11. APPROXIMATION ALGORITHMS

- load balancing
- center selection
- pricing method: vertex cover
- LP rounding: vertex cover
- generalized load balancing
- knapsack problem

Coping with NP-completeness

Q. Suppose I need to solve an NP-hard problem. What should I do?

A. Sacrifice one of three desired features.
   i. Solve arbitrary instances of the problem.
   ii. Solve problem to optimality.
   iii. Solve problem in polynomial time.

\( \rho \)-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.
Load balancing on 2 machines is NP-hard

**Claim.** Load balancing is hard even if only 2 machines.

**Pf.** \textsc{Number-Partitioning} \textbf{NP} \textless \text{P} \textsc{Load-Balance}.

NP-complete by Exercise 8.26

- \[
\begin{array}{cccc}
  a & b & c & d \\
  e & f & g
\end{array}
\]

- length of job \( f \)

machine 1

- \[
\begin{array}{ccc}
  a & d & f
\end{array}
\]

machine 2

- \[
\begin{array}{cccc}
  b & c & e & g
\end{array}
\]

- yes

Load balancing: list scheduling

**List-scheduling algorithm.**

- Consider \( n \) jobs in some fixed order.
  - Assign job \( j \) to machine whose load is smallest so far.

```c
List-Scheduling(m, n, t_1, t_2, \ldots, t_n) {
    for i = 1 to m {
        L_i ← 0 ← load on machine i
        J(i) ← \emptyset ← jobs assigned to machine i
    }
    for j = 1 to n {
        i = \arg \min_k L_k
        J(i) ← J(i) \cup \{j\} ← assign job \( j \) to machine \( i \)
        L_i ← L_i + t_j ← update load of machine \( i \)
    }
    return J(1), \ldots, J(m)
}
```

**Implementation.** \( O(n \log m) \) using a priority queue.

Load balancing: list scheduling analysis

**Theorem.** [Graham 1966] Greedy algorithm is a 2-approximation.

- 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 \( L^* \geq \frac{1}{n} \sum_j t_j \).

**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.

Believe it or not
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 $1 \leq k \leq m$.

\[ L_i - t_j \leq L_k \text{ for all } 1 \leq k \leq m. \]

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

**Q.** Is our analysis tight?
**A.** Essentially yes.

**Ex:** $m$ machines, $m(m-1)$ jobs length 1 jobs, one job of length $m$. 

**Optimal makespan = 10**
Load balancing: LPT rule

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

```
LPT-List-Scheduling(m, n, t₁, t₂,..., tₙ) {
    Sort jobs so that $t₁ ≥ t₂ ≥ ... ≥ tₙ$
    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 $J(1), ..., J(m)$
}
```

Load Balancing: LPT rule

Q. Is our 3/2 analysis tight?
A. No.

**Theorem.** [Graham 1969] LPT rule is a 4/3-approximation.
**Pf.** More sophisticated analysis of same algorithm.

Q. Is Graham's 4/3 analysis tight?
A. Essentially yes.

**Ex:** $m$ machines, $n = 2m + 1$ jobs, 2 jobs of length $m$, $m + 1$, ..., $2m - 1$ and one more job of length $m$.

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 $t₁, ..., t_{m+1}$.
- 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.

$$L_i = \frac{(L_i - t_j)}{L^*} + \frac{t_j}{L^*} ≤ \frac{3}{2} L^*.$$  

(by observation, can assume number of jobs $> m$)

11. APPROXIMATION ALGORITHMS

- load balancing
- center selection
- pricing method: vertex cover
- LP rounding: vertex cover
- generalized load balancing
- knapsack problem
Center selection problem

**Input.** Set of $n$ sites $s_1, \ldots, s_n$ and an integer $k > 0$.

**Center selection problem.** Select set of $k$ centers $C$ so that maximum distance $r(C)$ from a site to nearest center is minimized.

![Diagram of center selection problem](image)

**Center selection example**

**Ex:** each site is a point in the plane, a center can be any point in the plane, $\text{dist}(x, y)$ = Euclidean distance.

