CSE515 Advanced Algorithms
Lecture 19
The Knapsack Problem

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

May 4, 2021

Introduction

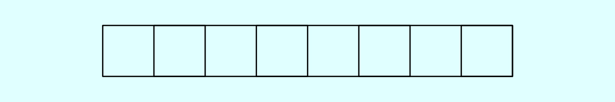
- Assignment 3 will be graded by Thursday.
- Assignment 4 will be posted on Friday.
- Reference: Section 3.1 in The design of approximation algorithms by David P. Williamson and David B. Shmoys.

Example

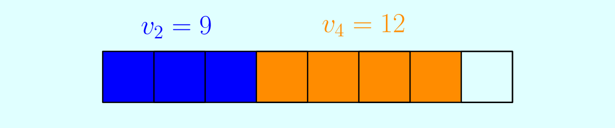
- INPUT:

object	1	2	3	4	5	6	7
size	2	3	3	4	5	6	8
value	1	9	8	12	10	19	12

  and a knapsack of capacity $B = 8$.



- OUTPUT: a subset of the objects with total size at most $B = 8$ and maximum value.
- Optimal answer: $S = \{2, 4\}$, size 7, value 21.



Notation

INPUT:

- A set $I = \{1, \ldots, n\}$ of *objects*.
- Each object i has an integer *size* $s_i > 0$ and an integer *value* $v_i > 0$.
- The *capacity* B of the knapsack.

Problem (Knapsack problem)

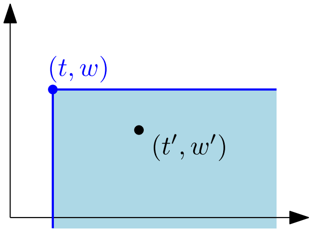
Find a subset of objects $S \subseteq I$ with maximum value $\sum_{i \in S} v_i$, under the constraint $\sum_{i \in S} s_i \leqslant B$.

- The knapsack problem is **NP**-hard, so we will try to find an approximation algorithm.

Dominated Pairs

Definition (Dominated pairs)

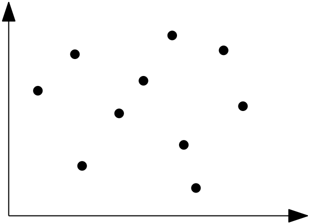
A pair (t, w) is said to *dominate* pair (t', w') if $t \leqslant t'$ and $w \geqslant w'$. This relation is denoted by $(t', w') \prec (t, w)$.



- Idea: if $s_1 < s_2$ and $v_1 > v_2$, then object 1 is clearly better than object 2, as it is smaller and more valuable.
- Remark: related to the notion of maxima of a point-set, or the skyline problem.

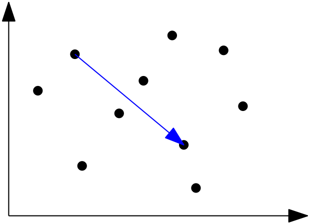
Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Example:



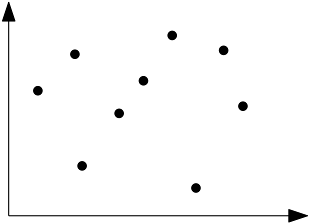
Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Example:



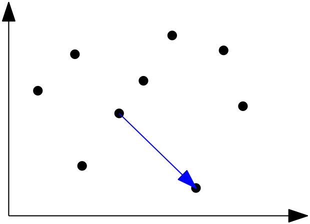
Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Example:



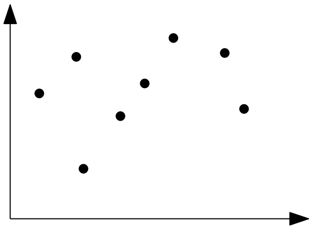
Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Example:



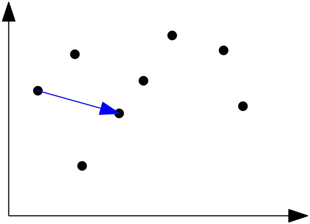
Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Example:



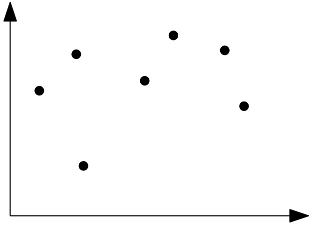
Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Example:



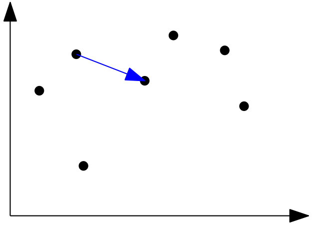
Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Example:



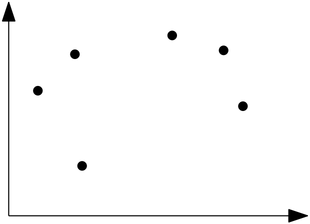
Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Example:



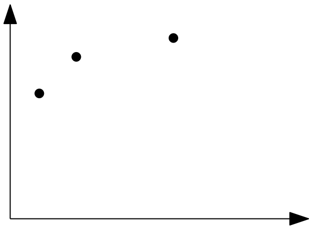
Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Example:



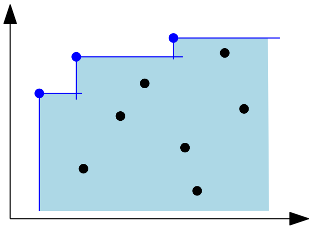
Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Output:



Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.
- Output:



- The input points are below a "staircase" defined by the three output points.

Removing Dominated Pairs

- Our algorithm maintains a set A of pairs, and removes from it any dominated pair.

