Minimum spanning tree

A *spanning tree* is a non-cyclic sub-graph of a connected and undirected graph G that connects all the vertices together. A graph G can have multiple spanning trees.

A *minimum spanning tree* is a spanning tree which has minimal total weight. In other words, minimum spanning tree is the one which contains the least weight among all other spanning tree of some particular graph.

Two classic algorithms for solving the MST problem

Prim's Algorithm

Grows the MST from a single root vertex, much in the same way as Dijkstra's shortest-path algorithm. Will fail in non-connected graph.

In the pseudocode below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue Q based on a *key* attribute. For each vertex v, the attribute v.key is the minimum weight of any edge connecting v to a vertex in the tree; by convention, $v.key = \infty$ if there is no such edge. The attribute $v.\pi$ names the parent of v in the tree.

```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

Prims' complexity analysis

Let G be a simple graph with n vertices and m edges. The algorithm relies on a priority queue Q data structure. We initially perform n insertions into Q, later perform n extract-min operations, and may update a total of m priorities as part of the algorithm. Those steps are the primary contributions to the overall running time. With a heap-based priority queue, each operation runs in O(logn) time, and the overall time for the algorithm is O((n+m) logn), which is O(m logn) for a connected graph. Alternatively, we can achieve $O(n^2)$ running time by using an unsorted list as a priority queue.

Technique 2: Executing the depth-first search and recording the edges you've traveled to make the search, you automatically create a minimum spanning tree. The only difference between the minimum spanning tree method mst(), and the depth-first search method dfs(), is that mst() must somehow record the edges traveled.

Kruskal's Algorithm

Can be used to non-connected graph but will find the MST for individual disconnected graphs.

Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight. In below implementation of Kruskal's algorithm it uses a *disjoint-set data structure* to maintain several disjoint sets of elements.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

The second loop checks, for each edge (u, v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and hence the edge is discarded.

Kruskal's complexity analysis

Let graph G be a simple graph with n vertices and m edges.There are two primary contributions to the running time of Kruskal's algorithm. The *first* is the need to consider the edges in non-decreasing order of their weights, and the *second* is the management of the cluster partition. Analyzing its running time requires that we give more details on its implementation.

The ordering of edges by weight can be implemented in *O*(m logm), either by use of a sorting algorithm or a priority queue Q. The remaining task is the management of clusters. To implement Kruskal's algorithm, we must be able to find the clusters for vertices u and v that are endpoints of an edge e, to test whether those two clusters are distinct, and if so, to merge those two clusters into one. In the context of Kruskal's algorithm, we perform at most 2m "find" operations and n − 1 "union" operations. We will see that a simple union-find structure can perform that combination of operations in *O*(m + n logn) time. For a connected graph, (m ≥ n−1) therefore, the bound of *O*(m logn) time for ordering the edges dominates the time for managing the clusters. We conclude that the running time of Kruskal's algorithm is *O*(m logn).

These algorithms are both applications of the *greedy method*, and is based on choosing objects to join a growing collection by iteratively picking an object that minimizes some cost function. There are many possible minimum spanning trees for a given set of vertices.

Note that the number of edges E in a minimum spanning tree is always one less than the number of vertices V i.e. $E = V - 1$

Application of minimum spanning tree

- Network design
  telephone, electrical, hydraulic, TV cable, computer, road
  The standard application is to a problem like phone network design. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.
- Approximation algorithms for NP-hard problems
  traveling salesperson problem, Steiner tree
  A convenient formal way of defining this problem is to find the shortest path that visits each point at least once. The Steiner tree problem in graphs can be seen as a generalization of two other famous combinatorial optimization problems: the (non-negative) shortest path problem and the minimum spanning tree problem.
- Cluster analysis
  k clustering problem can be viewed as finding an MST and deleting the k-1 most

expensive edges.

- Indirect applications
  – max bottleneck paths
  – learning salient features for real-time face verification