Chapter 10, 11,12

Extending the Limits of Tractability

Slides by Kevin Wayne.

Coping With NP-Completeness

Q. Suppose I need to solve an NP-complete problem. What should I do?
A. Theory says you're unlikely to find poly-time algorithm.

Must sacrifice one of three desired features.

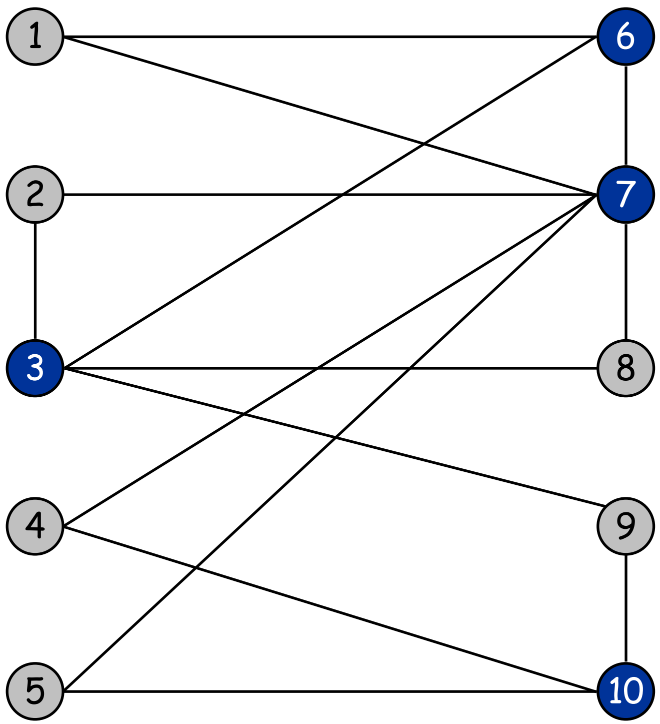
- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

This lecture. Solve some special cases of NP-complete problems that arise in practice.

10.1 Finding Small Vertex Covers

Vertex Cover

VERTEX COVER: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge (u, v) either $u \in S$, or $v \in S$, or both.



$k = 4$
$S = \{ 3, 6, 7, 10 \}$

Finding Small Vertex Covers

Q. What if k is small?

Brute force. $O(k\, n^{k+1})$.

- Try all $C(n, k) = O(n^k)$ subsets of size k.
- Takes $O(k\, n)$ time to check whether a subset is a vertex cover.

Goal. Limit exponential dependency on k, e.g., to $O(2^k\, k\, n)$.

Ex. $n = 1{,}000$, $k = 10$.
Brute. $k\, n^{k+1} = 10^{34} \Rightarrow$ infeasible.
Better. $2^k\, k\, n = 10^7 \Rightarrow$ feasible.

Remark. If k is a constant, algorithm is poly-time; if k is a small constant, then it's also practical.

Finding Small Vertex Covers

Claim. Let u-v be an edge of G. G has a vertex cover of size $\leq$ k iff at least one of $G - \{ u \}$ and $G - \{ v \}$ has a vertex cover of size $\leq$ k-1.

delete v and all incident edges

Pf. $\Rightarrow$

- Suppose G has a vertex cover S of size $\leq$ k.
- S contains either u or v (or both). Assume it contains u.
- $S - \{ u \}$ is a vertex cover of $G - \{ u \}$.

Pf. $\Leftarrow$

- Suppose S is a vertex cover of $G - \{ u \}$ of size $\leq$ k-1.
- Then $S \cup \{ u \}$ is a vertex cover of G. ▪

Claim. If G has a vertex cover of size k, it has $\leq$ k(n-1) edges.

Pf. Each vertex covers at most n-1 edges. ▪

Finding Small Vertex Covers: Algorithm

Claim. The following algorithm determines if G has a vertex cover of size $\leq k$ in $O(2^k kn)$ time.

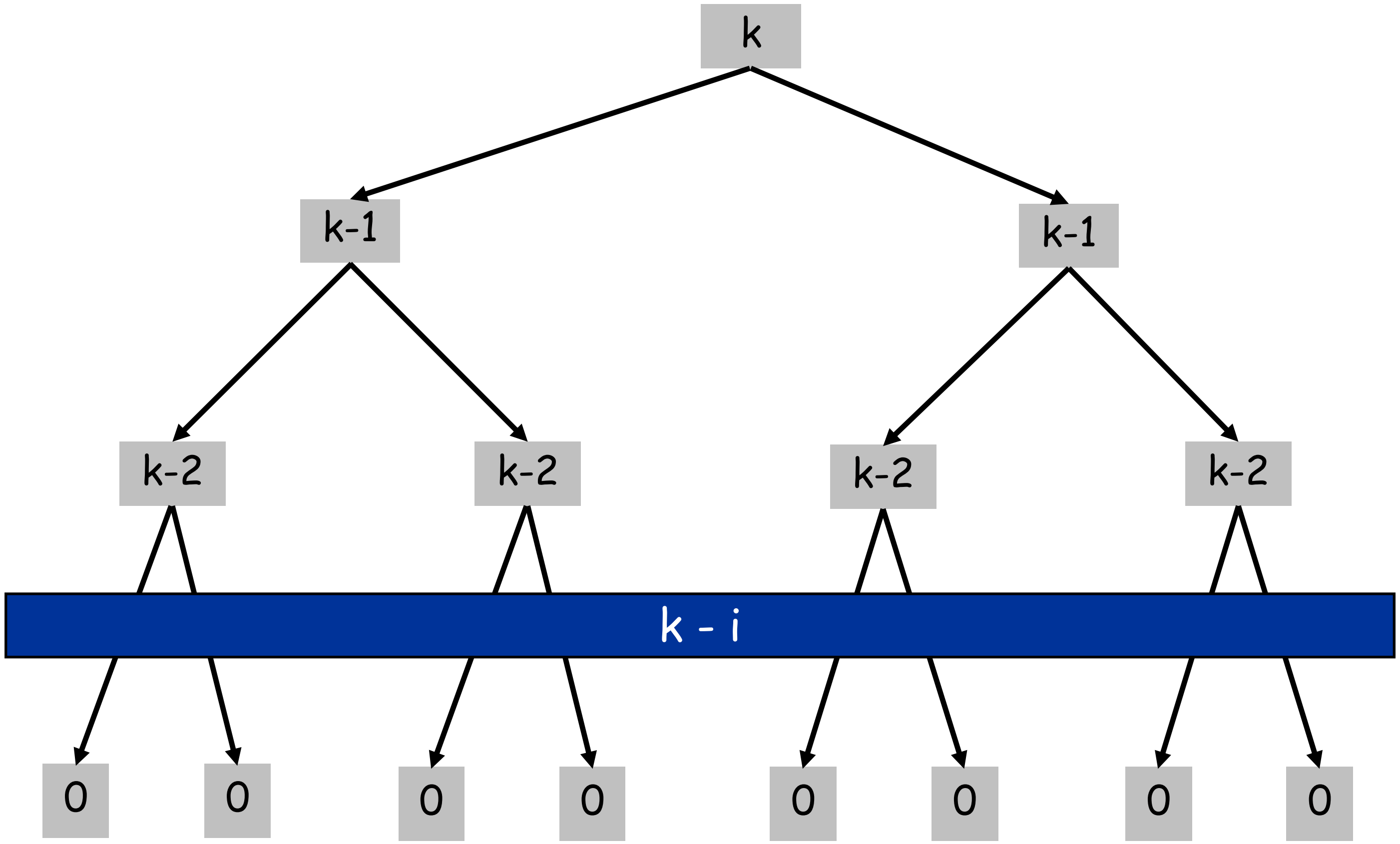
```
boolean Vertex-Cover(G, k) {
   if (G contains no edges)   return true
   if (G contains ≥ kn edges) return false

   let (u, v) be any edge of G
   a = Vertex-Cover(G - {u}, k-1)
   b = Vertex-Cover(G - {v}, k-1)
   return a or b
}
```

