DAPO

Project 1 - Approximation Algorithms

FCT UNL

Filipe Medeiros - 47967

Prof. Margarida Mamede

Problem

The chosen problem is the Knapsack problem. It consists of the following: given a set of objects, each with a weight and a value, find the subset of objects that has the greatest value, but whose combined weights do not exceed a given limit. For example, if you have the set of (weight, value) objects S = {(3, 4), (5, 6), (4, 7)}, with a weight limit of 8, the optimal solution would be {(3, 4), (4, 7)}. The decision problem for the Knapsack problem is NP-complete (although the optimization problem is NP-hard). In this report, I will be considering the 0/1 discrete version of the problem, which means there is only one bag, and every item can only be taken once.

Approximation Algorithms

I will be implementing 2 approximation algorithms. They are: an adjusted greedy algorithm and a dynamic programming algorithm. In general, both achieve satisfactory results, which will be detailed later in the report. Note that edge cases, like the input set being empty, were ignored in the implementation.

Next, I will provide a brief explanation of both algorithms.

Adjusted Greedy

1) Receive input consisting of:
    a) the list of items (each with a value and a weight)
    b) the limit weight of the knapsack
2) Sort the list of items in descending order, according to the following function: given any two items i_1 and i_2, i_1 is greater than i_2 if the ratio between i_1's value and i_1's weight is greater than that of i_2's. In pseudo code, if i_1.value / i_1.weight > i_2.value / i_2.weight, then i_1 will be first in the list than i_2.
3) Loop through the sorted list, "taking" items while the sum of their weights is less than or equal to the weight limit. If an item doesn't fit in the knapsack, or in other words, if the sum of all current taken items' weights plus the current item is greater than the weight limit, just skip to the next one.
4) Repeat step 3 until one of two happens: the entire list has been looped through, or no item exists that can still fit in the bag.
5) The solution will be the sum of the values of all items taken in step 3.

Dynamic Programming

1) Receive input consisting of:
    a) the list of items (each with a value and a weight)
    b) the limit weight of the knapsack
2) Create a matrix with dimensions N + 1 x W + 1, where N is the number of objects in the instance, and W is the weight limit of the knapsack, and fill the matrix with 0.
3) Loop through the matrix (except the first column and row) and fill the cells according to the rule: in the cell (n, w), if the weight of item n (order of the input list is arbitrary) if greater than the weight limit, then the cell's value is equal to the value of the cell (n - 1, w). If not, the value of cell (n, w) is the maximum between the value of cell (n - 1, w) and the value of item n plus the value of the cell (n - 1, w - items[n].weight).
4) Repeat step 3 until the last cell, in position (n, w), is filled.
5) The solution will be equal to the value of this last cell.

Algorithm Complexities

I will now indicate, for each algorithm, its time and space complexities, as a function of the input. A small explanation will be provided, but the code has comments that explain more extensively how each step of the algorithm contributes to these complexities. Note that for either algorithm, running time of reading the input was not accounted for, and it is assumed that an input of proper form is given. In the following complexities and explanations, n represents the number of objects in the input set and maxValue represents the value of the object with the highest value in the input set.

Adjusted Greedy

Time complexity: O(n * log(n))

Time complexity explanation: Sorting the objects according to their value/weight ratio is O(n * log(n)), and iterating through the sorted set to make the decisions is O(n).

Space complexity: O(n)

Space complexity explanation: The helper SortedMap will contain as many elements as the input set, so it is O(n). Even if you want to store the solution's chosen objects, this will be at worst also O(n).

Dynamic Programming

Time complexity: O(n^3 / ε)

Time complexity explanation: As explained in the lectures, an exact algorithm that returns the optimal solution runs in O(n^2 * maxValue). Because of the approximation that is made, based on a factor ε, we can rewrite the complexity as a polynomial function of n for any fixed ε O(n^3 / ε).

Space complexity: O(n^3 / ε)

Space complexity explanation: As explained in the lectures, an exact algorithm that returns the optimal solution takes a matrix of size O(n^2 * maxValue). Because of the approximation that is made, based on a factor ε, we can rewrite the complexity as a polynomial function of n for any fixed ε O(n^3 / ε).

Algorithm Approximation Ratios

Both algorithms always return a solution that is garranteed to be at least a fraction of the optimal solution (since this is a maximization problem) for that instance. A reference will be given for each one.

Adjusted Greedy

The greedy algorithm's ratio is equal to 2. More information at https://www.brainkart.com/article/Approximation-Algorithms-for-the-Knapsack-Problem_8066/.

Dynamic Programming

The dynamic programming algorithm's ratio works in a different way. A precision parameter is passed to the algorithm. This precision parameter **ε** will determine the approximation ratio of the algorithm, making the computed solution greater than or equal to (1 - ε) * S* where S* is the optimal solution. More information can be found at:
http://web.cs.iastate.edu/~cs511/handout08/Approx_Knapsack.pdf
https://en.wikipedia.org/wiki/Knapsack_problem

Experimental Results Analysis

Theoretically, we should expect that the greedy algorithm runs faster than the FTPAS algorithm for any given instance, which was the case in the experimental results. However, the approximation ratio of the FTPAS algorithm should be better than the 0.5 provided by the greedy algorithm, under certain circumstances (very high values and high accuracy needed), which was not observed. This is probably due to badly generated test instances.

The instances were generated randomly, given a number of objects, a weight limit (that served as an upper bound for the capacity of the knapsack and consequently for the weights of the objects) and a value limit, which no object's value could surpass.

I think this way of generating instances led to results being skewed, also because the greedy algorithm got an impressive ~0.95 accuracy performance. Under normal condition, in the long run, both algorithms would have accuracies much closer to those predicted in theory.

The exact results will be included in the repository, along with the instances used and the code used to generate them.

One last thing to note is that, in some of the test that were run (but are not presented), the JVM ran out of memory, because of the size of the input object list being relatively big (from about 50 objects), and it had problems allocating memory for the dynamic matrix.

Code, Source and Results

All the source code and results will be email along with this report and can also be found at https://ww