MI-PAA: Assignment #3

Petr Hanzl
hanzlpe2@fit.cvut.cz

Czech Technical University - Faculty of Information Technology — February 16, 2019

Tasks

1. Prozkoumejte citlivost metod řešení problému batohu na parametry instancí generovaných generátorem náhodných instancí. Máte-li podezření na další závislosti, modifikujte zdrojový tvar generátoru.
2. Na základě zjištění navrhněte a proveďte experimentální vyhodnocení kvality řešení a výpočetní náročnosti.
3. Zkoumejte zejména následující metody:
   (a) hrubá síla (pokud z implementace není evidentní úplná necitlivost na vlastnosti instancí),
   (b) metoda větví a hranic, případně ve více variantách,
   (c) dynamické programování (dekompozice podle ceny a/nebo hmotnosti). FPTAS algoritmus není nutné testovat, pouze pokud by bylo podezření na jiné chování, než DP.
   (d) heuristika - poměr cena/váha.
4. Pozorujte zejména závislosti výpočetního času (případně počtu testovaných stavů) a rel. chyby (v případě heuristiky) na:
   (a) maximální váze věcí,
   (b) maximální ceně věcí,
   (c) poměru kapacity batohu k sumární váze,
   (d) granularitě (pozor - zde si uvědomte smysl exponentu granularity).
5. Doporučuje se zafixovat všechny parametry na konstantní hodnotu a vždy plynule měnit jeden parametr. Je nutné naměřit výsledky pro aspoň čtyři (opravdu minimálně) vhodně zvolené hodnoty parametru, jinak některé závislosti nebude možné vypozorovat.

Introduction

The knapsack problem is a basic algoritmic problem, taught at universities all around the world. Therefore, it is considered to be the basic knowledge of every computer science graduate. The knapsack problem has various modifications, the most common version is known as 0/1 knapsack problem.

0/1 knapsack problem

Given the knapsack weight capacity of W kilograms and a set of N items, each has its own unique index $i \in N^+$, a value of $v_i \in R^+$ money and weight of $w_i \in R^+$ kilograms. Determine which items should be put into the knapsack to maximize its value so that the sum of the weights is less than or equal to the weight capacity.

$$\text{maximize} \sum_{i=1}^{n} v_i x_i \tag{1}$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0, 1\} \tag{2}$$

Info: x_i represents whether the i-th item is present in the knapsack or not.

1 Bruteforce

To find the optimal solution for 0/1 knapsack problem, it means finding such a combination of binary elements x_i of a vector, such that no other vector with different elements has higher value.

Apparently, every vector of n binary values has 2^n possible combinations, thus naive brute force algorithm can find optimal solution for worst case in exponential time of 2^n, because every combination has to be checked by the algorithm. Even though some algorithms have lower computation times in average, none are discussed in this assignment as it is outside the scope of this problem.

Algorithm 1: `01Knapsack-Bruteforce`

Input: $(maxWeight, items[N])$
$maxWeight$ - a weight capacity;
$items[N]$ - a numbered set of N items, each with $value$ and $weight$
Result: $(maxValue, maxConfiguration)$
$maxValue$ - an optimal value of the knapsack;
$maxConfiguration$ - an integer whose bits represent presence of an i-th item in the knapsack

---

$maxValue \leftarrow 0$;
$maxConfiguration \leftarrow 0$;
for $configuration \leftarrow 0..2^N$ **do**
  $tmpValue \leftarrow 0$;
  $tmpWeight \leftarrow 0$;
  for *every* i*-th bit of* $configuration$ **do**
    if i*-th bit is set* **then**
      $tmpValue \leftarrow tmpValue + items[i].value$;
      $tmpWeight \leftarrow tmpWeight + items[i].weight$;
    end
  end
  if $(tmpWeight \leq maxWeight) \wedge (tmpValue \geq maxValue)$ **then**
    $maxValue \leftarrow tmpValue$;
    $maxConfiguration \leftarrow configuration$;
  end
end
return $(maxValue, maxConfiguration)$;

2 Branch and Bound

Although the time complexity of B&B algorithm is still $O(2^n)$ for the worst case, it reduces number of visited nodes in the state space by denying not reasonable choices. For the 0/1 knapsack problem the denied state sub-space is represented by sub-trees, that could not give a better solution than so far best given solution, even if all items were added to the knapsack.

