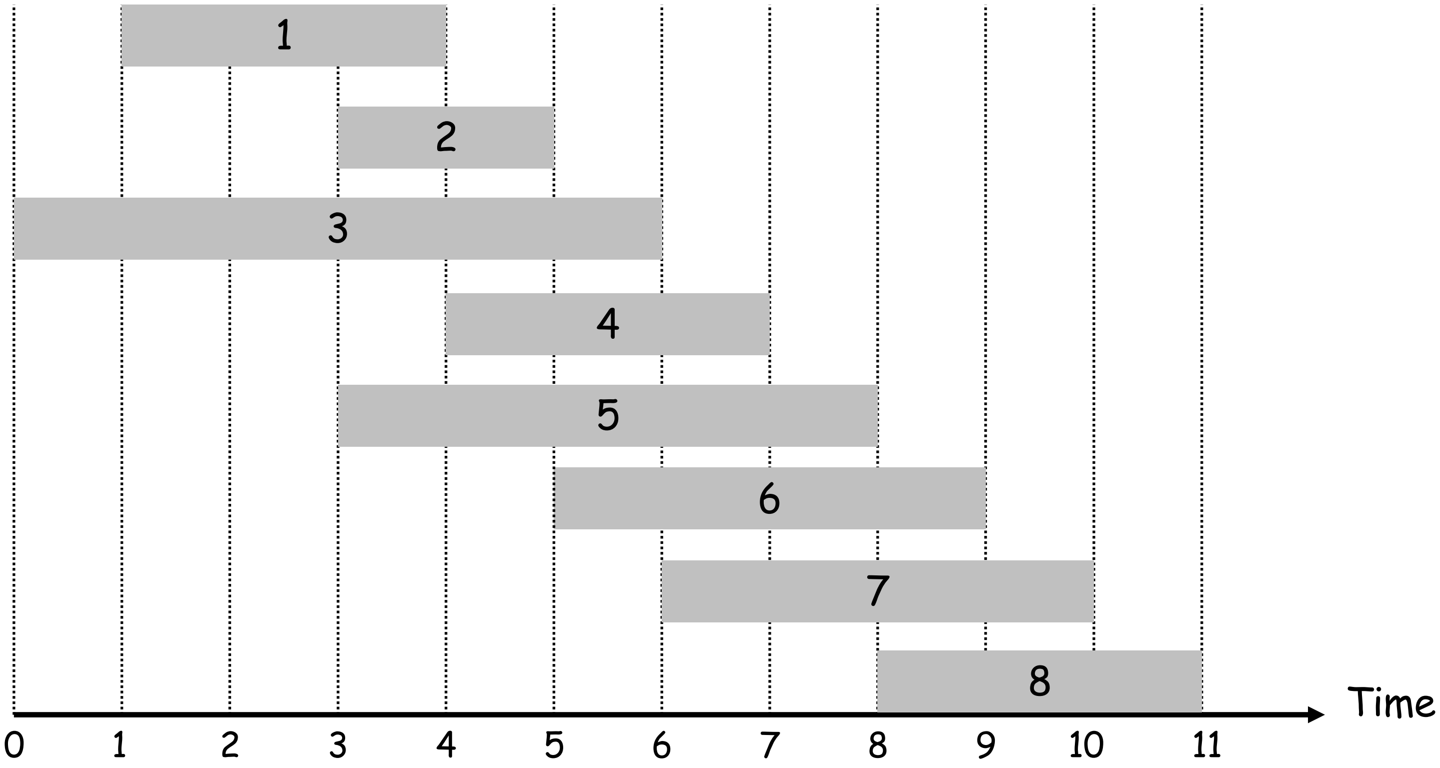
# Final Exam Review

# Weighted Interval Scheduling

**Notation.** Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

**Def.** p(j) = largest index i < j such that job i is compatible with j.

**Ex:** p(8) = 5, p(7) = 3, p(2) = 0.



# Dynamic Programming: Binary Choice

**Notation.** OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

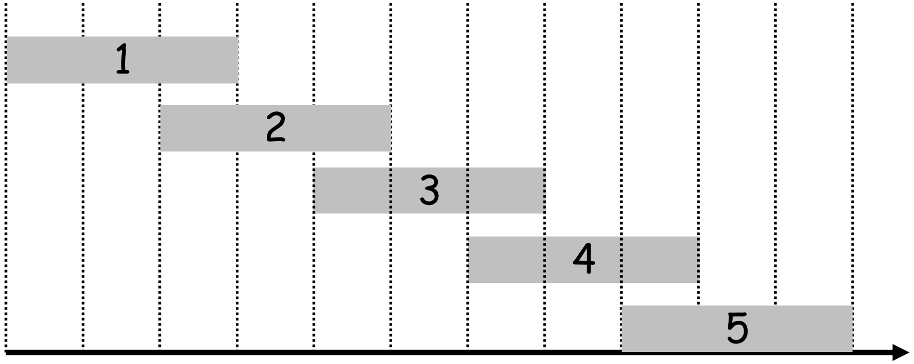
- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

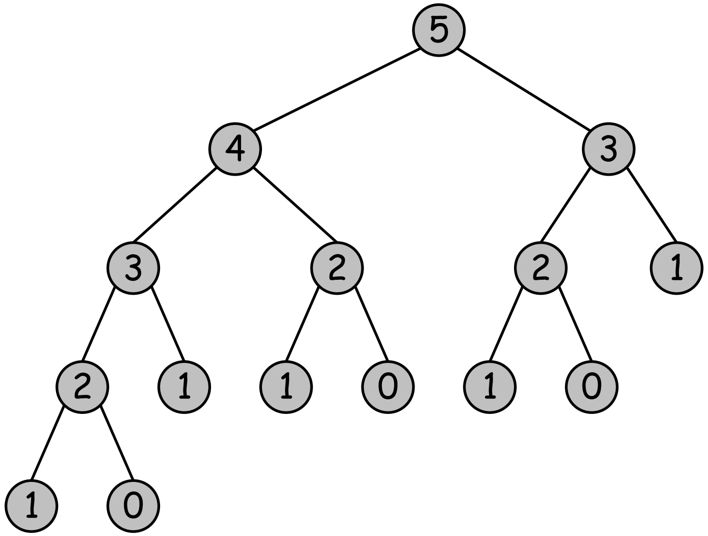
# Weighted Interval Scheduling: Brute Force

**Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems ⇒ exponential algorithms.

**Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



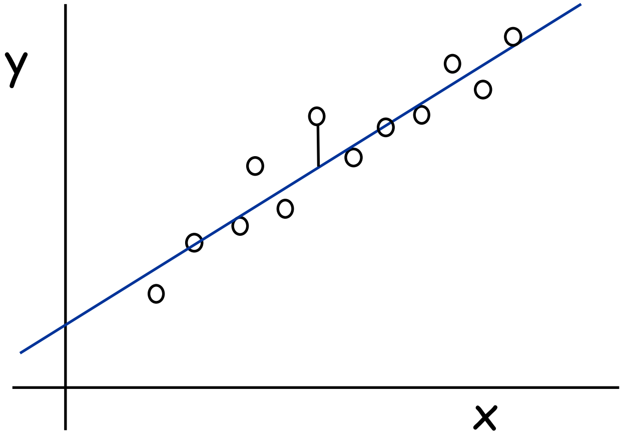
# 6.3 Segmented Least Squares

# Segmented Least Squares

**Least squares.**

- Foundational problem in statistic and numerical analysis.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line y = ax + b that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



**Solution.** Calculus ⇒ min error is achieved when

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^{\,2} - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

# Segmented Least Squares

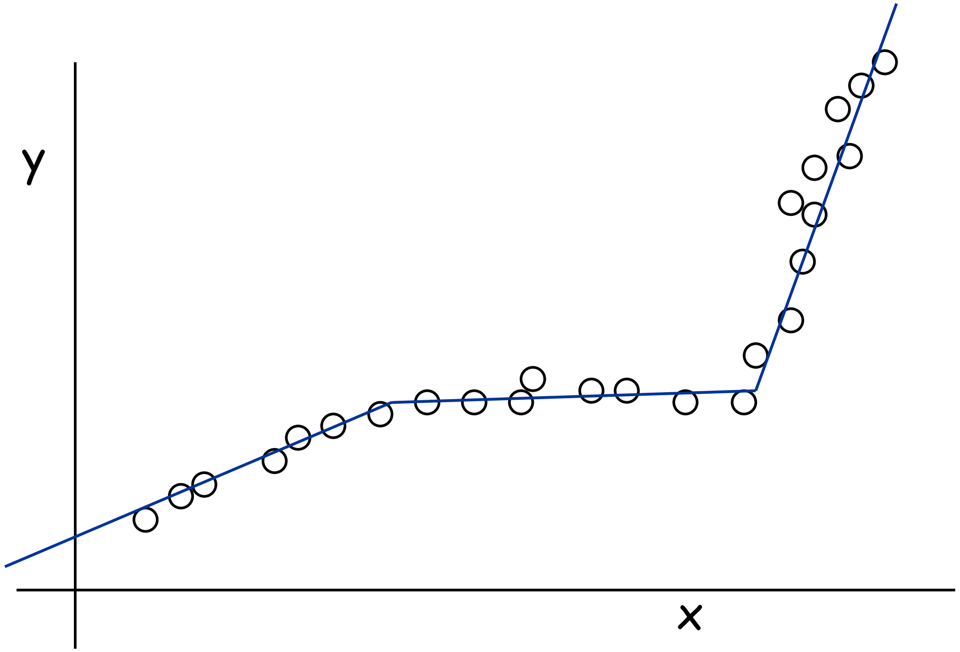
Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
- $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes f(x).

Q. What's a reasonable choice for f(x) to balance accuracy and parsimony?

↑ goodness of fit

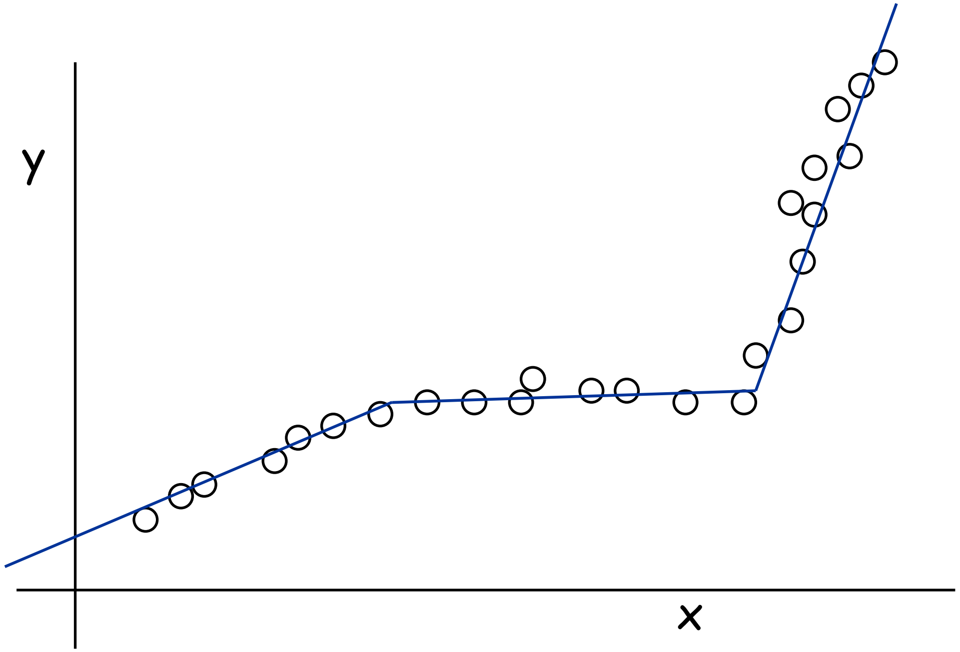
↑ number of lines



# Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
- $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes:
  - the sum of the sums of the squared errors E in each segment
  - the number of lines L
- Tradeoff function: $E + c L$, for some constant $c > 0$.



