CMPUT 605 Approximation Algorithms Individual Study

Keven Qiu
Instructor: Zachary Friggstad

Fall 2025

Contents

Chapter 1

Classic Approximations

Definition: α-Approximation Algorithm

For an optimization problem, it is a polynomial time algorithm that for all instances of the problem produces a solution whose value is within a factor α of the value of an optimal solution.

For minimization problems, we have $\alpha > 1$ and for maximization problems, $\alpha < 1$.

1.1 Vertex Cover

Problem: Vertex Cover

Given an undirected graph $G = (V, E)$ and a cost function $c : V \to \mathbb{Q}^+$, find a min cost vertex cover.

A way to establish an approximation guarantee is by lower bounding OPT. For cardinality vertex cover, we can get a good polynomial time computable lower bound on the size of the optimal cover.

Algorithm: 2-Approximation for Cardinality Vertex Cover

Find a maximal matching in G and output set of matched vertices.

Theorem (Cardinality Vertex Cover)

The algorithm is a 2-approximation algorithm for the cardinality vertex cover problem.

Proof. No edge can be left uncovered by the set of vertices picked. Otherwise, such an edge can have been added to the matching, contradicting maximality. Let M be this maximal matching. Since any vertex cover has to pick at least one endpoint of each matched edge, $|M| \leq \text{OPT}$. Our cover picked has cardinality $2|M| \leq 2 \cdot \text{OPT}$. ∎

Tight example: Complete bipartite graphs $K_{n,n}$. The algorithm will pick all $2n$ vertices, whereas optimal cover is picking one bipartition of n vertices.

The lower bound, of size of a maximal matching, is half the size of an optimal vertex cover. Consider complete graph K_n where n is odd. Then the size of any maximal matching is $\frac{n-1}{2}$, where as size of an optimal cover is $n-1$.

A NO certificate for maximum matchings in general graphs are odd set covers. These are a collection of disjoint odd cardinality subsets of V, $S_1, \ldots, S_k$ and vertices $v_1, \ldots, v_\ell$ such that each edge of G is incident with v_i or has both ends in S_j. Let C be the odd set cover, then it has cost

$$w(C) = \ell + \sum_{i=1}^{k} \frac{|S_i| - 1}{2}$$

Theorem (Generalized König)

In any graph,

$$\max_{\text{matching } M} |M| = \min_{\text{odd set cover } C} |C|$$

Corollary

In any graph,

$$\max_{\text{matching } M} |M| \leq \min_{\text{vertex cover } U} |U| \leq 2 \cdot \left(\max_{\text{matching } M} |M| \right)$$

1.2 Set Cover

Problem: Set Cover

Given a universe U of n elements, a collection of subsets of U, $\mathcal{S} = \{S_1, \ldots, S_k\}$, and a cost function $c : \mathcal{S} \to \mathbb{Q}^+$, find a min cost subcollection of $\mathcal{S}$ that covers all elements of U.

Define f as the frequency of the most frequent element. Set cover has f and $O(\log n)$ approximations. We present an $O(\log n)$-approximation here.

When $f = 2$, this is essentially the vertex cover problem.

A way to design approximation algorithms is by greedy. This is when we pick the most cost-effective choice at a particular time. Let C be the set of elements already covered. Define cost-effectiveness of a set S to be the average cost it covers new elements

$$\frac{c(S)}{|S - C|}$$

Lemma

For all $k \in \{1, \ldots, n\}$, $\text{price}(e_k) \leq \frac{\text{OPT}}{n-k+1}$

Proof. Let $e_1, \ldots, e_n$ be the order the algorithm covers the e_i's. Consider the time before e_k is covered. The remaining $n-k+1$ elements can be covered at a price/cost of no more than OPT. We can cover each element at a cost of no more than $\frac{\text{OPT}}{n-k+1}$ on average.