**Remark:** search can be infinite!

**Greedy algorithm: a false start**

**Greedy algorithm.** Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.

**Remark:** arbitrarily bad!
Center selection: greedy algorithm

Repeatedly choose next center to be site farthest from any existing center.

\[
\text{GREEDY-CENTER-SELECTION} (k, n, s_1, s_2, \ldots, s_n) \\
C \leftarrow \emptyset. \\
\text{REPEAT} k \text{ times} \\
\quad \text{Select a site } s_i \text{ with maximum distance } \text{dist}(s_i, C). \\
\quad C \leftarrow C \cup s_i. \\
\text{RETURN } C.
\]

Property. Upon termination, all centers in \(C\) are pairwise at least \(r(C)\) apart.

Pf. By construction of algorithm.

Center selection

Lemma. Let \(C^*\) be an optimal set of centers. Then \(r(C) \leq 2r(C^*)\).

Theorem. Greedy algorithm is a 2-approximation for center selection problem.

Remark. Greedy algorithm always places centers at sites, but is still within a factor of 2 of best solution that is allowed to place centers anywhere.

\[\text{e.g., points in the plane}\]

Question. Is there hope of a 3/2-approximation? 4/3?

Center selection: analysis of greedy algorithm

Lemma. Let \(C^*\) be an optimal set of centers. Then \(r(C) \leq 2r(C^*)\).

Pf. [by contradiction] Assume \(r(C^*) < \frac{1}{2} r(C)\).

- For each site \(c_i \in C\), consider ball of radius \(\frac{1}{2} r(C)\) around it.
- Exactly one \(c_i^*\) in each ball; let \(c_i\) be the site paired with \(c_i^*\).
- Consider any site \(s\) and its closest center \(c_i^* \in C^*\).
- \(\text{dist}(s, C) \leq \text{dist}(s, c_i) \leq \text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i) \leq 2r(C^*)\).
- Thus, \(r(C) \leq 2r(C^*)\).

Dominating set reduces to center selection

Lemma. \(C^*\) satisfies the triangle inequality.

Theorem. Unless \(P = NP\), there no \(\rho\)-approximation for center selection problem for any \(\rho < 2\).

Pf. We show how we could use a \((2 - \epsilon)\) approximation algorithm for CENTER-SELECTION to solve DOMINATING-SET in poly-time.

- Let \(G = (V, E)\), \(k\) be an instance of DOMINATING-SET.
- Construct instance \(G'\) of CENTER-SELECTION with sites \(V\) and distances
  - \(\text{dist}(u, v) = 1\) if \((u, v) \in E\)
  - \(\text{dist}(u, v) = 2\) if \((u, v) \notin E\)
- Note that \(G'\) satisfies the triangle inequality.
- \(G\) has dominating set of size \(k\) iff there exists \(k\) centers \(C^*\) with \(r(C^*) = 1\).
- Thus, if \(G\) has a dominating set of size \(k\), a \((2 - \epsilon)\)-approximation algorithm for CENTER-SELECTION would find a solution \(C^*\) with \(r(C^*) = 1\) since it cannot use any edge of distance 2.
null
### Pricing method example

**Theorem.** Pricing method is a 2-approximation for WEIGHTED-VERTEX-COVER.

- Algorithm terminates since at least one new node becomes tight after each iteration of while loop.

- Let $S = \text{set of all tight nodes upon termination of algorithm.}$
- $S$ is a vertex cover: if some edge $(i,j)$ is uncovered, then neither $i$ nor $j$ is tight. But then while loop would not terminate.

- Let $S^*$ be optimal vertex cover. We show $w(S) \leq 2w(S^*)$.

$$w(S) = \sum_{i \in S} w_i \leq \sum_{i \in S} \sum_{e \in (i,j)} p_e \leq \sum_{i \in V} \sum_{e \in (i,j)} p_e = 2 \sum_{e \in E} p_e \leq 2w(S^*).$$

### Weighted vertex cover

Given a graph $G = (V, E)$ with vertex weights $w_i \geq 0$, find a min weight subset of vertices $S \subseteq V$ such that every edge is incident to at least one vertex in $S$. 