# Dynamic Programming: Multiway Choice

Notation.

- OPT(j) = minimum cost for points $p_1$, $p_{i+1}$ , . . . , $p_j$.
- e(i, j) = minimum sum of squares for points $p_i$, $p_{i+1}$ , . . . , $p_j$.

To compute OPT(j):

- Last segment uses points $p_i$, $p_{i+1}$ , . . . , $p_j$ for some i.
- Cost = e(i, j) + c + OPT(i-1).

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \left\{ e(i,j) + c + OPT(i-1) \right\} & \text{otherwise} \end{cases}$$

# 6.4 Knapsack Problem

# Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- 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 |

Greedy: repeatedly add item with maximum ratio $v_i / w_i$.

Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

# Dynamic Programming: Adding a New Variable

**Def.** OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- Case 2: OPT selects item i.
  - new weight limit = w – $w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

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

# Knapsack Algorithm

W + 1

n + 1

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\phi$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| { 1 } | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| { 1, 2 } | 0 | 1 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| { 1, 2, 3 } | 0 | 1 | 6 | 7 | 7 | 18 | 19 | 24 | 25 | 25 | 25 | 25 |
| { 1, 2, 3, 4 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 24 | 28 | 29 | 29 | 40 |
| { 1, 2, 3, 4, 5 } | 0 | 1 | 6 | 7 | 7 | 18 | 22 | 28 | 29 | 34 | 34 | 40 |

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

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

# Dynamic Programming Summary

Recipe.

- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.

- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares. ←
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

Viterbi algorithm for HMM also uses DP to optimize a maximum likelihood tradeoff between parsimony and accuracy

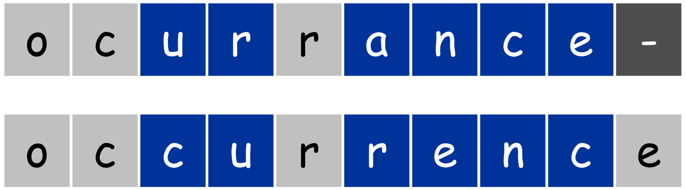
CKY parsing algorithm for context-free grammar has similar structure

Top-down vs. bottom-up: different people have different intuitions.

# String Similarity

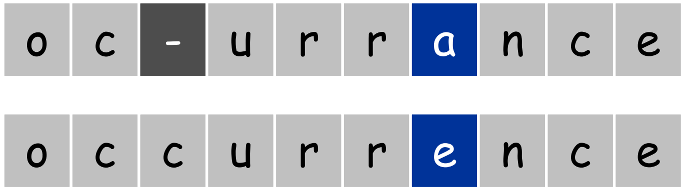
## How similar are two strings?

- `ocurrance`
- `occurrence`



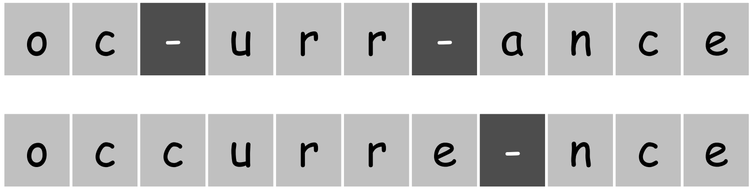
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| o | c | u | r | r | a | n | c | e | - |
| o | c | c | u | r | r | e | n | c | e |

5 mismatches, 1 gap



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| o | c | - | u | r | r | a | n | c | e |
| o | c | c | u | r | r | e | n | c | e |

1 mismatch, 1 gap



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| o | c | - | u | r | r | - | a | n | c | e |
| o | c | c | u | r | r | e | - | n | c | e |

0 mismatches, 3 gaps

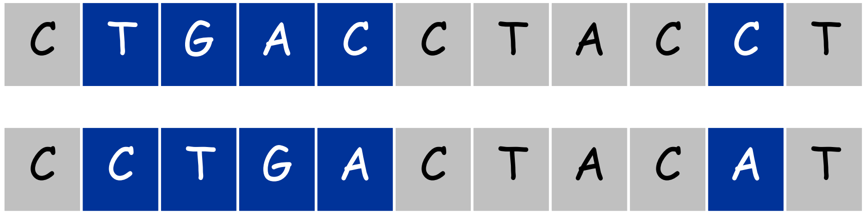
# Edit Distance

**Applications.**

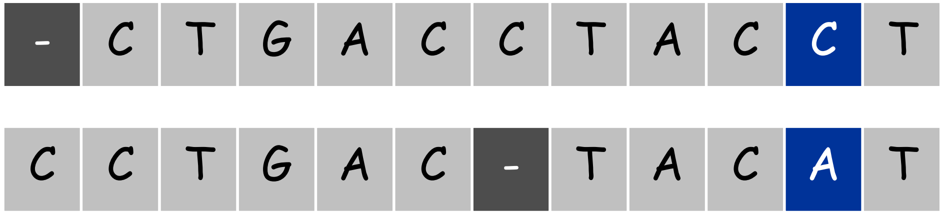
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

**Edit distance.** [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty $\delta$; mismatch penalty $\alpha_{pq}$.
- Cost = sum of gap and mismatch penalties.



$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$



$2\delta + \alpha_{CA}$

# Sequence Alignment

**Goal:** Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ find alignment of minimum cost.

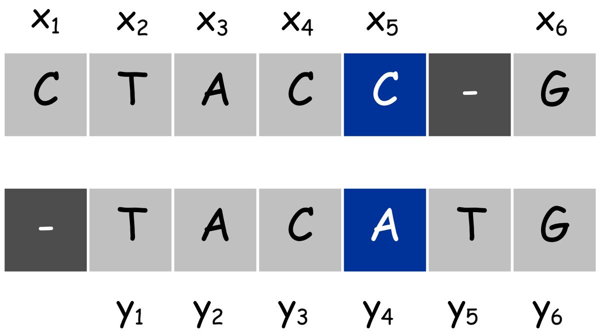
**Def.** An alignment $M$ is a set of ordered pairs $x_i$-$y_j$ such that each item occurs in at most one pair and no crossings.

**Def.** The pair $x_i$-$y_j$ and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i\,:\,x_i \text{ unmatched}} \delta + \sum_{j\,:\,y_j \text{ unmatched}} \delta}_{\text{gap}}$$

**Ex:** `CTACCG` vs. `TACATG`.

**Sol:** $M = x_2$-$y_1$, $x_3$-$y_2$, $x_4$-$y_3$, $x_5$-$y_4$, $x_6$-$y_6$.



| | | | | | | |
|---|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | | $x_6$ |
| C | T | A | C | C | - | G |
| - | T | A | C | A | T | G |
| | $y_1$ | $y_2$ | $y_3$ | $y_4$ | $y_5$ | $y_6$ |

# Sequence Alignment: Problem Structure

**Def.** OPT(i, j) = min cost of aligning strings $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$.

- Case 1: OPT matches $x_i$-$y_j$.
  - pay mismatch for $x_i$-$y_j$ + min cost of aligning two strings $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_{j-1}$
- Case 2a: OPT leaves $x_i$ unmatched.
  - pay gap for $x_i$ and min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_j$
- Case 2b: OPT leaves $y_j$ unmatched.
  - pay gap for $y_j$ and min cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_{j-1}$

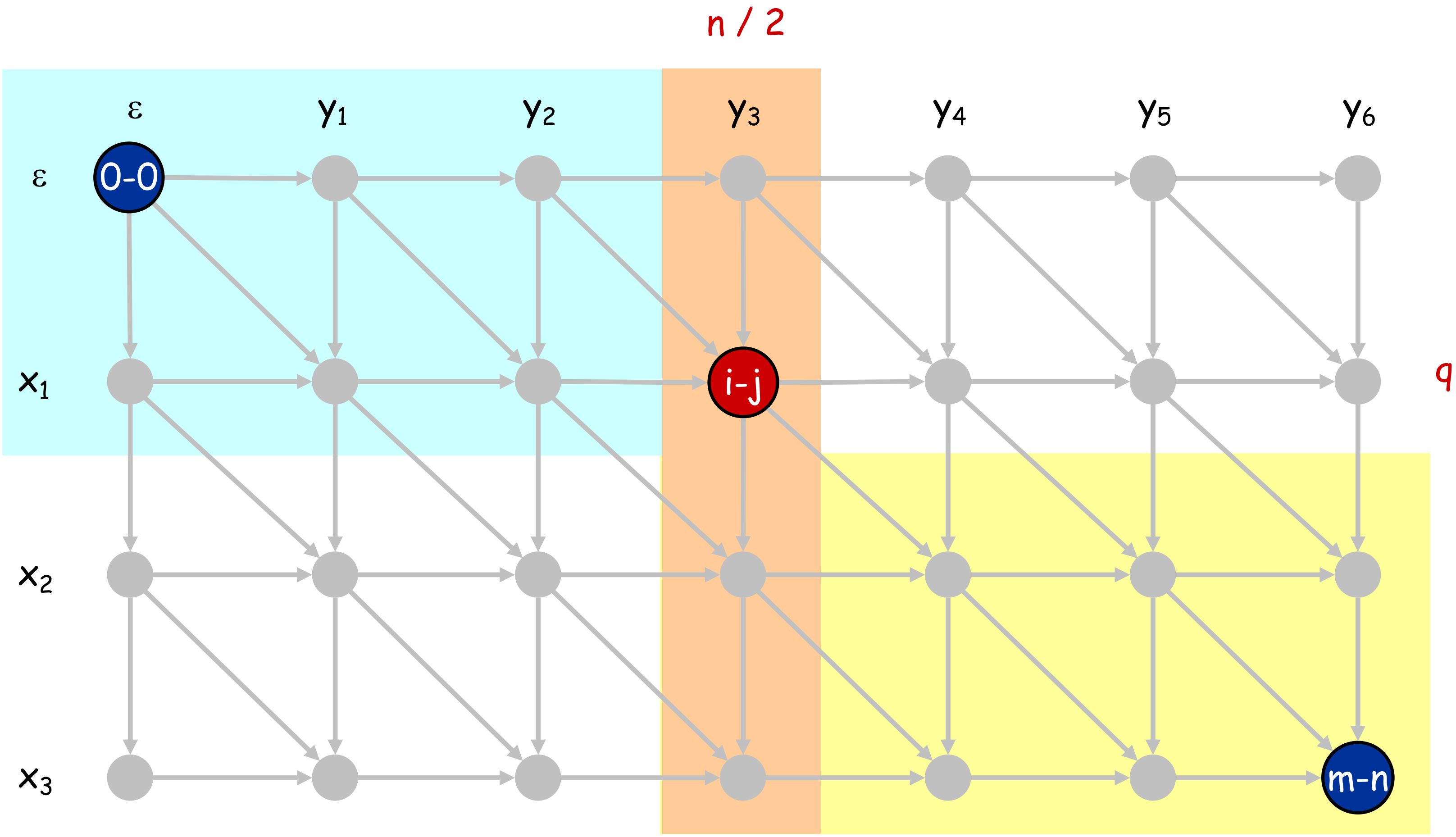
$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

# Sequence Alignment: Linear Space

**Divide:** find index q that minimizes $f(q, n/2) + g(q, n/2)$ using DP.

- Align $x_q$ and $y_{n/2}$.

