CS 6901 (Applied Algorithms) – Lecture 3

Antonina Kolokolova

September 16, 2014

1 Representative problems: brief overview

In this lecture we will look at several problems which, although look somewhat similar to each other, have quite different complexity. We will look at some algorithm design techniques that can be used to solve some of these problems (much of the course will be devoted to in-depth study of these techniques), and at some problems for which no efficient algorithms are known.

1.1 Scheduling problem

Let's start with the following problem. Suppose there is a list of jobs (tasks), for example to be scheduled on a processor. Each task takes a certain amount of time and gives a certain amount of profit (gain). Moreover, to get the benefit of the task it has to finish by its deadline. We want to design an algorithm which would take a list of such jobs and create a correct schedule for which the sum of gains of scheduled jobs is maximal. More precisely, define the problem as follows. Let t_i be the time a job i takes (its duration), d_i be the deadline by which it has to finish, and g_i a reward for completing this job on time. So a job $(3, 5, 10)$ has to start at time 2 or earlier, and if this job is scheduled and finishes on time, there is a gain of 10. In a correct schedule no two jobs overlap, and each job finishes by its deadline.

Scheduling with deadlines, profits and durations
Input: $(t_1, d_1, g_1) \ldots, (t_n, d_n, g_n)$
Output: A schedule (array) S maximizing $\Sigma_{i,S[i]\geq 0} g_i$, where $S(i) = s_i$ is the starting time of i^{th} job; let $s_i = -1$ if a job cannot be scheduled.

When stated in this way, the Scheduling problem is NP-hard, and we do not know of any efficient, that is, polynomial-time algorithm for it.

1.2 NP-hardness

What does it mean for a problem to be NP-hard? A problem is NP-hard if every problem in the complexity class NP can be reduced to (think "disguised as") this problem. In particular, solving any problem in NP would only take as much time as the NP-hard problem times, roughly, the increase in size of the input description that disguising created (plus the time it takes to do the disguising).

The complexity class NP itself is the class of all decision problems with efficiently verifiable solutions, where "efficiently" = "in time polynomial in the length of the input in binary". A problem is a decision problem if its output is a yes/no answer. For example, if a scheduling problem is asking "is there a schedule with profit

at least K?" is a decision problem, whereas "compute the best possible schedule" is not a decision problem. In complexity theory, most of the time decision problems are considered – if nothing else, it avoids the case of problems with very long solutions.

Now, a problem is NP-complete if it is both in NP (that is, it is a decision problem with polynomial-time verifiable solutions) and is NP-hard (that is, any problem in NP can be disguised to look like it in polynomial time). Sometimes people use terms "NP-hard" and "NP-complete" as if they are synonyms (usually to mean "hard"), but this is not quite correct: a problem can easily be NP-hard without being NP-complete. So, the scheduling, where the output is the optimal set of schedule, would be NP-hard, but not NP-complete. For such problems (although definitely not for all NP-hard problems) it holds that if P=NP, then they are solvable in polynomial time; this applies to all NP-complete problems, but not all NP-hard ones (though it does apply to "find an optimal solution" type of problems). Also, all such efficiently verifiable problems are solvable given exponential time (even when the amount of available space is still polynomial). However, there are problems solvable in exponential time which are NP-hard, but provably not solvable in polynomial time.

To illustrate what we mean by "disguising" one problem as another, consider the Simple Knapsack problem. Suppose you are trying to fly a prospecting mission to Arctic, and the aircraft can only take certain amount of equipment, and you are choosing what to bring with you to maximize usefulness, and yet staying within the aircraft weight limits.

Simple Knapsack
Input: Weights $w_1, \ldots, w_n$; capacity C
Output: A set $S \subseteq \{1..n\}$ such that $\Sigma_{i \in S} w_i \leq C$ and, additionally, $\Sigma_{i \in S} w_i$ is maximal possible over all such S

This problem is a simplified example of the Knapsack problem, in which each item has both a weight and a profit.

Now, we can disguise Simple Knapsack as Scheduling as follows. Make an instance of scheduling in which job i will take time $t_i = w_i$, have deadline $d_i = C$ and gain $g_i = w_i$. That is, if the input to Simple Knapsack is $(w_1, \ldots, w_n, C)$, then the corresponding input to Scheduling is $(w_1, C, w_1), \ldots (w_n, C, w_n)$. For example, if an instance of Simple Knapsack was $(50, 50, 51; 100)$, the corresponding instance of Scheduling is $(50, 100, 50), (50, 100, 50), (51, 100, 51)$.

