Bin packing and scheduling

Overview

- Bin packing: problem definition
- Simple 2-approximation (Next Fit)
- Better than 3/2 is not possible
- Asymptotic PTAS
- Scheduling: minimizing the makespan (repeat)
- PTAS
Bin packing: problem definition

- **Input:** $n$ items with sizes $a_1, \ldots, a_n \in (0, 1]$

- **Goal:** pack these items into a **minimal number of bins**

- **Each bin has size 1**
Bin packing: problem definition

- Input: $n$ items with sizes $a_1, \ldots, a_n \in (0, 1]$  
- Goal: pack these items into a minimal number of bins  
- Each bin has size 1
A simple (online!) algorithm is **Next Fit**

- Place items in a bin until next item *does not fit*
- Then, close the bin and start a new bin
- Approximation ratio is 2 (and competitive ratio is also 2)
A simple (online!) algorithm is **Next Fit**

- Place items in a bin until next item does not fit
- Then, close the bin and start a new bin
- Approximation ratio is 2 (and competitive ratio is also 2)
A simple (online!) algorithm is **Next Fit**

- Place items in a bin until next item does not fit
- Then, close the bin and start a new bin
- Approximation ratio is 2 (and competitive ratio is also 2)
A simple (online!) algorithm is **Next Fit**

- Place items in a bin until next item does not fit
- Then, close the bin and start a new bin
- Approximation ratio is 2 (and competitive ratio is also 2)
A general lower bound

It is possible to improve on Next Fit, for instance by using First Fit.

However...

**Lemma 1.** There is no algorithm with approximation ratio below \(\frac{3}{2}\), unless \(P=NP\)

Proof: reduction from **PARTITION**

**PARTITION** = given a set of items of total size \(B\), can you split them into two subsets of equal size?

This problem is known to be **NP-hard**
The reduction

- Input is a set of items of total size 2
- Does this input fit in two bins?
- An algorithm with approximation ratio $< \frac{3}{2}$ must give a packing in \textit{two} bins (not three) if one exists
- Thus, it must solve PARTITION, which is NP-hard
The asymptotic performance ratio

- This result deals with “small” inputs
- What about more reasonable instances?
- For a given input $I$, let $\text{OPT}(I)$ denote the optimal number of bins needed to pack it
- Idea: we are interested in the worst ratio for large inputs
The asymptotic performance ratio

\[
\frac{\mathcal{A}(I)}{\text{OPT}(I)}
\]
The asymptotic performance ratio

\[ \sup_I \frac{\mathcal{A}(I)}{\text{OPT}(I)} \]
The asymptotic performance ratio

\[ R_A = \limsup_{n \to \infty} \sup_{I} \left\{ \frac{A(I)}{OPT(I)} \middle| OPT(I) = n \right\}. \]

Note: we also use this measure to compare online algorithms.
A positive result

We can show the following theorem:

**Theorem 1.** *For any* \( \epsilon > 0 \), there is an algorithm \( A_\epsilon \) that runs in time polynomial in \( n \) and for which

\[
A_\epsilon(I) \leq (1 + 2\epsilon)\text{OPT}(I) + 1 \quad \forall I
\]

Meaning: you can get as close to the optimal solution as you want

The degree of the polynomial depends on \( \epsilon \): the closer you want to get to the optimum, the more time it takes
A simple case

- All items have size at least $\varepsilon$
  - $\Rightarrow$ at most $M = \lceil 1/\varepsilon \rceil$ items fit in a bin

- There are only $K$ different item sizes
  - $\Rightarrow$ at most $R = \binom{M+K}{M}$ bin types
    - ($M$ “items” in a bin, $K+1$ options per item)

- We know that at most $n$ bins are needed to pack all items
  - $\Rightarrow$ at most $\binom{n+R}{R}$ feasible packings need to be checked

- We can do this in polynomial (in $n$) time

Note: this is extremely impractical

Example: $n = 50$, $K = 6$, $\varepsilon = 1/3$, then $1.98 \cdot 10^{37}$ options
Generalizing the simple case (1)

Suppose there are more different item sizes (at most $n$).

Do the following:

- Sort items
- Make groups containing $\lfloor n\varepsilon^2 \rfloor$ items
- In each group, round sizes up to largest size in group
Generalizing the simple case (2)

So far we had a lower bound of $\varepsilon$ on the item sizes.

How do we pack instances that also contain such small items?