Pf.

- Correctness follows previous two claims.
- There are $\leq 2^{k+1}$ nodes in the recursion tree; each invocation takes $O(kn)$ time. ▪

Finding Small Vertex Covers: Recursion Tree

$$T(n, k) \le \begin{cases} cn & \text{if } k = 1 \\ 2T(n, k-1) + ckn & \text{if } k > 1 \end{cases} \quad \Rightarrow \quad T(n, k) \le 2^k c\,k\,n$$



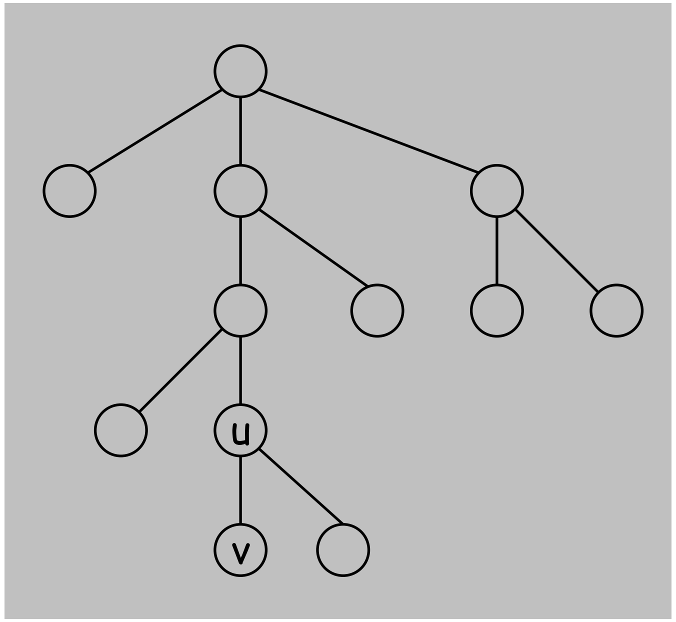
10.2 Solving NP-Hard Problems on Trees

Independent Set on Trees

Independent set on trees. Given a tree, find a maximum cardinality subset of nodes such that no two share an edge.

Fact. A tree on at least two nodes has at least two leaf nodes.

degree = 1



Key observation. If v is a leaf, there exists a maximum size independent set containing v.

Pf. (exchange argument)

- Consider a max cardinality independent set S.
- If $v \in S$, we're done.
- If $u \notin S$ and $v \notin S$, then $S \cup \{v\}$ is independent $\Rightarrow$ S not maximum.
- IF $u \in S$ and $v \notin S$, then $S \cup \{v\} - \{u\}$ is independent. ▪

Independent Set on Trees: Greedy Algorithm

Theorem. The following greedy algorithm finds a maximum cardinality independent set in forests (and hence trees).

```
Independent-Set-In-A-Forest(F) {
   S ← φ
   while (F has at least one edge) {
      Let e = (u, v) be an edge such that v is a leaf
      Add v to S
      Delete from F nodes u and v, and all edges
         incident to them.
   }
   return S
}
```

Pf. Correctness follows from the previous key observation. ▪

Remark. Can implement in $O(n)$ time by considering nodes in postorder.

Weighted Independent Set on Trees

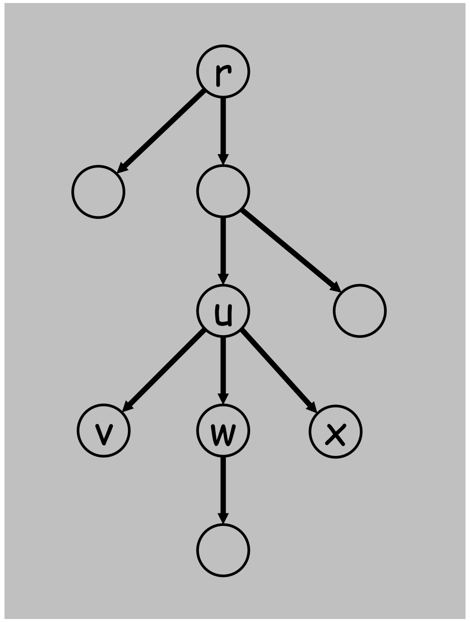
Weighted independent set on trees. Given a tree and node weights $w_v > 0$, find an independent set S that maximizes $\Sigma_{v \in S}\, w_v$.

Observation. If (u, v) is an edge such that v is a leaf node, then either OPT includes u, or it includes all leaf nodes incident to u.

Dynamic programming solution. Root tree at some node, say r.

- $OPT_{in}(u)$ = max weight independent set rooted at u, containing u.
- $OPT_{out}(u)$ = max weight independent set rooted at u, not containing u.

$$\begin{aligned} OPT_{in}(u) &= w_u + \sum_{v \,\in\, \text{children}(u)} OPT_{out}(v) \\ OPT_{out}(u) &= \sum_{v \,\in\, \text{children}(u)} \max\left\{OPT_{in}(v),\ OPT_{out}(v)\right\} \end{aligned}$$



children(u) = { v, w, x }

Independent Set on Trees: Greedy Algorithm

Theorem. The dynamic programming algorithm find a maximum weighted independent set in trees in O(n) time.