**Conquer:** recursively compute optimal alignment in each piece.

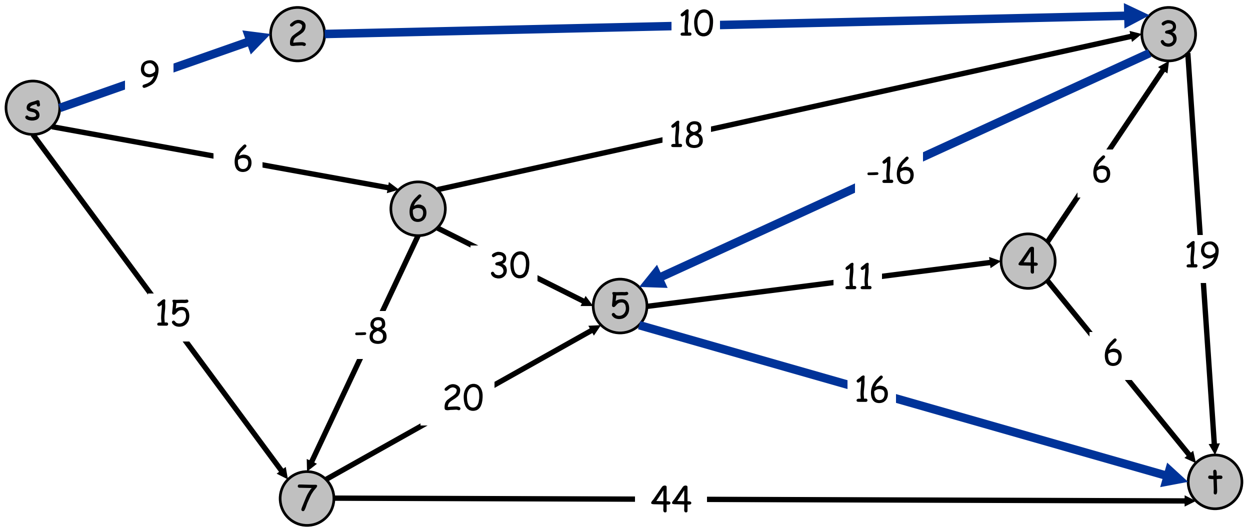


# Shortest Paths

Shortest path problem. Given a directed graph $G = (V, E)$, with edge weights $c_{vw}$, find shortest path from node s to node t.

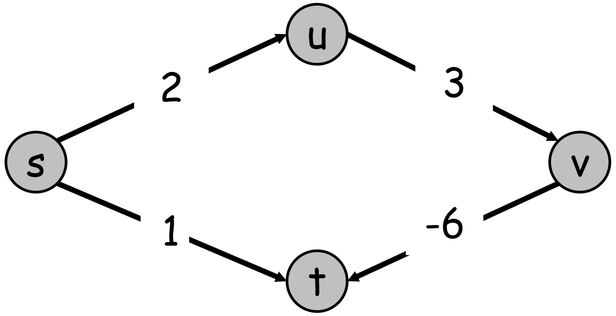
allow negative weights

Ex. Nodes represent agents in a financial setting and $c_{vw}$ is cost of transaction in which we buy from agent v and sell immediately to w.

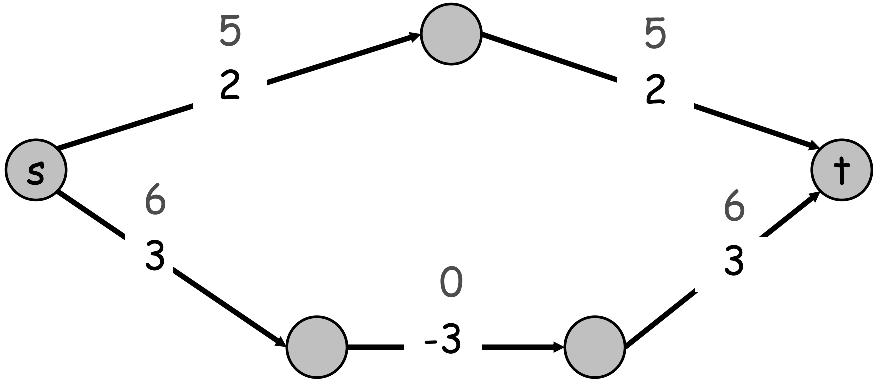


# Shortest Paths: Failed Attempts

**Dijkstra.** Can fail if negative edge costs.



**Re-weighting.** Adding a constant to every edge weight can fail.



# Shortest Paths: Dynamic Programming

**Def.** OPT(i, v) = length of shortest v-t path P using at most i edges.

- Case 1: P uses at most i-1 edges.
  - OPT(i, v) = OPT(i-1, v)

- Case 2: P uses exactly i edges.
  - if (v, w) is first edge, then OPT uses (v, w), and then selects best w-t path using at most i-1 edges

$$OPT(i,v) = \begin{cases} 0 & \text{if } i = 0 \\ \min\left\{ OPT(i-1, v),\ \min\limits_{(v,w)\in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{otherwise} \end{cases}$$

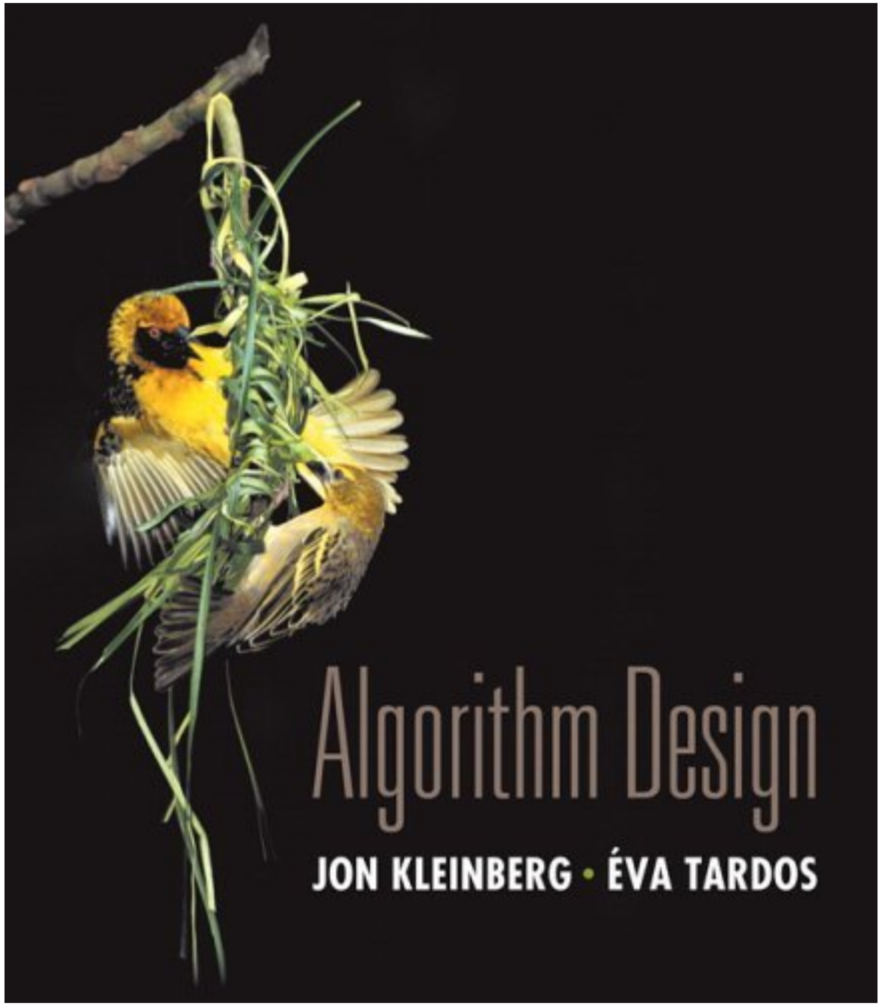
**Remark.** By previous observation, if no negative cycles, then OPT(n-1, v) = length of shortest v-t path.

# Shortest Paths: Implementation

```
Shortest-Path(G, t) {
   foreach node v ∈ V
      M[0, v] ← ∞
   M[0, t] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← M[i-1, v]
      foreach edge (v, w) ∈ E
         M[i, v] ← min { M[i, v], M[i-1, w] + $c_{vw}$ }
}
```

**Analysis.** $\Theta(mn)$ time, $\Theta(n^2)$ space.

**Finding the shortest paths.** Maintain a "successor" for each table entry.



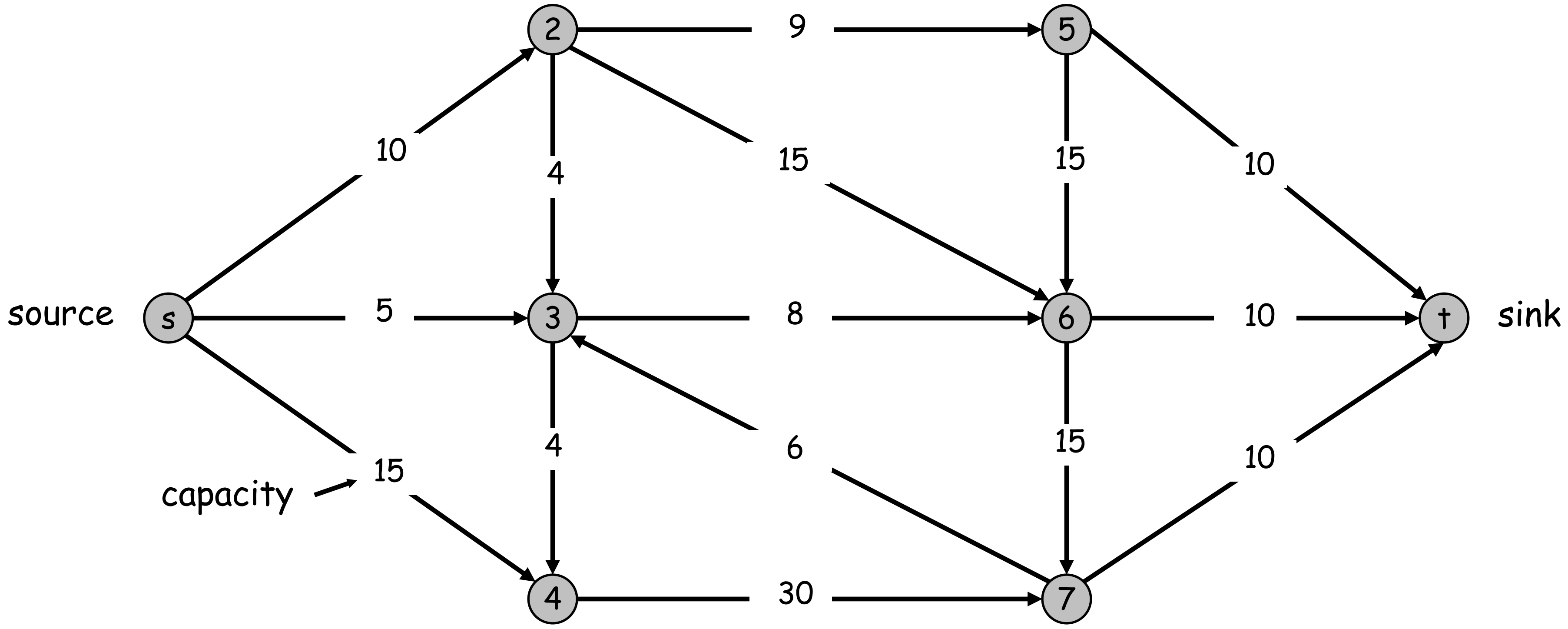
# Network Flow

Slides by Kevin Wayne.

# Minimum Cut Problem

Flow network.

