# Homework 3 for CMSC 858E

Due 10/02/2018

Note: problem and page number are for the PDF version provided on course website. In case the problem/section/page numbers are not the same, the corresponding pages are also included in this pdf.

## Problem 1

Solve problem 2.2 on page 59 in *The Design of Approximation Algorithms* .

## Problem 2

Consider Theorem 2.2 on page 37 in *The Design of Approximation Algorithms.* Please give a counter-example for this theorem when we no longer assume all deadlines are non-positive.

## Problem 3

Solve problem 2.5 on page 59 in *The Design of Approximation Algorithms*

## Problem 4

Solve problem 2.6 on page 60 in *The Design of Approximation Algorithms*

## Problem 5

Solve problem 2.10 on page 60 in *The Design of Approximation Algorithms*

**Lemma 2.1:** *For each subset $S$ of jobs,*

$$L^*_{\max} \geq r(S) + p(S) - d(S).$$

*Proof.* Consider the optimal schedule, and view this simply as a schedule for the jobs in the subset $S$. Let job $j$ be the last job in $S$ to be processed. Since none of the jobs in $S$ can be processed before $r(S)$, and in total they require $p(S)$ time units of processing, it follows that job $j$ cannot complete any earlier than time $r(S) + p(S)$. The due date of job $j$ is $d(S)$ or earlier, and so the lateness of job $j$ in this schedule is at least $r(S) + p(S) - d(S)$; hence, $L^*_{\max} \geq r(S) + p(S) - d(S)$. □

A job $j$ is *available* at time $t$ if its release date $r_j \leq t$. We consider the following natural algorithm: at each moment that the machine is idle, start processing next an available job with the earliest due date. This is known as the *earliest due date (EDD) rule.*

**Theorem 2.2:** *The EDD rule is a 2-approximation algorithm for the problem of minimizing the maximum lateness on a single machine subject to release dates with negative due dates.*

*Proof.* Consider the schedule produced by the EDD rule, and let job $j$ be a job of maximum lateness in this schedule; that is, $L_{\max} = C_j - d_j$. Focus on the time $C_j$ in this schedule; find the earliest point in time $t \leq C_j$ such that the machine was processing without any idle time for the entire period $[t, C_j)$. Several jobs may be processed in this time interval; we only require that the machine not be idle for some interval of positive length within it. Let $S$ be the set of jobs that are processed in the interval $[t, C_j)$. By our choice of $t$, we know that just prior to $t$, none of these jobs were available (and clearly at least one job in $S$ is available at time $t$); hence, $r(S) = t$. Furthermore, since only jobs in $S$ are processed throughout this time interval, $p(S) = C_j - t = C_j - r(S)$. Thus, $C_j \leq r(S) + p(S)$; since $d(S) < 0$, we can apply Lemma 2.1 to get that

$$L^*_{\max} \geq r(S) + p(S) - d(S) \geq r(S) + p(S) \geq C_j. \tag{2.1}$$

On the other hand, by applying Lemma 2.1 with $S = \{j\}$,

$$L^*_{\max} \geq r_j + p_j - d_j \geq -d_j. \tag{2.2}$$

Adding inequalities (2.1) and (2.2), we see that the maximum lateness of the schedule computed is

$$L_{\max} = C_j - d_j \leq 2L^*_{\max},$$

which completes the proof of the theorem. □

## 2.2 The $k$-center problem

The problem of finding similarities and dissimilarities in large amounts of data is ubiquitous: companies wish to group customers with similar purchasing behavior, political consultants group precincts by their voting behavior, and search engines group webpages by their similarity of topic. Usually we speak of *clustering* data, and there has been extensive study of the problem of finding good clusterings.

Here we consider a particular variant of clustering, the $k$-center problem. In this problem, we are given as input an undirected, complete graph $G = (V, E)$, with a *distance* $d_{ij} \geq 0$ between each pair of vertices $i, j \in V$. We assume $d_{ii} = 0$, $d_{ij} = d_{ji}$ for each $i, j \in V$, and

**2.2** Prove Lemma 2.8: show that for any input to the problem of minimizing the makespan on identical parallel machines for which the processing requirement of each job is more than one-third the optimal makespan, the longest processing time rule computes an optimal schedule.

**2.3** We consider scheduling jobs on identical machines as in Section 2.3, but jobs are now subject to *precedence constraints*. We say $i \prec j$ if in any feasible schedule, job $i$ must be completely processed before job $j$ begins processing. A natural variant on the list scheduling algorithm is one in which whenever a machine becomes idle, then any remaining job that is *available* is assigned to start processing on that machine. A job $j$ is available if all jobs $i$ such that $i \prec j$ have already been completely processed. Show that this list scheduling algorithm is a 2-approximation algorithm for the problem with precedence constraints.

**2.4** In this problem, we consider a variant of the problem of scheduling on parallel machines so as to minimize the length of the schedule. Now each machine $i$ has an associated speed $s_i$, and it takes $p_j/s_i$ units of time to process job $j$ on machine $i$. Assume that machines are numbered from 1 to $m$ and ordered such that $s_1 \geq s_2 \geq \cdots \geq s_m$. We call these *related* machines.

(a) A *$\rho$-relaxed decision procedure* for an scheduling problem is an algorithm such that given an instance of the scheduling problem and a deadline $D$ either produces a schedule of length at most $\rho \cdot D$ or correctly states that no schedule of length $D$ is possible for the instance. Show that given a polynomial-time $\rho$-relaxed decision procedure for the problem of scheduling related machines, one can produce a $\rho$-approximation algorithm for the problem.

