DAPO 2017/18

1st Report - Programming Project

Ana Garcia, 46783
Matti Berthold, 52226

April 22

Contents

1 Definition of the Problem

In the Knapsack Problem, we are given a set of n item $I = \{1, \ldots, n\}$, where each item i had a value V_i and a weight W_i. All the values and weights are positive integers. The knapsack has a capacity C which is a positive integer as well. The goal is to find a subset of items whose total weight does not exceed C. We assume that we consider only items that could fit into the Knapsack, so that $w_i \leq C$ for each $i \in I$. The decision problem of the Knapsack is NP-complete.

2 Approximation Algorithm of the Knapsack Problem

2.1 Approximation Schemes

First we start by solving the Knapsack Problem with dynamic programming. From the problem description we have a maximum capacity C, n items, with a value v_i and a weight w_i each. We want to calculate the max-value for the items first i items and capacity j, such that $0 \leq i \leq n$ and $0 \leq j \leq C$.

- If the number of items is 0, $(i = 0)$, then the max value is 0 for each $j \geq 0$, i.e. $V(i, j) = 0$
- If there is an item that is to heavy to the current capacity, i.e. $w_i > j$, we do not add i value to the subset value, $V(i, j) = i - 1, j$
- If there is an item that does fit, i.e. $w_i \leq j$, then we want the max value between:
  - we add is value to the subset value $V(i - 1, j - w_i)$
  - we do not add i to the subset value $V(i - 1, j)$

$$V(i,j) = \begin{cases} 0 & i = 0 \\ V(i-1,j) & i \geq 0 \wedge w_i > j \\ max(v_i + V(i-1, j-w_i), V(i-1,j)) & i \geq 0 \wedge w_i \leq j \end{cases}$$

The solution value is V(n,C). This algorithm has time complexity $O(nC)$, the size of the matrix is $n \times C$, so the algorithm is pseudo-polynomial in the size of the instance.

At this point we can see that for this algorithm scaling the values has no impact on the size of the instance, and we cannot scale capacities, as solutions we would obtain on the scaled instance might not be real solutions(the total of the real weights may be bigger than the capacity). Accordingly, we approach the algorithm as a minimization of the Weights: We want the min weight using the first i items, whose total value is j. Therefore we use pairs of i and j with $0 \leq i \leq n$ and $0 \leq j \leq \sum_{k=0}^{n} v_k$

- If the value is 0 ($j = 0$) then the min weight is 0 for each $i \geq 0$, i.e. $W(i, j) = 0$
- If there is still some positive value, but no more items, $i = 0$, $j > 0$, then it is impossible to achieve the value, $W(i, j) = \infty$
- If item i is too valuable, $i \geq 1$ and $v_i > j$: we do not add w_i, to the subset weight, $W(i, j) = W(i - 1, j)$
- If item i can be selected, $i > 1$, $v_i \leq j$, the we want to min weight between:
  - we add w_i to the subset weight, $W(i - 1, j - v_i)$
  - we do not add w_i to the subset weight, $W(i - 1, j)$

$$W(i,j) = \begin{cases} 0 & j = 0 \\ \infty & i = 0 \wedge j > 0 \\ W(i-1,j) & i \geq 1 \wedge v_i > j \\ min(w_i + W(i-1, j-v_i), W(i-1,j)) & i \geq 1 \wedge v_i \leq j \end{cases}$$

The solution value is the maximal total value v such that $W(n, v) \leq C$. This algorithm is an exponential algorithm that solves MAX-Knapsack.
Proof:

- The algorithm returns an optimal solution.
- The time complexity is $\Theta(n \sum_{i=1}^{n} v_i)$ and $\sum_{i=1}^{n} v_i)$ can be exponential in the size of the instance. Since $\sum_{i=1}^{n} v_i) \leq n * max\ v_i$ the running time is also $O(n^2 * max\ v_i)$

Now we are going to lose precision (by decreasing values) to gain efficiency. To do this, we scale down all values so that they lie in a smaller range then we execute the algorithm on the new scales instance and return the optimal solution of the scaled instance. The scale depends on the approximation parameter $\epsilon \in]0,1[$. The code of our approximation scheme starts by scaling down all values of the instance, using the input parameter ϵ, and then executes the exact dynamic algorithm to determine the min weight, with the scaled instance. In the end, the return statement is the optimal solution for the scaled instance.

