Minimum Spanning Trees

DEFINITION:

Given an undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbf{R}$ which describes weights on the edges of the graph. *A minimum spanning tree* of G is a subset $T \subseteq E$ such that

- T is acyclic (no cycles)
- T connects all of the vertices in V
- its total weight (= sum of the weights of all edges in T)

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

is minimized. Here $w(u, v)$ denotes a weight of the edge $(u, v) \in E$.

DEFINITION:

An algorithm uses the *greedy strategy* if at each step of the algorithm the best at the moment choice is made. This (heuristic) strategy is not guaranteed to find globally optimal solutions but for some problems it works reasonably well.

Growing A Minimum Spanning Tree (MST)

Given a connected, undirected graph $G = (V, E)$ with a real weight function w.

Definitions:

Let A is a subset of some MST. An edge (u, v) is a *safe edge* for A if $A \cup \{(u, v)\}$ is also a subset of a MST.

Let S be a subset of V. A *cut* $(S, V - S)$ is a partition of V such that for any vertex v, we have that $v \in S$ or $v \in V - S$.

An edge $(u, v) \in E$ *crosses* the cut $(S, V - S)$ if $u \in S$ and $v \in V - S$, or $v \in S$ and $u \in V - S$.

A cut *respects* the set A of edges if no edge in A crosses the cut.

An edge is a *light edge* crossing a cut if its weight is the minimum of the weights of edges crossing the cut. Note that there can be more than one light edge crossing the cut.

Generic-MST(G, w):

```
Initialize A as an empty set
 while (A is not a spanning tree) {
   find an edge (u, v) which is safe for A
   add the safe edge (u, v) to the set A
 }
 return A
```

THEOREM:

Let $G = (V, E)$ be a connected, undirected graph with a weight function w defined on E. Let A be a subset of E that is included in some MST, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, the edge (u, v) is safe for A.

The Generic-MST algorithm always returns a MST. There are two variants of this algorithm: Kruskal's algorithm and Prim's algorithm.

Kruskal's Algorithm

Kruskal's algorithm is based on the Generic-MST algorithm. The set A is a forest (a set of trees). The safe edge added to A is always a least-weight edge in the graph that connects two distinct components (two distinct trees in the forest). Kruskal's algorithm is a greedy algorithm since at each step it adds to the forest an edge of least possible weight.

One of implementations of Kruskal's algorithm is to use a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in a tree of the current forest. Here we consider a graph $G = (V, E)$ with a weight function w.

MST-Kruskal(G, w):

1. initialize A as an empty set;
2. `for` (each vertex $v \in V$) {
3. Make-Set(v); }
4. sort the edges of E by increasing weight w;
5. `for` (each edge $(u, v) \in E$, in sorted order)
6. `if` (Find-Set(u) ! = Find-Set(v)) {
7. add the edge (u, v) to the set A;
8. Union(u, v); }
9. return A;

Analysis of Kruskal's Algorithm

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the implementation of the disjoint-set data structure. The fastest implementation of this data structure is based on disjoint set forest implementation with union-by-rank and path-compression heuristics.

Initialization (steps 1–3) takes $O(V)$ time.

Sorting the edges (step 4) takes $O(E \lg E)$ time.

Disjoint set forest (steps 5–8): $2E$ `for` loop iterations and therefore the number of Find-Set and Union operations is $O(E)$. $4E$ Find-Set and E Union take $O(E \log^* E)$ time.

Total (average) running time: $O(V + E \lg E)$ time (since $\log^* E \leq \lg E$).

Prim's Algorithm

Prim's algorithm has the property that the edges in the set A always form a single tree. The tree starts from a arbitrary root vertex r and grows until the tree spans all the vertices of V. At each step, a light edge connecting a vertex in A to a vertex in $V - A$ is added to the tree. This rule adds only edges that are safe for A. This is a greedy algorithm.

The algorithm uses a priority queue Q based on a weight w field. For each vertex v, $w[v]$ is the minimum weight of any edge connecting v to a vertex in the tree, or it is ∞ if there is no such edge. The field $p[v]$ is a parent of v in the tree. The minimum spanning tree for a graph $G = (V, E)$ is

$$A = \{(v, p[v]) \ : \ v \in V - \{r\}\}.$$

MST-Prim(G, w, r):

1. initialize Q with all the vertices from V;
2. `for` (each $u \in Q$) $w[u] = \infty$;
3. $w[r] = 0$;
4. $p[r] =$ `NIL`;
5. $A = \emptyset$;
6. `while` (Q is not empty) {
7. $u =$ Extract-Min(Q);
8. `if` ($u != r$) $A = A \cup (p[u], u)$;
9. `for` (each $v \in Adj[u]$)
10. `if` ($v \in Q$ and $w(u, v) < w[v]$) {
11. $p[v] = u$;
12. $w[v] = w(u, v)$; }
13. } /* end while */
14. return A;

Extract-Min(Q) is a operation for the priority queue Q which removes and returns a vertex with a minimal value of the w field. Note that this priority queue uses the operation $<$ (less than) for comparisons, therefore vertices in the queue Q with smaller weights precede vertices with larger weights.

Analysis of Prim's Algorithm

The performance of Prim's algorithm depends on implementation of the priority queue Q. The simplest (and reasonably fast) is a binary heap implementation. Other possibility is to choose Fibonacci heaps implementation which is more complex but faster than the binary heap implementation.

In a binary heap implementation the steps 1–3 use Build-Heap procedure: it takes $O(V)$ time. The `while` loop is executed $|V|$ times. Extract-Min operation takes $O(\lg V)$ time, so the total time for Extract-Min is $O(V \lg V)$. The `for` loop in steps 9–12 is executed $O(E)$ times (sum of lengths of all adjacency lists is $2|E|$). To test $v \in Q$ we usually use an additional binary field (say `in_queue` having `YES` or `NO` values), and such test takes $O(1)$ time. Updating the field w in the step 12 requires restoring the heap property of the queue, which takes $O(\lg V)$ time.

Total time: $O(V \lg V + E \lg V) = O((V + E) \lg V)$ which is asymptotically the same as Kruskal's algorithm (since $E = O(V^2)$ and $\lg E = O(2V)$ for dense graphs).