CS711008Z Algorithm Design and Analysis

Lecture 11. Approximation algorithm: a brief introduction [1]

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

---

[1]The slides are made based on Chapter 11 of *Algorithm design*,
Approximation Algorithm by V. Vazirani, and *Combinatorial optimization
algorithm and complexity* by C. H. Papadimitriou and K. Steiglitz, and slides by
D. P. Williamson.

Outline

- Introduction and the first example: MAKESPAN problem;
- The key idea in approximation algorithm design: 1) constructing a feasible solution, and 2) "comparing with lower bound of OPT" rather than "comparing with OPT";
- How to find a lower bound of OPT? Combinatorial technique, LP-based techniques (LP-relaxation, duality, etc.);
- SETCOVER: a good example to demonstrate four techniques: Greedy, LP+Rounding, Dual-LP+Rounding, Primal_and_dual;
- Other techniques: scaling for KNAPSACK, pruning for k-CENTER, TSP, DISJOINTPATHS;

The process of design of approximation algorithm is not very different from that of design of exact algorithms. It still involves unraveling the problem structure and finding algorithm techniques to exploit it.

How to deal with hard problems?

- Most natural optimization problem, including those arising in important application areas, are NP-Hard.
- It is widely believed that there is no efficient algorithm to a NP-Hard problem.
- Recall that **efficient algorithm** refers to a **polynomial-time deterministic** algorithm to find **optimal** solution even in **worst case**.
- How to deal with hard problems? Trade-off "quality" and "time".

Trade-off "quality" and "time"

We have a couple of options:

1. Give up **polynomial-time** in **worst-case** restriction: though our algorithm takes exponential time in the worst case, we hope that the algorithm run fast on the practical instances, e.g. branch-and-bound, branch-and-cut, and branch-and-pricing algorithms are used to solve a TSP instance with over 24978 Swedish Cities (see http://www.tsp.gatech.edu/history/pictorial/sw24978.html)

2. Give up optimum restriction: from optimal solution to nearly optimal solution in the hope that nearly optimal is easy to find, e.g. approximation algorithm (with theoretical guarantee), heuristics, local search (without theoretical guarantee;

3. Give up deterministic restriction: we hope that the expectation of running time of a randomized algorithm might be polynomial;

Approximation strategy

Although this may seem a paradox, all exact science is dominated by the idea of approximation.

—— B. Russel

- Why do we study approximation algorithms?
  1. As an algorithm with theoretical guarantee;
  2. As a core algorithmic idea to solve practical problems after fine tuning;
  3. As a mathematically rigorous way to analyze heuristics;
  4. As a way to explore deeper into the combinatorial problem structure, to uncover problem structure;

Definition (α-approximation algorithm)

An algorithm is an α-approximation algorithm for a minimization problem if:

1. Time: the algorithm runs in polynomial time;
2. Quality: the algorithm outputs a solution S whose value is within a factor of α of the value of optimal solution (denoted as OPT), i.e. $Value(S) \leq \alpha OPT$.

Definition (PTAS)

A $PTAS$ (*polynomial time approximation schema*) for a minimization problem is a family of algorithms $\{A_\epsilon : \epsilon > 0\}$ such that for each ϵ, A_ϵ is a $(1 + \epsilon)$-approximation algorithm running in polynomial time in input size.
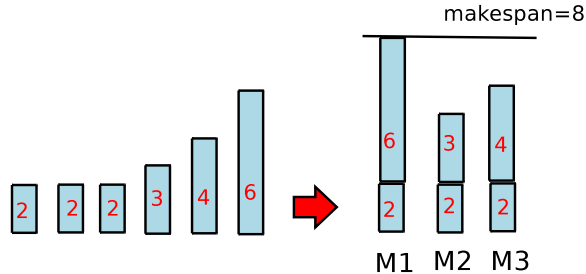
The first example: MAKESPAN problem

- Practical problem:
  - We have multiple servers to process a set of jobs. Intuitively, we try to make the loads as balanced as possible.
  - How to schedule jobs to machines to finish all jobs as early as possible?
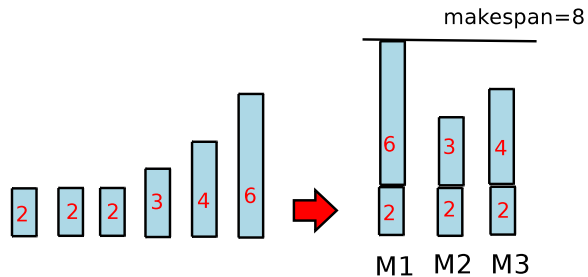
INPUT:
m servers $M_1, M_2, ..., M_m$, n jobs (each job j has a processing time t_j);
OUTPUT:
An assignment of jobs to machines to minimize the *makespan* , i.e. the maximum load on any machine, $T = \max_i \sum_{j \in A(i)} t_j$, where $A(i)$ denotes the jobs assigned to machine i.

Hardness of MakeSpan problem

Theorem

MakeSpan *is a NP-Complete problem.*

Proof.

- We will prove that PARTITION $\leq_P$ MakeSpan.
- **Transformation:**
  Given an instance of PARTITION problem, we construct an
  instance of MakeSpan problem as follows:
  1. For each number $s_i \in S$, new a job i with $t_i = s_i$;
  2. The objective is to find a schedule of these jobs on 2 machines
     with makespan $T = \frac{1}{2} \sum_i t_i$.
- **Equivalence:** It is obvious that a partition corresponds to a
 schedule.

□

- **Key observation:** solution is a partition of jobs.
- Basic idea: Imagine the solving process as a series of decisions. At each decision step, we assign a job to a machine. Consider a specific job. We have m options.
- Greedy rule: To make loads as balanced as possible, it is reasonable to assign a job to the machine with the smallest load.

GREEDYMAKESPAN1 algorithm

1: **for** $i = 1$ to m **do**
2:     $T_i = 0$; // T_i denotes the load of machine i;
3:     $A_i = $ NULL; //initializing all machines with 0 jobs;
4: **end for**
5: **for** $j = 1$ to n **do**
6:     Let $k = \arg\min T_i$;
7:     $A_k = A_k \bigcup \{j\}$; //assigning job j to machine M_k
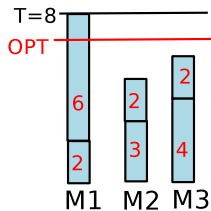8: $T_k = T_k + t_j$;
9: **end for**

6 Jobs
3 Machines

makespan=8

M1 M2 M3
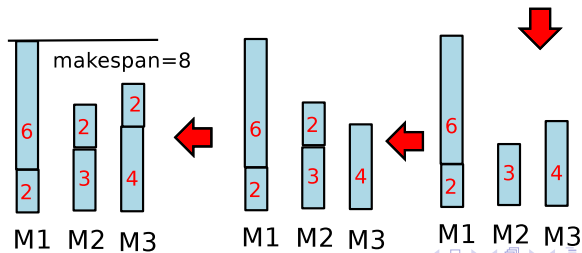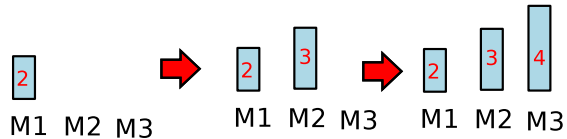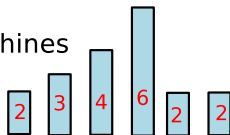
- Let T be the makespan reported by GREEDYMAKESPAN1 algorithm, and OPT be the optimal makespan.
- The objective is to measure the quality of T via comparing T with OPT.

