CSCI 670: Advanced Analysis of Algorithms
1st Assignment

August 2021

1 Introduction

Problem 1 (Activity selection)**.** Assume that there are n activities with start time s_i and finish time t_i. Two activities i and j are not conflicting if their occurrence times are disjoint, i.e. either $t_i < s_j$ or $t_j < s_i$. Design an $O(n \log n)$ algorithm to select the maximum number of non-conflicting activities.

Solution 1. Please see the Section 4.1 of Algorithm Design by Jon Kleinberg and Eva Tardos.

Problem 2. Consider an array of integers $a_1, \dots a_n$. We say a sequence π is a permutation of length n if π contains every element of $\{1, 2, \dots n\}$ exactly once and its length is n. Design an $poly(n)$ algorithm to find a permutation π of length n such that ℓ_1 norm of $|\pi - a|_1$ is the smallest, i.e. $\sum_{i=1}^{n} |\pi_i - a_i|$.

Solution 2. For a given array a, let π_i^* be the index of a_i when the sequence a is sorted in ascending order. It is easy to see that π^* is a permutation. It remains to prove that π^* is the optimal permutation. One can show that for any solution π and indices i, j if $a_i > a_j$ and $\pi_i < \pi_j$, swapping π_i and π_j weakly improves the result. Thus, π^* is an optimal solution. Moreover, one can find such a permutation π^* by sorting the sequence a and a constant number of passes over a. So, we can obtain the optimal permutation to this problem in $O(n \log n)$.

Problem 3 (From Fall 2019)**.** A computer received n tasks, where task i requires t_i seconds for completion. Tasks should be done one by one, without interruptions. For each task, $f_i(t) \geq 0$ defines the penalty applied when the i-th tasks waits in the queue for t seconds. The total penalty is defined as the sum of penalties of all tasks. For each of the following two cases, design an $O(n \log n)$ algorithm that finds an order of processing of tasks such that the total penalty is minimized:

(a) $f_i(t) = c_i t, c_i > 0, i = 1..n.$

(b) $f_i(t) = c_i e^t, c_i > 0, i = 1..n.$

Solution 3. We need to find an ordering of tasks, π, such that it minimizes the total penalty. Let us look at any optimal solution π^*. If task j is just after task i, then we should not be able to improve the total penalty by swapping the order of these two tasks. Let T denote the total time required to complete all tasks that come before task i. Then in the optimal solution π^*, the penalty coming from task i is $f_i(T)$, while the penalty coming from task j is $f_j(T + t_i)$. The total penalty coming from tasks i and j is $f_i(T) + f_j(T + t_i)$. If we swap the tasks, that penalty will become $f_j(T) + f_i(T + t_j)$. Notice that after swapping the penalty coming from all other tasks stays the same. Therefore, from optimality of π^* we get

$$f_i(T) + f_j(T + t_i) \leq f_j(T) + f_i(T + t_j)$$

(a) For this case, Eq. (1) becomes

$$c_i T + c_j (T + t_i) \leq c_j T + c_i (T + t_j) \Longleftrightarrow c_j t_i \leq c_i t_j \Longleftrightarrow \frac{t_i}{c_i} \leq \frac{t_j}{c_j}$$

Therefore, if we denote $v_i \triangleq \frac{t_i}{c_i}$, then in any optimal solution the tasks should be sorted by v_i. It is easy to notice that changing the order of tasks that have the same v_i does not affect the total penalty. Consequently, any sequence of tasks that sorts v values is optimal. If there are no two tasks with equal v_i, there will be a unique optimal solution.

(b) For this case, Eq. (1) becomes

$$c_i e^T + c_j e^{T+t_i} \leq c_j e^T + c_i e^{T+t_j} \Longleftrightarrow c_i + c_j e^{t_i} \leq c_j + c_i e^{t_j} \Longleftrightarrow \frac{e^{t_i} - 1}{c_i} \leq \frac{e^{t_j} - 1}{c_j}$$

After denoting $v_i \triangleq \frac{e^{t_i}-1}{c_i}$, the rest of the proof is the same as that of case (a). In both cases the complexity will be $O(n \log n)$, because we need to sort the tasks.

