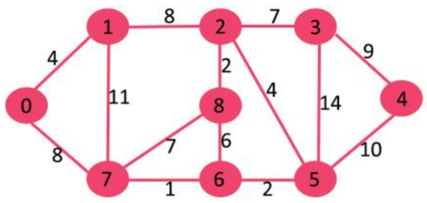
A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted directed or undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. It is a spanning tree whose sum of edge weights is as small as possible.

Kruskal's Minimum Spanning Tree using STL in C++

Given an undirected, connected and weighted graph, find **M**inimum **S**panning **T**ree (MST) of the graph using Kruskal's algorithm.



```
Input :   Graph as an array of edges
Output :  Edges of MST are
          6 - 7
          2 - 8
          5 - 6
          0 - 1
          2 - 5
          2 - 3
          0 - 7
          3 - 4

          Weight of MST is 37

Note: There are two possible MSTs, the other MST includes 1-2 in place of 0-7.
```

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.

```
// 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;

// Creating shortcut for an integer pair
typedef pair<int, int> ii;

// Structure to represent a graph
```

A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted directed or undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. It is a spanning tree whose sum of edge weights is as small as possible.

```
struct Graph
{
	int V, E;
	vector< pair<int, ii> > edges;

	// Constructor
	Graph(int V, int E)
	{
		this->V = V;
		this->E = E;
	}

	// Utility function to add an edge
	void addEdge(int u, int v, int w)
	{
		edges.push_back({w, {u, v}});
	}

	// Function to find MST using Kruskal's
	// MST algorithm
	int kruskalMST();
};

// To represent Disjoint Sets
struct DisjointSets
{
	int *parent, *rnk;
	int n;

	// Constructor.
	DisjointSets(int n)
	{
		// Allocate memory
		this->n = n;
		parent = new int[n+1];
		rnk = new int[n+1];
		// Initially, all vertices are in
		// different sets and have rank 1.
		for (int i = 0; i <= n; i++)
		{
			rnk[i] = 1;
			//every element is parent of itself
			parent[i] = i;
		}
	}
	// Find the parent of a node 'u'
	// Path Compression
	int root(int u)
	{
		while(u != parent[u])
```

A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted directed or undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. It is a spanning tree whose sum of edge weights is as small as possible.

```
            {
                  parent[u] = parent[parent[u]];
                  u = parent[u];
            }
            return u;
      }

      // Union by rank
      void union_(int x, int y)
      {
            x = root(x), y = root(y);
            /* Make tree with smaller height
            a subtree of the other tree */
            if (rnk[x] > rnk[y])
                  parent[y] = x, rnk[x] += rnk[y];
            else
                  parent[x] = y, rnk[y] += rnk[x];
      }
};

/* Functions returns weight of the MST*/

int Graph::kruskalMST()
{
      int mst_wt = 0; // Initialize result

      // Sort edges in increasing order on basis of cost
      sort(edges.begin(), edges.end());

      // Create disjoint sets
      DisjointSets ds(V);

      // Iterate through all sorted edges
      vector< pair<int, ii> >::iterator it;
      for (it=edges.begin(); it!=edges.end(); it++)
      {
            int u = it->second.first;
            int v = it->second.second;

            int set_u = ds.root(u);
            int set_v = ds.root(v);

            // Check if the selected edge is creating
            // a cycle or not (Cycle is created if u
            // and v belong to same set)
            if (set_u != set_v)
            {
                  // Current edge will be in the MST
                  // so print it
                  cout << u << " - " << v << endl;
```

A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted directed or undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. It is a spanning tree whose sum of edge weights is as small as possible.

```
                  // Update MST weight
                  mst_wt += it->first;

                  // Merge two sets
                  ds.union_(set_u,set_v);
            }
      }

      return mst_wt;
}

// Driver program to test above functions
int main()
{
      /* Let us create above shown weighted
      and undirected graph */
      int V = 9, E = 14;
      Graph g(V, E);

      // making above shown graph
      g.addEdge(0, 1, 4);
      g.addEdge(0, 7, 8);
      g.addEdge(1, 2, 8);
      g.addEdge(1, 7, 11);
      g.addEdge(2, 3, 7);
      g.addEdge(2, 8, 2);
      g.addEdge(2, 5, 4);
      g.addEdge(3, 4, 9);
      g.addEdge(3, 5, 14);
      g.addEdge(4, 5, 10);
      g.addEdge(5, 6, 2);
      g.addEdge(6, 7, 1);
      g.addEdge(6, 8, 6);
      g.addEdge(7, 8, 7);

      cout << "Edges of MST are \n";
      int mst_wt = g.kruskalMST();

      cout << "\nWeight of MST is " << mst_wt;

      return 0;
}
```

Prim’s algorithm using priority_queue in STL

1) Initialize keys of all vertices as infinite and parent of every vertex as -1.

A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted directed or undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. It is a spanning tree whose sum of edge weights is as small as possible.

2) Create an empty priority_queue pq. Every item of pq is a pair (weight, vertex). Weight (or key) is used used as first item of pair as first item is by default used to compare two pairs.

