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

3) While mstSet doesn’t include all vertices

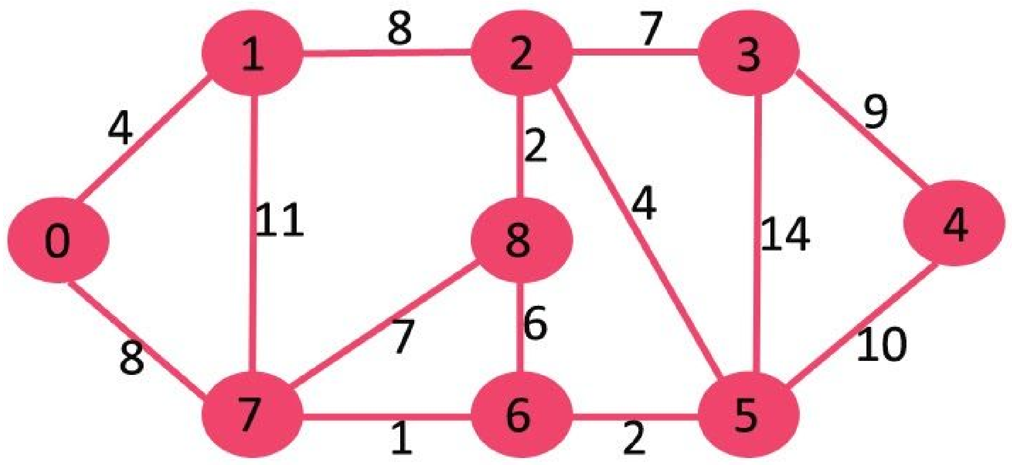
….**a)** Pick a vertex *u* which is not there in *mstSet* and has minimum key value.

….**b)** Include *u* to mstSet.

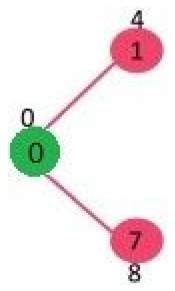
….**c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u*-*v* is less than the previous key value of *v*, update the key value as weight of *u*-*v*

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

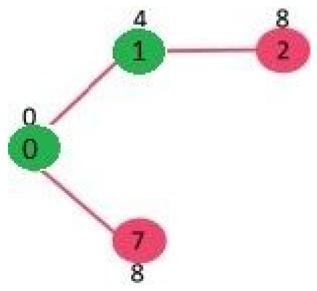
Let us understand with the following example:



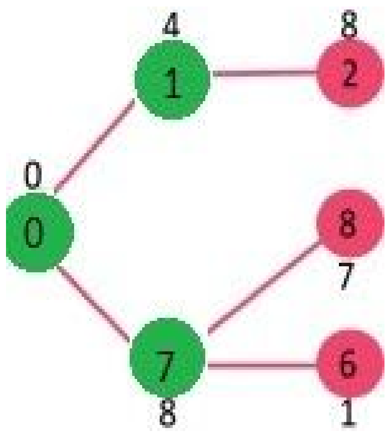
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



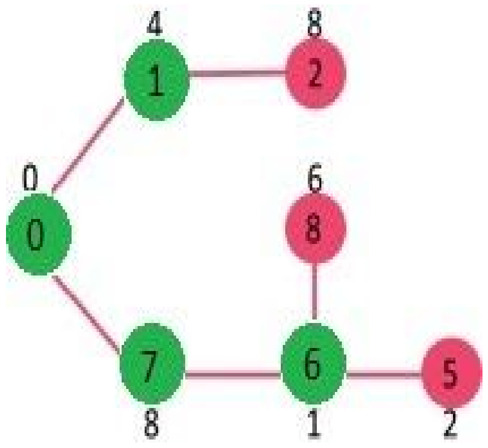
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