Problem 4 (From Fall 2019)**.** Consider n factory tasks, each of which has to undergo two processing steps to be completed. The first step of processing of the i-th task takes a_i seconds, while the second step of processing takes b_i seconds. The first step has to be done before starting the second step. At any point of time there can be up to one task undergoing the first processing step, and up to one task undergoing the second processing step. Design an $O(n \log n)$ algorithm that can find a scheduling of tasks minimizing the time of completing all tasks.

Solution 4. In essence, we need to decide the order of tasks for each step. If the order of tasks in the first step is π, then it is optimal to do the second step in the same order. Let us fix one order π and see what happens in the second step. Let δ_i denote the time between finishing the second step of $(i-1)$-th task and starting the second step of the i-th task. The first task will finish its first step at time a_{π_1}, then immediately start its second step. Therefore, $\delta_1 = a_{\pi_1}$. The second step of the first task be finished at time $\delta_1 + b_{\pi_1}$. The second task will finish its first step at time $a_{\pi_1} + a_{\pi_2}$, then will wait for $\delta_2 = \max\{0, (a_{\pi_1} + a_{\pi_2}) - (\delta_1 + b_{\pi_1})\}$ seconds to start its second step, and will finish its second step at time $\delta_1 + b_{\pi_1} + \delta_2 + b_{\pi_2}$. More generally, the i-th task will finish its first step at time $\sum_{j=1}^{i} a_{\pi_j}$, then will wait for $\delta_i = \max\left\{0, \left(\sum_{j=1}^{i} a_{\pi_j|}\right) - \left(\sum_{j=1}^{i-1} \left(\delta_j + b_{\pi_j}\right)\right)\right\}$ seconds to start its second step, and will finish its second step at time $\sum_{j=1}^{i} \left(\delta_j + b_{\pi_j}\right)$. Our task is to select such π that $\sum_{j=1}^{n} \left(\delta_j + b_{\pi_j}\right)$ is minimized, which is equivalent to minimizing $\sum_{j=1}^{n} \delta_j$. Let us understand the form of $\sum_{i=1}^{n} \delta_i$

$$\begin{aligned}
\sum_{i=1}^{n} \delta_i &= \delta_n + \sum_{i=1}^{n-1} \delta_i \\
&= \max\left\{0, \left(\sum_{j=1}^{n} a_{\pi_j}\right) - \left(\sum_{j=1}^{n-1} \left(\delta_j + b_{\pi_j}\right)\right)\right\} + \sum_{i=1}^{n-1} \delta_i \\
&= \max\left\{\sum_{i=1}^{n-1} \delta_i, \left(\sum_{j=1}^{n} a_{\pi_j}\right) - \left(\sum_{j=1}^{n-1} b_{\pi_j}\right)\right\}
\end{aligned}$$

Applying the same trick for $\sum_{i=1}^{n-1} \delta_i$, and repeating the same, we will get that:

$$\sum_{i=1}^{n} \delta_i = \max_{i \in [n]} \{K_i\}$$

where $K_i = \left(\sum_{j=1}^{i} a_{\pi_j}\right) - \left(\sum_{j=1}^{i-1} b_{\pi_j}\right)$

Now let us consider an optimal solution π^* and two neighboring positions, u and $v = u+1$. Since π^* is optimal, we cannot improve the solution by swapping tasks π_u and π_v. If we do swap, we will get new K'_i values, where

$$K'_i = \begin{cases} K_i, & \text{if } i \neq u \text{ and } i \neq v \\ a_{\pi_v} + \left(\sum_{j=1}^{u-1} a_{\pi_j}\right) - \left(\sum_{j=1}^{u-1} b_{\pi_j}\right), & \text{if } i = u \\ a_{\pi_u} + a_{\pi_v} + \left(\sum_{j=1}^{u-1} a_{\pi_j}\right) - \left(\sum_{j=1}^{u-1} b_{\pi_j}\right) - b_{\pi_v}, & \text{if } i = v \end{cases}$$

