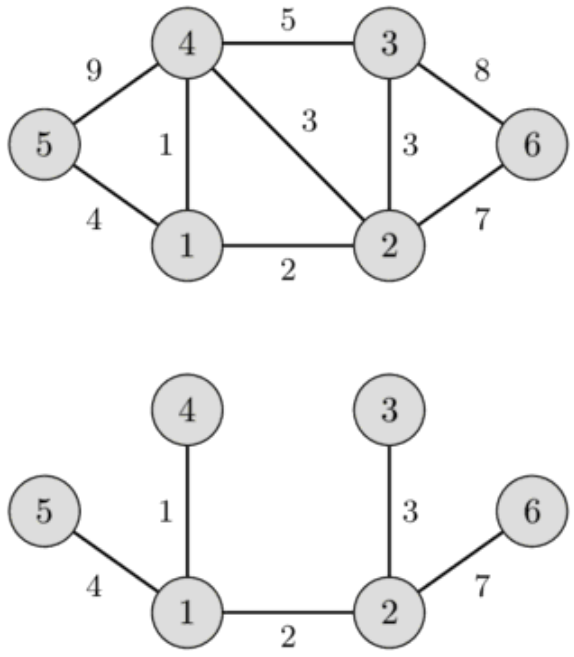
Last update: December 20, 2022 Translated From: e-maxx.ru

Minimum spanning tree - Prim's algorithm

Given a weighted, undirected graph G with n vertices and m edges. You want to find a spanning tree of this graph which connects all vertices and has the least weight (i.e. the sum of weights of edges is minimal). A spanning tree is a set of edges such that any vertex can reach any other by exactly one simple path. The spanning tree with the least weight is called a minimum spanning tree.

In the left image you can see a weighted undirected graph, and in the right image you can see the corresponding minimum spanning tree.



It is easy to see that any spanning tree will necessarily contain $n - 1$ edges.

This problem appears quite naturally in a lot of problems. For instance in the following problem: there are n cities and for each pair of cities we are given the cost to build a road between them (or we know that is physically impossible to build a road between them). We have to build roads, such that we can get from each city to every other city, and the cost for building all roads is minimal.

Prim's Algorithm

This algorithm was originally discovered by the Czech mathematician Vojtěch Jarník in 1930. However this algorithm is mostly known as Prim's algorithm after the American mathematician Robert Clay Prim, who rediscovered and republished it in 1957. Additionally Edsger Dijkstra published this algorithm in 1959.

Algorithm description

Here we describe the algorithm in its simplest form. The minimum spanning tree is built gradually by adding edges one at a time. At first the spanning tree consists only of a single vertex (chosen arbitrarily). Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree. After that the spanning tree already consists of two vertices. Now select and add the edge with the minimum weight that has one end in an already selected vertex (i.e. a vertex that is already part of the spanning tree), and the other end in an unselected vertex. And so on, i.e. every time we select and add the edge with minimal weight that connects one selected vertex with one unselected vertex. The process is repeated until the spanning tree contains all vertices (or equivalently until we have $n - 1$ edges).

In the end the constructed spanning tree will be minimal. If the graph was originally not connected, then there doesn't exist a spanning tree, so the number of selected edges will be less than $n - 1$.

Proof

Let the graph G be connected, i.e. the answer exists. We denote by T the resulting graph found by Prim's algorithm, and by S the minimum spanning tree. Obviously T is indeed a spanning tree and a subgraph of G. We only need to show that the weights of S and T coincide.

Consider the first time in the algorithm when we add an edge to T that is not part of S. Let us denote this edge with e, its ends by a and b, and the set of already selected vertices as V ($a \in V$ and $b \notin V$, or vice versa).

In the minimal spanning tree S the vertices a and b are connected by some path P. On this path we can find an edge f such that one end of f lies in V and the other end doesn't. Since the algorithm chose e instead of f, it means that the weight of f is greater or equal to the weight of e.

We add the edge e to the minimum spanning tree S and remove the edge f. By adding e we created a cycle, and since f was also part of the only cycle, by removing it the resulting graph is again free of cycles. And because we only removed an edge from a cycle, the resulting graph is still connected.

The resulting spanning tree cannot have a larger total weight, since the weight of e was not larger than the weight of f, and it also cannot have a smaller weight since S was a minimum spanning tree. This means that by replacing the edge f with e we generated a different minimum spanning tree. And e has to have the same weight as f.

