CS 6901 (Applied Algorithms) – Lecture 5

Antonina Kolokolova

September 27, 2016

1 Minimum Spanning Trees

1.1 Electrification of South Moravia

In 1926, Czech mathematician Otakar Borůvka considered a very practical problem: how to get electricity to all the towns in South Moravia, a region of Czech republic. He got the problem from a friend working at the West-Moravian Powerplants, and set out to solve it, producing what is now known as Borůvka's algorithm. He published both a mathematical paper explaining the algorithm and proving its correctness, and a paper at the "Electrotechnical news" explaining how to apply this algorithm to solve the electrification problem. See the paper by Nesetril, Milkova,Nesetrilova "Otakar Borůvka on minimum spanning tree problem" for more historical details, as well as a translation of the original papers.

The electrification of South Moravia problem is stated as follows. There is a number of towns in the region, and the distances between pairs of the town are known. The goal is to create a system of electric power lines that would minimize the total distance (and thus the construction cost), yet reach every town. In this scenario, it is not necessary to connect each town to the "source of electricity" directly – it is enough to connect it to another town that already got the power.
Also, here we are not considering the cost of connecting the first town to the electric grid, just interconnecting towns in South Moravia.
The picture to the right is the map of South Moravia (from south-moravia.webnode.cz).

The mathematical representation of this problem, which became known as the Minimum Spanning Tree (MST) problem, is as follows. Recall that a spanning tree T of a connected undirected graph G is a acyclic connected subgraph of G which includes all edges. In Czech language, a spanning tree is called "kostra grafu", which literally translates as a "skeleton" (or a "frame") of a graph. In any connected undirected graph with a cycle there are multiple possible spanning trees. If there are costs/weights on the edges of the graph, then the cost or weight of a tree is the sum of costs of its edges. Returning to the electrification example, the vertices of the graphs are town, the edges are potential power lines connecting towns, and costs/weights on the edges are distances between town, which translate into costs of connecting these towns by a power line.

More formally, the minimum spanning tree problem asks, given a (connected undirected) graph G with costs (weights) on its edges that are positive numbers, to produce a spanning tree that has a cost minimal among all spanning trees. That is, given an undirected connected graph $G = (V, E)$ with n vertices and m edges $e_1, e_2, \ldots, e_m$, where $c(e_i) =$ "cost of edge e_i", we want to find a minimum cost spanning tree T: a spanning tree T such that $\Sigma_{e_i \in T} c(e_i)$ is minimal among all spanning trees. Note that there can be multiple minimum spanning trees (for example, if G has a cycle on which all edges have the same cost), but the cost of a minimum spanning tree will always be the same.

1.2 Three algorithms for the MST problem

There are several algorithms that are used to solve this problem. The following three are greedy algorithms with the same underlying idea: starting from a situation where every vertex is isolated, add cheapest edges one at a time, until the spanning tree is built. However, they choose the next edge in different ways.

The first is *Kruskal's algorithm*, which operates by sorting edges from smallest to largest cost, and then considering edges in this order, at each step adding the edge if it does not form a cycle with edges already added. This is the simplest algorithm; however, checking whether adding an edge creates a cycle is a somewhat non-trivial operation.

MST-Kruskal(G,c)
Sort the edges: $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
$T = \emptyset$
for $i = 1$ **to** m
  if $T \cup \{e_i\}$ has no cycle **then**
    $T = T \cup \{e_i\}$
return T

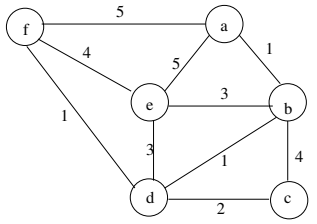
Prim-Jarnik algorithm works on a slightly different premise, growing the tree starting from a specified vertex. It can produce the same tree as Kruskal's algorithm at the end, though (all of these algorithms can produce all minimum spanning trees depending on the sorting order of the edges that have the same value). For the practical application, think of connecting one town in South Moravia to the electric grid, and then extending the grid to more towns, as opposed to building power lines where it is cheap, but with no electricity expected possibly until the very last town is connected. This algorithm uses a priority queue Q to implicitly sort the edges.

MST-Prim-Jarnik(G,c,s)
Initialize $s.key = 0$, $s.pred = null$ and $T = \emptyset$
$\forall v \neq s, v.key = \infty; v.pred = null$
Insert all vertices into Q
while $Q \neq \emptyset$
  $u = ExtractMin(Q)$
  $T = T \cup \{(u.pred, u)\}$
  for each v adjacent to u
    if $v \in Q$ and $c(u, v) < v.key$ **then**
      $v.key = c(u, v); v.pred = u$
return T

Finally, *the algorithm that Borůvka gave* works as follows. For each town, connect it with its nearest neighbour. Now, for each of the resulting groups of towns, connect them to their nearest neighbour... Proceed until there is just one group left. This algorithm has been a basis for several subsequent parallel algorithms.

