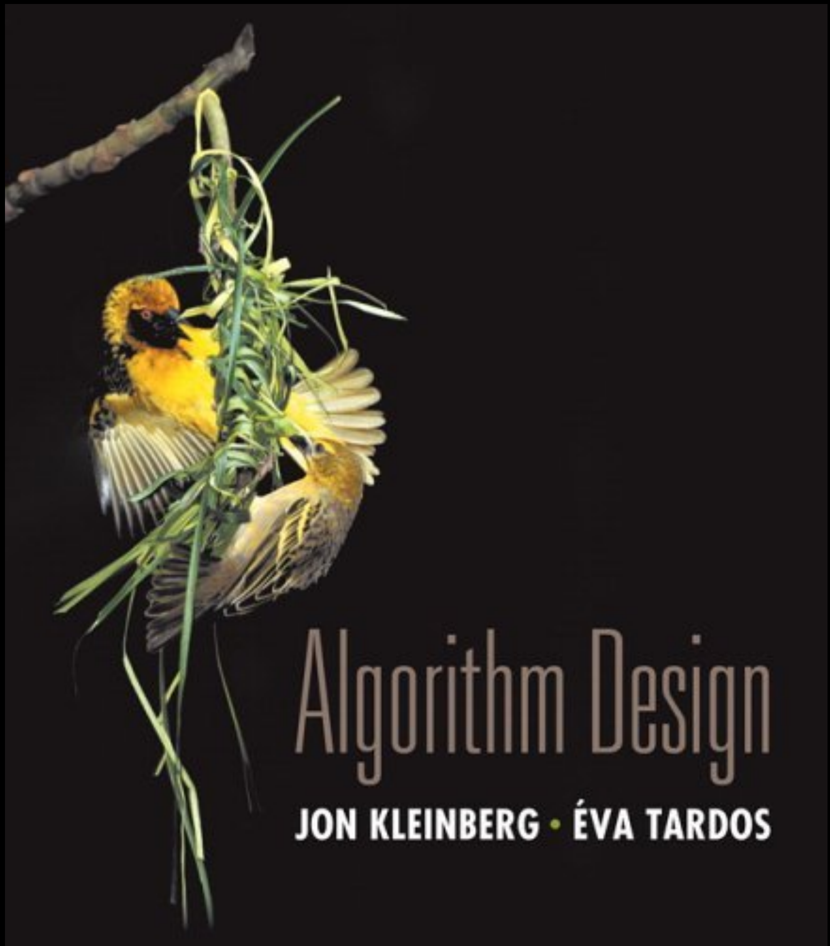



# Chapter 11

## Approximation Algorithms

Slides by Kevin Wayne.

# 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!

# 11.1 Load Balancing

# 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 $L_i = \Sigma_{j \in J(i)} t_j$.

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: List Scheduling

List-scheduling algorithm.

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

```
List-Scheduling(m, n, t1,t2,…,tn) {
   for i = 1 to m {
      Li ← 0          ← load on machine i
      J(i) ← φ        ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = argmink Lk           ← machine i has smallest load
      J(i) ← J(i) ∪ {j}        ← assign job j to machine i
      Li ← Li + tj             ← update load of machine i
   }
   return J(1), …, 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 $L^* \geq \frac{1}{m} \sum_j t_j$.

Pf.

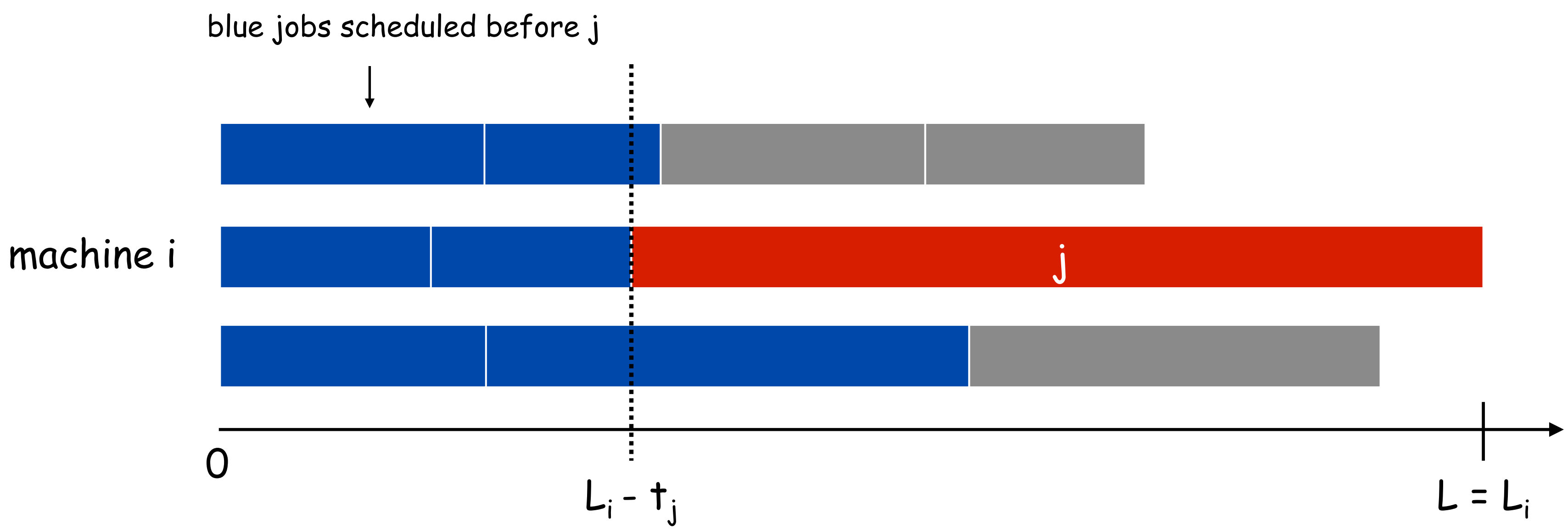
- The total processing time is $\Sigma_j t_j$.
- One of m machines must do at least a 1/m fraction of total work. ▪

# 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 $L_i - t_j \Rightarrow L_i - t_j \le L_k$ for all $1 \le k \le m$.



# 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 $L_i - t_j \Rightarrow L_i - t_j \le L_k$ for all $1 \le k \le m$.
- Sum inequalities over all k and divide by m:
- $m (L_i - t_j) \le \Sigma L_k$

$$\begin{aligned} L_i - t_j &\le \tfrac{1}{m} \sum_k L_k \\ &= \tfrac{1}{m} \sum_k t_k \\ \text{Lemma 1} \longrightarrow \quad &\le L^* \end{aligned}$$

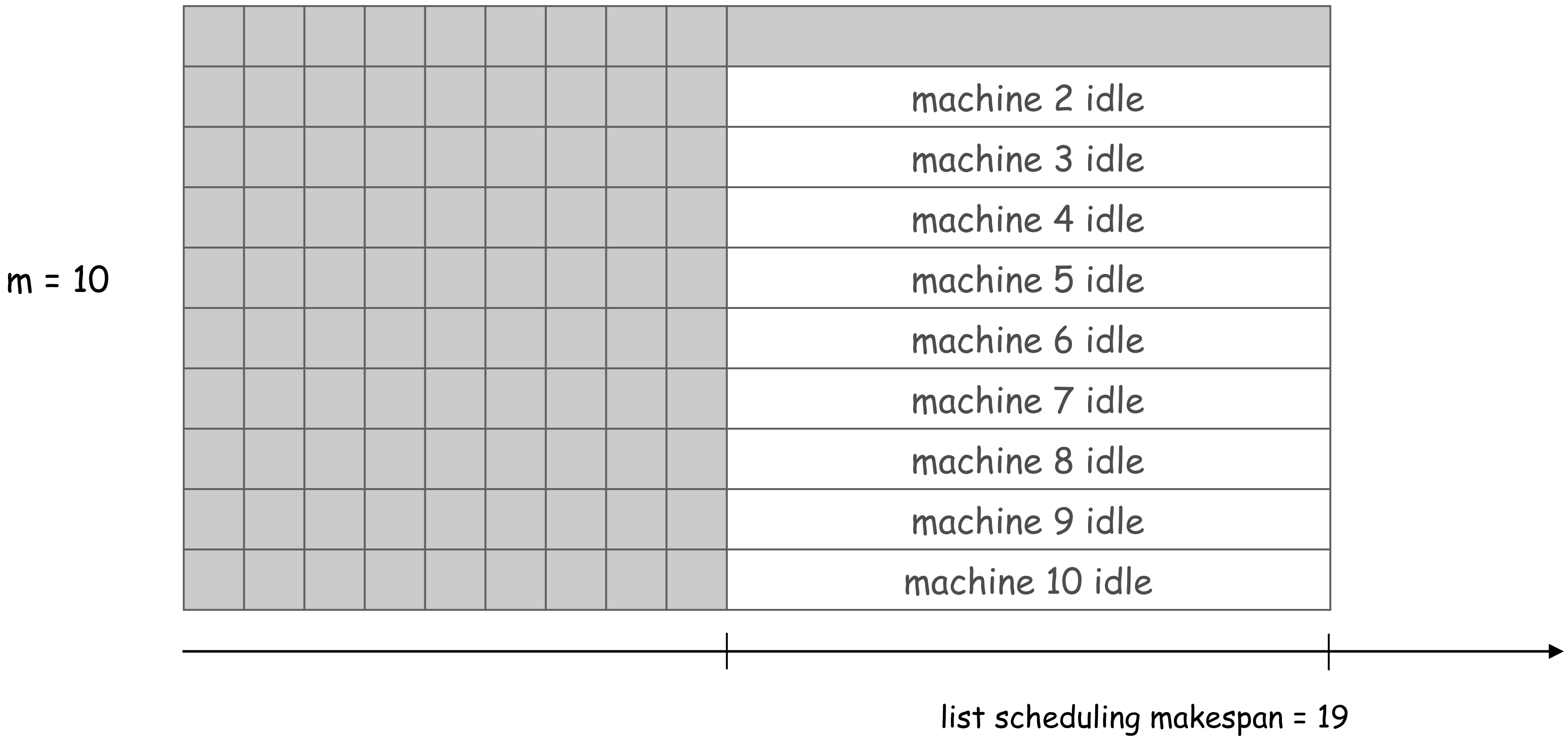
- Now $$L_i = \underbrace{(L_i - t_j)}_{\le L^*} + \underbrace{t_j}_{\le L^*} \le 2L^*. \quad \blacksquare$$

Lemma 2

# Load Balancing: List Scheduling Analysis

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

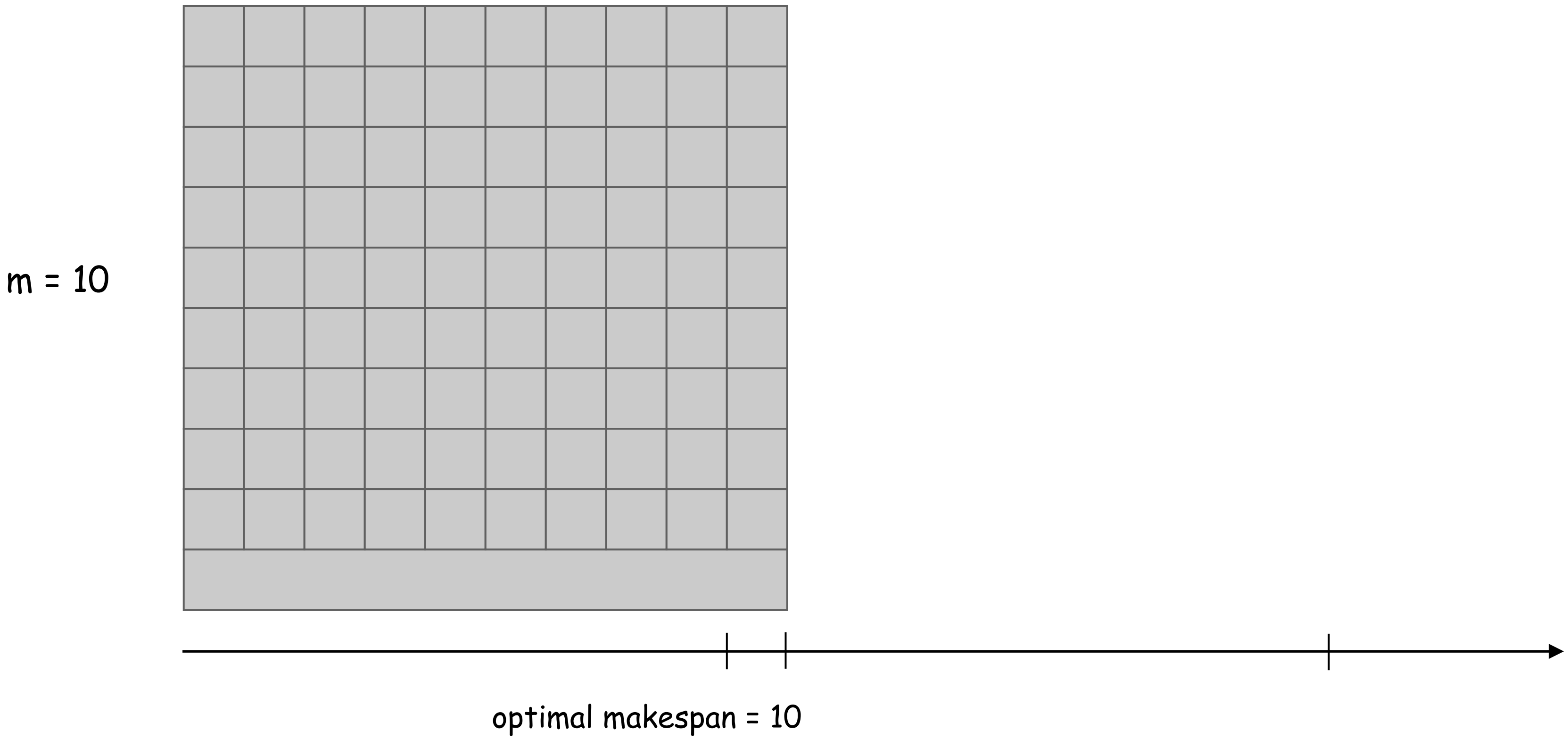
Ex: m machines, m(m-1) jobs length 1 jobs, one job of length m



# Load Balancing: List Scheduling Analysis

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

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



# 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 {
      Li ← 0             ← load on machine i
      J(i) ← φ           ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = argmin_k Lk          ← machine i has smallest load
      J(i) ← J(i) ∪ {j}        ← assign job j to machine i
      Li ← Li + tj             ← 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, $\mathbf{L^* \geq 2\, t_{m+1}}$.
Pf.
- Consider first m+1 jobs $t_1, \ldots, t_{m+1}$.
- 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.

▪

↑

Let $t_j$ be the last task assigned to the worst machine. Observe that **j >= m +1**. So by Lemma 3 :

$t_j$ <= $t_{m+1}$ <= (1/2) L*

Now, repeat the same reasoning of List Scheduling, and get:

