Approximation Algorithms

Q. Suppose I need to solve an NP-hard problem. What should I do?
A. Theory says you're unlikely to find a poly-time algorithm.

Must sacrifice one of three desired features.

- Solve problem to optimality.
- Solve problem in poly-time.
- Solve arbitrary instances of the problem.

$\rho$-approximation algorithm.

- Guaranteed to run in poly-time.
- Guaranteed to solve arbitrary instance of the problem
- Guaranteed to find solution within ratio $\rho$ of true optimum.

Challenge. Need to prove a solution's value is close to optimum, without even knowing what optimum value is!
Load Balancing

Input. \( m \) identical machines; \( n \) jobs, job \( j \) has processing time \( t_j \).
- Job \( j \) must run contiguously on one machine.
- A machine can process at most one job at a time.

Def. Let \( J(i) \) be the subset of jobs assigned to machine \( i \). The load of machine \( i \) is \( L_i = \sum_{j \in J(i)} t_j \).

Def. The **makespan** is the maximum load on any machine \( L = \max_i L_i \).

Load balancing. Assign each job to a machine to minimize makespan.
List-scheduling algorithm.
  - Consider $n$ jobs in some fixed order.
  - Assign job $j$ to machine whose load is smallest so far.

```plaintext
List-Scheduling(m, n, t_1, t_2, ..., t_n) {
    for i = 1 to m {
        L_i ← 0 ← load on machine i
        J(i) ← ∅ ← jobs assigned to machine i
    }

    for j = 1 to n {
        i = \text{argmin}_k L_k ← machine i has smallest load
        J(i) ← J(i) ∪ \{j\} ← assign job j to machine i
        L_i ← L_i + t_j ← update load of machine i
    }

    return J(1), ..., J(m)
}
```

Implementation. $O(n \log m)$ using a priority queue.
Load Balancing: List Scheduling Analysis

- First worst-case analysis of an approximation algorithm.
- Need to compare resulting solution with optimal makespan $L^*$.

Lemma 1. The optimal makespan $L^* \geq \max_j t_j$.

Pf. Some machine must process the most time-consuming job. •

Lemma 2. The optimal makespan $L^* \geq \frac{1}{m} \sum_j t_j$.

Pf.
- The total processing time is $\sum_j t_j$.
- One of $m$ machines must do at least a $1/m$ fraction of total work. •
Load Balancing: List Scheduling Analysis

**Theorem.** Greedy algorithm is a 2-approximation.

**Pf.** Consider load \( L_i \) of bottleneck machine \( i \).

- Let \( j \) be last job scheduled on machine \( i \).
- When job \( j \) assigned to machine \( i \), \( i \) had smallest load. Its load before assignment is \( L_i - t_j \Rightarrow L_i - t_j \leq L_k \) for all \( 1 \leq k \leq m \).
- Sum inequalities over all \( k \) and divide by \( m \):

\[
L_i - t_j \leq \frac{1}{m} \sum_k L_k = \frac{1}{m} \sum_k t_k \leq L^* \quad \text{Lemma 1} \quad \rightarrow \quad L_i \leq 2L^*. \quad \text{Lemma 2}
\]
Load Balancing: LPT Rule

Longest processing time (LPT). Sort $n$ jobs in descending order of processing time, and then run list scheduling algorithm.

```plaintext
LPT-List-Scheduling(m, n, t_1, t_2, ..., t_n) {
    Sort jobs so that $t_1 \geq t_2 \geq ... \geq t_n$

    for $i = 1$ to $m$
        $L_i \leftarrow 0$ ← load on machine $i$
        $J(i) \leftarrow \emptyset$ ← jobs assigned to machine $i$
    }

    for $j = 1$ to $n$
        $i = \text{argmin}_k L_k$ ← machine $i$ has smallest load
        $J(i) \leftarrow J(i) \cup \{j\}$ ← assign job $j$ to machine $i$
        $L_i \leftarrow L_i + t_j$ ← update load of machine $i$
    }

    return $J(1), ..., J(m)$
}
```
Load Balancing: LPT Rule

Observation. If at most m jobs, then list-scheduling is optimal.
Pf. Each job put on its own machine.

Lemma 3. If there are more than m jobs, L* ≥ 2t_{m+1}.
Pf.
- Consider first m+1 jobs t_1, ..., t_{m+1}.
- Since the t_i's are in descending order, each takes at least t_{m+1} time.
- There are m+1 jobs and m machines, so by pigeonhole principle, at least one machine gets two jobs.

Theorem. LPT rule is a 3/2 approximation algorithm.
Pf. Same basic approach as for list scheduling.

\[
L_i = \frac{(L_i - t_j)}{L^*} + \frac{t_j}{L^*} \leq \frac{3}{2}L^*.
\]

Lemma 3
(by observation, can assume number of jobs > m)
Polynomial Time Approximation Scheme

PTAS. $(1 + \varepsilon)$-approximation algorithm for any constant $\varepsilon > 0$.
- Load balancing. [Hochbaum-Shmoys 1987]
- Euclidean TSP. [Arora 1996]

Consequence. PTAS produces arbitrarily high quality solution, but trades off accuracy for time.

This section. PTAS for knapsack problem via rounding and scaling.
Knapsack Problem

Knapsack problem.
- Given $n$ objects and a "knapsack."
- Item $i$ has value $v_i > 0$ and weighs $w_i > 0$.  
  
