

16.5 A task-scheduling problem

An interesting problem that can be solved using matroids is the problem of optimally scheduling unit-time tasks on a single processor, where each task has a deadline, along with a penalty that must be paid if the deadline is missed. The problem looks complicated, but it can be solved in a surprisingly simple manner using a greedy algorithm.

A **unit-time task** is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete. Given a finite set S of unit-time tasks, a **schedule** for S is a permutation of S specifying the order in which these tasks are to be performed. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on.

The problem of **scheduling unit-time tasks with deadlines and penalties for a single processor** has the following inputs:

- a set $S = \{a_1, a_2, \dots, a_n\}$ of n unit-time tasks;
- a set of n integer **deadlines** d_1, d_2, \dots, d_n , such that each d_i satisfies $1 \leq d_i \leq n$ and task a_i is supposed to finish by time d_i ; and
- a set of n nonnegative weights or **penalties** w_1, w_2, \dots, w_n , such that we incur a penalty of w_i if task a_i is not finished by time d_i and we incur no penalty if a task finishes by its deadline.

We are asked to find a schedule for S that minimizes the total penalty incurred for missed deadlines.

Consider a given schedule. We say that a task is **late** in this schedule if it finishes after its deadline. Otherwise, the task is **early** in the schedule. An arbitrary schedule can always be put into **early-first form**, in which the early tasks precede the late tasks. To see this, note that if some early task a_i follows some late task a_j , then we can switch the positions of a_i and a_j , and a_i will still be early and a_j will still be late.

We similarly claim that an arbitrary schedule can always be put into **canonical form**, in which the early tasks precede the late tasks and the early tasks are scheduled in order of monotonically increasing deadlines. To do so, we put the schedule into early-first form. Then, as long as there are two early tasks a_i and a_j finishing at respective times k and $k + 1$ in the schedule such that $d_j < d_i$, we swap the positions of a_i and a_j . Since a_j is early before the swap, $k + 1 \leq d_j$. Therefore, $k + 1 \leq d_i$, and so a_i is still early after the swap. Task a_j is moved earlier in the schedule, so it also stays early after the swap.

The search for an optimal schedule thus reduces to finding a set A of tasks that are to be early in the optimal schedule. Once A is determined, we can create the actual schedule by listing the elements of A in order of monotonically increasing deadline, then listing the late tasks (i.e., $S - A$) in any order, producing a canonical ordering of the optimal schedule.

We say that a set A of tasks is **independent** if there exists a schedule for these tasks such that no tasks are late. Clearly, the set of early tasks for a schedule forms an independent set of tasks. Let \mathcal{I} denote the set of all independent sets of tasks.

Consider the problem of determining whether a given set A of tasks is independent. For $t = 0, 1, 2, \dots, n$, let $N_t(A)$ denote the number of tasks in A whose deadline is t or earlier. Note that $N_0(A) = 0$ for any set A .

Lemma 16.12

For any set of tasks A , the following statements are equivalent.

1. The set A is independent.
2. For $t = 0, 1, 2, \dots, n$, we have $N_t(A) \leq t$.
3. If the tasks in A are scheduled in order of monotonically increasing deadlines, then no task is late.

Proof Clearly, if $N_t(A) > t$ for some t , then there is no way to make a schedule with no late tasks for set

A , because there are more than t tasks to finish before time t . Therefore, (1) implies (2). If (2) holds, then (3) must follow: there is no way to "get stuck" when scheduling the tasks in order of monotonically increasing deadlines, since (2) implies that the i th largest deadline is at most i . Finally, (3) trivially implies (1).

Using property 2 of [Lemma 16.12](#), we can easily compute whether or not a given set of tasks is independent (see [Exercise 16.5-2](#)).

The problem of minimizing the sum of the penalties of the late tasks is the same as the problem of maximizing the sum of the penalties of the early tasks. The following theorem thus ensures that we can use the greedy algorithm to find an independent set A of tasks with the maximum total penalty.

Theorem 16.13

If S is a set of unit-time tasks with deadlines, and I is the set of all independent sets of tasks, then the corresponding system (S, I) is a matroid.

Proof Every subset of an independent set of tasks is certainly independent. To prove the exchange property, suppose that B and A are independent sets of tasks and that $|B| > |A|$. Let k be the largest t such that $N_t(B) = N_t(A)$. (Such a value of t exists, since $N_0(A) = N_0(B) = 0$.) Since $N_n(B) = |B|$ and $N_n(A) = |A|$, but $|B| > |A|$, we must have that $k < n$ and that $N_j(B) > N_j(A)$ for all j in the range $k + 1 \leq j \leq n$. Therefore, B contains more tasks with deadline $k + 1$ than A does. Let a_i be a task in $B - A$ with deadline $k + 1$. Let $A' = A \setminus \{a_i\}$.

We now show that A' must be independent by using property 2 of [Lemma 16.12](#). For $0 \leq t = k$, we have $N_t(A') = N_t(A) = t$, since A is independent. For $k < t \leq n$, we have $N_t(A') = N_t(B) = t$, since B is independent. Therefore, A' is independent, completing our proof that (S, I) is a matroid.

By [Theorem 16.11](#), we can use a greedy algorithm to find a maximum-weight independent set of tasks A . We can then create an optimal schedule having the tasks in A as its early tasks. This method is an efficient algorithm for scheduling unit-time tasks with deadlines and penalties for a single processor. The running time is $O(n^2)$ using GREEDY, since each of the $O(n)$ independence checks made by that algorithm takes time $O(n)$ (see [Exercise 16.5-2](#)). A faster implementation is given in [Problem 16-4](#).

[Figure 16.7](#) gives an example of a problem of scheduling unit-time tasks with deadlines and penalties for a single processor. In this example, the greedy algorithm selects tasks a_1, a_2, a_3 , and a_4 , then rejects a_5 and a_6 , and finally accepts a_7 . The final optimal schedule is

$\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$,

which has a total penalty incurred of $w_5 + w_6 = 50$.

	Task						
a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Figure 16.7: An instance of the problem of scheduling unit-time tasks with deadlines and penalties for a single processor.

Exercises 16.5-1

Solve the instance of the scheduling problem given in [Figure 16.7](#), but with each penalty w_i replaced by 80