- In the example, GREEDYMAKESPAN1 reports $T = 8$, which is not too bad since T will not be greater than $2OPT$.

- But how can we compare T against OPT while OPT is yet unknown?
- Note that though it is difficult to know OPT, it is usually easy to obtain a lower bound of OPT.
- Thus we can use the lower bound as a bridge to build connection between T and OPT. More specifically, we compare T with the lower bound of OPT rather than compare T with OPT directly.

- Take MAKESPAN problem as an example.
- Though OPT is unknown, we can easily set lower bound of OPT as follows:
    1. Key observation 1: $OPT \geq \frac{1}{m} \sum_j t_j = \frac{19}{3}$.
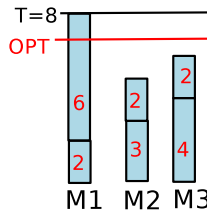   2. Key observation 2: $OPT \geq t_j$ for any j; thus, $OPT \geq 6$.
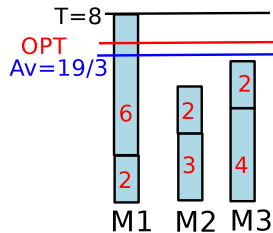
MakeSpanAlgo1 is not too bad

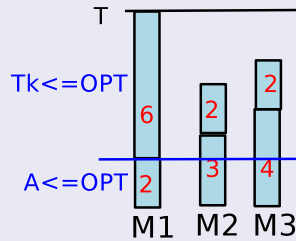- Using the lower bounds as bridge, we can prove the following theorem:

Theorem

$T \leq 2OPT$, i.e GreedyMakeSpanAlgo1 *is a 2-approximation algorithm.*

Proof.

- Let M_i be the machine with the heaviest load T;
- Divide T into two parts: the last job k, and the previous jobs. Thus $T = t_k + A$, where A denotes the total load of the previous jobs.
- We have $T \leq 2OPT$ since
  1. $t_k \leq OPT$ (by observation 2)
  2. $A \leq OPT$. Why?
     - Consider the state whenhe tjob k was assigned to M_i. At that time, M_i had a total load of A, which is the smallest load of all machines (by the greedy rule of the algorithm).
     - Formally, $A \leq \frac{1}{m}(\sum_{j=1}^{n} t_j - t_k) \leq \frac{1}{m}\sum_{j=1}^{n} t_j \leq OPT$ (by key observation 1).
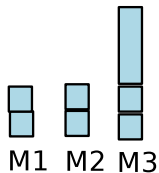
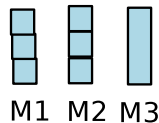Question: is this an accurate analysis?

- Consider a special instance: a total of $n = m(m-1) + 1$ jobs with loads $t_1 = t_2 = ... = t_{n-1} = 1$, and $t_n = m$.

GreedyMakeSpan1
returns T=2m-1

OPT=m

M1 M2 M3

M1 M2 M3

- GREEDYMAKESPAN1: $T = 2m - 1$ (if the largest job n is processed at the last step).
- OPT: $OPT = m$.
- Thus, approximation factor is: $\alpha = \frac{T}{OPT} = 2 - \frac{1}{m}$. α can be arbitrarily close to 2 when m increases.

Another greedy algorithm for MakeSpan

- Basic idea: The tight example for GREEDYMAKESPANALGO1 algorithm implies that it is not wise to process the largest job finally. In other words, it might be a good idea to process large jobs first.

T=7 OPT=7

M1 M2 M3
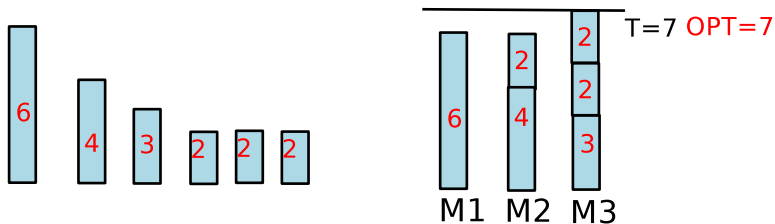
GREEDYMAKESPAN2 algorithm

1: **for** $i = 1$ to m **do**
2:     $T_i = 0$; //assigning all machines with 0 jobs;
3:     $A_i = $ NULL;
4: **end for**
5: **sort jobs in decreasing order of t_j;** //process large jobs first
6: **for** $j = 1$ to m **do**
7:     Let $k = argmin \ T_i$; //assign job j to M_k;
8:     $A_k = A_k \bigcup \{j\}$;
9:     $T_k = T_k + t_j$;
10: **end for**

- **Key observation 3:** $OPT \geq 2t_j$ for any $j \geq m + 1$ if all jobs were sorted decreasingly.
  (Why? Consider the first $m + 1$ jobs only. At least two jobs should be assigned to a machine. Thus, $OPT \geq 2t_{m+1}$.)

Analysis
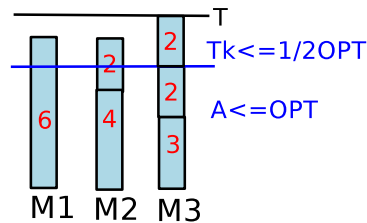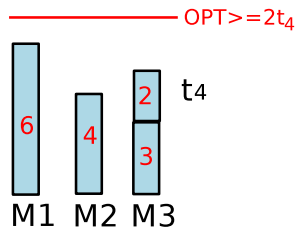
Theorem

$T \leq 1.5OPT$, *i.e,* GREEDYMAKESPANALGO2 *is a* 1.5-*approximation algorithm.*

Proof.

- Let M_i be the machine with the heaviest load T reported by GREEDYMAKESPANALGO2 algorithm;
- Divide T into two parts: the last job k, and the previous jobs. Thus $T = t_k + A$, where A denotes the total load of the previous jobs.
- $T \leq 1.5OPT$ since:
  1. $t_k \leq \frac{1}{2}OPT$ (by observation 3)
  2. $A \leq OPT$ (the same argument to the last theorem)

$\square$

The key steps in approximation algorithm design

A dilemma

- We are facing a dilemma:
    1. In order to establish the approximation guarantee, we should compare the quality of a solution against the quality of the optimal solution OPT.
    2. However, it is NP-Hard to compute the optimal solution value OPT and to find an optimal solution.

- Question: How can we establish the connection with OPT **without** the exact value of OPT?

- The answer to this question provides a key step in the design of approximation algorithms.

- **Strategy:** Finding a lower bound of OPT and **comparing a solution with the lower bound** rather than **comparing with OPT directly!**
- **Key step 1:** Although it is NP-Hard to find optimal solution, it might be polynomial-time computable to find a **"tight lower bound"** of OPT.
- **Key step 2:** We should figure out a process to yield a feasible solution, and compare this solution with the lower bound of OPT.

- Then how to find a lower bound of OPT?
  1. Combinatorial ways: problem specific and thus cohesive; then we design greedy or DP algorithm;
  2. LP-based method as a united way: LP-relaxation, duality, etc.