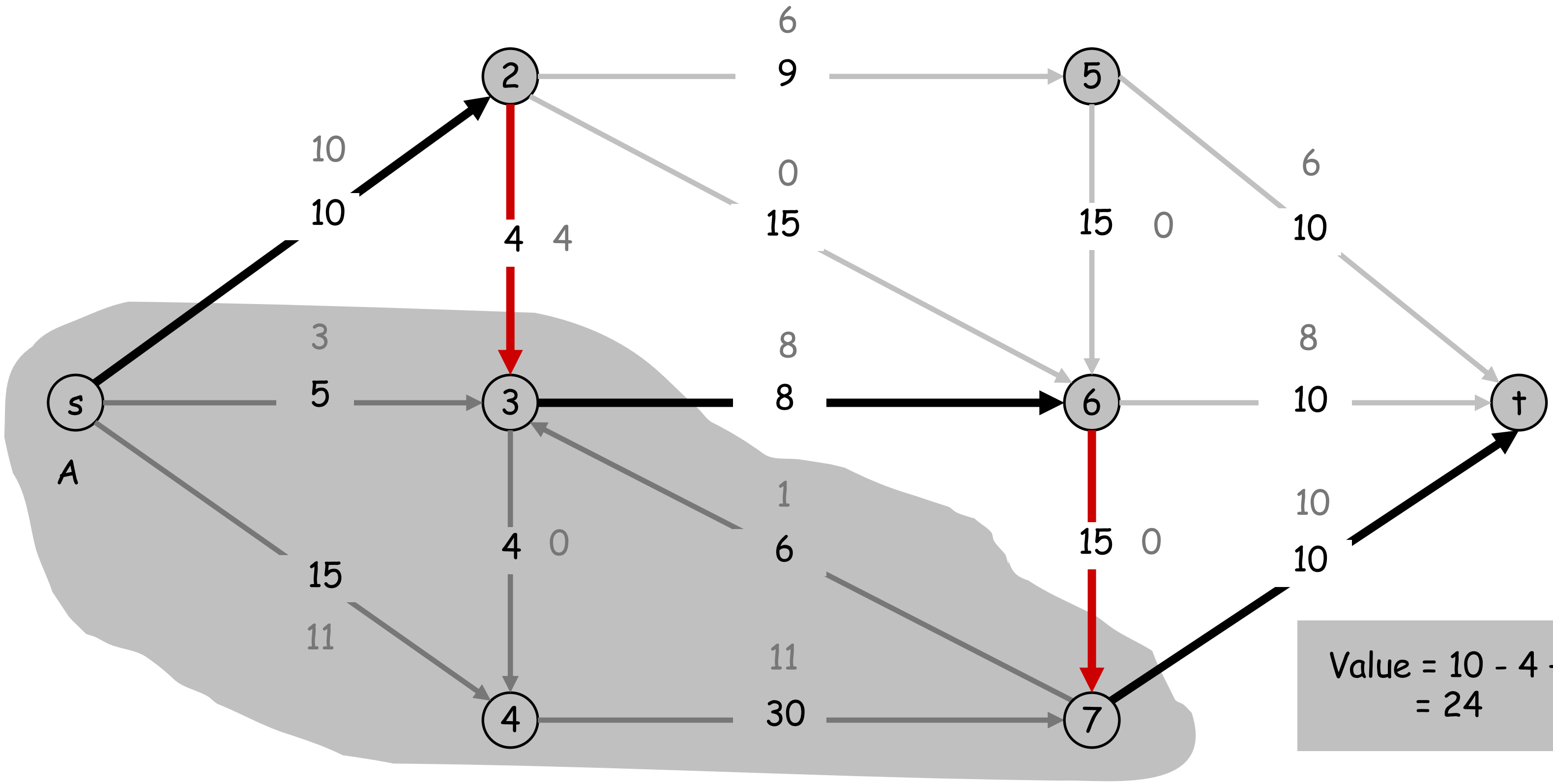
- Abstraction for material flowing through the edges.
- G = (V, E) = directed graph, no parallel edges.
- Two distinguished nodes: s = source, t = sink.
- c(e) = capacity of edge e.



# Flows and Cuts

**Flow value lemma.** Let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s.

$$\sum_{e \text{ out of } A} f(e) \;-\; \sum_{e \text{ in to } A} f(e) \;=\; v(f)$$



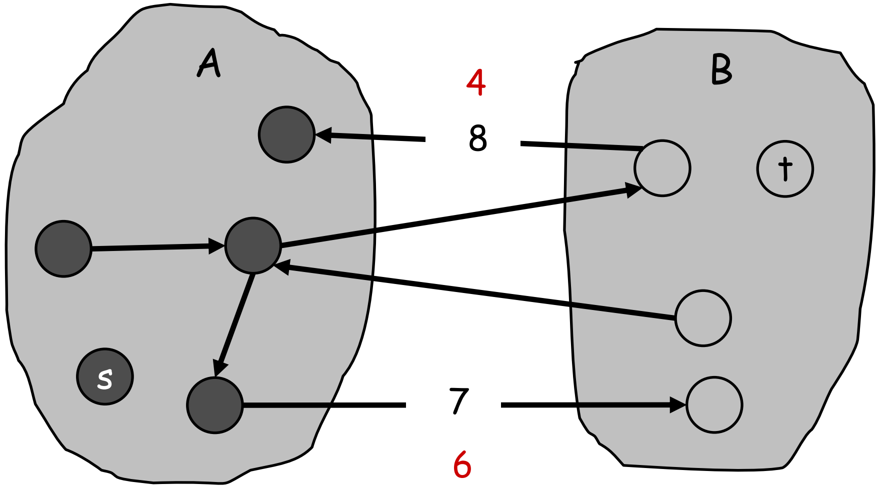
Value = 10 - 4 + 8 - 0 + 10
= 24

# Flows and Cuts

**Weak duality.** Let f be any flow. Then, for any s-t cut (A, B) we have v(f) ≤ cap(A, B).

**Pf.**

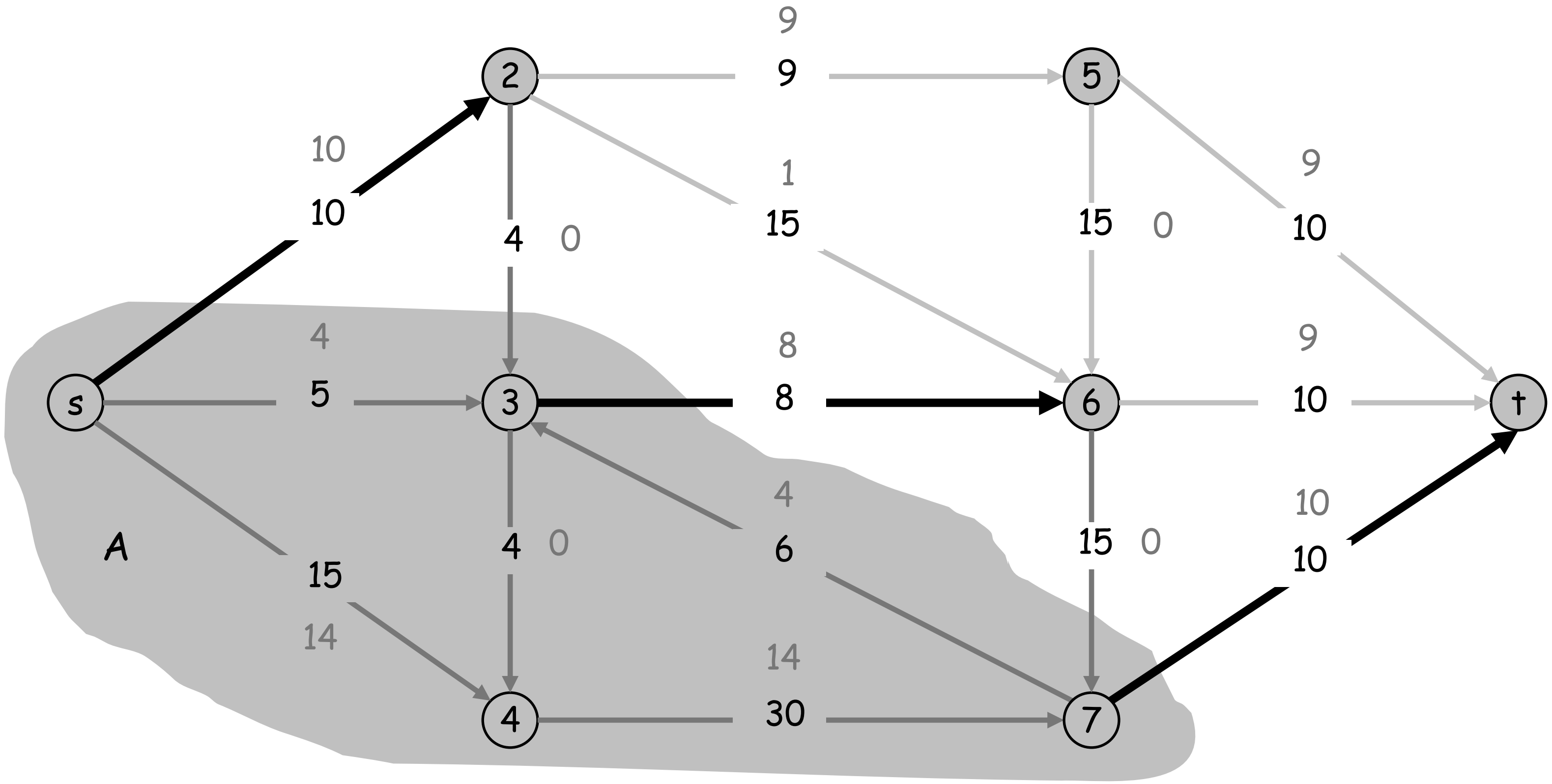
$$
\begin{aligned}
v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
&\le \sum_{e \text{ out of } A} f(e) \\
&\le \sum_{e \text{ out of } A} c(e) \\
&= \text{cap}(A,B) \quad \blacksquare
\end{aligned}
$$



# Certificate of Optimality

**Corollary.** Let f be any flow, and let (A, B) be any cut.
If v(f) = cap(A, B), then f is a max flow and (A, B) is a min cut.

Value of flow = 28
Cut capacity = 28 ⇒ Flow value ≤ 28



# Max-Flow Min-Cut Theorem

**Augmenting path theorem.** Flow f is a max flow iff there are no augmenting paths.

**Max-flow min-cut theorem.** [Ford-Fulkerson 1956] The value of the max flow is equal to the value of the min cut.

**Proof strategy.** We prove both simultaneously by showing the TFAE:

(i) There exists a cut (A, B) such that v(f) = cap(A, B).
(ii) Flow f is a max flow.
(iii) There is no augmenting path relative to f.

**(i) $\Rightarrow$ (ii)** This was the corollary to weak duality lemma.

**(ii) $\Rightarrow$ (iii)** We show contrapositive.

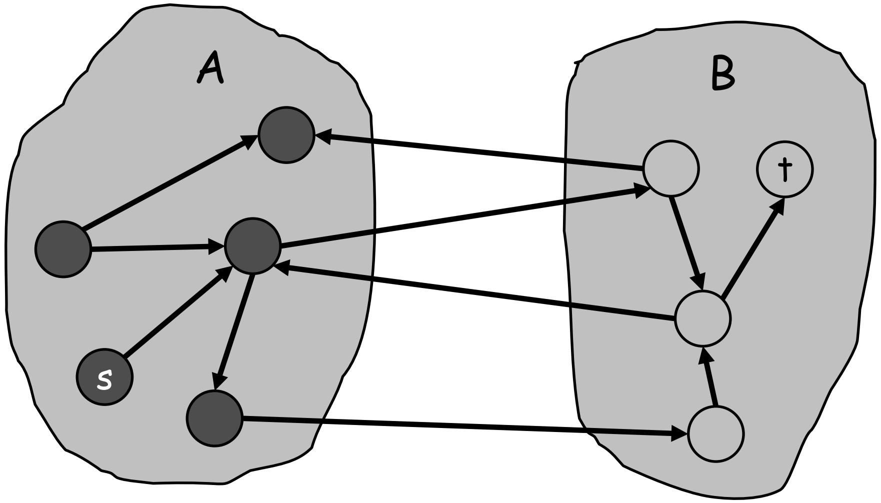
- Let f be a flow. If there exists an augmenting path, then we can improve f by sending flow along path.

