CS 6901 (Applied Algorithms) – Lecture 8

Antonina Kolokolova

October 7, 2014

0.1 Kruskal's algorithm correctness and running time

Recall the first of the algorithms for Minimum Spanning Tree problem that we considered last class, Kruskal's algorithm. This is a classic example of a greedy algorithm, that is, an algorithm that works by considering its items one by one, making a decision about each one, and never reversing the decision. As greedy algorithms are usually very fast, they are often used in practice. In vast majority of cases, though, a greedy heuristic does not produce an optimal solution. In this class we will show that Kruskal's algorithm indeed does give an optimal solution, and describe the general framework for proving correctness of greedy algorithms.

First, let us analyse the running time of Kruskal's algorithm.

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

But how do we test for a cycle? 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. So our initialization becomes:

$T \leftarrow \emptyset$
for $i : 1..n$
  $D[i] \leftarrow i$
end for

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] ← k
    end if
end for
```

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

```
Sort the edges so that: c(e_1) ≤ c(e_2) ≤ ... ≤ c(e_m)
T ← ∅
for i : 1..n
    D[i] ← i
end for
for i : 1..m
    Assign to r and s the endpoints of e_i
    if D[r] ≠ D[s] then
       T ← T ∪ {e_i}
       k ← D[r]
       l ← D[s]
       for j : 1..n
           if D[j] = l then
              D[j] ← k
           end if
       end for
    end if
end for
```

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 m, this is a polynomial-time algorithm.

0.2 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 $O(\log n)$ 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.

By making this data structure a little fancier with the path compression heuristic (that is, after looking up each vertex, connect all vertices on the path from that vertex to the root directly to the root of the tree), and using amortized analysis, it is possible to bring the time to do m operations down to $O(m\alpha(n))$, where $\alpha(n)$ is an inverse of the Ackerman function which grows so slowly that it is ≤ 4 for all practical purposes, that is, for n small enough to fit in our universe.

0.3 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; this will be our loop invariant. 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 is promising after stage i.

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

For the 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
$T_i \subseteq T_{opt} \subseteq T_i \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$. If 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
$T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.

Now consider the case that $T_i \cup \{e_{i+1}\}$ does *not* contain a cycle, so we have $T_{i+1} = T_i \cup \{e_{i+1}\}$. If $e_{i+1} \in T_{opt}$, then we have $T_{i+1} \subseteq T_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$.
So assume that $e_{i+1} \notin T_{opt}$. Then according to the Exchange Lemma below (letting T_1 be T_{opt} and T_2 be T_{i+1}), there is an edge $e_j \in T_{opt} - T_{i+1}$ such that $T'_{opt} = T_{opt} \cup \{e_{i+1}\} - \{e_j\}$ is a spanning tree. Clearly $T_{i+1} \subseteq T'_{opt} \subseteq T_{i+1} \cup \{e_{i+2}, \ldots, e_m\}$. It remains to show that T'_{opt} is optimal. Since $T_{opt} \subseteq T_i \cup \{e_{i+1}, e_{i+2}, \ldots, e_m\}$ and $e_j \in T_{opt} - T_{i+1}$, we have $j > i+1$. So (because we sorted the edges) $c(e_{i+1}) \leq c(e_j)$, so $c(T'_{opt}) = c(T_{opt}) + c(e_{i+1}) - c(e_j) \leq c(T_{opt})$. Since T_{opt} is optimal, we must in fact have $c(T'_{opt}) = c(T_{opt})$, and T'_{opt} is optimal.

This completes the proof of the above lemma, except for the Exchange Lemma.

