CSE 6140/ CX 4140
Computational Science and Engineering
ALGORITHMS

Review - Coping with NPC

Instructor: Xiuwei Zhang
Assistant Professor
School of Computational Science and Engineering

End of semester!

Almost there…

Test3: Dec. 2, Wednesday, 9am - 11:59pm

Project: Dec. 4, Friday, due 11:59pm

No copying of whole sentences/paragraphs from papers or chunks of code from online resources

Office hours during the following weeks:

Week of 11/23: Thursday and Friday are holiday

Week of 11/30: normal OH

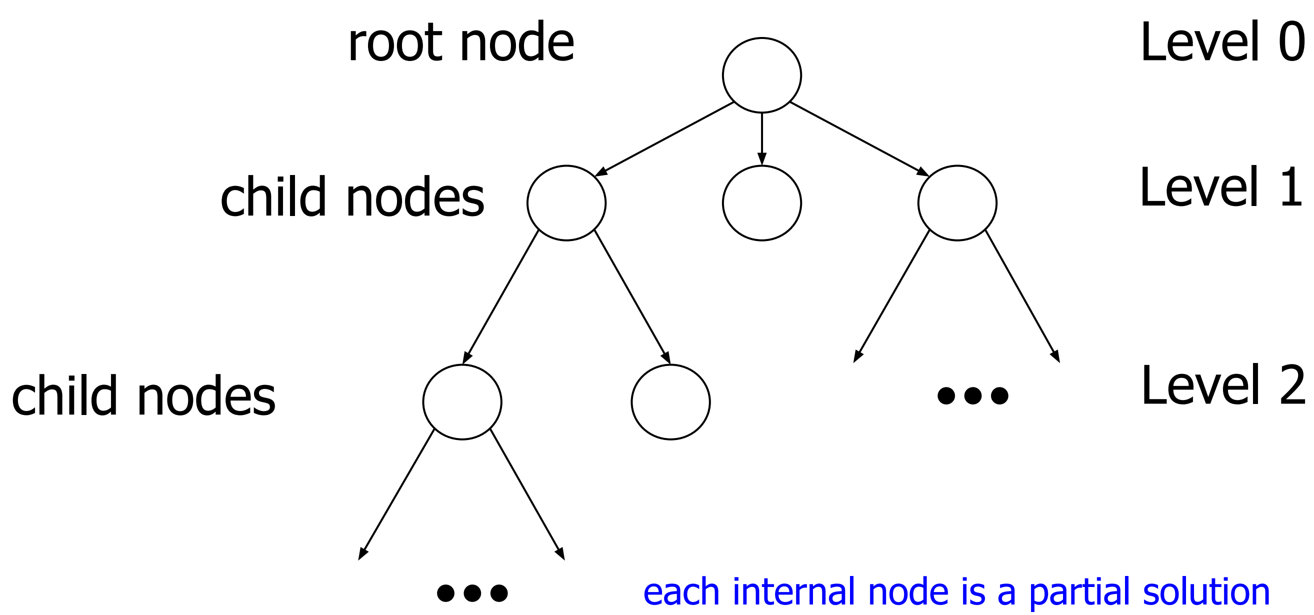
Please refer to Canvas->Calendar

Dealing with NP-complete problems

Branch & bound, Backtracking	Sacrifice running time: create an algorithm with running time exponential in the input size (but which might do well on the inputs you use) Advantage: Guarantee optimal solution when the algorithm finishes Disadvantage: Running time can be exponential
Local search	Quickly find a solution for which you cannot give any quality guarantee (but which might often be good in practice on real problem instances) Advantage: returns increasingly better feasible solutions quickly Disadvantage: not guaranteed to return optimal solution
Approximation	Sacrifice quality of the solution: quickly find a solution that is *provably not very bad* Advantage: worst-case running time polynomial; provides a bound on the quality of the solution Disadvantage: not guaranteed to return optimal solution

Backtracking

- Constructs solutions component by component (grows a partial solution)
- This processing is often implemented by constructing a tree of choices being made, called the ***state-space tree***.
- **Root**: **initial state** before the search for a solution begins.
- **Nodes of the first level** in the tree: the **choices made for the first component** of a solution
- The nodes of the second level represent the choices for the second component, and so on.
- **Leaves**: “dead end” or “solution found”