Suppose not, that is we cannot cover the rest of each element at a cost of no more than $\frac{\text{OPT}}{n-k+1}$ on average. Then the cost of covering the rest of the elements is $> (n-k+1) \cdot \frac{\text{OPT}}{n-k+1} = \text{OPT}$ which contradicts that the rest of the elements can be covered by $\leq$ OPT. ■

Theorem (Set Cover)

The greedy algorithm is an H_n-approximation algorithm, where $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$.

Proof. The total cost is

$$\sum_{k=1}^{n} \text{price}(e_k) \leq \sum_{k=1}^{n} \frac{\text{OPT}}{n-k+1} = \text{OPT}\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + 1\right) = H_n \cdot \text{OPT}$$

■

Tight example: Let $\varepsilon > 0$ be a small constant. $U = \{e_1, \ldots, e_n\}$, $\mathcal{S} = \{S_0, \ldots, S_n\}$, $c(S_0) = 1 + \varepsilon, c(S_k) = \frac{1}{k}$ for $k = 1, \ldots, n$. The cost of OPT is $1 + \varepsilon$ by choosing S_0. But greedy chooses S_k which has cost $\frac{1}{k} < 1 + \varepsilon$ for all $k = 1, \ldots, n$. So total cost is H_n.

1.3 Steiner Tree

Problem: Steiner Tree

Given $G = (V, E)$ with cost function $c : E \to \mathbb{R}_{\geq 0}$ and $V = R \cup S$ where R is the required set and S is the Steiner set, find a min cost tree in G that contains all vertices in R and any subset of S.

Theorem (Metric Steiner Tree Reduction)

There is an approximation factor preserving reduction from the Steiner tree problem to the metric Steiner tree problem.

Proof. Transform in polynomial time an instance I of G to an instance I' of the metric Steiner tree. Let G' be the complete undirected graph on V.

We construct G' as follows: $c(u, v) =$ shortest uv-path in G and the set of terminals is the same as G.

Claim 1: Cost of OPT in $G' \leq$ cost of OPT in G.
Proof of Claim 1. For all edges u, v in G, $c_{G'}(uv) \leq c_G(uv)$. ■

Claim 2: Cost of OPT in $G \leq$ cost of OPT in G'.
Proof of Claim 2. Let T' be a Steiner tree in G'. For all $uv \in E(G')$, replace uv with the shortest uv-path to obtain the subgraph T of G. Remove edges that create cycles in T. The cost does not increase, so $c_G(uv) \leq c_{G'}(uv)$. ■

■

Algorithm: Steiner Tree 2-Approximation

Find minimum spanning tree in induced subgraph $G[R]$.

Theorem (Steiner Tree 2-Approximation)

The minimum spanning tree on R is $\leq 2 \cdot \text{OPT}$.

Proof. Sketch: Optimal Steiner tree, double each edge, find Euler tour, shortcut vertices not in R and already seen vertices and delete heaviest edge. ■

1.4 Traveling Salesman Problem

Theorem

For any polynomial time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $\mathbf{P} = \mathbf{NP}$.

Proof. Assume for contradiction that it can be $\alpha(n)$-approximated with a polynomial time algorithm $\mathcal{A}$. We show $\mathcal{A}$ can be used to decide Hamiltonian cycle in polynomial time, implying $\mathbf{P} = \mathbf{NP}$.

Reduce the graph G on n vertices to an edge-weighted complete graph G' such that

- if G has a Hamiltonian cycle, then cost of optimal TSP tour in G' is n, and
- if G does not have a Hamiltonian cycle, then cost of optimal TSP your in G' is $> \alpha(n) \cdot n$.

Assign a weight of 1 to edges of G and weight $\alpha(n) \cdot n$ to non-edges to get G'. Now if G has a Hamiltonian cycle, then the corresponding tour has cost n in G'. Otherwise, if G has no Hamiltonian cycle, any tour in G' uses an edge of cost $\alpha(n) \cdot n$ and has cost $> \alpha(n) \cdot n$. ■

This violates the triangle inequality, so even though metric TSP is **NP**-complete, it is no longer hard to approximate.