(b) Consider the following variant of the list scheduling algorithm, now for related machines. Given a deadline $D$, we label every job $j$ with the slowest machine $i$ such that the job could complete on that machine in time $D$; that is, $p_j/s_i \leq D$. If there is no such machine for a job $j$, it is clear that no schedule of length $D$ is possible. If machine $i$ becomes idle at a time $D$ or later, it stops processing. If machine $i$ becomes idle at a time before $D$, it takes the next job of label $i$ that has not been processed, and starts processing it. If no job of label $i$ is available, it looks for jobs of label $i+1$; if no jobs of label $i+1$ are available, it looks for jobs of label $i+2$, and so on. If no such jobs are available, it stops processing. If not all jobs are processed by this procedure, then the algorithm states that no schedule of length $D$ is possible.

Prove that this algorithm is a polynomial-time 2-relaxed decision procedure.

**2.5** In the *minimum-cost Steiner tree problem*, we are given as input a complete, undirected graph $G = (V, E)$ with nonnegative costs $c_{ij} \geq 0$ for all edges $(i,j) \in E$. The set of vertices is partitioned into *terminals* $T$ and *nonterminals* (or *Steiner vertices*) $V - T$. The goal is to find a minimum-cost tree containing all terminals.

(a) Suppose initially that the edge costs obey the triangle inequality; that is, $c_{ij} \leq c_{ik} + c_{kj}$ for all $i, j, k \in V$. Let $G[T]$ be the graph induced on the set of terminals; that is, $G[T]$ contains the vertices in $T$ and all edges from $G$ that have both endpoints in $T$. Consider computing a minimum spanning tree in $G[T]$. Show that this gives a 2-approximation algorithm for the minimum-cost Steiner tree problem.

(b) Now we suppose that edge costs do not obey the triangle inequality, and that the input graph $G$ is connected but not necessarily complete. Let $c'_{ij}$ be the cost of the shortest path from $i$ to $j$ in $G$ using input edge costs $c$. Consider running the algorithm above in the complete graph $G'$ on $V$ with edge costs $c'$ to obtain a tree $T'$. To compute a tree $T$ in the original graph $G$, for each edge $(i,j) \in T'$, we add to $T$ all edges in a shortest path from $i$ to $j$ in $G$ using input edge costs $c$. Show that this is still a 2-approximation algorithm for the minimum-cost Steiner tree problem on the original (incomplete) input graph $G$. $G'$ is sometimes called the *metric completion* of $G$.

**2.6** Prove that there can be no $\alpha$-approximation algorithm for the minimum-degree spanning tree problem for $\alpha < 3/2$ unless P = NP.

**2.7** Suppose that an undirected graph $G$ has a Hamiltonian path. Give a polynomial-time algorithm to find a path of length at least $\Omega(\log n/(\log\log n))$.

**2.8** Consider the local search algorithm of Section 2.6 for finding a minimum-degree spanning tree, and suppose we apply a local move to a node whenever it is possible to do so; that is, we don't restrict local moves to nodes with degrees between $\Delta(T)-\ell$ and $\Delta(T)$. What kind of performance guarantee can you obtain for a locally optimal tree in this case?

**2.9** As given in Exercise 2.5, in the Steiner tree problem we are given an undirected graph $G = (V, E)$ and a set of terminals $T \subseteq V$. A Steiner tree is a tree in $G$ in which all the terminals are connected; a non-terminal need not be spanned. Show that the local search algorithm of Section 2.6 can be adapted to find a Steiner tree whose maximum degree is at most $2\,\mathrm{OPT} + \lceil \log_2 n \rceil$, where OPT is the maximum degree of a minimum-degree Steiner tree.

**2.10** Let $E$ be a set of items, and for $S \subseteq E$, let $f(S)$ give the value of the subset $S$. Suppose we wish to find a maximum value subset of $E$ of at most $k$ items. Furthermore, suppose that $f(\emptyset) = 0$, and that $f$ is *monotone* and *submodular*. We say that $f$ is *monotone* if for any $S$ and $T$ with $S \subseteq T \subseteq E$, then $f(S) \le f(T)$. We say that $f$ is *submodular* if for any $S, T \subseteq E$, then

$$f(S) + f(T) \ge f(S \cup T) + f(S \cap T).$$

Show that the greedy $(1 - \frac{1}{e})$-approximation algorithm of Section 2.5 extends to this problem.

**2.11** In the *maximum coverage problem*, we have a set of elements $E$, and $m$ subsets of elements $S_1, \ldots, S_m \subseteq E$, each with a nonnegative weight $w_j \ge 0$. The goal is to choose $k$ elements such that we maximize the weight of the subsets that are covered. We say that a subset is covered if we have chosen some element from it. Thus we want to find $S \subseteq E$ such that $|S| = k$ and that we maximize the total weight of the subsets $j$ such that $S \cap S_j \ne \emptyset$.

(a) Give a $(1 - \frac{1}{e})$-approximation algorithm for this problem.

(b) Show that if an approximation algorithm with performance guarantee better than $1 - \frac{1}{e} + \epsilon$ exists for the maximum coverage problem for some constant $\epsilon > 0$, then every NP-complete problem has a $O(n^{O(\log\log n)})$ time algorithm (Hint: Recall Theorem 1.13).