MST-Boruvkal(G,c)
Make each vertex a separate subtree, $T = \emptyset$
while T is not connected
  Connect each subtree to nearest one
  Add these edges to T
return T

Example 1 *Consider the graph on the right, and suppose that all ties are resolved by considering edges and vertices in lexicographic order. The algorithms will add edges in the following order:*
Kruskal: (a, b), (b, d), (d, f), (c, d), (b, e)
Prim-Jarnik, starting from e: (e, b), (b, a), (b, d), (d, f), (c, d)
Borůvka: All in one round: (a, b), (c, d), (d, b), (e, b), (f, d).



It turns out (miraculously) that in this case, an obvious greedy algorithm (Kruskal's algorithm) always works.

Kruskal's algorithm is the following: first, sort the edges in increasing (or rather nondecreasing) order of costs, so that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$; then, starting with an initially empty tree T, go through the edges one at a time, putting an edge in T if it will not cause a cycle, but throwing the edge out if it would cause a cycle.

1.3 Kruskal's Algorithm:

Sort the edges so that: $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
$T \leftarrow \emptyset$
for $i : 1..m$
(*) if $T \cup \{e_i\}$ has no cycle then
    $T \leftarrow T \cup \{e_i\}$
  end if
end for

But how do we test for a cycle (i.e. execute (*))? After each execution of the loop, the set T of edges divides the vertices V into a collection $V_1 \ldots V_k$ of *connected components*. Thus V is the disjoint union of $V_1 \ldots V_k$, each V_i forms a connected graph using edges from T, and no edge in T connects V_i and V_j, if $i \neq j$.

A simple way to keep track of the connected components of T is to use an array $D[1..n]$ where $D[i] = D[j]$ iff vertex i is in the same component as vertex j. First, initialize $D[i] \leftarrow i$ for all i. To check whether $e_i = [r, s]$ forms a cycle with T, check whether $D[r] = D[s]$. If not, and we therefore want to add e_i to T, we merge the components containing r and s as follows:

$k \leftarrow D[r]$; $l \leftarrow D[s]$
for $j : 1..n$
  if $D[j] = l$ then
    $D[j] \leftarrow k$

The complete program for Kruskal's algorithm then becomes as follows:

Sort the edges so that: $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$
$T \leftarrow \emptyset$
for $i : 1..n$
  $D[i] \leftarrow i$
for $i : 1..m$
  Assign to r and s the endpoints of e_i
  if $D[r] \neq D[s]$ then
    $T \leftarrow T \cup \{e_i\}$
    $k \leftarrow D[r]$
    $l \leftarrow D[s]$
    for $j : 1..n$
      if $D[j] = l$ then
        $D[j] \leftarrow k$

We wish to analyze the running of Kruskal's algorithm, in terms of n (the number of vertices) and m (the number of edges); keep in mind that $n-1 \leq m$ (since the graph is connected) and $m \leq \binom{n}{2} < n^2$. Let us assume that the graph is input as the sequence $n, I_1, I_2, \ldots, I_m$ where n represents the vertex set $V = \{1, 2, \ldots, n\}$,

and I_i is the information about edge e_i, namely the two endpoints and the cost associated with the edge. To analyze the running time, let's assume that any two cost values can be either added or compared in one step. The algorithm first sorts the m edges, and that takes $O(m \log m)$ steps. Then it initializes D, which takes time $O(n)$. Then it passes through the m edges, checking for cycles each time and possibly merging components; this takes $O(m)$ steps, plus the time to do the merging. Each merge takes $O(n)$ steps, but note that the total number of merges is the total number of edges in the final spanning tree T, namely (by the above lemma) $n - 1$. Therefore this version of Kruskal's algorithm runs in time $O(m \log m + n^2)$. Alternatively, we can say it runs in time $O(m^2)$, and we can also say it runs in time $O(n^2 \log n)$. Since it is reasonable to view the size of the input as n, this is a polynomial-time algorithm.

1.4 Union-Find data structure

A better way to implement testing for connectivity is by using the Union-Find data structure. That allows to bring the running time of Kruskal's algorithm down to $O(m \log n)$ time.

Union-Find data structure supports three operations:

1) $MakeUnionFind(S)$: for a set of elements S, returns a Union-Find data structure with each element of S in its own disjoint set. This is used in Kruskal's algorithm when initializing the structure with all vertices of the graph (no edges at that point).

2) $Find(u)$ returns the name of a set containing u (usually a set is named after one of its elements). In Kruskal's algorithm, the check whether $Find(u) == Find(v)$ checks if u and v are in the same connected component.

3) $Union(A, B)$: merge sets A and B (e.g., merge two connected components in Kruskal's).

The array implementation we discussed before is not the most efficient one for this data structure. A better implementation is pointer-based: for every element, a node is created. When two sets are merged in a union operation, the pointer of a node at the root of the tree representing the smaller set is pointed to the root representing the larger set. That is, merging two disjoint nodes results in a tree with a root and one child; merging another disjoint node to it gives a tree with the same root as before, but two children and so on. To know which set is larger, we need an additional field keeping the size of a tree rooted in a given node. In this representation, $Union()$ operation takes constant time (update the pointer of a smaller set's root and the size of the larger set's root). The $Find()$ takes $O(logn)$ time because in the worst case one has to follow the path from a leaf to the root of the tree; however because of every time the smaller tree gets attached to the root of the larger one, if a path increased by 1 the size of the whole tree at least doubled. And $MakeUnionFind()$ takes $O(n)$ time, which is OK since it is only done once.