# Proof of Max-Flow Min-Cut Theorem

(iii) $\Rightarrow$ (i)

- Let f be a flow with no augmenting paths.
- Let A be set of vertices reachable from s in residual graph.
- By definition of A, s $\in$ A.
- By definition of f, t $\notin$ A.

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to A}} f(e) \\ &= \sum_{e \text{ out of } A} c(e) \\ &= cap(A, B) \quad \blacksquare \end{aligned}$$



original network

# Running Time

**Assumption.** All capacities are integers between 1 and $C$.

**Invariant.** Every flow value $f(e)$ and every residual capacities $c_f(e)$ remains an integer throughout the algorithm.

**Theorem.** The algorithm terminates in at most $v(f^*) \leq nC$ iterations.
**Pf.** Each augmentation increase value by at least 1. ▪

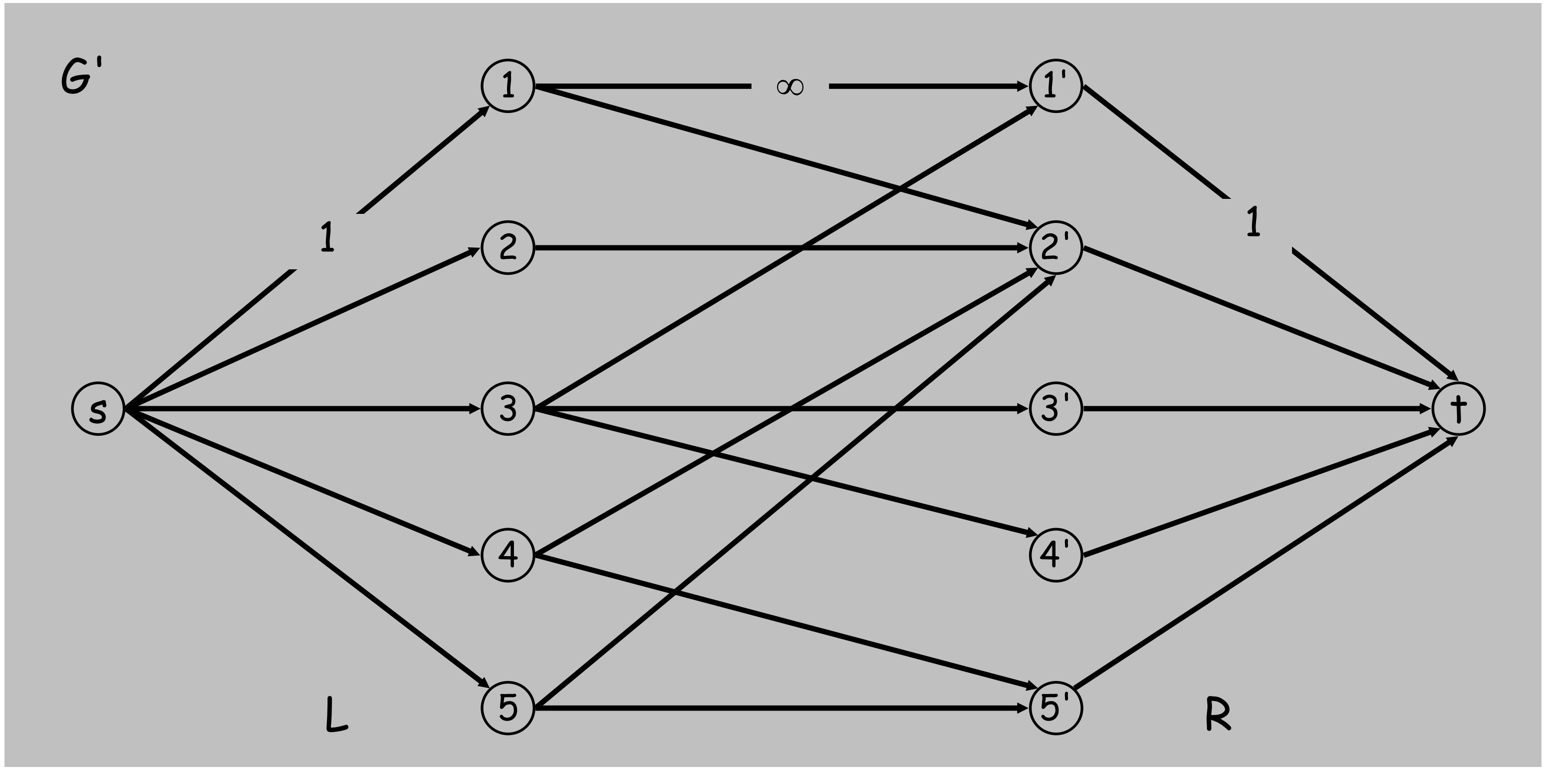
**Corollary.** If $C = 1$, Ford-Fulkerson runs in $O(m)$ time.

**Integrality theorem.** If all capacities are integers, then there exists a max flow $f$ for which every flow value $f(e)$ is an integer.
**Pf.** Since algorithm terminates, theorem follows from invariant. ▪

# Bipartite Matching

## Max flow formulation.

- Create digraph $G' = (L \cup R \cup \{s, t\}, E')$.
- Direct all edges from L to R, and assign infinite (or unit) capacity.
- Add source s, and unit capacity edges from s to each node in L.
- Add sink t, and unit capacity edges from each node in R to t.

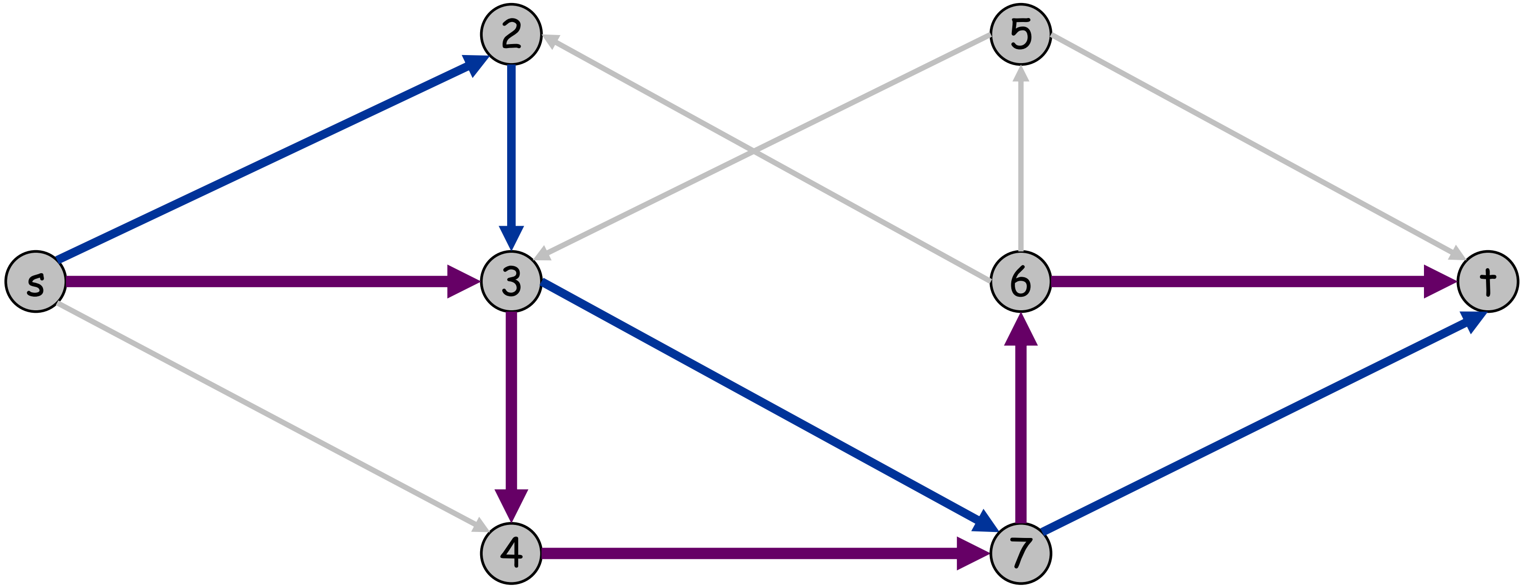


# Edge Disjoint Paths

**Disjoint path problem.** Given a digraph G = (V, E) and two nodes s and t, find the max number of edge-disjoint s-t paths.

**Def.** Two paths are edge-disjoint if they have no edge in common.

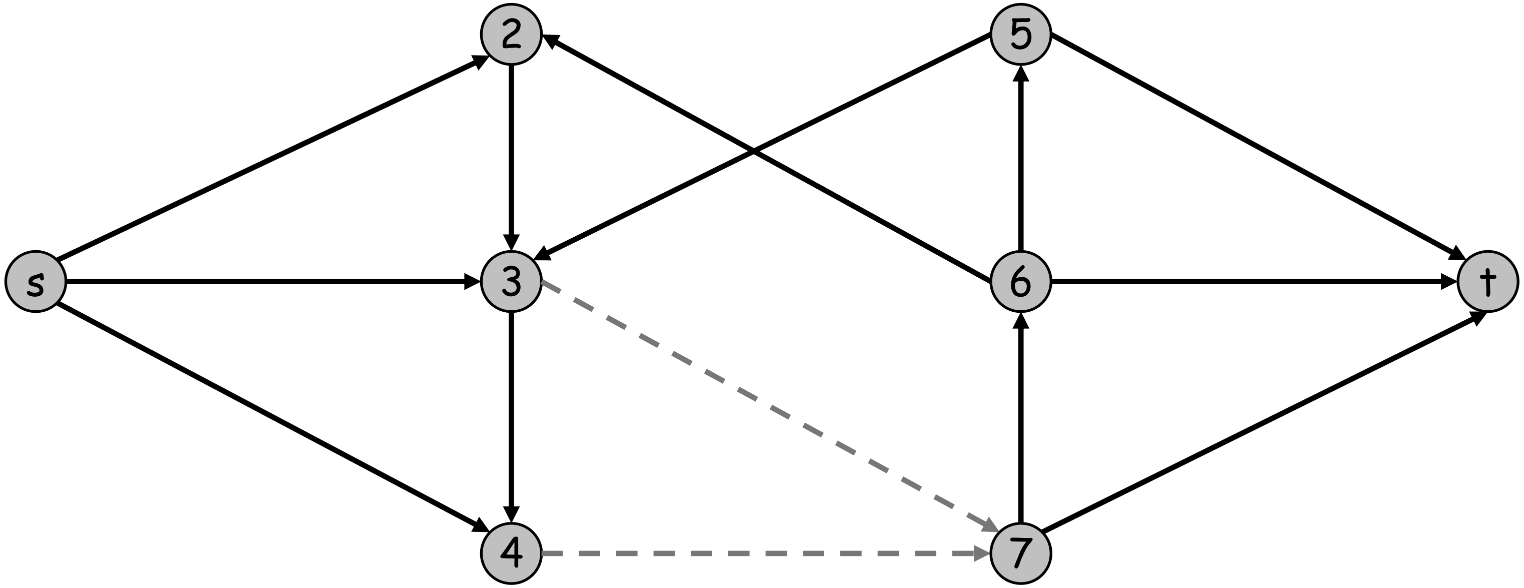
**Ex:** communication networks.



# Network Connectivity

Network connectivity. Given a digraph $G = (V, E)$ and two nodes s and t, find min number of edges whose removal disconnects t from s.

Def. A set of edges $F \subseteq E$ disconnects t from s if all s-t paths uses at least on edge in F.