Lemma 2 *(Exchange Lemma) Let G be a connected graph, let T_1 be any spanning tree of G, and let T_2 be be a set of edges not containing a cycle. Then for every edge $e \in T_2 - T_1$ there is an edge $e' \in T_1 - T_2$ such that $T_1 \cup \{e\} - \{e'\}$ is a spanning tree of G.*

Proof:
Let T_1 and T_2 be as in the lemma, and let $e \in T_2 - T_1$. Say that $e = [u, v]$. Since there is a path from u to v in T_1, $T_1 \cup \{e\}$ contains a cycle C, and it is easy to see that C is the only cycle in $T_1 \cup \{e\}$. Since T_2 is acyclic, there must be an edge e' on C that is not in T_2, and hence $e' \in T_1 - T_2$. Removing a single edge of C from $T_1 \cup \{e\}$ leaves the resulting graph acyclic but still connected, and hence a spanning tree. So $T_1 \cup \{e\} - \{e'\}$ is a spanning tree of G. □

We have now proven Lemma 4. 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, Kruskals algorithm outputs a minimum spanning tree for G.*

0.4 A Greedy Algorithm for Scheduling Jobs with Deadlines and Profits

To see another, a little more involved proof of a correctness of a greedy algorithm, recall the Scheduling with Deadlines and Profits that we discussed earlier.

The setting is that we have n jobs, each of which takes unit time, and a processor on which we would like to schedule them in as profitable a manner as possible. Each job has a profit associated with it, as well as a deadline; if the job is not scheduled by the deadline, then we don't get the profit. Because each job takes the same amount of time, we will think of a Schedule S as consisting of a sequence of job "slots" $1, 2, 3, \ldots$ where $S(t)$ is the job scheduled in slot t.
(If one wishes, one can think of a job scheduled in slot t as beginning at time $t-1$ and ending at time t, but this is not really necessary.)

More formally, the input is a sequence $(d_1, g_1), (d_2, g_2), \cdots, (d_n, g_n)$ where g_i is a nonnegative real number representing the profit obtainable from job i, and $d_i \in \mathbb{N}$ is the deadline for job i; it doesn't hurt to assume that $1 \leq d_i \leq n$. (The reason why we can assume that every deadline is less than or equal to n is because even if some deadlines were bigger, every feasible schedule could be "contracted" so that no job was placed in a slot bigger than n.)

Definition 2 *A* schedule *S is an array: $S(1), S(2), ..., S(n)$ where $S(t) \in \{0, 1, 2, \cdots n\}$ for each $t \in \{1, 2, \cdots, n\}$.*

The intuition is that $S(t)$ is the job scheduled by S in slot t; if $S(t) = 0$, this means that no job is scheduled in slot t.

Definition 3 *S is* feasible *if*
(a) If $S(t) = i > 0$, then $t \leq d_i$. (Every scheduled job meets its deadline)
(b) If $t_1 \neq t_2$ and $S(t_1) \neq 0$, then $S(t_1) \neq S(t_2)$. (Each job is scheduled at most once.)

We define the *profit* of a feasible schedule S by
$P(S) = g_{S(1)} + g_{S(2)} + ... + g_{S(n)}$, where $g_0 = 0$ by definition.

Goal: Find a feasible schedule S whose profit $P(S)$ is as large as possible; we call such a schedule *optimal*.

We shall consider the following greedy algorithm. This algorithm begins by sorting the jobs in order of decreasing (actually nonincreasing) profits. Then, starting with the empty schedule, it considers the jobs one at a time; if a job can be (feasibly) added, then it is added to the schedule in the latest possible (feasible) slot.

Greedy:
Sort the jobs so that: $g_1 \geq g_2 \geq \ldots \geq g_n$
for $t : 1..n$
    $S(t) \leftarrow 0$ {Initialize array $S(1), S(2), ..., S(n)$}
end for
for $i : 1..n$
    Schedule job i in the latest possible free slot meeting its deadline;
    if there is no such slot, do not schedule i.
end for

Example. Input of **Greedy**:

Job i:	1	2	3	4	Comments
Deadline d_i:	3	2	3	1	(when job must finish by)
Profit g_i:	9	7	7	2	(already sorted in order of profits)

Initialize $S(t)$:

t	1	2	3	4
$S(t)$	0	0	0	0

Apply **Greedy**: Job 1 is the most profitable, and we consider it first. After 4 iterations:

t	1	2	3	4
$S(t)$	3	2	1	0