$$L_i = \underbrace{(L_i - t_j)}_{\leq\, L^*} + \underbrace{t_j}_{\leq\, \frac{1}{2}L^*} \leq \frac{3}{2}L^*.$$

Lemma 3
( by observation, can assume number of jobs > 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 (At home): m machines, n = 2m+1 jobs, 2 jobs of length m+1, 2 of m +2, …, 2m-1 and one job of length m.

# Bin Packing

**Bin Packing**.
**Input**: $I = \{a_1, a_2, \dots, a_n\}$, $a_i < [0,1]$;
**Solution**: Partition $B=\{B1,..,Bk\}$ (Bins) of $I$ into
k subsets of size at most 1;
**Goal**: Minimize $k$.

**Thm. 1** Bin Packing is NP-hard.

Approximation algorithms ?

# Bin Packing

**1° STEP:** Lower bound on the Optimum k*.

Since each bin can have at most load 1 →

**Lemma 2.** k* >= S where **S** = **Σ ai (Liquid solution)**

# Bin Packing

Algorithm NEXT FIT:

- 1° item is assigned to Bin 1;
- Generic item i is assigned to **the last used Bin** if there is space otherwise open a new Bin and put it inside.

**Thm. 3** NEXT FIT is a 2-APX algorithm for B.P.

Proof.

The **sum** of items into 2 consecutive open bins is larger than 1.

So,

$$\mathbf{K(NEXT\ FIT)} < 2 * \mathbf{S} = 2 * \boldsymbol{\Sigma}\ \mathbf{ai}$$

**From Lemma 2 = k* >= S, we get the thesis.**

Remark. The bound **2** for NEXT FIT is almost tight.

Consider instances such as:

4n items: 1/2, 1/n , 1/2 , 1/n,…,1/2, 1/n;

HomeWork: Analyze the apx ratio

# Bin Packing

How to improve NEXT FIT ?

Two ideas:

- Order the Items w.r.t. non-increasing size
- For every new Item, try ALL open Bins before open a new one !
  If there is a good one, choose the **first** Bin.

=

FIRST FIT DECREASING ALGORITHM : FFD

Lemma 4. FFD is 1.5 apx algorithm for Bin Packing.
Proof.
Assume I = {a1,…,an} is ordered (non-increasing size) and
Let's partition I into:

A = {ai | ai > 2/3}; B= {bi | 1/2 < bi <= 2/3};
C = {ci | 1/3 < ci <= 1/2 } ; D = {di | di <= 1/3}

**Claim 1**. IF there is **at least one** Bin with only D-items THEN
there is at most one bin (the last one) with load < 2/3.
In this case the 1.5 apx is proved:

$\Sigma^{k-1}$ Sj (k-1) * Sj < S with Sj = ( Load of Bin j) > 2/3
From Lemma 2 : k* >= S

# Apx Solution vs Optimal Solution

Opt Solution: $k^*$ Bins, **B1, B2,....., Bk***

Apx Solution: $k > k^*$ Bins of Load = 2/3 (worst-case),

**B1, B2, .....,Bk*,...,Bk**

Facts:

- Free Space available in the Opt Solution **is not larger** then

  (1/3) $k^*$

- Load left from the Apx Solution **is not smaller** than

  (2/3) $(k-k^*)$

# Bin Packing

- **The apx solution** : **K** bins with load at least **2/3** (*forget* the last bin)

→ worst-case: each bin has load = 2/3 → has free space **1/3**

- The **Liquid/optimal solution**: it can use this free space and save bins: k → k*

**(free space) (1/3) k*** must be >= **(2/3) (k-k*)** (the rest of liquid)

so **k <= (3/2) k***

So we can assume that NO BIN j exists that has ONLY **D-items**.