Algorithm: Metric TSP 2-Approximation

1. Find MST T of G.
2. Double every edge of T to get Eulerian graph.
3. Find Eulerian tour $\mathcal{T}$.
4. Shortcut tour to get tour C.

Algorithm: Metric TSP $\frac{3}{2}$-Approximation (Christofides)

1. Find MST T in G.
2. Find min-cost perfect matching M on odd degree vertices.
3. $G' = T + M$.
4. Find Eulerian tour C and shortcut.

Lemma

Let $V' \subseteq V$, $|V'|$ is even, and M is min-cost perfect matching on V'. Then

$$\text{cost}(M) \leq \frac{\text{OPT}}{2}$$

Proof. Take an optimal TSP tour T of G. Let T' be tour on V' by shortcutting T. By triangle inequality, $\text{cost}(T') \leq \text{cost}(T)$.

T' is the union of 2 perfect matchings on V', consisting of alternating edges of T'. Cheapest of the matchings has cost $\leq$ cost(T') $/2 \leq \text{OPT}/2$ since M is a min-cost perfect matching.
■

Theorem (Christofides Algorithm)

Christofides is a $\frac{3}{2}$-approximation algorithm.

Proof.

$$c(C) \leq c(T) + c(M) \leq \text{OPT} + \frac{\text{OPT}}{2} = \frac{3}{2}\text{OPT}$$

■

1.5 Multiway Cut and k-Cuts

Problem: Multiway Cut

Given a set of terminals $S = \{s_1, \dots, s_k\}$, find a min-cost set of edges that when removed, disconnects S.

Algorithm: Multiway Cut $2 - \frac{2}{k}$-Approximation

1. For each $i = 1, \dots, k$, compute min-weight isolating cut for s_i, say C_i.
2. Discard heaviest cut C_j and output the union of all $\bigcup_{i=1}^{k} C_i \setminus C_j$.

Problem: Min k-Cut

Find min-cost set of edges whose removal leaves k connected components.

Algorithm: k-Cut $2 - \frac{2}{k}$-Approximation

1. Compute a Gomory-Hu tree T for G.
2. Output union C of the lightest $k - 1$ cuts from the $n - 1$ cuts associated with edges of T.

1.6 k-Center

Problem: k-Center

Given an undirected graph $G = (V, E)$ with distance $d_{ij} \geq 0$ for all pairs $i, j \in V$ and an integer k, find a set $S \subseteq V, |S| = k$ of k cluster centers, where we minimize the maximum distance of a vertex to its cluster center.

Algorithm: k-Center 2-Approximation

1. Pick arbitrary $i \in V$.
2. $S = \{i\}$.
3. While $|S| < k$, $S = S \cup \{\arg\max_{j \in V} d(j, S)\}$.

Theorem

The algorithm is a 2-approximation algorithm.

Proof. Let $S^* = \{j_1, \dots, j_k\}$ be the optimal solution with associated radius r^*. This partitions V into clusters $V_1, \dots, V_k$ where each $j \in V$ is placed in V_i if it is closest to j_i

among all in S^*. Each pair of points j and j' in the same cluster V_i are $\leq 2r^*$ apart. This is from triangle inequality; $d_{jj'} \leq d_{jj_i} + d_{j_ij'} = 2r^*$.

Let $S \subseteq V$ be points selected by the greedy algorithm. If one center in S is selected from each cluster of the optimal solution S^*, then every point in V is clearly within $2r^*$ of some point in S.

However, suppose in some iteration, the algorithm selects two points j, j' in the same cluster. The distance is at most $2r^*$. Suppose j' is selected first. Then it selects j since it was the furthest from the points already in S. Hence, all points are within a distance of at most $2r^*$ of some center already selected for S. Clearly, this remains true as the algorithm adds more centers in subsequent iterations. ■

1.7 Scheduling Jobs on Parallel Machines

Problem: Scheduling on Parallel Machines

Suppose there are n jobs, m machines, processing time p_j and no release dates. Complete all jobs as soon as possible, i.e.

$$\min \max_{j=1,\ldots,n} C_j$$

or the makespan of the schedule.