Another example: SET COVER problem

- Practical problems:
  - An anti-virus package identifies a viruses based on characteristic "keywords" set, and a keyword corresponds to several viruses. The question is how to select a small "representative" keywords set to detect all viruses.
  - To form a committee, containing as few people as possible, to cover all requisite skills;

Formalized Definition:

INPUT: a set of n elements $U = \{1, 2, ..., n\}$, and m subsets of U, denoted as $S_1, S_2, ..., S_m$. Each subset S_i has a weight w_i.

OUTPUT: to find a collection of subsets, with the minimal weight sum, such that all elements of U are covered.

Note: SET COVER problem plays an important role in approximation algorithm design as MATCHING problem in the design of exact algorithm.
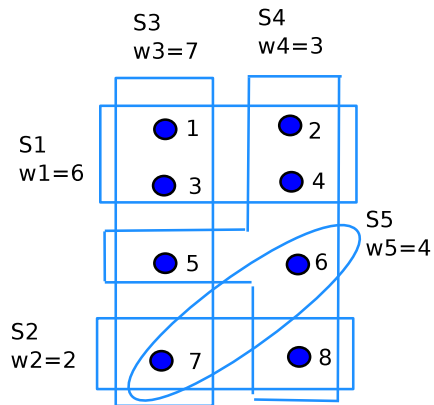
- Question: how to choose several subsets, with the minimal weight sum, such that all elements are covered?

- **Key observation:** solution is a collection of subsets. Imagine the solving process as a series of decisions. At each decision step, we decide to choose a subset or abandon it.
- **Greedy selection rule:** we should consider two aspects of a subset:
  1. the smaller the weight, the better;
  2. the more items it covers, the better.

  Thus, it is reasonable to select a subset based on **"the ratio of weight over the number of items it covers"**.

Greedy-Set-Cover algorithm

1: $I = \texttt{NULL}$; $//I$ denotes the index of selected subsets;
2: $R = U$; $// R$ denotes the remaining elements;
3: **while** $R \neq \texttt{NULL}$ **do**
4:     $j = \arg\min_i \frac{w_i}{|S_i \cap R|}$;
5:     Let $p_j = \frac{w_j}{|S_j \cap R|}$;
6:     **for all** remaining element $e \in R \cap S_j$ **do**
7:         Set $price(e) = p_j$;
8:     **end for**
9:     $I = I \cup \{j\}$;
10:     $R = R - S_j$;
11: **end while**
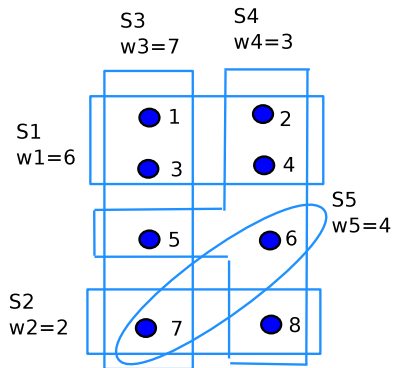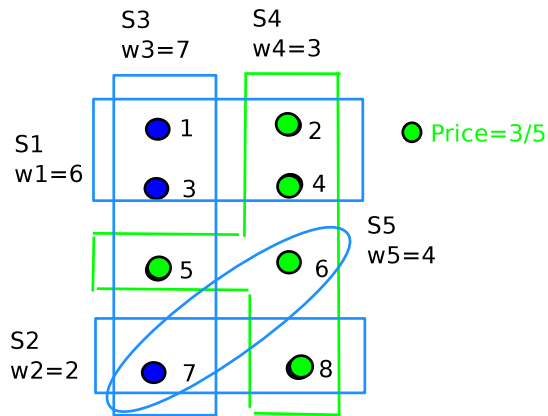
Step 1:

- $R = U = \{1, 2, 3, 4, 5, 6, 7, 8\}$;
- $p_1 = \frac{w_1}{|S_1 \cap R|} = \frac{6}{4}$; $p_2 = \frac{w_2}{|S_2 \cap R|} = \frac{2}{2}$; $p_3 = \frac{w_3}{|S_3 \cap R|} = \frac{7}{4}$;
  $p_4 = \frac{w_4}{|S_4 \cap R|} = \frac{3}{5}$; $p_5 = \frac{w_5}{|S_5 \cap R|} = \frac{4}{2}$;
- Choose S_4 since p_4 is the smallest one.

S3
w3=7

S4
w4=3

● 1   ● 2   ● Price=3/5

S1
w1=6

● 3   ● 4

S5
w5=4

● 5   ● 6

S2
w2=2

● 7   ● 8

Step 2:

- $R = \{1, 3, 7\}$;
- $p_1 = \frac{w_1}{|S_1 \cap R|} = \frac{6}{2}$; $\ p_2 = \frac{w_2}{|S_2 \cap R|} = \frac{2}{1}$; $\ p_3 = \frac{w_3}{|S_3 \cap R|} = \frac{7}{3}$; $p_5 = \frac{w_5}{|S_5 \cap R|} = \frac{4}{1}$;
- Choose S_2 since p_2 is the smallest one.

Step 3:

- $R = \{1, 3\}$;
- $p_1 = \frac{w_1}{|S_1 \cap R|} = \frac{6}{2}$; $p_3 = \frac{w_3}{|S_3 \cap R|} = \frac{7}{2}$;
- Choose S_1 since p_1 is the smallest one.
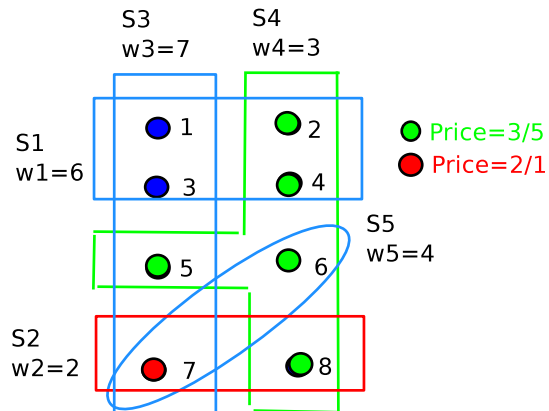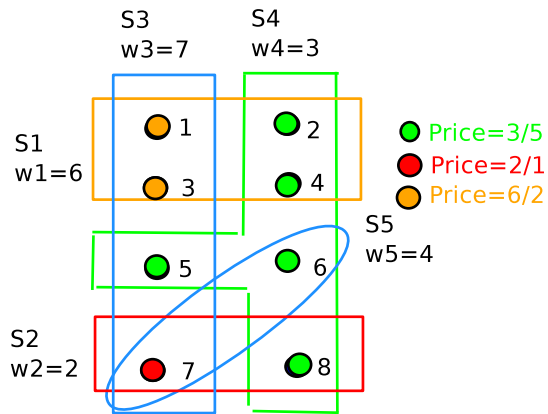
Step 4:

- $R = \{\}$. Done!
- Solution: We select $I = \{S_1, S_2, S_4\}$ with the sum of weight: 11.
- Optimal solution: selecting $\{S_3, S_4\}$ with the sum of weight: 10

- We will prove the following theorem:

Theorem

GREEDY-SET-COVER *algorithm is an $H(f)$-approximation algorithm, where $f = \max_i |S_i|$.*

- Meaning: The algorithm returns a subset $I = \{S_1, S_2, S_4\}$ with the total weight of 11. We guarantee that $W = 11 \leq H(5)OPT = H(5)10$.

Proof.