- Ignore items $< \varepsilon$ (small items) at the start
- Apply algorithm on remaining items
- Fill up bins with small items
The small items

- If all small items fit in the bins used to pack $L$, we use no more than $\text{OPT}(L)$ bins
- Else, all bins except the last are full by at least $1 - \varepsilon$

- $\text{OPT}(I)$ is at least the total size of all the items

\[ \text{OPT}(I) \geq \frac{\text{ALG}(I) - 1}{1 - \varepsilon} \rightarrow \text{ALG}(I) \leq (1 + 2\varepsilon)\text{OPT}(I) + 1 \]

This proves the theorem.
A better solution

☐ The core algorithm is very much **brute force**

☐ We can improve by using dynamic programming

☐ We no longer need a lower bound on the sizes

☐ There are \( k \) different item sizes

☐ An input is of the form \((n_1, \ldots, n_k)\)

☐ We want to calculate \( \text{OPT}(n_1, \ldots, n_k) \), the optimal number of bins to pack this input
The base case

- Consider an input \((n_1, \ldots, n_k)\) with \(n = \sum n_j\) items
- Determine set of \(k\)-tuples (subsets of the input) that can be packed into a single bin
- That is, all tuples \((q_1, \ldots, q_k)\) for which \(\text{OPT}(q_1, \ldots, q_k) = 1\) and for which \(0 \leq q_j \leq n_j\) for all \(j\)
- There are at most \(n^k\) such tuples, each tuple can be checked in linear time
- (Exercise: there are at most \((n/k)^k\) such tuples)
- Denote this set by \(Q\)
Dynamic programming

- For each $k$-tuple $q \in Q$, we have $\text{OPT}(q) = 1$
- Calculate remaining values by using the recurrence

\[
\text{OPT}(i_1, \ldots, i_k) = 1 + \min_{q \in Q} \text{OPT}(i_1 - q_1, \ldots, i_k - q_k)
\]

- Exercise: think about the order in which we can calculate these values
- Each value takes $O(n^k)$ time, so we can calculate all values in $O(n^{2k})$ time
- This gives us in the end the value of $\text{OPT}(n_1, \ldots, n_k)$
Advantages

☐ Much faster than simple brute force

☐ Can be used to create PTAS for load balancing!

PTAS from Algorithmentechnik:

☐ separate $\ell$ largest jobs

☐ assign them optimally

☐ add smallest jobs greedily

Time $O(m^\ell + n)$. For $\varepsilon = 1/3, m = 15$ we have $m^\ell = 8.5 \cdot 10^{32}$. This PTAS could only be used for very small $m$ and large $\varepsilon$. 
Scheduling Independent Weighted Jobs on Parallel Machines

\( x(j): \) Machine where job \( j \) is executed

\( L_i: \) \( \sum_{x(j)=i} t_j \), load of machine \( i \)

Objective: Minimize Makespan

\( L_{\text{max}} = \max_i L_i \)

Details: Identical machines, independent jobs, known processing times, offline

NP-hard
Old results

- Greedy algorithm is \((2 - \frac{1}{m})\)-approximation
- LPT is \((4/3 - \frac{1}{3m})\)-approximation

New result: **PTAS for load balancing**

Idea: find optimal makespan using **binary search**
A step in the binary search

Let current guess for the makespan be $t$
A step in the binary search

- Let current guess for the makespan be $t$
- Remove “small” items: smaller than $t\varepsilon$
A step in the binary search

- Let current guess for the makespan be $t$
- Remove “small” items: smaller than $t\varepsilon$
- Round remaining sizes down using geometric rounding
A step in the binary search

1. Let current guess for the makespan be $t$
2. Remove “small” items: smaller than $t\epsilon$
3. Round remaining sizes down using geometric rounding
4. Find optimal solution in bins of size $t$
A step in the binary search

- Let current guess for the makespan be $t$
- Remove “small” items: smaller than $t \varepsilon$
- Round remaining sizes down using geometric rounding
- Find optimal solution in bins of size $t$
- Extend to near-optimal solution for entire input
A step in the binary search

- Let current guess for the makespan be $t$
- Remove “small” items: smaller than $t\varepsilon$
- Round remaining sizes down using geometric rounding
- Find optimal solution in bins of size $t$
- Extend to near-optimal solution for entire input
- More than $m$ bins needed: increase $t$
- At most $m$ bins needed: decrease $t$
Geometric rounding

- Each large item is rounded down so that its size is of the form
  \[ t\varepsilon(1 + \varepsilon)^i \]
  for some \( i \geq 0 \)

- Since large items have size at least \( t\varepsilon \), this leaves
  \( k = \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil \) different sizes
Geometric rounding

- Each large item is rounded down so that its size is of the form
  \[ t\varepsilon(1 + \varepsilon)^i \]
  for some \( i \geq 0 \)

- Since large items have size at least \( t\varepsilon \), this leaves
  \( k = \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil \) different sizes

We find a packing for the rounded down items in bins of size \( t \)
Geometric rounding

- Each large item is rounded down so that its size is of the form
  \[ t\varepsilon(1 + \varepsilon)^i \]
  for some \( i \geq 0 \)

- Since large items have size at least \( t\varepsilon \), this leaves
  \( k = \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil \) different sizes

We find a packing for the rounded down items in bins of size \( t \)

This gives a valid packing in bins of size \( t(1 + \varepsilon) \)
Geometric rounding

Each large item is rounded down so that its size is of the form

\[ t\varepsilon(1 + \varepsilon)^i \]

for some \( i \geq 0 \)

Since large items have size at least \( t\varepsilon \), this leaves \( k = \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil \) different sizes

We find a packing for the rounded down items in bins of size \( t \)

