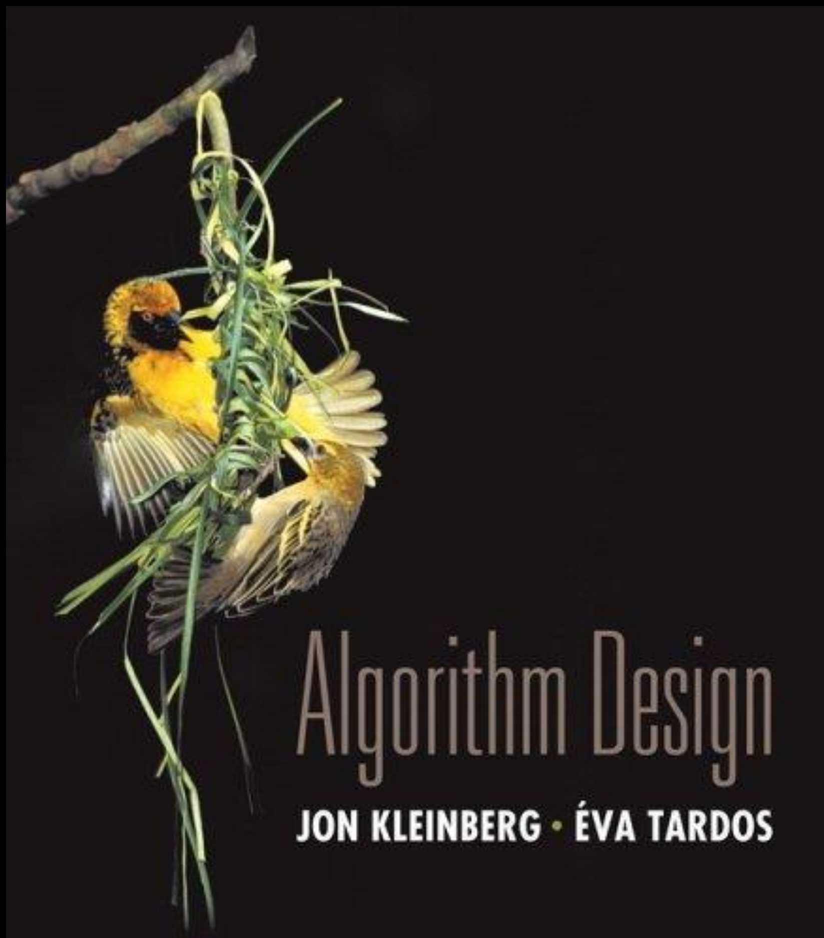
Homework 5

Homework 5

- Covers chapters 9, 10, 11, 13
- To be released today in Blackboard
- Due: Dec. 19 before class
- No late homework will be accepted!
- Submit your solutions in gradescope.



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.

ρ-approximation algorithm.

- Guaranteed to run in poly-time.
- Guaranteed to solve arbitrary instance of the problem
- Guaranteed to find solution within ratio ρ 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

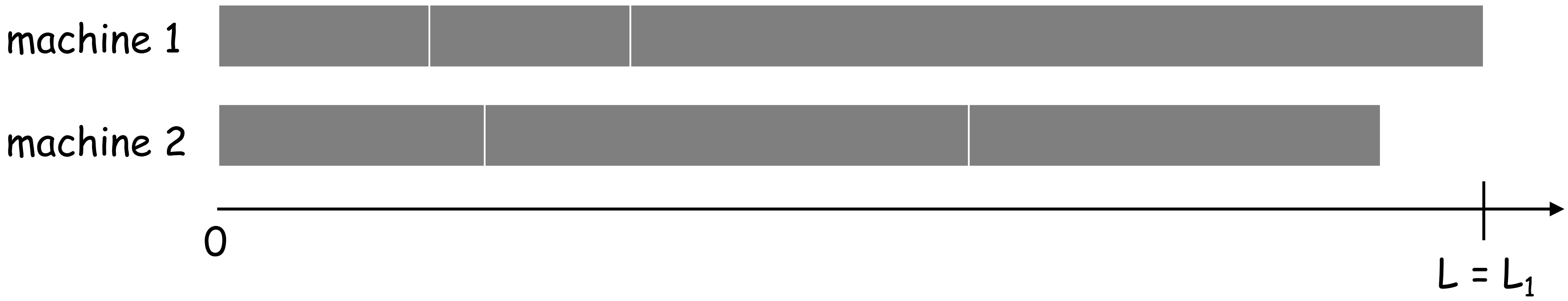
Input. m identical machines; n jobs, job j has processing time t_j.

- 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

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 $L = \max_i L_i$.