Algorithm: Local Search 2-Approximation

Start with any schedule and consider job j which finishes last. Check if there exists a machine to which j can be reassigned that would cause j to finish earlier. Repeat this until the last job cannot be transferred.

Theorem (Local Search 2-Approximation)

The local search for scheduling on multiple machines is a 2-approximation algorithm.

Proof. Let $C^*_{\max}$ be the length of an optimal schedule. Since each job must be processed,

$$C^*_{\max} \geq \max_{j=1,\ldots,n} p_j$$

There are in total $P = \sum p_j$ units of processing to accomplish. On average a machine will be assigned P/m units of work. At least one job must have at least that much work, so

$$C^*_{\max} \geq \sum_{j=1}^{n} p_j/m$$

Let ℓ be the last job in the final schedule of the algorithm and C_ℓ is completion time. Every machine must busy from time 0 to start of job ℓ at time $S_\ell = C_\ell - p_\ell$. Partition the schedule from time 0 to S_ℓ and S_ℓ to C_ℓ.

The latter interval has length at most $C^*_{\max}$ by first inequality.

The first interval has total work being mS_ℓ, which is no more than total work to be done P, so $S_\ell \leq \sum p_j/m$.

Combining with second inequality, $S_\ell \leq C^*_{\max}$, so in total the makespan is at most $2C^*_{\max}$.

We can refine this proof even more. $S_\ell \leq \sum p_j/m$ includes p_ℓ, but S_ℓ does not include job ℓ, so

$$S_\ell \leq \sum_{j \neq \ell} p_j/m$$

and so total length is at most

$$p_\ell + \sum_{j \neq \ell} p_j/m = \left(1 - \frac{1}{m}\right) p_\ell + \sum_{j=1}^{n} p_j/m$$

Applying two lower bounds at the start, we have $\leq \left(2 - \frac{1}{m}\right) C^*_{\max}$. ■

To show running time, we use $C_{\min}$ and show that it cannot decrease and that we never transfer the same job twice.

Algorithm: Greedy (List Scheduling) 2-Approximation

Order jobs in a list and whenever a machine becomes idle, assign next job on that machine.

If we use this schedule with local search, it would end immediately. Consider a job ℓ that is last to complete. Each machine is busy until $C_\ell - p_\ell$, since otherwise we would have assigned job ℓ to that other machine. So no transfers are possible.

Theorem

The longest processing time rule ($p_1 \geq \cdots \geq p_n$) is a $\frac{4}{3}$-approximation algorithm.

Chapter 2

Polynomial-Time Approximation Schemes

Definition: Polynomial Time Approximation Scheme (PTAS)

Let Π be an **NP**-hard optimization problem with objective function f_Π. $\mathcal{A}$ is a polynomial time approximation scheme if on input (I, ε) for fixed $\varepsilon > 0$, it outputs

- $f_\Pi(I, s) \leq (1 + \varepsilon) \cdot \text{OPT}$ if Π is a minimization problem.
- $f_\Pi(I, s) \geq (1 - \varepsilon) \cdot \text{OPT}$ if Π is a maximization problem.

and its running time is bounded by a polynomial in the size of I.

Definition: Fully Polynomial Time Approximation Scheme (FPTAS)

An approximation scheme where the running time of $\mathcal{A}$ is bounded by a polynomial in the size of instance I and $1/\varepsilon$.

2.1 Knapsack

Problem: Knapsack

Given a set $I = \{1, \ldots, n\}$ of items, with specified weight and values in $\mathbb{Z}^+$ and a knapsack capacity $B \in \mathbb{Z}^+$, find a subset of items whose total weight is bounded by B and total profit is maximized.

