Min(imum) Spanning Tree (MST)

1. Introduction

A **Spanning Tree (ST)** of a connected undirected weighted graph **G** is a subgraph of **G** that is a **tree** and **connects (spans) all vertices of G**. A graph **G** can have many STs (see this or this), each with different total weight (the sum of edge weights in the ST).

A **Min(imum) Spanning Tree (MST)** of **G** is an ST of **G** that has the **smallest total weight** among the various STs.

1-1. MST Problem

The **MST** problem is a standard graph (and also optimization) problem defined as follows: Given a connected undirected weighted graph **G = (V, E)**, select a **subset** of edges of **G** such that the graph is still connected but with minimum total weight. The output is either the actual MST of **G** (there can be several possible MSTs of **G**) or usually just the minimum total weight itself (this is unique).

1-2. Motivating Example

Government wants to link **N** rural villages in the country with **N-1** roads.
(that is a *spanning tree* with **N** vertices and **N-1** edges).

The cost to build a road to connect two villages depends on the terrain, distance, etc.
(that is a *complete undirected weighted graph* of **N*(N-1)/2** weighted edges).

You want to minimize the total building cost. How are you going to build the roads?
(that is *minimum spanning tree*).

PS: There is a variant of this problem that requires more advanced solution, e.g., see this.

1-3. MST Algorithms

The **MST** problem has polynomial solutions.

In this visualization, we will learn two of them: Kruskal's algorithm and Prim's algorithm. Both are classified as **Greedy** Algorithms. Note that there are other MST algorithms outside the two presented here.

2. Visualisation

View the visualisation of MST algorithm above.

Originally, all vertices and edges in the input graph are colored with the standard black color on white background.

At the end of the MST algorithm, **|V|-1** MST edges (and all **|V|** vertices) will be colored orange and non-MST edges will be colored grey.

3. Input Graph

There are two different sources for specifying an input graph:

1. **Edit Graph**: You can edit the currently displayed connected undirected weighted graph or draw your own input graph.
2. **Example Graphs**: You can select from the list of example connected undirected weighted graphs to get you started.

4. Kruskal's Algorithm

Kruskal's algorithm: An O(**E** log **V**) greedy MST algorithm that grows a forest of minimum spanning trees and eventually combine them into one MST.

Kruskal's requires a good sorting algorithm to sort edges of the input graph (usually stored in an Edge List data structure) by non-decreasing weight and another data structure called Union-Find Disjoint Sets (UFDS) to help in checking/preventing cycle.

4-1. The Basic Idea

Kruskal's algorithm first sort the set of edges **E** in non-decreasing weight (there can be edges with the same weight), and if ties, by increasing smaller vertex number of the edge, and if still ties, by increasing larger vertex number of the edge.

Discussion: Is this the only possible sort criteria?

4-2. The Answer

[This is a hidden slide]

4-3. The Basic Idea - Continued

Then, Kruskal's algorithm will perform a loop through these sorted edges (that already have non-decreasing weight property) and **greedily** taking the next edge **e** if it does **not** create any cycle w.r.t. edges that have been taken earlier.

Without further ado, let's try Kruskal on the default example graph (that has three edges with the same weight). Go through this animated example first before continuing.

4-4. Short Proof of Correctness - 1

To see on why the **Greedy Strategy** of Kruskal's algorithm works, we define a **loop invariant**: Every edge **e** that is added into tree **T** by Kruskal's algorithm is part of the **MST**.

At the start of Kruskal's main loop, **T = {}** is always part of **MST** by definition.

Kruskal's has a special cycle check in its main loop (using UFDS data structure) and only add an edge **e** into **T** if it will never form a cycle w.r.t. the previously selected edges.

At the end of the main loop, Kruskal's can only select **V**-1 edges from a connected undirected weighted graph **G** without having any cycle. This implies that Kruskal's produces a Spanning Tree.

On the default example, notice that after taking the first 2 edges: 0-1 and 0-3, in that order, Kruskal's **cannot** take edge 1-3 as it will cause a cycle 0-1-3-0. Kruskal's then take edge 0-2 but it cannot take edge 2-3 as it will cause cycle 0-2-3-0.

4-5. Short Proof of Correctness - 2

We have seen in the previous slide that Kruskal's algorithm will produce a tree **T** that is a Spanning Tree (ST) when it stops. But is it the minimum ST, i.e., the **MST**?

To prove this, we need to recall that **before** running Kruskal's main loop, we have already sort the edges in non-decreasing weight, i.e., the latter edges will have equal or **larger** weight than the earlier edges.

