Approximation Algorithm

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 04

Lecture 17

Lecture 17 : Pseudo Polynomial Time Algorithm for Knapsack

In the last couple of lectures, we have been seeing how greedy algorithms and local search heuristics can be used for designing approximation algorithms with probable guarantees. Now, we use another popular algorithm design techniques known as dynamic programming and we will see how dynamic programming can be effectively used for designing polynomial time approximation algorithms. So, we start. to see the use of dynamic programming for designing polynomial time. approximation algorithms. So, again we will see we will learn this through various applications through various examples.

So, our first problem is the classical knapsack problem. So, what is the input? We have n items with value with respective values $v_1, \ldots, v_n$ and sizes $s_1, \ldots, s_n$. We have one knapsack or bag of capacity B the goal the goal is to compute the subset of items such that their total size is less than equal to the capacity of the bag and their total value is maximized. It can be shown that this is NP complete you can take it as a homework.

The knapsack problem is NP complete. And now we will design a dynamic programming algorithm for the knapsack problem. So, the algorithm is very simple. So, we Now design dynamic programming algorithm for knapsack. because it is NP complete the runtime of the algorithm will not be ah polynomial, but it will be what is called weakly polynomial.

It will be polynomial of the values of the integers ah which are input . So, it is a simple algorithm. So, let us write the pseudo code . So, we will maintain or let me write the idea we maintain or we compute the set which I call A_i containing all value size pair pairs of undominated subset of items from 1 to i.

Now, what is this undominated? So, 2 subsets of items or let me write a subset of items is said to dominate another. subset of items subset y of items if the value of x which is defined as the sum of the values of the items in x this is greater than equal to value of y

and the size of x again this is sum of the sizes of the items this is less than equal to S of y. So, if both of them hold and at least one of the above two inequalities is tight is strict. So, the idea is if I have 2 bundles 2 set of items the bundle x who that weight the size of x is less than equal to size of y, but the value is more than y then it is then we say x dominates y or the value may be same, but the size of x is strictly smaller then also we say that x dominates y. of course, if both size is size of x is strictly smaller than size of y and value of x is strictly greater than value of y then also obviously, we say that x dominates y.

So, in this set A y we will store the value size pair of all undominated subsets of items and we will see how we maintain that set. and in the at the end we will argue that why this set cannot be too large and why it is enough in a n a n is the set of all value size pairs of all undominated subsets of 1 to n and the optimal solution must be undominated. So, that is why it makes sense to store undominated pairs. So, note that any optimal solution any optimal solution O subset of $[n]$ for any optimal solution O this pair the value of O and size of O. this pair is an undominated pair and hence this belongs to A_n.

So, by looking at A_n and if it has it has small number of elements then I can find out which I can find out the optimal solution by picking that pair whose size is less than equal to b and value is highest possible ok. So, now let us see the pseudocode. So, A_1 is the set of all undominated pairs of the singleton item I, it has two subsets singleton I and empty set and both is both are undominated. So, for A_1 I know that for singleton set value is 0 and size is 0 and for singleton for singleton item 1 not I this is 1 this is (v_1, s_1). So, this is we know a 1 now I will iteratively compute A_{i+1} from A_i.

So, for each i equal to 2 to n, A_i is A_{i-1}. So, I start from there and I from for every pair of for every pair I add the value and size of the ith item ok. So, here I am looking at this set i which is 1 2 and I have already I already have A_{i-1} this is for the set of items $i-1$ I have value and size pair of all undominated sets. Now, you see the undominated sets of a i which is a subset of 1 to $i-1$ they are already present in A_{i-1}. So, that is why I am keeping them and for each of them I am trying to augment i.

So, what I do is that for each $(t, w) \in A_{i-1}$ add $(t+v_i, w+s_i)$ to A_i. So, after this for loop the size of A_i is exactly twice the size of A_{i-1}. And then I compare every pair of pairs and if I see that one pair dominates other pair then I remove the dominated pair. And this that means, I check if I have these two conditions are satisfied or not and at least one of them should hold strictly and that can be done in time $O(|A_i|^2)$.