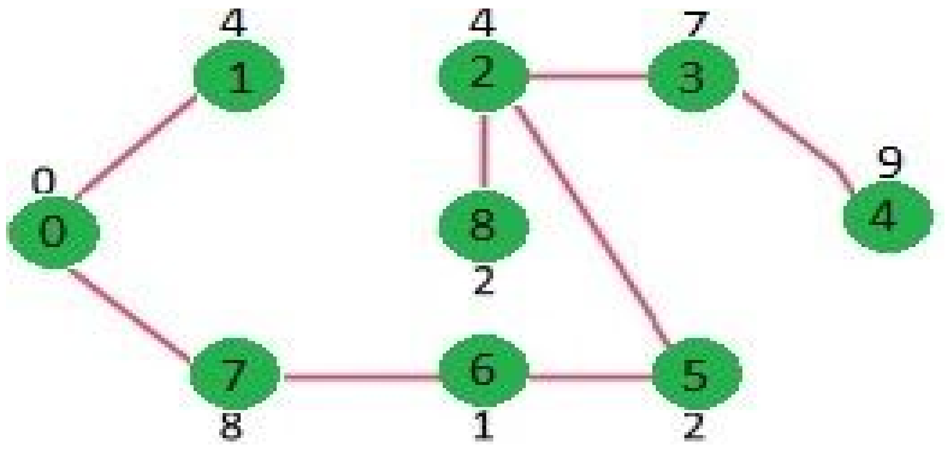
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).



Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



Finally, we get the following graph.



How to implement the above algorithm?

We use a boolean array mstSet[] to represent the set of vertices included in MST. If a value mstSet[v] is true, then vertex v is included in MST, otherwise not. Array key[] is used to store key values of all vertices. Another array parent[] to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

Code. This code is taken from geeksforgeeks.org

```
#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(int key[], 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(int parent[], int graph[V][V])
{
	cout<<"Edge \tWeight\n";

	for (int i = 1; i < V; i++)

		cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";

}



// Function to construct and print MST for

// a graph represented using adjacency

// matrix representation

void primMST(int graph[V][V])

{

	// Array to store constructed MST

	int parent[V];



	// Key values used to pick minimum weight edge in cut

	int key[V];



	// To represent set of vertices included in MST

	bool mstSet[V];
```

```
// Initialize all keys as INFINITE

for (int i = 0; i < V; i++)

        key[i] = INT_MAX, mstSet[i] = false;


// Always include first 1st vertex in MST.

// Make key 0 so that this vertex is picked as first vertex.

key[0] = 0;

parent[0] = -1; // First node is always root of MST


// The MST will have V vertices

for (int count = 0; count < V - 1; count++)

{

        // Pick the minimum key vertex from the

        // set of vertices not yet included in MST

        int u = minKey(key, mstSet);


        // Add the picked vertex to the MST Set

        mstSet[u] = true;
        // Update key value and parent index of

        // the adjacent vertices of the picked vertex.

        // Consider only those vertices which are not

        // yet included in MST

        for (int v = 0; v < V; v++)


            // graph[u][v] is non zero only for adjacent vertices of m

            // mstSet[v] is false for vertices not yet included in MST

            // Update the key only if graph[u][v] is smaller than key[v]

            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

                parent[v] = u, key[v] = graph[u][v];

    }


    // print the constructed MST

    printMST(parent, graph);

}


// Driver code

int main()
```

```
{

        /* Let us create the following graph

                2 3

        (0)--(1)--(2)

        | / \ |

        6| 8/ \5 |7

        | / \ |

        (3)-------(4)

                    9        */

        int graph[V][V] = { { 0, 2, 0, 6, 0 },

                                        { 2, 0, 3, 8, 5 },

                                        { 0, 3, 0, 0, 7 },

                                        { 6, 8, 0, 0, 9 },

                                        { 0, 5, 7, 9, 0 } };



        // Print the solution

        primMST(graph);



        return 0;
}
```

Output:

```
Edge   Weight

0 - 1    2

1 - 2    3

0 - 3    6

1 - 4    5
```

Time Complexity of the above program is O(V^2). If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to O(E log V) with the help of binary heap.

Kruskal's Minimum Spanning Tree Algorithm

Below are the steps for finding MST using Kruskal's algorithm

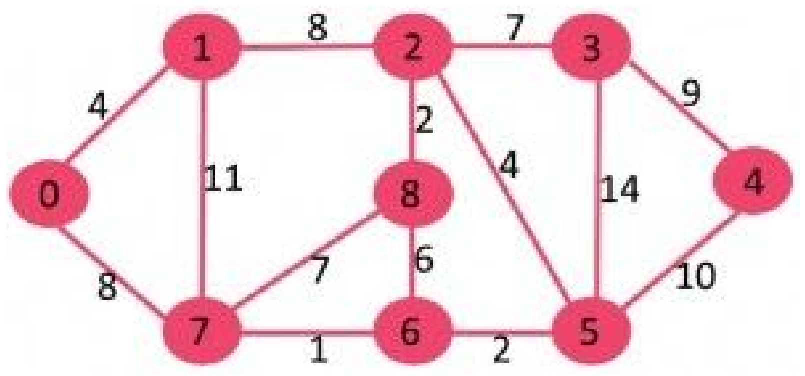
1. *Sort all the edges in non-decreasing order of their weight.*

2. *Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*

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

The step#2 uses Union-Find algorithm to detect cycle. So we recommend to read following post as a prerequisite.

