2025 Fall CSE5025

Combinatorial Optimization

组合优化

Instructor: 刘晟材

Lecture 3-3: Packing Problems and Monotone Submodular Maximization

Agenda for Today's Lecture

In this lecture, we will focus on

- Knapsack Problem, Bin-packing Problem, and their approximation algorithms
- Monotone Submodular Maximization

Learning Objectives for this lecture

- Know packing problems and classic approximation/heuristic algorithms for them
- Master the concept of monotonicity and submodularity
- Master the greedy algorithm for monotone submodular maximization problem

What is an Approximation Algorithm?

An **approximation algorithm** is an algorithm that finds an approximate (i.e., not necessarily optimal) solution to an NP-hard optimization problem.

Key Properties:

- It must run in polynomial time.
- It must output a feasible solution
- There must be a provable guarantee on the quality of the solution

The Approximation Ratio (ρ)

This guarantee is quantified by the approximation ratio (or factor), denoted by ρ (rho).

For a minimization problem:

- An algorithm is a ρ-approximation if: Solution's Cost $\leq \rho \cdot$ Optimal Cost

For a maximization problem:

- An algorithm is a ρ-approximation if: Optimal Value $\leq \rho \cdot$ Solution's Value

In the above definition, the ratio ρ is always ≥ 1.

A 1-approximation algorithm is an exact algorithm.

Alternative Definition of Approximation Ratio (ρ_1)

For a minimization problem:

- An algorithm is a ρ_1-approximation ($\rho_1 \geq 1$) if:

$$\frac{\text{Solution's Cost}}{\text{Optimal Cost}} \leq \rho_1$$

For a maximization problem:

- An algorithm is a ρ_1-approximation ($\rho_1 \leq 1$) if:

$$\frac{\text{Solution's Value}}{\text{Optimal Value}} \geq \rho_1$$

A 1-approximation algorithm is an exact algorithm.

This definition can be transformed to the previous definition with ease, and vice versa.

Packing Problems

In our previous lectures (Vertex Cover, Set Cover, TSP), we focused on **selection and routing**. Today, we look at **Packing Problems**.

General Idea:

- You have a set of items, each with a “size” or “weight”.
- You have one or more “containers” (bins, knapsacks) with a fixed “capacity”.
- Goal: Pack the items into the containers to optimize some objective.

Two canonical examples:

- The Knapsack Problem: One container. Maximize the total value of items packed without exceeding capacity.
- The Bin Packing Problem: Many identical containers. Minimize the number of containers used to pack all items.

The 0/1 Knapsack Problem (KP)

Given

- A set of n items, $N = \{1, 2, \ldots, n\}$.
- Each item i has a weight $w_i > 0$ and a value $v_i > 0$.
- A knapsack with a total weight capacity W

Goal

- Select a subset of items $S \subseteq N$ that maximizes the total value, subject to the total weight not exceeding the capacity

The 0/1 KP: Problem Formulation

For each item $i = 1, \ldots, n$, we define a **binary decision variable** $x_i \in \{0,1\}$:

$$x_i = \begin{cases} 1 & \text{if } i\text{tem } i \text{ is selected,} \\ 0 & \text{if } i\text{tem } i \text{ is not selected} \end{cases}$$

Problem Formulation:

$$\text{Maximize} \quad \sum_{i=1}^{n} v_i x_i$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq W$$

$$x_i \in \{0,1\} \quad \text{for} \quad i \in \{0,1 \ldots, n\}$$

Variants of KP

Bounded Knapsack (有界背包): Each item i has b_i copies available.

- maximize $\sum_i v_i x_i$ s.t. $\sum_i w_i x_i \leq W$, $0 \leq x_i \leq b_i$, $x_i \in \mathbb{Z}$

Unbounded Knapsack (无界背包) : An unlimited supply of each item.

- maximize $\sum_i v_i x_i$ s.t. $\sum_i w_i x_i \leq W$, $x_i \in \{0,1,2,\dots\}$

Fractional Knapsack (分数背包): Can take fractions of items.

- maximize $\sum_i v_i x_i$ s.t. $\sum_i w_i x_i \leq W$, $x_i \in [0,1]$
- This is solvable in polynomial time by a simple greedy algorithm (sort by v_i/w_i).
- **This problem belongs to P class**

Complexity of 0/1 KP

The 0/1 KP is **NP-hard**.

We can show this by a Cook reduction from the **Partition Problem (**Known NP-Complete**)**.

Partition Problem: Given a set of positive integers $A = \{a_1, \dots, a_n\}$, can we partition A into two subsets S_1 and S_2 such that the sum of elements in S_1 equals the sum of elements in S_2?

Note the partition problem is a **decision** problem.

Proving 0/1 KP is NP-Hard (1)

Claim: If we have a oracle solver for the 0/1 KP, then we can solve the partition problem in polynomial time.

