Learning Goals

- Different NP-hard problems have different hardness for approximations
- Arbitrarily good approximation algorithms: Fully polynomial-time approximation schemes (FPTAS)
- Dynamic programming in the design of approximation algorithms
- The FPTAS for the Knapsack problem

The Knapsack Problem

- Input: n items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, and a knapsack capacity W. All weights and values are nonnegative integers; $w_i \leq W$ for all i.

The Knapsack Problem

- Input: n items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, and a knapsack capacity W. All weights and values are nonnegative integers; $w_i \leq W$ for all i.
- Decision version: given a value target V, does there exist a subset of items whose total weight is no more than W and whose total value is at least V?

The Knapsack Problem

- Input: n items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, and a knapsack capacity W. All weights and values are nonnegative integers; $w_i \leq W$ for all i.
- Decision version: given a value target V, does there exist a subset of items whose total weight is no more than W and whose total value is at least V?
- Optimization version: output a subset S of items whose total weights do not exceed W and whose total value is maximum
- Formally, $\max_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq W$.

The Knapsack Problem

- Input: n items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, and a knapsack capacity W. All weights and values are nonnegative integers; $w_i \leq W$ for all i.
- Decision version: given a value target V, does there exist a subset of items whose total weight is no more than W and whose total value is at least V?
- Optimization version: output a subset S of items whose total weights do not exceed W and whose total value is maximum
- Formally, $\max_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq W$.
- We already showed the decision version to be NP-complete.

Attempt at Approximation: Greedy

- Greedy approach 1: in each step, among all items that can still be added to the knapsack, choose the one with the maximum value and add it to the knapsack.

Attempt at Approximation: Greedy

- Greedy approach 1: in each step, among all items that can still be added to the knapsack, choose the one with the maximum value and add it to the knapsack.
- This does not guarantee any finite approximation ratio.

Attempt at Approximation: Greedy

- Greedy approach 1: in each step, among all items that can still be added to the knapsack, choose the one with the maximum value and add it to the knapsack.
- This does not guarantee any finite approximation ratio.
- Exercise: Remedy this and get a 2-approximation with a greedy approach (Question 3 in PS5 is a special case)

Attempt at Approximation: Dynamic Programming (DP)

- Recall another fact: when $w_i = v_i$ for all i, the problem is subset sum, and there is a dynamic programming algorithm that exactly solves the problem in time $O(nW)$.

Attempt at Approximation: Dynamic Programming (DP)

- Recall another fact: when $w_i = v_i$ for all i, the problem is subset sum, and there is a dynamic programming algorithm that exactly solves the problem in time $O(nW)$.
- Easy to generalize this DP.

Attempt at Approximation: Dynamic Programming (DP)

- Recall another fact: when $w_i = v_i$ for all i, the problem is subset sum, and there is a dynamic programming algorithm that exactly solves the problem in time $O(nW)$.
- Easy to generalize this DP.
- Approximation idea: maybe we could round things up and run DP.

Attempt at Approximation: Dynamic Programming (DP)

- Recall another fact: when $w_i = v_i$ for all i, the problem is subset sum, and there is a dynamic programming algorithm that exactly solves the problem in time $O(nW)$.
- Easy to generalize this DP.
- Approximation idea: maybe we could round things up and run DP.
- But weights are hard constraints, and rounding them easily lead us to infeasible solutions or bad approximations.

Attempt at Approximation: Dynamic Programming (DP)

- Recall another fact: when $w_i = v_i$ for all i, the problem is subset sum, and there is a dynamic programming algorithm that exactly solves the problem in time $O(nW)$.
- Easy to generalize this DP.
- Approximation idea: maybe we could round things up and run DP.
- But weights are hard constraints, and rounding them easily lead us to infeasible solutions or bad approximations.
- Rounding values avoids these problems, but our DP's running time doesn't benefit from such rounding.

Attempt at Approximation: Dynamic Programming (DP)

- Recall another fact: when $w_i = v_i$ for all i, the problem is subset sum, and there is a dynamic programming algorithm that exactly solves the problem in time $O(nW)$.
- Easy to generalize this DP.
- Approximation idea: maybe we could round things up and run DP.
- But weights are hard constraints, and rounding them easily lead us to infeasible solutions or bad approximations.
- Rounding values avoids these problems, but our DP's running time doesn't benefit from such rounding.
- We need a new DP!

Another Dynamic Programming

Idea: Iteratively compute an array where $A[v]$ is the minimum weight needed to achieve value v.

Another Dynamic Programming

Idea: Iteratively compute an array where $A[v]$ is the minimum weight needed to achieve value v.

The algorithm:

- Initialize $A[v] \leftarrow \infty$ for $v = 1, 2, \ldots, v^* n$ where $v^* := \max_i v_i$.
  Initialize $A[0] \leftarrow 0$.

Another Dynamic Programming

Idea: Iteratively compute an array where $A[v]$ is the minimum weight needed to achieve value v.