Let S^* be the optimal solution, and S be the solution returned by GREEDY-SET-COVER algorithm. We have:

$$
\begin{aligned}
\sum_{S_i \in S} w_i &= \sum_{e \in U} price(e) \quad \text{(by Line 7-9)} & (1) \\
&\leq \sum_{S_j \in S^*} \sum_{e \in S_j} price(e) \quad \text{(by S* covers U)} & (2) \\
&\leq \sum_{S_j \in S^*} H(|S_j|) w_j \quad \text{(by the lemma)} & (3) \\
&\leq \sum_{S_j \in S^*} H(f) w_j \quad (d \text{ is the largest}) & (4) \\
&= H(f)OPT & (5)
\end{aligned}
$$

$\square$

Finding a lower bound

Lemma

For each subset S_i, $\sum_{e \in S_i} price(e) \leq H(|S_i|)w_i$.

- Take S_1 as an example. We have:

$$
\begin{aligned}
\Sigma_{e \in S_1} price(e) &= (\frac{w_4}{5} + \frac{w_4}{5}) + (\frac{w_1}{2} + \frac{w_1}{2}) & (6) \\
&\leq (\frac{w_1}{4} + \frac{w_1}{4}) + (\frac{w_1}{2} + \frac{w_1}{2}) & (7) \\
&\leq (\frac{w_1}{4} + \frac{w_1}{3}) + (\frac{w_1}{2} + \frac{w_1}{1}) & (8) \\
&= w_1 H(4)
\end{aligned}
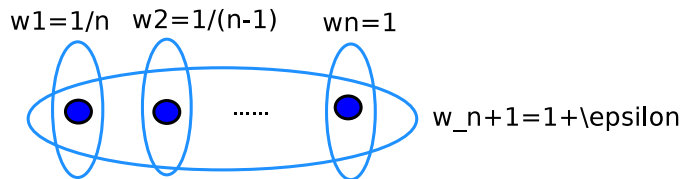$$

- Here, the first $\leq$ is due to the selecting criteria in Step 2, i.e. $p_4 = \frac{w_4}{5}$ is smaller than $p_1 = \frac{w_1}{4}$, and the selecting criteria in Step 3, i.e. $p_1 = \frac{w_1}{2}$ is smaller than $p_3 = \frac{w_3}{2}$.

Question: is this an accurate analysis?

A tight example

- Consider a special instance: $n + 1$ subsets with weights $\frac{1}{n}, \frac{1}{n-1}, ..., \frac{1}{2}, 1$ and $1 + \epsilon$, respectively.

- Optimal solution: selecting only one subset with weight $1 + \epsilon$;
- GREEDY-SET-COVER result: selecting n subsets with the sum of weight $\frac{1}{n} + \frac{1}{n-1} + ... + \frac{1}{2} + 1 = H(n)$.

LP as a unified approach framework to lower bound OPT

Lower bound 1: $z_{LP} \leq z_{ILP} = OPT$

- **Step 1:** Formulate the problem as an integer linear program (ILP);
- We define a 0/1 variable x_j for each subset S_j, where $x_j = 1$ means the selection of S_j, and 0 means abandon.
- ILP model:

$$
\begin{aligned}
\min \quad z = \quad & \sum_{j=1}^{m} w_j x_j \\
s.t. \quad & \sum_{j:e \in s_j} x_j \geq 1 \quad \forall e \in U \\
& x_j = 0/1 \quad \forall i
\end{aligned}
$$

- Thus we have $OPT = z_{ILP}$, where z_{ILP} denotes the optimal objective value of the ILP model.

- **Step 2:** Relaxing ILP into LP;
- LP relaxation:

$$
\begin{array}{llll}
\min & z = & \sum_{j=1}^{m} w_j x_j & \\
s.t. & & \sum_{j:e \in s_j} x_j & \geq 1 \quad \forall e \in U \\
& & x_j & \geq 0 \quad \forall i \\
& & x_j & \leq 1 \quad \forall i
\end{array}
$$

- Key observation 1: $z_{LP} \leq z_{ILP} = OPT$, where z_{LP} denotes the optimal objective value of the LP model.

- **Step 3:** Since the LP model might generate a fractional solution, a clever way should be figured out to transform the optimal solution to LP (in polynomial-time) to a feasible solution to ILP, whose objective value is close to the optimal LP objective value.

- The final difficulty: how to obtain a feasible solution to ILP based on the optimal solution to LP? **Rounding!**
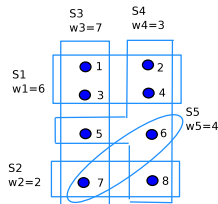
LP+Rounding algorithm

1: Solve the linear program LP to get an optimal solution x^*;
2: $I = \texttt{NULL}$;
3: **for all** subset S_j **do**
4:     **if** $x_j^* \geq \frac{1}{f}$ **then**
5:       $I = I + \{j\}$;
6:     **end if**
7: **end for**
8: **return** I;

- Here d denotes the largest coverage of any element in U, i.e. $d = \max_{e \in U} |\{j : e \in S_j\}|$.

$$
\begin{array}{rllllllllll}
\min & 6x_1 & + & 2x_2 & + & 7x_3 & + & 3x_4 & + & 4x_5 & & \\
s.t. & x_1 & & & + & x_3 & & & & & \geq 1 & \text{(item 1,3)} \\
& x_1 & & & & & + & x_4 & & & \geq 1 & \text{(item 2,4)} \\
& & & & & x_3 & + & x_4 & & & \geq 1 & \text{(item 5)} \\
& & & x_2 & & & & & + & x_5 & \geq 1 & \text{(item 6)} \\
& & & x_2 & + & x_3 & & & + & x_5 & \geq 1 & \text{(item 7)} \\
& & & x_2 & & & + & x_4 & & & \geq 1 & \text{(item 8)} \\
& & & & & & & & x_i & & \geq 0 & \\
& & & & & & & & x_i & & \leq 1 & \\
\end{array}
$$

- LP optimal solution: $x_1 = 0.5$; $x_2 = 1$; $x_3 = 0.5$; $x_4 = 0.5$; $x_5 = 0$;
- Objective value of LP: 10. ($z_{LP} \leq z_{ILP} = OPT$.)
- Rounding solution: $x_1' = 1$; $x_2' = 1$; $x_3' = 1$; $x_4' = 1$; $x_5' = 0$;
- Objective value: $18 \leq d \times z_{LP}$.

Theorem

Algorithm1 yields a set cover.

Proof.

(by contradiction)

- Suppose there is an element e not covered, i.e. $e \notin \cup_{j \in I} S_j$.
- Then for each S_j that contains e as a member, we have $x_j^* < \frac{1}{f}$; (Otherwise, S_j should be selected by Algorithm1.)
- Thus, $\sum_{j:e \in S_j} x_j^* < \frac{1}{d} |\{j : i \in S_j\}| \leq 1$.
- A contradiction against the linear program constraints.

$\square$

Theorem

(Hochbaum '82) Algorithm1 is an f-approximation algorithm for SETCOVER.

Proof.