Proof:

1) To solve a partition problem instance $A = \{a_1, \dots, a_n\}$, calculate $\text{sum} = a_1 + a_2 + \cdots + a_n$. Then check the sum:

- If sum is odd, return "No" (a partition is impossible)
- Otherwise, define target sum $K = \text{sum}/2$

2) Create a special 0/1 KP instance with n items

- for each item i, set its weight $w_i = a_i$, and set its value $v_i = a_i$
- set knapsack capacity $W = K$

Proving 0/1 KP is NP-Hard (2)

3) Call the oracle on the special KP instance. The oracle returns $V_{\max}$, the maximum possible value (sum) that can be achieved with a total weight less than or equal to K. Then Make the decision:

- If $V_{\max} == K$, return "Yes"; If $V_{\max} < K$, return "No"

4) The above **polynomial-time** algorithm works because

- Case 1 (the true answer to the partition instance is Yes): there is a subset $A' \subseteq A$ that sums to K. It is exactly a valid solution to the KP instance. Then the optimal value $V_{\max} \geq K$. Since the capacity $W = K$ and $(w_i = a_i)$, $V_{\max} \leq K$. Hance $V_{\max} = K$.
- Case 2 (the true answer to the partition instance is No): there is no subset $A' \subseteq A$ that sums to K. This means every subset that fits within the weight limit $W = K$ must have a **sum strictly** less than K.Since $w_i = a_i$, $V_{\max} < K$.

The Greedy Algorithm for 0/1 KP

Algorithm Steps:

1) For each item i, calculate its value density: $d_i = v_i/w_i$.

2) Sort all items in descending order of their density.

3) Iterate through the sorted items. For each item, if it fits into the remaining capacity of the knapsack, add it.

4) Stop when no more items can be added.

This algorithm has no **constant-factor approximation ratio** for the 0/1 KP. Why?

PTAS (Polynomial-Time Approximation Scheme)

A PTAS (多项式时间近似方案) is an algorithm A that takes an input instance I and a parameter $\epsilon > 0$. It produces a solution S with an approximation ratio of at most $1 + \epsilon$:

- Minimization: Solution's Cost $\leq (1 + \epsilon) \cdot$ Optimal Cost
- Maximization: Optimal Value $\leq (1 + \epsilon) \cdot$ Solution's Value

The runtime of A must be **polynomial in the input size m for any fixed ϵ.**

Example: $O(m^{1/\epsilon})$ or $O(m^{\exp(1/\epsilon)})$.

FPTAS (Fully PTAS)

An FPTAS (完全多项式时间近似方案) is a PTAS whose runtime is **polynomial** in both the input size $\boldsymbol{m}$ **AND** $\boldsymbol{1/\epsilon}$.

Example: $O(m^3/\epsilon)$ or $O(m^2(1+1/\epsilon))$.

FPTAS is **stronger than** PTAS in approximating NP-hard problems.

Note that only **a few NP-hard problems have FPTAS.**

Pseudo-polynomial time algorithm for 0/1 KP

The 0/1 KP is NP-hard, but it's **not strongly** NP-hard

It admits a **pseudo-polynomial** time algorithm.

An algorithm is **pseudo-polynomial** if its running time is polynomial in the **numeric value of the input** (e.g., W, $\sum v_i$), but **exponential in the length** (number of bits) of the input (e.g., n, $\log W$).

Core Idea of Dynamic Programming (DP): Break down a complex problem into a collection of simpler, overlapping subproblems. Solve each subproblem only once and store its solution.

Dynamic Programming for 0/1 KP (1)

DP Approach 1: Let's consider the optimal solution for such a 0/1 KP problem instance: using only items from $\{1, \ldots, n'\}$ with a total weight capacity of W'

- **Case 1: Item n' is not in the optimal solution**. Then the optimal solution must **be the same as** the optimal solution for a smaller 0/1 KP instance: using only items from $\{1, \ldots, n' - 1\}$ with a total weight capacity of W'.
- **Case 2: Item n' is in the optimal solution**. Then the optimal solution's value is $v_{n'}$ (the value of item n') plus the value of the optimal solution for a smaller 0/1 KP instance: using only items from $\{1, \ldots, n' - 1\}$ with a total weight capacity of $W' - w_{n'}$.

Dynamic Programming for 0/1 KP (2)

DP Approach 1 (runtime based on weight):

- Let $DP(n', W')$ be the maximum value (optimal value) we can get using only items from $\{1, \dots, n'\}$ with a total weight capacity of W'. We have $n' \leq n$ and $W' \leq W$
- **Recurrence**: $DP(n', W')$ equals to the larger value between the following two subproblems' optimal values:
  1) $\mathrm{DP}(n' - 1, W')$: If we don't take item n'
  2) $v_{n'} + DP(n' - 1, W' - w_i)$: If we take item n' and $W' \geq w_{n'}$
- **Base Case**: $DP(0, 0) = 0$, $DP(0, w) = 0$ for any $w > 0$.
- **Final Answer**: $DP(n, W)$
- Runtime: The DP table size is $n \cdot W$. Each entry in the table takes $O(1)$ to compute.
- **Total Time**: $O(nW)$, it's **pseudo-polynomial** since depending on the numerical value

Dynamic Programming for 0/1 KP (3)

Instead of finding the maximum value for a given capacity, we find the **minimum total weight required to achieve a specific value**.