So, remove all dominated pairs from A_i and that is it. Now, what should what is the

maximum possible value that I get within the capacity of B that is I look at the set A_n and there I look at the maximum I maximum possible values. So, max (t, w) ok and I add this if the size $s+w$ is less than equal to B capacity of the bag. So, it does not make any sense to store the information of any subset whose capacity is greater than B ok. So, $s+w$ is less than equal to B and here also we can assume that here is a important assumption we assume without loss of generality that the size of every item is less than equal to B for all how it is without loss of generality? Because if there exist an item whose size is more than B that item can never be part of any optimal solution.

So, we delete that item from that from the instance and work with the reduced instance. So, this way because I am storing only the information of those pairs whose size is less than equal to B in A_n I just need to find out the pair whose value is the highest ok. So, this you prove it formally using loop invariant. that is the proof of correctness. Proof that the above dynamic programming algorithm always outputs an optimal solution ok.

So, what is the time taken time complexity of for that all we need to do is we need to bound the maximum size of A_i. Now, here is an important observation. So, let A_i with this pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$. what we do we arrange them in in say decreasing order of values. So, by renaming suppose first is value yeah.

suppose $x_1 \geq x_2 \geq x_3 \geq \ldots \geq x_m$. first observe that I cannot have any equality 2 items cannot have same value because if 2 item has same value suppose $x_1 = x_2$ then their y values must be different if y values are also same then they are same pair which is not allowed in a set. Now, if y values are different then one dominates another. So, suppose if $x_1 = x_2$, but $y_1 < y_2$ then (x_1, y_1) dominates (x_2, y_2) and we are removing all undominated pairs that is why all this inequalities must be strict.

So, this will hold. what we have and all these are integers this and x_i is an integer, hence cardinality A_i is less than equal to maximum value possible. V which is or $V+1$ it can range from 0 to V. So, is $V+1$ which is where $V = \sum_{i \in [n]} v_i$. total value of all items. Similarly, it follows that cardinality A_i is less than equal to $1+B$ capacity of the bag the y values can range from 0 to B only ok.

So, what is the running time of the algorithm? it iterates over n iterations and each one can be executed in time big O of cardinality A_i each iteration. So, the runtime is big O of n times minimum of B and V because from here what we get is cardinality A_i is less than equal to $1+min(B, V)$ for all $i \in [n]$. So, this is the runtime you see it is not a polynomial time algorithm because the input is input size is log of these values and sizes. So, such algorithms are called pseudo polynomial time algorithms. polynomial in the values of

integers and input size is called pseudo polynomial time algorithm.

An algorithm whose run time is polynomial in the values of the integers ah integer inputs if input contains integers integer inputs and other input size is called pseudo polynomial time So, next we will see that we can use this pseudo polynomial time algorithm for knapsack for designing what is called an FPTAS. So, what is the definition of PTAS polynomial time approximation scheme. polynomial time approximation scheme PTAS. What is it? It is a family of algorithms PTAS is a family of algorithms A_ϵ for every ϵ greater than 0 such that A_ϵ is $1+\epsilon$ approximation algorithm for minimization problem and $1-\epsilon$ approximation algorithm for maximization problem and 1 minus epsilon approximation algorithm for problems for maximization problems and runs in time big O of $n^{f(\epsilon)}$. So, for every epsilon it is polynomial time algorithm, there is another concept which is called FPTAS which is called fully polynomial time approximation scheme.

FPTAS which is same as peters approximation guarantee is the same as petas, but A epsilon runs in time polynomial on n and 1 over epsilon. So, in the next class we will see an FPTAS algorithm for knapsack problem using this dynamic programming algorithm ok. So, let us stop here. Thank you.