Algorithm 1 Scheme Knapsack($S = \{(v_i, w_i) \dots (v_n, w_n)\}, C, \epsilon$)

$v_{max} \leftarrow max_{0 \leq i \leq n} v_i$
$\delta \leftarrow (\epsilon \times v_{max})/n$
$\hat{S} \leftarrow \{(\hat{v}_1, w_1), \dots, (\hat{v}_n, w_n)\}$ where $\hat{v}_i = \frac{v_i}{\delta}$
return ExactKnapsack($\hat{S}, C$)$\times\delta$

Algorithm 2 Exact Knapsack($S = \{(v_i, w_i) \dots (v_n, w_n)\}, C$)

```
valSum ← Σ_{k=1}^{n} v_k
Let T be an array of the size n + 1 × valSum + 1
for i = 0 to n do
    T[i][0] ← 0
end for
for j = 0 to valSum do
    T[0][j] ← ∞
end for
for i = 1 to n do
    for j = 1 to v_i − 1 do
        T[i][j] ← T[i − 1][j]
    end for
    for j = v_i to valSum do
        T[i][j] ← min(w_i + T[i − 1][j − v_i], T[i − 1][j])
    end for
end for
for j = valSum to 1 do
    if T[n][j] ≤ C then
        return j
    end if
end for
```

Time Complexity

$$SchemeKnapsack(S, C, \epsilon) \in O(|S|^2 \times \hat{v}_{max}) = O(\frac{|S|^3}{\epsilon})$$

$$\left(\hat{v}_{max} = [\frac{v_{max}}{\delta}] = [v_{max}\frac{n}{\epsilon v_{max}}] = [\frac{n}{\epsilon}] = [\frac{|S|}{\epsilon}]\right)$$

So the running time is polynomial in $|S|$ and $\frac{1}{\epsilon}$

Space Complexity

$$O(|S|^2 \times \hat{v}_{max}) = O(\frac{|S|^3}{\epsilon})$$

Approximation Ratio

Let I be the computed solution, whose value is $v(I) = \sum_{i \in I} v_i$
and I^* be the optimal solution, whose value is $v(I^*) = \sum_{i \in I^*} v_i$
The approximation ratio is:

$$v(I) \geq (1-\epsilon)v(I^*) \Leftrightarrow \frac{v(I^*)}{v(I)} \leq \frac{1}{1-\epsilon} \text{ with } \epsilon \in]0,1[$$

To prove the approximation ratio, we need to prove that our our algorithm is an approximation scheme. This also proves that SchemeKnapsack is an FPTAS together with the proof the the running time is polynomial in: n, $log(C)$ and $\frac{1}{\epsilon}$; already provided.
schemeKnapsack is an approx. scheme:
value if an optimal solution in the rounded instance

$$\begin{aligned}
\sum_{i \in I^*} \hat{v}_i &= \sum_{i \in I^*} v_i \\
&\geq \sum_{i \in I^*} (\frac{\hat{v}_i}{\delta} - 1) \\
&= \frac{1}{\delta} \sum_{i \in I^*} v_i - \sum_{i \in I^*} 1 && \text{arithmetic properties} \\
&= \frac{1}{\delta} v(I^*) - |I^*| && \text{value of the optimal solution} \\
&= \frac{1}{\delta} v(I^*) - n && |I^*| \leq n
\end{aligned}$$

value of the computed solution in the original instance

$$\sum_{i \in I} v_i \geq \sum_{i \in I} \hat{v}_i \delta = \delta \sum_{i \in I} \hat{v}_i \qquad v_i \geq \frac{v_i}{\delta}\delta = \hat{v}_i\delta$$

$$\geq \delta \sum_{i \in I} \hat{v}_i \qquad I \text{ is an optimal solution in rounded instance}$$

$$\geq \delta(\frac{1}{\delta}v(I^*) - n)$$

$$= v(I^*) - \delta_n$$

$$= v(I^*) - \epsilon v_{max} \qquad \delta = \frac{\epsilon v_{max}}{n}$$

$$\geq v(I^*) - \epsilon v(I^*) \qquad v_{max} \leq v(I^*) \text{ because the algorithm can select an item whose value is } v_{max}$$

$$= (1 - \epsilon)v(I^*) \qquad q.e.d.$$

Note: The presented proof was taken from the class slides

2.2 The Primal-Dual Approach

For this approach we are going to use the minimum covering knapsack problem. (given a set of items I each with weight w_i and value v_i, we want to find a subset of items with minimum weight such that the total value is at least D.) The knapsack problem formulated as an integer programming problem is:
variable:

$$x_i = \begin{cases} 0 & \text{if the item does not belong to the subset} \\ 1 & \text{if the item does not belong to the subset} \end{cases}$$