**total weight = 6 + 9 + 10 + 32 = 57**
Weighted vertex cover: IP formulation

Given a graph $G = (V, E)$ with vertex weights $w_j \geq 0$, find a min weight subset of vertices $S \subseteq V$ such that every edge is incident to at least one vertex in $S$.

Integer programming formulation.
- Model inclusion of each vertex $i$ using a 0/1 variable $x_i$.
  
  $$x_i = \begin{cases} 
  0 & \text{if vertex } i \text{ is not in vertex cover} \\
  1 & \text{if vertex } i \text{ is in vertex cover}
  \end{cases}$$

  Vertex covers in 1–1 correspondence with 0/1 assignments:
  $S = \{ i \in V : x_i = 1 \}$.

- Objective function: minimize $\sum w_i x_i$.
- Must take either vertex $i$ or $j$ (or both): $x_i + x_j \geq 1$.

Observation. If $x^*$ is optimal solution to (ILP), then $S = \{ i \in V : x^*_i = 1 \}$ is a min weight vertex cover.

Linear programming

Given integers $a_{ij}$, $b_i$, and $c_j$, find real numbers $x_j$ that satisfy:

$$(P) \quad \max \ c' x \quad \text{s.t. } \sum_{j=1}^n a_{ij} x_j \leq b_i \quad 1 \leq i \leq m$$

$$(P) \quad \max \ c' x \quad \text{s.t. } A x \leq b \quad x \geq 0$$

Observation. Vertex cover formulation proves that INTEGER-PROGRAMMING is an NP-hard search problem.


LP feasible region

LP geometry in 2D.

### Weighted vertex cover: LP relaxation

**Linear programming relaxation.**

\[
(LP) \quad \begin{array}{ll}
\min & \sum_{i \in V} w_i x_i \\
\text{s.t.} & x_i + x_j \geq 1 \quad (i,j) \in E \\
& x_i \geq 0 \quad i \in V
\end{array}
\]

**Observation.** Optimal value of (LP) is \( \leq \) optimal value of (ILP).

**Pf.** LP has fewer constraints.

**Note.** LP is not equivalent to vertex cover.

**Q.** How can solving LP help us find a small vertex cover?

**A.** Solve LP and round fractional values.

### Weighted vertex cover: LP rounding algorithm

**Lemma.** If \( x^* \) is optimal solution to (LP), then \( S = \{ i \in V : x_i^* \geq \frac{1}{2} \} \) is a vertex cover whose weight is at most twice the min possible weight.

**Pf.** [S is a vertex cover]

- Consider an edge \((i,j) \in E\).
- Since \( x_i^* + x_j^* \geq 1 \), either \( x_i^* \geq \frac{1}{2} \) or \( x_j^* \geq \frac{1}{2} \) \( \Rightarrow \) \((i, j) \) covered.

**Pf.** [S has desired cost]

- Let \( S^* \) be optimal vertex cover. Then

\[
\sum_{i \in S^*} w_i \geq \sum_{i \in S} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i
\]

**Theorem.** The rounding algorithm is a 2-approximation algorithm.

**Pf.** Lemma + fact that LP can be solved in poly-time.

### Weighted vertex cover inapproximability

**Theorem.** [Dinur-Safra 2004] If \( P \neq NP \), then no \( \rho \)-approximation for WEIGHTED-VERTEX-COVER for any \( \rho < 1.3606 \) (even if all weights are 1).

---

**On the Hardness of Approximating Minimum Vertex Cover**

Irit Dinur† Samuel Safra‡

May 26, 2004

**Abstract**

We prove the Minimum Vertex Cover problem to be NP-hard to approximate to within a factor of 1.3606, extending on previous PCP hardness of approximation techniques. To that end, one needs to develop a new proof framework, and borrow and extract ideas from several fields.