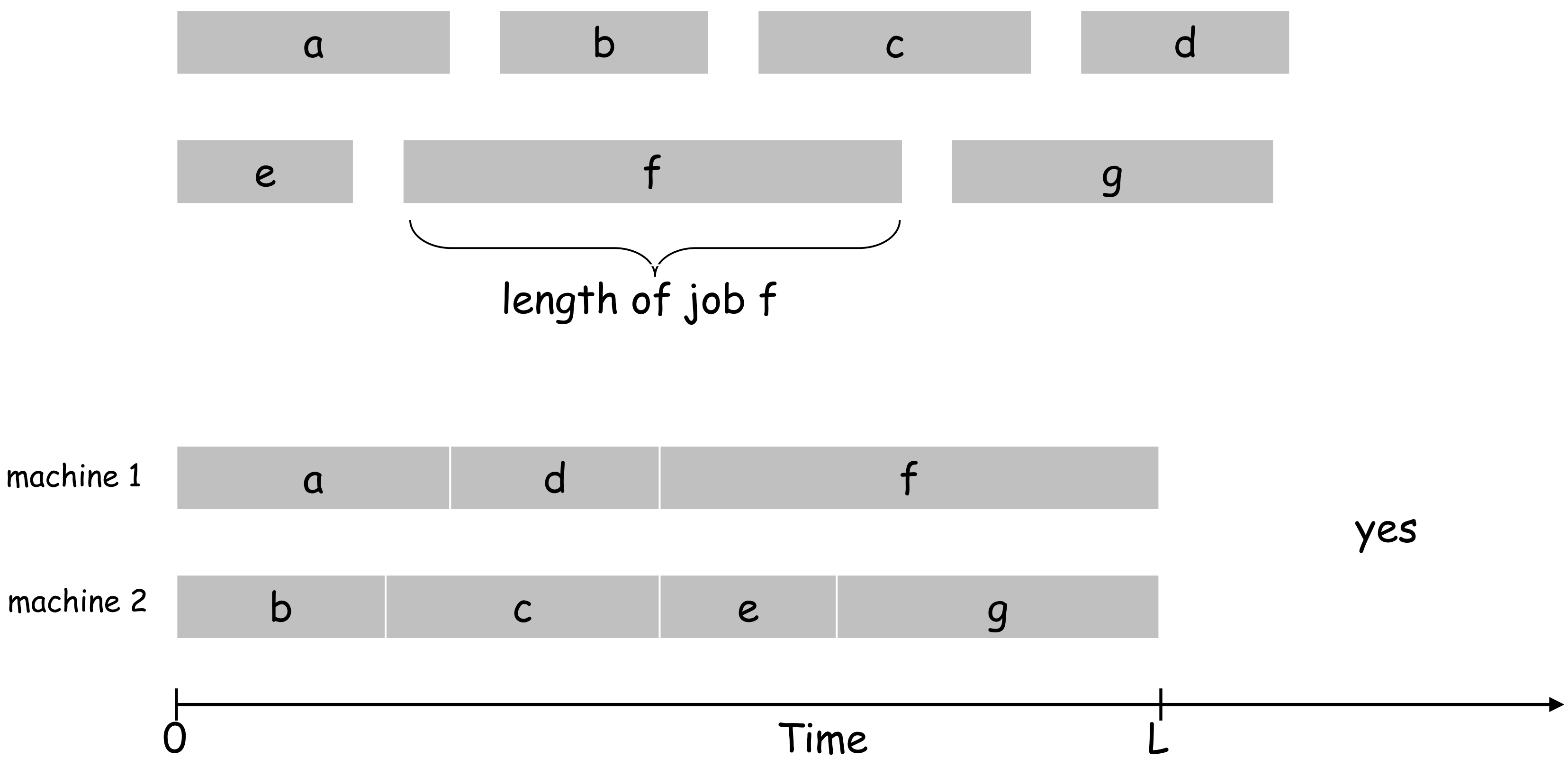
Load balancing. Assign each job to a machine to minimize makespan.

Claim. Load balancing is NP-hard.
Pf. NUMBER-PARTITION $\le_P$ LOAD-BALANCE

Load Balancing on 2 Machines

Claim. Load balancing is NP-hard.