This gives a valid packing in bins of size \( t(1 + \varepsilon) \)

We add the small items to those bins (and to new bins if needed)
Comparing to the optimal solution

We use bins of size \( t(1 + \varepsilon) \)

*Claim:* OPT needs at least as many bins of size \( t \) to pack these items
Comparing to the optimal solution

We use bins of size \( t(1 + \varepsilon) \)

**Claim:** OPT needs at least as many bins of size \( t \) to pack these items

**Proof:** If we need no extra bins for the small items, we have found an optimal packing for the rounded down items in bins of size \( t \)
Comparing to the optimal solution

We use bins of size $t(1 + \varepsilon)$

**Claim:** OPT needs at least as many bins of size $t$ to pack these items

**Proof:** If we need no extra bins for the small items, we have found an optimal packing for the rounded down items in bins of size $t$

Else, all bins (except maybe the last one) are full by at least $t$
Connection between bin packing and scheduling

- We look for the smallest $t$ such that we can pack the items in $m$ bins (machines).
- Suppose that we can find the exact value of $t$
- Then, OPT also needs $m$ bins of size $t$ to pack these items
- In other words, the makespan on $m$ machines is at least $t$. (For smaller $t$, the items cannot all be placed below a level of $t$.)
The binary search

- We start with the following lower bound on \( \text{OPT} \):\[ LB = \max \left\{ \sum_{j} t_j/m, \max_j t_j \right\} \]

- Greedy gives a schedule which is at most twice this value, this is an upper bound for \( \text{OPT} \)

- Each step of the binary search halves this interval

- We repeat until the length of the interval is at most \( \varepsilon \cdot LB \)

- Let \( T \) be the upper bound of this interval

- Then \( T \leq \text{OPT} + \varepsilon \cdot LB \leq (1 + \varepsilon) \cdot \text{OPT} \)

- The makespan of our algorithm is at most \( (1 + \varepsilon)T \)
Conclusion

**Theorem 2.** For any $\varepsilon > 0$, there is an algorithm $A_\varepsilon$ which works in polynomial time in $n$ and which gives a schedule with makespan at most $(1 + \varepsilon)^2 \text{OPT} < (1 + 3\varepsilon)\text{OPT}$. 
Conclusion

**Theorem 2.** For any $\varepsilon > 0$, there is an algorithm $A_\varepsilon$ which works in polynomial time in $n$ and which gives a schedule with makespan at most $(1 + \varepsilon)^2 \text{OPT} < (1 + 3\varepsilon)\text{OPT}$.

Notes:

- The number of item sizes is $k = \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil$
Conclusion

**Theorem 2.** For any $\varepsilon > 0$, there is an algorithm $A_\varepsilon$ which works in polynomial time in $n$ and which gives a schedule with makespan at most $(1 + \varepsilon)^2 \text{OPT} < (1 + 3\varepsilon)\text{OPT}$.

Notes:

- The number of item sizes is $k = \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil$
- The number of iterations in the binary search is $\lfloor \log_2 \frac{1}{\varepsilon} \rfloor$
Conclusion

**Theorem 2.** For any $\varepsilon > 0$, there is an algorithm $A_\varepsilon$ which works in polynomial time in $n$ and which gives a schedule with makespan at most $(1 + \varepsilon)^2 \text{OPT} < (1 + 3\varepsilon)\text{OPT}$. 

Notes:

- The number of item sizes is $k = \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil$
- The number of iterations in the binary search is $\lceil \log_2 \frac{1}{\varepsilon} \rceil$
- The running time of the dynamic programming algorithm is $O(n^{2k})$
Conclusion

**Theorem 2.** For any $\varepsilon > 0$, there is an algorithm $\mathcal{A}_\varepsilon$ which works in polynomial time in $n$ and which gives a schedule with makespan at most $(1 + \varepsilon)^2 \text{OPT} < (1 + 3\varepsilon) \text{OPT}$.

Notes:

- The number of item sizes is $k = \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil$
- The number of iterations in the binary search is $\lceil \log_2 \frac{1}{\varepsilon} \rceil$
- The running time of the dynamic programming algorithm is $O(n^{2k})$
- The running time of our algorithm is $O(\lceil \log_2 \frac{1}{\varepsilon} \rceil n^{2k})$

$n = 50, \varepsilon = 1/3 \rightarrow 7.8 \cdot 10^{13}$ options
Conclusion

**Theorem 2.** For any $\varepsilon > 0$, there is an algorithm $A_\varepsilon$ which works in polynomial time in $n$ and which gives a schedule with makespan at most $(1 + \varepsilon)^2 \text{OPT} < (1 + 3\varepsilon) \text{OPT}$.

Notes:

- The number of item sizes is $k = \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil$
- The number of iterations in the binary search is $\lceil \log_2 \frac{1}{\varepsilon} \rceil$
- The running time of the dynamic programming algorithm is $O(n^{2k})$
- The running time of our algorithm is $O(\lceil \log_2 \frac{1}{\varepsilon} \rceil n^{2k})$
- There is no FPTAS for this problem