Thus all the edges we pick in Prim's algorithm have the same weights as the edges of any minimum spanning tree, which means that Prim's algorithm really generates a minimum spanning tree.

Implementation

The complexity of the algorithm depends on how we search for the next minimal edge among the appropriate edges. There are multiple approaches leading to different complexities and different implementations.

Trivial implementations: $O(nm)$ and $O(n^2 + m \log n)$

If we search the edge by iterating over all possible edges, then it takes $O(m)$ time to find the edge with the minimal weight. The total complexity will be $O(nm)$. In the worst case this is $O(n^3)$, really slow.

This algorithm can be improved if we only look at one edge from each already selected vertex. For example we can sort the edges from each vertex in ascending order of their weights, and store a pointer to the first valid edge (i.e. an edge that goes to an non-selected vertex). Then after finding and selecting the minimal edge, we update the pointers. This give a complexity of $O(n^2 + m)$, and for sorting the edges an additional $O(m \log n)$, which gives the complexity $O(n^2 \log n)$ in the worst case.

Below we consider two slightly different algorithms, one for dense and one for sparse graphs, both with a better complexity.

Dense graphs: $O(n^2)$

We approach this problem from a different angle: for every not yet selected vertex we will store the minimum edge to an already selected vertex.

Then during a step we only have to look at these minimum weight edges, which will have a complexity of $O(n)$.

After adding an edge some minimum edge pointers have to be recalculated. Note that the weights only can decrease, i.e. the minimal weight edge of every not yet selected vertex might stay the same, or it will be updated by an edge to the newly selected vertex. Therefore this phase can also be done in $O(n)$.

Thus we received a version of Prim's algorithm with the complexity $O(n^2)$.

In particular this implementation is very convenient for the Euclidean Minimum Spanning Tree problem: we have n points on a plane and the distance between each pair of points is the Euclidean distance between them, and we want to find a minimum spanning tree for this complete graph. This task can be solved by the described algorithm in $O(n^2)$ time and $O(n)$ memory, which is not possible with Kruskal's algorithm.

```
int n;
vector<vector<int>> adj; // adjacency matrix of graph
const int INF = 1000000000; // weight INF means there is no edge

struct Edge {
    int w = INF, to = -1;
};

void prim() {
    int total_weight = 0;
    vector<bool> selected(n, false);
    vector<Edge> min_e(n);
    min_e[0].w = 0;

    for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j = 0; j < n; ++j) {
            if (!selected[j] && (v == -1 || min_e[j].w < min_e[v].w))
                v = j;
        }

        if (min_e[v].w == INF) {
            cout << "No MST!" << endl;
            exit(0);
        }
```

```
        selected[v] = true;
        total_weight += min_e[v].w;
        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (int to = 0; to < n; ++to) {
            if (adj[v][to] < min_e[to].w)
                min_e[to] = {adj[v][to], v};
        }
    }

    cout << total_weight << endl;
}
```

The adjacency matrix `adj[][]` of size $n \times n$ stores the weights of the edges, and it uses the weight `INF` if there doesn't exist an edge between two vertices. The algorithm uses two arrays: the flag `selected[]`, which indicates which vertices we already have selected, and the array `min_e[]` which stores the edge with minimal weight to an selected vertex for each not-yet-selected vertex (it stores the weight and the end vertex). The algorithm does n steps, in each iteration the vertex with the smallest edge weight is selected, and the `min_e[]` of all other vertices gets updated.

Sparse graphs: $O(m \log n)$

In the above described algorithm it is possible to interpret the operations of finding the minimum and modifying some values as set operations. These two classical operations are supported by many data structure, for example by `set` in C++ (which are implemented via red-black trees).

The main algorithm remains the same, but now we can find the minimum edge in $O(\log n)$ time. On the other hand recomputing the pointers will now take $O(n \log n)$ time, which is worse than in the previous algorithm.