If $\max\{K'_u, K'_v\} \geq \max\{K_u, K_v\}$ then we cannot improve the solution by swapping tasks π_u and π_v. We

have that:

$$\begin{aligned} \max\{K'_u, K'_v\} \geq \max\{K_u, K_v\} &\iff \max\{a_{\pi_v}, a_{\pi_u} + a_{\pi_v} - b_{\pi_v}\} \geq \max\{a_{\pi_u}, a_{\pi_u} + a_{\pi_v} - b_{\pi_u}\} \\ &\iff a_{\pi_u} + a_{\pi_v} + \max\{-a_{\pi_u}, -b_{\pi_v}\} \geq a_{\pi_u} + a_{\pi_v} + \max\{-a_{\pi_v}, -b_{\pi_u}\} \\ &\iff \max\{-a_{\pi_u}, -b_{\pi_v}\} \geq \max\{-a_{\pi_v}, -b_{\pi_u}\} \\ &\iff -\min\{a_{\pi_u}, b_{\pi_v}\} \geq -\min\{a_{\pi_v}, b_{\pi_u}\} \\ &\iff \min\{a_{\pi_u}, b_{\pi_v}\} \leq \min\{a_{\pi_v}, b_{\pi_u}\} \end{aligned}$$

Summing up, we have that if task j comes right after task i and $\min\{a_i, b_j\} \leq \min\{a_j, b_i\}$, then we cannot improve the solution by swapping these tasks. It is easy to prove that this relation is transitive, i.e. if $\min\{a_i, b_j\} \leq \min\{a_j, b_i\}$ and $\min\{a_j, b_k\} \leq \min\{a_k, b_j\}$, then $\min\{a_i, b_k\} \leq \min\{a_k, b_i\}$. Therefore, the relation defines a linear order between tasks. Our algorithm will sort all tasks according to that relation. To prove that our algorithm finds an optimal answer, it remains to show that there is an optimal answer where tasks are sorted. To see this, we can take any optimal answer and swap two neighboring elements if they are not sorted according to our rule. We have already proven that this cannot increase the total processing. By repeating this swapping step as much as it is possible, we will end up with a sorted ordering without increasing the total processing time. Finally, the order tasks that are equal to each other according to our rule does not matter (we can simply swap them whenever needed), implying that any sorted sequence of tasks is optimal.

Problem 5 (Minimum vertex cover of tree)**.** For a graph $G = (V, E)$, a **vertex cover** of G is a set of vertices that includes at least one endpoint of each edge of G. Finding the minimum vertex cover of a graph is a well-known optimization problem in computer science. This problem is known to be NP-Hard, so that no polynomial-time can solve it unless $P = NP$. Moreover, a proof to Unique Games Conjecture, well-known conjecture in Computer Science, implies that it is hard to approximate within a factor smaller than 2, where such an approximation is a standard algorithm.

In this problem, we restrict our focus to a particular case: trees. Assume that the input graph G is a tree, design a $poly(|V|, |E|)$ algorithm to find a minimum vertex cover of G.

Solution 5. We design a greedy algorithm to find a minimum vertex cover of a tree (or forest in general).

Algorithm 1 Greedy Vertex Cover

Input: A tree $G = (V, E)$.
if $|V| < 2$ **then**
  Output $\emptyset$
else if $|V| = 2$ **then**
  Output $\{v_1\}$ for $V = \{v_1, v_2\}$
else
  Let I be the set of vertices that are incident to a leaf. Set $S \leftarrow S \cup I$.
  Let E' be the set of edges not covered by S, $E' = E \setminus \{(u, v) \in E : S \cap \{u, v\} \neq \emptyset\}$.
  Find optimal vertex cover by recursively calling Algorithm 1 for each connected component of (V, E').
  Let S' be the union of solutions obtained by recursive calls. Set $S \leftarrow S \cup S'$.
end if
Output S.

First of all, observe that we can implement the algorithm in $poly(|V|, |E|)$. So, we need to prove the correctness of the algorithm. Consider the base case, a tree G with $|V| = 2$ consists of a single edge. Thus, picking any of the two vertices yields an optimal vertex cover. Observe that when $|V| > 2$, there exists an internal (non-leaf) vertex. The following lemma is crucial for greedy selection, and the proof is left to the reader.

Lemma 1. For a given tree $G = (V, E)$ with $|V| > 2$, there exists a minimum vertex cover C not containing any leaves. So, C includes all vertices incident to a leaf.