DP Approach 2: Let's consider the optimal solution (minimum weight) using only items from $\{1, \dots, n'\}$ to achieve a total value of V'

- **Case 1: Item n' is not in the optimal solution**. Then the optimal solution must be the same as the optimal solution for a smaller instance: using only items from $\{1, \dots, n' - 1\}$ to achieve a total value of V'
- **Case 2: Item n' is in the optimal solution**. Then the optimal solution's weight is $w_{n'}$ (the weight of item n') plus the weight of the optimal solution for a smaller instance: using only items from $\{1, \dots, n' - 1\}$ to achieve a total value of $V' - v_{n'}$

Dynamic Programming for 0/1 KP (4)

DP Approach 2 (based on value):

- Let $DP(n', V')$ be the minimum weight required to achieve an exact total value of V', using only items from $\{1, \dots, n'\}$. If V' is impossible to achieve, set $DP(n', V') = \infty$.
- **Recurrence**: $DP(n', W')$ equals to the smaller value between the following two subproblems' optimal values:

  1) $\mathrm{DP}(n' - 1, V')$. This means we don't take item n'

  2) $w_{n'} + DP(n' - 1, V' - v_{n'})$ if $V' \geq v_{n'}$. This means we take item n'
- **Base Case**: $DP(0, 0) = 0$, $DP(0, V) = \infty$ for any $V > 0$.
- **Final Answer**: The largest V such that $DP(n, V) \leq W$
- Runtime: The DP table size is $n \cdot \sum v_i \leq n \cdot v_{max}$. Each entry in the table takes $O(1)$ to compute.
- **Total Time**: $O(n^2 \cdot v_{max})$, it's **pseudo-polynomial** since depending on numerical value

Towards an FPTAS

Recall our second DP: $O(n^2 \cdot v_{max})$ runtime, where $v_{max} = \max\{v_i | \mathrm{w}_i \leq W\}$

This DP is pseudo-polynomial because v_{max} can be large.

Key Idea: What if we "scale down" the values to make them small?

- This will make the DP run faster.
- But, scaling will introduce errors in our solution.

Goal: Can we control the error?

FPTAS: Scaling and Rounding

Let $\epsilon > 0$ be our desired approximation factor (e.g., $\epsilon = 0.01$). We want a $(1 + \epsilon)$-approximation.

Algorithmic Steps:

1) Set a scaling factor K. (We will define K based on ϵ later).

2) Create new, "scaled" values for each item: $v_i' = \left\lfloor \frac{v_i}{K} \right\rfloor$

3) Solve the new 0/1 KP instance using these values v_i' and original weights.

- Use the DP with runtime $O(n^2 v'_{max})$, where $v'_{max} = \max\{v_i' | \mathrm{w}_i \leq W\}$
- Let S_{ALG} be the optimal solution for this scaled problem instance

4) Return the better between S_{ALG} and the trivial solution (only containing v_{max}) as the approximate solution for the original KP instance.

The Challenge

Let ALG be the value of S_{ALG} on the original 0/1 KP instance.

We need to choose K such that

- **Runtime**: The DP is fast, polynomial in n and $1/\epsilon$
- **Accuracy**: The solution S_{ALG} is a $(1+\epsilon)$-approximation solution to the original 0/1 KP instance. That is, $OPT \leq (1+\epsilon) \cdot ALG$ or $\frac{OPT}{ALG} \leq (1+\epsilon)$

Let L be a lower bound on OPT, i.e., $OPT \geq L$. A good lower bound is v_{max}.

The scaling factor K is set as:

$$K = \left(\frac{\epsilon}{1+\epsilon}\right) \cdot \frac{L}{n} = \epsilon' \frac{v_{max}}{n}$$

Analysis of Runtime

Our DP has runtime $O(n^2 v'_{max})$. And the following holds:

$$v'_{max} \leq \frac{v_{max}}{K} \leq \frac{v_{max}}{\epsilon' \frac{v_{max}}{n}} = \frac{n}{\epsilon'} = \frac{n}{\frac{\epsilon}{1+\epsilon}} = n(1 + 1/\epsilon)$$

Hence, the runtime is $O(n^3(1 + 1/\epsilon))$, which is polynomial in both n and $1/\epsilon$.

Proof of Approximation Ratio (1)

FPTAS: Approximation Ratio Proof

- S_{OPT}: The set of items in the optimal solution. $OPT = \sum_{i \in S_{OPT}} v_i$.
- S_{ALG}: The solution (set of items) returned by our DP. $ALG = \sum_{i \in S_{ALG}} v_i$. *(We use ALG to denote $VAL(S_{ALG})$, and we will return $\max(ALG, L)$).*
- $VAL'(S)$: The *scaled* value. $VAL'(S) = \sum_{i \in S} v'_i$.
- L: Our lower bound, $L = v_{max}$. We know $L \leq OPT$.
- ϵ': Our *internal* parameter, $\epsilon' = \frac{\epsilon}{1+\epsilon}$.
- K: Our scaling factor, $K = \frac{\epsilon' \cdot L}{n}$.

Goal: Prove that $\frac{OPT}{max(ALG,L)} \leq (1+\epsilon)$.