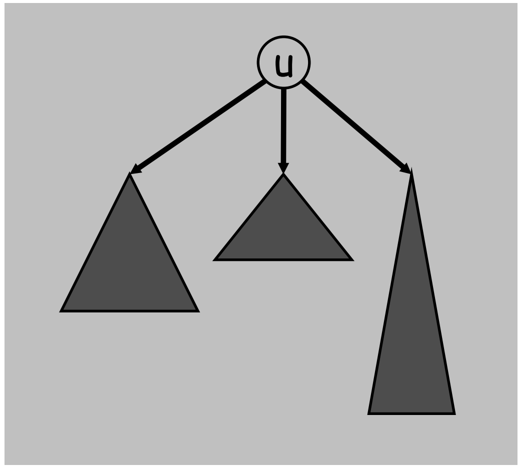
```
Weighted-Independent-Set-In-A-Tree(T) {
   Root the tree at a node r
   foreach (node u of T in postorder) {
      if (u is a leaf) {
         M_in [u] = w_u
         M_out[u] = 0
      }
      else {
         M_in [u] = Σ_v∈children(u) M_out[v]  +  w_v
         M_out[u] = Σ_v∈children(u) max(M_out[v], M_in[v])
      }
   }
   return max(M_in[r], M_out[r])
}
```

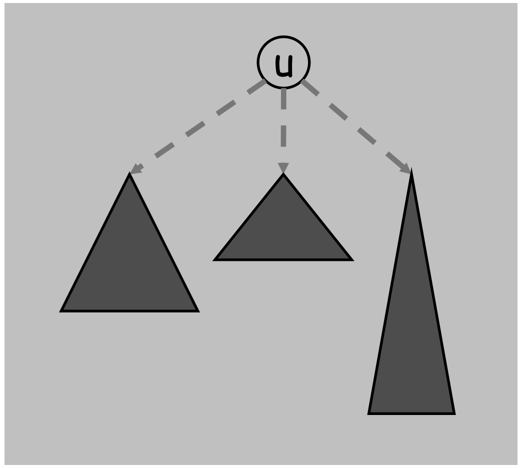
ensures a node is visited after all its children

Pf. Takes O(n) time since we visit nodes in postorder and examine each edge exactly once. ▪

Context

Independent set on trees. This structured special case is tractable because we can find a node that breaks the communication among the subproblems in different subtrees.





see Chapter 10.4, but proceed with caution

Graphs of bounded tree width. Elegant generalization of trees that:

- Captures a rich class of graphs that arise in practice.
- Enables decomposition into independent pieces.

Extra Slides

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: 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.

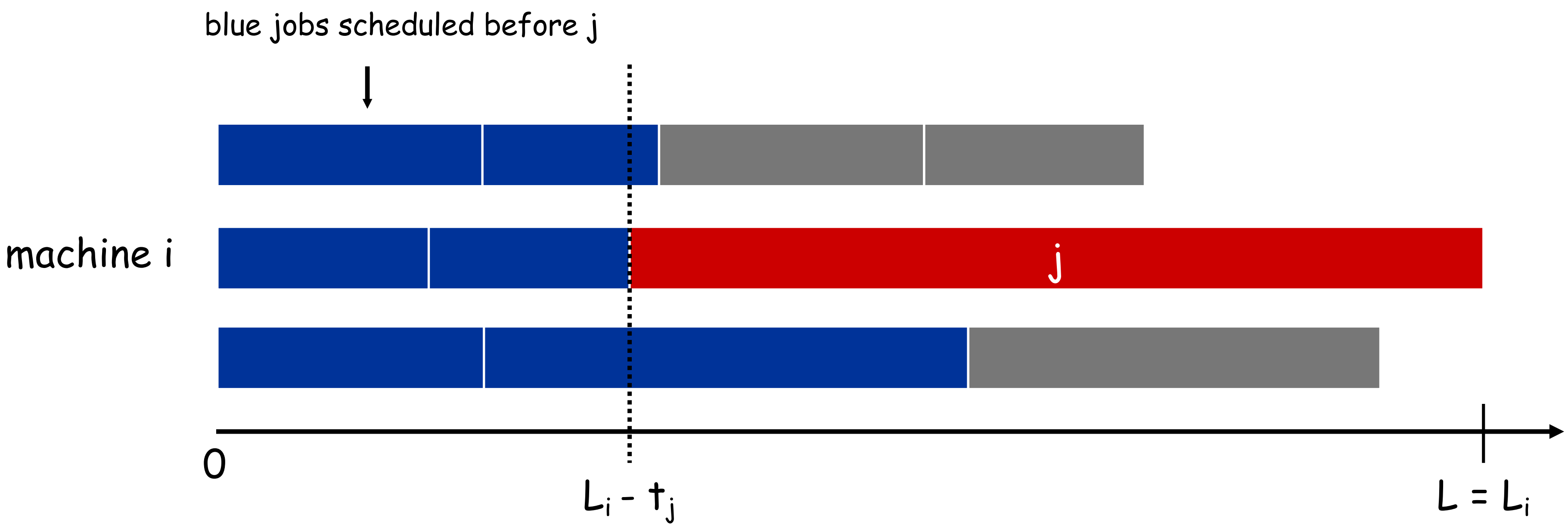
Not very strong lower bound. What if one job is very big and others are small jobs ? ▪

Load Balancing: List Scheduling Analysis

Theorem. Greedy algorithm is a 2-approximation.

Pf. Consider load L_i of bottleneck machine i.

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



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:

$$
\begin{aligned}
L_i - t_j &\le \tfrac{1}{m} \sum_k L_k \\
&= \tfrac{1}{m} \sum_j t_j \\
\text{Lemma 2} \longrightarrow \quad &\le L^*
\end{aligned}
$$