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



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
   }
}
```

Implementation. $O(n \log n)$ 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. ▪

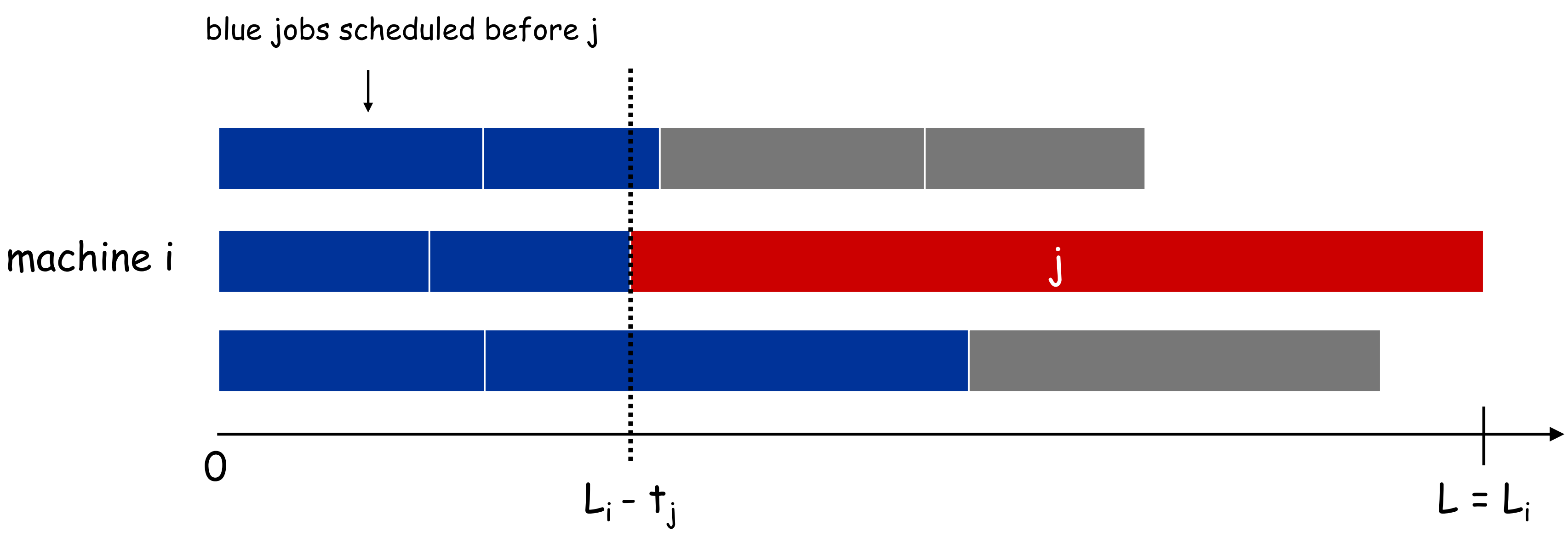
Load Balancing: List Scheduling Analysis

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:

  $L_i - t_j \leq L_k$ for all $1 \leq k \leq m$.



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:

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

$$\begin{aligned} L_i - t_j &\le \tfrac{1}{m} \sum_{k=1}^{m} L_k \\ &= \tfrac{1}{m} \sum_{l=1}^{n} t_l \\ \text{Lemma 2} \longrightarrow \quad &\le L^* \end{aligned}$$