Proof of Approximation Ratio (2)

Ratio Proof (Part 1: Key Inequalities)

From the definition $v'_i = \lfloor v_i/K \rfloor$, we know for any item i:

1. $K \cdot v'_i \leq v_i$ (from $v'_i \leq v_i/K$)
2. $v_i < K \cdot v'_i + K$ (from $v_i/K - 1 < v'_i$)

Let's analyze OPT (the value of the optimal solution):

$$OPT = \sum_{i \in S_{OPT}} v_i$$

Using inequality (2):

$$OPT < \sum_{i \in S_{OPT}} (Kv'_i + K) = K \left(\sum_{i \in S_{OPT}} v'_i \right) + |S_{OPT}| \cdot K$$

Since $|S_{OPT}| \leq n$:

$$OPT < K \cdot VAL'(S_{OPT}) + nK$$

Proof of Approximation Ratio (3)

Let's analyze ALG (the value of our algorithm's solution S_{ALG}):

$$ALG = \sum_{i \in S_{ALG}} v_i$$

Using inequality (1):

$$ALG \geq \sum_{i \in S_{ALG}} (Kv_i') = K \cdot VAL'(S_{ALG})$$

Proof of Approximation Ratio (4)

Ratio Proof (Part 2: Combining)

We have three key facts: (A) $OPT < K \cdot VAL'(S_{OPT}) + nK$ (B) $ALG \geq K \cdot VAL'(S_{ALG})$ (C) $VAL'(S_{ALG}) \geq VAL'(S_{OPT})$ *(This is true because S_{ALG} is the **optimal solution** for the scaled problem v').*

Now, let's chain these facts together:

$$OPT \overset{(A)}{<} K \cdot VAL'(S_{OPT}) + nK$$

Apply (C):

$$OPT \overset{(C)}{\leq} K \cdot VAL'(S_{ALG}) + nK$$

From (B), we know $K \cdot VAL'(S_{ALG}) \leq ALG$. Substitute this:

$$OPT \overset{(B)}{\leq} ALG + nK$$

This is the most important intermediate result:

$$OPT < ALG + nK$$

Proof of Approximation Ratio (5)

Ratio Proof (Part 3: The Conclusion)

We have shown:

$$OPT < ALG + nK$$

Now, substitute our definition of $K = \frac{\epsilon' \cdot L}{n}$:

$$OPT < ALG + n\left(\frac{\epsilon' \cdot L}{n}\right)$$

$$OPT < ALG + \epsilon' \cdot L$$

We know $L \leq OPT$ (our lower bound is less than or equal to the optimal).

$$OPT < ALG + \epsilon' \cdot OPT$$

Now, rearrange the terms:

$$OPT - \epsilon' \cdot OPT < ALG$$

$$OPT(1 - \epsilon') \leq ALG$$

Proof of Approximation Ratio (6)

This proves the quality of the DP solution ALG. Our final returned value is $max(ALG, L)$.
Since $L \leq OPT$, $OPT(1-\epsilon')$ is the stronger bound.

$$OPT(1-\epsilon') \leq \max(ALG, L)$$

Finally, substitute our *internal* parameter $\epsilon' = \frac{\epsilon}{1+\epsilon}$:

$$OPT\left(1-\frac{\epsilon}{1+\epsilon}\right) \leq \max(ALG, L)$$

$$OPT\left(\frac{1}{1+\epsilon}\right) \leq \max(ALG, L)$$

Multiplying by $(1+\epsilon)$ gives our goal:

$$OPT \leq (1+\epsilon) \cdot \max(ALG, L)$$

FPTAS for 0/1 KP: Summary

Strategy: We "shrink" the problem by scaling and rounding values $v_i' = \lfloor v_i / K \rfloor$.

Goal: To achieve a $(1+\epsilon)$ approximation ratio

Key Insight: We must choose our scaling factor K based on an *internal* parameter $\epsilon' = \frac{\epsilon}{1+\epsilon}$ and a lower bound $L = v_{max}$ (where $v_{max} = \max\{v_i \mid w_i \leq W\}$).

$$K = \left(\frac{\epsilon}{1+\epsilon}\right) \cdot \frac{v_{max}}{n}$$

Algorithm: We run the exact DP on the scaled v_i' values and return the max of that solution's original value and the lower bound L.

Time: The runtime is $O(n^3(1+1/\epsilon))$, which is **polynomial in n and $1/\epsilon$**.

Ratio: The proof $OPT < ALG + nK$ combined with our specific choice of K yields $OPT \leq (1+\epsilon)ALG_{final}$.

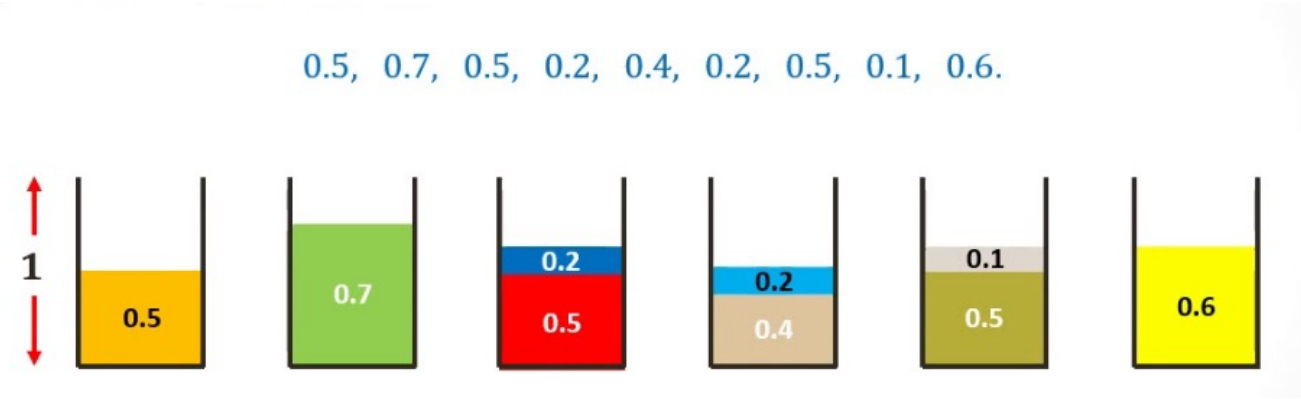
The Bin Packing Problem (BP)