3) Initialize all vertices as not part of MST yet. We use boolean array inMST[] for this purpose. This array is required to make sure that an already considered vertex is not included in pq again. This is where Prim's implementation differs from Dijkstra. In Dijkstra's algorithm, we didn't need this array as distances always increase. We require this array here because key value of a processed vertex may decrease if not checked.

4) Insert source vertex into pq and make its key as 0.

```
5) While either pq doesn't become empty
a) Extract minimum key vertex from pq. Let the extracted vertex be u.
b) Include u in MST using inMST[u] = true.
c) Loop through all adjacent of u and do
   following for every vertex v.
        // If weight of edge (u,v) is smaller than
        // key of v and v is not already in MST
        If inMST[v] = false && key[v] > weight(u, v)

           (i) Update key of v, i.e., do
                key[v] = weight(u, v)
           (ii) Insert v into the pq
           (iv) parent[v] = u
6) Print MST edges using parent array.
```

```
#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f

// iPair ==> Integer Pair
typedef pair<int, int> iPair;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
	int V; // No. of vertices

	// In a weighted graph, we need to store vertex
	// and weight pair for every edge
	list< pair<int, int> > *adj;

public:
	Graph(int V); // Constructor

	// function to add an edge to graph
	void addEdge(int u, int v, int w);

	// Print MST using Prim's algorithm
	void primMST();
```

A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted directed or undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. It is a spanning tree whose sum of edge weights is as small as possible.

```
};

// Allocates memory for adjacency list
Graph::Graph(int V)
{
	this->V = V;
	adj = new list<iPair> [V];
}

void Graph::addEdge(int u, int v, int w)
{
	adj[u].push_back(make_pair(v, w));
	adj[v].push_back(make_pair(u, w));
}

// Prints shortest paths from src to all other vertices
void Graph::primMST()
{
	// Create a priority queue to store vertices that
	// are being preinMST. This is weird syntax in C++.
	// Refer below link for details of this syntax
	// http://geeksquiz.com/implement-min-heap-using-stl/
	priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

	int src = 0; // Taking vertex 0 as source

	// Create a vector for keys and initialize all
	// keys as infinite (INF)
	vector<int> key(V, INF);

	// To store parent array which in turn store MST
	vector<int> parent(V, -1);

	// To keep track of vertices included in MST
	vector<bool> inMST(V, false);

	// Insert source itself in priority queue and initialize
	// its key as 0.
	pq.push(make_pair(0, src));
	key[src] = 0;

	/* Looping till priority queue becomes empty */
	while (!pq.empty())
	{
		// The first vertex in pair is the minimum key
		// vertex, extract it from priority queue.
		// vertex label is stored in second of pair (it
		// has to be done this way to keep the vertices
		// sorted key (key must be first item
		// in pair)
```

A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted directed or undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. It is a spanning tree whose sum of edge weights is as small as possible.

```
            int u = pq.top().second;
            pq.pop();

            //Different key values for same vertex may exist in the priority queue.
            //The one with the least key value is always processed first.
            //Therefore, ignore the rest.
            if(inMST[u] == true){
                  continue;
            }
            inMST[u] = true; // Include vertex in MST
            // 'i' is used to get all adjacent vertices of a vertex
            list< pair<int, int> >::iterator i;
            for (i = adj[u].begin(); i != adj[u].end(); ++i)
            {
                  // Get vertex label and weight of current adjacent
                  // of u.
                  int v = (*i).first;
                  int weight = (*i).second;
                  // If v is not in MST and weight of (u,v) is smaller
                  // than current key of v
                  if (inMST[v] == false && key[v] > weight)
                  {
                        // Updating key of v
                        key[v] = weight;
                        pq.push(make_pair(key[v], v));
                        parent[v] = u;
                  }
            }
      }
      // Print edges of MST using parent array
      for (int i = 1; i < V; ++i)
            printf("%d - %d\n", parent[i], i);
}

// Driver program to test methods of graph class
int main()
{
      // create the graph given in above fugure
      int V = 9;
      Graph g(V);

      // making above shown graph
      g.addEdge(0, 1, 4);
      g.addEdge(0, 7, 8);
      g.addEdge(1, 2, 8);
      g.addEdge(1, 7, 11);
      g.addEdge(2, 3, 7);
      g.addEdge(2, 8, 2);
      g.addEdge(2, 5, 4);
      g.addEdge(3, 4, 9);
```

Commented [L1]: This is a greedy approach and works with all weights.
if key[v] > weight && inMST[v]==true it means this edge must have been used to connect to 'u' by 'v' earlier or there was a better edge to connect to 'u'.

The inMST[v] array is required because it might happen that the algorithm falls into a graph cycle.

A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted directed or undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. It is a spanning tree whose sum of edge weights is as small as possible.

```
	g.addEdge(3, 5, 14);
	g.addEdge(4, 5, 10);
	g.addEdge(5, 6, 2);
	g.addEdge(6, 7, 1);
	g.addEdge(6, 8, 6);
	g.addEdge(7, 8, 7);

	g.primMST();

	return 0;
}
```

Dijkstra calculates the shortest path tree from one node whereas Prim/Kruskal calculates the minimum spanning tree between all nodes. The concept of an MST is **not defined for directed graphs -** the connections have to be undirected. For a directed graph you'll be looking to find a minimum cost aborescence, which can't be done using Prim/Kruskal.