The algorithm:

- Initialize $A[v] \leftarrow \infty$ for $v = 1, 2, \ldots, v^* n$ where $v^* := \max_i v_i$. Initialize $A[0] \leftarrow 0$.
- For each item $i = 1, 2, \ldots, n$: iterate v from $(n-1)v^*$ down to 0, and update $A[v + v_i] \leftarrow \min(A[v + v_i], A[v] + w_i)$.

Another Dynamic Programming

Idea: Iteratively compute an array where $A[v]$ is the minimum weight needed to achieve value v.

The algorithm:

- Initialize $A[v] \leftarrow \infty$ for $v = 1, 2, \ldots, v^* n$ where $v^* := \max_i v_i$. Initialize $A[0] \leftarrow 0$.
- For each item $i = 1, 2, \ldots, n$: iterate v from $(n-1)v^*$ down to 0, and update $A[v + v_i] \leftarrow \min(A[v + v_i], A[v] + w_i)$.
- Return the largest v such that $A[v] \leq W$.

Another Dynamic Programming

Idea: Iteratively compute an array where $A[v]$ is the minimum weight needed to achieve value v.

The algorithm:

- Initialize $A[v] \leftarrow \infty$ for $v = 1, 2, \ldots, v^* n$ where $v^* := \max_i v_i$. Initialize $A[0] \leftarrow 0$.
- For each item $i = 1, 2, \ldots, n$: iterate v from $(n-1)v^*$ down to 0, and update $A[v + v_i] \leftarrow \min(A[v + v_i], A[v] + w_i)$.
- Return the largest v such that $A[v] \leq W$.

Running time: for each item i, we go through the array which has length nv^*, so total running time $O(v^* n^2)$.

Rounding the values for approximation

- The dynamic programming is pseudopolynomial time because of its running time dependence on v^*.

Rounding the values for approximation

- The dynamic programming is pseudopolynomial time because of its running time dependence on v^*.
- Intuitively, if we round up the values to multiples of a small integer b and run the dynamic programming, we shouldn't be far off.

Rounding the values for approximation

- The dynamic programming is pseudopolynomial time because of its running time dependence on v^*.
- Intuitively, if we round up the values to multiples of a small integer b and run the dynamic programming, we shouldn't be far off.
- Let $\hat{v}_i := \lceil v_i / b \rceil$, and $\tilde{v}_i := \hat{v}_i b$.

Rounding the values for approximation

- The dynamic programming is pseudopolynomial time because of its running time dependence on v^*.
- Intuitively, if we round up the values to multiples of a small integer b and run the dynamic programming, we shouldn't be far off.
- Let $\hat{v}_i := \lceil v_i/b \rceil$, and $\tilde{v}_i := \hat{v}_i b$.
- Run the dynamic programming on $\hat{v}_1, \ldots, \hat{v}_n$, then the running time would be $O(n^2 v^*/b)$.

Rounding the values for approximation

- The dynamic programming is pseudopolynomial time because of its running time dependence on v^*.
- Intuitively, if we round up the values to multiples of a small integer b and run the dynamic programming, we shouldn't be far off.
- Let $\hat{v}_i := \lceil v_i/b \rceil$, and $\tilde{v}_i := \hat{v}_i b$.
- Run the dynamic programming on $\hat{v}_1, \ldots, \hat{v}_n$, then the running time would be $O(n^2 v^*/b)$.
- How good an approximation is S, the set of items chosen by the algorithm?

Rounding the values for approximation

- The dynamic programming is pseudopolynomial time because of its running time dependence on v^*.
- Intuitively, if we round up the values to multiples of a small integer b and run the dynamic programming, we shouldn't be far off.
- Let $\hat{v}_i := \lceil v_i/b \rceil$, and $\tilde{v}_i := \hat{v}_i b$.
- Run the dynamic programming on $\hat{v}_1, \ldots, \hat{v}_n$, then the running time would be $O(n^2 v^*/b)$.
- How good an approximation is S, the set of items chosen by the algorithm?
- Let S^* be any other feasible set of items, then

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \tilde{v}_i = b \sum_{i \in S^*} \hat{v}_i \leq b \sum_{i \in S} \hat{v}_i = \sum_{i \in S} \tilde{v}_i \leq nb + \sum_{i \in S} v_i.$$

Final step

$$\sum_{i \in S^*} v_i \le \sum_{i \in S^*} \tilde{v}_i = b \sum_{i \in S^*} \hat{v}_i \le b \sum_{i \in S} \hat{v}_i = \sum_{i \in S} \tilde{v}_i \le nb + \sum_{i \in S} v_i .$$

- If we could make nb a small fraction of $\sum_{i \in S} v_i$, say, at most ϵ fraction, then RHS is $(1+\epsilon)\sum_{i \in S} v_i$.