- Algorithm1 returns a collection I with cost: $C = \sum_{j \in I} w_j$;

$$
\begin{aligned}
C &= \sum_{j \in I} w_j \\
&\leq \sum_{j \in I} w_j x_j^* d \\
&\leq \sum_{j=1}^{m} w_j x_j^* d \\
&= d \sum_{j=1}^{m} w_j x_j^* \\
&= d z_{LP} \\
&\leq f OPT
\end{aligned}
$$

- Here, the first inequality follows due to the "rounding criteria", i.e. $x_j^* \geq \frac{1}{d}$.
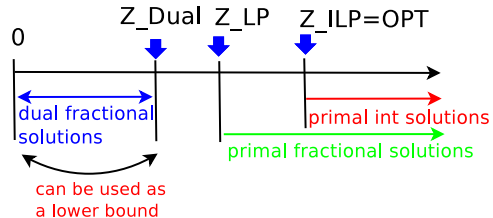
Lower bound 2: $z_{Dual} \leq z_{LP} \leq z_{ILP} = OPT$

- Basic idea: rounding the dual solution.
- Dual:

$$\begin{array}{rrcl} \max d = & \sum_{e \in U} y_e \\ s.t. & \sum_{e:e \in S_j} y_e & \leq & w_j & \forall S_j \\ & y_e & \geq & 0 & \forall e \in U \end{array}$$

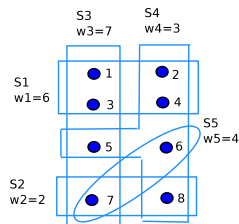**Key observation 2:** $\sum_e y_e \le z_{LP} \le OPT$ for any feasible dual solution y;

Dual LP + Rounding Algorithm

1: Solve the dual LP to get an optimal solution y^*;
2: $I = \texttt{NULL}$;
3: **for all** subset S_j **do**
4:   **if** $\sum_{e:e \in S_j} y_e^* = w_j$ **then**
5:     $I = I + \{j\}$;
6:   **end if**
7: **end for**
8: **return** I;

An example

$$
\begin{array}{llllllllll}
\max & y_1 & + & y_2 & + & y_3 & + & y_4 & + & y_5 & + & y_6 & + & y_7 & + & y_8 \\
s.t. & y_1 & + & y_2 & + & y_3 & + & y_4 & & & & & & & & & \leq 6 \\
& & & & & & & & & & & & & y_7 & + & y_8 & \leq 2 \\
& y_1 & & & + & y_3 & & & + & y_5 & & & + & y_7 & & & \leq 7 \\
& & & y_2 & & & + & y_4 & + & y_5 & + & y_6 & & & + & y_8 & \leq 3 \\
& & & & & & & & & & & y_6 & + & y_7 & & & \leq 4 \\
& & & & & & & & & & & & & & & y_i & \geq 0
\end{array}
$$

- Dual LP optimal solution: $y_1 = 6$; $y_2 = 0$; $y_3 = 0$; $y_4 = 0$; $y_5 = 0$; $y_6 = 3$; $y_7 = 1$; $y_8 = 0$;
- Objective value of Dual: 10 ($z_{DualLP} \leq z_{LP} \leq z_{ILP} = OPT$.)
- Tight constraints: $I = \{1, 3, 4, 5\}$;
- Objective value: $20 \leq d * z_{DualLP}$.

Correctness of Algorithm2

Theorem

Algorithm2 yields a set cover.

Proof.

(by contradiction again)

- Suppose there is an element $\hat{e}$ that is not covered by the selected subsets in I;
- Then for ALL S_j containing $\hat{e}$, we have $\sum_{e:e\in S_j} y_e^* < w_j$; (Otherwise, a S_j should be selected and thus e will be covered.)
- Thus, increasing $y_{\hat{e}}^*$ (by a small positive amount) will increase the objective function value of dual LP without violating the constraints. A contradiction with the optimality of y^*.

$\square$

Theorem

Algorithm2 is a d-approximation algorithm.

Proof.

Let C denote the sum weight of the subsets that Algorithm2 returns, i.e. $C = \sum_{j \in I} w_j$. We have:

$$
\begin{aligned}
C &= \sum_{j \in I} w_j \\
&= \sum_{j \in I} \sum_{e:e \in S_j} y_e^* \\
&\leq d \sum_{e:e \in U} y_e^* \quad \text{(since any e was included at most d times)} \\
&= d z_{Dual\ LP} \\
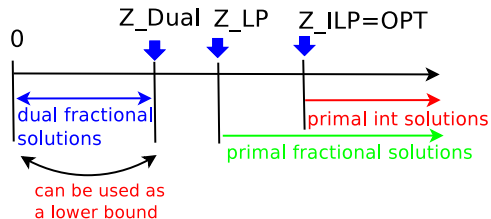&\leq d OPT \quad \text{(since $z_{Dual\ LP} \leq z_{LP} \leq z_{ILP}$)}
\end{aligned}
$$

$\square$

Lower bound 3:
value of any dual feasible solution $\leq z_{Dual} \leq z_{LP} \leq z_{ILP} = OPT$

- Basic idea: a feasible dual solution is enough for lower bounding OPT. Thus, we can employ primal_dual strategy to construct a feasible dual solution rather than solving Dual LP to get an optimal dual solution.
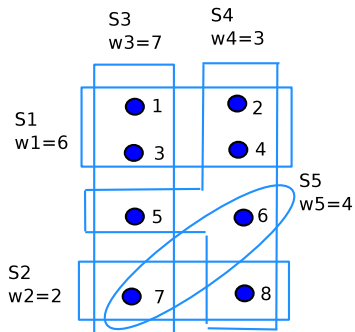
Primal_Dual + Rounding Algorithm3

1: $I = \mathtt{NULL}$;
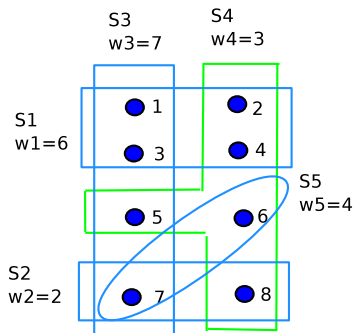2: $y_e = 0$ for all $e \in U$;
3: **while** exists an element $\hat{e}$ not covered **do**
4:     **for all** subset S_j containing $\hat{e}$ **do**
5:         $g_j = w_j - \sum_{e:e \in S_j} y_e$; // calculate the gap of constraint j;
6:     **end for**
7:     $i = \arg\min_j g_j$;
8:     $y_{\hat{e}} = y_{\hat{e}} + g_i$;
9:     $I = I \cup \{i\}$;
10: **end while**
11: **return** I;

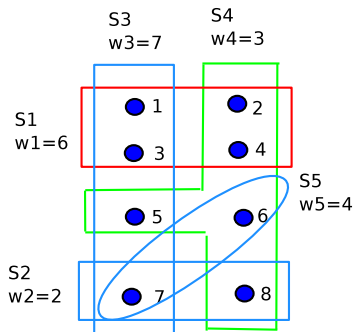- Initially, we have $y_i = 0$ for all $1 \le i \le 8$.
- Consider element y_2. There are two constraints S_1 and S_4.
- We increase y_2 to 3 (selecting S_4).

- Consider element y_1. There are two constraints S_1 and S_3.
- We increase y_1 to 3 (selecting S_1).

