Fundamental Computer Science

Malin Rau and Denis Trystram

9th of March, 2020

Introduction to Approximation Algorithms

Decision Problem vs Optimization Problem

So far: **Decision problems**:

- Is there a Vertex Cover of size k in G?
- Is the given formula satisfiable?

Now: **Maximization or Minimization problems**:

- Find a smallest $\textsc{Vertex Cover}$ in G.
- Find a largest $\textsc{Clique}$ in G.
- Find the largest $\textsc{Independent Set}$ in G.

Obstacle: For all the problems for which the decision variant is NP-hard, we cannot hope to find an polynomial time algorithm to solve the corresponding maximization or minimization problem, unless $P = NP$.

Why?

Approximation Algorithms: Definition

An algorithm A for a **minimization** problem Π is called α-approximation if for each instance $I \in \Pi$ it holds that

$$A(I) \leq \alpha \cdot \mathrm{OPT}(I)$$

Examples: Vertex Cover, Bin Packing

An algorithm A for a **maximization** problem Π is called α-approximation if for each instance $I \in \Pi$ it holds that

$$\alpha \cdot A(I) \geq \mathrm{OPT}(I)$$

Examples: Independent Set, Clique, Max-2Sat, Knapsack

The $\{0, 1\}$-Knapsack Problem

- Given: A container (knapsack) of size $B \in \mathbb{N}$, and a set of items $\mathcal{I}$, such that each $i \in \mathcal{I}$ has a size $s(i) \in \{1, \ldots, B\}$ and a profit $p(i) \in \mathbb{N}$.
- Decision Problem: Is there a subset $I' \subset I$ that fits inside the container and has profit P?
- Optimization Problem: Find a subset $I' \subset I$ that fits inside the container and maximizes the profit of the items.

$$\max_{I' \subseteq I} \sum_{i \in I'} p(i)$$

$$\text{subject to } \sum_{i \in I'} s(i) \leq B \quad (I' \text{ fits inside the container})$$

Remark:
Each item fits inside the bin on its own since $s(i) \leq B$ for each $i \in \mathcal{I}$.

Example

Knapsack size: 15

items	1	2	3	4	5
size	12	2	1	1	4
profit	4	2	2	1	10

Optimum?

Take the items 2,3,4,5.
Total Profit: 15
Total Size: 8

$\{0,1\}$-KNAPSACK is NP-hard

SUBSET SUM (Decision Problem)
Given: A set of positive integer numbers $I = \{i_1, \ldots, i_n\}$, a positive number S
Question: Is there a subset $I' \subseteq I$ such that the sum of the numbers in I' equal S, i.e., $\sum_{i \in I'} i = S$?

Theorem

SUBSET SUM *is NP-complete.*

Exercise:
Prove: If there exists a polynomial time algorithm that solves the optimization problem $\{0,1\}$-KNAPSACK, then there exists a polynomial time algorithm that decides SUBSET SUM.

Corollar

There exists no polynomial time algorithm for the optimization problem $\{0,1\}$-KNAPSACK *unless* $P = NP$.

Solution of the exercise

Proof:
In the following we describe an algorithm that decides the Subset Sum problem in polynomial time if there exits a polynomial time algorithm A that finds the optimal solution for each instance of the $\{0,1\}$-Knapsack problem.

- Given an instance $(I = \{i_1, ..., i_n\}, S)$ of the Subset Sum problem generate an instance of the knapsack problem as follows:
  - Define $B := S$.
  - For each $i_j \in I$ define one item j with profit $p(j) := i_j$ and size $s(j) := i_j$ and define $\mathcal{I}$ as the set of all these items.
- Solve the generated instance optimally with the polynomial time algorithm for $\{0,1\}$-Knapsack.
- If the packed profit equals B return Yes otherwise return No.

The above algorithm works in time polynomial of the input size of (I, S).

Solution of the exercise

It reminds to be shown that the algorithm is correct.

If (I, S) is a yes-instance there exists a set of items $I' \subseteq I$ such that $\sum_{i \in I'} i = S$. The corresponding items all fit inside the container. Hence the solution of the algorithm for the $\{0, 1\}$-$\textsc{Knapsack}$ problem has at least profit $S = B$. On the other hand, the container cannot contain a set with larger profit, since all the items have the same profit as size. Hence the algorithm returns $\textsc{Yes}$ in this case.

On the other hand, if the $\{0, 1\}$-$\textsc{Knapsack}$ algorithm returns $\textsc{Yes}$, it has a solution with profit $B = S$. Hence, there exists a set of items which profits and sizes sum up to exactly S. Therefore, there exists a subset $I' \in I$ with $\sum_{i \in I'} i = S$. As a consequence, the given instance (I, S) is a yes-instance.