Definition: Pseudopolynomial Time Algorithm

An algorithm for problem Π whose running time on instance I is bounded by a polynomial in $|I_u|$ (number of bits need to write the unary size of I).

Algorithm: Pseudopolynomial Knapsack

Let P be most valuable object, $P = \max_{i \in I} v_i$, then nP is the upper bound on the profit of any solution. Let $S_{i,v}$ be the subset of $\{1, \dots, i\}$ whose total value is exactly v and whose total size is minimized. Let $A(i, v)$ be the size of the set $S_{i,v}$. $A(1, v)$ is known for $\{0, \dots, nP\}$.

$$A(i+1, p) = \begin{cases} \min\{A(i,p), w_{i+1} + A(i, p - v_{i+1})\} & \text{if } v_{i+1} \le p \\ A(i,p) & \text{otherwise} \end{cases}$$

This dynamic programming algorithm runs in $O(n^2 P)$. The maximum profit achievable is $\max\{p : A(n,p) \le B\}$.

Algorithm: FPTAS for Knapsack

1. Given ε and $P = \max_{i \in I} v_i$, let $K = \frac{\varepsilon P}{n}$.
2. $v_i' = \left\lfloor \frac{v_i}{K} \right\rfloor$.
3. Solve Knapsack with Dynamic Programming on new profits to get S.

Theorem

For all $\varepsilon > 0$, there is an FPTAS for Knapsack that has value $\ge (1 - \varepsilon)\mathrm{OPT}$.

Proof. Let S^* be the optimal solution. Note that $\mathrm{OPT} \ge P$ and $\frac{v_i}{K} - 1 \le v_i' \le \frac{v_i}{K}$.

The last fact gives $v_i' \le \frac{v_i}{K} \le \frac{P}{K} \le \frac{n}{\varepsilon}$. Since DP solves knapsack in $O(n^2 P)$, then this FPTAS runs in $O(n^3/\varepsilon)$.

Now we bound the value of S, the set outputted by our FPTAS.

$$\begin{aligned}
\sum_{i \in S} v_i &\ge K \sum_{i \in S} v_i' \\
&\ge K \sum_{i \in S^*} v_i' && \text{(Since } S \text{ is optimal for values } v_i'\text{)} \\
&\ge K \sum_{i \in S^*} \left(\frac{v_i}{K} - 1 \right) \\
&= \sum_{i \in S^*} (v_i - K) \\
&= \sum_{i \in S^*} v_i - K\,|S^*| \\
&\ge \mathrm{OPT} - nK && (|S^*| \le n) \\
&= \mathrm{OPT} - \varepsilon P \\
&\ge \mathrm{OPT} - \varepsilon \mathrm{OPT} && (\mathrm{OPT} \ge P) \\
&= (1 - \varepsilon)\mathrm{OPT}
\end{aligned}$$

■

2.2 Strong NP-Hardness and Existence of FPTAS

Very few of the known **NP**-hard problems admit a FPTAS.

Definition: Strongly NP-Hard

A problem Π is strongly **NP**-hard if every problem in **NP** can be polynomially reduced to Π in such a way that numbers in the reduced instance are always written in unary.

A strongly **NP**-hard problem cannot have a pseudo-polynomial time algorithm, assuming $\mathbf{P} \neq \mathbf{NP}$. Therefore, knapsack is not strongly **NP**-hard.

Theorem

Let p be a polynomial and Π be an **NP**-hard minimization problem such that the objective function f_Π is integer valued and on any instance I, $\text{OPT}(I) < p(|I_u|)$. If Π admits an FPTAS, then it also admits a pseudo-polynomial time algorithm.

Proof. Suppose there is an FPTAS for Π whose running time on instance I and error parameter ε is $q(|I|, 1/\varepsilon)$, where q is a polynomial.

