CS 3719 (Theory of Computation and Algorithms) – Lecture 1

Antonina Kolokolova*

January 6, 2011

This course will focus on the theory of computation, that is, problems that can (and cannot) be solved computationally, and methods of doing so, when they exist. We will start by reviewing some methods of solving problems efficiently, and corresponding classes of problems. Then, we will move to problems for which no efficient solution is known to exists (computational complexity). In order to formalize that, we will introduce some formal models of computations: finite automata and Turing machines, in particular. Later, we will move to proving unsolvability (computability theory).

1 Review of algorithm design paradigms

In this lecture, we review some of the main algorithmic design paradigms, using as an example various versions of the Knapsack problem. Most of this material you probably saw in cs2711.

1.1 The Knapsack problem

Input: A list of n objects, each with a weight w_i and a profit p_i associated with it, and a bound $B \geq 0$. More precisely, the algorithm receives as an input n pairs of numbers $(w_1, p_1) \ldots (w_n, p_n)$ and a number B. These numbers are not necessarily integers, but let's use integers to simplify our examples.

Output: A subset of maximal profit of input objects that "fits" (as a sum of weights) below B. More formally, the output is a set $S \subset \{1, \ldots, n\}$ such that $\Sigma_{j \in S} w_j \leq B$ and $\forall S', \Sigma_{k \in S'} w_k \leq B \rightarrow \Sigma_{j \in S} p_j \geq \Sigma_{k \in S'} p_k$

*The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364.

To visualize this problem, you can think of a container ship that can only take certain limited weight, and is trying to pick out the most profitable cargo. Or of a thief coming to museum and trying to steal as much as can be carried out – likely picking a less expensive small painting over an expensive statue.

For example, suppose that the input is $\{(3,4),(5,10),(7,1),B=9\}$. Then, $S=\{1,2\}$ will give an optimal profit of 14. If $B=7$, the best is $S=\{2\}$. And there could be several optimal solutions: for example, for $\{(3,4),(5,10),(7,14),B=9\}$ input the optimal answers are either $S=\{1,2\}$ or $S=\{3\}$.

We will show here how to solve restricted versions of Knapsack, and give a general, not efficient algorithm. Later in the course we'll show that there is (under a believable assumption) no algorithm for this problem better than backtracking or brute-force search.

1.2 Greedy algorithm example: Fractional Knapsack

Consider a version of Knapsack in which a portion of an object can be taken (e.g., think of taking just a part of a bottle of medicine or perfume, or transporting bulk grains/beans). In this case, the following simple algorithm solves the problem.

```
**Algorithm FracKnapsack**
Input: $(w_1, p_1) \dots (w_n, p_n), B$
Output: $S$, $Frac$, where $S$ contains the set of whole objects,
and $Frac = (index, portion)$ the name and fraction of the fractional object, if any.

 Sort inputs in the order of decreasing profit per unit weight $p_1/w_1 \geq \dots \geq p_n/w_n$.
// The indices in $S, Frac$ are in this order.
$Sum \leftarrow 0; i \leftarrow 1$

$S \leftarrow \emptyset; Frac \leftarrow null$
while $Sum + w_i \leq B$ do
    $S \leftarrow S \cup \{i\}$; $Sum \leftarrow Sum + w_i$
    $i \leftarrow i + 1$
end while
if  $Sum < B$ do
    $Frac \leftarrow (i, p_i * (B - Sum)/w_i)$
return $S, Frac$
```

To show that this algorithm works, we'll argue by induction. First, notice that if $B \geq 0$ then there exists some solution, and so there is an optimal one among them (since two solutions can always be compared on their profit). Now, suppose that at step i of the algorithm the partial solution $S_i.Frac_i$ computed by the algorithm can be extended to some optimal solution $S_{opt}, Frac_{opt}$ (that is, $S_i \subseteq S_{opt}$ and $Frac_i$ is either *null* or $Frac_{opt}$). We will show that at step $i+1$ there is a (possibly different) optimal solution $S'_{opt}, Frac'_{opt}$ such that $S_{i+1} \subseteq S_{opt}$ and $Frac_{i+1} = null$ or $Frac_{i+1} = Frac'_{opt}$.

There are three stages of the algorithm: 1) adding a whole object to S, 2) adding a fractional object to $Frac$ 3) Doing nothing (when the knapsack is already filled). We are only concerned with the first two stages, because $S_{i+1} = S_i$ for the subsequent stages. In the first stage, there are two possibilities a) either an object $i+1$ being added is already in S_{opt}, then there is nothing to prove b) or $i+1 \notin S_{opt}$. But notice that the only way b) can happen is when there is another object j with exactly the same ratio: $p_{i+1}/w_{i+1} = p_j/w_j$: if j had better ratio, we'd already consider it in our sorting order. In that case, make S'_{opt} by replacing units of i with units of j. If there is more of i than of j, pushing the last object off S_{opt} and $Frac$ can only make the solution better; if there is more of j than of i, then there must be other stuff as good as j to fill in the rest of the solution S'_{opt}.

Now, since this argument works for every step, and nothing can be added to the solution after reaching B, the resulting solution must be optimal.

1.3 Dynamic programming: knapsack with integer weights and small bound

We can use dynamical programming to solve the Knapsack problem, but it would only be efficient, loosely saying, if the bound is small, and weights are integers. In that case, we build a dynamic programming table with objects (1 to n) and weights (from 0 to B). A cell (i, w) in this table encodes the best way to pack objects from 1 to i into knapsack with bound w.

Notice that this algorithm would not be good if B is large, e.g., on the order of 2^n. In that case, backtracking or brute-force search is preferred.

1.4 Backtracking: most general method for Knapsack problem

Build a decision tree, where on level i nodes split on putting vs. not putting i in the set. Stop the branches that exceeded B. Record the best solution so far, and backtrack. In the worst case, would have to visit pretty much all the leaves (e.g., all weights are the same and all, maybe except one, fit). So this algorithm will have exponential running time in the worst case.

Alternatively, can do a brute-force search: go through all subsets of weights as possible S, and keep track of the best solution. This also takes exponential time.