Given

- A set of n items, $N = \{1, 2, \ldots, n\}$.
- Each item i has a size $s_i \in (0, 1]$
- An infinite supply of bins, each with capacity 1

Goal

- Pack all n items into the **minimum** number of bins.
- The sum of sizes of items in any single bin (the load) must not exceed the capacity 1



Complexity of BP

The BP is **NP-hard**.

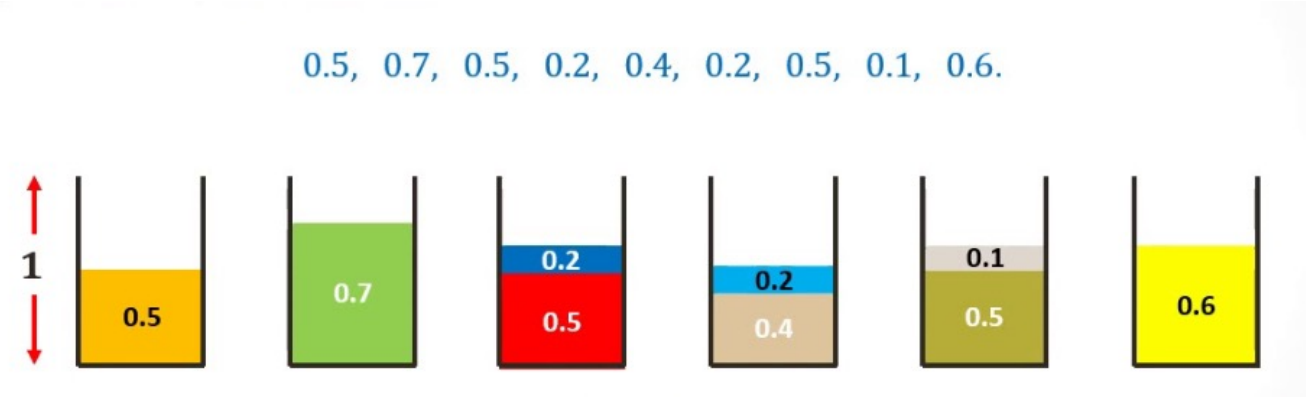
We can also show this by a reduction from the **Partition Problem (**Known NP-Complete**)**.

The BP is actually harder than the 0/1 KP, since it cannot have an FPTAS.

Next Fit (NF) Heuristic for BP

Heuristic 1: Next Fit (NF)

- Process items in arbitrary order $i = 1, \ldots, n$.
- Maintain one "open" bin.
- When item i arrives:
  - If i fits in the open bin, place it there.
  - If i does not fit, close the current bin and open a new bin. Place i in the new bin.



Analysis of Next Fit (NF)

The runtime of NF is $O(n)$.

Theorem: NF is a 2-approximation algorithm for BP. That is, for any BP instance, $m \leq 2 \cdot OPT$, where m is the number of bins used by NN and OPT is the optimal value.

Proof of Approximation ratio (1)

Consider any two *consecutive* bins, Bin j and Bin $j+1$.

Let S_j be the total size of items in Bin j.

When we opened Bin $j+1$, it was because an item *failed* to fit in Bin j.

This means $S_j + S_{j+1} > 1$. (If $S_j + S_{j+1} \leq 1$, the first item of B_{j+1} would have fit in B_j, a contradiction).

This holds for all pairs $(B_1, B_2), (B_3, B_4), ..., (B_{m-1}, B_m)$.

We have $\lfloor m/2 \rfloor$ such disjoint pairs.

Proof of Approximation ratio (2)

The total size of all items, $S = \sum s_i$, is:

$$S = \sum_{j=1}^{m} S_j = (S_1 + S_2) + (S_3 + S_4) + \ldots$$

Each pair $(S_j + S_{j+1}) > 1$.

So, $S > \lfloor m/2 \rfloor$.

We know $OPT \geq S$ (a simple lower bound).

Therefore, $OPT > \lfloor m/2 \rfloor$.

Proof of Approximation ratio (3)

If m is even, $m = 2k$. $OPT > \lfloor 2k/2 \rfloor = k = m/2$. So $2 \cdot OPT > m$.

If m is odd, $m = 2k + 1$. $OPT > \lfloor (2k + 1)/2 \rfloor = k = (m - 1)/2$. So $2 \cdot OPT > m - 1$.

This gives $m \leq 2 \cdot OPT$. (A more careful proof shows $m \leq 2 \cdot OPT - 1$).

First Fit (FF) Heuristic for BP

Heuristic 2: First Fit (FF)

- Process items in arbitrary order $i = 1, \ldots, n$.
- Keep **all** previous bins open.
- When item i arrives:
  - Try to place it in Bin 1.
  - If it doesn't fit, try Bin 2.
  - …
  - If it fits in no existing bin, open a new bin and place it there.

