COMPSCI 330: Design and Analysis of Algorithms **Feb 5, 2020**

Lecture 8: Greedy Algorithm II

Lecturer: Rong Ge *Scribe: Haoming Li*

Overview

In this lecture, we design and analyze greedy algorithms that solve the fractional knapsack problem and the Horn-satisfiability problem.

In general, to design a greedy algorithm for a probelm is to break the problem into a sequence of decision, and to identify a rule to make the "best" decision at each step. After designing the greedy algorithm, it is important to analyze it, as it often fails if we cannot find a proof for it. We usually prove the correctnesst of a greedy algorithm by contradiction: assuming there is a better solution, show that it is actually no better than the greedy algorithm.

8.1 Fractional Knapsack

Just like the original knapsack problem, you are given a knapsack that can hold items of total weight at most W. There are n items with weights $w_1, w_2, \dots, w_n$ and value $v_1, v_2, \dots, v_n$. The difference is that now the items are infinitely divisible: can put $\frac{1}{2}$ (or any fraction) of an item into the knapsack. The goal, therefore, is to select fractions $p_1, p_2, \dots, p_n$ to maximize $p_1v_1 + p_2v_2 + \cdots + p_nv_n$, subjected to the constraint of $p_1w_1 + p_2w_2 + \cdots + p_nw_n \leq W$.

Here's an example. Given a knapsack of capacity $W = 5$ and three items, each with weight $w_1 = 1$, $w_2 = 2$, $w_3 = 3$ and value $v_1 = 2$, $v_2 = 3$, $v_3 = 4$. If this were the original "0/1" knapsack problem, the optimal solution would be to pick item 2 and 3 and achieve a total value of 7. Since now we are allowed to pick fractions, the optimal solution is to take item 1, 2, and $\frac{2}{3}$ of item 3 to achieve a total value of $\frac{23}{3} > 7$.

8.1.1 Algorithm

We want to make a sequence of decision on what to put in the knapsack. Consider the following algorithm:

Algorithm: Iteratively picks the item with the greatest value-per-weight ratio ($\frac{v_i}{w_i}$).

If, at the end, the knapsack cannot fit the entire last item with greatest value-per-weight ratio among the remaining items, we will take a fraction of it to fill the knapsack.

8.1.2 Analysis

Running Time: This algorithm takes $O(n \log n)$ time to sort the items by the ratio in decreasing order, and another $O(n)$ time to traverse and pick from the list of items until the knapsack is full. Hence the total running time is $O(n \log n)$. We will then prove the correctness of this greedy algorithm.

Proof of Correctness: Assume towards contradiction that there is an instance of fractional knapsack such that the solution of this algorithm (ALG) is not optimal. Let OPT denote the optimal solution. Without loss of generality, assume that items are sorted in decreasing order by $\frac{v_i}{w_i}$, and that no two items that have the same $\frac{v_i}{w_i}$ [1]. Let $ALG = \{p_1, p_2, \ldots, p_m\}$ denote the sequence of decision (fractions taken) made by our greedy algorithm and $OPT = \{q_1, q_2, \ldots, q_m\}$ denote that in OPT.

Therefore, by assumption we have $\sum_{i=1}^m p_i v_i < \sum_{i=1}^m q_i v_i$, i.e. ALG is not optimal. Let l be there first coordinate at which $p_l \neq q_l$. By the design of our algorithm, it must be that $p_l > q_l$ and that $\sum_{i=1}^l p_i v_i > \sum_{i=1}^l q_i v_i$. By the optimality of OPT, there must exist a coordinate $h > l$ such that $q_h > p_h \geq 0$. Consider a new solution $q' = \{q'_1, q'_2, \ldots, q'_m\}$ where $q'_j = q_j$ for all $j \neq l, h$. q' will take a little more of item l and a little less of item h compared to OPT: $q'_l = q_l + \epsilon$, $q'_h = q_h - \epsilon \frac{w_l}{w_h}$. The total weight does not change: $\sum_{i=1}^m q'_i w_i = \sum_{i=1}^m q_i w_i$. Yet the total value strictly increases: $\sum_{i=1}^m q'_i v_i = \sum_{i=1}^m q_i v_i + \epsilon v_l - \epsilon \frac{w_l}{w_h} v_h > \sum_{i=1}^m q_i v_i$.

That q' is a valid and better solution than OPT is a contradiction. Hence, our the solution from our algorithm is optimal. QED

8.2 Horn-Satisfiability

The most primitive object in a Horn-SAT problem is a *Boolean variable*, taking value either true or false. A *literal* is either a variable x or its negation $\bar{x}$. In Horn-SAT, knowledge about variables is represented by three kinds of clauses:

1. *Implications*, whose left-hand side is an AND of any number of positive literals and whose right-hand side is a single positive literal. For instance, $(z \wedge w) \implies u$ is an implication clause.
2. *True variables*, where one variable is required to be true. such as u. (Often this is considered as a special case of Implications where the left-hand side is empty, in that case it is written as $\implies u$.)
3. *Negative clauses*, consisting of an OR of any number of negative literals, as in $(\bar{u} \vee \bar{v} \vee \bar{y})$

Given a set of clauses of these two types, the goal is to determine whether there is a satisfying assignment: an assignment of true/false values to the variables that satisfies all the clauses. For instance, given

$$(x \wedge y) \implies z, \implies x, (\bar{x} \vee \bar{y} \vee \bar{z})$$

One satisfying assignment would be $x =$ true, $y =$ false and $z =$ false; on the other hand, $x = y =$ true and $z =$ false is not an satisfying assignment as it violates the first clause. However, consider the following example

$$(x \wedge y) \implies z, \implies x, \implies y, (\bar{x} \vee \bar{y} \vee \bar{z})$$

We certainly need to have $x = y =$ true to satisfy the second and the third clause. Consequently, we need to have $z =$ true to satisfy the first clause. However, such assignment will not satisfy the fourth clause. Hence, we shall conclude that there is no satisfying assignment for this set of clauses.

8.2.1 Algorithm

If all variables are initialized to false, we want to make a sequence of decision on which variable should be assigned true. Consider the following algorithm:

Algorithm:
Set all variables to false;

[1]Observe that if for two items the ratios are the same, we can consider "merge" them to a single item whose weight and value are each the sum of the two items.

```
While there is an implication that is not satisfied:
  set the right-hand variable of the implication to true;
if all pure negative clauses are satisfied: return the assignment;
else: return "not satisfiable";
```

8.2.2 Analysis

We want to show that our algorithm outputs "not satisfiable" only if there exists no satisfying assignment.

Proof of Correctness: Assume towards contradiction that there is a satisfying assignment. Let $x_1, x_2, \ldots, x_k$ be the variables set to true by the algorithm in the order that they are set to true. At the last step of the algorithm, one of the negative clauses was violated. That means this particular negative clause C must only have variables within $x_1, ..., x_k$.

If $x_1, x_2, \ldots, x_k$ are all true in the satisfying assignment, then the negative clause C that the algorithm violated is still violated.

Otherwise, let x_i be the first variable that is false in the satisfying assignment.

If x_i is in an implication clause with no left-hand side, then the clause is violated. Otherwise, we consider the clause C' that the algorithm used when it sets x_i to true. All the variables on the left-hand side of C' must have been set to true earlier by the algorithm, and since x_i is the first variable that is false in the satisfying assignment, all the previous variables on the left-hand side of C' must be true for the satisfying assignment. However, x_i is false in the satisfying assignment. This violates the clause C'.

Therefore in all cases we have found a clause that is violated by the hypothetical "satisfying" assignment. There cannot be a satisfying assignment.