4-6. Short Proof of Correctness - 3

At the start of every loop, **T** is always part of MST.

If Kruskal's only add a legal edge **e** (that will not cause cycle w.r.t. the edges that have been taken earlier) with **min cost**, then we can be sure that **w(T U e) ≤ w(T U any other unprocessed edge e' that does not form cycle)** (by virtue that Kruskal's has sorted the edges, so **w(e) ≤ w(e')**).

Therefore, at the end of the loop, the Spanning Tree **T** must have minimal overall weight **w(T)**, so **T** is the final MST.

On the default example, notice that after taking the first 2 edges: 0-1 and 0-3, in that order, and ignoring edge 1-3 as it will cause a cycle 0-1-3-0, we can safely take the next smallest legal edge 0-2 (with weight 2) as taking any other legal edge (e.g., edge 2-3 with **larger** weight 3) will either create **another** MST with equal weight (not in this example) or **another** ST that is not minimum (which is this example).

4-7. Implementation Sketch

There are two parts of Kruskal's algorithm: Sorting and the Kruskal's main loop.

The sorting of edges is easy. We just store the graph using **Edge List** data structure and sort **E** edges using any O(**E** log **E**) = O(**E** log **V**) sorting algorithm (or just use C++/Python/Java sorting library routine) by non-decreasing weight, smaller vertex number, higher vertex number. This O(**E** log **V**) is the bottleneck part of Kruskal's algorithm as the second part is actually lighter, see below.

Kruskal's main loop can be easily implemented using Union-Find Disjoint Sets data structure. We use **IsSameSet(u, v)** to test if taking edge **e** with endpoints **u** and **v** will cause a cycle (same connected component -- there is another path in the subtree that can connect **u** to **v**, thus adding edge **(u, v)** will cause a cycle) or not. If **IsSameSet(u, v)** returns false, we greedily take this next smallest and legal edge **e** and call **UnionSet(u, v)** to prevent future cycles involving this edge. This part runs in O(**E**) as we assume UFDS **IsSameSet(u, v)** and **UnionSet(u, v)** operations run in O(**1**) for a relatively small graph.

5. Prim's Algorithm

Prim's algorithm: Another O(**E** log **V**) greedy MST algorithm that grows a Minimum Spanning Tree from a starting source vertex until it spans the entire graph.

Prim's requires a Priority Queue data structure (usually implemented using Binary Heap but we can also use Balanced Binary Search Tree too) to dynamically order the currently considered edges based on non-decreasing weight, an Adjacency List data

structure for fast neighbor enumeration of a vertex, and a Boolean array (a Direct Addressing Table) to help in checking cycle.

Another name of Prim's algorithm is Jarnik-Prim's algorithm.

5-1. The Basic Idea

Prim's algorithm starts from a designated source vertex **s** (usually vertex 0) and enqueues all edges incident to **s** into a Priority Queue (PQ) according to non-decreasing weight, and if ties, by increasing vertex number (of the neighboring vertex number). Then it will repeatedly do the following greedy steps: If the vertex **v** of the front-most edge pair information **e: (w, v)** in the PQ has **not** been visited, it means that we can greedily extends the tree **T** to include vertex **v** and enqueue edges connected to **v** into the PQ, otherwise we discard edge **e** (because Prim's grows one spanning tree from **s**, the fact that **v** is already visited implies that there is another path from **s** to **v** and adding this edge will cause a cycle).

Without further ado, let's try Prim(1) on the default example graph (that has three edges with the same weight). That's it, we start Prim's algorithm from source vertex **s = 1**. Go through this animated example first before continuing.

5-2. Short Proof of Correctness - Part 1

Prim's algorithm is a **Greedy Algorithm** because at each step of its main loop, it always try to select the next valid edge **e** with minimal weight (that is greedy!).

To convince us that Prim's algorithm is correct, let's go through the following simple proof: Let **T** be the spanning tree of graph **G** generated by Prim's algorithm and **T*** be the spanning tree of **G** that is known to have minimal cost, i.e. **T*** is the **MST**.

If **T == T***, that's it, Prim's algorithm produces exactly the same **MST** as **T***, we are done.

But if **T != T***...

5-3. If T != T*, Part 1

Assume that on the default example, **T = {0-1, 0-3, 0-2}** but **T* = {0-1, 1-3, 0-2}** instead.

Let $\mathbf{e_k}$ **= (u, v)** be the first edge chosen by Prim's Algorithm at the **k**-th iteration that is not in **T*** (on the default example, **k = 2**, $\mathbf{e_2}$ **= (0, 3)**, note that **(0, 3)** is not in **T***).