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

Lemma 1

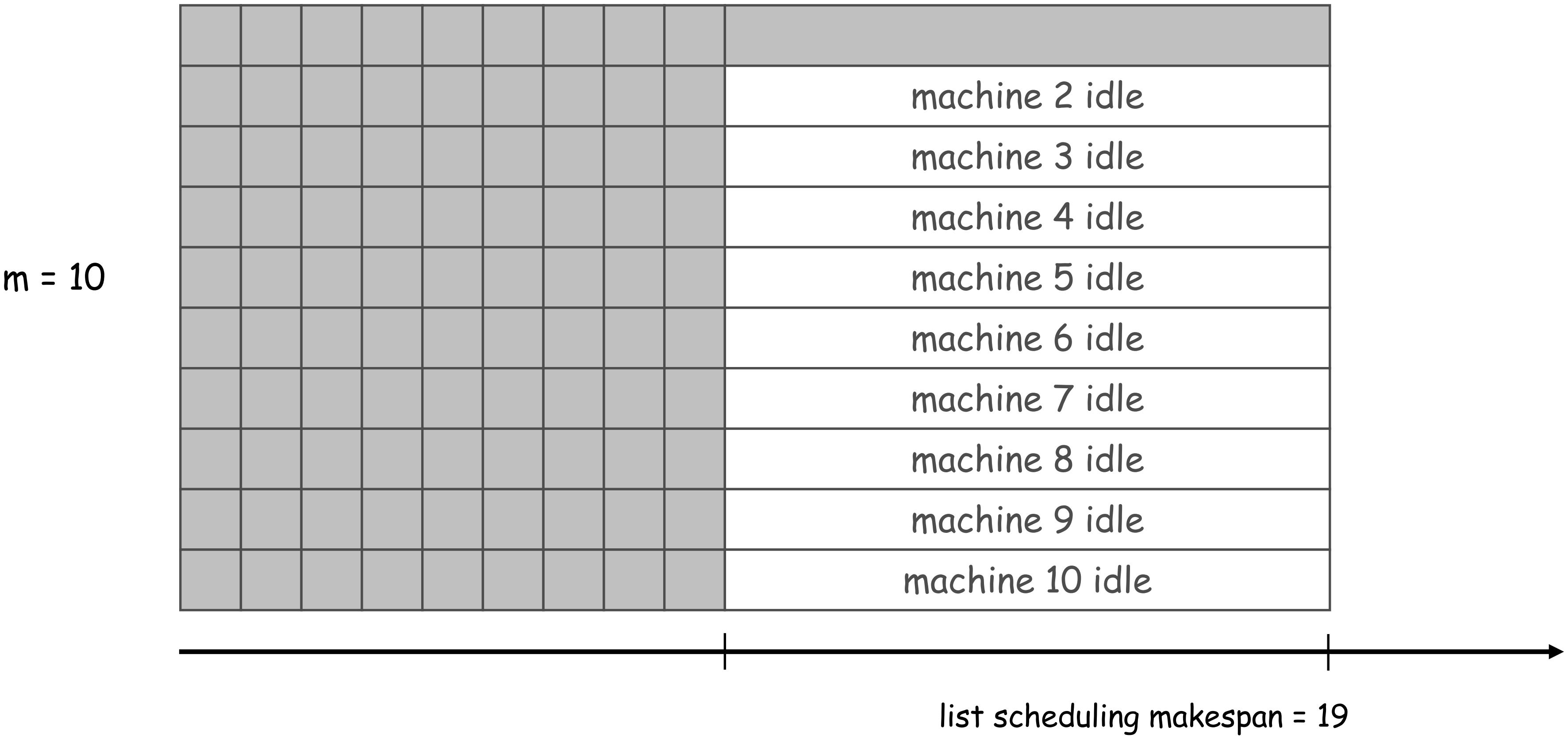
- The solution attained by the greedy algorithm is less 2 times the optimal solution

Load Balancing: List Scheduling Analysis

Q. Is our analysis tight?

A. Essentially yes.

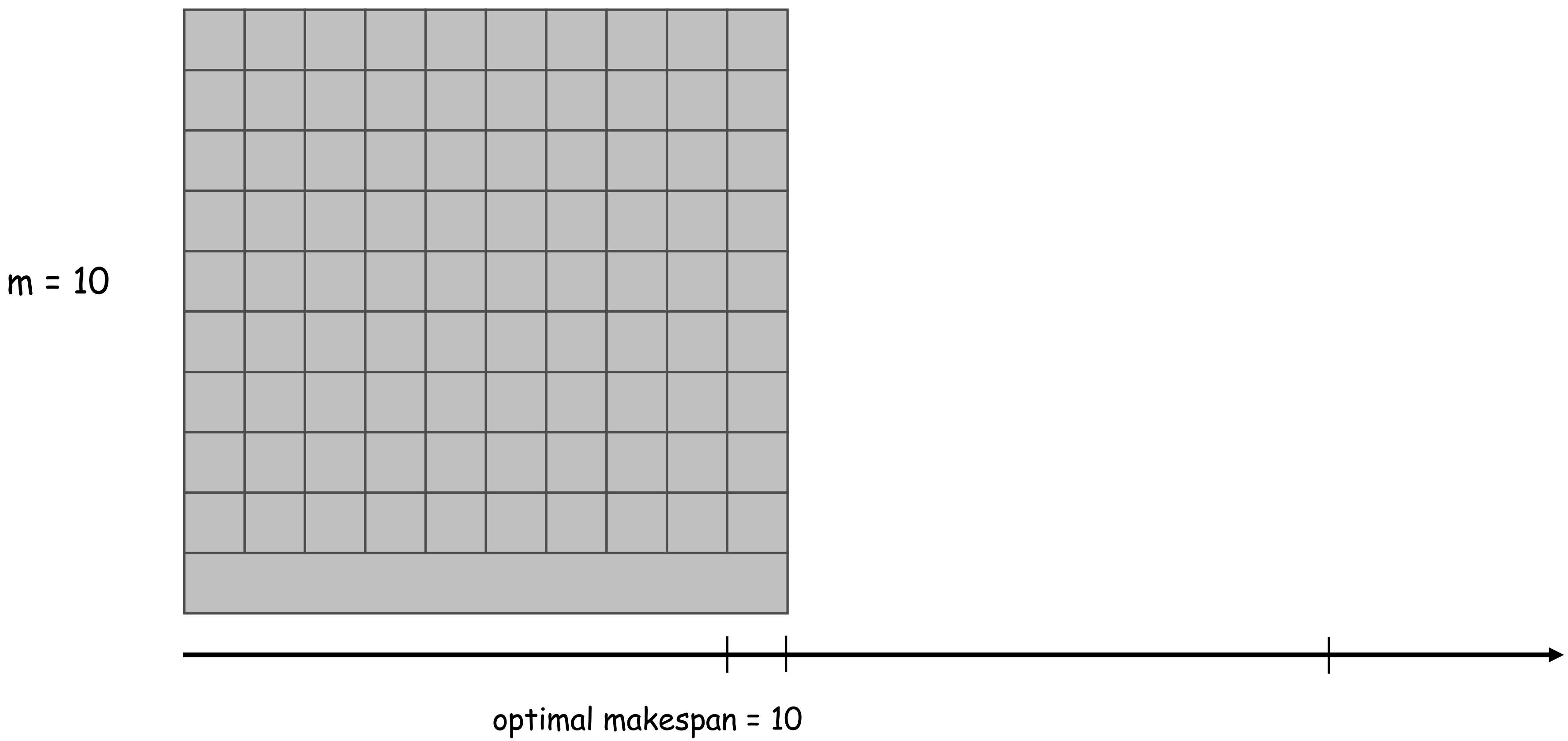
Ex: m machines, m(m-1) jobs length 1 jobs, one job of length m



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 = 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. Same basic approach as for list scheduling.

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

Lemma 3
(by observation, can assume number of jobs > m)

Coping With NP-Hardness

Q. Suppose I need to solve an NP-hard problem. What should I do?
A. Theory says you're unlikely to find poly-time algorithm.

Must sacrifice one of three desired features.