Analysis of First Fit (FF)

The runtime of FF is $O(n^2)$.

Theorem: For any BP instance, $m \leq 1.7 \cdot OPT + 2$, where m is the number of bins used by FF and OPT is the optimal value.

The proof is much more complex than for NF. It relies on a **Key Lemma**: At any point in the algorithm, there can be **at most one bin** whose load (total size of all items in the bin) is half-full or less (i.e., ≤ 0.5).

Why this matters: This lemma implies that if FF uses m bins, at least $m - 1$ of them must be > 0.5 full. This property is the foundation for proving its approximation ratio.

Proof Sketch of the Lemma (1)

1. **Assume** the lemma is false. This means at some point, there are *at least two* bins that are ≤ 0.5 full.
2. Let's pick the first two such bins, B_j and B_k, and assume $j < k$ (meaning B_j was opened before B_k).
3. Now, think about the **very first item** x that was placed into the *later* bin, B_k.
4. Why was x placed in B_k? By the **First Fit** rule, it must be because it did *not* fit in *any* of the bins before it, including B_j.
5. This means: $\text{size}(x) + \text{current\_load}(B_j) > 1$.
6. However, we know two things:
   - Since x is the first item in B_k (which is ≤ 0.5 full), $\text{size}(x)$ must be ≤ 0.5.
   - Since B_j *also* ends up ≤ 0.5 full, its $\text{current\_load}(B_j)$ must be ≤ 0.5.

Proof Sketch of the Lemma (2)

7. If both $\text{size}(x) \leq 0.5$ and $\text{current\_load}(B_j) \leq 0.5$, their sum *cannot* be greater than 1.
   - $\text{size}(x) + \text{current\_load}(B_j) \leq 0.5 + 0.5 = 1.0$
8. **This is a contradiction!** The item x *should* have fit into B_j. The FF rule would have put it there, and B_k would not have been used for this item.
9. Therefore, our initial assumption (that two such bins exist) is impossible.

Best Fit (BF) Heuristic for BP

Heuristic 3: Best Fit (FF)

- Process items in arbitrary order $i = 1, \ldots, n$.
- Keep **all** previous bins open.
- When item i arrives:
  - Place the item in the **fullest** bin that can still accommodate it.
  - If it fits in no existing bin, open a new bin and place it there.

Analysis of Best Fit (BF)

The runtime of BF is $O(n^2)$.

Theorem: For any BP instance, $m \leq 1.7 \cdot OPT + 2$, where m is the number of bins used by BF and OPT is the optimal value.

The "Decreasing" Trick can further improve the ratio to $m \leq \frac{11}{9} \cdot OPT + 4$

1) Sort items from largest to smallest.

2) Run FF on the sorted list

Check https://en.wikipedia.org/wiki/Bin_packing_problem for complete proof of the approximation ratios of FF and BF (not required for this course).

A More General View of CO

We have seen:

- Packing (KP, BP): Max value / Min bins.
- Covering (Vertex Cover, Set Cover): Min cost to cover all elements.
- Routing (TSP, CVRP): Min cost to visit all nodes.

Many of these problems involve selecting a subset S from a ground set N to maximize $f(S)$ subject to some constraints, and we are often interested in functions that satisfy two key properties: **monotonicity (单调性) and submodularity (子模性).**

The Problem Definition

We are given a **ground set** of n items, $N = \{1, ..., n\}$.

We have a **set function** $f : 2^N \to \mathbb{R}_{\geq 0}$.

- This function assigns a non-negative value $f(S)$ to every possible subset $S \subseteq N$.
- We assume $f(\emptyset) = 0$.

The Problem: We want to find a subset S of at most k items that maximizes this function:

$$\max_{S \subseteq N} f(S)$$

$$\text{subject to} \quad |S| \leq k$$

Monotonicity (单调性)

Monotonicity (Monotone)

- **Intuition:** Adding more items *never* decreases the total value.
- **Formal Definition:** For any $A \subseteq B \subseteq N$, we have:

$$f(A) \leq f(B)$$

Submodularity (子模性，收益边际递减)

Intuition: The marginal benefit of adding a new item decreases (or stays the same) as the set it is being added to grows.

Formal Definition: For any $A \subseteq B \subseteq N$ and any item $v \in N \setminus B$:

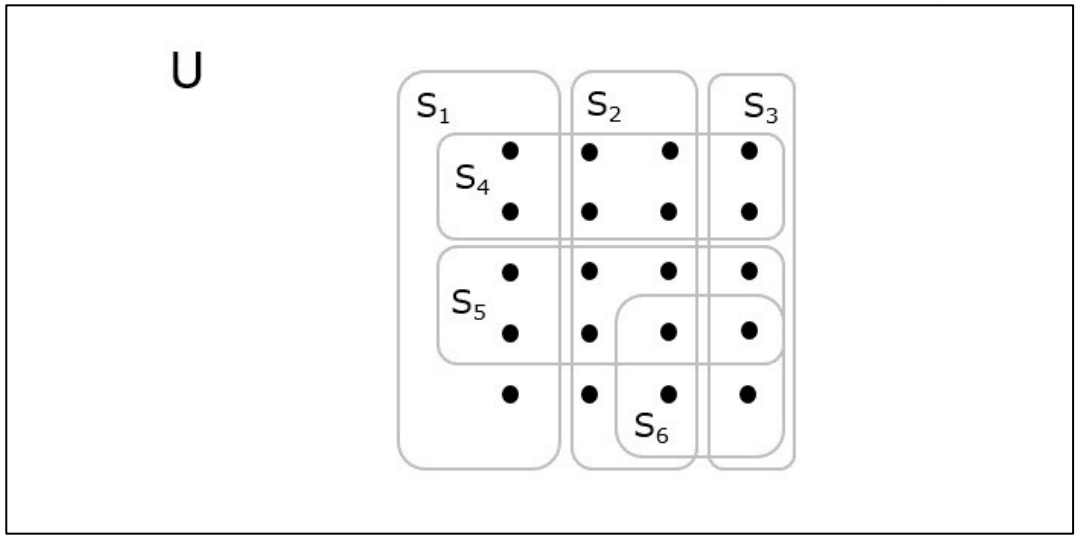
$$f(A \cup \{v\}) - f(A) \geq f(B \cup \{v\}) - f(B)$$