**Claim 2.** In this case, FFD finds the **optimal** solution.

Proof.
Wlog may consider the new instance in which all **D**-Items are discarded.
Since the number of bins is the same! So we can analyze the
New instance!

- **A-Items** cannot be matched with any other item (= optimal)
- **no** Bin can contain more than 2 Items (= optimal)
- **B-Items** are processed by *first* and they are matched with **C-Items** (= optimal)
- Then the remaining **C-items** are matched among themselves

# Euclidean-TSP

We consider a complete weighted graph **G(V,E,w)** where
**w : E → R⁺** satisfies the **Δ-Inequality : w(x,z) <= w(x,y)+w(y,z).**

**Euclidean-TSP** = TSP restricted to Euclidean Graphs.

**THM.** Euclidean TSP is 2-Approximable

**Proof.**

**Claim 1** (Lower Bound on the Optimum)

**TSP(G) >= MST(G)**

Proof of the Claim. A Tour (without one edge) is a spanning tree!

# Euclidean TSP

Idea: Use any MST **T** and then transform it into a TOUR !!!

TAKE any MST and start by any node. Follow the tree according
The

***DEPTH FIRST SEARCH***

- Every edge is used at most twice → 2 * MST (2-apx ok!)

- Transform into a tour:
Whenever you have to come back to a visited node you jump
to the next unvisited node and use **Δ-Inequality** .

# GENERAL TSP: APX-HARDNESS

THM.
If there is a **c**-apx poly-time algorithm
for Min-TSP for some constant **c**, then P=NP.

Proof. The *GAP* technique.
Assume that a c-apx algorithm exists for TSP.

Strong Reduction from **Hamiltonian Circuit** to **TSP**:

Given an (unweighted) graph ***G(V,E)*** we construct the following complete weighted graph **G'(V,E',w):**

**w(e) = 1 if e € E and w(e) = 1 + c n otherwise**

## TSP = APX HARDNESS

**Claim 1:** **$G$** admits an Hamiltonian Circuit iff **$G'$** admits a Tour of size **$n$**

**Claim 2:** If there is no H.C. then the minimum Tour has size **>** $(n-1) + (1+cn) = n + cn = n\,(c+1)$

We can use the **$c$**-apx alg. to DECIDE the existence of H.C. in **$G$**:

- If H.C. exists then **OPT Tour = $n$** and ANY other Tour **> $(c+1)n$**.
So, $c$-apx algo **must** find the **OPT Tour** of size **$n$**. Say YES for HC

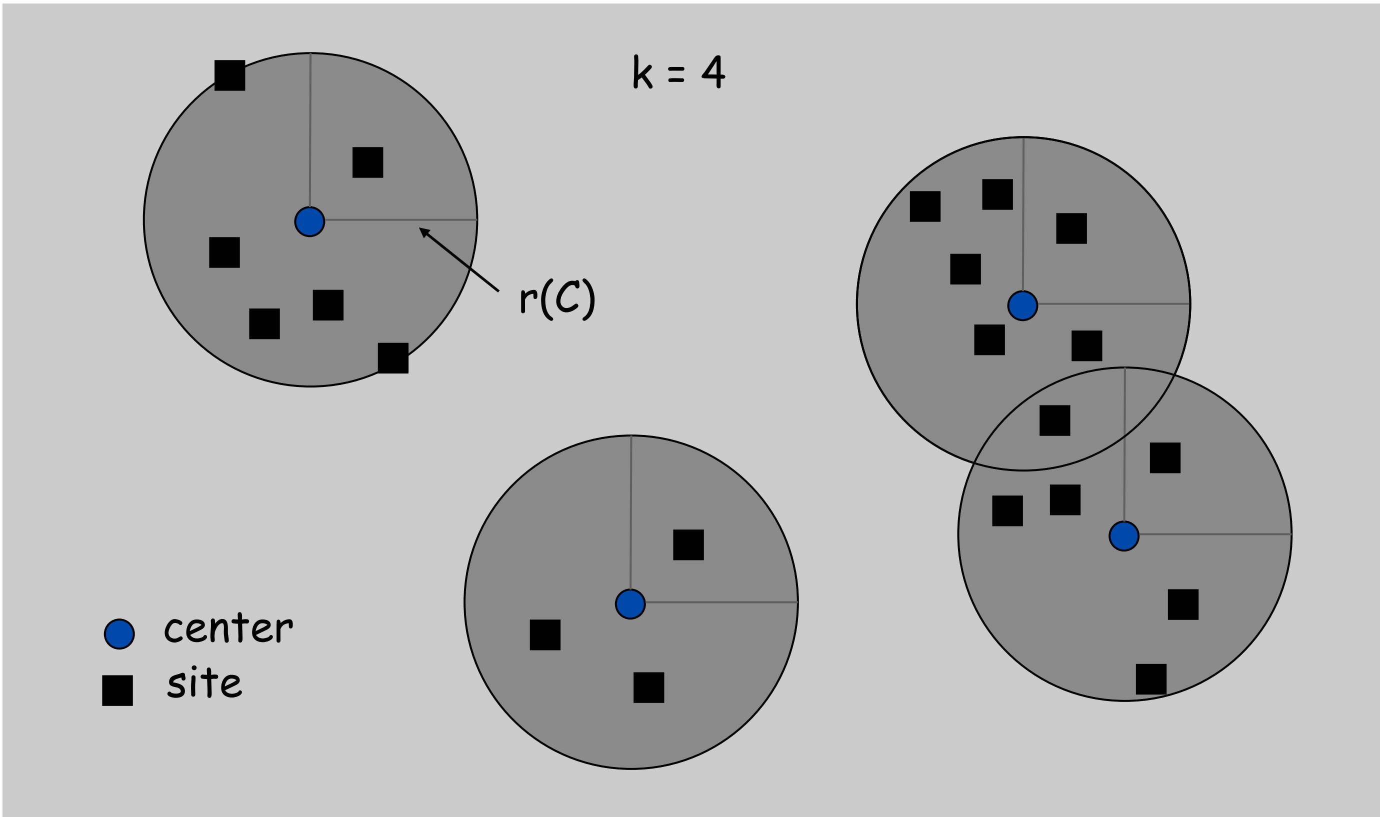
- If H.C. does not exist then the **$c$**-apx algorithm will find a Tour of size **at least $(c+1)n$**. Say NO for HC

# 11.2 Center Selection

# Center Selection Problem

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

Center selection problem. Select k centers $C$ so that maximum distance from a site to nearest center is minimized.



# Center Selection Problem

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

Center selection problem. Select k centers $C$ so that maximum distance from a site to nearest center is minimized.

Notation.

- $dist(x, y)$ = distance between x and y.
- $dist(s_i, C) = \min_{c \in C} dist(s_i, c)$ = distance from $s_i$ to closest center.
- $r(C) = \max_i dist(s_i, C)$ = smallest covering radius.

Goal. Find set of centers $C$ that minimizes $r(C)$, subject to $|C| = k$.

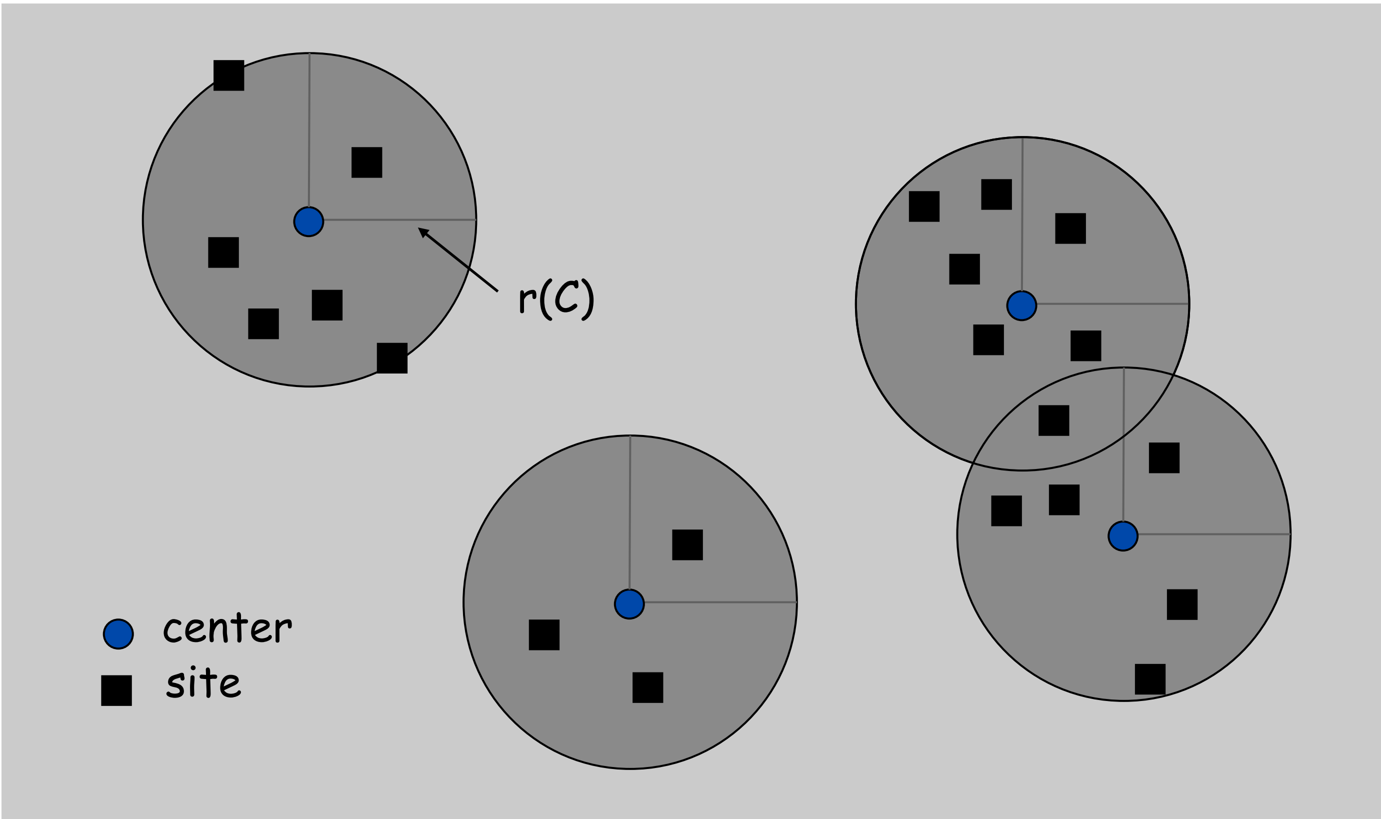
Distance function properties.