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

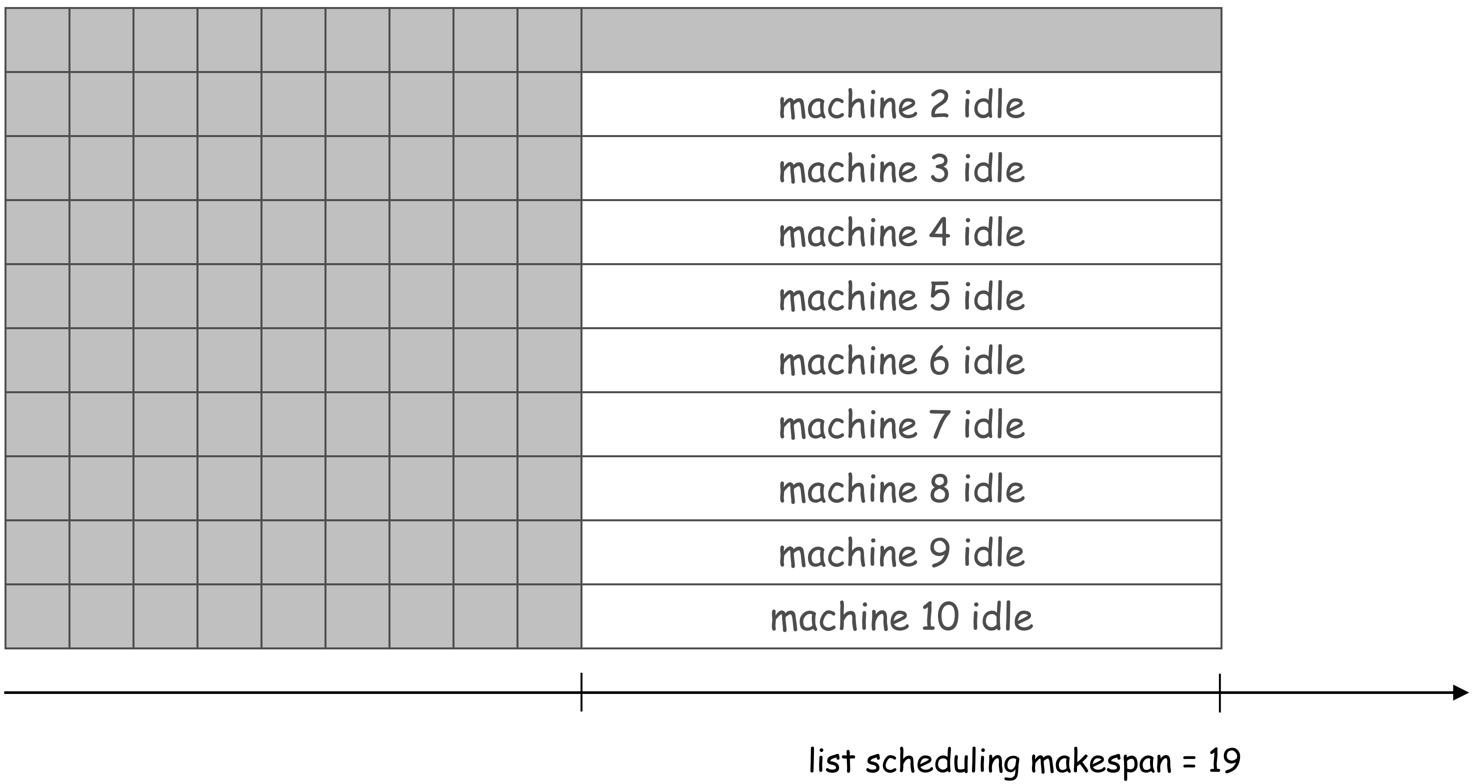
Lemma 1

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
$L = 2m-1$;

m = 10



Load Balancing: List Scheduling Analysis

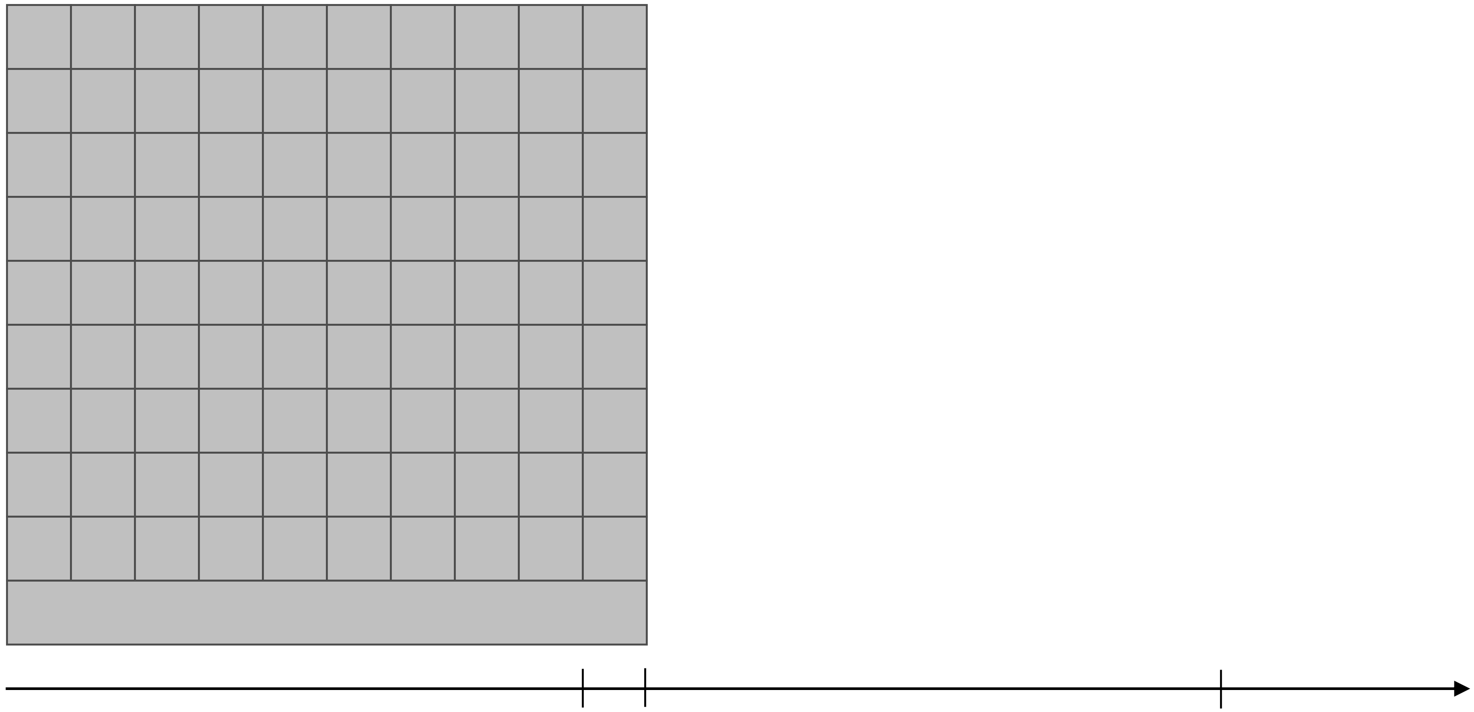
Q. Is our analysis tight?

A. Essentially yes.

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

$L = 2m-1$; $L^* = m$

m = 10



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 {
      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
   }
}
```