Removing dominated pairs

procedure $\textsc{RemoveDominatedPairs}$(A)
  while there exists two pairs $(t_1, w_1) \prec (t_2, w_2)$ in A **do**
    remove (t_1, w_1) from A.

A Dynamic Programming Algorithm for Knapsack

Dynamic programming algorithm for knapsack

$A(1) \leftarrow \{(0,0),(s_1,v_1)\}$
for $j \leftarrow 2, n$ **do**
    $A(j) \leftarrow A(j-1)$
    for each $(t,w) \in A(j-1)$ **do**
        if $t + s_j \leqslant B$ **then**
            insert $(t+s_j, w+v_j)$ into $A(j)$.
    $\textsc{RemoveDominatedPairs}(A(j))$.
return $\max_{(t,w)\in A(n)} w$

- A pair (t,w) in $A(j)$ indicates that there is a set $S \subseteq \{1,\ldots,j\}$ that uses space exactly $t \leqslant B$ and has value w.

A Dynamic Programming Algorithm for Knapsack

object	1	2	3	4	5	6	7
size	2	3	3	4	5	6	8
value	1	9	8	12	10	19	12

$B = 8$

We obtain the following values of $A(j)$:

- $A(1) = \{(0,0),(2,1)\}$
- $A(2) = \{(0,0),(2,1),(3,9),(5,10)\}$
- $A(3) = \{(0,0),(2,1),(3,9),(5,10),(6,17),(8,18)\}$
- $A(4) = \{(0,0),(2,1),(3,9),(4,12),(6,17),(7,21)\}$
- $A(5) = \{(0,0),(2,1),(3,9),(4,12),(6,17),(7,21)\}$
- $A(6) = \{(0,0),(2,1),(3,9),(4,12),(6,19),(7,21)\}$
- $A(7) = \{(0,0),(2,1),(3,9),(4,12),(6,19),(7,21)\}$

A Dynamic Programming Algorithm for Knapsack

Problem

This algorithm only returns the value of the optimal solution. How do we recover an optimal subset $S^* \subseteq I$?

- We can trace back an optimal solution using the $A(j)$'s. It can be done without increasing the running time, as is often the case with dynamic programming. (See for example histogram construction in the notes on Lecture 4.)

Proof of Correctness

Lemma

The dynamic programming algorithm returns the optimal value OPT to the knapsack problem.

- Proof done in class.

Analysis

- Let $V = \sum_{i=1}^{m} v_i$. With a careful implementation of the dominated-pair removal procedure:

Lemma

The running time of the dynamic programming algorithm for knapsack is

$$O(n \times \min(B, V)).$$

- Proof done in class.

Running Time

Is it polynomial?

- No, because B is usually encoded into $\log_2 B$ bits so B can be exponential in the input size.
- If the input is encoded in unary:
  6 is encoded by 111111 (instead of 101 in binary).

We say that this algorithm is *pseudopolynomial*:

Definition (Pseudopolynomial algorithm)

An algorithm is said to be pseudopolynomial if its running time is polynomial on the size of the input when the input numbers are encoded in unary.

- A pseudopolynomial algorithm can often be turned into an efficient approximation algorithm through *rounding*. (See next slides.)

Approximation Schemes

- We consider a maximization problem for a non-negative function f over a domain $\mathcal{D}$: We want to compute $x^* \in \mathcal{D}$ such that $f(x^*) = \max_{\mathcal{D}} f$. We denote by n the input size. Recall that

Definition (α-approximation algorithm)

When $0 < \alpha < 1$, an α-approximation algorithm is an algorithm that returns $x \in \mathcal{D}$ such that $f(x) \geqslant \alpha \max_{\mathcal{D}} f$ in time polynomial in n.

- We will obtain a stronger result for the knapsack problem: a fully polynomial-time approximation scheme (FPTAS).

Approximation Schemes

Definition (FPTAS)

A *fully polynomial-time approximation scheme* (FPTAS) is an algorithm that takes an extra parameter $0 < \varepsilon < 1$, and returns $x \in \mathcal{D}$ such that $f(x) \geqslant (1-\varepsilon) \max_{\mathcal{D}} f$ in time polynomial in n and $1/\varepsilon$.

- This is difference from an α-approximation algorithm, because α is fixed, while ε is an input parameter, assumed to be small.
- For instance $\varepsilon = 1/100$ means that a 1% error is acceptable.

Rounding

Idea:

- We will map the input values v_i to a small set of integers, and then apply the dynamic programming algorithm above.

Reduction:

- Let $\mu > 0$ be an arbitrary positive rational number.
  - Intuitively, it is the granularity we will use.
- We replace each value v_i with $v_i' = \lfloor v_i/\mu \rfloor$.
- We run the dynamic programming algorithm using sizes s_i and values v_i'.
- We obtain an optimal set $S_\mu \subset I$ for this rounded instance.
- We output S_μ as the approximate solution the the original problem.
- With an appropriate choice of μ, we will show that it is an FPTAS.

Proof

- We denote by OPT the optimal value to the original (non-rounded) problem, and we denote $M = \max_{i \in I} v_i$.
- We assume that the size s_i of each object is at most B (otherwise we can discard this object as it does not fit in the knapsack), hence $OPT \geqslant M$.

Lemma

$$\sum_{i \in S_\mu} v_i \geqslant \left(1 - \frac{n\mu}{M}\right) OPT.$$

- Proof done in class.

Result

Theorem

The rounding algorithm is an FPTAS for knapsack. More precisely, if we set $\mu = \varepsilon M/n$, it returns a solution which is at least $(1-\varepsilon)$ times the optimal in time $O(n^3/\varepsilon)$.