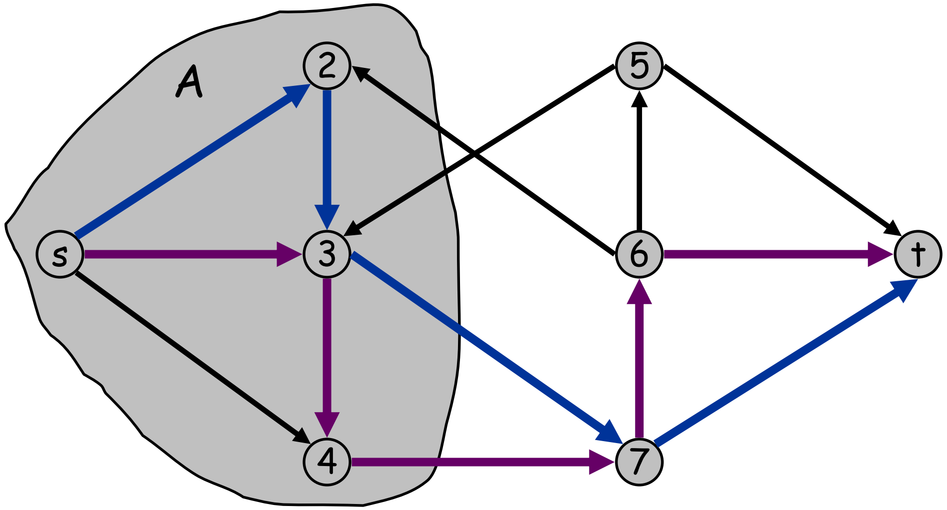


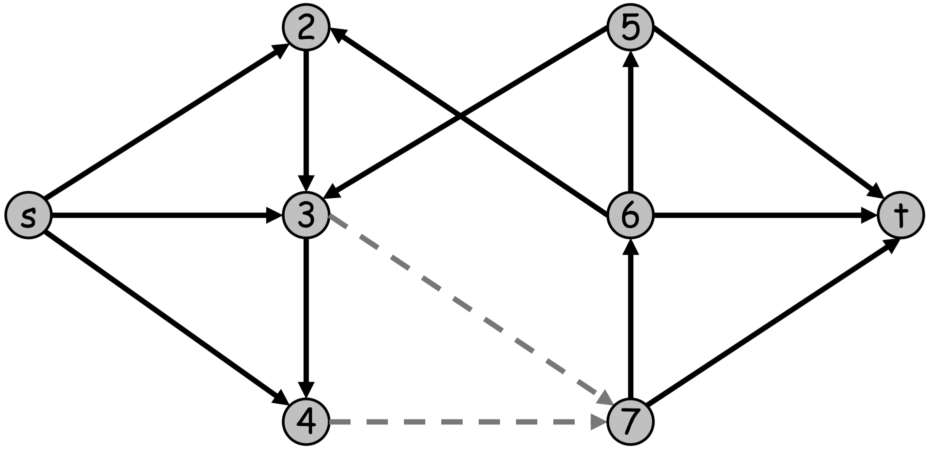
# Disjoint Paths and Network Connectivity

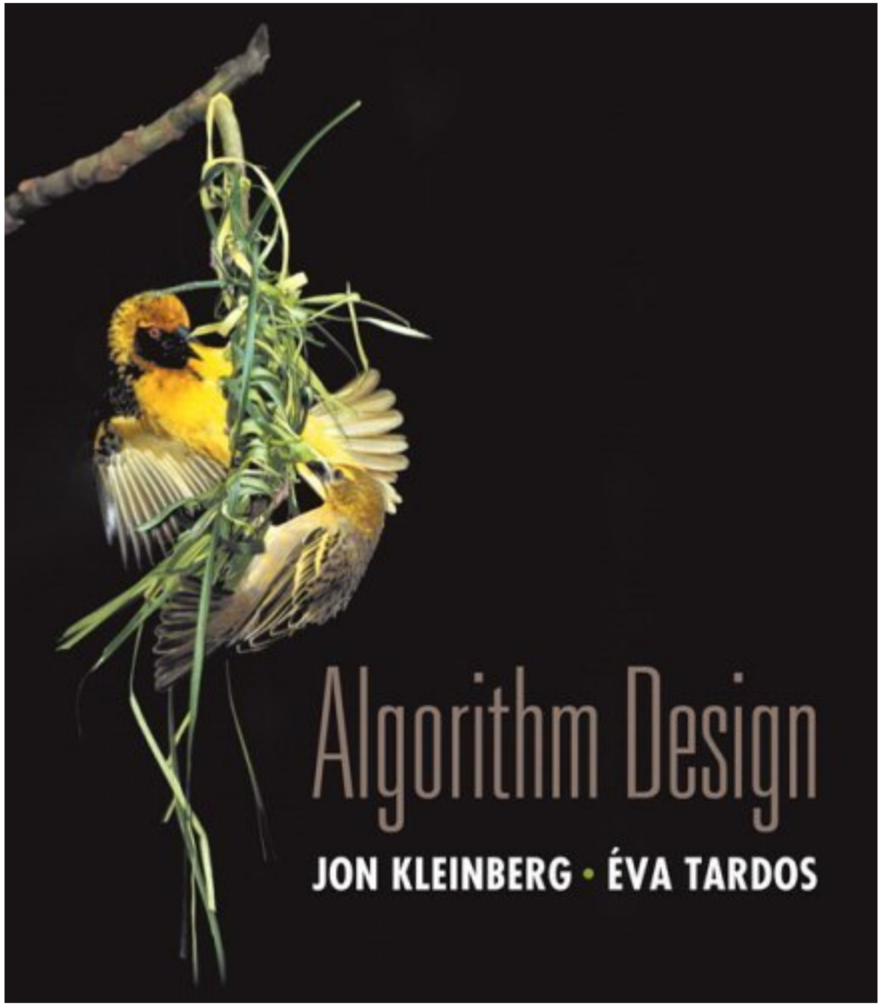
**Theorem.** [Menger 1927] The max number of edge-disjoint s-t paths is equal to the min number of edges whose removal disconnects t from s.

**Pf.** $\geq$

- Suppose max number of edge-disjoint paths is k.
- Then max flow value is k.
- Max-flow min-cut $\Rightarrow$ cut (A, B) of capacity k.
- Let F be set of edges going from A to B.
- |F| = k and disconnects t from s. ▪







# NP and Computational Intractability

PEARSON
Addison
Wesley

Slides by Kevin Wayne.

# Polynomial-Time Reduction

Purpose. Classify problems according to relative difficulty.

Design algorithms. If $X \leq_P Y$ and $Y$ can be solved in polynomial-time, then $X$ can also be solved in polynomial time.

Establish intractability. If $X \leq_P Y$ and $X$ cannot be solved in polynomial-time, then $Y$ cannot be solved in polynomial time.

Establish equivalence. If $X \leq_P Y$ and $Y \leq_P X$, we use notation $X \equiv_P Y$.

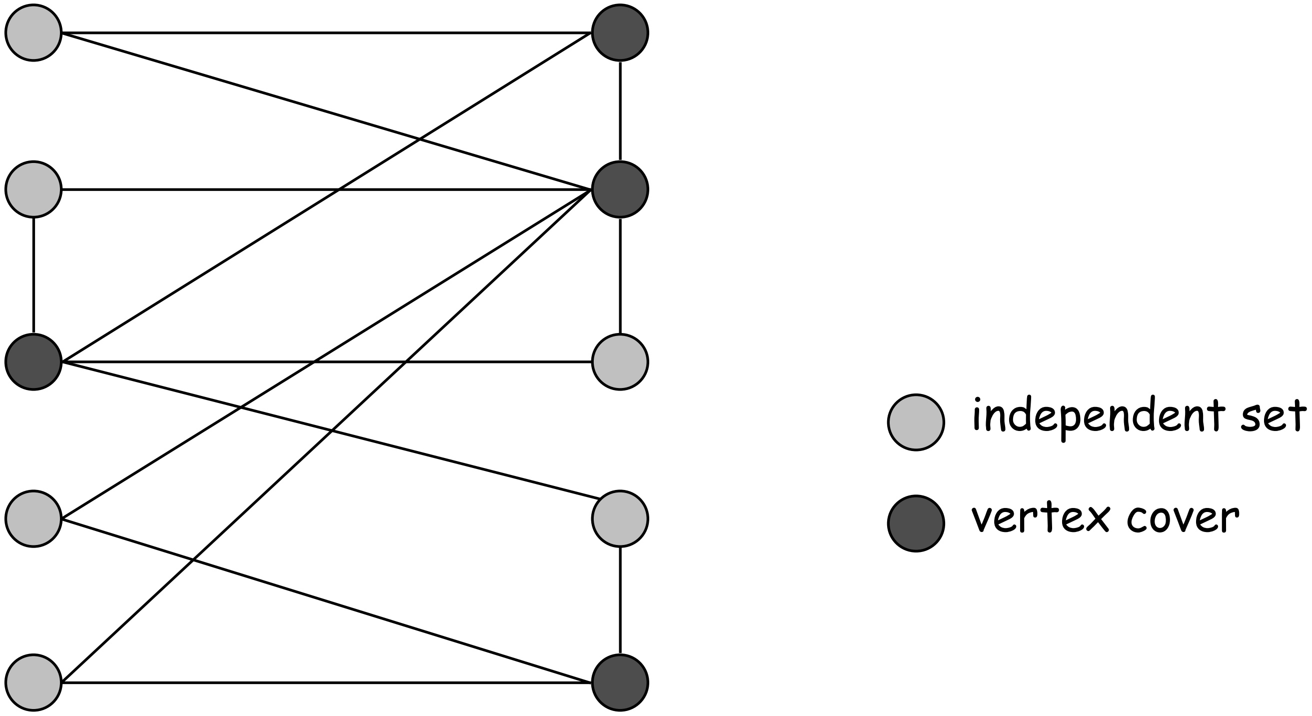
↑

up to cost of reduction

# Vertex Cover and Independent Set

**Claim.** VERTEX-COVER $\equiv_P$ INDEPENDENT-SET.

**Pf.** We show S is an independent set iff V – S is a vertex cover.



# Vertex Cover and Independent Set

Claim. VERTEX-COVER $\equiv_P$ INDEPENDENT-SET.

Pf. We show S is an independent set iff V – S is a vertex cover.

$\Rightarrow$

- Let S be any independent set.
- Consider an arbitrary edge (u, v).
- S independent $\Rightarrow$ $u \notin S$ or $v \notin S$ $\Rightarrow$ $u \in V - S$ or $v \in V - S$.
- Thus, V – S covers (u, v).

$\Leftarrow$

- Let V – S be any vertex cover.
- Consider two nodes $u \in S$ and $v \in S$.
- Observe that $(u, v) \notin E$ since V – S is a vertex cover.
- Thus, no two nodes in S are joined by an edge $\Rightarrow$ S independent set. ▪

# Set Cover

SET COVER: Given a set U of elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of U, and an integer k, does there exist a collection of $\leq$ k of these sets whose union is equal to U?

Sample application.

- m available pieces of software.
- Set U of n capabilities that we would like our system to have.
- The ith piece of software provides the set $S_i \subseteq U$ of capabilities.
- Goal: achieve all n capabilities using fewest pieces of software.