Job 3 is scheduled in slot 1 because its deadline $t = 3$, as well as slot $t = 2$, has already been filled.

$P(S) = g_3 + g_2 + g_1 = 7 + 7 + 9 = 23$.

Theorem 2 *The schedule output by the greedy algorithm is optimal, that is, it is feasible and the profit is as large as possible among all feasible solutions.*

We will prove this using our standard method for proving correctness of greedy algorithms.
We say feasible schedule S' *extends* feasible schedule S iff for all t $(1 \leq t \leq n)$,
if $S(t) \neq 0$ then $S'(t) = S(t)$.

Definition 4 *A feasible schedule is* promising after stage i *if it can be extended to an optimal feasible schedule by adding only jobs from* $\{i+1, \cdots, n\}$.

Lemma 3 *For* $0 \leq i \leq n$, *let* S_i *be the value of* S *after* i *stages of the greedy algorithm, that is, after examining jobs* $1, \cdots, i$. *Then the following predicate* $P(i)$ *holds for every* i, $0 \leq i \leq n$:

$P(i)$: S_i *is promising after stage* i.

This Lemma implies that the result of **Greedy** is optimal. This is because $P(n)$ tells us that the result of **Greedy** can be extended to an optimal schedule using only jobs from $\emptyset$. Therefore the result of **Greedy** must be an optimal schedule.

Proof of Lemma: To see that $P(0)$ holds, consider any optimal schedule S_{opt}. Clearly S_{opt} extends the empty schedule, using only jobs from $\{1, \cdots, n\}$.

So let $0 \leq i < n$ and assume $P(i)$. We want to show $P(i+1)$. By assumption, S_i can be extended to some optimal schedule S_{opt} using only jobs from $\{i+1, \cdots, n\}$.

Case 1: Job $i+1$ cannot be scheduled, so $S_{i+1} = S_i$.
Since S_{opt} extends S_i, we know that S_{opt} does not schedule job $i+1$. So S_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$.

Case 2: Job $i+1$ is scheduled by the algorithm, say at time t_0 (so $S_{i+1}(t_0) = i+1$ and t_0 is the latest free slot in S_i that is $\leq d_{i+1}$).

Subcase 2A: Job $i+1$ occurs in S_{opt} at some time t_1 (where t_1 may or may not be equal to t_0).

Then $t_1 \leq t_0$ (because S_{opt} extends S_i and t_0 is as large as possible) and $S_{opt}(t_1) = i+1 = S_{i+1}(t_0)$.

If $t_0 = t_1$ we are finished with this case, since then S_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$. Otherwise, we have $t_1 < t_0$. Say that $S_{opt}(t_0) = j \neq i+1$. Form S'_{opt} by interchanging the values in slots t_1 and t_0 in S_{opt}. Thus $S'_{opt}(t_1) = S_{opt}(t_0) = j$ and $S'_{opt}(t_0) = S_{opt}(t_1) = i+1$. The new schedule S'_{opt} is feasible (since if $j \neq 0$, we have moved job j to an earlier slot), and S'_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$. We also have $P(S_{opt}) = P(S'_{opt})$, and therefore S'_{opt} is also optimal.

Subcase 2B: Job $i+1$ does not occur in S_{opt}.

Define a new schedule S'_{opt} to be the same as S_{opt} except for time t_0, where we define $S'_{opt}(t_0) = i+1$. Then S'_{opt} is feasible and extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$.

To finish the proof for this case, we must show that S'_{opt} is optimal. If $S_{opt}(t_0) = 0$, then we have $P(S'_{opt}) = P(S_{opt}) + g_{i+1} \geq P(S_{opt})$. Since S_{opt} is optimal, we must have $P(S'_{opt}) = P(S_{opt})$ and S'_{opt} is optimal. So say that $S_{opt}(t_0) = j$, $j > 0$, $j \neq i+1$. Recall that S_{opt} extends S_i using only jobs from $\{i+1, \cdots, n\}$. So $j > i+1$, so $g_j \leq g_{i+1}$. We have $P(S'_{opt}) = P(S_{opt}) + g_{i+1} - g_j \geq P(S_{opt})$. As above, this implies that S'_{opt} is optimal. □

We still have to discuss the running time of the algorithm. The initial sorting can be done in time $O(n \log n)$, and the first loop takes time $O(n)$. It is not hard to implement each body of the second loop in time $O(n)$, so the total loop takes time $O(n^2)$. So the total algorithm runs in time $O(n^2)$. Using a more sophisticated data structure one can reduce this running time to $O(n \log n)$, but in any case it is a polynomial-time algorithm.