**Open research problem.** Close the gap.
11. APPROXIMATION ALGORITHMS

- load balancing
- center selection
- pricing method: vertex cover
- LP rounding: vertex cover
- generalized load balancing
- knapsack problem

Generalized load balancing: integer linear program and relaxation

**ILP formulation.** $x_{ij} =$ time machine $i$ spends processing job $j$.

\[
(IP) \quad \begin{align*}
\text{min} & \quad L \\
\text{s.t.} & \quad \sum_{j \in J} x_{ij} = t_j \quad \text{for all } j \in J \\
& \quad \sum_{i \in M} x_{ij} \leq L \quad \text{for all } i \in M \\
& \quad x_{ij} \in \{0, t_j\} \quad \text{for all } j \in J \text{ and } i \in M_j \\
& \quad x_{ij} = 0 \quad \text{for all } j \in J \text{ and } i \notin M_j
\end{align*}
\]

**LP relaxation.**

\[
(LP) \quad \begin{align*}
\text{min} & \quad L \\
\text{s.t.} & \quad \sum_{j \in J} x_{ij} = t_j \quad \text{for all } j \in J \\
& \quad \sum_{i \in M} x_{ij} \leq L \quad \text{for all } i \in M \\
& \quad x_{ij} \geq 0 \quad \text{for all } j \in J \text{ and } i \in M_j \\
& \quad x_{ij} = 0 \quad \text{for all } j \in J \text{ and } i \notin M_j
\end{align*}
\]

Generalized load balancing: lower bounds

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

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

**Lemma 2.** Let $L$ be optimal value to the LP. Then, optimal makespan $L^* \geq L$.

**Pf.** LP has fewer constraints than IP formulation. □

Input. Set of $m$ machines $M$; set of $n$ jobs $J$.

- Job $j \in J$ must run contiguously on an authorized machine in $M_j \subseteq M$.
- Job $j \in J$ has processing time $t_j$.
- Each 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 $= \max_i L_i$. 

Generalized load balancing. Assign each job to an authorized machine to minimize makespan.
**Generalized load balancing: structure of LP solution**

**Lemma 3.** Let \( x \) be solution to LP. Let \( G(x) \) be the graph with an edge between machine \( i \) and job \( j \) if \( x_{ij} > 0 \). Then \( G(x) \) is acyclic.

**Pf.** (deferred)

\[ \begin{align*}
G(x) \text{ acyclic} & \Rightarrow \text{can transform } x \text{ into another LP solution where} \\
& \text{G}(x) \text{ is acyclic if LP solver doesn’t return such an } x
\end{align*} \]

**Generalized load balancing: rounding**

**Rounded solution.** Find LP solution \( x \) where \( G(x) \) is a forest. Root forest \( G(x) \) at some arbitrary machine node \( r \).

- If job \( j \) is a leaf node, assign \( j \) to its parent machine \( i \).
- If job \( j \) is not a leaf node, assign \( j \) to any one of its children.

**Lemma 4.** Rounded solution only assigns jobs to authorized machines.

**Pf.** If job \( j \) is assigned to machine \( i \), then \( x_{ij} > 0 \). LP solution can only assign positive value to authorized machines.

\[ \begin{align*}
G(x) \text{ cyclic} & \Rightarrow \text{can transform } x \text{ into another LP solution where} \\
& \text{G}(x) \text{ is acyclic if LP solver doesn’t return such an } x
\end{align*} \]

**Generalized load balancing: analysis**

**Theorem.** Rounded solution is a 2-approximation.

**Pf.**

- Let \( J(i) \) be the jobs assigned to machine \( i \).
- By Lemma 6, the load \( L_i \) on machine \( i \) has two components:

  - **leaf nodes:**
    \[ \sum_{j \in J(i)} t_j = \sum_{j \in J(i)} x_{ij} \leq \sum_{j \in J} x_{ij} \leq L \leq L^* \]
    
    **Lemma 5**
    
    \[ \sum_{j \in J(i)} x_{ij} \]
    