- Consider element y_7. There are three constraints S_2, S_3 and S_5.
- We increase y_7 to 2 (selecting S_2).
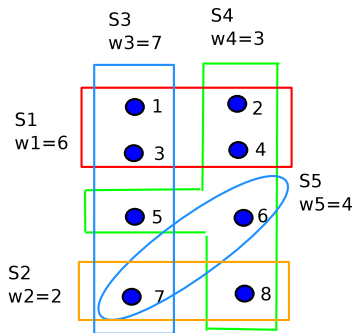
- All elements have been covered. Done!
- Dual solution:
  $y_1 = 3; y_2 = 3; y_7 = 2; y_3 = y_4 = y_5 = y_6 = y_8 = 0;$.
- Dual objective value: 8. (Dual feasible solution,
  $z \leq z_{Dual} \leq z_{LP} \leq z_{ILP} = OPT$)
- Tight constraints: $I = \{S_4, S_1, S_2\}$
- Objective value: $11 \leq d \times z \leq d \times z_{Dual} \leq d \times OPT$

Correctness

> **Lemma**
>
> Algorithm3 yields a dual feasible solution y, and $\sum_{e:e \in S_j} y_e = w_j$ for any $j \in I$.

> **Proof.**
>
> - Base case: $y = 0$ is dual feasible since $\sum_{e:e \in S_j} y_e = 0 \leq w_j$;
> - Induction: assuming that y is dual feasible before an iteration of while loop, i.e. $\sum_{e:e \in S_j} y_e \leq w_j$. Suppose we increase $y_{\hat{e}}$ by g_i to generate a new solution y'. We will show that y' is also a dual feasible solution.
>
> $$
> \begin{aligned}
> \sum_{e:e \in S_j} \hat{y'_e} &= \sum_{e:e \in S_j} y_e + g_i \quad (g_i \text{ is minimal}) \\
> &= \sum_{e:e \in S_j} y_e + (w_i - \sum_{e:e \in S_i} y_e) \\
> &\leq \sum_{e:e \in S_j} y_e + (w_j - \sum_{e:e \in S_j} y_e) \\
> &= w_j
> \end{aligned}
> $$

Performance analysis

Theorem

(Bar-Yehuda, Even '81) Algorithm3 is a d-approximation algorithm.

Proof.

Let C denote the sum weight of subsets generated by Algorithm2, i.e. $C = \sum_{j \in I} w_j$. We have:

$$
\begin{aligned}
C &= \sum_{j \in I} w_j \\
&= \sum_{j \in I} \sum_{e : e \in S_j} y_e \quad (\text{ by lines 9-10}) \\
&\leq d \sum_{e : e \in U} y_e \quad (\text{ e is covered at most f times}) \\
&\leq d z_{Dual} \quad (y \text{ is a dual feasible solution }) \\
&\leq d OPT \quad (\text{since y is only a dual feasible solution})
\end{aligned}
$$

□

Another example of rounding: the generalization of MAKESPAN
problem

Practical problem: We have multiple servers to process a set of jobs. However, some machines cannot be assigned to a job. How to schedule jobs to machines as "balanced" as possible?
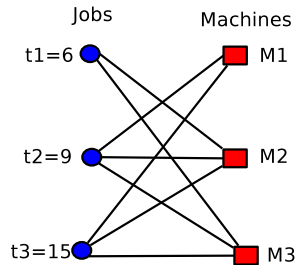
INPUT:
m servers $M_1, M_2, ..., M_m$, n jobs $J = \{1, .., n\}$ (each job j has a processing time t_j). Each job j has a subset of machines $S_j \subset \{M_1, ..., M_m\}$ that it can use.

OUTPUT:
An assignment of jobs to machines to minimize the *makespan* , i.e. the maximum load on any machine, $T = \max_i \sum_{j \in A(i)} t_j$, where $A(i)$ denotes the jobs assigned to machine i;

(ILP)

$$
\begin{array}{rlcl}
\min & L & & \\
s.t. & \sum_{i=1}^{m} x_{ji} & = & t_j \qquad \text{for all } j \in J \\
& \sum_{j=1}^{n} x_{ji} & \leq & L \qquad \text{for all } i \\
& \textcolor{red}{x_{ji}} & \textcolor{red}{=} & \textcolor{red}{0/t_j} \quad \text{for all } j \in J, i \in S_j \\
& x_{ji} & = & 0 \quad \text{for all } j \in J, i \notin S_j
\end{array}
$$

(Intuition: x_{ji} indicates how much of the load of t_j is assigned to machine M_i;)

(LP)

$$
\begin{array}{rlll}
\min & L & & \\
s.t. & \sum_{i=1}^{m} x_{ji} & = & t_j \qquad \text{for all } j \in J \\
& \sum_{j=1}^{n} x_{ji} & \leq & L \qquad \text{for all } i \\
& {\color{red}x_{ji}} & {\color{red}\leq} & {\color{red}t_j} \quad \text{for all } j \in J, i \in S_j \\
& {\color{red}x_{ji}} & {\color{red}\geq} & {\color{red}0} \quad \text{for all } j \in J, i \in S_j \\
& x_{ji} & = & 0 \quad \text{for all } j \in J, i \notin S_j
\end{array}
$$

- **Key observation 1:** $z_{LP} \leq z_{ILP} = OPT$
- **Key observation 2:** $\max_j t_j \leq OPT$

(LP)
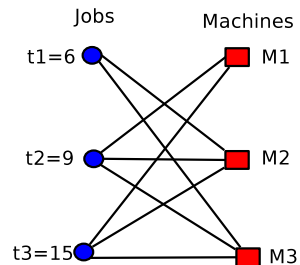
$$
\begin{aligned}
\min z = &\quad L \\
s.t. \quad &\qquad\quad x_{12} \;+x_{13} \;= \;\;6 \\
&x_{21} \;+x_{22} \;+x_{23} \;= \;\;9 \\
&x_{31} \;+x_{32} \;+x_{33} \;= \;15 \\
&\qquad\quad x_{21} \;+x_{31} \;\leq \;\;L \\
&x_{12} \;+x_{22} \;+x_{32} \;\leq \;\;L \\
&x_{13} \;+x_{23} \;+x_{33} \;\leq \;\;L \\
&\qquad\qquad\qquad\;\; x_{ji} \;\geq \;\;0
\end{aligned}
$$

Jobs Machines

t1=6 ● 6
        0
        5
        4
t2=9 ● 0
        5
        0
t3=15 ● 10

M1 ■
M2 ■
M3 ■

- Suppose we have already obtained a solution to the LP. The final question is: how to construct a valid schedule from the LP solution?
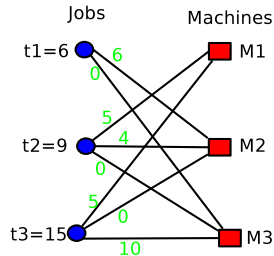
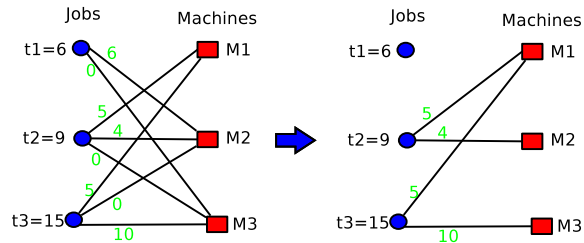- Recall that for any job j, we have $t_j = x_{j1} + x_{j2} + ... + x_{jm}$. There are two possible cases:
  1. $\exists i, x_{ji} = t_j$ (called "integral job"): Intuitively, job j was scheduled to machine M_i as a whole, e.g. job 1.
  2. $\forall i, x_{ji} < t_j$ (called "fractional job"): Intuitively, job j was decomposed into parts, which were distributed to different machines, e.g., two parts of job 2 was distributed to M_1 and M_2.
- The difficulty is how to deal with fractional jobs.