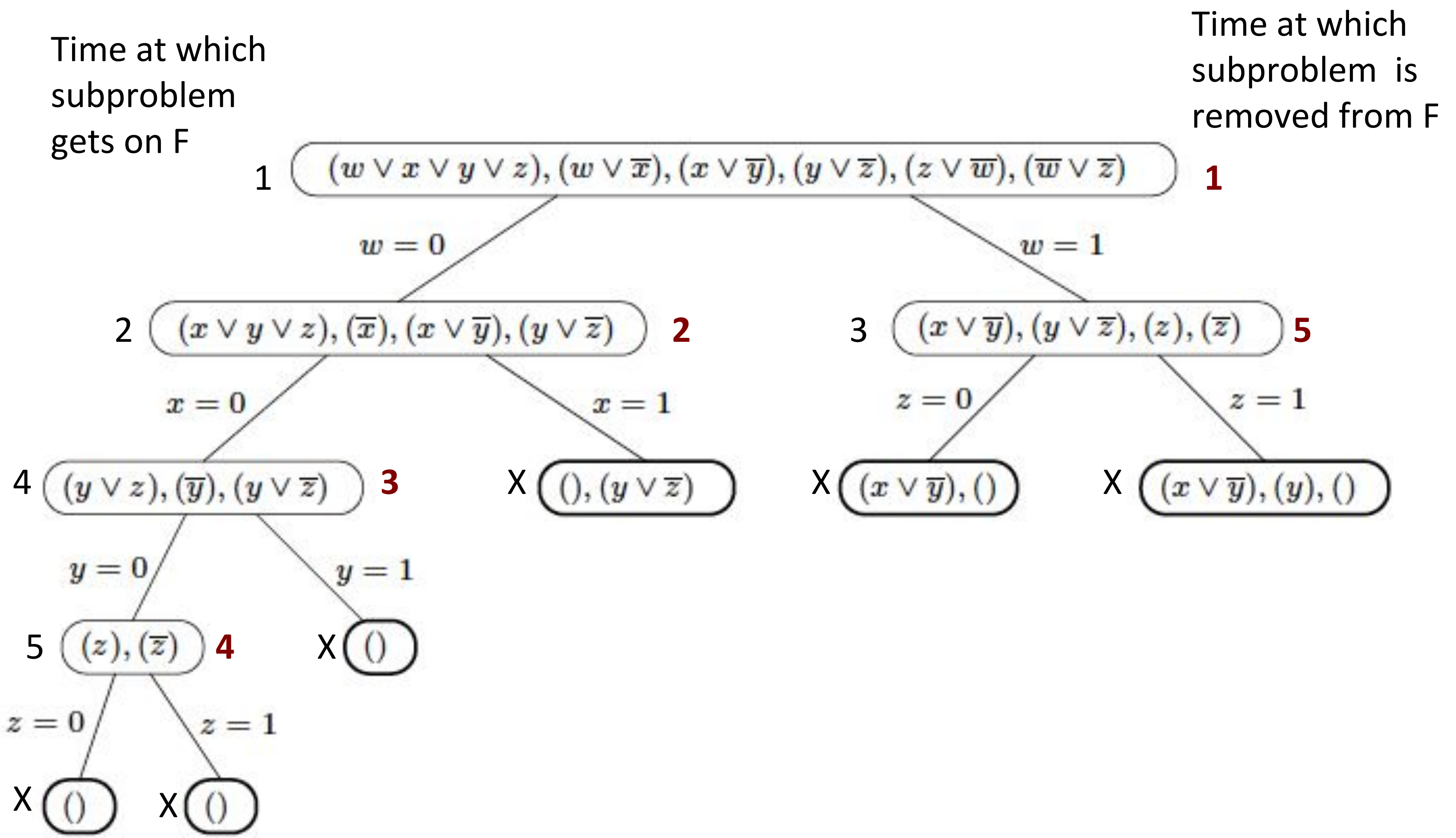


Backtracking

```
Backtracking(P) // Input: problem P
F ← {(∅,P)}     // Frontier set of configurations
while F ≠ ∅ do
  Choose (X,Y)∈ F – the most "promising" configuration
  Expand (X,Y), to candidate extensions (new choices)
  Let (X_1,Y_1), (X_2,Y_2), ..., (X_k,Y_k) be extended candidates
  for each new configuration (X_i,Y_i) do
     "Check" (X_i,Y_i)
     if "solution found" then
        return the solution derived from (X_i,Y_i)
     if not "dead end" then
        F ← F ∪ {(X_i,Y_i)}
  // else nothing to expand from
return "no solution"
```

(X,Y) associated with each node, where X is a partial solution, and Y is the remaining subproblem

Satisfiability



Unsatisfiable formula

Branch-and-Bound

- Find optimal solution to optimization problem, by exploring the whole *tree* of solutions

- Assuming it is a minimization problem:
- Upper bound: keep track of BEST solution found so far
- Lower bound (LB): for each node (partial solution), computes a LB on the value of the objective function for all descendants of the node (extensions of the partial solution)
  - Use LB for:
    - Ruling out certain nodes as "nonpromising" to prune the tree – if a node's bound is not better than the best solution seen so far
    - Guiding the search through state-space as a measure of "promise"