But when we consider that we only need to update $O(m)$ times in total, and perform $O(n)$ searches for the minimal edge, then the total complexity will be $O(m \log n)$. For sparse graphs this is better than the above algorithm, but for dense graphs this will be slower.

```
const int INF = 1000000000;

struct Edge {
    int w = INF, to = -1;
    bool operator<(Edge const& other) const {
        return make_pair(w, to) < make_pair(other.w, other.to);
    }
};

int n;
vector<vector<Edge>> adj;

void prim() {
    int total_weight = 0;
    vector<Edge> min_e(n);
    min_e[0].w = 0;
    set<Edge> q;
    q.insert({0, 0});
    vector<bool> selected(n, false);
    for (int i = 0; i < n; ++i) {
        if (q.empty()) {
```

```
            cout << "No MST!" << endl;
            exit(0);
        }

        int v = q.begin()->to;
        selected[v] = true;
        total_weight += q.begin()->w;
        q.erase(q.begin());

        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (Edge e : adj[v]) {
            if (!selected[e.to] && e.w < min_e[e.to].w) {
                q.erase({min_e[e.to].w, e.to});
                min_e[e.to] = {e.w, v};
                q.insert({e.w, e.to});
            }
        }
    }

    cout << total_weight << endl;
}
```

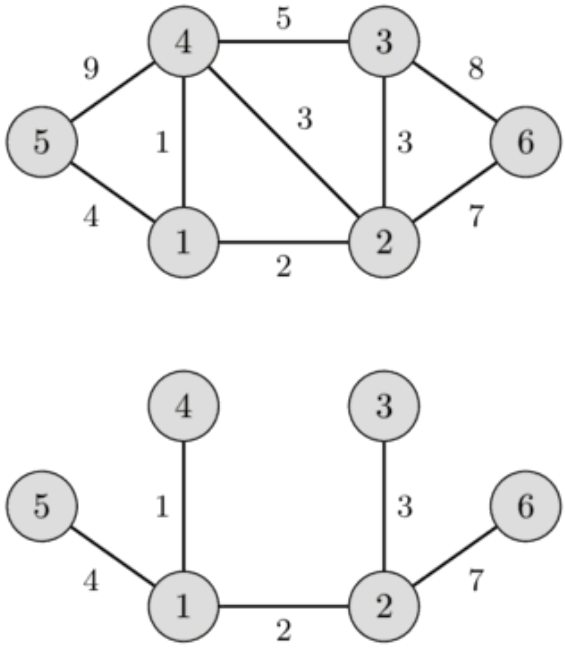
Here the graph is represented via a adjacency list `adj[]`, where `adj[v]` contains all edges (in form of weight and target pairs) for the vertex `v`. `min_e[v]` will store the weight of the smallest edge from vertex `v` to an already selected vertex (again in the form of a weight and target pair). In addition the queue `q` is filled with all not yet selected vertices in the order of increasing weights `min_e`. The algorithm does `n` steps, on each of which it selects the vertex `v` with the smallest weight `min_e` (by extracting it from the beginning of the queue), and then looks through all the edges from this vertex and updates the values in `min_e` (during an update we also need to also remove the old edge from the queue `q` and put in the new edge).

Last update: June 8, 2022 Translated From: e-maxx.ru

Minimum spanning tree - Kruskal's algorithm

Given a weighted undirected graph. We want to find a subtree of this graph which connects all vertices (i.e. it is a spanning tree) and has the least weight (i.e. the sum of weights of all the edges is minimum) of all possible spanning trees. This spanning tree is called a minimum spanning tree.

In the left image you can see a weighted undirected graph, and in the right image you can see the corresponding minimum spanning tree.



This article will discuss few important facts associated with minimum spanning trees, and then will give the simplest implementation of Kruskal's algorithm for finding minimum spanning tree.

Properties of the minimum spanning tree

- A minimum spanning tree of a graph is unique, if the weight of all the edges are distinct. Otherwise, there may be multiple minimum spanning trees. (Specific algorithms typically output one of the possible minimum spanning trees).
- Minimum spanning tree is also the tree with minimum product of weights of edges. (It can be easily proved by replacing the weights of all edges with their logarithms)
- In a minimum spanning tree of a graph, the maximum weight of an edge is the minimum possible from all possible spanning trees of that graph. (This follows from the validity of Kruskal's algorithm).

- The maximum spanning tree (spanning tree with the sum of weights of edges being maximum) of a graph can be obtained similarly to that of the minimum spanning tree, by changing the signs of the weights of all the edges to their opposite and then applying any of the minimum spanning tree algorithm.

Kruskal's algorithm

This algorithm was described by Joseph Bernard Kruskal, Jr. in 1956.

Kruskal's algorithm initially places all the nodes of the original graph isolated from each other, to form a forest of single node trees, and then gradually merges these trees, combining at each iteration any two of all the trees with some edge of the original graph. Before the execution of the algorithm, all edges are sorted by weight (in non-decreasing order). Then begins the process of unification: pick all edges from the first to the last (in sorted order), and if the ends of the currently picked edge belong to different subtrees, these subtrees are combined, and the edge is added to the answer. After iterating through all the edges, all the vertices will belong to the same sub-tree, and we will get the answer.