- Let's focus on the fractional jobs via:
  - Removing the "integer jobs";
  - Removing the **unused assignments**, i.e. $x_{ji} = 0$;
- There are two possibilities:
  1. The assignment contains no cycle;
  2. The assignment contains one or more cycles;

Case 1: the assignment contains no cycle

- In this case, a valid schedule can be constructed via the following rounding strategy:
  1. Rounding: If a part of a fractional job j was scheduled to machine M_i, we can simply schedule the whole job j to M_i.
  2. Restriction: Each machine receives **at most** one fractional job.
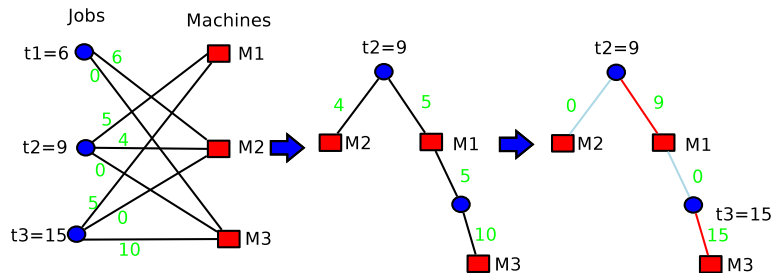
- The rounding operation is feasible since:
  1. "No cycle" means a tree rooted at a job.
  2. Notice that any leaf should be a "machine" rather than a "job" (Reason: a "leaf job" is connected with its parent only; thus it should be an "integer job")
  3. So we can simply assign a fractional job **completely** to an **arbitrary** child, say assign J_2 to M_1, and assign J_3 to M_2.
- Time complexity: $O(mn)$.

- Combining the "integer jobs", we will have the following schedule:

- $J_{M1} = \{2\}; J_{M2} = \{1\}; J_{M3} = \{3\}$.
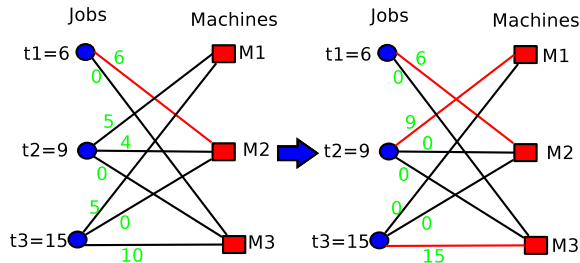
- MakeSpan$= 15 \leq 2 \times z_{LP} = 20$

- We can show that the rounding strategy won't introduce too many load to any machine.

Theorem

$MakeSpanApprox$ is a 2-approximation algorithm.

Proof.

- Let T denotes the makespan that $MakeSpanApprox$ yields, and T is obtained from machine M_i.
- Let J_i denote the jobs assigned to M_i. J_i consists of two parts: integral jobs assigned to machine i $(x_{ij} = t_j)$, and at most **one** fractional job (denoted as f_i) that is assigned to multiple machines $(x_{ij} < t_j)$. We have:

$$
\begin{align}
T &= \sum_{j \in J_i} t_j \tag{9} \\
&= \sum_{j \in J_i, j \neq f_i} t_j + t_{f_i} \tag{10} \\
&= \sum_{j \in J_i, j \neq f_i} x_{ij} + t_{f_i} \quad \text{(by the definition of integral jobs)} \tag{11} \\
&\leq \sum_{j \in J_i} x_{ij} + t_{f_i} \tag{12} \\
&\leq z_{LP} + t_{f_i} \tag{13} \\
&\leq z_{LP} + OPT \quad \text{(by key observation 2)} \tag{14} \\
&\leq 2OPT \quad \text{(by key observation 1)} \tag{15}
\end{align}
$$

Case 2: the assignment contains cycles

- In fact, any cycle can be eliminated without load changes for any machine using the following **"augmentation"** operation:
  1. Consider a cycle C. We first find the smallest load (denoted as δ);
  2. Increasing/decreasing loads by δ on edges in the cycle;
  3. Thus the cycle will be eliminated without no influence on the loads. For example, edge $J_1 - M_3$ is cut to break the cycle.
- Time complexity: $O(|C|)$ for cycle C. The operation will be repeated at most $O(mn)$ times to remove all cycles.
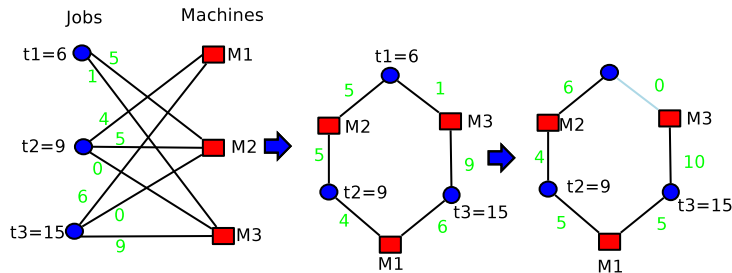
MAKESPANAPPROX Algorithm

1: Solving the LP model to get solution x_{ij};
2: Removing cycles in the schedule graph via **augmentation**;
3: **for** $i = 1$ to m **do**
4:     assigning integral job j to i if $x_{ij} = t_j$;
5:     assigning at most **one** fractional job to i;
6: **end for**

Other useful technique: scaling and parametric pruning

- Scaling: rounding a real number to an integer; grid;
- Parametric pruning: Suppose we have already known OPT, we might be able to prune away irrelevant parts of the input and thereby simplify the search for a good solution.

Scaling technique: from pseudo polynomial time algorithm to PTAS

Given a set of items, each item has a weight and a value, to determine a set of items such that the total weight is less than a given limit and the total value is as large as possible.

Formalized Definition:

- **Input:**
  a set of items. Item i has weight w_i and value v_i, and a total weight limit W;

- **Output:**
  the set of items which maximize the total value with total weight below W

What's the best solution?
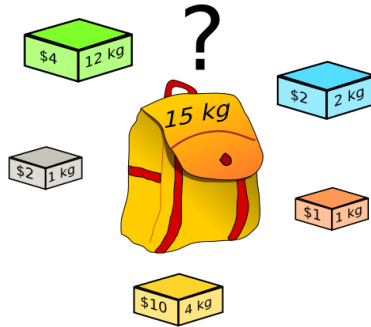
Greedy solution:
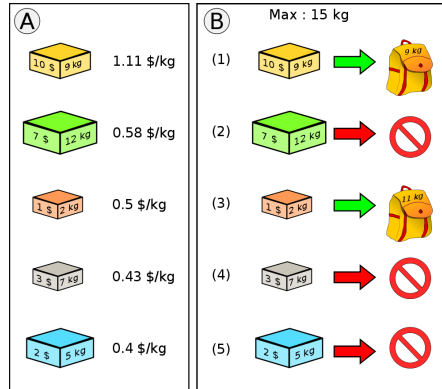
Note: there are two types of dynamic programming algorithms to solve KNAPSACK problem.