Now suppose there is a solution S' to the scheduling instance; in our example, let $S[1] = 50$, $S[2] = 0$, $S[3] = -1$. We can recover the solution to the original Knapsack problem by taking all i such that $S[i] \geq 0$; in our example, it will be $(1, 2)$, corresponding to the first two items in $(50, 50, 51)$.

Interestingly enough, Knapsack itself is NP-hard (and a version of the Simple Knapsack asking if there is a set with sum at least B for some bound B is NP-complete). If you have done a course on complexity theory, you might have seen the SubsetSum problem (asking if, given a set of numbers, there is a subset summing up to a specified number t). You can see that this problem can be reduced to Simple Knapsack decision problem by treating these numbers as weights, and asking if there is a subset of size both at most t (so set $C = t$) and at least t (so $B = t$ as well). But we just showed, by disguising Simple Knapsack as Scheduling, that Scheduling is even harder, because if we could solve Scheduling, we can automatically solve Simple Knapsack in its disguised form. (Note that in complexity theory we would use a more precise version of a reduction as opposed to the disguising we just described; however the idea would be quite similar).

Problems of these types do arise in practice, though: what can we do then? One possibility is to use heuristics (we will look at them later in the course) and hope that they work. Another is to analyze the problem carefully to see if maybe only a special case needs to be solved, and if that special case is easier than the general problem. Yet another approach is to design an algorithm which, although not guaranteed to produce an optimal solution, can produce a solution which is "not too far" from the optimal, and can do it in a reasonably fast way.

1.3 Restricting the length of deadlines: dynamic programming

Dynamic programming technique works by iteratively computing an array with values of subproblems of a problem (in our case, a subproblem will be scheduling jobs to finish by the time d and using the first i jobs). We can design a dynamic programming algorithm for the Scheduling problem with time complexity $O(n \cdot \max_i d_i)$, that is, the number of jobs times the largest deadline. In general, it would not be efficient, as writing down deadlines on the order of 2^n only takes n bits, and we can still have the input of length comparable to n if there are few such large deadlines. However, if we add a restriction $\forall i, d_i \leq n^k$ for some fixed constant k that does not depend on the size of the instance, then our $O(n \cdot \max_i d_i) = O(n \cdot n^k) = O(n^{k+1})$ algorithm is polynomial time. Thus, such a natural restriction can allow for an efficient algorithm for scheduling. Also, as capacity in the Simple Knapsack is disguised as deadlines, the same algorithm would solve Simple Knapsack efficiently for $C \leq n^k$.

However, if k is large, this algorithm would still run fairly slow. Is it possible to make a different restriction to do better? What if we do not have a variety of deadlines, if all d_i are the same, would it make the algorithm run faster?

1.4 Scheduling with deadlines and profits: greedy algorithms

Now, suppose that we make all t_i to be the same; say $t_1 = \cdots = t_n = 1$. That is, the problem will look as follows (for simplicity, we will view the schedule here as an array of time slots).

Scheduling with Deadlines and Profits
Input: $(d_1, g_1), (d_2, g_2) \ldots, (d_n, g_n)$
Output: A schedule (array) S, where $|S| \leq n$, $S(i) = j$ means that the job i is scheduled in time slot j, no job is scheduled more than once and $\forall i$, if $S(i) = j$ then $d_j \geq i$, such that $\Sigma_{j:\exists i S(i)=j} g_j$ is maximized.

Then it is possible to design a much faster algorithm, a greedy algorithm running in time $O(n \log n)$, to solve the problem.

