Homework 3: Greedy Algorithms

Due: October 03, 2023

This homework must be typed in \LaTeX{} and submitted via Gradescope.

Please ensure that your solutions are complete, concise, and communicated clearly. Use full sentences and plan your presentation before your write. Except where indicated, consider every problem as asking for a proof.

Problem 1.

(a) Let \( G = (V, E) \) be a finite, undirected graph. Let \( \ell \) be the collection of \( E \) such that

\[ A \in \ell \iff A \text{ is an acyclic subset of } E \]

Show that \((E, \ell)\) is a matroid.

**Hint:** It might be helpful to consider components.

(b) Let \( w : E \to \mathbb{R}^+ \) be a function that assigns a non-negative weight to each element of \( E \). Let \( \text{IsAcyclic} \) be a predicate that checks whether a subset of \( E \) is acyclic or not. Assume this algorithm runs in \( O(T) \) time.

Design an efficient algorithm to find a maximum weight acyclic subset of \( E \). Analyze its runtime and argue the correctness/optimality.

**Solution.**

(a) To show that \((E, \ell)\) is a matroid we have to verify that

1. \( E \) is a finite set.
2. For every \( B \in \ell \), if \( A \subseteq B \) then \( A \in \ell \).
3. For every \( A, B \in \ell \), if \(|A| < |B|\) then there exists an \( x \in B \setminus A \) such that \( A \cup \{x\} \in \ell \).

The first condition clearly holds since \( E \) is the set of edges of a finite graph.

The second condition also holds since any subset of an acyclic set of edges is also acyclic. Suppose, for the sake of contradiction, that a subset of an acyclic set contained a cycle. Then the original acyclic set contains a cycle, a contradiction!

The final condition (i.e. the extension property) is slightly tricky to verify. Let \( A, B \in \ell \) be two acyclic subsets such that \(|A| < |B|\). Consider the induced subgraphs \( G_A = (V, A) \) and \( G_B = (V, B) \).

Note that the number of connected components in \( G_A \) is strictly larger than the number of connected components in \( G_B \). To convince yourself, start with the completely disconnected graph \((V, \emptyset)\). As we add an edge one at a time to this graph the total number of connected components must either decrease or stay the same. Notice, however, for \( G_A \) or \( G_B \) the added edge does not form a cycle. If the edge were to connect two vertices in
the same connected component, it would necessarily form a cycle. Therefore, the edge
must connect two disconnected components. Thus, the number of connected components
strictly decreases.

Since $G_B$ has fewer connected components than $G_A$, by the pigeonhole principle
there exist two vertices that are a part of the same component in $G_B$ but different
components in $G_A$. Therefore, there is an edge on the unique path from these vertices
in $G_B$ not contained in $G_A$. The addition of this edge to $A$ will result in a new, larger
acyclic subset of $E$.

(b) We’ll modify the algorithm presented in lectures 5 and 6 using the knowledge that
$(E, \ell, w)$ is a weighted matroid.

\begin{algorithm}
\caption{Max Weight Acyclic Subset of a Graph}
\begin{algorithmic}
\State \textbf{Input:} A set of edges for a graph, $E$
\State An edge weight function, $w$
\State \textbf{Output:} A maximum weight acyclic subset of $E$ with respect to $w$
\Function{MaxWeightSubset}{$E$, $w$}
\State Initialize $A$ to be an empty set
\State \textbf{Sort}($E$) such that $E$ is in decreasing order with respect to $w$
\For{$e \in E$}
\State \textbf{if} $\text{IsAcyclic}(A \cup e)$ \textbf{then}
\State \hspace{1em} Add $e$ to $A$
\EndFor
\State \Return $A$
\EndFunction
\end{algorithmic}
\end{algorithm}

The correctness of this algorithm in the case that $(E, \ell, w)$ is a weighted matroid
was shown in class!

We can sort our input set of edges in $O(n \log n)$ time using a sorting algorithm
like \textsc{MergeSort} or \textsc{QuickSort}. We then loop through each of the $n$ edges in the set
and perform an $O(T)$ check to see if the resulting set is acyclic. Therefore, the overall
runtime of this algorithm is $O(n \log n + nT)$.
Problem 2. Suppose you are given a set \( S = \{a_1, a_2, \ldots, a_n\} \) of tasks, where task \( a_i \) requires \( p_i \) units of processing time to complete, once it has started. You have access to a computer to run these tasks one at a time. Let \( c_i \) be the completion time of task \( a_i \), i.e. the time at which task \( a_i \) completes processing. Your goal is to minimize the average completion time:

\[
\frac{1}{n} \sum_{i=1}^{n} c_i
\]

For example, suppose there are two tasks, \( a_1 \) and \( a_2 \), with \( p_1 = 3 \) and \( p_2 = 5 \), and consider the schedule in which \( a_2 \) runs first, followed by \( a_1 \). Then, \( c_2 = 5 \), \( c_1 = 8 \), and the average completion time is 6.5.

