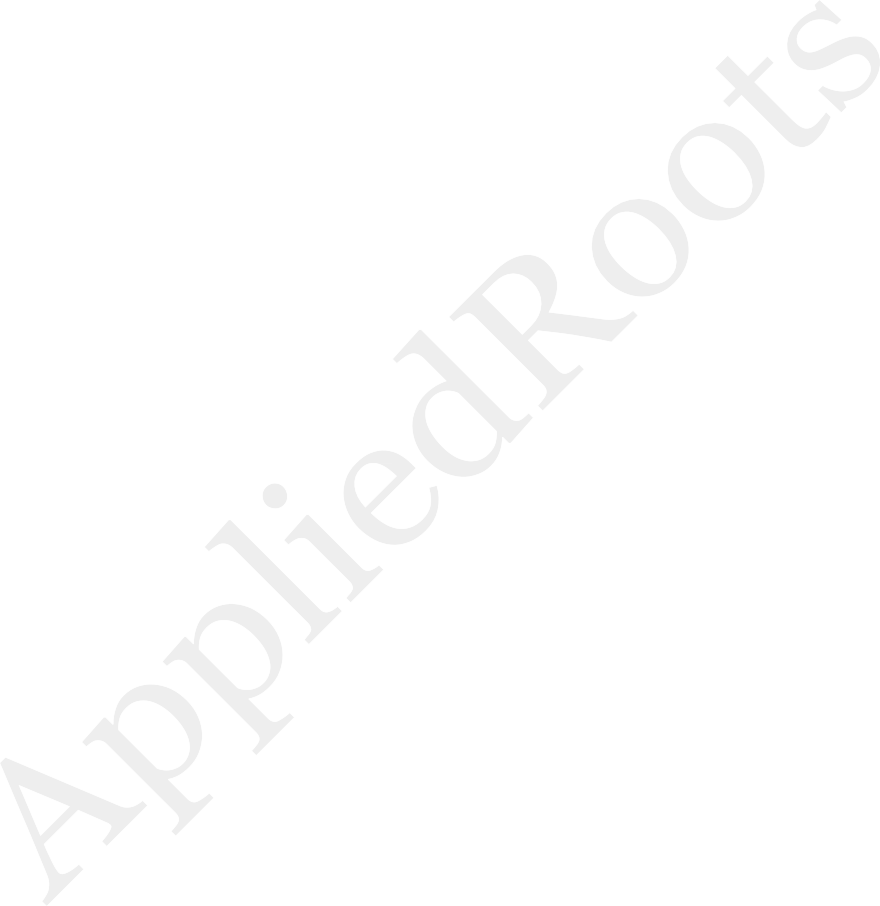
17.5 A task-scheduling problem

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 and 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* = {1, 2, . . . , *n*} of *n* unit-time tasks;

a set of *n* integer ***deadlines*** *d*1, *d*2, . . . , *dn,* such that each *di* satisfies 1 *di n* and task *i* is supposed to finish by time *di;* and

a set of *n* nonnegative weights or ***penalties*** *w*1,*w*2, . . . , *wn*, such that a

penalty *wi* is incurred if task *i* is not finished by time *di* and no penalty is incurred 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 *x* follows some late task *y*, then we can switch the positions of *x* and *y* without affecting *x* being early