Algorithm 2: `01Knapsack-BranchAndBound`

Input: $(maxWeight, items[N])$
$maxWeight$ - a weight capacity;
$items[N]$ - a numbered set of N items, each with $value$ and $weight$
Result: $(maxValue)$
$maxValue$ - an optimal value of the knapsack;

```
bestPrice ← 0;
sum ← 0;
for i ← items.length − 1; i ≥ 0; i++ do
    sum += items[i].price;
    maxProfitTable[i] = sum;
end
return rec(0,0,0);

Function rec(currentWeight, currentElement)
    if currentElement+1 > items.length then
        if bestPrice < currentPrice then
            bestPrice = currentPrice;
        end
        return 0;
    end
    if currentPrice + maxProfitTable[currentElement] < bestPrice then
        return 0;
    end
    if currentWeight+items[currentElement].weight > maxWeight then
        return rec(currentWeight, currentPrice, currentElement+1);
    end
    right ← items[currentElement].price + rec(currentWeight+items[currentElement].weight,
     currentPrice+items[currentElement].price, currentElement+1);
    left ← rec(currentWeight, currentPrice, currentElement+1);
    thisNodeMaxPrice ← Math.max(left, right);
    if bestPrice < thisNodeMaxPrice then
        bestPrice ← thisNodeMaxPrice;
    end
    return thisNodeMaxPrice;
```

3 Dynamic programming (Decomposition by price)

Dynamic programming has proven to be a very good approach for 0/1 knapsack problem. It has its limitations such that the prices (or weights) have to be only discrete values and the sum of prices is not extremely high. Specificity of dynamic programming is that it reuses already computed values that are continuously saved and stored in a 2d array. The time complexity for the version with decomposition by price is $O(n * P)$, where n is number of items and P is a sum of all prices. The time complexity for the version with decomposition by weight is $O(n * W)$, W is the capacity of knapsack.

Algorithm 3: `01Knapsack-Dynamic`

Input: $(maxWeight, items[N])$
$maxWeight$ - a weight capacity;
$items[N]$ - a numbered set of N items, each with $value$ and $weight$
Result: $(maxValue, maxConfiguration)$
$maxPrice, maxPriceConfiguration$ - a suboptimal value;

```
maxPossiblePrice ← items.reduce( (a,b) => a+b.price, 0);
W ← [];
for var i=0; i≤items.length; i++ do
    W[i] ← [];
    for price ← 0; price≤maxPossiblePrice; price++ do
        W[i][price] ← 0;
    end
end
for price ← 1; price≤maxPossiblePrice; price++ do
    W[0][price] = Infinity;
end
for n←1; n ≤ items.length; n++ do
    item←items[n-1];
    for price←0; price≤maxPossiblePrice; price++ do
        itemIncluded ← Infinity;
        itemNotIncluded ← W[n-1][price];
        if price ≥ item.price && W[n-1][price-item.price] != Infinity then
            itemIncluded ← W[n-1][price-item.price] + item.weight;
        end
        W[n][price] ← Math.min(itemIncluded, itemNotIncluded);
    end
end
for price ← maxPossiblePrice; price ≥ 0; price– do
    if maxWeight ≥ W[items.length][price] then
        maxPrice ← price;
        break;
    end
end
tempPrice ← maxPrice;
maxPriceConfiguration ← [];
for i ← items.length-1; i ≥ 0; i– do
    if W[i+1][tempPrice] != W[i][tempPrice] then
        maxPriceConfiguration[i] ← true;
        price ← price-items[i].price;
    end
    else
        maxPriceConfiguration[i] ← false;
    end
end
return {maxPrice, maxPriceConfiguration}
```

4 Heuristic approach

Finding the optimal solution might be impractical for applications where the goal is to find a satisfactory but immediate solution. Heuristic based algorithms can be used to speed up the process of finding these solutions. A good example of heuristic approach for 0/1 knapsack problem is to put most valuable items into knapsack at first.