On instance I, set the error parameter to $\varepsilon = 1/p(|I_u|)$ and run the FPTAS. Now, the solution produced will have objective function value less than or equal to

$$(1+\varepsilon)\text{OPT}(I) < \text{OPT}(I) + \varepsilon p(|I_u|) = \text{OPT}(I) + 1$$

With this error parameter, the FPTAS will be forced to produce an optimal solution. The running time will be $q(|I|, p(|I_u|))$, i.e. polynomial in $|I_u|$. Therefore, we have obtained a pseudo-polynomial algorithm for Π. ■

Corollary

Let Π be an **NP**-hard optimization problem satisfying the restrictions of the theorem. If Π is strongly **NP**-hard, then Π does not admit an FPTAS, assuming $\mathbf{P} \neq \mathbf{NP}$.

Proof. If Π admits an FPTAS, then it admits a pseudo-polynomial time algorithm by theorem. But the it is not strongly **NP**-hard, assuming $\mathbf{P} \neq \mathbf{NP}$, a contradiction. ■

2.3 Bin Packing

Problem: Bin Packing

Given n items I with sizes $s_1, \ldots, s_n \in (0, 1]$, find a packing in unit-sized bins that minimizes number of bins used.

The simple 2-approximation algorithm called First-Fit is as follows: Consider items in an arbitrary order. In the ith step, it has a list of partially packed bins $B_1, \ldots, B_k$. It attempts to put the item s_i in one of these bins in order. If s_i does not fit in any of these bins, it opens a new bin B_{k+1} and puts s_i in it.

If the algorithm uses m bins, then at least $m-1$ bins are more than half full. Therefore,

$$\sum_{i=1}^{n} s_i > \frac{m-1}{2}$$

Since the sum of the item sizes is a lower bound on OPT, $m-1 < 2\text{OPT} \implies m \leq 2\text{OPT}$.

Theorem

For any $\varepsilon > 0$, there is no approximation algorithm having a guarantee of $\frac{3}{2} - \varepsilon$ for the bin packing problem, unless $\mathbf{P} \neq \mathbf{NP}$.

Proof. If there were such an algorithm, then we show how to solve the **NP**-hard problem of deciding if there is a way to partition n nonnegative numbers $a_1, \ldots, a_n$ into two sets, each adding up to $\frac{1}{2}\sum_i a_i$. Clearly, the answer to this question is YES iff the n items can be packed in 2 bins of size $\frac{1}{2}\sum_i a_i$.

We can think of normalizing the Partition problem instance so that $\sum_i a_i = 2$. Since the sum is 2, the optimal bin packing requires ≥ 2 bins. If we had a $\frac{3}{2} - \varepsilon$-approximation, then we can solve this using < 3 bins, which means we can solve it optimally exactly. But if there is no solution to the Partition problem, we need ≥ 3 bins. ■

2.3.1 Asymptotic PTAS

Definition: Asymptotic PTAS (APTAS)

A family of algorithms $\{A_\varepsilon\}$ along with a constant c where is an algorithm A_ε for each $\varepsilon > 0$ such that A_ε returns a solution of value at most $(1+\varepsilon)\text{OPT}+c$ for minimization problems.

Theorem

For any $0 < \varepsilon \leq 1$, there is an algorithm A_ε that runs in time $n^{O(1/\varepsilon^2)}$ and finds a packing using at most $(1+\varepsilon)\text{OPT}+1$ bins.

Idea is to ignore small items and approximately add the large items. However, we cannot scale down items like we did with knapsack and solve with DP since we may overpack bins when returning items to original size.

We instead scale items up, but we need to scale them up in a way so that the optimum value does not increase too much.

Definition: SIZE(I)

$$\text{SIZE}(I) = \sum_{i \in I} s_i$$

Lemma

Given a packing of $I_{large} = \{i \in I : s_i \geq \varepsilon/2\}$ into b bins, we can efficiently find a packing of I using $\max\{b, (1+\varepsilon)\text{OPT} + 1\}$ bins.