$\square$

A First Algorithmic Idea

Define the *efficiency* of an item as $e(i) := p(i)/s(i)$.

Algorithm NaiveGreedy:
Sort the items by efficiency. Greedily take the most efficient item until the next item does not fit inside the container.

Exercise:
Prove that this algorithm has no constant approximation ratio.

Hint 1: Denote by $\mathrm{NaiveGreedy}(I)$ the profit of the solution generated by the above algorithm for an instance I, and denote by $\mathrm{OPT}(I)$ the optimal profit for this instance. Prove that for each k there exists an instance such that $k\mathrm{NaiveGreedy}(I) < OPT(I)$.

Hint 2: The corresponding instance consists only of 2 items!

Exercise Solution

Proof.

Assume for contradiction that the above algorithm has a constant ratio of k for some $k > 0$.

Consider the following instance: $B = 2k + 1$, $\mathcal{I} = \{i, i'\}$, $p(i) = 2$, $s(i) = 1$, $p(i') = 2k + 1$, $s(i') = 2k + 1$.

Item i has an efficiency of $e(i) = \frac{p(i)}{s(i)} = 2$, while item i' has an efficiency of $e(i') = \frac{p(i')}{s(i')} = 1$. Therefore, the algorithm will choose the item i, while the optimal algorithm will choose item i'. It holds that $k\mathrm{NaiveGreedy}(I) = k \cdot 2 < 2k + 1 = \mathrm{OPT}(I)$. Hence, the algorithm is not a k-approximation.

Since we have shown for each constant $k > 0$, the algorithm NaiveGreedy does not have a constant approximation ratio. $\square$

Improved Algorithm

Algorithm ImprovedGreedy:

- Sort the items by efficiency.
- Define a first solution S_1 by greedily taking the most efficient item until the next item does not fit inside the container.
- Define a second solution S_2 that only contains the item with the largest profit.
- Return the solution S_1 or S_2 that has the maximum profit among these two.

Theorem

The above algorithm ImprovedGreedy has an approximation ratio of 2.

Proof

What to prove?

For each instance I it holds that $2A(I) \geq \mathrm{OPT}(I)$, where $A(I)$ is the profit of the solution generated by the algorithm and $\mathrm{OPT}(I)$ is the optimal profit for that instance.

- Let I be any instance of the knapsack problem.
- Consider the following set of items I' that contains all the items from the solution S_1 and the next item $i_\top$ that did not fit into the bin.
- The set I' is no solution to the problem, since the items do not fit inside the bin.
- It holds that $p(I') \geq \mathrm{OPT}(I)$, where $p(I')$ is the summed profit of the items in I', since there is no space left inside the bin and we took the most efficient items.
- Now consider the set of items $S_1 \cup S_2$. It holds that $p(S_1 \cup S_2) \geq p(I') \geq \mathrm{OPT}(I)$, since $p(S_2) \geq i_\top$ because it contains the item with the largest profit.
- If $p(S_1 \cup S_2) \geq \mathrm{OPT}(I)$, one of the solutions has to be larger than $\mathrm{OPT}(I)/2$.
- As a consequence $2A(I) \geq \mathrm{OPT}(I)$.

Dynamic Program for Knapsack

Idea

- Construct a two dimensional table `T`.
- Entry `T[p][i]` contains the minimum size that is needed to gain profit p with the first i items and is $\infty \hat{=} B+1$ if this profit cannot be reached.
- Optimum profit can be found at the last entry in the row n that is not ∞.
- Recursive formula:
  $T(p,i) = \min\{T(p,i-1), T(p-p(i),i-1)+s(i)\}$

Dynamic Program for Knapsack

Initialization

```
input: p[], s[], n, B

int pMax =0;
for i = 0 to n-1 {
  pMax += p[i];
}
initialize T with size [pMax][n];
for i = 0 to n-1{
  T[0][i] = 0;
}
for p = 1 to pMax{
  T[p][0] = B+1;
  if p = p[0] {
    T[p][0] = s[0];
  }
}
```

Dynamic Program for Knapsack

Filling the rest of the table

```
for p = 1 to pMax{
  for i = 1 to n-1{
    T[p][i] = T[p][i-1]
    if p-p[i] >= 0 && T[p][i]>T[p-p[i]][i-1] + s[i]{
      T[p][i]= T[p-p[i]][i-1] + s[i]
    }
  }
}
```