    **LP**
    
    **Lemma 2 (LP is a relaxation)**
    
    **optimal value of LP**
    
    **- parent:** \( t_{\text{parent}(i)} \leq L^* \)
    
    **Thus, the overall load** \( L_i \) \( \leq 2L^* \).

\[ \begin{align*}
G(x) \text{ acyclic} & \Rightarrow \text{can transform } x \text{ into another LP solution where} \\
& \text{G}(x) \text{ is acyclic if LP solver doesn’t return such an } x
\end{align*} \]
Generalized load balancing: flow formulation

Flow formulation of LP.

\[ \sum_{i} x_{ij} = t_j \text{ for all } j \in J \]

\[ \sum_{i} x_{ij} \leq L \text{ for all } i \in M \]

\[ x_{ij} \geq 0 \text{ for all } j \in J \text{ and } i \in M \]

\[ x_{ij} = 0 \text{ for all } j \in J \text{ and } i \notin M \]

**Observation.** Solution to feasible flow problem with value \( L \) are in 1-to-1 correspondence with LP solutions of value \( L \).

Conclusions

**Running time.** The bottleneck operation in our 2-approximation is solving one LP with \( mn + 1 \) variables.

**Remark.** Can solve LP using flow techniques on a graph with \( m + n + 1 \) nodes: given \( L \), find feasible flow if it exists. Binary search to find \( L^* \).

**Extensions: unrelated parallel machines.** [Lenstra-Shmoys-Tardos 1990]

- Job \( j \) takes \( t_j \) time if processed on machine \( i \).
- 2-approximation algorithm via LP rounding.
- If \( P \neq NP \), then no \( \rho \)-approximation exists for any \( \rho < 3/2 \).

Generalized load balancing: structure of solution

**Lemma 3.** Let \( (x, L) \) be solution to LP. Let \( G(x) \) be the graph with an edge from machine \( i \) to job \( j \) if \( x_{ij} > 0 \). We can find another solution \( (x', L) \) such that \( G(x') \) is acyclic.

**Pf.**

- Let \( C \) be a cycle in \( G(x) \).
  - Augment flow along the cycle \( C \). flow conservation maintained
  - At least one edge from \( C \) is removed (and none are added).
  - Repeat until \( G(x') \) is acyclic.

11. APPROXIMATION ALGORITHMS

- load balancing
- center selection
- pricing method: vertex cover
- LP rounding: vertex cover
- generalized load balancing
- knapsack problem
Polynomial-time approximation scheme

PTAS. \((1 + \epsilon)\)-approximation algorithm for any constant \(\epsilon > 0\).
- Load balancing. [Hochbaum-Shmoys 1987]
- Euclidean TSP. [Arora, Mitchell 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 is NP-complete

**Knapsack.** Given a set \(X\), weights \(w_i \geq 0\), values \(v_i \geq 0\), a weight limit \(W\), and a target value \(V\), is there a subset \(S \subseteq X\) such that:

\[
\sum_{j \in S} w_j \leq W \\
\sum_{j \in S} v_j \geq V
\]

**Subset-Sum.** Given a set \(X\), values \(u_i \geq 0\), and an integer \(U\), is there a subset \(S \subseteq X\) whose elements sum to exactly \(U\)?

**Theorem.** Subset-Sum \(\leq_p\) Knapsack.

**Pf.** Given instance \((u_1, \ldots, u_n, U)\) of Subset-Sum, create Knapsack instance:

\[
v_i = w_i = u_i \\
V = W = U \\
\sum_{i \in S} u_i \leq U
\]

Knapsack problem

**Knapsack problem.**
- Given \(n\) objects and a knapsack.
- Item \(i\) has value \(v_i > 0\) and weighs \(w_i > 0\). \(\quad\text{we assume } w_i \leq W\) for each \(i\)
- Knapsack has weight limit \(W\).
- 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>

original instance (\(W = 11\))

Knapsack problem: dynamic programming I

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

**Case 1.** \(OPT\) does not select item \(i\).
- \(OPT\) selects best of \(1, \ldots, i-1\) using up to weight limit \(w\).

**Case 2.** \(OPT\) selects item \(i\).
- New weight limit \(W\) \(= W - w_i\).
- \(OPT\) selects best of \(1, \ldots, i-1\) using up to weight limit \(W - w_i\).