Minimize: $\sum_{i=0} w_i x_i$
Subject to: $\sum_{i=0} v_i x_i \leq D$

Is this a good integer program in terms of relaxation?
No, its horrible.
Consider the instance $I = \{(D, 1), (D-1, 0)\}$ The integer solution must take item 1 and incurs a weight of 1. The LP solution can take all of item 2 and just $\frac{1}{D}$ fraction of item 1, incurring a weight of $\frac{1}{D}$.
$\frac{\text{integer solution}}{\text{LP solution}} = \frac{1}{1/D} = D$
D is as large as we want! So there is no constat C such that $D \leq C$.

After some research we found the following approach (by Carr, Fleischer, Leung and Phillips (2000), knapsack-cover Inequalities)

- consider a subst I' of I.
- If we were to take all items in I', then we still need to take enough items leftover demand:

$$\sum_{i \in I \setminus I'} v_i x_i \geq D - v(I') \quad \text{where } v(I') = \sum_{i \in I'} c_i$$

- This inequality adds nothing new, but we can now restrict the values of the items

$$\sum_{i \in I \setminus I'} v_i^{I'} x_i \geq D - v(I') \quad \text{where } v_i^{I'} = min(v_i, D - v(I'))$$

  Let $D_{I'} = D - v(I')$ so now we have

$$\sum_{i \in I \setminus I'} v_i^{I'} x_i \geq D_{I'} \quad \text{where } v_i^{I'} = min(v_i, D_{I'})$$

  With this we can give the following integer programming formulation of the problem:

Minimize: $\sum_{i \in I} w_i x_i$
Subject to: $\sum_{i \in I \setminus I'} v_i^{I'} x_i \geq D_{I'}$ for each $I' \subseteq I$
$x_i \in \{0, 1\}$ for each $i \in I$

Now we considerer the linear programming relaxation of the integer programming above, relaxing $x_i \in \{0, 1\}$ to $x_i \geq 0$

Minimize: $\sum_{i \in I} w_i x_i$
Subject to: $\sum_{i \in I \setminus I'} v_i^{I'} x_i \geq D_{I'}$ for each $I' \subseteq I$
$x_i \geq 0$ for each $i \in I$

The dual of the LP is:

Maximize: $\sum_{I' \subseteq I} D_{I'} y_{I'}$ Subject to: $\sum_{I' \subseteq I, i \notin I'} v_i^{I'} y_{I'} \leq w_i$ for each $i \in I$
$y_{I'} \geq 0$ for each $I' \subseteq I$

This algorithm is a 2-approximation for the minimum knapsack problem. (The proof for this theorem is in the book "The Design of Approximation Algorithms", page 180, Proof of theorem 7.10.)
We can now consider a primal-dual algorithm for the problem using primal dual formulations as given above.
We start with an empty set of selected items, and a dual solution $y_{I'} = 0$ for all $I' \subseteq I$. Now we must select some dual variable to increase.
Which one should it be?
From the constraint of the dual algorithm:

$$\frac{w_i}{v_i^{I'}} \geq y_{I'}$$

Thinking about this, we suggest that we must calculate $w_i / v_i^{I'}$ of the elements and then choosing the variable corresponding to the minimal object of this $(w_i / v_i^{I'})$, we increase th dual variable.

Now a dual constraint will become tight for some item $i \in I$, and we will add this item to our set of selected items. We continue to increase $y_{I'}$ for some $I' = \{i\}$ and adding the items to the set whenever a dual constraint become tight for those items. The algorithm terminates when $V(I') \geq D$, and we return the set of the solutions.