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, $L^* \geq 2\, t_{m+1}$.
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. If max-load machine M_i has only one job, then it's optimal (lemma 1)
Otherwise, for its last job t_j we have j>m. Use the same approach as for list scheduling.

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

Lemma 3

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.

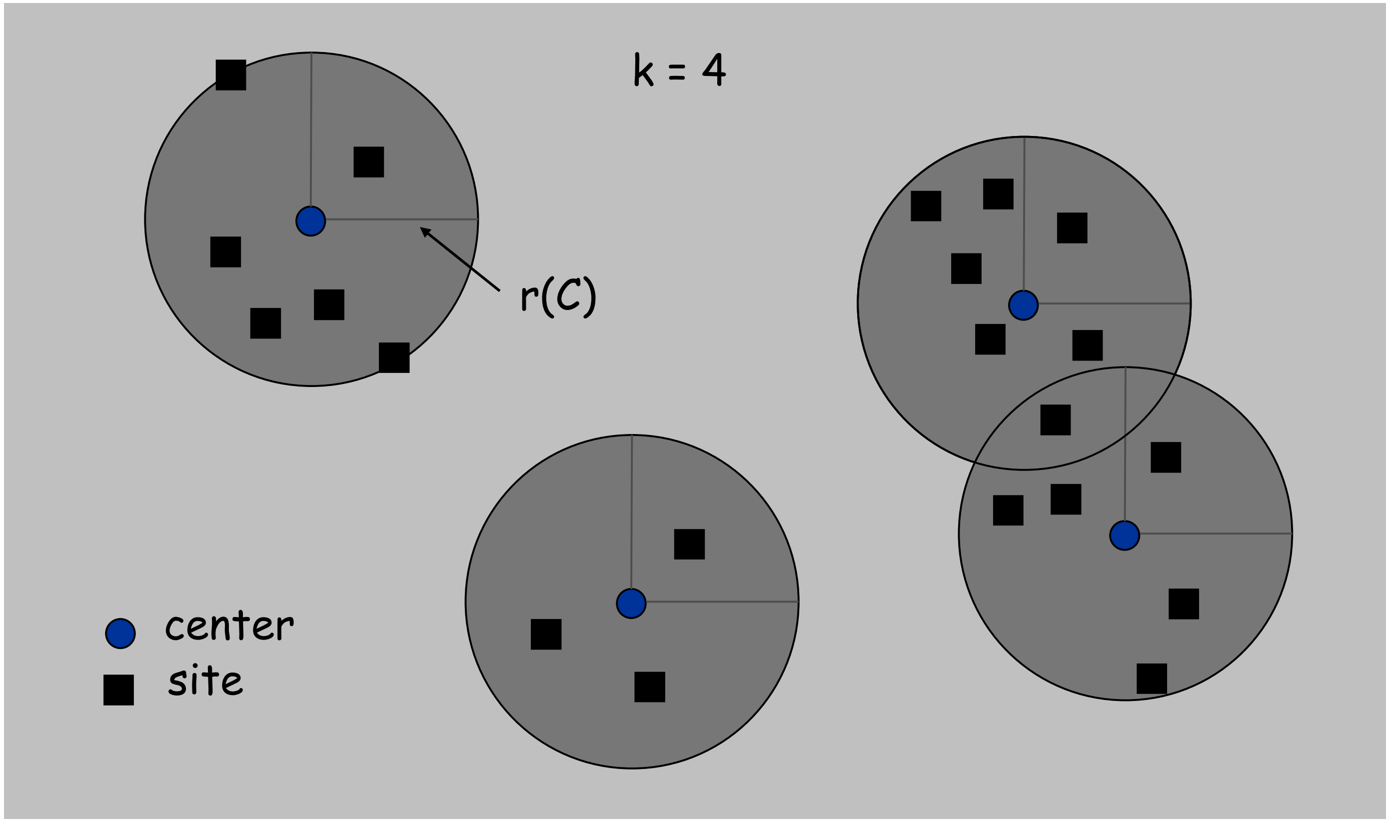
Ex: m machines, n = 2m+1 jobs, 2 jobs of length 2m-1, 2m-2, …, m+1 and 3 jobs of length m.

11.2 Center Selection

Center Selection Problem

Input. Set of n sites $s_1, \ldots, s_n$.

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 $s_1, \ldots, s_n$.

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

Notation.

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

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

Distance function properties.

- $\text{dist}(x, x) = 0$ (identity)
- $\text{dist}(x, y) = \text{dist}(y, x)$ (symmetry)
- $\text{dist}(x, y) \le \text{dist}(x, z) + \text{dist}(z, y)$ (triangle inequality)