The required heuristic for this assigment is to use ratio of value to weight. The algorithm below continously puts items into knapsack until the sum of weights of all the items in knapsack plus new possibly added item, is not greater than knapsack weight capacity. From the given results of heuristic algorithm, the average and the maximal approximation errors of the heuristic and bruteforce method are calculated and plotted onto a separated graph.

Algorithm 4: `01Knapsack-Heuristic`

Input: $(maxWeight, items[N])$
$maxWeight$ - a weight capacity;
$items[N]$ - a numbered set of N items, each with $value$ and $weight$
Result: $(maxValue, maxConfiguration)$
$maxValue$ - a suboptimal value;
$maxConfiguration$ - an integer whose bits represent presence of an i-th item in the knapsack

---

$sortedItems \leftarrow sortItemsByRatioOfValueToWeight(items)$;
$maxValue \leftarrow 0$;
$maxConfiguration \leftarrow arrayOfSize(N)$;
$maxConfiguration.fillWith(0)$;
$tmpWeight \leftarrow 0$
for $i = 0..N$ **do**
  if $(tmpWeight + sortedItems[i].weight \leq maxWeight)$ **then**
    $maxValue \leftarrow maxValue + sortedItems[i].value$;
    $maxConfiguration[i] \leftarrow 1$;
  end
end
return $(maxValue, maxConfiguration)$;

5 Assumption

Every method for solving knapsack problem has its weaknesses. These weaknesses are well-known for each method, but it can vary for different algorithms or implementations respectively. The summary underneath shows, which parameters of the given knapsack problem generator were chosen to experimentally proof the assumptions.

5.1 Branch and Bounch

This method counts on temporary best found price and sum of weight of all not-yet added items. Branch and Bound can not sufficiently deal with instances, which do not allow significantly prune branches. Prunning of branch happens when total sum of prices of not-yet added items and actual price is lower than the best yet found price. Apparently, the ratio of knapsack capacity against the sum of all weight seems to be the most reasonable parameter to be tested.

5.2 Dynamic programming (decomposion by price)

The time complexity of the dynamic programming depents on the sum of prices of all items and number of items. More specificaly, it depents on the size of the table with dimension of **number_of_items** * **sum_of_prices**. We could measure both parameters, but the sum of prices seems to be more reasonable, because it can rise much easier than the number of items.

5.3 Heuristic approach

The heuristic approach has always the same time complexity (only the larger numbers of items has an influence on the time complexity). Thus, error rate is measuared instead. I assumed, that the heuristic approach may gives suboptimal results with lower error rate for instances with smaller ratio of knapsack capacity againts the total sum of prices. Also, the error rate could possibly depends on a granulatiry of items.