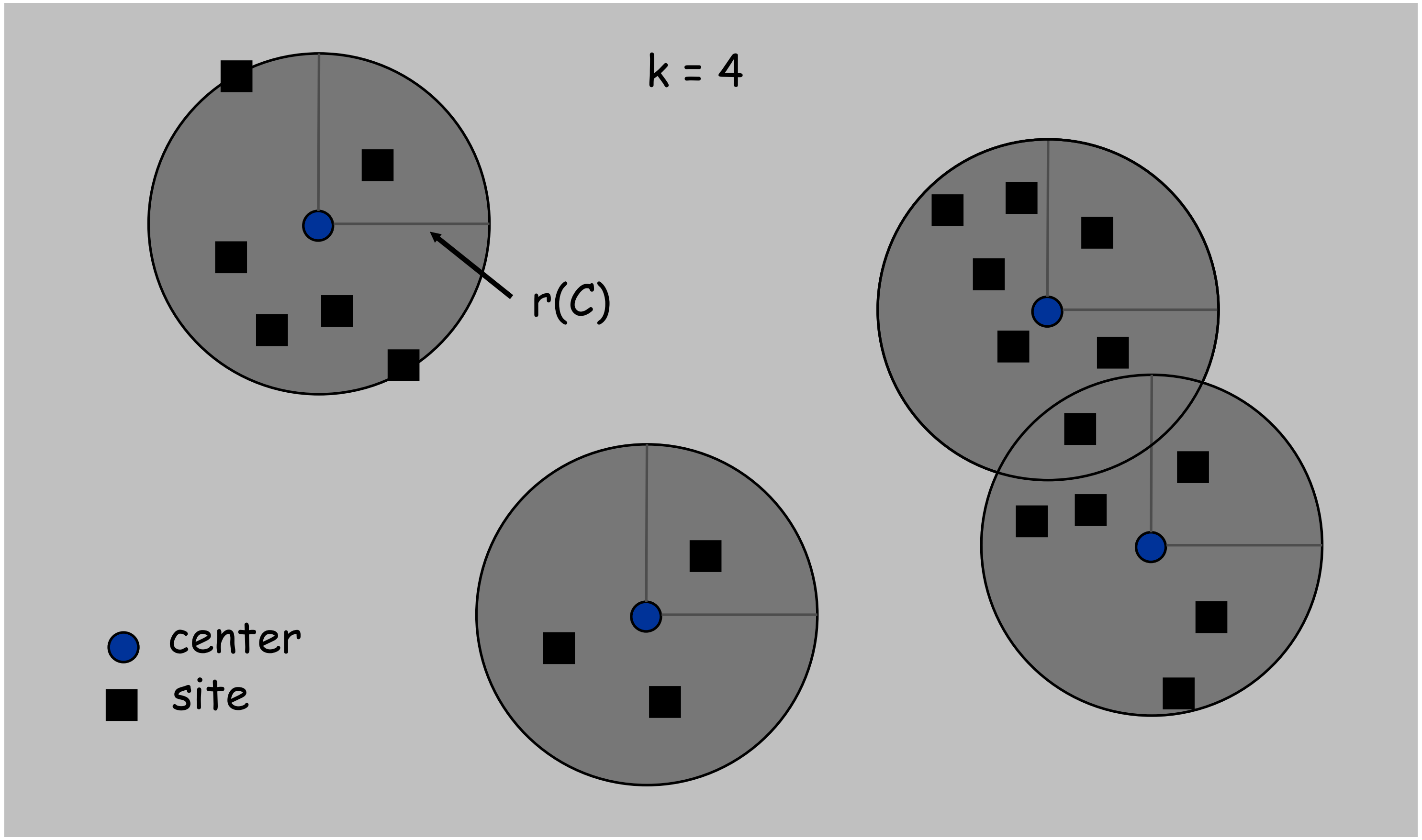
- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

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.

- 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 $|C| = k$.

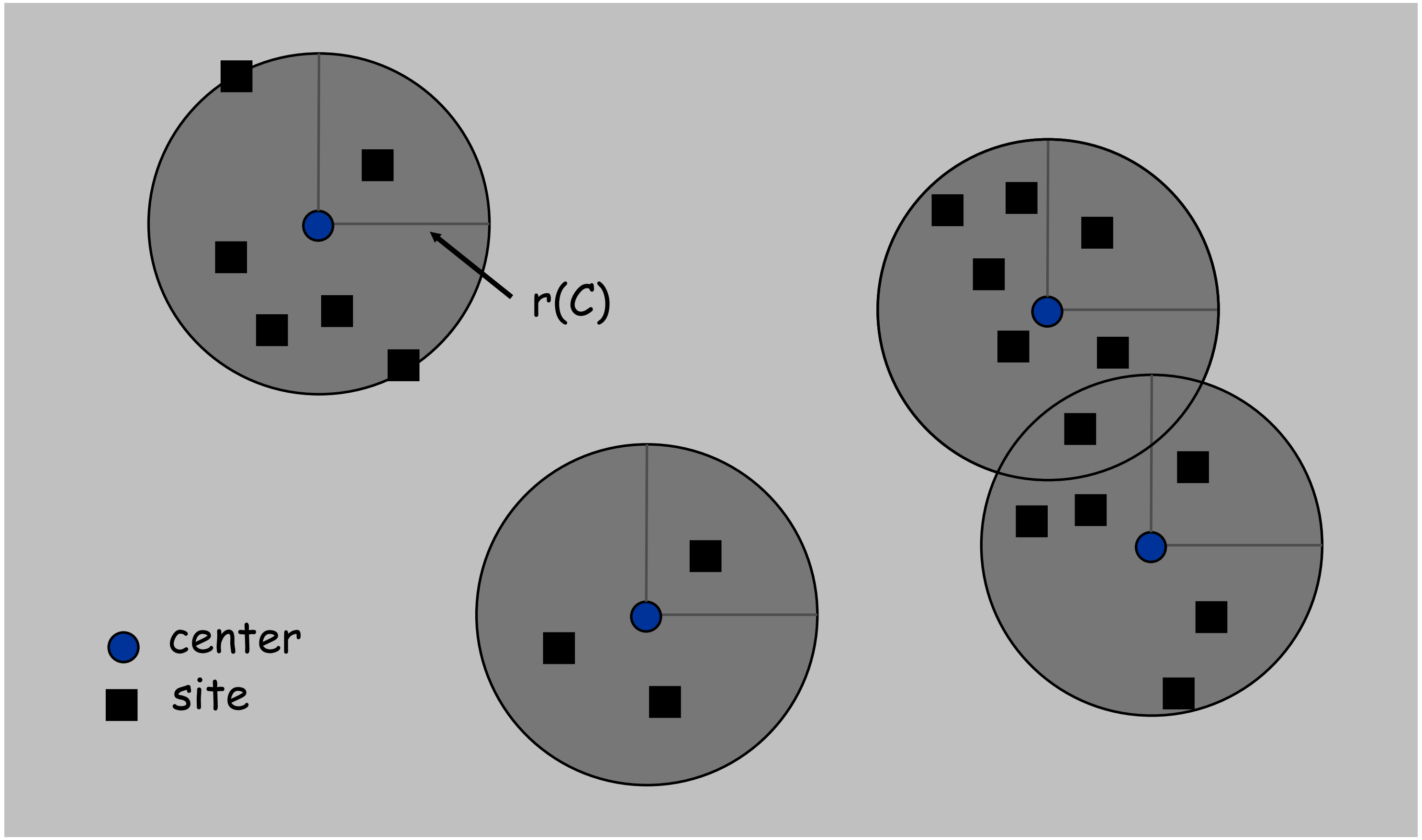
Distance function properties.