(The gain of v on A) $\geq$ (The gain of v on B)

Equivalent Definition of Submodularity: For any two sets $A, B \subseteq N$:

$$f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$$

This Problem Appears Everywhere (1)



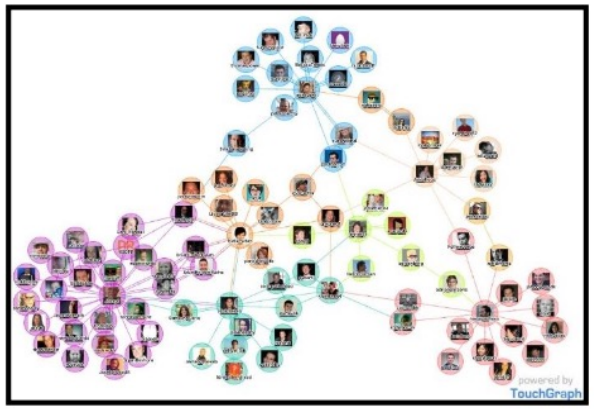
Max k-Coverage (from Set Cover)

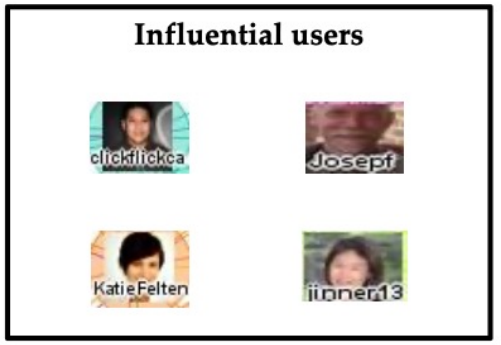
- **Ground Set N:** A collection of n subsets $\{S_1, \ldots, S_n\}$ whose union is U.
- **Function $f(S)$:** Let S be a selection of these subsets. $f(S) = |\cup_{S_i \in S} S_i|$.
- This is the "coverage" function. It is monotone (more sets can't cover less) and submodular (a new set covers fewer *new* elements if added to a large collection).

This Problem Appears Everywhere (2)

Influence Maximization in Networks

- **Ground Set N:** The nodes in a social network.
- **Function $f(S)$:** The *expected* number of nodes that will be "influenced" (e.t., adopt a product) if we "seed" the influence starting at the nodes in S.
- This is monotone and (provably) submodular under common cascade models.





The Greedy Algorithm

Algorithm: `Greedy_Submodular_Max(N, k, f)`

1. Initialize the solution set: $S = \emptyset$.
2. For $i = 1$ to k:
   - Find the item $v^* \in N \setminus S$ that provides the **maximum marginal gain** relative to the current set S.
   - $$v^* = \arg\max_{v \in N \setminus S} [f(S \cup \{v\}) - f(S)]$$
   - Update the solution: $S = S \cup \{v^*\}$.
3. Return S.

Polynomial Runtime: $O(nk)$ (since and $k \leq n$ and we have k iterations in total and we enumerate every remaining element in every single iteration),

The Greedy Algorithm: Approximation Ratio

This simple algorithm is incredibly effective for this class of problems.

Theorem (Nemhauser, Wolsey, Fisher 1978): Let S_G be the solution from the Greedy Algorithm and S_{OPT} be an optimal solution for the monotone submodular maximization problem with a **cardinality constraint** (meaning, the constraint that $|S| \leq k$).

Their values $f(S_G)$ and $f(S_{OPT})$ are guaranteed to satisfy:

$$\frac{f(S_{OPT})}{f(S_G)} \leq \frac{e}{e-1}$$

- e is the base of the natural logarithm ($e \approx 2.718...$).
- The ratio is $\frac{e}{e-1} \approx \mathbf{1.582}$.
- This approximation bound is **tight**; there are problem instances where the greedy algorithm achieves exactly this ratio.

Proof of Approximation Ratio (1)

We first define Notations:

- S_i = the greedy set after i iterations. ($S_0 = \emptyset$, $S_k = S_G$)
- S_{OPT} = the set of items in an **optimal solution**.
- $OPT = f(S_{OPT})$ = the value of the optimal solution.

Proof of Approximation Ratio (2)

Key Idea: At any step i, the "remaining value" $OPT - f(S_i)$ can be bounded by the sum of marginal gains of adding the items from the optimal solution.