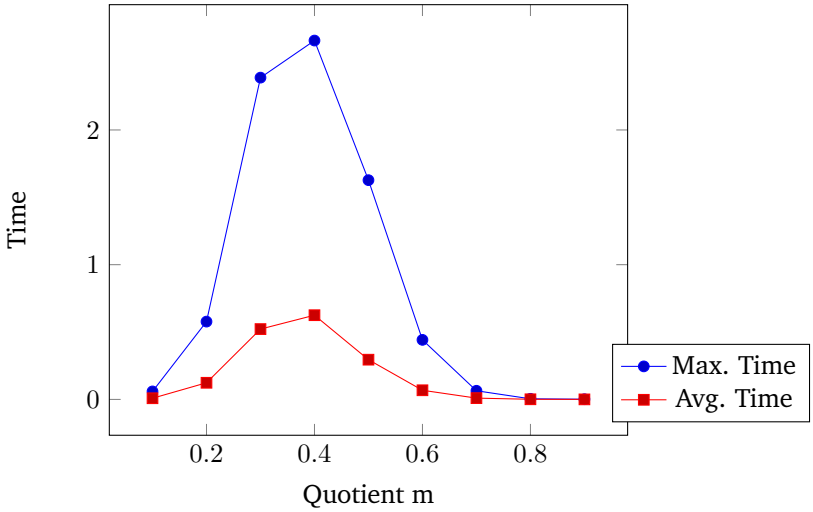
Summary

	Branch and Bound	**DP**	**Heuristic (1)**	**Heuristic (2)**
n (Number of items)	35	50	100	100
N (Number of instances)	50	50	100	100
m (ratio n:sum of w_i)	0.1 . . . 0.9	0.5	0.1 . . . 0.9	0.5
W (max w_i)	1000	100	1000	1000
C (max v_i)	10000	10 . . . 10000	1000	1000
k (exponent)	1	1	1	0.1 . . . 0.9
d (heavier items)	0	0	0	0

6 Results

6.1 Branch and Bound - quotient m

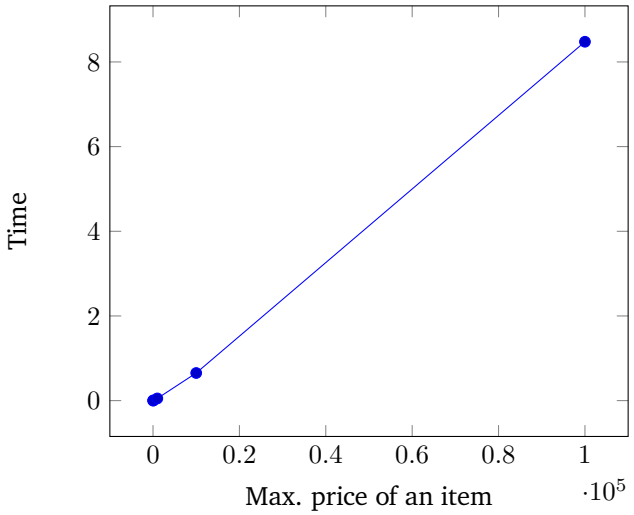
Graph 1 - Dependency of computational time on quotient m



m quotient	Avg. Time	Max. Time
0.1	0.008584	0.059272
0.2	0.12345	0.577465
0.3	0.521912	2.38835
0.4	0.624982	2.663614
0.5	0.294825	1.627344
0.6	0.067811	0.441936
0.7	0.009491	0.0641
0.8	0.000913	0.00446
0.9	0.000222	0.001423

6.2 Dynamic programming - maximal price of a single item

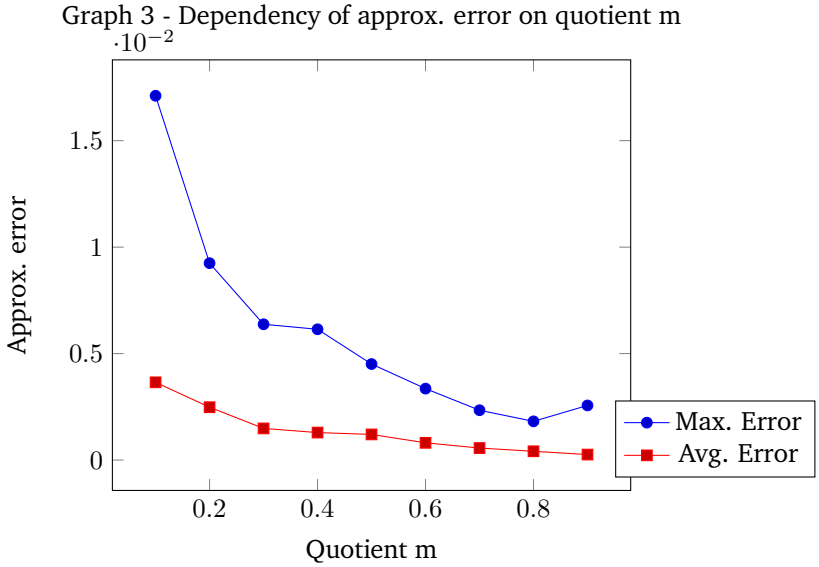
Graph 2 - Dependency of computational time on maximal price of a single item