Lemma implies that greedy selection in each recursive call is valid. In each recursive call, we choose a subset of vertices that are extensible to a minimum vertex cover all the time. So, it remains to prove that the union of vertex covers of connected components and vertices found initially consists of a minimum vertex cover. To prove that, we can use the fact that the cardinality of the minimum vertex-cover is a monotone increasing set function of edges.

Problem 6 (Knapsack)**.** Suppose you are given a set of n items with costs c_i and values v_i. The knapsack problem is to find a set of items with maximum total value such that their total cost does not exceed the budget B. Without loss of generality, we can assume that all items meet the budget constraint, i.e. $c_i \leq B$. It is known to be NP-Hard to decide whether we can achieve a value of V or not in all cases. In this problem, we design an approximation algorithm for the knapsack problem.

(a) First, we relax the problem so that we allow to have fractional items. Design a $O(n \log n)$ algorithm to find optimal solutions for the knapsack problem with divisible items. In other words, any fraction of an item can be in the knapsack. Observe that the optimal value of the relaxation is always higher than the optimal value of the original problem.

(b) A feasible solution S said to be 1/2-approximation to knapsack problem if

$$\sum_{i \in S} v_i \geq \frac{1}{2} \cdot OPT$$

where OPT is the total value of items in the optimal solution. Design a $O(n \log n)$ algorithm to knapsack problem that outputs a 1/2-approximate solution in all cases.

Solution 6. (a) Let's state the greedy algorithm for fractional knapsack problem.

One can show that replacing any fraction of an item in the solution with another item (possibly fractional) weakly decreases the total value. We left proof of this claim to the reader.

Algorithm 2 Greedy Fractional Knapsack Algorithm

Compute densities $d_i := \frac{v_i}{c_i}$ for all $i \in \{1, \dots n\}$
Sort items in decreasing order of d_i's. So, v_i and c_i are the value and the cost of item i which are decreasing order of densities.
for i in $\{1, \dots \text{n}\}$ **do**
  Let C be the total cost of items in the current set of solution.
  if $C + c_i \leq B$ **then**
    Add item i to solution.
  else
    Add $\frac{B-C}{c_i}$ fraction of item i to solution.
  end if
end for

(b) Let OPT_f be the optimal value of the relaxed problem. Observe that a feasible solution to the 0-1 knapsack problem is feasible for the fractional knapsack problem, or we can say $OPT_f \geq OPT$.

Let's state an algorithm for 0-1 knapsack problem.

Algorithm 3 0-1 Knapsack Algorithm

Run the Algorithm 2.
Let S be the set of items belonging to solution and k be the index of the item fractionally included to the solution.
if $\sum_{i \in S} v_i \geq v_k$ **then**
  Output S
else
  Output k
end if

The following inequality proves that Algorithm 3 is $\frac{1}{2}$ approximation to 0-1 Knapsack problem.

$$2 \cdot \max \left\{ v_k, \sum_{i \in S} v_i \right\} \geq v_k + \sum_{i \in S} v_i \geq OPT_f \geq OPT.$$

Problem 7 (Submodular Function Definition)**.** Let E be a finite set of elements, and $f : 2^E \to \mathbb{R}$ be a set function. We say that f is submodular if it satisfies one of the following equivalent conditions.

1. $f(T \cup \{e\}) - f(T) \leq f(S \cup \{e\}) - f(S)$ for all $S \subseteq T \subseteq E$ and $e \in E \setminus T$.
2. $f(S) + f(T) \geq f(S \cup T) + f(S \cap T)$ for any $S, T \subseteq E$

Prove that two conditions are equivalent.