- $dist(x, x) = 0$ (identity)
- $dist(x, y) = dist(y, x)$ (symmetry)
- $dist(x, y) \leq dist(x, z) + dist(z, y)$ (triangle inequality)

# Center Selection Example

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

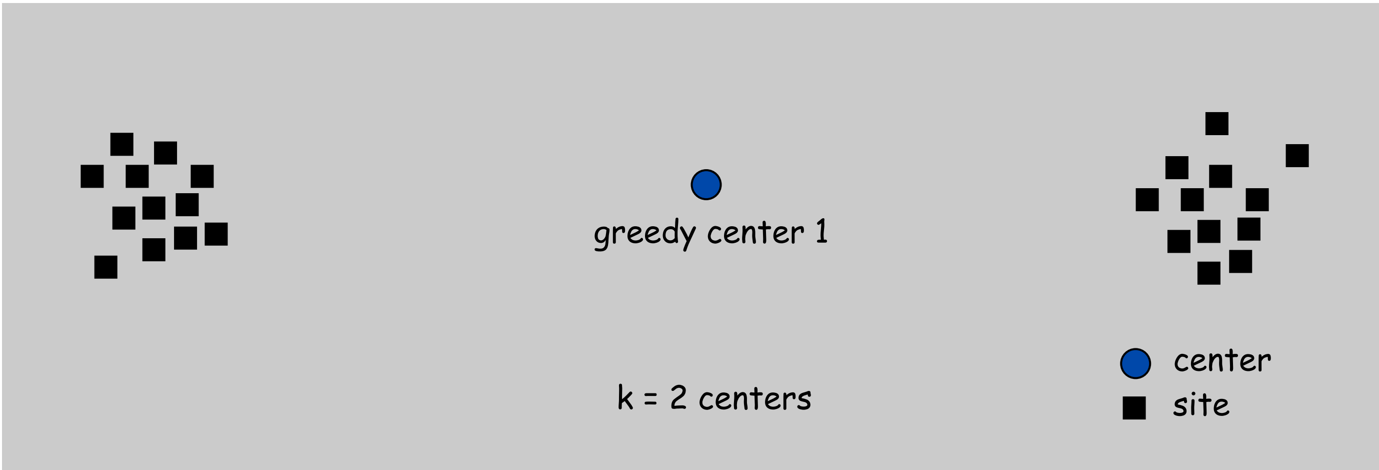
Remark: search can be infinite!



# 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

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

```
Greedy-Center-Selection(k, n, s1,s2,…,sn) {

   C = φ
   repeat k times {
      Select a site si with maximum dist(si, C)
      Add si to C
   }                      site farthest from any center
   return C
}
```

Observation. Upon termination all centers in $C$ are pairwise at least $r(C)$ apart.

Pf. By construction of algorithm.

# Center Selection: Analysis of Greedy Algorithm

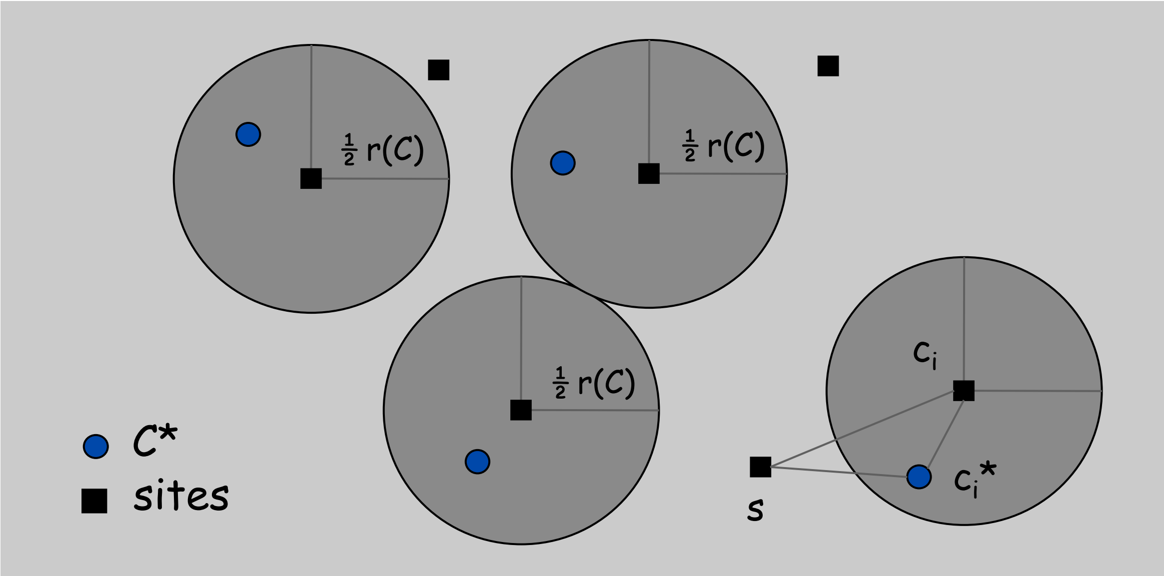
**Theorem.** Let $C^*$ be an optimal set of centers. Then $r(C) \le 2r(C^*)$.

**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 $C^*$.
- $\text{dist}(s, C) \le \text{dist}(s, c_i) \le \text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i) \le 2r(C^*)$.
- Thus $r(C) \le 2r(C^*)$. ▪

Δ-inequality

≤ $r(C^*)$ since $c_i^*$ is closest center



# Center Selection

Theorem. 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.

e.g., points in the plane

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

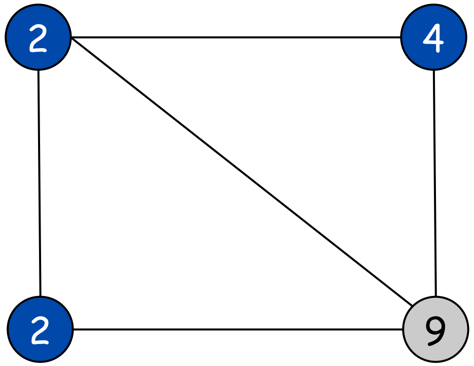
Theorem. Unless P = NP, there no $\rho$-approximation for center-selection problem for any $\rho < 2$.

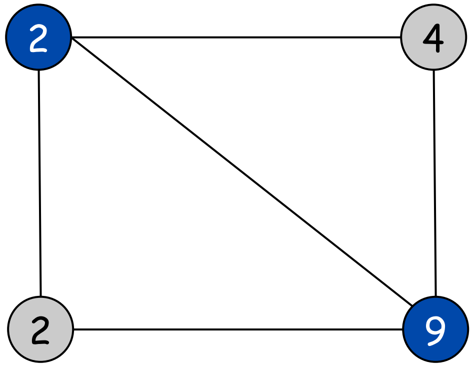
# 11.4 The Pricing Method: Vertex Cover

# Weighted Vertex Cover

**Definition.** Given a graph G = (V, E), a vertex cover is a set $S \subseteq V$ such that each edge in E has at least one end in S.

**Weighted vertex cover.** Given a graph G with vertex weights, find a vertex cover of minimum weight.




weight = 2 + 2 + 4

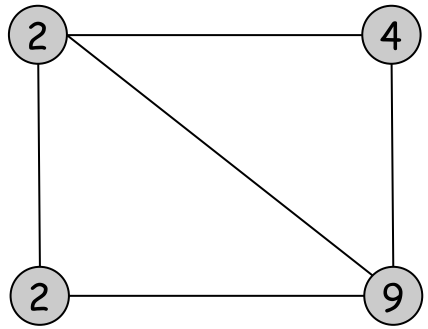
weight = 11

# Pricing Method

Pricing method. Each edge must be covered by some vertex.