Branch-and-Bound algorithm

```
Branch-and-Bound(P) // Input: minimization problem P
F <- {(∅,P)}    // Frontier set of configurations
B <- (+∞ , (∅,P)) ; UB <- +∞    // Best cost and solution
while F not empty do
  Choose (X,Y) in F – the most "promising" configuration
  Expand (X,Y), by making a choice(s)
  Let (X1,Y1), (X2,Y2), ..., (Xk,Yk) be the new configurations
  for each new configuration (Xi,Yi) do
    "Check" (Xi,Yi)
    if "solution found" then // a feasible solution is found
      if cost(Xi) < UB cost then // update upper bound UB
        B <- (cost(Xi),(Xi,Yi)); UB <- cost(Xi)
    if not "dead end" then
      if LB(Xi) < UB cost then // check lower bound
        F <- F U {(Xi,Yi)}   // else prune by LB
return B
```

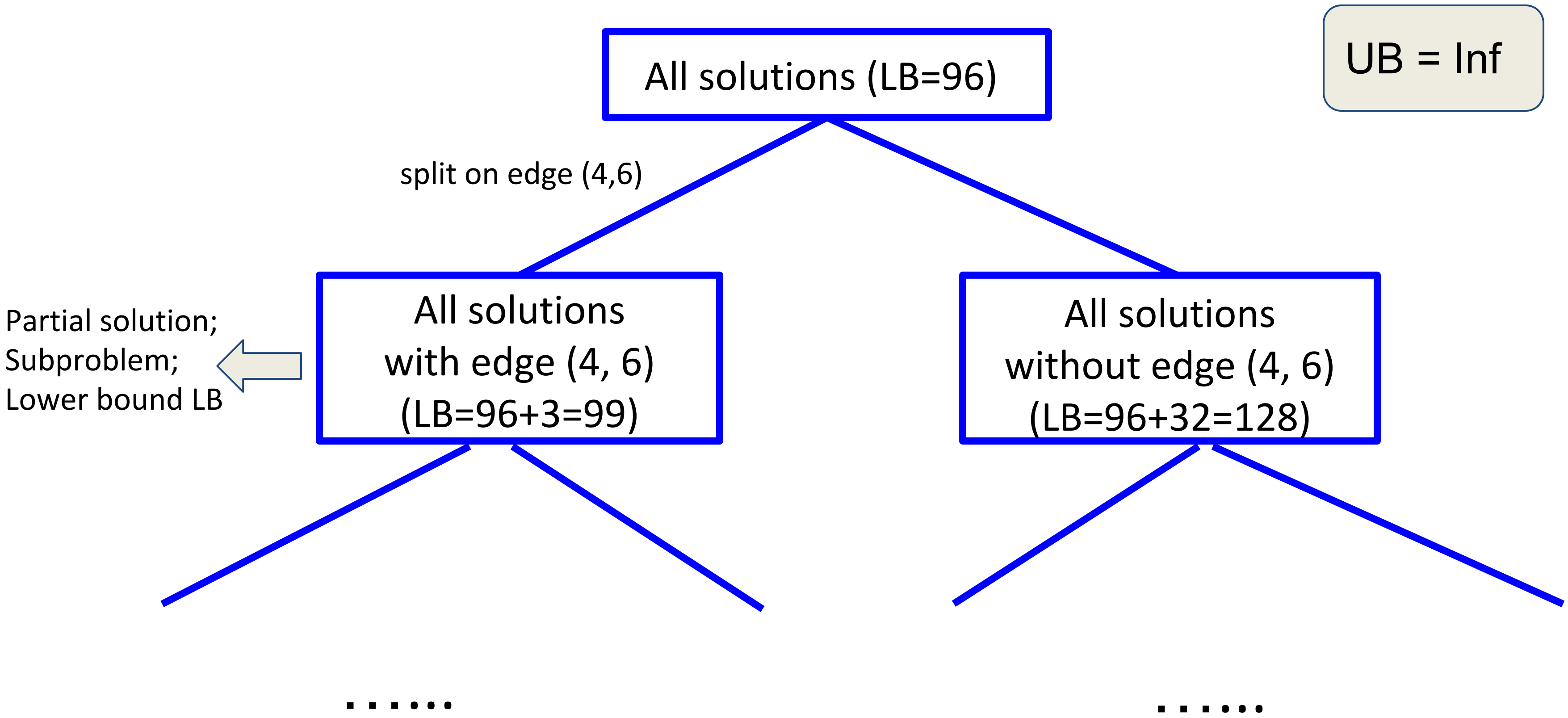
Possible results of the "check" step:

1. feasible (complete) solution
2. LB > UB: prune the subtree
3. LB < UP: add to frontier

Leaves in the search tree:
"dead end"(pruned subtree); feasible but not optimal solutions; optimal solutions.