Final step

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \tilde{v}_i = b \sum_{i \in S^*} \hat{v}_i \leq b \sum_{i \in S} \hat{v}_i = \sum_{i \in S} \tilde{v}_i \leq nb + \sum_{i \in S} v_i .$$

- If we could make nb a small fraction of $\sum_{i \in S} v_i$, say, at most ϵ fraction, then RHS is $(1+\epsilon) \sum_{i \in S} v_i$.
- How big should be b?

Final step

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \tilde{v}_i = b \sum_{i \in S^*} \hat{v}_i \leq b \sum_{i \in S} \hat{v}_i = \sum_{i \in S} \tilde{v}_i \leq nb + \sum_{i \in S} v_i.$$

- If we could make nb a small fraction of $\sum_{i \in S} v_i$, say, at most ϵ fraction, then RHS is $(1+\epsilon)\sum_{i \in S} v_i$.
- How big should be b?
- We want $nb \leq \epsilon \sum_{i \in S} v_i$. Fixing ϵ, this asks for lower bounding $\sum_{i \in S} v_i$.
  - Let S^* be the singleton set containing the item with the largest value, the above inequality gives $\sum_{i \in S} v_i \geq v^* - nb$.

Final step

$$\sum_{i\in S^*} v_i \leq \sum_{i\in S^*} \tilde{v}_i = b\sum_{i\in S^*} \hat{v}_i \leq b\sum_{i\in S} \hat{v}_i = \sum_{i\in S} \tilde{v}_i \leq nb + \sum_{i\in S} v_i.$$

- If we could make nb a small fraction of $\sum_{i\in S} v_i$, say, at most ϵ fraction, then RHS is $(1+\epsilon)\sum_{i\in S} v_i$.
- How big should be b?
- We want $nb \leq \epsilon \sum_{i\in S} v_i$. Fixing ϵ, this asks for lower bounding $\sum_{i\in S} v_i$.
    - Let S^* be the singleton set containing the item with the largest value, the above inequality gives $\sum_{i\in S} v_i \geq v^* - nb$.
    - As long as $b \leq \epsilon v^*/(2n)$, we have $nb \leq \epsilon v^* - nb \leq \epsilon(v^* - nb) \leq \epsilon \sum_{i\in S} v_i$ for $\epsilon < 1$.

Final step

$$\sum_{i \in S^*} v_i \le \sum_{i \in S^*} \tilde{v}_i = b \sum_{i \in S^*} \hat{v}_i \le b \sum_{i \in S} \hat{v}_i = \sum_{i \in S} \tilde{v}_i \le nb + \sum_{i \in S} v_i.$$

- If we could make nb a small fraction of $\sum_{i \in S} v_i$, say, at most ϵ fraction, then RHS is $(1+\epsilon)\sum_{i \in S} v_i$.
- How big should be b?
- We want $nb \le \epsilon \sum_{i \in S} v_i$. Fixing ϵ, this asks for lower bounding $\sum_{i \in S} v_i$.
    - Let S^* be the singleton set containing the item with the largest value, the above inequality gives $\sum_{i \in S} v_i \ge v^* - nb$.
    - As long as $b \le \epsilon v^*/(2n)$, we have $nb \le \epsilon v^* - nb \le \epsilon(v^* - nb) \le \epsilon \sum_{i \in S} v_i$ for $\epsilon < 1$.
- Running time: $O(n^2 v^*/b) = O(n^3 \epsilon^{-1})$.

PTAS and FPTAS

Theorem

For any $\epsilon > 0$, the Knapsack problem can be approximated to a factor of $1 + \epsilon$ by an algorithm that runs in time $O(n^3 \epsilon^{-1})$.

PTAS and FPTAS

Theorem

For any $\epsilon > 0$, the Knapsack problem can be approximated to a factor of $1+\epsilon$ by an algorithm that runs in time $O(n^3\epsilon^{-1})$.

Definition

A family of approximation algorithms is a *polynomial-time approximation scheme* (PTAS) for an optimization problem if for any $\epsilon > 0$, there is an algorithm in the family that is a $(1+\epsilon)$-approximation algorithm for the problem, with polynomial running time when ϵ is treated as a constant. If the running time depends polynomially on ϵ^{-1}, the family is said to be a *fully polynomial-time approximation scheme* (FPTAS).

PTAS and FPTAS

Theorem

For any $\epsilon > 0$, the Knapsack problem can be approximated to a factor of $1 + \epsilon$ by an algorithm that runs in time $O(n^3 \epsilon^{-1})$.

Definition

A family of approximation algorithms is a *polynomial-time approximation scheme* (PTAS) for an optimization problem if for any $\epsilon > 0$, there is an algorithm in the family that is a $(1 + \epsilon)$-approximation algorithm for the problem, with polynomial running time when ϵ is treated as a constant. If the running time depends polynomially on ϵ^{-1}, the family is said to be a *fully polynomial-time approximation scheme* (FPTAS).

We have obtained an FPTAS for the Knapsack problem.