The simplest implementation

The following code directly implements the algorithm described above, and is having $O(M \log M + N^2)$ time complexity. Sorting edges requires $O(M \log N)$ (which is the same as $O(M \log M)$) operations. Information regarding the subtree to which a vertex belongs is maintained with the help of an array `tree_id[]` - for each vertex `v`, `tree_id[v]` stores the number of the tree , to which `v` belongs. For each edge, whether it belongs to the ends of different trees, can be determined in $O(1)$. Finally, the union of the two trees is carried out in $O(N)$ by a simple pass through `tree_id[]` array. Given that the total number of merge operations is $N - 1$, we obtain the asymptotic behavior of $O(M \log N + N^2)$.

```
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<int> tree_id(n);
vector<Edge> result;
for (int i = 0; i < n; i++)
    tree_id[i] = i;

sort(edges.begin(), edges.end());

for (Edge e : edges) {
    if (tree_id[e.u] != tree_id[e.v]) {
        cost += e.weight;
        result.push_back(e);

        int old_id = tree_id[e.u], new_id = tree_id[e.v];
        for (int i = 0; i < n; i++) {
            if (tree_id[i] == old_id)
                tree_id[i] = new_id;
```

```
            }
        }
    }
```

Proof of correctness

Why does Kruskal's algorithm give us the correct result?

If the original graph was connected, then also the resulting graph will be connected. Because otherwise there would be two components that could be connected with at least one edge. Though this is impossible, because Kruskal would have chosen one of these edges, since the ids of the components are different. Also the resulting graph doesn't contain any cycles, since we forbid this explicitly in the algorithm. Therefore the algorithm generates a spanning tree.

So why does this algorithm give us a minimum spanning tree?

We can show the proposal "if F is a set of edges chosen by the algorithm at any stage in the algorithm, then there exists a MST that contains all edges of F" using induction.

The proposal is obviously true at the beginning, the empty set is a subset of any MST.

Now let's assume F is some edge set at any stage of the algorithm, T is a MST containing F and e is the new edge we want to add using Kruskal.

If e generates a cycle, then we don't add it, and so the proposal is still true after this step.

In case that T already contains e, the proposal is also true after this step.

In case T doesn't contain the edge e, then $T + e$ will contain a cycle C. This cycle will contain at least one edge f, that is not in F. The set of edges $T - f + e$ will also be a spanning tree. Notice that the weight of f cannot be smaller than the weight of e, because otherwise Kruskal would have chosen f earlier. It also cannot have a bigger weight, since that would make the total weight of $T - f + e$ smaller than the total weight of T, which is impossible since T is already a MST. This means that the weight of e has to be the same as the weight of f. Therefore $T - f + e$ is also a MST, and it contains all edges from $F + e$. So also here the proposal is still fulfilled after the step.

This proves the proposal. Which means that after iterating over all edges the resulting edge set will be connected, and will be contained in a MST, which means that it has to be a MST already.

Improved implementation

We can use the **Disjoint Set Union** (DSU) data structure to write a faster implementation of the Kruskal's algorithm with the time complexity of about $O(M \log N)$. This article details such an approach.

Practice Problems

- SPOJ - Koicost
- SPOJ - MaryBMW
- Codechef - Fullmetal Alchemist

- Codeforces - Edges in MST
- UVA 12176 - Bring Your Own Horse
- UVA 10600 - ACM Contest and Blackout
- UVA 10724 - Road Construction
- Hackerrank - Roads in HackerLand
- UVA 11710 - Expensive subway
- Codechef - Chefland and Electricity
- UVA 10307 - Killing Aliens in Borg Maze
- Codeforces - Flea
- Codeforces - Igon in Museum
- Codeforces - Hongcow Builds a Nation
- UVA - 908 - Re-connecting Computer Sites
- UVA 1208 - Oreon
- UVA 1235 - Anti Brute Force Lock
- UVA 10034 - Freckles
- UVA 11228 - Transportation system
- UVA 11631 - Dark roads
- UVA 11733 - Airports
- UVA 11747 - Heavy Cycle Edges
- SPOJ - Blinet
- SPOJ - Help the Old King
- Codeforces - Hierarchy
- SPOJ - Modems
- CSES - Road Reparation
- CSES - Road Construction