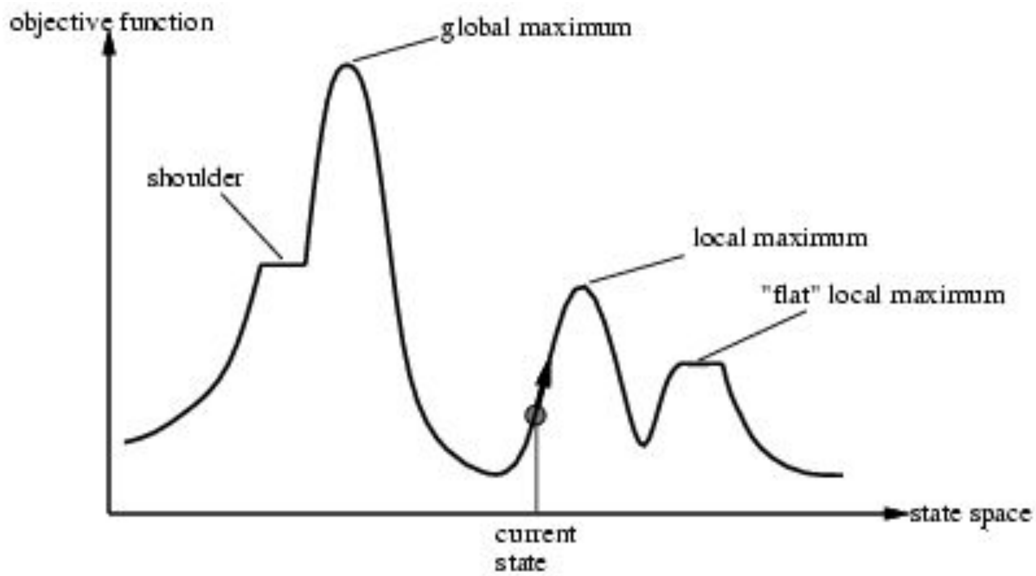
Branch and bound: example (TSP bound with reduced cost matrix)

- Total cost reduced: 84+7+1+4 = 96 (LB) decision tree:



Local search algorithms

- Start from initial position
- Iteratively move from current position to one of neighboring positions (neighborhood relationship)
- Use evaluation function to choose among neighboring positions



Different LS techniques

- Hill climbing
  - First-improvement or best-improvement neighbor as the next state
- Stochastic local search
  - Randomize <u>initialization</u> step
  - Randomize search steps such that <u>suboptimal/worsening steps</u> are allowed
- Simulated annealing
  - Temperature decreasing with time
  - More worsening steps with high temperature
- Tabu search
  - Short-term memory to avoid revisiting previous states
- Iterated local search
  - Perturbations on initial state s, several local searches

Approximation Algorithms

- Approximation algorithms guarantee performance bounds – i.e. a bound on the worst deviation from the optimal quality
- **Ratio bound** (approximation factor)
  - Assume solution costs are positive
  - Given input X of size n, OPT(X) is optimum, A(X) is solution quality produced by algorithm A

$$\max\left(\frac{A(X)}{OPT(X)}, \frac{OPT(X)}{A(X)}\right) \leq \rho(n)$$

- For minimization

$$OPT(X) \leq A(X) \leq \rho(n) OPT(X)$$

- In general, $\rho(n) \geq 1$, and if it does not depend on n, we have **constant factor approximation**

Approximation Algorithm Examples

Load Balancing

List scheduling: 2 approximation

Longest processing time (LPT): 4/3 approximation

Clustering (center selection):

Greedy algorithm: 2 approximation

How to prove an approx. ratio

Quality of optimal solution: OPT

Quality of the approx. algorithm: A

Assume a minimization problem, and prove an approx. ratio ρ:

To show: A ≤ ρ * OPT

Derive a lower bound for OPT: OPT ≥ some quantity x

Derive an upper bound for A: A ≤ some quantity y

...