**Theorem.** Computes the optimal value in \(O(n W)\) time.
- Not polynomial in input size.
- Polynomial in input size if weights are small integers.
Knapsack problem: dynamic programming II

**Def.** \( OPT(i, v) = \min \text{ weight of a knapsack for which we can obtain a solution of value } \geq v \text{ using a subset of items } 1, ..., i \).

**Note.** Optimal value is the largest value \( v \) such that \( OPT(n, v) \leq W \).

**Case 1.** \( OPT \) does not select item \( i \).
- \( OPT \) selects best of \( 1, ..., i-1 \) that achieves value \( \geq v \).

**Case 2.** \( OPT \) selects item \( i \).
- Consumes weight \( w_i \), need to achieve value \( \geq v - v_i \).
- \( OPT \) selects best of \( 1, ..., i-1 \) that achieves value \( \geq v - v_i \).

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

Knapsack problem: polynomial-time approximation scheme

**Intuition for approximation algorithm.**
- Round all values up to lie in smaller range.
- Run dynamic programming algorithm II on rounded/scaled 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>934221</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5956342</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>17810013</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>21217800</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>27343199</td>
<td>7</td>
</tr>
</tbody>
</table>

original instance (\( W = 11 \))

<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>

rounded instance (\( W = 11 \))

**Observation.** Optimal solutions to problem with \( \tilde{\tau} \) are equivalent to optimal solutions to problem with \( \hat{\tau} \).

**Intuition.** \( \tilde{\tau} \) close to \( \tau \) so optimal solution using \( \tau \) is nearly optimal; \( \hat{\tau} \) small and integral so dynamic programming algorithm II is fast.

Knapsack problem: polynomial-time approximation scheme

**Theorem.** Dynamic programming algorithm II computes the optimal value in \( O(n^2 v_{\text{max}}) \) time, where \( v_{\text{max}} \) is the maximum of any value.

**Pf.**
- The optimal value \( V^\circ \leq n v_{\text{max}} \).
- There is one subproblem for each item and for each value \( v \leq V^\circ \).
- It takes \( O(1) \) time per subproblem.

**Remark 1.** Not polynomial in input size!

**Remark 2.** Polynomial time if values are small integers.
Knapsack problem: polynomial-time approximation scheme

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

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

\[
\begin{align*}
\sum_{i \in S^*} v_i & \leq \sum_{i \in S^*} \hat{v}_i & \text{always round up} \\
& \leq \sum_{i \in S} \hat{v}_i & \text{solve rounded instance optimally} \\
& \leq \sum_{i \in S} (v_i + \theta) & \text{never round up by more than } \theta \\
& \leq \sum_{i \in S} v_i + n\theta & |S| \leq n \\
& = \sum_{i \in S} v_i + \frac{1}{2} \epsilon n v_{\max} & \theta = \epsilon v_{\max} / 2n \\
& \leq (1 + \epsilon) \sum_{i \in S} v_i & v_{\max} \leq 2 \sum_{i \in S} v_i
\end{align*}
\]

subset containing only the item of largest value

choosing \( S^* = \{ \text{max} \} \)

\[
\begin{align*}
v_{\max} & \leq \sum_{i \in S} v_i + \frac{1}{2} \epsilon v_{\max} \\
& \leq \sum_{i \in S} v_i + \frac{1}{2} v_{\max} \\
\text{thus} \quad v_{\max} & \leq 2 \sum_{i \in S} v_i
\end{align*}
\]

Knapsack problem: polynomial-time approximation scheme

**Theorem.** For any \( \epsilon > 0 \), the rounding algorithm computes a feasible solution whose value is within a \((1 + \epsilon)\) factor of the optimum in \(O(n^2/\epsilon)\) time.

**Pf.**

- We have already proved the accuracy bound.
- Dynamic program \( I \) running time is \(O(n^2 \hat{v}_{\max})\), where

\[
\hat{v}_{\max} = \left\lceil \frac{v_{\max}}{\theta} \right\rceil = \left\lceil \frac{2n}{\epsilon} \right\rceil
\]