- $\text{dist}(x, x) = 0$ (identity)
- $\text{dist}(x, y) = \text{dist}(y, x)$ (symmetry)
- $\text{dist}(x, y) \leq \text{dist}(x, z) + \text{dist}(z, y)$ (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.

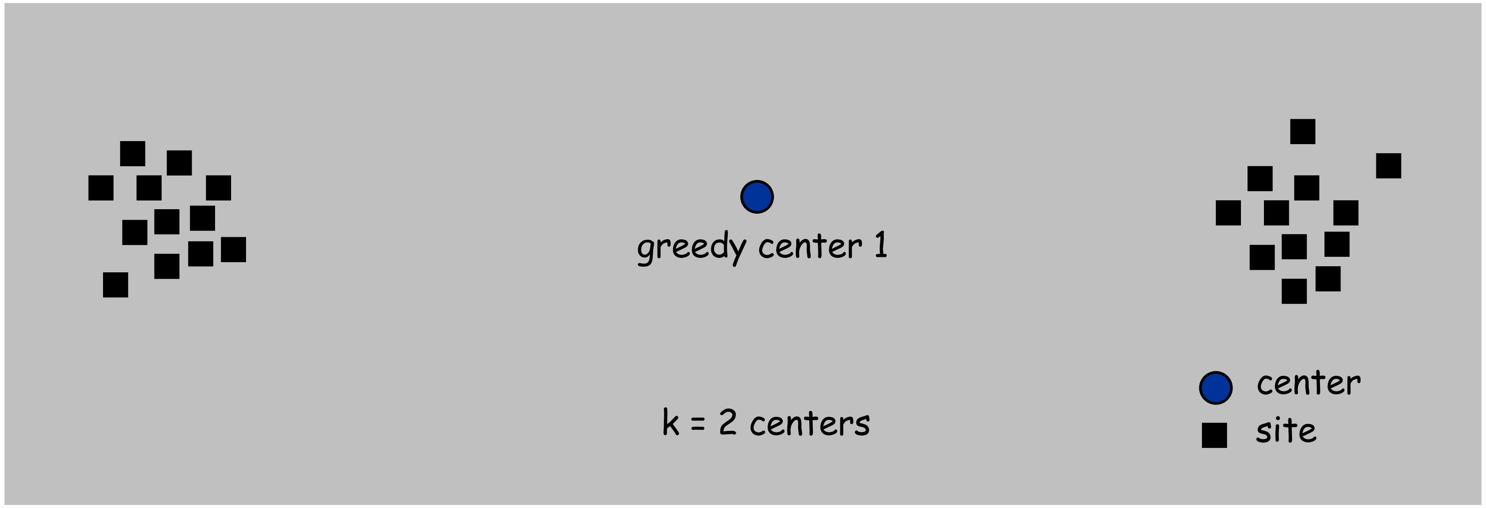
Remark: search can be infinite!



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

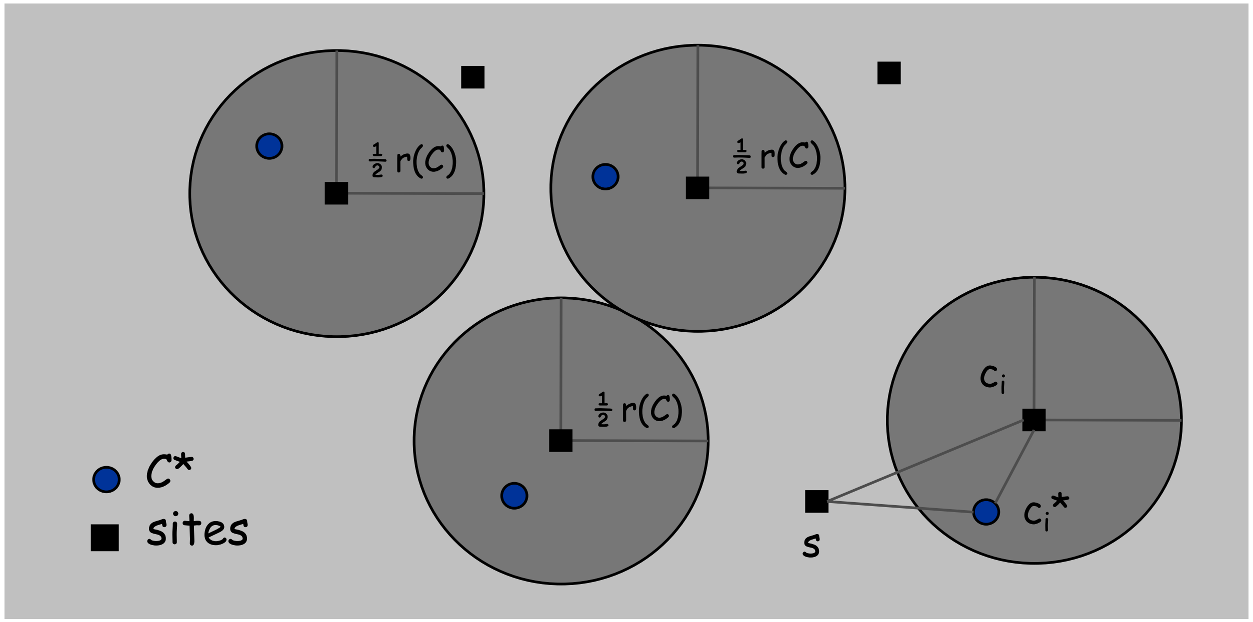
Theorem. Let C^* be an optimal set of centers. Then $r(C) \le 2r(C^*)$.

