CSC373— Algorithm Design, Analysis, and Complexity — Spring 2016

Solution Sketches for Tutorial Exercise 1: Greedy Algorithms

---

1. Truck Driver's Problem. Prove that no optimal solution for the Truck Driver's Problem (see lecture notes on Greedy Algorithms, pp 29-32) backtracks to a gas station that it has already been passed.

That is, suppose $\mathcal{O}$ is any optimal solution. Assume it uses exactly q stops at gas stations along the way (the minimum number possible). Suppose these stops are at the distances $\{f_k\}_{k=1}^{q}$ down the road. Prove that $f_{k+1} > f_k$ for all $k = 1, \ldots, q-1$.

Solution The point here was just for them to get the practice of clearly formulating (in math) and argument that should be obvious.

Draw a picture of the general idea (below). Note that a picture isn't a proof.

Proof by contradiction.

Suppose $\{f_k\}_{k=1}^{q}$ is an optimal solution (i.e., the number of gas station stops q is as small as possible within the constraints of the problem. And suppose $f_{r+1} < f_r$ for some r with $1 \leq r < q$.

In a student's proof, ideally they should take care with boundary cases (i.e., $r = 1$ and $r = q-1$ or q). Mention these, but don't go into detail.

For the case $1 < r < q-1$, consider the alternative set of breakpoints $F' = \{f_k\}_{k=1}^{r} \cup \{f_k\}_{k=r+2}^{q}$. Individually these two subsets of breakpoints satisfy the constraint that $f_{k+1} - f_k \leq C$, since the original f_k's satisfy this.

The only remaining step to check is from f_r to f_{r+2}. Note $f_{r+2} - f_r < f_{r+2} - f_{r+1}$ since, by assumption, $f_{r+1} < f_r$. Moreover, $f_{r+2} - f_{r+1} \leq C$ since the the original breakpoints are a feasible solution. Therefore $f_{r+2} - f_r < C$. It follows that F' is a feasible set of $q-1$ breakpoints, but this contradicts the optimality of F.

2. Certificate for Minimum Max-Lateness. Consider the problem of minimizing the maximum lateness for a set of jobs, as discussed on pp.17-22 of the lecture notes. Given a general instance of this problem, and the corresponding optimal solution $\mathcal{O}$ that you have computed (say, with the greedy algorithm described in the lecture notes), imagine that you wish to show your boss (who is intelligent but not technically inclined) that there is no better solution than your optimal solution $\mathcal{O}$. That is, given the input data for the problem, namely $\{(t_j, d_j)\}_{j=1}^{n}$ where t_j indicates the processing time required for the j^{th} job and d_j is that job's deadline, derive a simple formula to convince your boss that no solution can have a maximum lateness that is less than that of your optimal solution. Such a formula is called a **certificate** for the optimality of your solution.

Hint #1: If the maximum lateness for your optimal solution $\mathcal{O}$ is bigger than zero, how can you prove to your boss that no solution can have a maximum lateness of zero?

Hint #2: Make use of your optimal solution $\mathcal{O}$ to select your certificate.

Solution

BTW One thing that is important to the problem statement is that the duration of all jobs must all be non-negative, i.e., $t_j \geq 0$ for all j.

In your optimal solution $\mathcal{O}$, produced by sorting the jobs by deadline and then using greedy, find the finish time t at which the maximum lateness is achieved (if there is a tie, then use the first such job). Suppose this

is the j^{th} job and that it has the deadline d_j. The certificate is then

$$T(d_j) \equiv \sum_{\{i|d_i \leq d_j\}} t_i = d_j + (\text{maximum lateness of } \mathcal{O}) \tag{1}$$

where $\equiv$ denotes a definition.

That is, the total time, $T(d_j)$, required by all jobs with deadlines at or sooner than d_j is equal to the maximum lateness found by your solution. $T(d_j)$ is clearly a lower bound on the maximum lateness, and your solution has acheived that lower bound.

3. Greedy Vertex Cover for Trees. Professor Jot has proposed the following algorithm (see the next page) to solve the minimum vertex cover problem, which is itself described below. Decide whether or not Prof. Jot's algorithm is correct when the input graph is a tree. If you believe the algorithm is correct, prove it using an exchange argument (i.e., show that at each stage it is consistent with some optimal solution). Otherwise show a counter-example.

Terminology. An undirected graph $G = (V, E)$, consisting of vertices $v \in V$ and edges $(u, v) \in E$ (with $u \neq v$) is a **tree** if G is connected (i.e., there is a path in G from each vertex $u \in V$ to every other vertex $v \in V$) and there are no cycles in G (i.e., there is no path from a vertex u back to u without retraversing an edge in the opposite direction). A vertex v with at most one edge $(u, v) \in E$ is called a **leaf**. Also, an undirected graph G with no cycles is called a **forest** (in which each connected set is a tree).