SchedulingWithDeadlinesAndProfits$[(d_1, g_1), \ldots, (d_n, g_n)]$

```
1  Sort the jobs so that: g_1 ≥ g_2 ≥ ... ≥ g_n
2  for t = 1 to n
3      S(t) ← 0                    ▷ {Initialize array S(1), S(2), ..., S(n)}
4  for i = 1 to n
5      Schedule job i in the latest possible free slot meeting its deadline;
       ▷ if there is no such slot then do not schedule i.
```

Example 1 *Input:*

Job i:	*1*	*2*	*3*	*4*	*Comments*
Deadline d_i*:*	*3*	*1*	*3*	*1*	*(when job must finish by)*
Profit g_i*:*	*9*	*7*	*7*	*2*	*(already sorted in order of profits)*

Initialize $S(t)$*:*

t	*1*	*2*	*3*	*4*
$S(t)$	*0*	*0*	*0*	*0*

Apply the algorithm above: Job 1 is the most profitable, and we consider it first. After 4 iterations:

t	*1*	*2*	*3*	*4*
$S(t)$	*2*	*3*	*1*	*0*

Job 2 goes into slot 1, since this is the only available slot where it can meet its deadline. Job 3 is scheduled in slot 2 because its deadline is 3*, but slot* 3 *has already been filled. And job 4 is not scheduled.*

$P(S) = g_3 + g_2 + g_1 = 7 + 7 + 9 = 23$.

Here again if we use the array implementation of the schedule, then checking if there is an available slot can take linear amount of time (think of all jobs having the same deadline = n). In that case, the running time of the algorithm is dominated by the scheduling part. However, if this part is done using union-find data structure, then the running time of the algorithm is dominated by the sorting, giving us $O(n \log n)$ time.

For this algorithm to work though, having all durations the same is crucial. Consider for example a set of the following three jobs: $(10, 10, 10), (5, 10, 9), (5, 10, 8), (1, 5, 1)$. The greedy algorithm would pick the first job as it has the largest profit, but the optimal solution consists of second and third. Similarly, picking the shortest job or a job with the earliest deadline does not work.

1.5 Approximation algorithms

So since Simple Knapsack is NP-hard, we cannot hope to have a polynomial-time algorithm solving it exactly without resolving the P vs. NP problem (which currently seems out of reach, and it's million dollar prize from the Clay Mathematical Institute is still unclaimed). However, there is a greedy algorithm that can get us "close enough" – it is guaranteed to produce a solution which is at least 1/2 of the optimal solution (there is also a dynamic programming algorithm that can get us nearly as close as we like, with the "closeness" as a parameter, but we will not discuss it here).

SimpleKnapsack-1/2approx$[w_1, \ldots, w_n, C]$

```
1  Sort the weights so that: w_1 ≥ w_2 ≥ ... ≥ w_n
2  S ← ∅; M ← 0                          ▷ Initialize the set and the sum
3  for t = 1 to n
4      if w_i + M ≤ C then
5          S ← S ∪ {i}; M ← M + w_i      ▷ Add i^th element to knapsack
6  return S
```

We first sort the weights in decreasing (or rather nonincreasing order): $w_1 \geq w_2 \geq ... \geq w_d$ We then try the weights one at a time, adding each if there is room. Call this resulting estimate $\overline{M}$.

It is easy to find examples for which this greedy algorithm does not give the optimal solution; for example weights $\{501, 500, 500\}$ with $C = 1000$. Then $\overline{M} = 501$ but $M = 1000$. However, this is just about the worst case:

Lemma 1 $\overline{M} \geq \frac{1}{2}M$

Proof: We will show that if $\overline{M} \neq M$, then $\overline{M} > \frac{1}{2}C$. Since $C \geq M$, the Lemma follows.

Suppose, for the sake of contradiction, that $\overline{M} \leq C$. Since $M \neq \overline{M}$, there is at least one element i in the optimal solution which is not in S produced by our algorithm. Since i is in the optimal solution, $w_i \leq C$. Now, consider two cases. Either $w_i > 1/2C$; but in this case, the algorithm would have considered it early in the run (before putting in everything it did add). If at that point i did not fit, there must have been something already in the knapsack; and that something must have been even larger, i.e., with weight $w_j \geq w_i > 1/2C$. But then $\overline{M} \geq 1/2C$. So the $w_i < 1/2C$. But then, if there is more than $1/2C$ space left in the knapsack, w_i could easily fit, and so the algorithm would have put it there while considering it.

Therefore, in both cases assuming that $M \neq \overline{M}$ and $M \leq 1/2C$ lead to a contradiction. Indeed, the only way $\overline{M} \leq 1/2C$ is possible is when sum of all weights is $\leq 1/2C$. But in that case, there is no solution better than the sum of all weights, and the algorithm gets it. $\square$

The running time of this algorithm is dominated by the sorting's $O(n \log n)$ time ; if the elements are given to us already sorted, then the running time of the algorithm is $O(n)$.