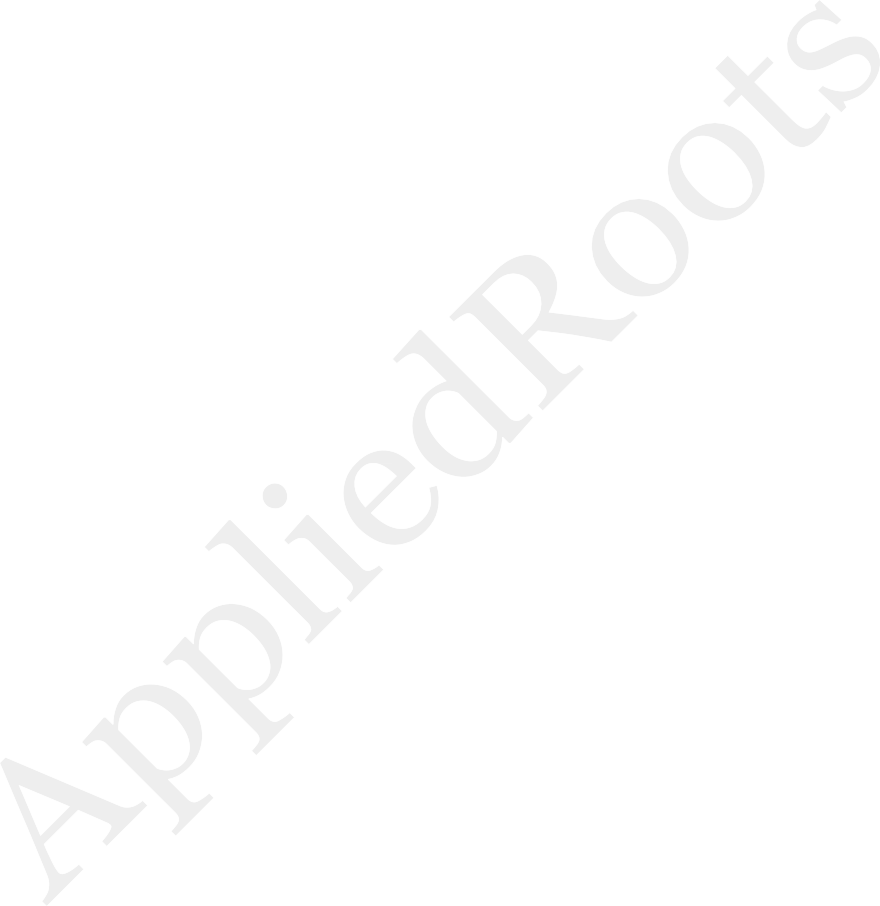
or *y* being late.

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 nondecreasing deadlines. To do so, we put the schedule into early-first form. Then, as long as there are two early tasks *i* and *j* finishing at respective times *k* and *k* + 1 in the schedule such that *dj* < *di*, we swap the positions of *i* and *j*. Since task *j* is early before the swap, *k* + 1 *dj*. Therefore, *k* + 1 < *di*, and so task *i* is still early after the swap. Task *j* is moved earlier in the schedule, so it also still 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 nondecreasing 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  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* = 1, 2, . . . , *n*, let *Nt*(*A*) denote the number of tasks in *A* whose deadline is *t* or earlier.

Lemma 17.11

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

1. The set *A* is independent.
2. For *t* = 1, 2, . . . , *n*, we have *Nt*(*A*) *t*.
3. If the tasks in *A* are scheduled in order of nondecreasing deadlines, then no task is late.

***Proof*** Clearly, if *Nt*(*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 nondecreasing deadlines, since (2) implies that the *i*th largest deadline is at most *i*. Finally, (3) trivially implies (1).

Using property 2 of Lemma 17.11, we can easily compute whether or not a given set of tasks is independent (see Exercise 17.5-2).