Edge e = (i, j) pays price $p_e \geq 0$ to use vertex i and j.

Fairness. Edges incident to vertex i should pay $\leq w_i$ in total.

for each vertex $i$: $\sum_{e=(i,j)} p_e \leq w_i$




Lemma. For any vertex cover S and any fair prices $p_e$: $\sum_e p_e \leq w(S)$.

Pf. ▪

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

each edge e covered by at least one node in S

sum fairness inequalities for each node in S

# Pricing Method

Pricing method. Set prices and find vertex cover simultaneously.

$$\sum_{e=(i,j)} p_e = w_i$$

```
Weighted-Vertex-Cover-Approx(G, w) {
   foreach e in E
      pe = 0

   while (∃ edge i-j such that neither i nor j are tight)
      select such an edge e
      increase pe as much as possible until i or j tight
   }

   S ← set of all tight nodes
   return S
}
```

# Pricing Method

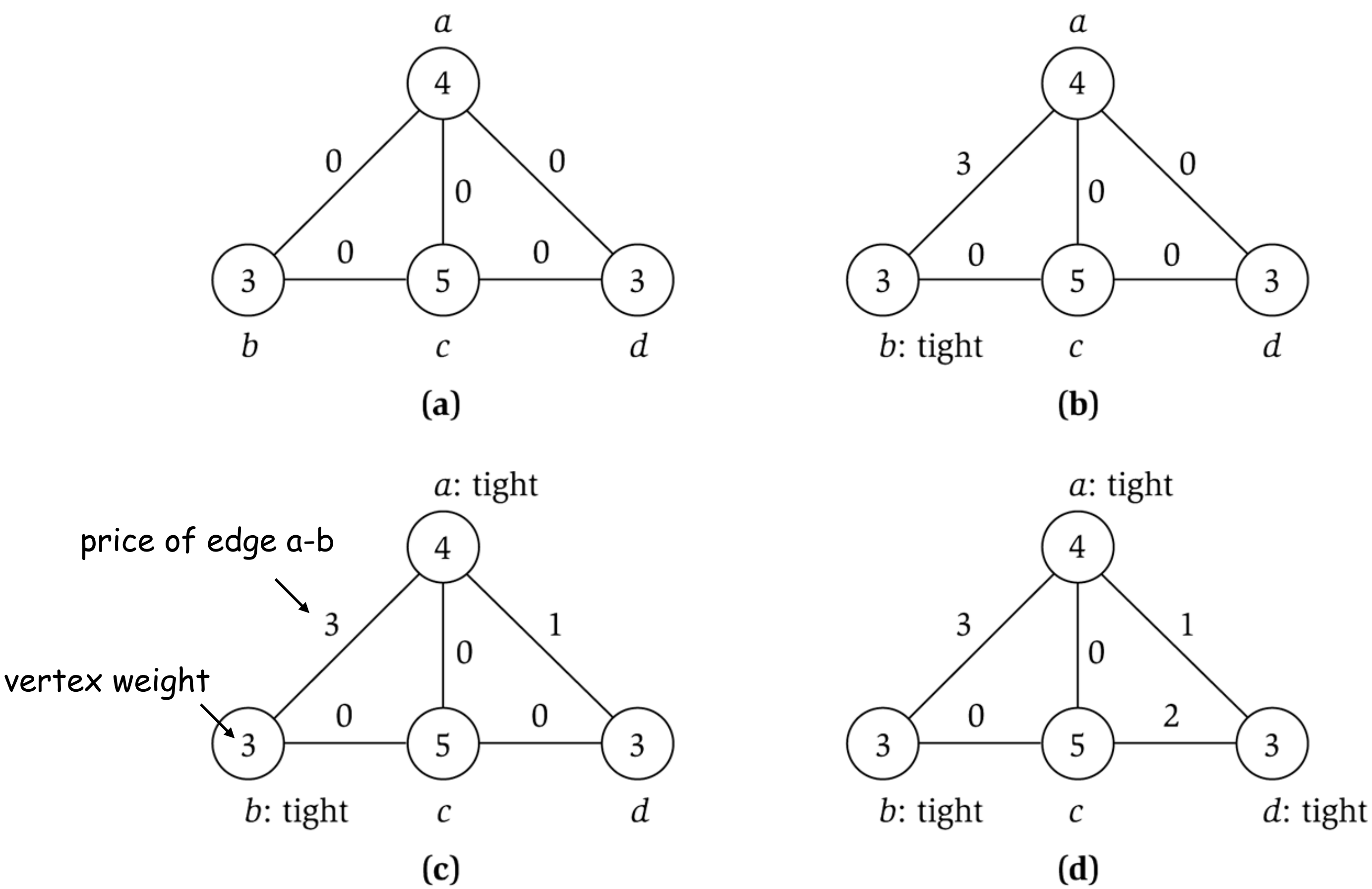



Figure 11.8

# Pricing Method: Analysis

Theorem. Pricing method is a 2-approximation.
Pf.

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

- Let S = 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.

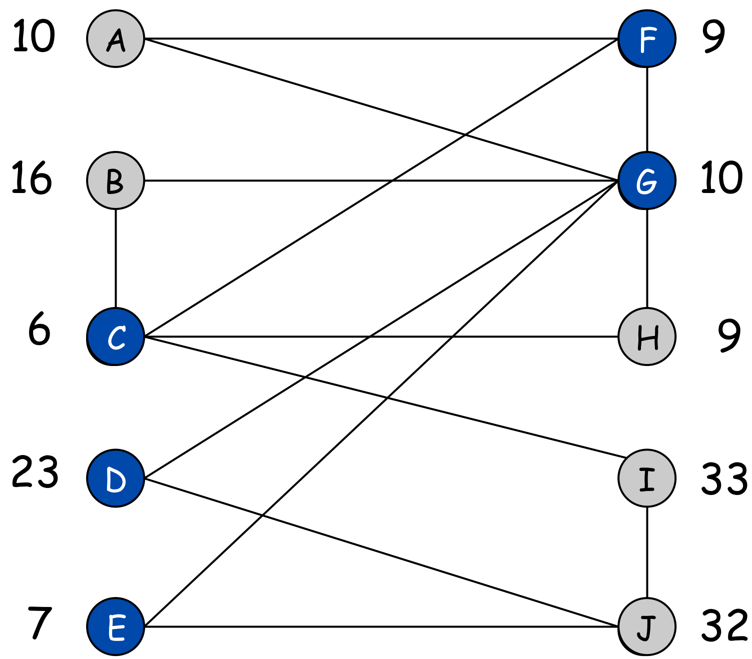
- Let S* be optimal vertex cover. We show w(S) ≤ 2w(S*).

$$w(S) = \sum_{i \in S} w_i \underbrace{=}_{\text{all nodes in S are tight}} \sum_{i \in S} \sum_{e=(i,j)} p_e \underbrace{\leq}_{S \subseteq V,\ \text{prices} \geq 0} \sum_{i \in V} \sum_{e=(i,j)} p_e \underbrace{=}_{\text{each edge counted twice}} 2 \sum_{e \in E} p_e \underbrace{\leq}_{\text{fairness lemma}} 2w(S^*). \quad \blacksquare$$

# 11.6 LP Rounding: Vertex Cover

# Weighted Vertex Cover

Weighted vertex cover. Given an undirected graph $G = (V, E)$ with vertex weights $w_i \geq 0$, find a minimum weight subset of nodes $S$ such that every edge is incident to at least one vertex in $S$.




total weight = 55

## Weighted Vertex Cover: IP Formulation

Weighted vertex cover. Given an undirected graph $G = (V, E)$ with vertex weights $w_i \geq 0$, find a minimum weight subset of nodes $S$ 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: maximize $\Sigma_i w_i x_i$.

- Must take either $i$ or $j$: $x_i + x_j \geq 1$.

# Weighted Vertex Cover: IP Formulation

Weighted vertex cover. Integer programming formulation.

$$
\begin{array}{llccl}
(ILP) & \min & \displaystyle\sum_{i \in V} w_i \, x_i & & \\
 & \text{s. t.} & x_i + x_j & \geq & 1 & (i,j) \in E \\
 & & x_i & \in & \{0,1\} & i \in V
\end{array}
$$

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

# Integer Programming

INTEGER-PROGRAMMING. Given integers $a_{ij}$ and $b_i$, find integers $x_j$ that satisfy:

$$\begin{array}{lrcl} \max & c^t x & & \\ \text{s. t.} & Ax & \geq & b \\ & x & & \text{integral} \end{array}$$

$$\begin{array}{rcll} \sum_{j=1}^{n} a_{ij} x_j & \geq & b_i & 1 \leq i \leq m \\ x_j & \geq & 0 & 1 \leq j \leq n \\ x_j & & \text{integral} & 1 \leq j \leq n \end{array}$$

Observation. Vertex cover formulation proves that integer programming is NP-hard search problem.

even if all coefficients are 0/1 and
at most two variables per inequality

# Linear Programming

Linear programming. Max/min linear objective function subject to linear inequalities.

