601.435/635 Approximation Algorithms **Lecturer:** Michael Dinitz
Topic: Rounding and Dynamic Programming: Knapsack **Date:** 2/13/24
Scribe: Michael Dinitz

7.1 Introduction

This week we're going to talk about two related algorithmic ideas: data rounding, and dynamic programming. These can be considered separately, but often go together, so we're going to combine them. You should all be familiar with dynamic programming, in which we break a problem up into overlapping subproblems, solve each subproblem, and then use those solutions to solve the overall problem. It might seem a bit mysterious how to use dynamic programming for approximation algorithms, though: if we lose a small amount at every "level" of the dynamic program, then we could end up losing an enormous amount overall. Dynamic programming lends itself much more to exact solutions. So, instead, we'll use dynamic programming to find an optimal solution to a slightly different problem, and then prove that the optimal solution to think different problem must be close to the optimal solution of the problem that we actually care about.

The other technique that we will use, data rounding, works well with dynamic programming. Consider some problem that involves "numbers" (e.g., weights) rather than just combinatorial objects. Then there are sometimes (dynamic programming) algorithms that can find the optimal solution, but take time that is polynomial in the numbers themselves (rather than their representations), and so are not in polynomial time overall. But if we can "round" the numbers to make it so that while the numbers may be big, there aren't too many different possibilities for what they could be, we can often give a dynamic programming algorithm that is much more efficient.

7.2 Knapsack Problem

7.2.1 Problem Definition

The most famous use of these techniques are for the well-known Knapsack problem:

- **Input**: Items $[n]$, profits $p : [n] \to \mathbb{N}$, sizes $s : [n] \to \mathbb{N}$, and the size of the knapsack $k \in \mathbb{N}$.
- **Feasible Solutions**: $I \subseteq [n]$ such that $\sum_{i \in I} s(i) \leq k$
- **Objective**: Maximize $\sum_{i \in I} p(i)$

7.2.2 Greedy Algorithms for the Knapsack Problem

Before we give our new approximation algorithms, let's consider what would happen if we started with some obvious greedy algorithms.

Algorithm 1 Greedy KNAPSACK Algorithm 1

Input: $[n]$, p, s, k. We assume $[n]$ is sorted by nonincreasing $\frac{p(i)}{s(i)}$
Output: $I \subseteq [n]$

```
I ← ∅
for i ∈ [n] do
  if Σ_{j∈I} s(j) + s(i) ≤ k then
    I ← I ∪ {i}
  end if
end for
return I
```

Theorem 7.2.1 *This algorithm is an $\Omega(k)$-approximation.*

Proof: Consider an instance of this problem such that there are exactly two items, such that $s(1) = 1$, $p(1) = 2$, $s(2) = k$, and $p(2) = k$. In such an instance, the algorithm chooses item 1 instead of 2 (the optimal solution). Thus the value of the algorithm is 2 while the optimal value is k, so it is no better than a $k/2$-approximation. ■

This algorithm performed poorly because the item right *after* it got stuck was the actual correct item to add. The next algorithms adds the items in greedy order (as before), but once it gets stuck it simply compares what it chose to the next item and chooses the better of the two.

Algorithm 2 Greedy KNAPSACK Algorithm 2

Input: $[n]$, p, s, k. We assume $[n]$ is sorted by nonincreasing $\frac{p(i)}{s(i)}$
Output: $I \subseteq [n]$

```
i = 1
while Σ_{j<i} s(j) + s(i) ≤ k do
  i ← i + 1
end while
if Σ_{j<i} p(j) > p(i) then
  return {1, 2, ..., i − 1}
else
  return {i}
end if
```

Theorem 7.2.2 *This algorithm is a 2-approximation.*

Proof: Let i^* be the first item that cannot fit into the knapsack. Because the items are arranged in decreasing order with respect to their "bang-for-buck", the optimal solution cannot do as well as the set $[i^*]$ since the average "bang-for-the-buck" in OPT has to be smaller. Thus $p(i^*) + \sum_{i=1}^{i^*-1} p(i) > OPT$. Hence $\max\left(p(i^*), \sum_{i=1}^{i^*-1} p(i)\right) \geq \frac{OPT}{2}$. This algorithm chooses $\max\left(p(i^*), \sum_{i=1}^{i^*-1} p(i)\right)$, so it is a 2-approximation. ■

7.2.3 Pseudo-Polynomial Algorithm for the Knapsack Problem

Let $M = \max_{i \in [n]} p(i)$. Let's define a simply dynamic programming algorithm. The first thing we need is a recurrence. Note that the achievable profit is at least 0 and at most nM. Now we're going to do something which might seem a little weird: we're going to make the subproblems involve not finding the most profit possible, but finding the minimum size necessary to achieve some profit. Formally, for all $i \in [n]$ and $0 \leq v \leq nM$, let $f(i, v)$ denote the minimum size necessary to achieve profit *exactly* v using only elements in $[i]$. Then clearly

$$f(i,v) = \begin{cases} 0 & \text{if } i = 0, v = 0 \\ \infty & \text{if } i = 0, v > 0 \\ f(i-1,v) & \text{if } i > 0, p(i) > v \\ \min\left(f(i-1,v), s(i) + f(i-1, v-p(i))\right) & \text{if } i > 0, p(i) \leq v \end{cases}$$

Note that if we can compute this function, we can solve the knapsack problem by finding the largest v such that $f(n, v) \leq k$ (using extra $O(nM)$ time). Using dynamic programming, we can clearly compute the above function in time $O(n^2 M)$, since there are at most nM options for v, at most n options for i, and evaluating a single table entry takes $O(1)$ time. Because of the factor of M, the runtime of this algorithm is pseudo-polynomial (and hence knapsack is only weakly NP-complete).

