CS 6783 (Applied Algorithms) – Lecture 3

Antonina Kolokolova

January 14, 2013

1 Representative problems: brief overview of the course

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 with deadlines and profits: greedy algorithms

Let's start with the following problem. Suppose there is a list of jobs (tasks) to be scheduled on a processor, each of which needs to be finished by its (given) deadline. In the simplest form, suppose that all jobs are equal; also, their lengths are the same. We want to design an algorithm which would take a list of such jobs and create a schedule with as many jobs as possible. More precisely, define the problem as

Scheduling with Deadlines
Input: $d_1, \ldots, d_n$
Output: A schedule (array) S with the maximal number of scheduled jobs, 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 $d_j \geq i$.

The following simple algorithm solves this problem. Consider the jobs one by one and schedule each job in the latest available time slot before its deadline. For example, if the input had jobs with deadlines $d_1 = 2, d_2 = 2, d_3 = 1, d_4 = 4$, then the schedule will be $S = [1, 2, 0, 4]$, where 0 denotes that there is no job scheduled at that time. You can convince yourself that this algorithm works and produces the optimal solution (we will talk about proving correctness of such algorithms later in the course). This algorithm looks at every job just once; however, the simplest implementation using an array for S would need up to n steps to check for the latest available time slot, giving quadratic in n running time (as we will write it, $O(n^2)$ – we will cover this notation in the next lecture). With the use of a more complex data structure "union-find" (sometimes called "disjoint set"), the running time could be made almost linear in n.

Now consider a slightly more complex version of the problem: suppose that jobs are not equal, and though they still take the same amount of time, there is also a profit (gain) value assigned to each job. Now, naturally, the task would be to maximize the profit (sum of the gains of individual jobs in the schedule) as opposed to the number of jobs.

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.

This time the algorithm is very similar, except now it does matter in which order we consider the jobs. Since we want to give higher-gain jobs "more chance" at being scheduled, the algorithm starts by sorting the jobs in order of diminishing gain, then proceeds as above.

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

1 Sort the jobs so that: $g_1 \geq g_2 \geq \ldots \geq g_n$
2 **for** $t = 1$ **to** n
3 $\quad S(t) \leftarrow 0$ $\quad \triangleright$ {Initialize array $S(1), S(2), ..., S(n)$}
4 **for** $i = 1$ **to** n
5 $\quad$ Schedule job i in the latest possible free slot meeting its deadline;
$\quad \triangleright$ 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*	*2*	*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*
$S(t)$	*3*	*2*	*1*	*0*

Job 3 is scheduled in slot 1 because its deadline $t = 3$*, as well as slot* $t = 2$*, has already been filled.*

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

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.

1.2 Scheduling with deadlines, profits and durations: dynamic programming

Now let's add one more parameter to our jobs. Suppose that in addition to deadline d_i and profit g_i each job has a duration t_i; a job has to finish on or before its deadline. So if a job has a deadline 5 and a duration 3, the latest it can start is 2.

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, where now $S(i) = s_i$ is the starting time of i^{th} job; let $s_i = -1$ if a job cannot be scheduled.

In this case, a greedy algorithm does not work anymore; 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. So we will resort to a more complex technique, dynamic programming.

Dynamic programming technique, which we will consider as our second algorithm design paradigm, 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). This algorithm is fairly fast when the number of jobs is comparable with the largest deadline. However, it could be the case that deadlines are very large (e.g., on the order of 2^n, where n is the number of jobs). In this case, this algorithm does not run fast (that is, in polynomial time) anymore. Moreover, we can show that this problem, as stated, is an NP-hard problem and therefore is unlikely to be solvable by an efficient algorithm.

1.3 NP-hardness of Scheduling and Knapsack, and a greedy algorithm for 1/2 approximation for Simple Knapsack

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, say, a version of Knapsack where the output is the optimal set of weights 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 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.

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. For example, 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, just like we had jobs with deadlines and profits, and jobs with just the deadlines. However, even in this simple case Knapsack 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). Also, Knapsack is a special case of Scheduling with deadlines, profits and durations: just treat weights as durations, profits, if there are any, as profits, and set all deadlines to be the capacity; this gives us NP-hardness of such Scheduling.

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. □

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 $O(n)$.

1.4 Assigning TAs to courses: network flows

There is another special class of problems for which it is possible (under certain reasonable restrictions) to find a solution in polynomial time. Consider, for example, the following problem. A secretary wants to assign grad. students to be teaching assistants (TAs) for courses. Some courses need more than one TA; say each grad student can have up to 2 TA positions. However, not all grad. students qualify to be teaching assistant in all courses: e.g., if a student has not done advanced graphics before, she cannot take a TA position for an advanced graphics course. This task is not quite the same as in the Stable Marriage problem: there, every pair was possible, just some were less desirable. However, such TA assignment problem can also be solved in polynomial time by using the Network Flows technique (provided the input is "reasonable" – the technique wouldn't work if there are e.g. irrational numbers).

This technique will give us a nice set of problems that can be modeled (and solved) using graphs.

1.5 AI planning: PSPACE, beyond NP

Finally, let me comment on some problems that are believed to be even harder than NP problems, and yet which arise in practice quite commonly. This is the class of problems that we know how to solve in exponential time, but using only polynomial amount of space: this class is called PSPACE. It contains many "winning strategy in a game" type problems: ones of the form "if the first player does this, the second should respond like that, and then if the first player...". Classical planning in artificial intelligence is another example of such problem. Even when defined as a decision problem (e.g., "is there a winning strategy starting with a given board?" in chess), it is not at all clear how to verify that a given solution is correct, since it has to account for all possible decisions by the other player, at every stage of the game. When complexity classes are described in terms of quantifier complexity of their problems (e.g., for NP it is one existential quantifier: $\exists y, |y| \leq |x|^d CheckCorrectness(x, y)$, where $CheckCorrectness(x, y)$ runs in time polynomial in $|x| + |y|$), PSPACE problem descriptions can have an arbitrary number of quantifiers.

Are they the hardest possible problems? Not at all – they can still be done by brute-force search in exponential time. It can be proven that for every exponential increase in time, there are problems that do require that much time; some require $2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}$ – a tower of n exponents – time. Moreover, some are not solvable at all in their full generality...