- Input: integers $c_j$, $b_i$, $a_{ij}$ .
- Output: real numbers $x_j$.

$$
\begin{array}{lllll}
(P) & \max & c^t x & & \\
 & \text{s.t.} & Ax & \geq & b \\
 & & x & \geq & 0
\end{array}
$$

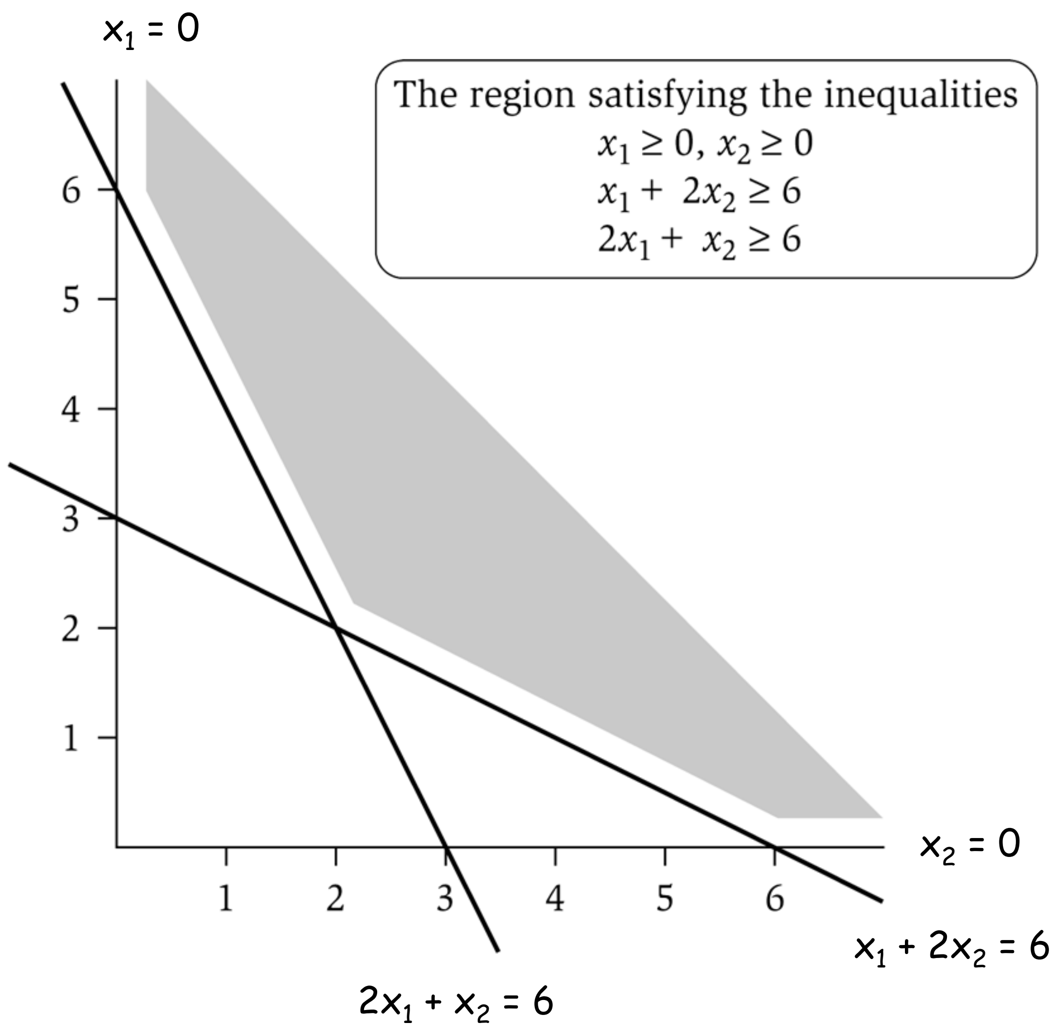
$$
\begin{array}{llllll}
(P) & \max & \sum_{j=1}^{n} c_j x_j & & & \\
 & \text{s.t.} & \sum_{j=1}^{n} a_{ij} x_j & \geq & b_i & 1 \leq i \leq m \\
 & & x_j & \geq & 0 & 1 \leq j \leq n
\end{array}
$$

Linear. No $x^2$, $xy$, $\arccos(x)$, $x(1-x)$, etc.

Simplex algorithm. [Dantzig 1947] Can solve LP in practice.
Ellipsoid algorithm. [Khachian 1979] Can solve LP in poly-time.

# LP Feasible Region

LP geometry in 2D.



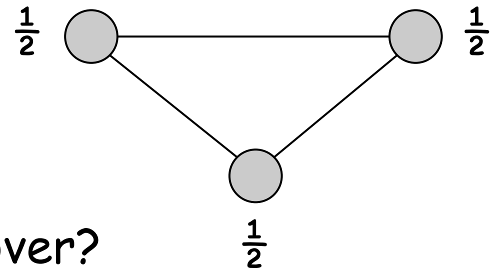
# Weighted Vertex Cover: LP Relaxation

Weighted vertex cover. Linear programming formulation.

$$
\begin{array}{llllll}
(LP) & \min & \sum\limits_{i \in V} w_i \, x_i & & & \\
& \text{s.t.} & x_i + x_j & \geq & 1 & (i,j) \in E \\
& & x_i & \geq & 0 & i \in V
\end{array}
$$

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

Theorem. 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

$$\sum_{i \in S^*} w_i \underset{\substack{\uparrow \\ \text{LP is a relaxation}}}{\geq} \sum_{i \in S} w_i\, x_i^* \underset{\substack{\uparrow \\ x^*_i \,\geq\, \frac{1}{2}}}{\geq} \tfrac{1}{2} \sum_{i \in S} w_i$$

# Weighted Vertex Cover

Theorem. 2-approximation algorithm for weighted vertex cover.

Theorem. [Dinur-Safra 2001] If P $\neq$ NP, then no $\rho$-approximation for $\rho < 1.3607$, even with unit weights.

$10\sqrt{5} - 21$

Open research problem. Close the gap.

# * 11.7 Load Balancing Reloaded

# Generalized Load Balancing

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

- Job j must run contiguously on an authorized machine in $M_j \subseteq M$.
- Job 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 $L_i = \Sigma_{j \in J(i)} t_j$.

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: Integer Linear Program and Relaxation

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

$$
\begin{array}{llllll}
(IP) & \min & L & & & \\
& \text{s.t.} & \sum_i x_{ij} & = & t_j & \text{for all } j \in J \\
& & \sum_j x_{ij} & \leq & L & \text{for all } i \in M \\
& & x_{ij} & \in & \{0,\ t_j\} & \text{for all } j \in J \text{ and } i \in M_j \\
& & x_{ij} & = & 0 & \text{for all } j \in J \text{ and } i \notin M_j
\end{array}
$$

**LP relaxation.**

$$
\begin{array}{llllll}
(LP) & \min & L & & & \\
& \text{s.t.} & \sum_i x_{ij} & = & t_j & \text{for all } j \in J \\
& & \sum_j x_{ij} & \leq & L & \text{for all } i \in M \\
& & x_{ij} & \geq & 0 & \text{for all } j \in J \text{ and } i \in M_j \\
& & x_{ij} & = & 0 & \text{for all } j \in J \text{ and } i \notin M_j
\end{array}
$$

# Generalized Load Balancing: Lower Bounds

Lemma 1. Let $L$ be the optimal value to the LP. Then, the optimal makespan $L^* \geq L$.

Pf. LP has fewer constraints than IP formulation.

Lemma 2. The optimal makespan $L^* \geq \max_j t_j$.

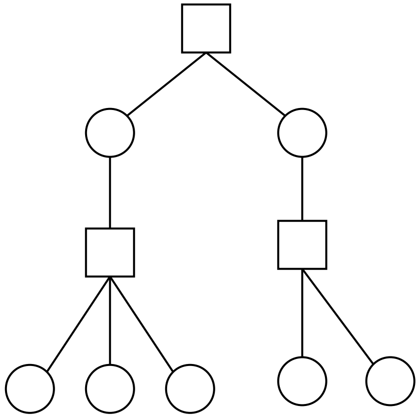
Pf. Some machine must process the most time-consuming job. ▪

# Generalized Load Balancing: Structure of LP Solution

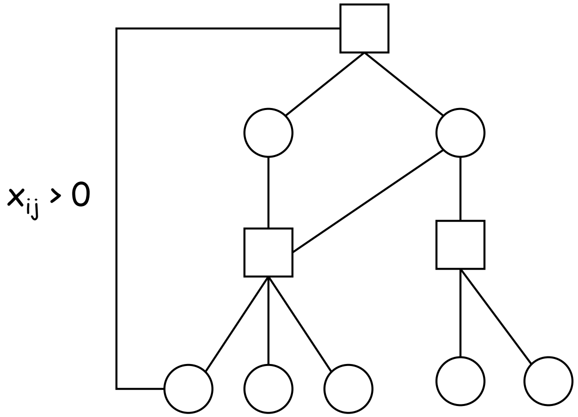
Lemma 3. Let x be solution to LP. Let G(x) be the graph with an edge from machine i to job j if $x_{ij} > 0$. Then G(x) is acyclic.