7.2.4 Fully Polynomial-time Approximation Scheme for the Knapsack Problem

A fully polynomial-time approximation scheme (FPTAS) is an algorithm which takes an instance of an optimization problem and a parameter $\epsilon > 0$, and, in time polynomial in the instance size and $1/\epsilon$, produces a solution that is within a factor of $1 + \epsilon$ of being optimal. Note that the *fully* in this definition is what requires the running time to be polynomial in $1/\epsilon$ – later we will see PTASes which run in times like $n^{1/\epsilon}$ and hence are not FPTASes. We will now construct an FPTAS for Knapsack.

Let $\delta = \frac{\epsilon M}{n}$. We construct a "rounded" instance by creating a new profit function p' where $p'(i) = \left\lfloor \frac{p(i)}{\delta} \right\rfloor$ for all $i \in [n]$ and $M' = \max_{i \in [n]} p'(i)$. Think of $p'(i)$ as first scaling $p(i)$ so the largest profit is approximately n/ϵ and then rounding down to an integer value.

Note that $p'(i)$ compresses $p(i)$ and also applies rounding (resulting in a loss of precision). Our algorithm simply runs the previous dynamic programming algorithm, substituting p' for p.

Theorem 7.2.3 *This algorithm runs in $O(n^3/\epsilon)$ time.*

Proof: We know from the last subsection that the running time is $O(n^2)$ times the value of the largest profit. Hence it is at most $O(n^2 M') = O\left(n^2 \left\lfloor \frac{M}{\delta} \right\rfloor\right) = O\left(n^2 \left\lfloor \frac{M}{\frac{\epsilon M}{n}} \right\rfloor\right) = O\left(\frac{n^3}{\epsilon}\right)$. ■

Theorem 7.2.4 *This algorithm is a $(1-\epsilon)$-approximation.*

Proof: Let I be the solution returned from this algorithm, and let I^* denote the optimal solution (under the original profits p). First note that I is a feasible solution – since we did not change the sizes, the total size of items in I must be at most k. To bound the total profit, note that from the

definition of p', we know that $p(i) \geq \delta p'(i)$. Hence

$$\sum_{i \in I} p(i) \geq \delta \sum_{i \in I} p'(i) \geq \delta \sum_{i \in I^*} p'(i),$$

since I is the optimal solution under the scaled and rounded p' profits. But now from the definition of p' we have that $p'(i) \geq (p(i)/\delta) - 1$. Thus

$$\begin{aligned} \delta \sum_{i \in I^*} p'(i) &\geq \delta \sum_{i \in I^*} \left(\frac{p(i)}{\delta} - 1 \right) = \sum_{i \in I^*} p(i) - \delta |I^*| \geq OPT - \delta n = OPT - \epsilon M \\ &\geq OPT - \epsilon OPT = (1 - \epsilon) OPT \end{aligned}$$

Putting these together, we get that the profit of the algorithm, $\sum_{i \in I} p(i)$, is at least $(1-\epsilon)OPT$. ■

Theorem 7.2.5 *This algorithm is an FPTAS.*

Proof: By Theorem 7.2.3 and Theorem 7.2.4 this algorithm runs in time polynomial in n and $1/\epsilon$, and produces a solution that is within a factor of $1 - \epsilon$ of being optimal. ■

7.3 Min-Makespan on Identical Parallel Machines

- **Input**: Jobs $J = \{j_1, \ldots, j_n\}$, machines $M = \{m_1, \ldots, m_k\}$, and processing times $p : J \to \mathbb{R}$.
- **Feasible Solutions**: $\Phi : J \to M$
- **Objective**: Minimize makespan $\implies$ Minimize $\max_{m \in M} \left(\sum_{j \in \Phi^{-1}(m)} p(j) \right)$

Algorithm 3 Greedy Min-Makespan Algorithm

Input: J, M, p
Output: $\Phi : J \to M$

```
Φ ← ∅
for j ∈ J do
    m ← least loaded machine in Φ
    Set Φ to assign job j to machine m.
end for
return Φ
```

Theorem 7.3.1 *This algorithm is a 2-approximation.*

Proof: Consider the following two simple observations:

1. $\forall j \in J, p(j) \leq \text{OPT}$ (since every job needs to be scheduled on some machine)
2. Let $I \subseteq J$. Then $\frac{\sum_{j \in I} p(j)}{|I|} \leq \text{OPT}$ (the makespan, i.e. the maximum load, must be at least the average load of any subset of jobs)

Let m be the machine whose load equals the makespan of the greedy algorithm (i.e. the most heavily loaded machine after the algorithm completes). Let j be the last job assigned to m by the algorithm. Then just before j was assigned to m, by the second observation we know that the load of m was at most OPT. By the first observation the processing time for j is at most OPT, and since j was the last job assigned to m we get that the total load on m is at most 2 OPT. ■