A ≤ ρ * OPT

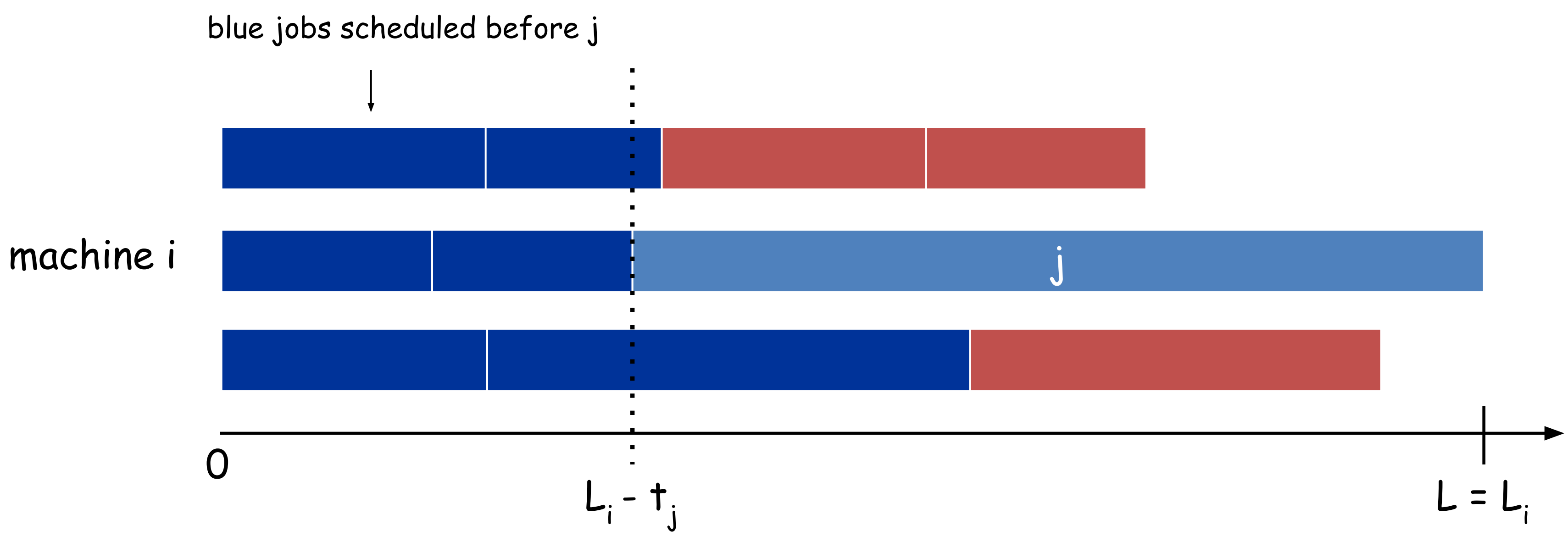
Load Balancing: List Scheduling Analysis

Theorem. Greedy algorithm (list scheduling) is a 2-approximation.

Notations: m machines; n jobs, job j has processing time t_j.

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

Lemma 2. The optimal makespan $L^* \geq (\Sigma_j t_j)/m$



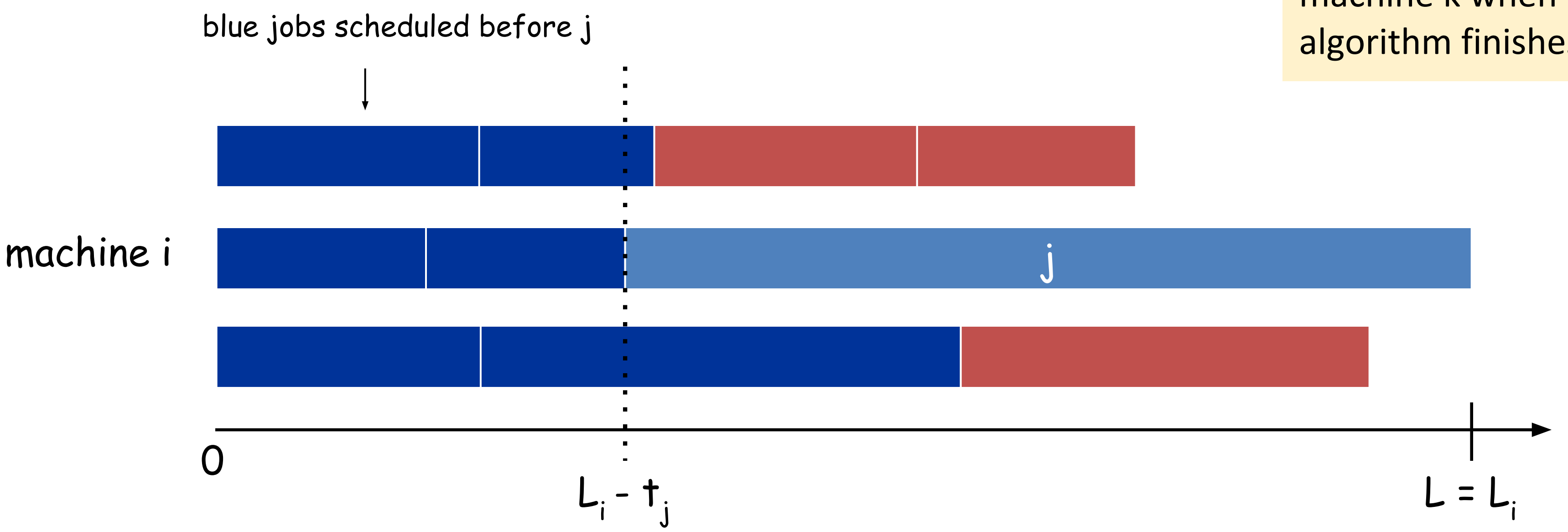
Load Balancing: List Scheduling Analysis

