Minimum Spanning Tree

Minimum Spanning Tree

A MST is a spanning tree with minimum weight for a graph G

- Tree (T)
  - A tree is a connected graph with V vertices and V-1 edges
  - There is only one unique path between any two vertices in a tree
- Spanning Tree (ST)
  - A tree that covers/spans every vertex in graph G

Understanding Real World Application

Vertex set V

- Similar to sets of houses or street intersections

Edge set E

- Similar to sets of streets, roads, and avenues
- Generally undirected
  - A road can go both ways
- Weighted
  - May be in terms of distance, time, or money etc.

Weight function w(a, b): E->R

- Sets the weight of edge from a to b
- Weighted graph G: G(V, E), w(a,b): E->R

When we talk about MST, we are dealing with connected, undirected and weighted graphs

- There is a path from any vertex a to any other vertex b in G
- We want to find the connection of vertices with the lowest possible cost/weight

For the spanning tree ST of connected, undirected and weighted graph G:

- Let w(ST), weight of ST, denote the total weight of edges in ST:
- $w(ST) = \sum_{(a,b)\in ST} w(a,b)$

MST of graph G is thus an ST of G with the minimum possible w(ST).

Ways of Solving this Problem

1. Brute Force/Complete Search

Consider all cycles in the graph. For every cycle, we remove the largest edge. If one or more edges in a cycle have already removed, move on to the next cycle.

However, this solution is way too inefficient as it is very difficult to get all cycles in a graph. We can have up to $O(2^N)$ different cycles! Listing them down one by one is slow.

2. Prim's Algorithm

The concept is to start from one vertex, and constantly pick the edge that has the minimum weight out of all accessible edges from already connected vertices.

Prim's Algorithm Implementation: PriorityQueue and Array
The above algorithm can be implemented using a PriorityQueue to enable to polling of the edge of minimum weight, as well as a Boolean array to track whether a vertex has been taken or not.

This is the pseudo code:

```
T ← {s}, a starting vertex s (usually vertex 0)
enqueue edges connected to s (only the other ending vertex and edge
      weight) into a priority queue PQ that orders elements based on
      increasing weight

while there are unprocessed edges left in PQ // O(E)
      take out the front most edge e // O(log E)
      if vertex v linked with this edge e is not taken yet
            T ← T ∪ v (including this edge e)
            enqueue each edge adjacent to v into the PQ if it is not
            already in T // O(log E)

T is an MST
```

The concept is to:

- Start from a vertex and mark it as visited
- Queue all of its edges into a PriorityQueue that's sorted by the edge weight
- Dequeue the edge with minimum weight
- If the vertex has already been visited, skip and poll the next edge
- Else add that vertex to the tree
- Queue all of that vertex's edges into the PriorityQueue
- Repeat until the all vertices have been added

The above algorithm can be changed according to output required, e.g. minimum weight or the minimum spanning tree itself.

The overall time complexity is O(E log E) = O(E log V^2) = O(E*2logV) = **O(E log V)**

Claim: Prim's Algorithm gives us a MST
Proof by Contradiction:
Prim's Algorithm is a greedy algorithm.

Assume that edge e is the first edge at iteration k chosen by Prim's which is not in any valid MST. Let T be the tree generated by Prim's before adding e. Now T must be a subtree of some valid MST T'.

Adding edge e to T' will now create a cycle. Since e has 1 endpoint in T (the valid endpoint) and one endpoint outside T, trace around this cycle in T' until we get to some edge e' that goes back to T.

By Prim's algorithm e and e' must be candidate edges at iteration k, but e was chosen meaning w(e) ≤ w(e'). Now replacing e' with e in T' must give us tree T'' covering all vertices of the graph s.t w(T'') ≤ w(T'). Contradiction that e is first edge chosen wrongly

3. Kruskal's Algorithm

Kruskal's Algorithm is slightly more intuitive than Prim's. We basically keep picking the edge of lowest weight to be part of the MST, with the condition that the edge must be connecting our current tree to a previously unconnected vertex, until all vertices are connected to the tree. That tree is thus the MST.

Kruskal's Algorithm Implementation: UFDS and Edge List

It makes sense to use UFDS here, as we want to prevent cycles, and need to have a quick way to check if a certain vertex has been included previously or not. The Edge List enables us to quickly sort the edges by weight, and allows us to pick the shortest edge from the front of the edge list until all vertices have been connected or all edges have been taken.

The pseudo code is as such:

```
sort the set of E edges by increasing weight // O(E log V)
T ← {}
while there are unprocessed edges left // O(E)
      pick an unprocessed edge e with min cost // O(1)
      if adding e to T does not form a cycle // O(α(V)) = O(1)
            add e to T // O(1)
T is an MST
```

What we get as we progress is a forest of subtrees of the MST, until all subtrees are joined into a single MST.