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 = {s}, s is any site
   repeat k-1 times {
      Select a site si with maximum dist(si, C)
      Add si to C
   }
   return C
}
```

site farthest from any center

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

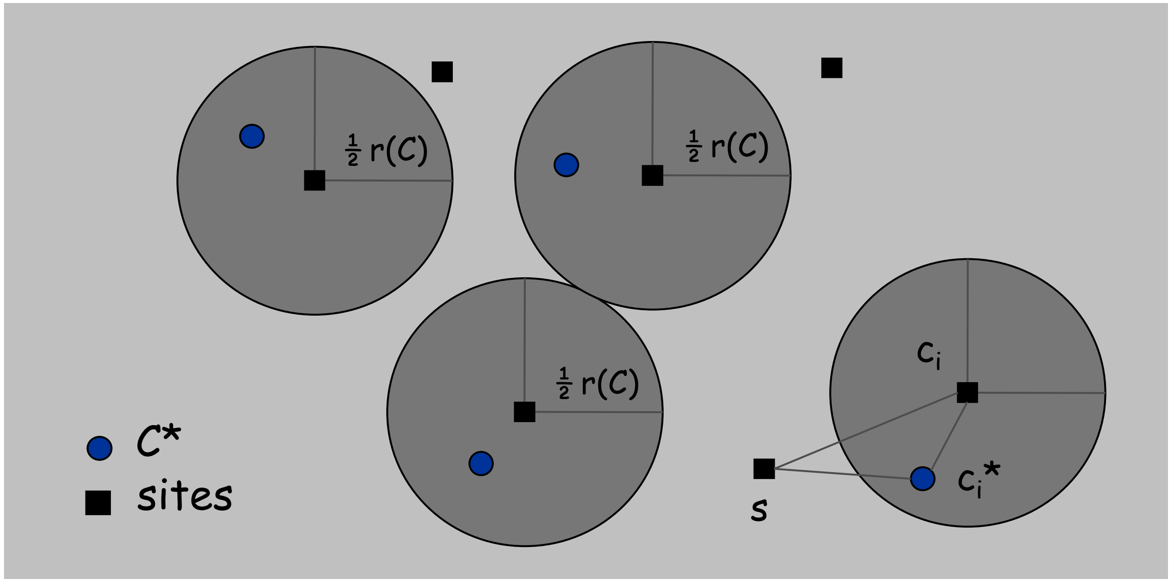
Pf. In the algorithm, the distance from a new center to other centers is at least the value of r(C) before the new center is added; with more centers, r(C) always monotonically decreases.

Center Selection: Analysis of Greedy Algorithm

Theorem. 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 center/site c_i in C, consider ball of radius $\frac{1}{2} r(C)$ around it.
  - No overlap between balls (by lemma)
- At least one $c_i^* \in C^*$ in each ball (because $r(C^*) < \frac{1}{2} r(C)$)
  - Therefore, exactly one $c_i^* \in C^*$ in each ball



Center Selection: Analysis of Greedy Algorithm

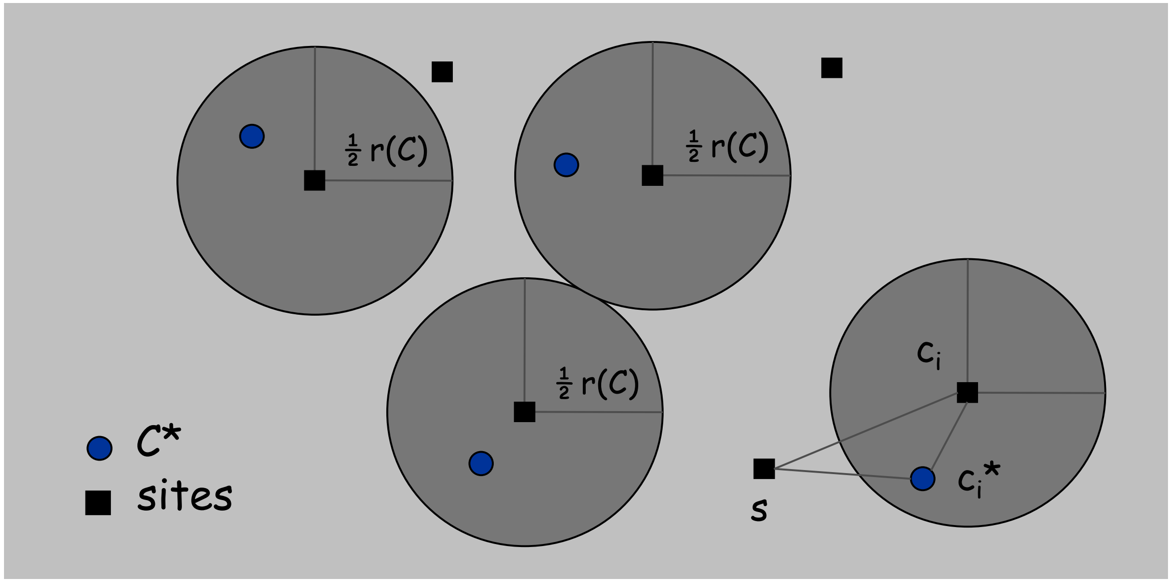
Theorem. Let C^* be an optimal set of centers. Then $r(C) \le 2r(C^*)$.

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

- 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

$\le r(C^*)$ 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