Proof. Extend the packing of large items by adding small items one at a time. Create a new bin one only if none of the current bins can hold the small item.

If no new bins were created, we have b bins.

Otherwise, let b' be the total number of bins used. Since $s_i < \varepsilon/2$ for $i \in I_{small}$, we only create bins if the other bins contain total size $\geq 1 - \varepsilon/2$.

$$(b' - 1)(1 - \varepsilon/2) \leq \text{SIZE}(I) \leq \text{OPT}$$

All bins, except possibly the last bin we created will contain $\geq 1 - \varepsilon/2$. So,

$$b' \leq \frac{\text{OPT}}{1 - \varepsilon/2} + 1 \leq (1+\varepsilon)\text{OPT} + 1$$

where the inequality $\frac{1}{1-\varepsilon/2} \leq 1 + \varepsilon$ holds for $0 \leq \varepsilon \leq 1$. ■

Linear Grouping: For a given value k, create a new instance I': Order $I_{large} = \{i \in I : s_i \geq \varepsilon/2\}$ in nonincreasing order $s_1 \geq s_2 \geq \cdots \geq s_{n_\ell}$ where $n_\ell = |I_{large}|$. Create groups $G_1 = \{s_1, \ldots, s_k\}, G_2 = \{s_{k+1}, \ldots, s_{2k}\}, \ldots$ where the last group G_h has at most k items.

The new instance I' contains items $\{k+1, \ldots, n_\ell\} = \bigcup_{i=2}^{h} G_i$, i.e. disregard first group. For an item $i \in I'$ that is in group G_a, let $s'_i = \max\{s_i : i \in G_a\}$ (round each item's size to the largest item's size of its group).

Lemma

For each $i \in I'$, $s_{i-k} \geq s'_i \geq s_i$.

We will denote $\text{OPT}(I_{large})$ as optimal solution for I_{large}, $\text{OPT}(I)$ as optimal solution for original instance I, and $\text{OPT}(I')$ as optimal solution for I'. Clearly, $\text{OPT}(I_{large}) \leq \text{OPT}(I)$.

Lemma

For instance I' with sizes s'_i obtained from I_{large} using linear grouping,

$$\text{OPT}(I') \leq \text{OPT}(I_{large}) \leq \text{OPT}(I') + k$$

Given a packing of I' into b bins, we can efficiently find a packing of I_{large} into at most $b + k$ bins.

Proof. First inequality: Consider an optimal solution for I_{large}. Pack each item $i \in I'$ in the same bin as where $i-k \in I_{large}$ is packed. Since $s'_i \leq s_{i-k}$, this produces a feasible packing for I' using at most $\mathrm{OPT}(I_{large})$ bins.

Second inequality: Consider a packing of I' into b bins. We can pack I_{large} by packing items $1, \ldots, k$ into their own separate bins and packing each item $i \geq k+1$ into the same bin as item $i \in I'$. Since $s_i \leq s'_i$, this produces a feasible packing of I_{large} using at most $b+k$ bins. ■

We use $k = \lfloor \varepsilon \cdot \mathrm{SIZE}(I_{large}) \rfloor$. Consider the input I', then the number of distinct piece sizes m is $m \leq \frac{n_\ell}{k}$, because we round each item in each group up, which is also $\leq h-1$ since h was the last group. $\mathrm{SIZE}(I_{large}) \geq \varepsilon n_\ell / 2$. Thus, for $k = \lfloor \varepsilon \cdot \mathrm{SIZE}(I_{large}) \rfloor$, we have

$$\begin{aligned} m &= h-1 \\ &\leq \frac{n_\ell}{k} \\ &= \frac{n_\ell}{\lfloor \varepsilon \cdot \mathrm{SIZE}(I_{large}) \rfloor} \\ &\leq \frac{n_\ell}{\varepsilon \cdot \mathrm{SIZE}(I_{large})/2} && (*) \\ &= \frac{2n_\ell}{\varepsilon \cdot \mathrm{SIZE}(I_{large})} \\ &= \frac{2n_\ell}{\varepsilon \cdot \varepsilon n_\ell / 2} \\ &= \frac{4}{\varepsilon^2} \end{aligned}$$