Only Edge List will enable us to sort all edges with a single O(E log E) = O(E log V) operation. Adjacency Matrices and Adjacency Lists will not work, and conversion may be needed.

The overall time complexity of Kruskal's Algorithm is thus O(E log V + Eα(V)). However, due to the Inverse Ackermann Function being close to O(1), overall O(E log V) dominates.

Claim: Kruskal's Algorithm will give a MST.
Proof by contradiction:
Kruskal's Algorithm is also a greedy algorithm.

Assume that edge e is the first edge at iteration k chosen by Kruskal's which is not in any valid MST. Let F be the forest generated by Kruskal's before adding e. Now F must be a part of some valid MST T'.

Putting e into T' will create a cycle. Trace the cycle until an edge e' which connects a vertex in F with another vertex not in F.

At iteration k, both e and e' are candidate (they are not chosen and do not form a cycle if chosen). Since e was chosen, $w(e) \leq w(e')$. Now replacing e' with e in T' must give us tree T'' covering all vertices of the graph s.t $w(T'') \leq w(T')$. Contradiction that e is first edge chosen wrongly.

MINiMAX and MAXiMIN Path
To find the MINiMAX path from a to b, simply find the Minimum Spanning Tree and find the path between a and b. This is the path that minimises the maximum edge weight from a to b.

The MAXiMIN path is the opposite. Find the Maximum Spanning Tree and the path between a and b would be the path that maximises the minimum edge weight from a to b. An application of this would be the identification of how heavy a truck driving from a to be can be, which would depend on the minimum weight allowed by one of the roads along the way.

Proving Questions from Tutorial
Claim: The edge weights of a connected undirected weighted graph G are unique. Therefore, the MST of G may not be unique (i.e. there can be a MST-1 of G that is structurally different than MST-2 of G, such that both MST-1 and MST-2 have same minimum weight.)

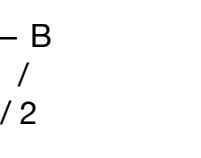
Proof:
False. Since there can only be one path from one vertex to another, and that path would be a minimum, we can take it as that any subset of this path would be the minimum from the start of that subpath to the end of the subpath. With this logic, the edge chosen between any two connected vertices would be of minimum weight. Since the edges cannot be of the same weight, the MST formed must be unique as there can only be one minimum weighted edge between two points connected at the end in the MST.

Proof by contradiction:
For the edges to be substitutable, there needs to be some equality.

Claim: The minimum spanning tree of a connected undirected weighted graph where some edge weights are not distinct is always not unique.

Proof:
False, “always not unique”, not “not always unique”.

Example would be:



The MST would be 2.

Claim: We can make Kruskal’s algorithm run faster than O(E lg V) if the input graph is a connected undirected weighted graph where all edge weight is within a small integer range [1 … 100].

True. Use radix sort for O(E) efficiency. Then the subsequent procedures are all O(E).

Kruskal's algorithm is O(E lg V) due to the sorting. If we get rid of the sorting, everything becomes O(E), due to O(1) UFDS operations * E edges.

Hence O(E) + O(E) = O(E).

Inserting a new Edge into an existing MST
We do not need to insert (A,B) into the original graph G and re-run Prim's or Kruskal's. This runs in O((E + 1) log V) = O(E log V). We can instead insert the edge (A,B) into the MST itself and then re-run Prim's or Kruskal's on this MST+1 edge. As a MST is a tree, it only has E = V−1 edges, so this is O((V− 1+1) log V) = O(V log V). But we can do better.

Before insertion of this edge (A,B) into the MST, there is only a path from vertex A to vertex B in the MST. After the insertion of edge (A,B), we have a cycle which consists of the edges of the path from A and B and the new edge (A,B). In order to update the MST, we only need to find the largest edge in this cycle and remove it. This will ensure that the resultant graph will remains the MST for the new Graph G∪(A,B).

We run DFS from A in the original MST, keeping track of the largest edge X seen as the DFS recursively explores all paths, until it hits B. Now the largest edge X must be the largest edge of the path from A to B. Stop the DFS and compare X with the new edge (A,B). If weight of (A,B) is larger than X, then the MST does not change. Otherwise, we remove X and insert (A,B) instead into the MST to have a better MST. This algorithm runs in O(V) since DFS is O(V) on a tree and removing X and inserting (A,B) can also be done in O(V) (e.g. we use a (sorted) Edge List, we can find X with an O(V) loop and we can insert (A,B) in the correct position also in O(V)).