Center Selection: Hardness of Approximation

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

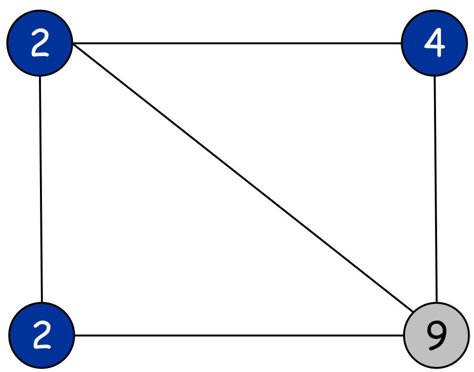
Theorem. Unless P = NP, there is no ρ-approximation algorithm for metric k-center problem for any $\rho < 2$.

Proof idea. Show how we could use a $(2 - \varepsilon)$ approximation algorithm for k-center to solve an NP-complete problem (DOMINATING-SET) in poly-time.

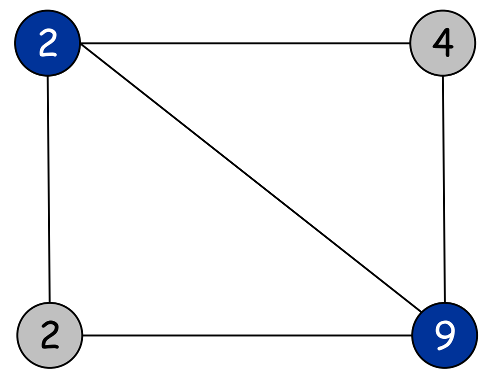
11.4 The Pricing Method: Vertex Cover

Weighted Vertex Cover

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



weight = 2 + 2 + 4



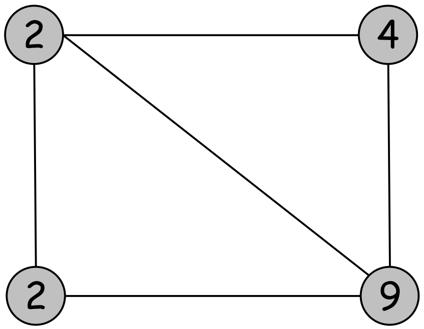
weight = 11

Weighted Vertex Cover

Pricing method. Each edge must be covered by some vertex. Edge e pays price $p_e \geq 0$ to use a vertex.

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

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



Fairness Lemma. For any vertex cover S and any fair prices p_e:

$$\sum_e p_e \leq w(S).$$

Proof.

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

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.

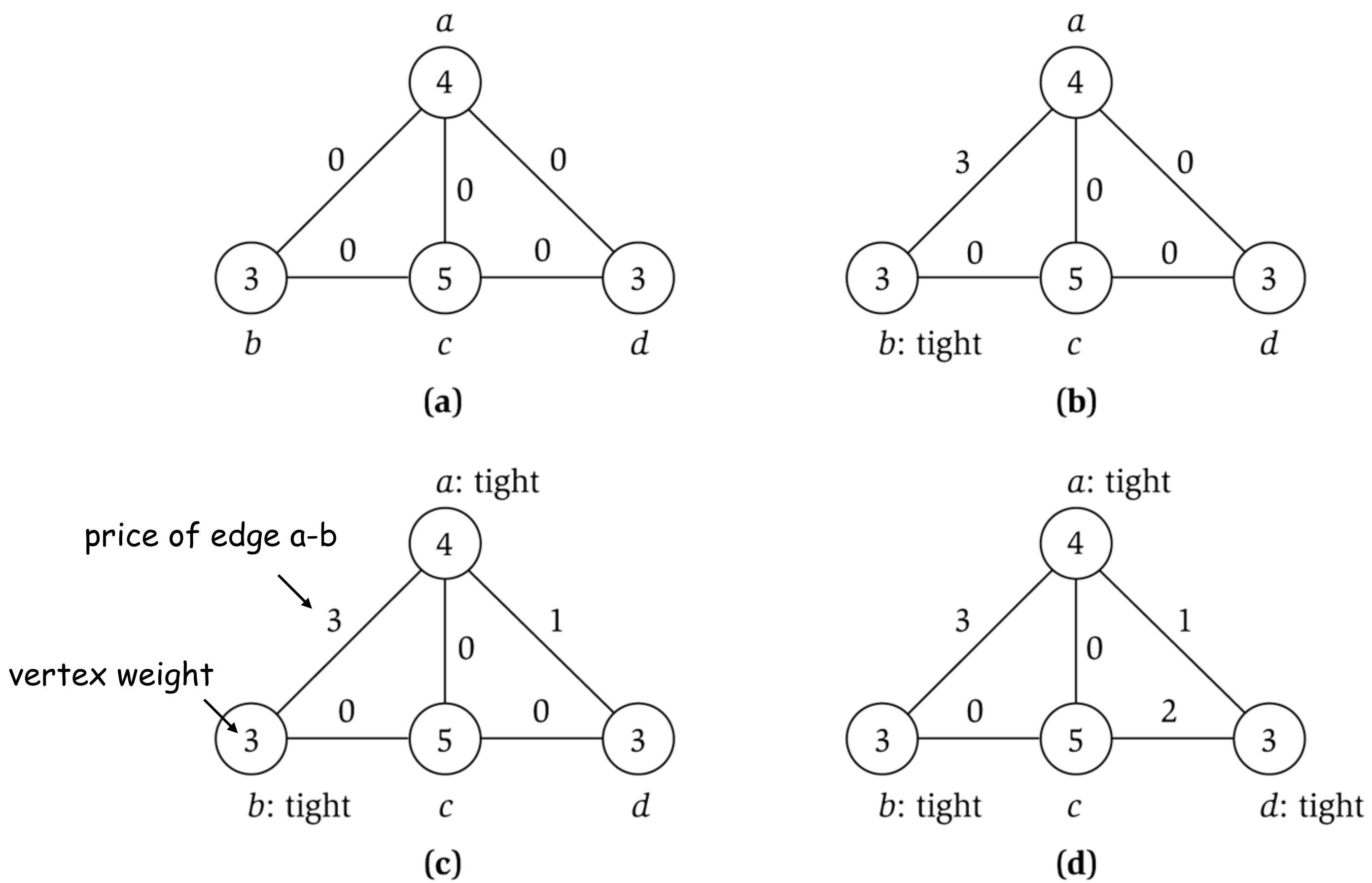
$$\sum_{e=(i,j)} p_e = w_i$$