Max-Knapsack Problem

This algorithm ends when adding the last item makes us reach a value that is greater then D. But we also know that the subset of the solution without the last item is also a result for the Min weight that reaches least the value $D = \sum_{i \in I'} v_i$, where I' is a subset without the last value.
So if we fixed D as the maximum value that D can take, $D = \sum_{i \in I} v_i$,

and change our algorithm to stop when adding a new item gets us over capacity C, returning the subset without this last element. In this case all the items of I are included in the solution, it will return the subset I. With this change we now have an algorithm to the Knapsack Problem, but we can not use the proof of the ratio=2 because we do not reach "at least" D as was before. We will try to calculate a ratio later on.
Because of this change on the algorithm we can now analyze:
We have $D = \sum_{i \in I} v_i$ and $v_i^{I'} = min(v_i, D - v(I'))$ with $i \in I \setminus I'$
but $D - v(I') \geq v(I') + v_i - v(I')$ (They are only equal if it is the last item)
then $v_i \leq D - v(I')$
so $v_i^I = v_i$

With this conclusion we can order the array of items at the beginning by their weight per value, $\frac{w_i}{v_i}$

Failed

By now we stop having a dual approach and we will create the Greedy approach algorithm based on this conclusions. This happens because at the beginning of the approach we start by analyzing the minimum (covering) Knapsack Problem. (It was assumed that at some point we could apply the constraint for the capacity)

2.3 Greedy Algorithm

Based on all the conclusions we reached at the last point, we can now make out greedy algorithm for the Knapsack Problem.
Input: Capacity C, Set of tuples S that each hold two integers, value v_i and weight w_i
Constraints: The sum of the weights in the solution subset is not greater than the capacity of the knapsack.
Goal: Find a solution with maximal value.
Output: The sum of the values of the items in the solution set.

1^{st} Greedy Algorithm

- order items by $\frac{w_i}{v_i}$ (increasingly)
- greedily add items until we hit an item that does not fit

Consider the instance: $S = \{(2, 1), (C, C)\}$, with Capacity C
The greedy algorithm will only pick the small item
optimal solution $S* = C$
computed solution $S* = 2$ (The two solutions may vary by an arbitrary amount, thus we cannot talk of an approximation scheme)

2^{nd} Greedy Algorithm

- order items by $\frac{w_i}{v_i}$ (increasingly)
- greedily add items until we hit an item that does not fit
- pick the best of $\{a_1, \ldots, a_{i-1}\}$ and $\{a_i\}$

This algorithm is a 2-approximation. Proof:

`https://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec10.pdf`

Theorem 10.2.2., page 2.
(The proof idea (as most of the time) is, that if two numbers add up to a third number, at least one of the two must be at least half as big as the third)

Algorithm 3 Greedy($S = \{(v_i, w_i) \dots (v_n, w_n)\}, C$)

$L \leftarrow$ order S by $(\frac{w_i}{v_i})$

$totalValue \leftarrow 0$

$titalWeight \leftarrow 0$

for each $item = (v_i, w_i)$ in L **do**
  if $totalWeight + w_i > C$ **then**
    if $totalValue < v_i$ **then**
      return v_i
    else if
      thenreturn $totalValue$
    end if
  else
    $totalWeight+ = w_i$
    $totalValue+ = v_i$
  end if
end for

The time complexity is $O(|l|log|l|)$
The space complexity is $O(|l|)$

3 Tests

The instances files to our tests have the format:
Capacity Size
v_1 w_1
v_2 w_2
...
v_i w_i

This instances where created by our generator that receives the size of elements, the range of the values, and the range of the weights, and compute a random instance. The capacity of the problem is also random generated, with a range between 1 and the sum of all weights.

We have generate the follow instances with the respective specifications:

- t1.txt:
  size=100; values = [1000000, 99000000]; weights = [100, 1000]
- t2.txt:
  size=100; values = [1000000, 99000000]; weights = [1, 10]
- t3.txt:
  size=100; values = [1000000, 99000000]; weights = [100, 1000]

4 Test Results

To provide the analyses of our results, we present a Table 1 bellow.
The results of an instance are similar from the different approaches. There for we can not reach any conclusion about the results values.
However we can say that the execution time of the scheme approach is real big when the parameter tends to 0. We can also conclude that the execution time of the for our greedy approach were always the best, with time 1.

Table 1: Results

Instance	Approx. Algor.	Parameter	Result	Execution Time
t1.txt	schemeKnapsack	0.01	1907396736	170
t1.txt	schemeKnapsack	0.9	335643768	8
t1.txt	greedyKnapsack	-	1907509910	1
t2.txt	schemeKnapsack	0.01	4202568	220
t2.txt	schemeKnapsack	0.9	4175996	9
t2.txt	greedyKnapsack	-	4197838	1
t3.txt	schemeKnapsack	0.01	4239144	189
t3.txt	schemeKnapsack	0.9	4843020	8
t3.txt	greedyKnapsack	-	4209330	1