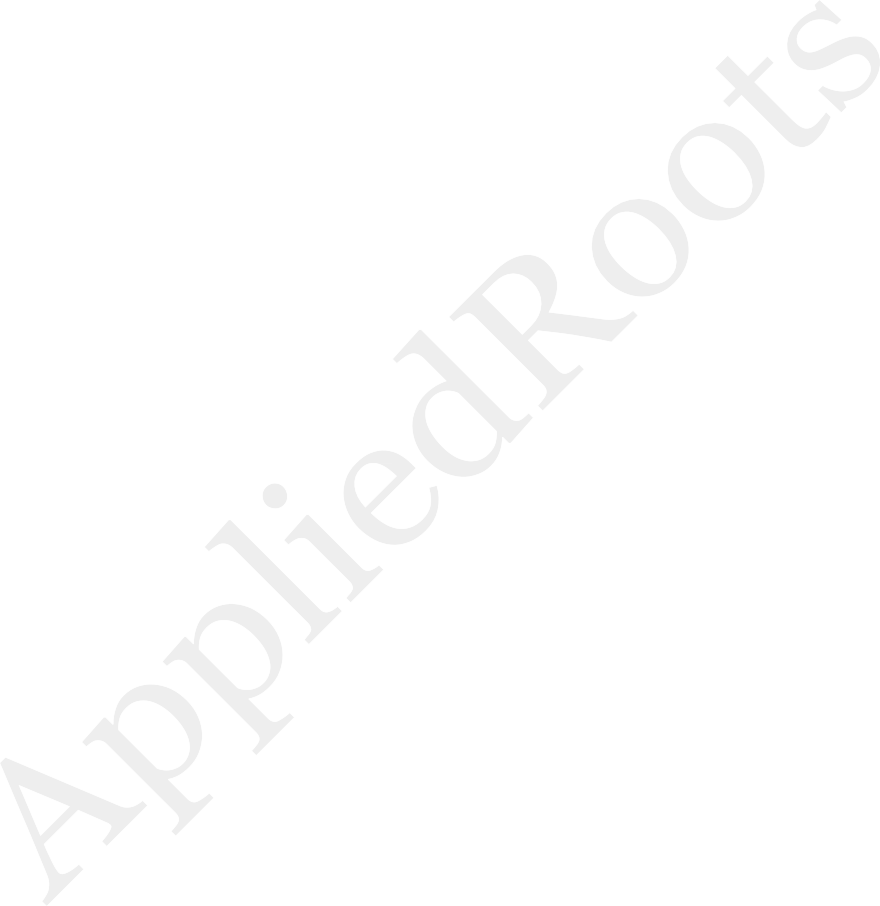
The problem of minimizing the sum of the penalties of the late tasks is the same as the problem of maximizing the sum of the penalties of the early tasks. The following theorem thus ensures that we can use the greedy algorithm to find an independent set *A* of tasks with the maximum total penalty.

Theorem 17.12

If *S* is a set of unit-time tasks with deadlines, and  is the set of all independent sets of tasks, then the corresponding system  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 *Nt*(*B*) *Nt*(*A*). Since *Nn*(*B*) = |*B| and* Nn*(*A*) = |*A|, but |*B| > |*A|, we must have that *k* < *n* and that *Nj*(*B*) > *Nj*(*A*) for all *j* in the range *k* + 1 *j* n*. Therefore,* B *contains more tasks with deadline* k *+ 1 than* A *does. Let* x *be a task in* B *-* A *with deadline* k *+ 1. Let* A*'* = *A* = *A* {*x*}.

We now show that *A*' must be independent by using property 2 of Lemma 17.11. For 1 *t k*, we have *Nt*(*A*'*) =* Nt*(*A*) t,*since *A* is independent. For *k* < *t n*, we have *Nt*(*A*'*)* Nt*(*B*) t*, since *B* is independent. Therefore, *A'* is independent, completing our proof that  is a matroid.

By Theorem 17.10, 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 17.5-2). A faster implementation is given in Problem 17-3.

Task

1 2 3 4 5 6 7

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *di* 4 | 2 | 4 3 | 1 | 4 | 6 |
| *wi* 70 | 60 | 50 40 | 30 | 20 | 10 |

**Figure 17.7 An instance of the problem of scheduling unit-time tasks with deadlines and penalties for a single processor.**

Figure 17.7 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 1, 2, 3, and 4, then rejects tasks 5 and 6, and finally accepts task 7. The final optimal schedule is

2, 4, 1, 3, 7, 5, 6 ,

which has a total penalty incurred of *w*5 +*w*6 = 50.