  → we'll assume $w_i \leq W$
- Knapsack can carry weight up to $W$.
- Goal: fill knapsack so as to maximize total value.

Ex: \{3, 4\} has value 40.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

$W = 11$
Knapsack is NP-Complete

**KNAPSACK**: Given a finite set $X$, nonnegative weights $w_i$, nonnegative values $v_i$, a weight limit $W$, and a target value $V$, is there a subset $S \subseteq X$ such that:

$$\sum_{i \in S} w_i \leq W$$

$$\sum_{i \in S} v_i \geq V$$

**SUBSET-SUM**: Given a finite set $X$, nonnegative values $u_i$, and an integer $U$, is there a subset $S \subseteq X$ whose elements sum to exactly $U$?

**Claim.** $\text{SUBSET-SUM} \leq_{P} \text{KNAPSACK}$.

**Pf.** Given instance $(u_1, \ldots, u_n, U)$ of SUBSET-SUM, create KNAPSACK instance:

$$v_i = w_i = u_i$$

$$\sum_{i \in S} u_i \leq U$$

$$V = W = U$$

$$\sum_{i \in S} u_i \geq U$$
Knapsack Problem: Dynamic Programming 1

**Def.** $OPT(i, w) =$ max value subset of items 1,..., i with weight limit $w$.

- **Case 1:** $OPT$ does not select item $i$.
  - $OPT$ selects best of 1, ..., i-1 using up to weight limit $w$
- **Case 2:** $OPT$ selects item $i$.
  - new weight limit = $w - w_i$
  - $OPT$ selects best of 1, ..., i-1 using up to weight limit $w - w_i$

$$OPT(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
OPT(i-1, w) & \text{if } w_i > w \\
\max \{ OPT(i-1, w), \ v_i + OPT(i-1, w-w_i) \} & \text{otherwise}
\end{cases}$$

**Running time.** $O(n W)$.

- $W =$ weight limit.
- **Not polynomial** in input size!
Knapsack Problem: Dynamic Programming II

Def. \( \text{OPT}(i, v) = \text{min weight subset of items } 1, \ldots, i \text{ that yields value exactly } v \).

- Case 1: \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( 1, \ldots, i-1 \) that achieves exactly value \( v \)

- Case 2: \( \text{OPT} \) selects item \( i \).
  - consumes weight \( w_i \), new value needed = \( v - v_i \)
  - \( \text{OPT} \) selects best of \( 1, \ldots, i-1 \) that achieves exactly value \( v \)

\[
\text{OPT}(i, v) = \begin{cases} 
0 & \text{if } v = 0 \\
\infty & \text{if } i = 0, v > 0 \\
\text{OPT}(i-1, v) & \text{if } v_i > v \\
\min\{ \text{OPT}(i-1, v), w_i + \text{OPT}(i-1, v - v_i) \} & \text{otherwise}
\end{cases}
\]

Running time. \( O(n V^*) = O(n^2 v_{\max}) \).

- \( V^* \) = optimal value = maximum \( v \) such that \( \text{OPT}(n, v) \leq W \).
- Not polynomial in input size!
Knapsack: FPTAS

Intuition for approximation algorithm.
- Round all values up to lie in smaller range.
- Run dynamic programming algorithm on rounded instance.
- Return optimal items in rounded instance.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>934,221</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5,956,342</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>17,810,013</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>21,217,800</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>27,343,199</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

original instance

rounded instance

W = 11
Knapssack: FPTAS

Knapssack FPTAS. Round up all values: $\overline{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta$, $\hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil$

- $v_{\text{max}}$ = largest value in original instance
- $\varepsilon$ = precision parameter
- $\theta$ = scaling factor = $\varepsilon v_{\text{max}} / n$

Observation. Optimal solution to problems with $\overline{v}$ or $\hat{v}$ are equivalent.

Intuition. $\overline{v}$ close to $v$ so optimal solution using $\overline{v}$ is nearly optimal; $\hat{v}$ small and integral so dynamic programming algorithm is fast.

Running time. $O(n^3 / \varepsilon)$.
- Dynamic program II running time is $O(n^2 \hat{v}_{\text{max}})$, where
  $\hat{v}_{\text{max}} = \left\lceil \frac{v_{\text{max}}}{\theta} \right\rceil = \left\lceil \frac{n}{\varepsilon} \right\rceil$
Knapsack: FPTAS

Knapsack FPTAS. Round up all values: \( \bar{v}_i = \left\lfloor \frac{v_i}{\theta} \right\rfloor \theta \)

Theorem. If \( S \) is solution found by our algorithm and \( S^* \) is any other feasible solution then
\[
(1 + \varepsilon) \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i
\]

Pf. Let \( S^* \) be any feasible solution satisfying weight constraint.

\[
\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \bar{v}_i \leq \sum_{i \in S} \bar{v}_i \leq \sum_{i \in S} (v_i + \theta) \leq \sum_{i \in S} v_i + n\theta \leq (1 + \varepsilon) \sum_{i \in S} v_i
\]

always round up

solve rounded instance optimally

never round up by more than \( \theta \)

\(|S| \leq n\)

DP alg can take \( v_{\text{max}} \)

\[n \theta = \varepsilon v_{\text{max}}, \quad v_{\text{max}} \leq \sum_{i \in S} v_i\]