Ex:

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$k = 2$

$S_1 = \{3, 7\}$

$S_2 = \{3, 4, 5, 6\}$

$S_3 = \{1\}$

$S_4 = \{2, 4\}$

$S_5 = \{5\}$

$S_6 = \{1, 2, 6, 7\}$

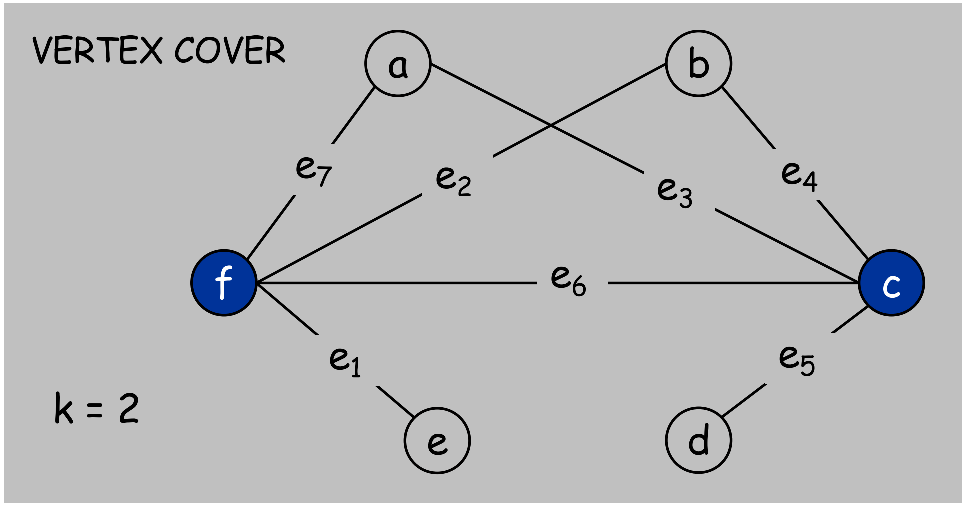
# Vertex Cover Reduces to Set Cover

**Claim.** VERTEX-COVER $\le_P$ SET-COVER.

**Pf.** Given a VERTEX-COVER instance $G = (V, E)$, $k$, we construct a set cover instance whose size equals the size of the vertex cover instance.

**Construction.**

- Create SET-COVER instance:
  - $k = k$, $U = E$, $S_v = \{e \in E : e \text{ incident to } v\}$
- Set-cover of size $\le k$ iff vertex cover of size $\le k$. ▪



SET COVER

$U = \{1, 2, 3, 4, 5, 6, 7\}$

$k = 2$

$S_a = \{3, 7\}$ $S_b = \{2, 4\}$

$S_c = \{3, 4, 5, 6\}$ $S_d = \{5\}$

$S_e = \{1\}$ $S_f = \{1, 2, 6, 7\}$

# Satisfiability

**Literal:** A Boolean variable or its negation. $x_i$ or $\overline{x_i}$

**Clause:** A disjunction of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$

**Conjunctive normal form:** A propositional formula $\Phi$ that is the conjunction of clauses. $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

**SAT:** Given CNF formula $\Phi$, does it have a satisfying truth assignment?

**3-SAT:** SAT where each clause contains exactly 3 literals.

↑ each corresponds to a different variable

**Ex:** $\left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(x_2 \vee x_3\right) \wedge \left(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}\right)$

**Yes:** $x_1$ = true, $x_2$ = true $x_3$ = false.

# 3 Satisfiability Reduces to Independent Set

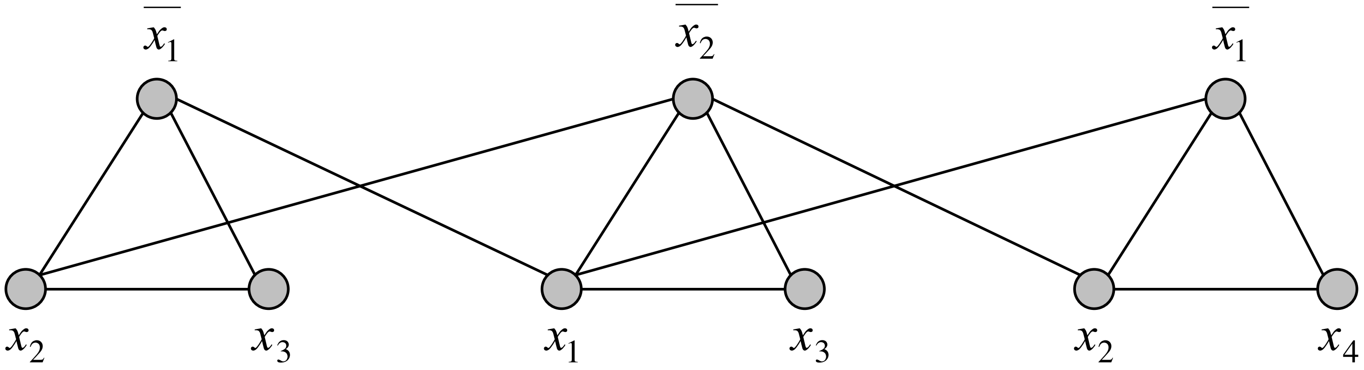
**Claim.** 3-SAT $\leq_P$ INDEPENDENT-SET.

**Pf.** Given an instance $\Phi$ of 3-SAT, we construct an instance (G, k) of INDEPENDENT-SET that has an independent set of size k iff $\Phi$ is satisfiable.

**Construction.**

- G contains 3 vertices for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.

G



k = 3

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

# Review

Basic reduction strategies.

- Simple equivalence: INDEPENDENT-SET $\equiv_P$ VERTEX-COVER.
- Special case to general case: VERTEX-COVER $\leq_P$ SET-COVER.
- Encoding with gadgets: 3-SAT $\leq_P$ INDEPENDENT-SET.

Transitivity. If X $\leq_P$ Y and Y $\leq_P$ Z, then X $\leq_P$ Z.

Pf idea. Compose the two algorithms.

Ex: 3-SAT $\leq_P$ INDEPENDENT-SET $\leq_P$ VERTEX-COVER $\leq_P$ SET-COVER.

# Decision Problems

Decision problem.
- X is a set of strings.
- Instance: string s.
- Algorithm A solves problem X: A(s) = `yes` iff s ∈ X.

Polynomial time. Algorithm A runs in poly-time if for every string s, A(s) terminates in at most p(|s|) "steps", where p(·) is some polynomial.

↑
length of s

Def. Algorithm C(s, t) is a certifier for problem X if for every string s, s ∈ X iff there exists a string t such that C(s, t) = `yes`.

NP. Decision problems for which there exists a poly-time certifier.

# Certifiers and Certificates: 3-Satisfiability

**SAT.** Given a CNF formula $\Phi$, is there a satisfying assignment?

**Certificate.** An assignment of truth values to the n boolean variables.

**Certifier.** Check that each clause in $\Phi$ has at least one true literal.

**Ex.**

$$\left( \overline{x_1} \vee x_2 \vee x_3 \right) \wedge \left( x_1 \vee \overline{x_2} \vee x_3 \right) \wedge \left( x_1 \vee x_2 \vee x_4 \right) \wedge \left( \overline{x_1} \vee \overline{x_3} \vee \overline{x_4} \right)$$

instance s

$$x_1 = 1,\ x_2 = 1,\ x_3 = 0,\ x_4 = 1$$

certificate t

**Conclusion.** SAT is in NP.

# P, NP, EXP

P. Decision problems for which there is a poly-time algorithm.

EXP. Decision problems for which there is an exponential-time algorithm.

NP. Decision problems for which there is a poly-time certifier.

Claim. P $\subseteq$ NP.

Pf. Consider any problem X in P.

- By definition, there exists a poly-time algorithm A(s) that solves X.
- Certificate: t = $\varepsilon$, certifier C(s, t) = A(s). ▪

Claim. NP $\subseteq$ EXP.

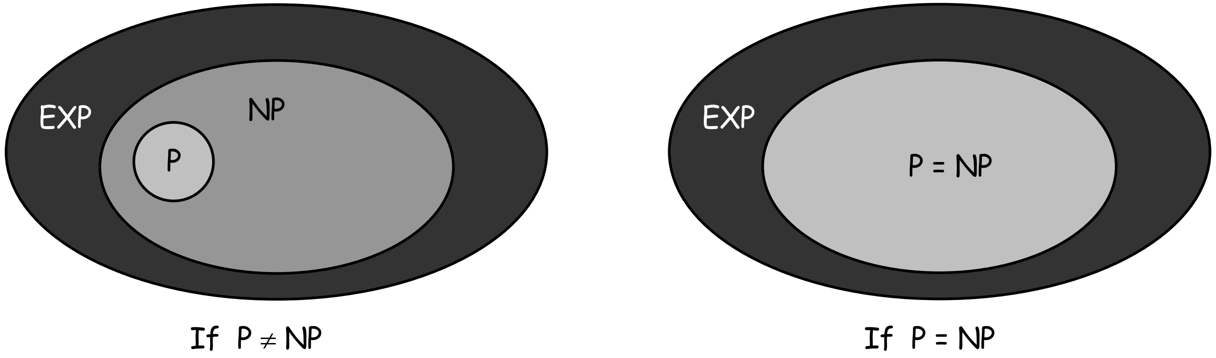
Pf. Consider any problem X in NP.

- By definition, there exists a poly-time certifier C(s, t) for X.
- To solve input s, run C(s, t) on all strings t with $|t| \leq p(|s|)$.
- Return `yes`, if C(s, t) returns `yes` for any of these. ▪

# The Main Question: P Versus NP

Does P = NP? [Cook 1971, Edmonds, Levin, Yablonski, Gödel]

- Is the decision problem as easy as the certification problem?
- Clay $1 million prize.



would break RSA cryptography
(and potentially collapse economy)

If yes: Efficient algorithms for 3-COLOR, TSP, FACTOR, SAT, …

If no: No efficient algorithms possible for 3-COLOR, TSP, SAT, …

Consensus opinion on P = NP? Probably no.

# NP-Complete

**NP-complete.** A problem Y in NP with the property that for every problem X in NP, $X \le_p Y$.

**Theorem.** Suppose Y is an NP-complete problem. Then Y is solvable in poly-time iff P = NP.

**Pf.** $\Leftarrow$ If P = NP then Y can be solved in poly-time since Y is in NP.

**Pf.** $\Rightarrow$ Suppose Y can be solved in poly-time.

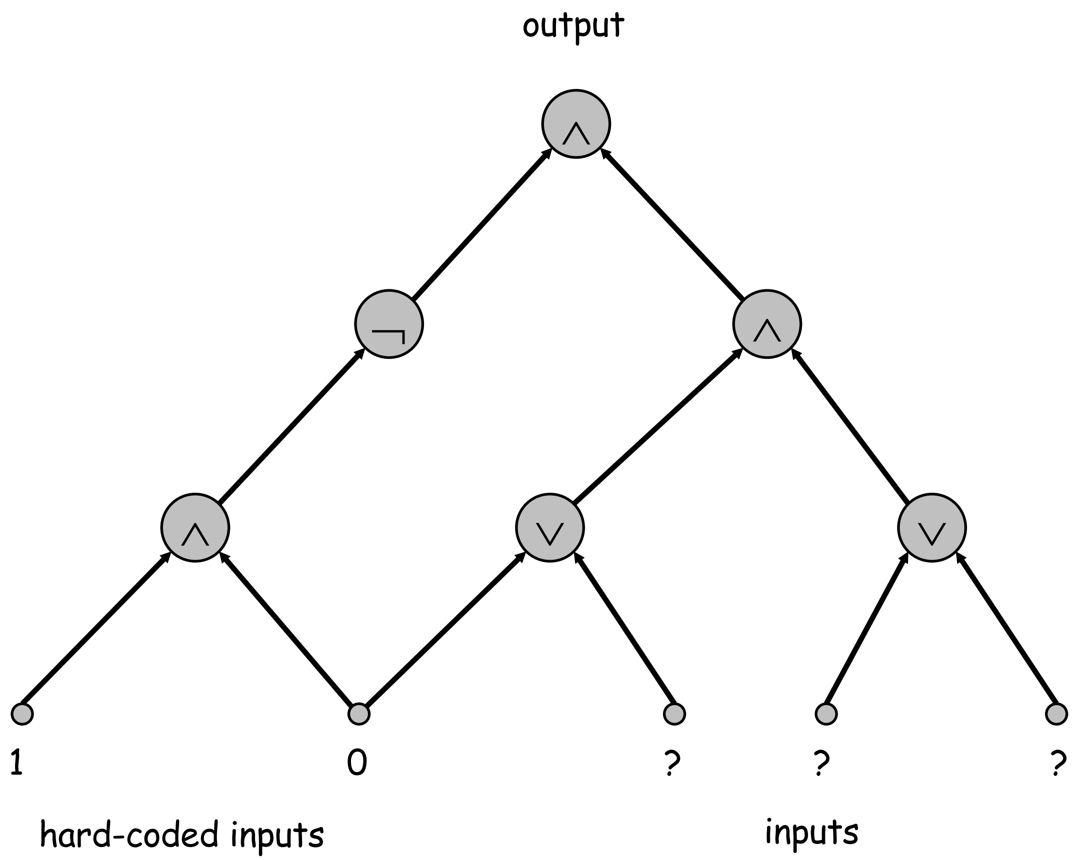
- Let X be any problem in NP. Since $X \le_p Y$, we can solve X in poly-time. This implies NP $\subseteq$ P.
- We already know P $\subseteq$ NP. Thus P = NP. ▪

