Listing 1: Kruskal-Algorithmus

```
#include "graph.h"
#include <numeric>

//Class defining the Disjoint Set data structure using rank by union and path
    compression.
class DisjointSet {
public:
        DisjointSet(int n) : rank(n,0), parent(n) {
                std::iota(parent.begin(), parent.end(), 0); //initializing array
                    with 0...n-1
        }

        void Union(int x,int y){
                Link(FindSet(x),FindSet(y));
        }
        void Link(int x, int y){
                if(rank[x] > rank[y]){
                        parent[y]=x;
                }
                else{
                        parent[x]=y;
                        if(rank[x] == rank[y]){
                                rank[y]++;
                        }
                }
        }
        int FindSet(int x){
                if(parent[x] != x){
                        parent[x] = FindSet(parent[x]);
                }
                return parent[x];
        }
private:
        std::vector<int> parent;
        std::vector<int> rank;

};

// Defining Struct for edges. We need a compact representation for edges, which
    our Graph Class doesn't support.
struct Edge{
        Edge(Graph::NodeId start ,Graph::NodeId end, double weight_) : startNode
            (start), endNode(end),weight(weight_) {}
        Graph::NodeId startNode;
        Graph::NodeId endNode;
        double weight;
};

// Defining Order on edges.
bool operator<(const Edge & a, const Edge & b)
{
        return a.weight < b.weight;
}

//Kruskals Algorithm
Graph kruskal(const Graph & g){
        Graph tree(g.num_nodes(), Graph::undirected);
        DisjointSet branching(g.num_nodes()); //Initializing DisjointSet data
            structure.
```

```
        std::vector<Edge> edges;
        // inserting edges into array.
        for(auto i = 0; i<g.num_nodes(); i++){
                for(auto neighbor: g.get_node(i).adjacent_nodes()){
                        edges.push_back(Edge(i,neighbor.id(),neighbor.edge_weight()));
                }
        }
        std::sort(edges.begin(), edges.end()); // sorting array of edges using std::sort.
        //adding cheapest nice edge.
        for(int i=0; i<edges.size();i++){
                int root_start = branching.FindSet(edges[i].startNode);
                int root_end = branching.FindSet(edges[i].endNode);
                if(root_start != root_end ){
                        branching.Link(root_start,root_end);
                        tree.add_edge1(edges[i].startNode,edges[i].endNode,edges[i].weight);
                }
        }
        return tree;

}

//counting number of edges and weight of graph.
void graph_weight(double *sum,int *num_edges, const Graph & g){
        for(auto i = 0; i<g.num_nodes(); i++){
                for(auto neighbor: g.get_node(i).adjacent_nodes()){
                        (*num_edges)++;
                        (*sum) += neighbor.edge_weight();
                }
        }
}


int main(int argc, char* argv[])
{
    if (argc > 1) {
        Graph g(argv[1], Graph::undirected);
        Graph mst = kruskal(g);
        double sum;
        int num_edges;
        graph_weight(&sum,&num_edges, mst);
        if(num_edges < g.num_nodes() -1) // checking whether graph is connected.
        {
                std::cout << "The graph is unconnected." << std::endl;
        }
        else{
                mst.print();
        }
    }


}
```