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

$\le r(C^*)$ 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

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

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

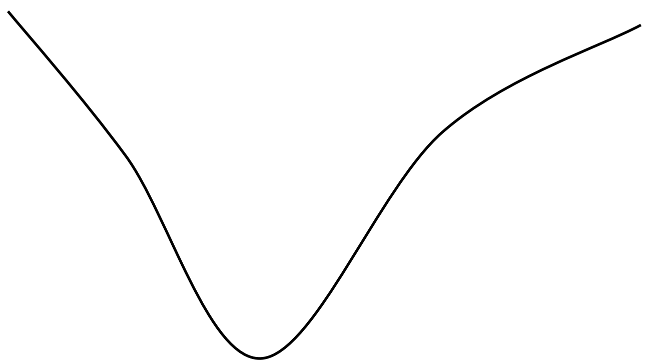
12.1 Landscape of an Optimization Problem

Local Search

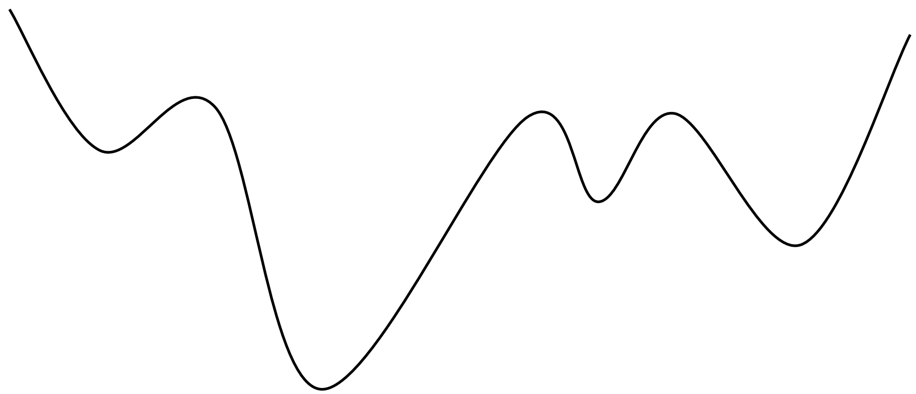
Local search. Algorithm that explores the space of possible solutions in sequential fashion, moving from a current solution to a "nearby" one.

Neighbor relation. Let $S \sim S'$ be a neighbor relation for the problem.

Gradient descent. Let S denote current solution. If there is a neighbor S' of S with strictly lower cost, replace S with the neighbor whose cost is as small as possible. Otherwise, terminate the algorithm.



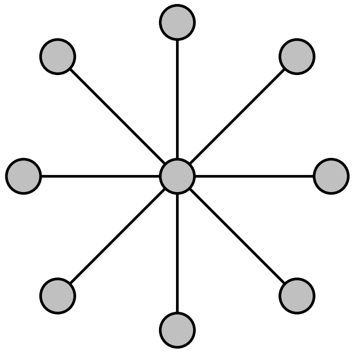
A funnel



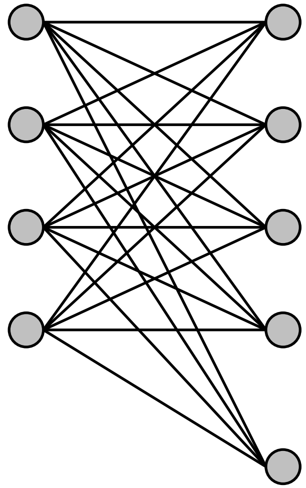
A jagged funnel

Gradient Descent: Vertex Cover

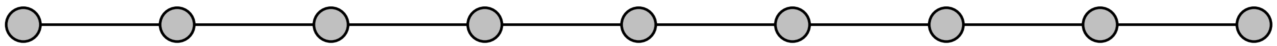
Local optimum. No neighbor is strictly better.



optimum = center node only
local optimum = all other nodes



optimum = all nodes on left side
local optimum = all nodes on right side



optimum = even nodes
local optimum = omit every third node

Gradient Descent: Vertex Cover

VERTEX-COVER. Given a graph $G = (V, E)$, find a subset of nodes S of minimal cardinality such that for each u-v in E, either u or v (or both) are in S.

Neighbor relation. $S \sim S'$ if S' can be obtained from S by adding or deleting a single node. Each vertex cover S has at most n neighbors.

Gradient descent. Start with $S = V$. If there is a neighbor S' that is a vertex cover and has lower cardinality, replace S with S'.

Remark. Algorithm terminates after at most n steps since each update decreases the size of the cover by one.

12.2 Metropolis Algorithm

Metropolis Algorithm

Metropolis algorithm. [Metropolis, Rosenbluth, Rosenbluth, Teller, Teller 1953]

- Simulate behavior of a physical system according to principles of statistical mechanics.
- Globally biased toward "downhill" steps, but occasionally makes "uphill" steps to break out of local minima.

Gibbs-Boltzmann function. The probability of finding a physical system in a state with energy E is proportional to $e^{-E/(kT)}$, where $T > 0$ is temperature and k is a constant.

- For any temperature $T > 0$, function is monotone decreasing function of energy E.
- System more likely to be in a lower energy state than higher one.
  - T large: high and low energy states have roughly same probability
  - T small: low energy states are much more probable

Metropolis Algorithm

Metropolis algorithm.

- Given a fixed temperature T, maintain current state S.
- Randomly perturb current state S to new state S' $\in$ N(S).
- If E(S') $\leq$ E(S), update current state to S'
  Otherwise, update current state to S' with probability $e^{-\Delta E / (kT)}$,
  where $\Delta E = E(S') - E(S) > 0$.

Theorem. Let $f_S(t)$ be fraction of first t steps in which simulation is in state S. Then, assuming some technical conditions, with probability 1:

$$\lim_{t \to \infty} f_S(t) = \frac{1}{Z} e^{-E(S)/(kT)},$$

$$\text{where } Z = \sum_{S \in N(S)} e^{-E(S)/(kT)}.$$

Intuition. Simulation spends roughly the right amount of time in each state, according to Gibbs-Boltzmann equation.

Simulated Annealing

Simulated annealing.

- T large $\Rightarrow$ probability of accepting an uphill move is large.
- T small $\Rightarrow$ uphill moves are almost never accepted.
- Idea: turn knob to control T.
- Cooling schedule: T = T(i) at iteration i.

Physical analog.

- Take solid and raise it to high temperature, we do not expect it to maintain a nice crystal structure.
- Take a molten solid and freeze it very abruptly, we do not expect to get a perfect crystal either.
- Annealing: cool material gradually from high temperature, allowing it to reach equilibrium at succession of intermediate lower temperatures.

12.3 Hopfield Neural Networks

Hopfield Neural Networks

Hopfield networks. Simple model of an associative memory, in which a large collection of units are connected by an underlying network, and neighboring units try to correlate their states.