```
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 without violating fairness
   }

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

Pricing Method



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 $w(S) \leq 2w(S^*)$.

$$w(S) = \underbrace{\sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e}_{\text{all nodes in S are tight}} \leq \sum_{i \in V} \sum_{e=(i,j)} p_e = 2 \sum_{e \in E} p_e \leq 2w(S^*). \quad \blacksquare$$

all nodes in S are tight

$S \subseteq V$, prices ≥ 0

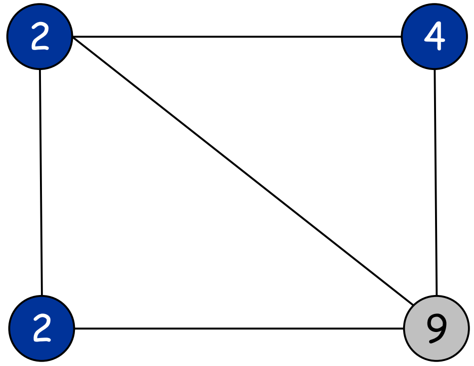
each edge counted twice

fairness lemma

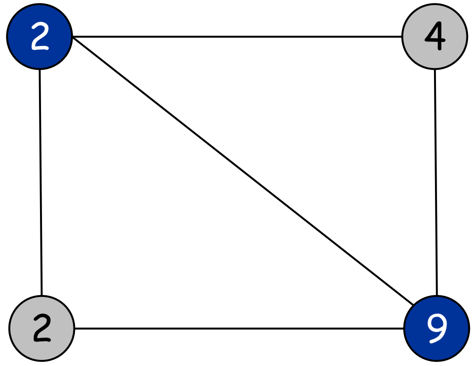
11.6 LP Rounding: Vertex Cover

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.



weight = 2 + 2 + 4



weight = 11

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: minimize $\Sigma_i w_i x_i$.

- For edge (i,j), must take either i or j (or both): $x_i + x_j \geq 1$.

Weighted Vertex Cover: IP Formulation

Weighted vertex cover. Integer programming formulation.

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

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 constant integers c_j, b_i, a_{ij}, find integers x_j that satisfy:

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

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.

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

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.

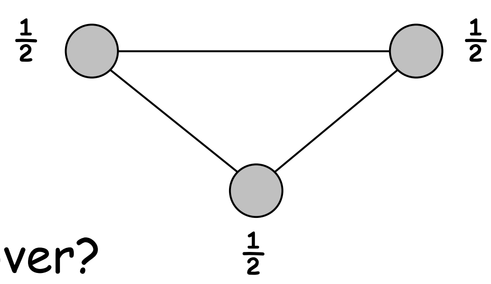
Weighted Vertex Cover: LP Relaxation

Weighted vertex cover. Linear programming formulation.

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

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.

[S has desired cost]

- Let S^* be optimal vertex cover. Then

$$\sum_{i \in S^*} w_i \underbrace{\geq}_{\text{LP is a relaxation}} \sum_{i \in V} w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \underbrace{\geq}_{x^*_i \geq \frac{1}{2}} \tfrac{1}{2} \sum_{i \in S} w_i$$

Weighted Vertex Cover

Theorem. 2-approximation algorithm for weighted vertex cover.

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

$10\sqrt{5} - 21$

Open research problem. Close the gap.

11.8 Knapsack Problem

Polynomial Time Approximation Scheme

PTAS. $(1 + \varepsilon)$-approximation algorithm for any constant $\varepsilon > 0$.

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

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

Knapsack Problem

Knapsack problem.

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

Ex: { 3, 4 } has value 40.

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

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

Running time. O(n W).

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

Knapsack Problem: Dynamic Programming II

Def. 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

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

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

- V^* = maximum possible total value $\leq n\, v_{max}$
- Not polynomial in input size!

Knapsack: PTAS

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	1,342,211	1
2	6,563,429	2
3	18,100,134	5
4	22,217,800	6
5	28,343,199	7

W = 11

original instance

Item	Value	Weight
1	2	1
2	7	2
3	19	5
4	23	6
5	29	7

W = 11

rounded instance

Knapsack: PTAS

Knapsack PTAS. Round up all values:

- v_{max} = largest value in original instance
- ε = precision parameter
- θ = scaling factor = $\varepsilon\, v_{max} / 2n$

$$\bar{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta \qquad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil$$

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

$$\hat{v}_{\max} = \left\lceil \frac{v_{\max}}{\theta} \right\rceil = \left\lceil \frac{2n}{\varepsilon} \right\rceil$$

Knapsack: PTAS

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

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

Pf. Let S* be any feasible solution satisfying weight constraint.

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

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

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

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

$$= \sum_{i \in S} v_i + \frac{1}{2} \varepsilon v_{\max} \qquad \theta = \varepsilon v_{max} / 2n$$

$$\leq (1+\varepsilon) \sum_{i \in S} v_i \qquad v_{max} \leq 2 \Sigma_{i \in S} v_i$$

Choosing $S^* = \{ max \}$

$$v_{max} \leq \sum_{i \in S} v_i + \frac{1}{2} \varepsilon \, v_{max}$$

$$\leq \sum_{i \in S} v_i + \frac{1}{2} v_{max}$$

Thus

$$v_{max} \leq 2 \sum_{i \in S} v_i$$

Chapter Summary

Approximation Algorithms

To solve an NP-hard problem, must sacrifice one of three desired features.

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

ρ-approximation algorithm.

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

Outline

Example problems

- Load Balancing
  - Greedy algorithm is a 3/2-approximation
- Center Selection
  - Greedy algorithm is a 2-approximation
- Weighted Vertex Cover
  - Pricing method is a 2-approximation
  - Linear programming + rounding is a 2-approximation
- Knapsack Problem
  - Polynomial Time Approximation Scheme: $(1 + \varepsilon)$-approximation algorithm for any constant $\varepsilon > 0$
  - Rounding and scaling