(a) Give an algorithm that schedules the tasks to minimize the average completion time. Each task must run non-preemptively, that is, once task \( a_i \) is started, it must run continuously for \( p_i \) units of time. Prove that your algorithm minimizes the average completion time, and prove the running time of your algorithm.

(b) Suppose now that the tasks are not available at once. Each task has a release time \( r_i \) before which it is not available to be processed. Suppose also that we allow preemption, meaning a task can be suspended and restarted later.

For example, a task \( a_i \) with processing time \( p_i = 6 \) may start running at time 1 and be preempted at time 4. It can then resume at time 10 but be preempted at time 11 and finally resume at time 13 and complete at time 15. Task \( a_i \) has run for a total of 6 time units, but its running time has been divided into three pieces. We say that the completion time of \( a_i \) is 15.

Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Solution.

(a) Consider the following algorithm:

**Algorithm 2 Non-Preemptive Task Scheduling**

**Input:** A set of tasks \( S \) with their processing times

**Output:** A schedule (list) of tasks that minimize the average completion time

1: function NonPreemptiveTaskScheduling(S)
2: return \( S \) sorted with respect to processing time in increasing order

To show this solution is optimal, we proceed by way of contradiction. Suppose, there exists a schedule \( S = [a_1, \ldots, a_n] \) that has a shorter average completion time than our above greedy solution. Since this schedule is not the greedy solution we know that there exists two tasks \( a_i \) and \( a_j \) such that \( i < j \) but \( p_i > p_j \).

Because task \( a_i \) takes longer than \( a_j \), we assert that the solution can be improved by first scheduling \( a_j \) and then later scheduling \( a_i \). One can verify this by comparing the
total completion times of the supposed optimal schedule and the schedule obtained by swapping \( p_i \) and \( p_j \).

The original schedule has a total completion time given by:

\[
p_1 + (p_1 + p_2) + \cdots + (p_1 + \cdots + p_n) = \sum_{k=1}^{n}(n - k - 1)p_k
\]

By swapping the positions of tasks \( a_i \) and \( a_j \), the new total completion time becomes:

\[
(j - i)p_j - (j - i) - p_i + \sum_{k=1}^{n}(n - k - 1)p_k
\]

Since we assumed that \( i < j \) and \( p_i > p_j \), it follows that \( (j - i)p_j < (j - i)p_i \). Therefore, the swapped schedule has a shorter total completion time and therefore, a shorter average completion time. This contradicts the assumption that the original schedule was optimal. Since this algorithm simply returns a sorted list, the runtime of this algorithm is \( O(n \log n) \).

(b) Consider the following algorithm:

**Algorithm 3 Preemptive Task Scheduling**

**Input:** A set of tasks \( S \) with their processing and release times

**Output:** A schedule (list) of tasks that minimize the average completion time

1: function PREEMPTIVE_TASK_SCHEDULING(\( S \))
2: Sort the tasks in the order of increasing release time
3: currentTime \( \leftarrow \) the first (minimum) release time
4: Make a priority queue, availableTasks, with all tasks whose release time is less than the currentTime, where the queue is ordered in ascending processing time
5: while currentTime \( \leq \) the final release time do
6: Schedule the next available task (top of priority queue) in availableTasks modifying its processing time to be the difference between the current time and the next release time. If the task is finished remove it from the queue entirely.
7: currentTime \( \leftarrow \) the next task release time
8: Add the newly released tasks to availableTasks
9: return a list containing the scheduled tasks

The same concept of always doing the shortest task applies from part (a). The difference is that we have to re-determine which task is shortest whenever new tasks become available. The while loop is structured such that whenever new tasks become available, they are inserted into a priority queue. This ensures that at all times, the shortest available task is scheduled.

First, the list of \( n \) tasks is sorted by release time. This takes \( O(n \log n) \) time. Then, the while loop performs the tasks. Each task gets added to the queue at most once, and removed from the queue at most once. Since adding and removing from a priority queue can be performed in \( O(\log n) \), adding and removing all elements will take \( O(n \log n) \) time. Therefore, the overall runtime is \( O(n \log n) \).