$(*)$ comes from the fact that $\lfloor x \rfloor \geq x/2$. We can assume in this case $x = \varepsilon \cdot \mathrm{SIZE}(I_{large}) \geq 1$ since otherwise there are at most $(1/\varepsilon)/(\varepsilon/2) = 2/\varepsilon^2$ large pieces and we could apply the DP algorithm to solve the input optimally without having to do linear grouping.

After linear grouping, we are left with a bin packing input where there are a constant number of distinct piece sizes and only a constant number of pieces can fit in each bin. So we can obtain an optimal packing for I' using DP (see next section on Minimum Makespan Scheduling). Say these distinct item sizes are $a_1, \ldots, a_m$. Any bin with items from I' can be identified with a tuple of nonnegative integers $(b_1, \ldots, b_m)$ where $\sum_{j=1}^m b_j a_j \leq 1$. Furthermore, since $a_j \geq \varepsilon/2$, then the number of items in this bin is at most $2/\varepsilon$, i.e. $\sum_{j=1}^m b_j \leq 2/\varepsilon$.

Let $\mathcal{C}$ be the set of all nonzero tuples $(b_1, \ldots, b_m)$ that represent a bin of size at most 1. Since $0 \leq b_j \leq 2/\varepsilon$, the number of such tuples is $\leq (2/\varepsilon + 1)^m \leq (3/\varepsilon)^{4/\varepsilon^2}$.

For any tuples $(b_1, \ldots, b_m)$, let $f(b_1, \ldots, b_m)$ be the min number of bins required to pack the set of items consisting of b_j items of size a_j. Since $|\mathcal{C}|$ is constant, then the DP algorithm for Minimum Makespan Scheduling can be used to compute $f(\bar{b}_1, \ldots, \bar{b}_m)$ in $n^{O(1/\varepsilon^2)}$, where $\bar{b}_j$ is the number of items in I' having size a_j (bins are machines and item sizes are processing times).

The packing for I' can be used to get a packing for ungrouped input, then extend with small items greedily.

Proof of Theorem. Compute an optimum packing of I' using DP in runtime $n^{O(1/\varepsilon^2)}$. By previous lemma, we can transform this to a packing of I_{large} using at most $b = \text{OPT}(I_{large}) + k = \text{OPT}(I_{large}) + \lfloor \varepsilon \cdot \text{SIZE}(I_{large}) \rfloor$ bins. Since $\text{OPT}(I) \geq \text{OPT}(I_{large}) \geq \text{SIZE}(I_{large})$, then

$$b \leq \text{OPT}(I) + \varepsilon\text{OPT}(I) = (1+\varepsilon)\text{OPT}(I)$$

The algorithm will open $\max\{b, (1+\varepsilon)\text{OPT}(I) + 1\} = (1+\varepsilon)\text{OPT} + 1$ bins to pack the small items, where b is the number of bins used to pack large items. ■

Algorithm: Bin Packing APTAS

1. Separate I into I_{small} and I_{large} by item size threshold of $\varepsilon/2$.
2. Linear grouping with $k = \lfloor \varepsilon \cdot \text{SIZE}(I_{large}) \rfloor$ and dynamic programming.
3. First-fit on the small items into the bins filled from previous step.

2.4 Minimum Makespan Scheduling

Problem: Minimum Makespan Scheduling

Given processing times for n jobs, $p_1, \ldots, p_n$, and an integer m, find an assignment of the jobs to m identical machines so that the completion time (makespan) is minimized.

We saw a 2-approximation using local search and greedy. However, this problem is strongly **NP**-hard, and thus, does not admit a FPTAS, assuming $\mathbf{P} \neq \mathbf{NP}$.