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



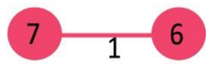
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

```
After sorting:
Weight    Src     Dest
1          7       6
2          8       2
2          6       5
4          0       1
4          2       5
6          8       6
7          2       3
7          7       8
8          0       7
8          1       2
9          3       4
10         5       4
```

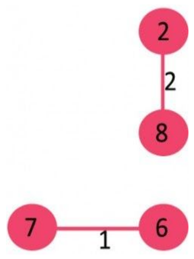
11	1	7
14	3	5

Now pick all edges one by one from sorted list of edges

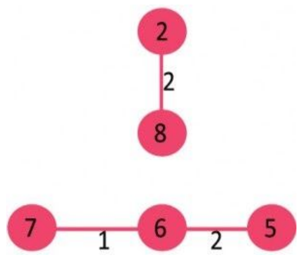
1. *Pick edge 7-6:* No cycle is formed, include it.



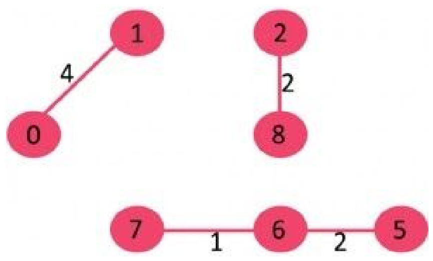
2. *Pick edge 8-2:* No cycle is formed, include it.



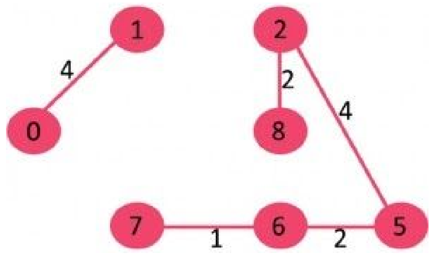
3. *Pick edge 6-5:* No cycle is formed, include it.



4. *Pick edge 0-1:* No cycle is formed, include it.

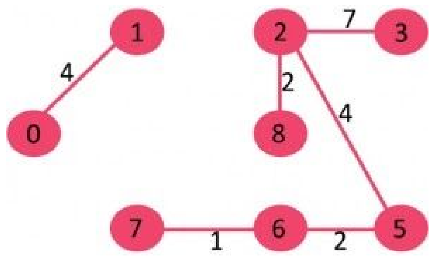


5. *Pick edge 2-5:* No cycle is formed, include it.



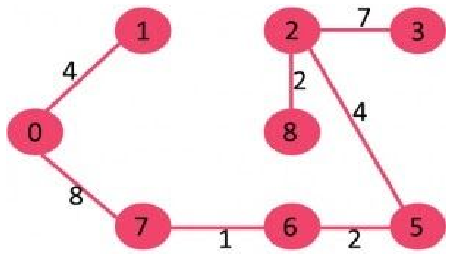
6. *Pick edge 8-6:* Since including this edge results in cycle, discard it.

7. *Pick edge 2-3:* No cycle is formed, include it.



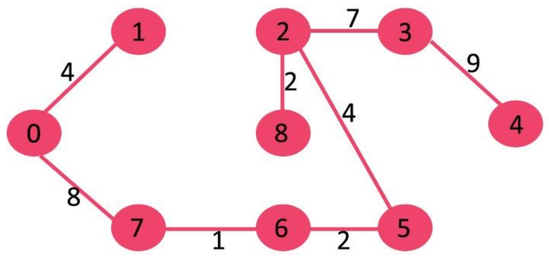
8. *Pick edge 7-8:* Since including this edge results in cycle, discard it.

9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in cycle, discard it.

11. *Pick edge 3-4:* No cycle is formed, include it.



Since the number of edges included equals (V – 1), the algorithm stops here.