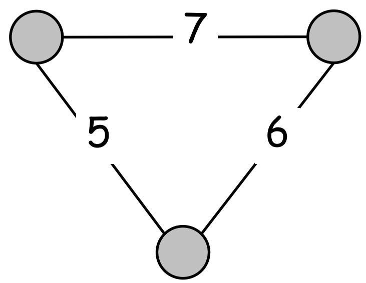
Input: Graph $G = (V, E)$ with integer edge weights w.

positive or negative

Configuration. Node assignment $s_u = \pm 1$.

Intuition. If $w_{uv} < 0$, then u and v want to have the same state; if $w_{uv} > 0$ then u and v want different states.

Note. In general, no configuration respects all constraints.



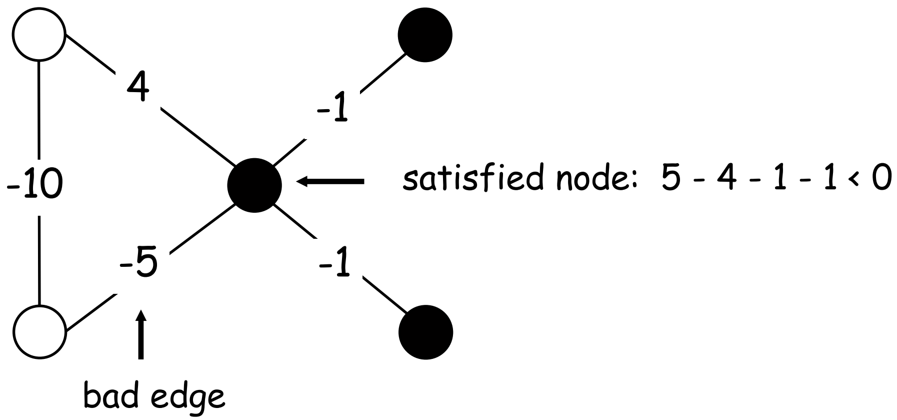
Hopfield Neural Networks

Def. With respect to a configuration S, edge e = (u, v) is good if $w_e s_u s_v < 0$. That is, if $w_e < 0$ then $s_u = s_v$; if $w_e > 0$, $s_u \neq s_v$.

Def. With respect to a configuration S, a node u is satisfied if the weight of incident good edges $\geq$ weight of incident bad edges.

$$\sum_{v:\ e=(u,v)\in E} w_e s_u s_v \leq 0$$

Def. A configuration is stable if all nodes are satisfied.



Goal. Find a stable configuration, if such a configuration exists.

Hopfield Neural Networks

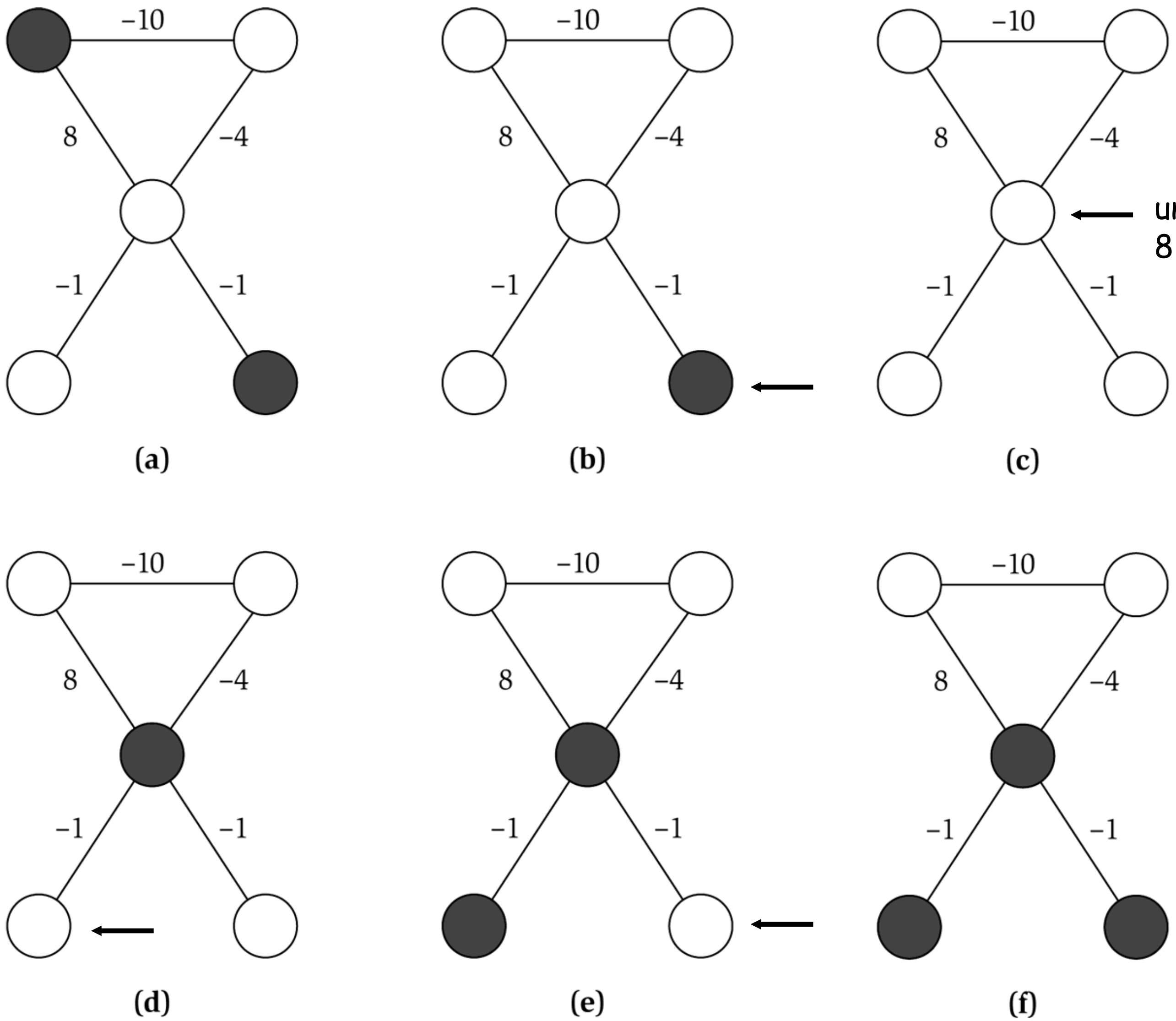
Goal. Find a stable configuration, if such a configuration exists.

State-flipping algorithm. Repeated flip state of an unsatisfied node.

```
Hopfield-Flip(G, w) {
   S ← arbitrary configuration

   while (current configuration is not stable) {
      u ← unsatisfied node
      s_u = -s_u
   }

   return S
}
```

State Flipping Algorithm

unsatisfied node
10 - 8 > 0



unsatisfied node
8 - 4 - 1 - 1 > 0

stable