Let **P** be the path from **u** to **v** in **T***, and let **e*** be an edge in **P** such that one endpoint is in the tree generated at the (**k**−1)-th iteration of Prim's algorithm and the other is not (on the default example, **P = 0-1-3** and **e* = (1, 3)**, note that vertex **1** is inside **T** at first iteration **k = 1**).

5-4. If T != T*, Part 2

If the weight of **e*** is less than the weight of $\mathbf{e_k}$, then Prim's algorithm would have chosen **e*** on its **k**-th iteration as that is how Prim's algorithm works.

So, it is certain that $\mathbf{w(e^*) \geq w(e_k)}$.
(on the example graph, **e* = (1, 3)** has weight 1 and $\mathbf{e_k}$ **= (0, 3)** also has weight 1).

When weight **e*** is = weight $\mathbf{e_k}$, the choice between the **e*** or $\mathbf{e_k}$ is actually arbitrary. And whether the weight of **e*** is ≥ weight of $\mathbf{e_k}$, **e*** can always be substituted with $\mathbf{e_k}$ while preserving minimal total weight of **T***. (on the example graph, when we replace **e* = (1, 3)** with $\mathbf{e_k}$ **= (0, 3)**, we manage to transform **T*** into **T**).

5-5. Short Proof of Correctness - Part 2

But if **T != T***... (continued)

We can repeat the substitution process outlined earlier repeatedly until **T* = T** and thereby we have shown that the spanning tree generated by any instance of Prim's algorithm (from any source vertex **s**) is an MST as whatever the optimal MST is, it can be transformed to the output of Prim's algorithm.

5-6. Implementation Sketch

We can easily implement Prim's algorithm with two well-known data structures:

1. A Priority Queue PQ (Binary Heap inside C++ STL priority_queue/Python heapq/Java PriorityQueue or Balanced BST inside C++ STL set/Java TreeSet), and
2. A Boolean array of size **V**, essentially a Direct Addressing Table (to decide if a vertex has been taken or not, i.e., in the same connected component as the source vertex **s** or not).

With these, we can run Prim's Algorithm in O(**E** log **V**) because we process each edge once and each time, we call **Insert((w, v))** and **(w, v) = ExtractMax()** from a PQ in O(log **E**) = O(log $\mathbf{V^2}$) = O(2 log **V**) = O(log **V**). As there are **E** edges, Prim's Algorithm runs in O(**E** log **V**).

6. Poll

Quiz: **Having seen both Kruskal's and Prim's Algorithms, which one is the better MST algorithm?**

- Prim's Algorithm
- It Depends
- Kruskal's Algorithm

Discussion: Why?

6-1. The Answer

[This is a hidden slide]

7. Extras

You have reached the end of the basic stuffs of this Min(imum) Spanning Tree graph problem and its two classic algorithms: Kruskal's and Prim's (there are others, like another O(E log V) Boruvka's algorithm, but not discussed in this visualization). We encourage you to explore further in the **Exploration Mode**.

However, the harder MST problems can be (much) more challenging that its basic version.

Once you have (roughly) mastered this MST topic, we encourage you to study more on harder graph problems where MST is used as a component, e.g., approximation algorithm for NP-hard (Metric No-Repeat) TSP and Steiner Tree problems.

7-1. MST Problem Variants

We write a few MST problem variants in the Competitive Programming book.

1. Max(imum) Spanning Tree,
2. Min(imum) Spanning Subgraph,
3. Min(imum) Spanning Forest,
4. Second Best Spanning Tree,

5. Minimax (Maximin) Path Problem, etc

Advertisement: Buy CP book to study more about these variants and see that sometimes Kruskal's is better and sometimes Prim's is better at some of these variants.

7-2. Online Quiz

For a few more challenging questions about this MST problem and/or Kruskal's/Prim's Algorithms, please practice on MST training module (no login is required, but on medium difficulty setting only).

However, for NUS students, you should login to officially clear this module and such achievement will be recorded in your user account.

7-3. Online Judge Exercises

This MST problem can be much more challenging than this basic form. Therefore we encourage you to try the following two ACM ICPC contest problems about MST: UVa 01234 - RACING and Kattis - arcticnetwork.

Try them to consolidate and improve your understanding about this graph problem.

You are allowed to use/modify our implementation code for Kruskal's/Prim's Algorithms:
kruskal.cpp | py | java | ml
prim.cpp | py | java | ml

7-4. Discussion

[This is a hidden slide]