Max. price of an item	Avg. Time	Max. Time
10	0.001094	0.012613
100	0.003862	0.022973
1,000	0.051513	0.097261
10,000	0.652274	0.897291
100,000	8.476636	8.651104

6.3 Heuristic approach - quotient m

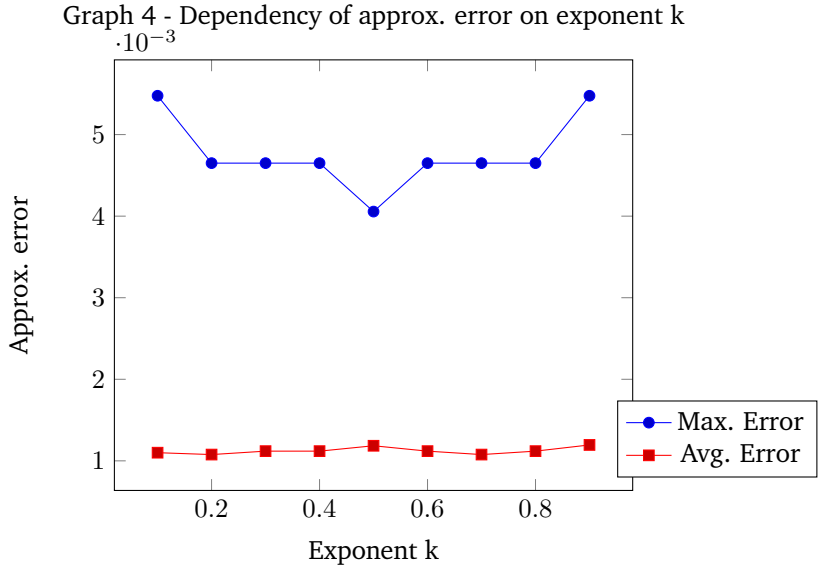
Graph 3 - Dependency of approx. error on quotient m



m quotient	Avg. Error	Max. Error
0.1	0.003651	0.017109
0.2	0.002482	0.009247
0.3	0.001486	0.006377
0.4	0.001291	0.006142
0.5	0.001204	0.004509
0.6	0.00081	0.003355
0.7	0.000564	0.00234
0.8	0.000412	0.001818
0.9	0.000258	0.002565

6.4 Heuristic approach - exponent k

Graph 4 - Dependency of approx. error on exponent k



k exponent	Avg. Error	Max. Error
0.1	0.0011	0.005476
0.2	0.001077	0.00465
0.3	0.001119	0.00465
0.4	0.001119	0.00465
0.5	0.001185	0.004056
0.6	0.001119	0.00465
0.7	0.001077	0.00465
0.8	0.001119	0.00465
0.9	0.001195	0.005476

7 Conclusion

The results of the experiments have proven the assumptions. It can be seen that the computational time for Branch and Bound method depends on the ratio of sum of all weights to knapsack capacity. The peak seems to be at 0.4 and then the average and maximal error rapidly falls. The rapid fall ends around 0.7.

The results have also proven, that the computation time for dynamic programming depends polynomially on the maximal price of an item.

I measured two parameters for the heuristic approach and it has proven the assumption, that the approximation error goes down for the instances, which have total sum of weights closer to knapsack capacity. Average approximation error seems fall linearly and the maximal approximation error tends to fall in an exponantional rate. The last parameter I measured, is the k exponent, which controls the granuality of weight in the set of items. The measurement for this parameter did not show any significant dependancy, expect the rise of the approximation error on the edges of measured interval. It seems that the granuality has almost no significant influence on the approximation error.