Theorem. Greedy algorithm (list scheduling) is a 2-approximation.

Pf. Consider load L_i of bottleneck machine i.

- Let j be the last job scheduled on machine i.
- When job j assigned to machine i, i had smallest load. Its load before assignment is $L_i - t_j \Rightarrow \boxed{L_i - t_j \leq L_k \text{ for all } 1 \leq k \leq m}$.

L_k is the load on machine k when the algorithm finishes



Load Balancing: List Scheduling Analysis

Theorem. Greedy algorithm is a 2-approximation.

Pf. Consider load L_i of bottleneck machine i.

- Let j be the last job scheduled on machine i.
- When job j assigned to machine i, i had smallest load. Its load before assignment is $L_i - t_j \Rightarrow \boxed{L_i - t_j \leq L_k \text{ for all } 1 \leq k \leq m}$.

Sum inequalities over all k and divide by m:

$$\begin{aligned} L_i - t_j &\leq \tfrac{1}{m} \sum_{k=1..m} L_k \\ &= \tfrac{1}{m} \sum_{s=1..n} t_s \\ &\leq L^* \end{aligned}$$

Lemma 2

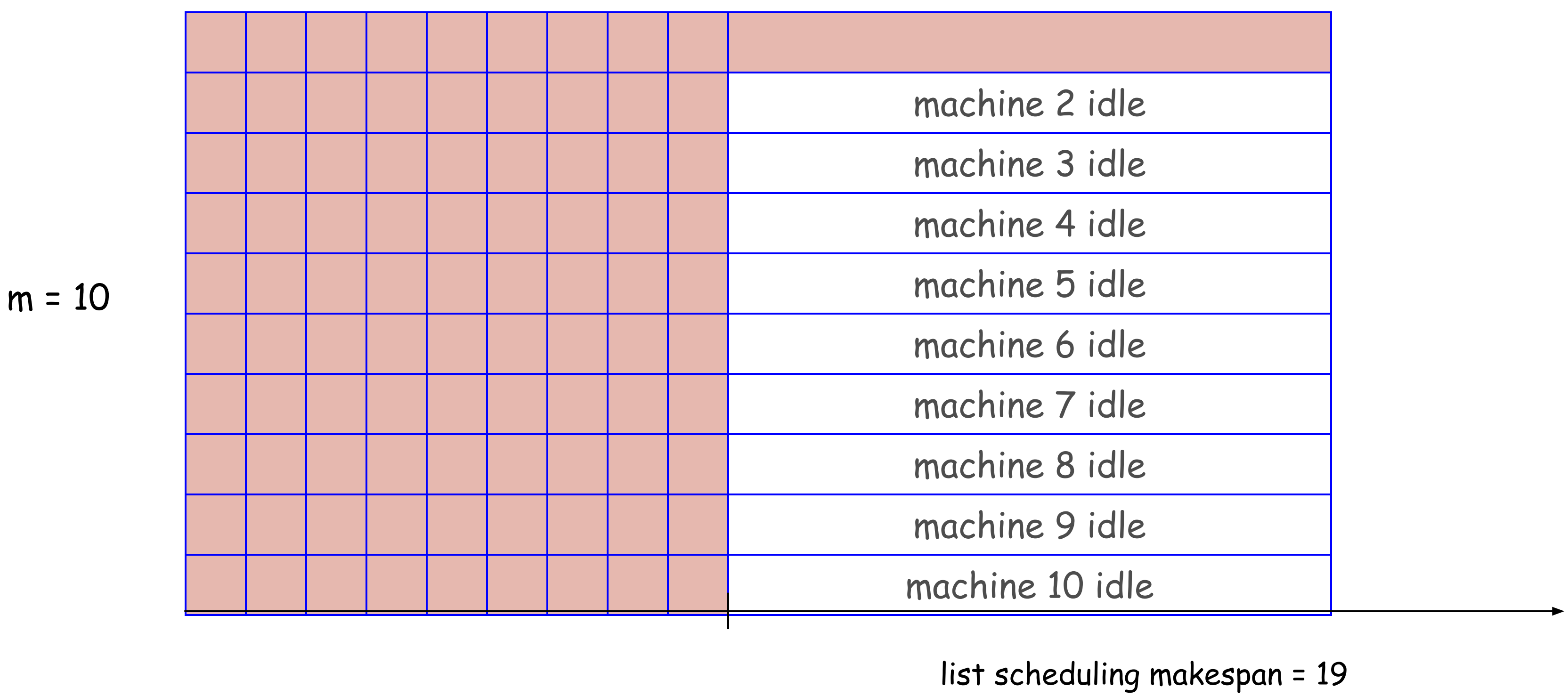
$$L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*$$