- Imagine the solving process as a series of decisions.
- Suppose we have already obtained the optimal solution S. Let's consider the first decision step, we have two options: select item n, or abandon it.
- Thus the general form of sub-problems can be set as: to calculate the maximum value of any solution using a subset of the items $\{1, 2, ..., i\}$ and a bag of size w, denoted as $OPT(i, w)$.
- Our objective: $OPT(n, W)$.
- Optimal sub-structure: $OPT(n, W) = max\{OPT(n-1, W), OPT(n-1, W - w_n) + v_n\}$;
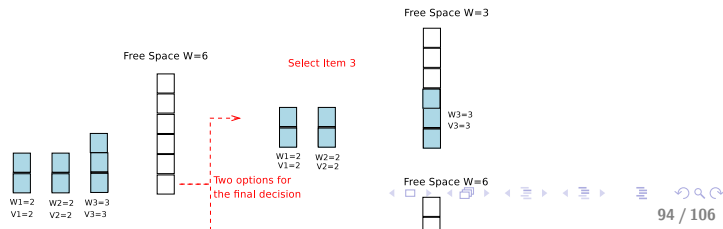
Knapsack DP1

1: **for** $w = 1$ to W **do**
2:     $M[0, w] = 0$;
3: **end for**
4: **for** $i = 1$ to n **do**
5:     **for** $w = 1$ to W **do**
6:         $M[i, w] = max\{M[i - 1, w], w_i + M[i - 1, w - w_i]\}$;
7:     **end for**
8: **end for**
9: **return** $M[n, W]$;

Time complexity: $O(nW)$. Pseudo polynomial time.

INPUT: a set of n items. An item i has weight w_i and value v_i. The value requirement V;

OUTPUT: to select a subset of items to minimise the total weight with value at least V;

Note: if this problem was solved, the Knapsack problem can be solved by by **finding the largest value V such that** $OPT(n, V) \leq W$.

- Imagine the solving process as a series of decisions.
- Suppose we have already obtained the optimal solution S. Let's consider the first decision step, there are two possibilities: select item n or abandon it;
- The general form of sub-problems: to calculate the smallest bag size, i.e. the bag can hold a subset of items $\{1, 2, ..., i\}$ with value at least V (denoted as $OPT(i, V)$);
- Our objective: the largest V such that $OPT(n, V) \leq W$. (Notice that $V \leq nv^*$, where $v^* = max_i\{v_i\}$);
- Optimal sub-structure:
$$OPT(n, V) = \min \begin{cases} OPT(n-1, V) & \text{abandon item n} \\ w_n & \text{select item n only} \\ w_n + OPT(n-1, V - v_n) & \text{otherwise} \end{cases}$$

Knapsack DP2

1: **for** $i = 0$ to n **do**
2:     $M[i, 0] = 0$;
3: **end for**
4: **for** $i = 1$ to n **do**
5:     **for** $V = 1$ to $\sum_{k=1}^{i} v_k$ **do**
6:         **if** $V > \sum_{k=1}^{i-1} v_k$ **then**
7:             $M[i, V] = w_i + M[i - 1, V - v_i]$;
8:         **else**
9:             $M[i, V] = min\{M[i - 1, V], w_i, w_i + M[n - 1, v - v_i]\}$;
10:         **end if**
11:     **end for**
12: **end for**

Time complexity: $O(n^2 v^*)$. Still pseudo-polynomial time.

- Basic idea: the algorithm is good when v^* is small. But how to deal with the case when v^* is large? Scaling!
- More specifically, a large v_i can be scaled to a smaller $\hat{v_i} = \lceil \frac{v_i}{b} \rceil$. We denote $\overline{v_i} = \hat{v_i}b$;

Rounding
b = 1000   Equivalent

V1	3278		$\overline{V1}$	4000	$\hat{V1}$	4
V2	1956		$\overline{V2}$	2000	$\hat{V2}$	2
V3	4123		$\overline{V3}$	5000	$\hat{V3}$	5
V4	2233		V4	3000	V4	3

Knapsack-Approx(ϵ)

1: Let $v^* = \max_i v_i$;
2: Let $b = \frac{\epsilon}{n} v^*$;
3: Calculate $\hat{v}_i = \lceil \frac{v_i}{b} \rceil$ for each item i;
4: Run KnapsackDP2 on the items with value $\hat{v}_i$ and return the optimal solution S;

Knapsack-Approx algorithm runs in polynomial time.
In fact, the running time is $O(n^2 \hat{v^*}) = O(n^2 \frac{v^*}{b}) = O(\frac{n^3}{\epsilon})$.

Experimental results on an instance:

b	$\overline{v}$	ϵ	W	# OP	Time (ms)
1	2223975	0.001	1768	889590000	18352.128
3	741325	0.010	1768	98843333	5990.893
5	444800	0.028	1768	35584000	3649.624
10	222400	0.112	1768	8896000	1836.567
30	74125	1.011	1768	988333	620.822
50	44475	2.810	1768	355800	381.982
100	22250	11.236	1768	89000	183.707
300	7425	101.010	1768	9900	60.422
500	4450	280.899	1768	3560	38.340
1000	2225	1123.6	1768	890	17.943
3000	750	10000	1809	100	6.872
5000	450	27777.8	1809	36	4.059
10000	225	111111	1809	9	3.134

Performance analysis

Theorem

Let S be the solution yielded by Knapsack-Approx algorithm, and S^* be any feasible solution such that $\sum_{i \in S^*} w_i \leq W$. We will show that $\sum_{i \in S} v_i \geq \frac{1}{1+\epsilon} \sum_{i \in S^*} v_i$.

Proof.

$$
\begin{aligned}
\sum_{i \in S^*} v_i & \leq \sum_{i \in S^*} \overline{v_i} \quad \text{(by } v_i \leq \overline{v_i}) & (16) \\
& \leq \sum_{i \in S} \overline{v_i} \quad \text{(S is optimal solution)} & (17) \\
& \leq \sum_{i \in S} (v_i + b) \quad \text{(by } \overline{v_i} \leq v_i + b) & (18) \\
& \leq nb + \sum_{i \in S} v_i & (19) \\
& \leq (1 + \epsilon) \sum_{i \in S} v_i \quad \text{(since } nb \leq \epsilon \sum_{i \in S} v_i) & (20)
\end{aligned}
$$

$\square$

Note: b is set to $b = \frac{\epsilon}{n}v^*$ according to two sides of considerations:

1. Time-complexity: $O(n^2 \frac{v^*}{b})$ is polynomial in n and $\frac{1}{\epsilon}$.
2. Approximation ratio: $nb \leq \epsilon \sum_{i \in S} v_i$.

Parametric pruning

Parametric pruning

The algorithm consists of three steps:

1. Pruning: Suppose we have a guess of OPT, denoted as parameter t. For each given t, the input instance will be pruned by removing the parts that will not be used in any solution with cost $> t$. Denote the pruned instance as $I(t)$.

2. Lower bound: the family of $I(t)$ is used for computing a lower bound of OPT, say t^*;

3. Good solution: a solution is found in instance $I(\alpha t^*)$ for a suitable choice of α.

(See extra slides)

Appendix: PARTITION is NP-Complete

- PARTITION problem is to decide whether a given multiset of integers can be partitioned into two "halves" that have the same sum.
- More precisely, given a multiset S of integers, is there a way to partition S into two subsets S_1 and S_2 such that the sum of the numbers in S_1 equals the sum of the numbers in S_2?
- PARTITION problem can be easily proved to be NP-complete via a reduction from SUBSETSUM problem.