Finding the largest possible profit

```
p = pMax;
while T[p][n-1]>B{
  p--;
}
(return p)
```

Dynamic Program for Knapsack

Finding the set of items

```
list items = new list();
i = n-1
while p>0 && i>0 {
  if T[p][i] == T[p][i-1]{
    i = i-1;
  }
  else{
    list.add(i);
    p = p-p[i];
  i = i-1;
  }
}
if p>0 && i=0{
  list.add(i);
}
return list;
```

Remarks to the dynamic program

Observation 1:
Instead of using the sum $P_{sum} := \sum_{i=1}^{n} p(i)$ as the maximal reachable value $P_{\max}$, we can find the solution to the 2-approximation P_2 and double it, i.e., $P_{\max} := \min\{P_{sum}, 2P_2\}$.

Observation 2:
We can improve the running time a little by remembering the largest profit P_{i-1} of the previous row and stop the calculation at $P_{i-1} + p(i)$. (This is useful when sorting the items by increasing profit)

Does this mean $P = NP$?

No!

Time complexity of above dynamic program:
$\mathcal{O}(n \cdot \sum_{i=1}^{n} p(i))$.

(Binary) encoding length of $\{0,1\}$-Knapsack:
$\log(B) + \sum_{i=1}^{n} \log(p(i)) + \log(s(i))$.

Consequence:
The dynamic program might be exponential in the encoding length of the problem, if there exist a profit that is larger than a polynomial in n, e.g., $p(i) = 2^n$ for some $i \in \{1, \ldots, n\}$.

Observation:
The algorithm is polynomial in the input size if the problem is encoded in unary. Unary encoding means that we need n symbols to encode the number n, i.e., the unary encoding length of $\{0,1\}$-Knapsack is given by $B + \sum_{i=1}^{n}(p(i) + s(i))$. The time complexity of algorithms which run in polynomial time in unary encoding is called **pseudo-polynomial**.

An $(1+\varepsilon)$-approximation for knapsack

Problem with the above dynamic program: The profit is to large.
Idea: Scale the profit down.

$(1+\varepsilon)$-approximation for Knapsack (Due to Kim and Ibarra)

- For some given error parameter $\varepsilon > 0$ define $k := \left\lfloor \frac{n}{\varepsilon} \right\rfloor$
- For every item $i \in \{1, \dots, n\}$, define $\hat{p}(i) := \left\lfloor \frac{p_i k}{p_{\max}} \right\rfloor$.
- Run the above dynamic program with the $\hat{p}$ as the profits for the items to get some optimal $\hat{S}$.
- return $\hat{S}$

Theorem

The above algorithm is an $\mathcal{O}(1+\varepsilon)$-approximation.

Proof of the theorem

- Let $\hat{S}$ be the solution computed by the algorithm and let OPT be an optimal solution.
- Since we obtain an optimal solution to the problem with the scaled profits we can deduce

$$\sum_{i\in\hat{S}} \hat{p}(i) \geq \sum_{i\in\mathrm{OPT}} \hat{p}(i)$$
$$\left(\frac{p_{\max}}{k}\right)\sum_{i\in\hat{S}} \hat{p}(i) \geq \left(\frac{p_{\max}}{k}\right)\sum_{i\in\mathrm{OPT}} \hat{p}(i)$$

- For the algorithms solution it holds that

$$\sum_{i\in\hat{S}} p(i) \geq \left\lfloor \sum_{i\in\hat{S}} \frac{p_i k}{p_{\max}} \right\rfloor \frac{p_{\max}}{k} \geq \frac{p_{\max}}{k} \sum_{i\in\hat{S}} \hat{p}(i)$$

Proof of the theorem

- On the other hand, we know that

$$\begin{aligned}
\left(\frac{p_{\max}}{k}\right)\sum_{i\in \mathrm{OPT}}\hat{p}(i) &= \left(\frac{p_{\max}}{k}\right)\sum_{i\in \mathrm{OPT}}\left\lfloor\frac{p_i k}{p_{\max}}\right\rfloor \\
&\geq \left(\frac{p_{\max}}{k}\right)\sum_{i\in \mathrm{OPT}}\left(\frac{p_i k}{p_{\max}}-1\right) \\
&\geq \sum_{i\in \mathrm{OPT}} p(i) - \sum_{i\in \mathrm{OPT}}\frac{p_{\max}}{k} \\
&\geq \sum_{i\in \mathrm{OPT}} p(i) - n\cdot\frac{p_{\max}}{k} \\
&\geq \sum_{i\in \mathrm{OPT}} p(i) - \varepsilon p_{\max}
\end{aligned}$$

- Since $p_{\max} \leq \mathrm{OPT}$ it holds that

$$\sum_{i\in\hat{S}} p(i) \geq (1-\varepsilon)\mathrm{OPT}$$

Time Complexity of the algorithm

Theorem

The time complexity of the algorithm is $\mathcal{O}(n^3/\varepsilon)$

Proof.