Lemma 1

Approx. Algorithms

Given an approx. algorithm, show that the ratio you proved is a tight bound $L_i \leq 2\ L^*$

- Give an example where $L_i = 2\ L^*$ (or infinitely close to $2\ L^*$)



m machines, $n=m(m-1)+1$ jobs; $m(m-1)$ jobs length 1, one job of length m

OPT: m; Approx: 2m-1

Inapproximability

Show there doesn't exist a poly-time approx. algorithm which can give ratio better than 2 OPT (inapproximability proof)

- Idea: reduce a yes/no (decision) NP-complete problem to the optimization problem, and create a gap in the corresponding optimal values between yes and no instances
- Example: center selection problem (k-center) -- there doesn't exist an approx. algorithm for the k-center problem with $\rho < 2$ unless P=NP.

Center Selection: Hardness of Approximation

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

- **Pf**. We show how we could use a (2 - ε) approximation algorithm for k-center to solve DOMINATING-SET in poly-time.
  - Let [G = (V, E), k] be an instance of DOMINATING-SET.
  - Construct instance [G',k'=k] of k-CENTER with sites V and distances
    - d(u, v) = 1 if (u, v) $\in$ E
    - d(u, v) = 2 if (u, v) $\notin$ E
  - If DOMINATING-SET is a yes instance, the optimal solution of k-center is r(C*)=1
  - If there exists an approx algo with ρ < 2, then approx provides r(C) < 2 r(C*) $\longrightarrow$ approx returns r(C) = 1
  - If DOMINATING-SET is a no instance, the optimal solution of k-center is r(C*)=2, approx provides r(C) >= r(C*)=2

Center Selection: Hardness of Approximation

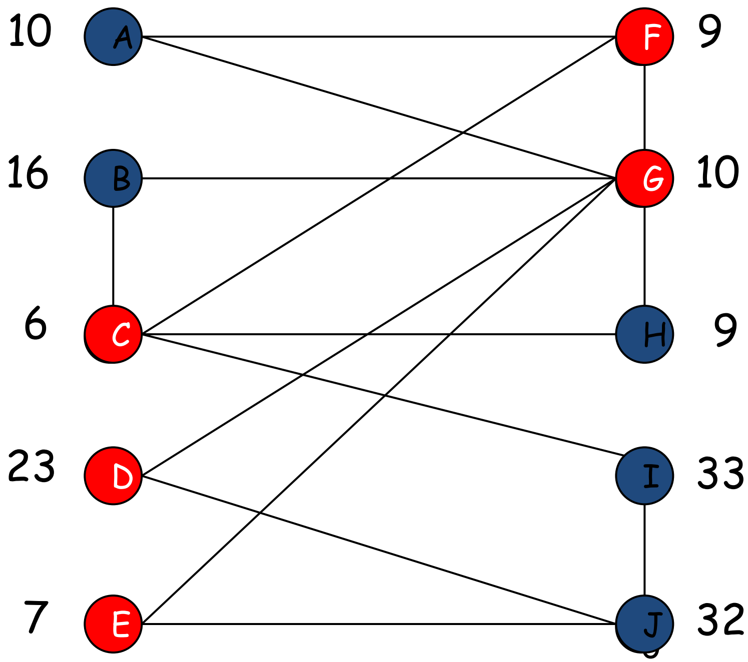
- **Theorem**. Unless P = NP, there is no ρ-approximation algorithm for metric k-center problem for any $\rho < 2$.

- **Pf**. We show how we could use a (2 - ε) approximation algorithm for k-center to solve DOMINATING-SET in poly-time.
  - Let [G = (V, E), k] be an instance of DOMINATING-SET.
  - Construct instance [G',k'=k] of k-CENTER with sites V and distances
    - $d(u, v) = 1$ if $(u, v) \in E$
    - $d(u, v) = 2$ if $(u, v) \notin E$
  - If DOMINATING-SET is a yes instance, the optimal solution of k-center is $r(C^*)=1$
  - If there exists an approx algo with $\rho < 2$, the DOMINATING-SET instance has a solution iff the approx. algorithm for k-center instance returns $r(C) = 1$
  - This means we can solve the DOMINATING-SET problem in polynomial time.