1. By **monotonicity**, $f(S_i \cup S_{OPT}) \geq f(S_{OPT}) = OPT$.
2. So, $OPT - f(S_i) \leq f(S_i \cup S_{OPT}) - f(S_i)$.
3. By **submodularity**, the gain from adding the set $S_{OPT} \setminus S_i$ *at once* is less than or equal to the sum of gains from adding each of its elements *separately* to S_i:

$$f(S_i \cup S_{OPT}) - f(S_i) \leq \sum_{v \in S_{OPT} \setminus S_i} [f(S_i \cup \{v\}) - f(S_i)]$$

4. Combining (2) and (3):

$$OPT - f(S_i) \leq \sum_{v \in S_{OPT} \setminus S_i} \left[\underbrace{f(S_i \cup \{v\}) - f(S_i)}_{\text{Marginal gain of } v \text{ added to } S_i} \right]$$

Proof of Approximation Ratio (3)

From the previous slide:

$$OPT - f(S_i) \leq \sum_{v \in S_{OPT} \setminus S_i} [f(S_i \cup \{v\}) - f(S_i)]$$

1. The **Greedy Algorithm** picks the *best* possible gain at step $i+1$. Let this gain be $\Delta_{i+1} = f(S_{i+1}) - f(S_i)$.
2. This greedy gain must be at least as large as the gain from *any* other single item, including *any* item $v \in S_{OPT} \setminus S_i$.

$$\Delta_{i+1} \geq f(S_i \cup \{v\}) - f(S_i) \quad \text{for all } v \in S_{OPT} \setminus S_i$$

3. Therefore, Δ_{i+1} must be $\geq$ the *average* gain of the items in $S_{OPT} \setminus S_i$.
4. The set $S_{OPT} \setminus S_i$ has at most k items (since $|S_{OPT}| \leq k$).

$$OPT - f(S_i) \leq \sum_{v \in S_{OPT} \setminus S_i} \Delta_{i+1} \leq k \cdot \Delta_{i+1}$$

5. Substituting $\Delta_{i+1} = f(S_{i+1}) - f(S_i)$:

$$OPT - f(S_i) \leq k \cdot (f(S_{i+1}) - f(S_i))$$

Proof of Approximation Ratio (4)

We have the key recurrence:

$$f(S_{i+1}) - f(S_i) \geq \frac{1}{k}(OPT - f(S_i))$$

- Let $g_i = OPT - f(S_i)$ be the "gap" from optimal at step i.
- The recurrence can be rewritten as: $g_{i+1} \leq \left(1 - \frac{1}{k}\right) \cdot g_i$
- Unrolling this recurrence k times from $g_0 = OPT$: $g_k \leq \left(1 - \frac{1}{k}\right)^k \cdot g_0 = \left(1 - \frac{1}{k}\right)^k \cdot OPT$
- Using the standard inequality $\left(1 - \frac{1}{k}\right)^k \leq \frac{1}{e}$: $g_k \leq \frac{1}{e} \cdot OPT$
- g_k is the final gap: $g_k = OPT - f(S_k) = OPT - f(S_G)$.

$$OPT - f(S_G) \leq \frac{1}{e} \cdot OPT$$

Proof of Approximation Ratio (5)

- Rearranging this inequality to find the ratio:

$$OPT - \frac{1}{e} \cdot OPT \leq f(S_G)$$

$$OPT\left(1 - \frac{1}{e}\right) \leq f(S_G)$$

$$OPT\left(\frac{e-1}{e}\right) \leq f(S_G)$$

- **Final Ratio:**

$$\frac{OPT}{f(S_G)} \leq \frac{e}{e-1}$$

Summary of Submodularity

Definition: Submodularity is the formal, mathematical property of **diminishing returns**.

Generality: It appears *everywhere* in optimization: coverage, influence, sensor placement, network design, and more.

The Greedy Algorithm: For the canonical problem (monotone submodular max with a cardinality constraint), the simple greedy "hill-climbing" algorithm is provably near-optimal.

The Ratio: The greedy algorithm gives a constant-factor approximation:

$$\frac{f(S_{OPT})}{f(S_G)} \leq \frac{e}{e-1}$$

This is one of the most fundamental and powerful results in approximation algorithms.

Assignment 2 (Part 1)

Q1. Prove the greedy algorithm has no **constant-factor approximation ratio** for the 0/1 KP. (3 points)

Assignment 2 (Part 1)

Q2. Prove the Maximum k-Coverage Problem is a monotone submodular maximization problem. (3 points)

Setup: You are given a "universe" of elements $U = \{e_1, e_2, ..., e_m\}$ and a collection of n subsets of this universe, $N = \{C_1, C_2, ..., C_n\}$, where each $C_i \subseteq U$.

We define a **set function** $f : 2^N \to \mathbb{Z}_{\geq 0}$ over the collection of subsets N. For any $S \subseteq N$ (where S is a selection of subsets from N), the function $f(S)$ is defined as the **total number of unique elements** covered by the subsets in S.

$$f(S) = \left| \bigcup_{C_i \in S} C_i \right|$$

The optimization problem is $\max_{S \subseteq N} f(S)$ subject to $|S| \leq k$.

Your Task: Prove that this coverage function $f(S)$ is **monotone** and **submodular**.