can transform x into another LP solution where G(x) is acyclic if LP solver doesn't return such an x

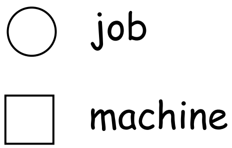
Pf. (deferred)



G(x) acyclic




G(x) cyclic

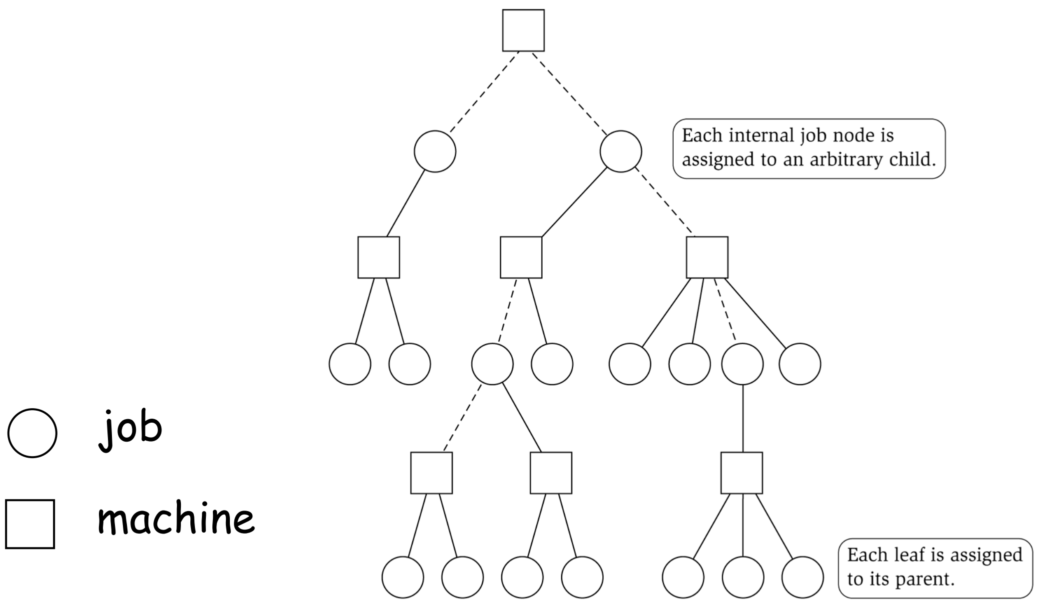


# Generalized Load Balancing: Rounding

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 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. ▪



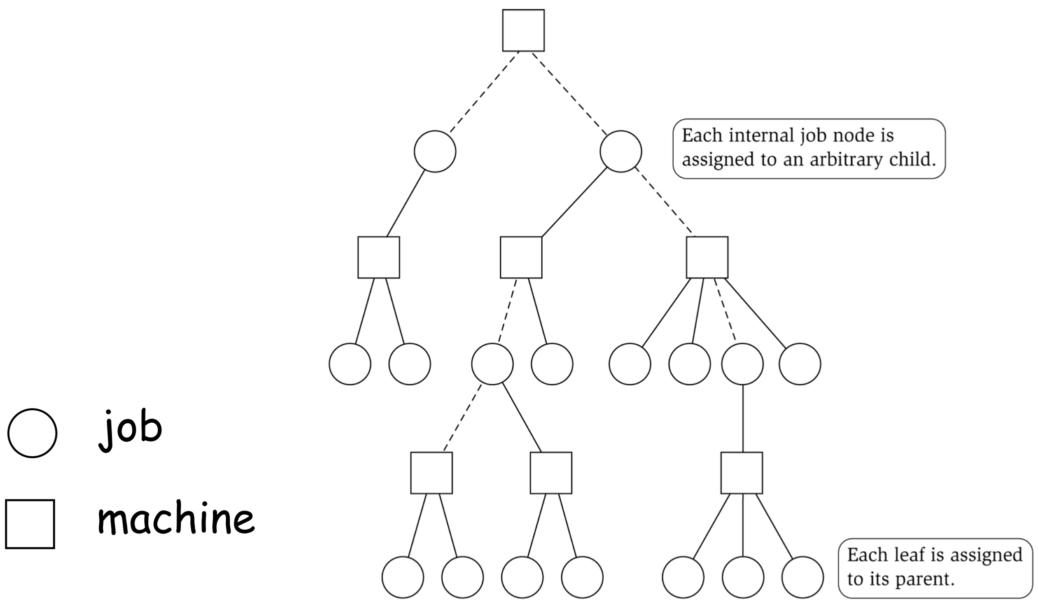
## Generalized Load Balancing: Analysis

Lemma 5. If job j is a leaf node and machine i = parent(j), then $x_{ij} = t_j$.

Pf. Since i is a leaf, $x_{ij} = 0$ for all $j \neq$ parent(i). LP constraint guarantees $\Sigma_i \, x_{ij} = t_j$. ▪

Lemma 6. At most one non-leaf job is assigned to a machine.

Pf. The only possible non-leaf job assigned to machine i is parent(i). ▪



# 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

$$\sum_{\substack{j \in J(i) \\ j \text{ is a leaf}}} t_j \overset{\text{Lemma 5}}{=} \sum_{\substack{j \in J(i) \\ j \text{ is a leaf}}} x_{ij} \le \sum_{j \in J} x_{ij} \overset{\text{LP}}{\le} \underset{\text{optimal value of LP}}{L} \overset{\text{Lemma 1 (LP is a relaxation)}}{\le} L^*$$

  - parent(i)

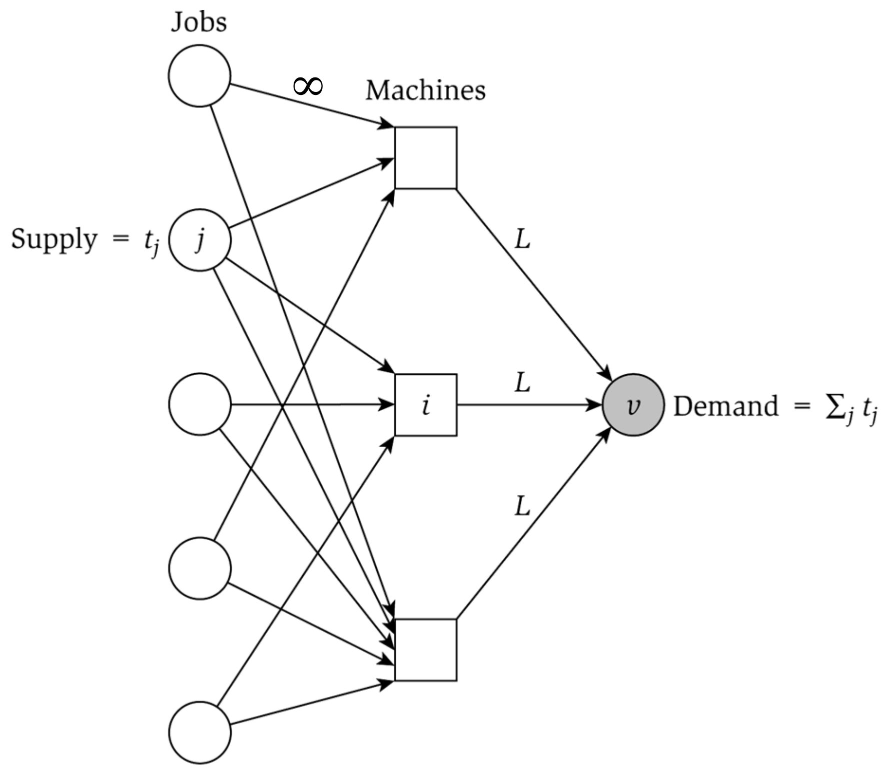
$$t_{\text{parent}(i)} \overset{\text{Lemma 2}}{\le} L^*$$

- Thus, the overall load $L_i \le 2L^*$. ▪

# Generalized Load Balancing: Flow Formulation