Also called "gap-introducing" reduction

KT 11.6: Weighted Vertex Cover

<u>Weighted vertex cover</u> 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 linear programming formulation

- Model inclusion of each vertex i using a 0/1 variable x_i.

$$x_i = \begin{cases} 0 & \text{if vertex } i \text{ is not in vertex cover} \\ 1 & \text{if vertex } i \text{ is in vertex cover} \end{cases}$$

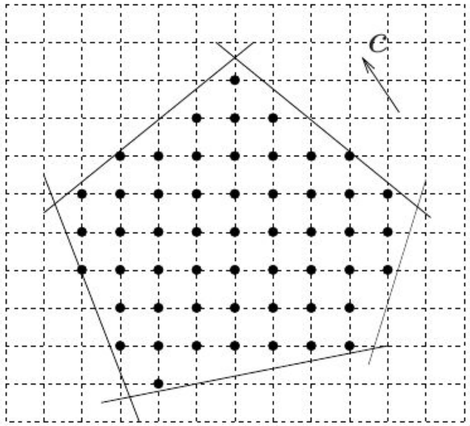
- Vertex covers in 1-1 correspondence with 0/1 assignments: $S = \{i \in V : x_i = 1\}$
- Objective function: minimize $\Sigma_i w_i x_i$.
- Constraints: must take either i or j for each edge (i,j) in E: $x_i + x_j \geq 1$.

Weighted Vertex Cover: ILP Formulation

Weighted vertex cover. Integer linear programming (ILP) formulation.

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

$$
\begin{array}{ll}
\text{minimize} & c^T x \\
\text{subject to} & Ax \leq b \\
& x \in \mathbf{Z}^n
\end{array}
$$



How does ILP help us find the vertex cover

Solving the ILP:

Relax to LP (linear programming)

Linear Programming

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

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

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

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

Weighted Vertex Cover: LP Relaxation

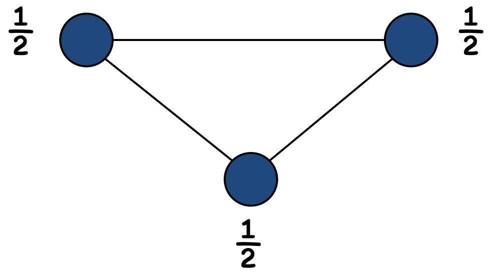
Weighted vertex cover. Linear programming formulation.

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

Observation. Optimal value of (LP) is ≤ optimal value of (ILP).

Pf. LP has fewer constraints. Any solution to ILP is also solution to LP

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: x_i>=1/2 become 1, x_i < ½ become 0

Weighted Vertex Cover

Theorem. If x* is optimal solution to (LP), then $S = \{i \in V : x^*_i \geq ½\}$ is a vertex cover whose weight $\Sigma_{i \in S} w_i$ is at most twice OPT(Vertex Cover).

Pf. [S is a vertex cover]

- Consider an edge $(i, j) \in E$.
- Since $x^*_i + x^*_j \geq 1$, either $x^*_i \geq ½$ or $x^*_j \geq ½ \Rightarrow (i, j)$ covered.

Pf. [S has desired cost, $w(S) <= 2w(S^{VCOPT})$]

- Let S^{VCOPT} be optimal vertex cover. Corresponds to a soln of LP with x_i=1 if i in S^{VCOPT}, and 0 otherwise. Then

$$w(S^{VCOPT}) = \sum_{i \in S^{VCOPT}} w_i 1 \geq \sum_{i \in V} w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \geq \tfrac{1}{2} \sum_{i \in S} w_i = \tfrac{1}{2} w(S)$$

soln corresponding to S^{VCOPT} cannot be better than opt LP solution x*, since LP is a relaxation

Drop i with $x^*_i < \frac{1}{2}$, Keep $x^*_i >= \frac{1}{2}$

$x^*_i \geq \frac{1}{2}$ For all i in S

Theorem. 2-approximation algorithm for weighted vertex cover.

Requirements

- Be able to design B&B and LS algorithms, understand the advantages/disadvantages of different approaches

- Know and understand definitions of approximation algorithms, be able to design an approx. algo and prove the approx. ratio.
- Be able to write an ILP for a problem
- Know how to prove an inapproximability result

Thank you

- We appreciate your feedback:
  - Course/instructor opinion survey (CIOS) to fill
  - “Thank a teacher” http://thankateacher.gatech.edu/
- Wish you all the best for the future!