A useful property is that, if $G = (V, E)$ is a tree then the number of edges $|E|$ satisfies $|E| = |V| - 1$.

A **vertex cover** is a subset $C \subseteq V$ such that for every edge $(u, v) \in E$ either $u \in C$ or $v \in C$ (or both). That is, the vertices in C "cover" (at least one end of) every edge in E.

We wish to find a **minimum-sized vertex cover** C. For example, if G is a simple chain, then the minimum size vertex cover can be formed by using every other vertex along the chain (skipping the first). Or if G is star shaped, with one center vertex connected to every other vertex, and these other vertices are all leaves, then the minimum vertex cover consists of just that center vertex.

```
[S] = MinTreeVertexCover(V, E)
% Return a vertex cover of minimum possible size for
% the input tree T = (V, E).
% Precondition: T is a valid undirected tree

Initialize F ← (V, E) and S ← { }.
While the forest F is not empty:
    Find a leaf vertex u in F.
    If there is no edge in F which includes u as an endpoint:
        Delete u from F.
    Else:
        Let (u, v) be such an edge in F.
        Set S ← S ⋃ {v}.
        Remove u, v, (u, v), and all other edges which include v from F.
    End if
End while
return S
```

Solution. Draw a few examples of the execution of the algorithm.

Fortunately for the Prof. his algorithm is correct.

Describe the Loop Invariant below, and you might illustrate the key arguments in the inductive step with an example. You may not have time to write out the whole proof.

Proof. First note that the algorithm terminates, since it removes either one or two vertices from W each iteration, and W initially has the finite size $|V|$.

The key idea is to prove that the algorithm is promising at each stage. That is, after k iterations (including $k = 0$, for the start case), we wish to prove that the following loop invariant is true.

Loop Invariant $L(k)$: Let $F = (W, E_F)$ be the current forest in the algorithm (of vertices and edges that are yet to be processed), and S be the subset of vertices selected so far. Then there exists a minimum sized vertex cover C, of the original graph $T = (V, E)$, such that $S = C \cap (V \backslash W)$. That is, S is promising. Moreover, the edges that have been removed so far, specifically, edges in $E \backslash E_F$, all have at least one endpoint in S.

Note that, the algorithm terminates on iteration k^* when the forest F is empty, and the correctness of the algorithm would follow from $L(k^*)$ for this last iteration.

We are left with proving $L(k)$ by induction, on $k = 0, 1, \ldots, k^*$.

The base case is $k = 0$. Note $L(0)$ follows from the existence of a minimal cover C.

Assume k is some non-negative integer and the algorithm has completed k iterations, and F is not empty. Moreover, assume $L(k)$ is true. We need to prove $L(k + 1)$.

It is useful to define $W^r = V \backslash W$, which is the set of vertices removed from V by the end of the k^{th} iteration. Below, let C denote the optimal solution described in $L(k)$, with $C \cap W^r = S$.

Consider $L(k + 1)$. Since there must be a leaf in a forest, one of the two conditions in the loop body must hold.

First, suppose that u is an isolated vertex in F. By $L(k)$, all the edges in the original set E that have an endpoint at u are already covered by vertices in S (their other endpoints must be in W^r). Therefore, u can be safely removed, preserving this edge covering property of S for all edges with on endpoint in $W^r \cup \{u\}$.

Moreover, if C were to include the vertex u, then it can be shown that it could be removed without destroying the vertex cover property. This is impossible because C is assumed to be a minimal vertex cover. Therefore, with S remaining unchanged this iteration, we still have $S = C \cap (W^r \cup \{u\})$. Therefore $L(k+1)$ is true in this case.

Alternatively, suppose the conditions for the else clause are satisfied. That is, for the leaf u there exists a unique (because it is a leaf) edge (u, v) in E_F which has u as its endpoint. It follows that any other edge $(u, w) \in E$ must have $w \in W^r$ and $L(k)$ ensures that this edge is already covered by S. Therefore it must be the case that $w \in S \cap C$ for all such edges $(w, u) \in E$ with $w \neq v$.

The edge (u, v) is therefore the only edge in E having u as an endpoint which is not covered already by S. Note that it must be covered by a vertex in C. Therefore C must contain either u or v.

If $v \in C$, then $L(k+1)$ follows. Otherwise, as we argued above, we must have $u \in C$. Consider the set $C' = (C \backslash \{u\}) \cup \{v\}$ (which involves "an exchange" of u for v). We saw above that the vertices in $C \cap W^r$ already cover all the edges in E ending at u, except for the one edge (u, v). Since $C \cap W^r = C' \cap W^r$ and $v \in C'$, it follows that C' covers all the edges in E ending at u. Similarly, since C' contains v, it must cover all edges ending at v. These are the only two types of edges that can change coverage between C and C'. So C' must also be a vertex cover. Moreover, since it is the same size as C, it must be a minimal vertex cover.

The statement $L(k+1)$ and the induction step now follow by using the minimum sized vertex cover C'.