Code. This code is taken from geeksforgeeks.org

```
// C++ program for Kruskal's algorithm to find Minimum Spanning Tree

// of a given connected, undirected and weighted graph

#include <bits/stdc++.h>

using namespace std;



// a structure to represent a weighted edge in graph

class Edge

{

        public:

        int src, dest, weight;

};
```

```
// a structure to represent a connected, undirected

// and weighted graph

class Graph

{

        public:

        // V-> Number of vertices, E-> Number of edges

        int V, E;


        // graph is represented as an array of edges.

        // Since the graph is undirected, the edge

        // from src to dest is also edge from dest

        // to src. Both are counted as 1 edge here.

        Edge* edge;

};


// Creates a graph with V vertices and E edges

Graph* createGraph(int V, int E)

{

        Graph* graph = new Graph;
```

```
    graph->V = V;

    graph->E = E;


    graph->edge = new Edge[E];


    return graph;

}


// A structure to represent a subset for union-find

class subset

{

    public:

    int parent;

    int rank;

};


// A utility function to find set of an element i

// (uses path compression technique)

int find(subset subsets[], int i)
```

```
{

        // find root and make root as parent of i

        // (path compression)

        if (subsets[i].parent != i)

                subsets[i].parent = find(subsets, subsets[i].parent);


        return subsets[i].parent;

}


// A function that does union of two sets of x and y

// (uses union by rank)

void Union(subset subsets[], int x, int y)

{

        int xroot = find(subsets, x);

        int yroot = find(subsets, y);


        // Attach smaller rank tree under root of high

        // rank tree (Union by Rank)

        if (subsets[xroot].rank < subsets[yroot].rank)
```

```
            subsets[xroot].parent = yroot;

        else if (subsets[xroot].rank > subsets[yroot].rank)

            subsets[yroot].parent = xroot;



        // If ranks are same, then make one as root and

        // increment its rank by one

        else

        {

            subsets[yroot].parent = xroot;

            subsets[xroot].rank++;

        }

}



// Compare two edges according to their weights.

// Used in qsort() for sorting an array of edges

int myComp(const void* a, const void* b)

{

        Edge* a1 = (Edge*)a;

        Edge* b1 = (Edge*)b;
```

```
    return a1->weight > b1->weight;
}


// The main function to construct MST using Kruskal's algorithm
void KruskalMST(Graph* graph)
{
    int V = graph->V;
    Edge result[V]; // Tnis will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges


    // Step 1: Sort all the edges in non-decreasing
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);


    // Allocate memory for creating V ssubsets
    subset *subsets = new subset[( V * sizeof(subset) )];
```

```
// Create V subsets with single elements

for (int v = 0; v < V; ++v)

{

        subsets[v].parent = v;

        subsets[v].rank = 0;

}


// Number of edges to be taken is equal to V-1

while (e < V - 1 && i < graph->E)

{

        // Step 2: Pick the smallest edge. And increment

        // the index for next iteration

        Edge next_edge = graph->edge[i++];


        int x = find(subsets, next_edge.src);

        int y = find(subsets, next_edge.dest);


        // If including this edge does't cause cycle,
            // include it in result and increment the index

            // of result for next edge

            if (x != y)

            {

                  result[e++] = next_edge;

                  Union(subsets, x, y);

            }

            // Else discard the next_edge

      }



      // print the contents of result[] to display the

      // built MST

      cout<<"Following are the edges in the constructed MST\n";

      for (i = 0; i < e; ++i)

            cout<<result[i].src<<" -- "<<result[i].dest<<" == "<<result[i].weight<<endl;

      return;

}



// Driver code
```

```
int main()

{

        /* Let us create following weighted graph

                        10

                0--------1

                | \ |

        6| 5\ |15

                | \ |

                2--------3

                        4 */

        int V = 4; // Number of vertices in graph

        int E = 5; // Number of edges in graph

        Graph* graph = createGraph(V, E);




        // add edge 0-1

        graph->edge[0].src = 0;

        graph->edge[0].dest = 1;

        graph->edge[0].weight = 10;
```

```
// add edge 0-2

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 6;


// add edge 0-3

graph->edge[2].src = 0;

graph->edge[2].dest = 3;

graph->edge[2].weight = 5;


// add edge 1-3

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 15;


// add edge 2-3

graph->edge[4].src = 2;

graph->edge[4].dest = 3;
```

```
        graph->edge[4].weight = 4;

        KruskalMST(graph);

        return 0;
}
```

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

Time Complexity: O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be atmost O(V2), so O(LogV) are O(LogE) same. Therefore, overall time complexity is O(ElogE) or O(ElogV)

References:

http://www.ics.uci.edu/~eppstein/161/960206.html

http://en.wikipedia.org/wiki/Minimum_spanning_tree