The largest rounded profit is $\lfloor n/\varepsilon \rfloor$ and hence `pMax` is bounded by n^2/ε. As a consequence the table has a size of $\mathcal{O}(n^3/\varepsilon)$. $\square$

PTAS and FPTAS

Definition (Approximation Scheme)

An algorithm is an approximation scheme for a problem if, given some parameter $\varepsilon > 0$, it acts as a $O(1+\varepsilon)$-approximation.

Definition (PTAS)

An approximation scheme is a polynomial time approximation scheme (PTAS) if for each *fixed* $\varepsilon > 0$, the running time is bounded by a polynomial in the size of the problem.

Remark:
This includes running times as $\mathcal{O}(n^{1/\varepsilon})$ or even $\mathcal{O}(n^{1/\varepsilon^{1/\varepsilon}})$, since the value $1/\varepsilon$ is considered a constant and not part of the problem.

Definition (FPTAS)

A fully polynomial time approximation scheme (FPTAS) is a PTAS with a running time that is bounded by a polynomial in the size of the problem **and** $1/\varepsilon$.

More on FPTASes

Remark:
The above algorithm for the knapsack problem is an FTPAS. It is a $\mathcal{O}(1+\varepsilon)$-approximation and it has a running time that is polynomial in the size of the input and $1/\varepsilon$.

Remark:
Only problems for which a pseudo-polynomial exact algorithm exist admit an FPTAS. These problems are called *weakly* NP-hard.

Definition (strongly NP-hard)

A problem is strongly NP-hard if every problem in NP can be polynomial reduced to it in such a way that numbers in the reduced instance are all written in unary.

Theorem

A strongly NP-hard problem admits no FPTAS and no pseudo-polynomial time exact algorithm for its optimization variant unless $P = NP$.

Minimum Makespan Scheduling ($P||C_{\max}$)

Given:

- m identical machines
- A set $\mathcal{J}$ if jobs. Each job $i \in \mathcal{J}$ has a processing time $p(j)$ and needs one machine to be processed.

Objective:
Find a schedule (assignment from jobs to machines) such that the largest total load on the machines is minimized. The total load of a machine m_i is the sum of all processing times assigned to this machine.

Hardness of $P||C_{\max}$

3-Partition
Given: An integer B and a multiset $\mathcal{I}$ of $3n$ integers with values in the open interval $(B/4, B/2)$ with $\sum_{i \in \mathcal{I}} = n \cdot B$.
Question: Is there a partition into n multisets (each containing exactly three integers) such that the integers in each set sum up to B?

Theorem

3-Partition *is strongly NP-complete*

Exercise:
Prove that the decision variant of $P||C_{\max}$ is strongly NP-complete.

Solution of the Exercise

To show that the decision variant of $P||C_{\max}$ is strongly NP-complete, we will prove that $3\text{-}\textsc{Partition} \leq_P P||C_{\max}$.

Given an instance $(B, \mathcal{I})$ of $3\text{-}\textsc{Partition}$, we define the following instance for $P||C_{\max}$:

- define $m := |\mathcal{I}|/3$
- define for each item $i \in \mathcal{I}$ one job j_i with processing time $p(j_i) = i$.
- Question: is there a schedule with makespan B?

Solution of the Exercise

We now have to prove that the instance of 3-Partition is a yes-instance **if and only if** the generated instance for $P||C_{\max}$ is a yes-instance.

If the 3-Partition instance is a yes-instance, then there exists a partition of the items into $|\mathcal{I}|/3$ sets such that the numbers in each set sum up to B. When we assign each of these sets to one machine the schedule has a makespan of B. Furthermore, there exists no schedule with makespan smaller than B. As a consequence, the $P||C_{\max}$ instance is a yes-instance.

If the $P||C_{\max}$ instance is a yes-instance, then there exists a schedule with makespan at most B. Since $\sum_{i \in \mathcal{I}} = n \cdot B$ each machine has a load of at least B in this schedule. As a consequence, partitioning the numbers $\mathcal{I}$ into the sets corresponding to the sets of jobs for the machines delivers a partition as required by the 3-Partition problem and hence it has to be a yes-instance as well.