## Flow formulation of LP.

$$
\begin{array}{lcll}
\sum_i x_{ij} & = & t_j & \text{for all } j \in J \\
\sum_j x_{ij} & \le & L & \text{for all } i \in M \\
x_{ij} & \ge & 0 & \text{for all } j \in J \text{ and } i \in M_j \\
x_{ij} & = & 0 & \text{for all } j \in J \text{ and } i \notin M_j
\end{array}
$$



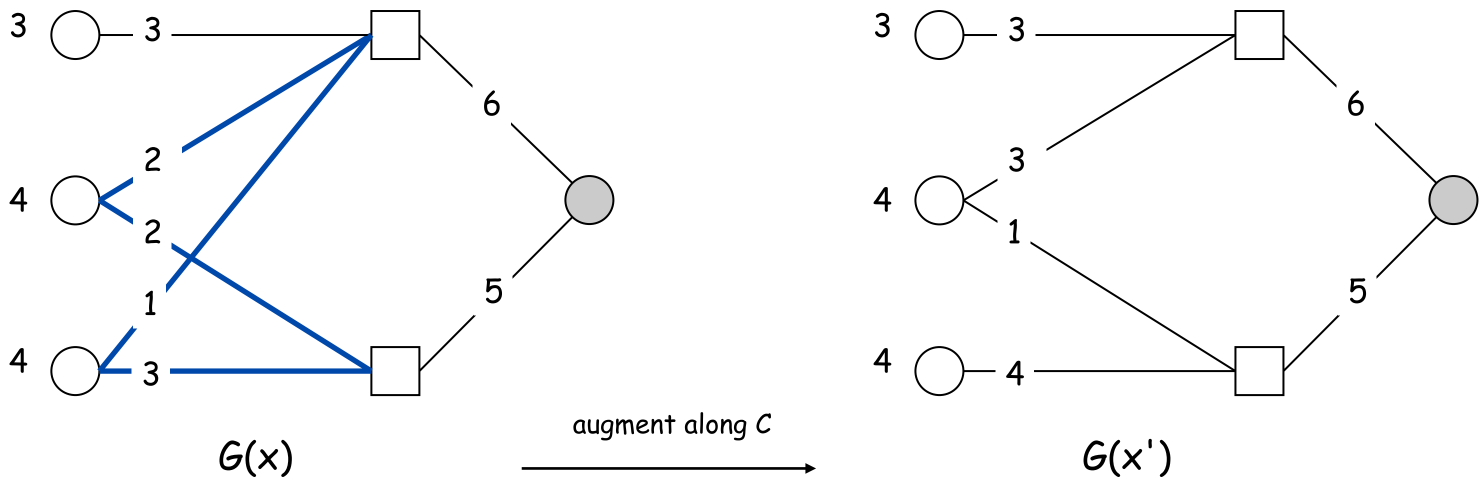

**Observation.** Solution to feasible flow problem with value L are in one-to-one correspondence with LP solutions of value L.

## 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.



# 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 $t_{ij}$ time if processed on machine i.
- 2-approximation algorithm via LP rounding.
- No 3/2-approximation algorithm unless P = NP.

# 11.8 Knapsack Problem

## 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.

- 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.

W = 11

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

# 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. SUBSET-SUM $\leq_P$ KNAPSACK.

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

$$v_i = w_i = u_i \qquad \sum_{i \in S} u_i \leq U$$
$$V = W = U \qquad \sum_{i \in S} u_i \geq U$$

# 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 – $w_i$

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

Running time. O(n W).

- W = weight limit.
- Not polynomial in input size!

# Knapsack Problem: Dynamic Programming II

Def. OPT(i, v) = min weight subset of items 1, …, i that yields value exactly v.

- Case 1: OPT does not select item i.
  - OPT selects best of 1, …, i-1 that achieves exactly value v
- Case 2: OPT selects item i.
  - consumes weight $w_i$, new value needed = $v - v_i$
  - OPT selects best of 1, …, i-1 that achieves exactly value v

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

$V^* \leq n\, v_{max}$

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

- $V^*$ = optimal value = maximum v such that OPT(n, v) ≤ 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.

| Item | Value | Weight |
|---|---|---|
| 1 | 934,221 | 1 |
| 2 | 5,956,342 | 2 |
| 3 | 17,810,013 | 5 |
| 4 | 21,217,800 | 6 |
| 5 | 27,343,199 | 7 |

W = 11

original instance

| Item | Value | Weight |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 6 | 2 |
| 3 | 18 | 5 |
| 4 | 22 | 6 |
| 5 | 28 | 7 |

W = 11

rounded instance

# Knapsack: FPTAS

**Knapsack FPTAS.** Round up all values: $\bar{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil$

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

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

**Intuition.** $\bar{v}$ close to v so optimal solution using $\bar{v}$ 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 $O(n^2 \, \hat{v}_{max})$, where

$$\hat{v}_{max} = \left\lceil \frac{v_{max}}{\theta} \right\rceil = \left\lceil \frac{n}{\varepsilon} \right\rceil$$

# Knapsack: FPTAS

**Knapsack FPTAS.** Round up all values: $\bar{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta$

**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.

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \bar{v}_i \qquad \text{always round up}$$

$$\leq \sum_{i \in S} \bar{v}_i \qquad \text{solve rounded instance optimally}$$

$$\leq \sum_{i \in S} (v_i + \theta) \qquad \text{never round up by more than } \theta$$

$$\leq \sum_{i \in S} v_i + n\theta \qquad |S| \leq n$$

$$\leq (1+\varepsilon) \sum_{i \in S} v_i \qquad n\theta = \varepsilon v_{max},\ \underbrace{v_{max} \leq \Sigma_{i \in S} v_i}_{\text{DP alg can take } v_{max}}$$

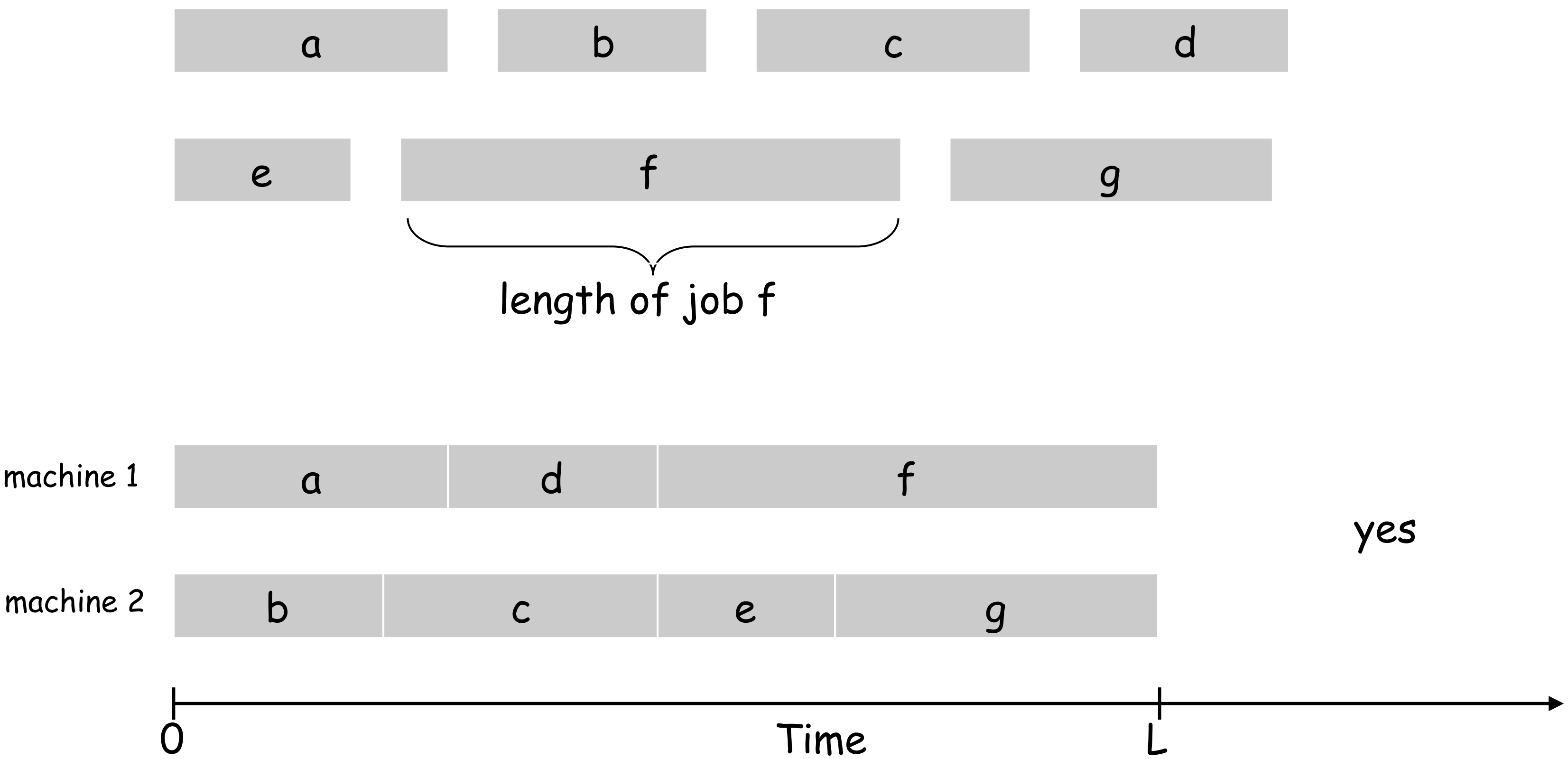
# Extra Slides

# Load Balancing on 2 Machines

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

Pf. NUMBER-PARTITIONING $\leq_P$ LOAD-BALANCE.

NP-complete by Exercise 8.26



# Center Selection: Hardness of Approximation

Theorem. Unless P = NP, there is no $\rho$-approximation algorithm for metric k-center problem for any $\rho < 2$.

Pf. We show how we could use a $(2 - \varepsilon)$ approximation algorithm for k-center to solve DOMINATING-SET in poly-time.

- Let $G = (V, E)$, k be an instance of DOMINATING-SET. ⟵ see Exercise 8.29
- Construct instance $G'$ of k-center with sites V and distances
  - $d(u, v) = 2$ if $(u, v) \in E$
  - $d(u, v) = 1$ if $(u, v) \notin E$
- Note that $G'$ satisfies the triangle inequality.
- Claim: $G$ 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 $(2 - \varepsilon)$-approximation algorithm on $G'$ must find a solution $C^*$ with $r(C^*) = 1$ since it cannot use any edge of distance 2.