Kruskal's Algorithm: Correctness Analysis

Valentine Kabanets

February 1, 2011

1 Minimum Spanning Trees: Kruskal's algorithm

A *spanning tree* of a connected graph $G = (V, E)$ is a subset $T \subseteq E$ of the edges such that (V, T) is a tree. (In other words, the edges in T must connect all nodes of G and contain no cycle.)

If a connected G has a cycle, then there is more than one spanning tree for G, and in general G may have exponentially many spanning trees, but each spanning tree has the same number of edges.

We are interested in finding a minimum cost spanning tree for a given connected graph G, assuming that each edge e is assigned a *cost* $c(e)$. (Assume for now that the cost $c(e)$ is a nonnegative real number.) In this case, the cost $c(T)$ is defined to be the sum of the costs of the edges in T. We say that T is a *minimum cost spanning tree* (or an optimal spanning tree) for G if T is a spanning tree for G, and given any spanning tree T' for G, $c(T) \leq c(T')$.

Given a 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. 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.

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

2 Correctness of Kruskal's Algorithm

It is not immediately clear that Kruskal's algorithm yields a spanning tree at all, let alone a minimum cost spanning tree. We will now prove that it does in fact produce an optimal

spanning tree. To show this, we reason that after each execution of the loop, the set T of edges can be expanded to an optimal spanning tree using edges that have not yet been considered. Hence after termination, since all edges have been considered, T must itself be a minimum cost spanning tree.

We can formalize this reasoning as follows:

Definition 1. *A set T of edges of G is* promising after stage i *if T can be expanded to a optimal spanning tree for G using edges from $\{e_{i+1}, e_{i+2}, \ldots, e_m\}$. That is, T is promising after stage i if there is an optimal spanning tree T_{opt} such that $T \subseteq T_{opt} \subseteq T \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$.*

Lemma 1. *For $0 \leq i \leq m$, let T_i be the value of T after i stages, that is, after examining edges $e_1, \ldots, e_i$. Then the following predicate $P(i)$ holds for every i, $0 \leq i \leq m$:*

$$P(i) : T_i \text{ is promising after stage } i.$$

Proof. We will prove this by induction on i.

Base case: $P(0)$ holds because T is initially empty. Since the graph is connected, there exists *some* optimal spanning tree T_{opt}, and $T_0 \subseteq T_{opt} \subseteq T_0 \cup \{e_1, e_2, \ldots, e_m\}$.

Induction step: Let $0 \leq i < m$, and assume $P(i)$. We want to show $P(i+1)$.

Since T_i is promising for stage i, let T_{opt} be an optimal spanning tree such that $T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$.

Case 1: e_{i+1} is rejected. Then $T_i \cup \{e_{i+1}\}$ contains a cycle and $T_{i+1} = T_i$. Since $T_i \subseteq T_{opt}$ and T_{opt} is acyclic, $e_{i+1} \notin T_{opt}$. So $T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.

Case 2: e_{i+1} is added to T_i. Then $T_i \cup \{e_{i+1}\}$ does *not* contain a cycle, so we have $T_{i+1} = T_i \cup \{e_{i+1}\}$. We have two sub-cases here.

Case 2.1: $e_{i+1} \in T_{opt}$. Then we have $T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.

Case 2.2: $e_{i+1} \notin T_{opt}$. We'll show that there is another minimum spanning tree T'_{opt} that witnesses the fact that T_{i+1} is promising. Indeed, consider T_{opt}. Imagine adding an edge e_{i+1} to T_{opt}. This will create a cycle (since T_{opt} was a tree, which is a maximal acyclic graph). In fact, this will create exactly one cycle! (Prove this!)

Claim 1. *The cycle in $T_{opt} \cup \{e_{i+1}\}$ will contain at least one edge from the edges $e_{i+2}, \ldots, e_m$.*

Proof of Claim. T_{opt} contains all edges of T_i, and can be obtained from T_i by adding some edges from the set $\{e_{i+1}, \ldots, e_m\}$. Adding e_{i+1} doesn't create a cycle among the edges of T_i. So, the cycle in $T_{opt} \cup \{e_{i+1}\}$ must contain some of the later edges $e_{i+2}, \ldots, e_m$. □