Solution 7. • (2 $\implies$ 1): Let $S \subseteq T \subseteq E$ and $e \in E \setminus T$. Define $S' = S \cup \{e\}, T' = T$ and apply the second inequality.

$$\begin{aligned} & f(S') + f(T') \geq f(S' \cup T') + f(S' \cap T') \\ \implies & f(S' \cup T') - f(T') \leq f(S') - f(S' \cap T') \\ \implies & f(T \cup \{e\}) - f(T) \leq f(S \cup \{e\}) - f(S) \end{aligned}$$

- (1 $\implies$ 2): Consider arbitrary $S, T \subseteq E$. Let $\{e_1, \dots e_k\}$ be an enumeration of $S \setminus T$. Define $S_0 = S \cap T, T_0 = T, S_j := S_0 \cup \bigcup_{i=1}^{j} e_i$, and $T_j := T_0 \cup \bigcup_{i=1}^{j} e_j$. Observe that $S_k = S$ and $T_k = S \cup T$. Now, let's apply the first inequality for k times:

$$\begin{aligned} & f(T_{i-1} \cup \{e_i\}) - f(T_{i-1}) \leq f(S_{i-1} \cup \{e_i\}) - f(S_{i-1}) && \forall i \in 1, \dots k \\ \iff & f(T_i) - f(T_{i-1}) \leq f(S_i) - f(S_{i-1}) && \forall i \in 1, \dots k \end{aligned}$$

  Summation of these k inequalities side by side completes the proof.

$$\begin{aligned} & f(T_k) - f(T_0) \leq f(S_k) - f(S_0) \\ \implies & f(T \cup S) - f(T) \leq f(S) - f(S \cap T) \\ \implies & f(T \cup S) + f(S \cap T) \leq f(S) + f(T). \end{aligned}$$

Problem 8. For each of the following functions f, prove it is submodular or explain why it is not.

(a) For every set $S \subseteq \{1, \dots, n\}$, function f is defined as $f(S) = g(|S|)$ where $g : \mathbb{R} \to \mathbb{R}$ is a concave function.

(b) Let G be undirected graph. For every subset S of edges, define $f(S)$ as the cardinality of maximum matchings which is a subset of S. A matching is a collection of edges where two of them does not share a vertex.

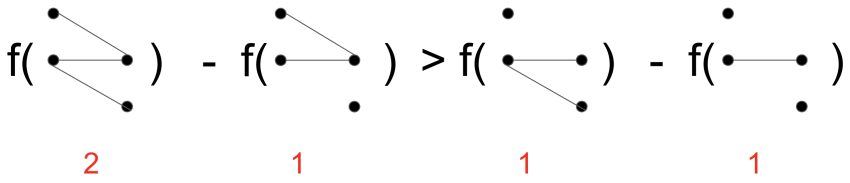
(c) Given a positively-weighted directed graph $G = (V, E)$, for every subset of nodes $S \subseteq V$, define function $f(S)$ as the total weight of edges outgoing from S, i.e.,

$$f(S) = \sum_{(u,v) \in E, u \in S, v \in V \setminus S} \omega_{(u,v)}$$

(d) Given an undirected graph $G = (V, E)$, for every subset of nodes $S \subseteq V$, define function $f(S)$ as the coloring number of S, i.e., minimum number of colors required for coloring nodes in S in a way that every two adjacent nodes have different colors.

Solution 8. (a) First of all, observe that $g(n+1) - g(n) \geq g(n+2) - g(n+1)$ for any non-negative integer n. So, f is a submodular function, since f satisfies the condition given in Problem 7 part (b).

(b) Here is a counter-example:



(c) For any $T \subseteq S \subseteq V$ and $x \in V \setminus S$

$$\begin{aligned} f(T \cup \{x\}) - f(T) &= \sum_{\substack{(x,v) \in E \\ v \in V \setminus (T \cup \{x\})}} \omega_{(x,v)} - \sum_{\substack{(u,x) \in E \\ u \in T}} \omega_{(u,x)} \\ &\geq \sum_{\substack{(x,v) \in E \\ v \in V \setminus (S \cup \{x\})}} \omega_{(x,v)} - \sum_{\substack{(u,x) \in E \\ u \in S}} \omega_{(u,x)} \qquad (T \subseteq S) \\ &= f(S \cup \{x\}) - f(S) \end{aligned}$$

(d) Define the graph $G = (V, E)$ where $V = \{1, 2, 3\}$ and $E = \{\{2, 3\}\}$. Then,

$$f(\{1, 3\}) - f(\{1\}) = 1 - 1 \leq 2 - 1 = f(\{1, 2, 3\}) - f(\{1, 2\}).$$

So, f is not a submodular function.