We will continue with our proof of Lemma 1. Let's pick edge e_j of the cycle, for some $j > i+1$ (which, by Claim 1 above, is always possible). Since the edges are ordered in the increasing order of their costs, we have

$$c(e_j) \geq c(e_{i+1}). \tag{1}$$

Now remove e_j, and get a tree $T_{opt} - \{e_j\} + \{e_{i+1}\}$. It is clearly a tree since we broke the only cycle we had. It is also clearly a spanning tree (as it has the same number of edges as T_{opt}). The cost of this new tree is $c(T_{opt}) - c(e_j) + c(e_{i+1}) \leq c(T_{opt})$, because of inequality (1) above. Hence the cost of the new tree as the same or smaller than that of T_{opt}. Since T_{opt} is optimal, the new tree cannot have smaller cost, and so must in fact have the same cost.

It follows that the new tree is also optimal, and this is our new optimal spanning tree T'_{opt} which we will use.

Since T'_{opt} differs from T_{opt} by having e_{i+1} instead of e_j for some $j > i+1$, we get that $T_{i+1} \subseteq T'_{opt}$, and also that $T'_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$. Hence, T_{i+1} is promising. □

We have now proven Lemma 1. We therefore know that T_m is promising after stage m; that is, there is an optimal spanning tree T_{opt} such that $T_m \subseteq T_{opt} \subseteq T_m \cup \emptyset = T_m$, and so $T_m = T_{opt}$. We can therefore state:

Theorem 1. *Given any connected edge-weighted graph G, Kruskal's algorithm outputs a minimum spanning tree for G.*

3 Discussion of Greedy Algorithms

Before we give another example of a greedy algorithm, it is instructive to give an overview of how these algorithms work, and how proofs of correctness (when they exist) are constructed.

A Greedy algorithm often begins with sorting the input data in some way. The algorithm then builds up a solution to the problem, one stage at a time. At each stage, we have a partial solution to the original problem – don't think of these as solutions to subproblems (although sometimes they are). At each stage we make some decision, usually to include or exclude some particular element from our solution; we never backtrack or change our mind. It is usually not hard to see that the algorithm eventually halts with some solution to the problem. It is also usually not hard to argue about the running time of the algorithm, and when it is hard to argue about the running time it is because of issues involved in the data structures used rather than with anything involving the greedy nature of the algorithm. The key issue is whether or not the algorithm finds an *optimal* solution, that is, a solution that minimizes or maximizes whatever quantity is supposed to be minimized or maximized. We say a greedy algorithm is optimal if it is guaranteed to find an optimal solution for every input.

Most greedy algorithms are not optimal! The method we use to show that a greedy algorithm is optimal (when it is) often proceeds as follows. At each stage i, we define our partial solution to be *promising* if it can be extended to an optimal solution by using elements that haven't been considered yet by the algorithm; that is, a partial solution is promising after stage i if there exists an optimal solution that is consistent with all the decisions made through stage i by our partial solution. We prove the algorithm is optimal by fixing the input problem, and proving by induction on $i \geq 0$ that after stage i is performed, the partial solution obtained is promising. The base case of $i = 0$ is usually completely trivial: the partial solution after stage 0 is what we start with, which is usually the empty partial solution, which of course can be extended to an optimal solution. The hard part is always the induction step, which we prove as follows. Say that stage $i+1$ occurs, and that the partial solution after stage i is S_i and that the partial solution after stage $i+1$ is S_{i+1}, and we know that there is an optimal solution S_{opt} that extends S_i ; we want to prove that there is an optimal solution S'_{opt} that extends S_{i+1} . S_{i+1} extends S_i by making only one decision; if S_{opt} makes the same decision, then it also extends S_{i+1}, and we can just let $S'_{opt} = S_{opt}$

and we are done. The hard part of the induction step is if S_{opt} does not extend S_{i+1}. In this case, we have to show either that S_{opt} could not have been optimal (implying that this case cannot happen), or we show how to change some parts of S_{opt} to create a solution S'_{opt} such that

- S'_{opt} extends S_{i+1}, and
- S'_{opt} has value (cost, profit, or whatever it is we're measuring) at least as good as S_{opt}, so the fact that S_{opt} is optimal implies that S'_{opt} is optimal.

For most greedy algorithms, when it ends, it has constructed a solution that cannot be extended to any solution other than itself. Therefore, if we have proven the above, we know that the solution constructed must be optimal.