**Fundamental question.** Do there exist "natural" NP-complete problems?

# Circuit Satisfiability

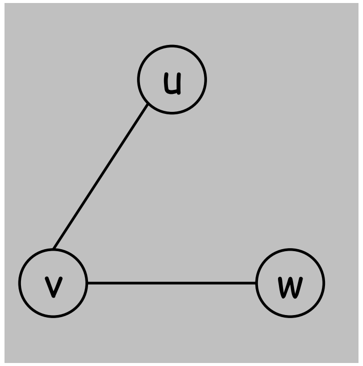
**CIRCUIT-SAT.** Given a combinational circuit built out of AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1?

yes: 1 0 1

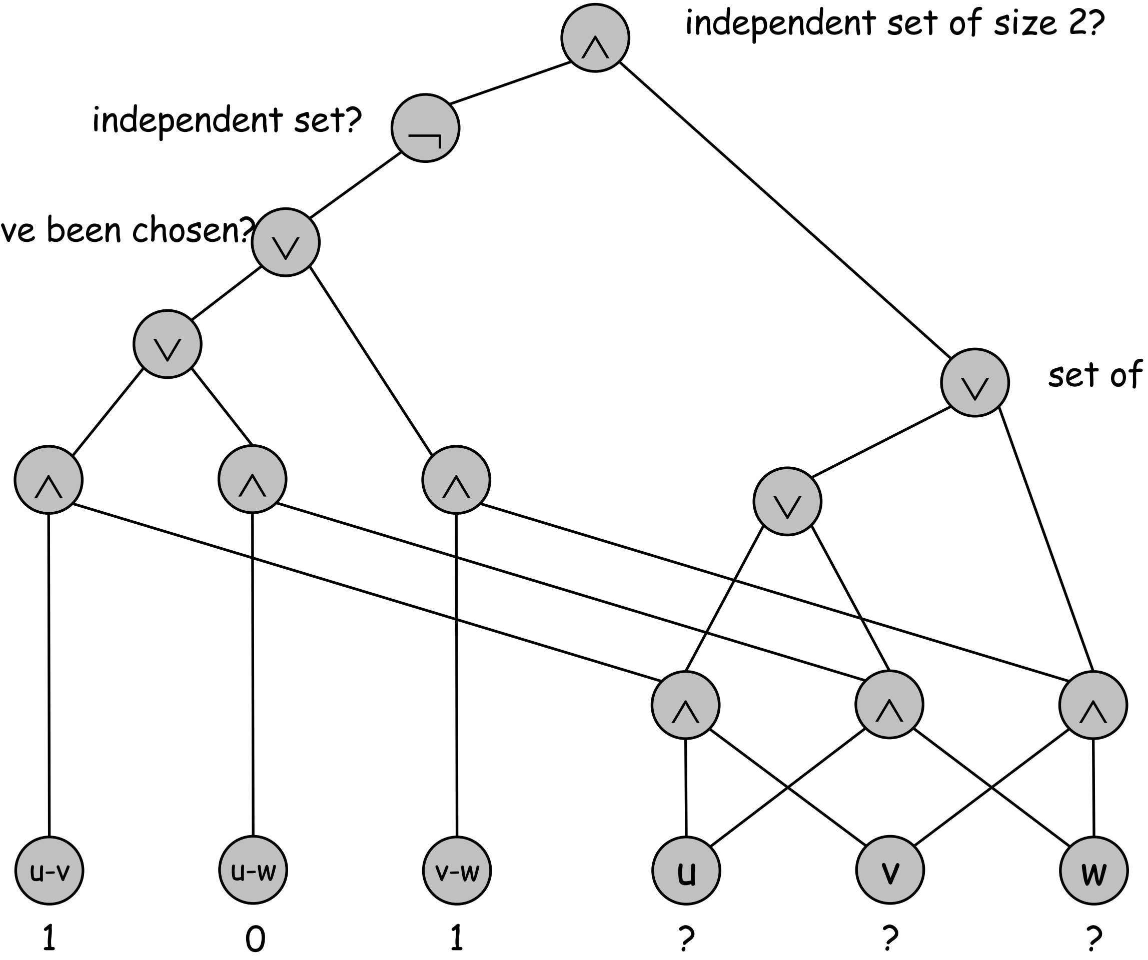


# Example

Ex. Construction below creates a circuit K whose inputs can be set so that K outputs true iff graph G has an independent set of size 2.



G = (V, E), n = 3



independent set of size 2?

independent set?

both endpoints of some edge have been chosen?

set of size 2?

$\binom{n}{2}$ hard-coded inputs (graph description)

n inputs (nodes in independent set)

# Establishing NP-Completeness

**Remark.** Once we establish first "natural" NP-complete problem, others fall like dominoes.

**Recipe to establish NP-completeness of problem Y.**

- Step 1. Show that Y is in NP.
- Step 2. Choose an NP-complete problem X.
- Step 3. Prove that $X \le_p Y$.

**Justification.** If X is an NP-complete problem, and Y is a problem in NP with the property that $X \le_P Y$ then Y is NP-complete.

**Pf.** Let W be any problem in NP. Then $W \underset{\substack{\uparrow \\ \text{by definition of} \\ \text{NP-complete}}}{\le_P} X \underset{\substack{\uparrow \\ \text{by assumption}}}{\le_P} Y$.

- By transitivity, $W \le_P Y$.
- Hence Y is NP-complete. ▪

# 3-SAT is NP-Complete

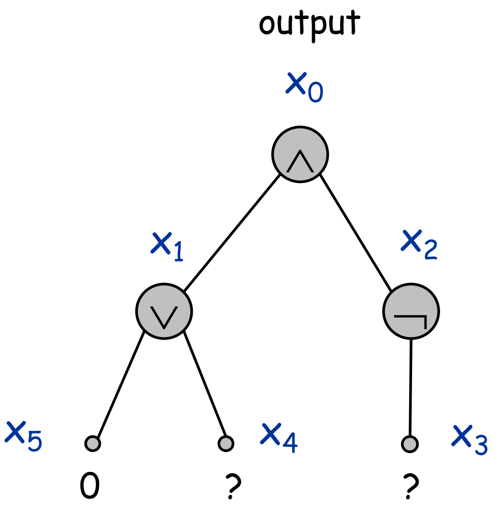
**Theorem.** 3-SAT is NP-complete.

**Pf.** Suffices to show that CIRCUIT-SAT $\le_P$ 3-SAT since 3-SAT is in NP.

- Let K be any circuit.
- Create a 3-SAT variable $x_i$ for each circuit element i.
- Make circuit compute correct values at each node:
  - $x_2 = \neg x_3$ $\Rightarrow$ add 2 clauses: $x_2 \vee x_3$ , $\overline{x_2} \vee \overline{x_3}$
  - $x_1 = x_4 \vee x_5$ $\Rightarrow$ add 3 clauses: $x_1 \vee \overline{x_4}$, $x_1 \vee \overline{x_5}$ , $\overline{x_1} \vee x_4 \vee x_5$
  - $x_0 = x_1 \wedge x_2$ $\Rightarrow$ add 3 clauses: $\overline{x_0} \vee x_1$, $\overline{x_0} \vee x_2$, $x_0 \vee \overline{x_1} \vee \overline{x_2}$

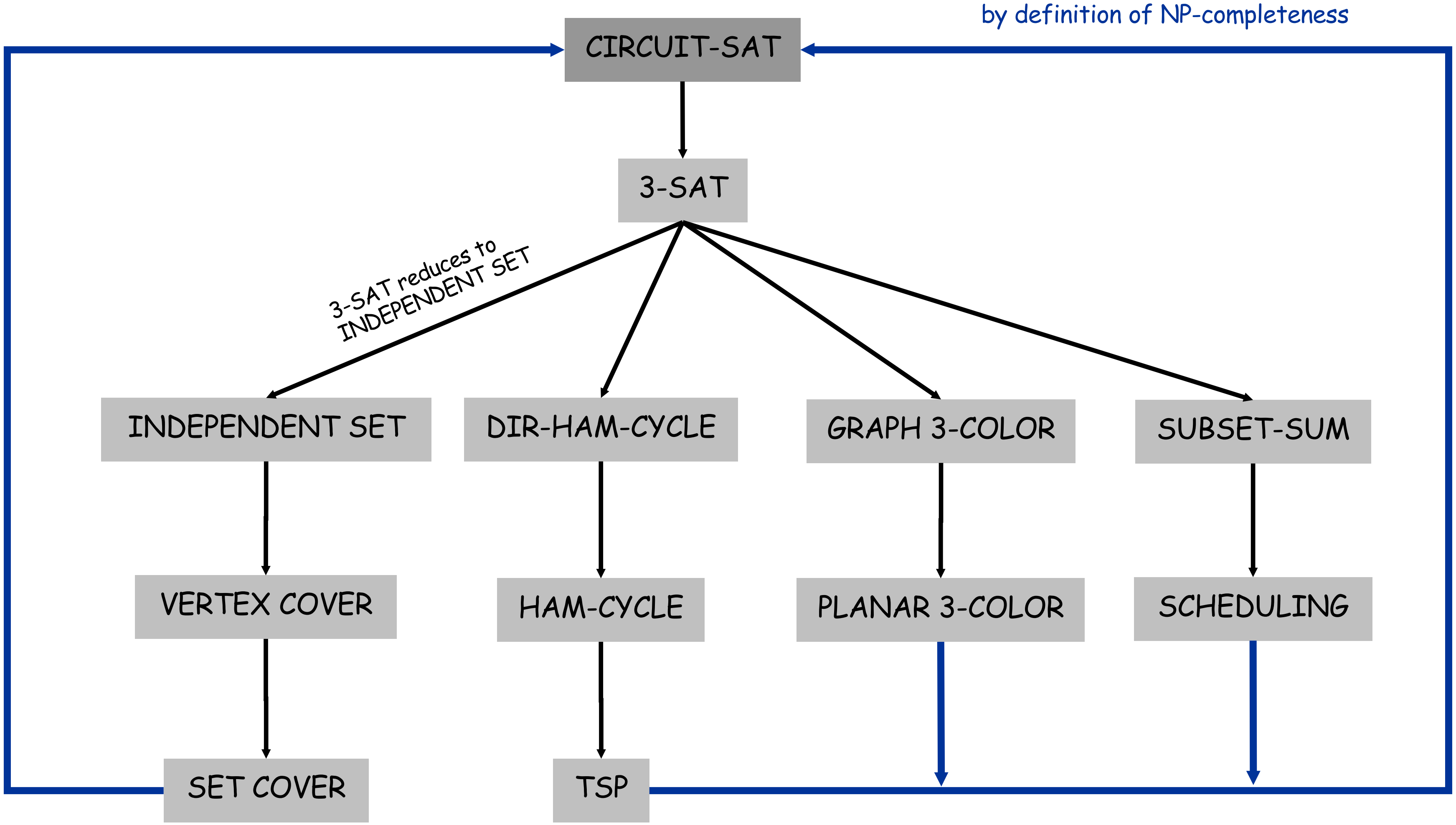
- Hard-coded input values and output value.
  - $x_5 = 0$ $\Rightarrow$ add 1 clause: $\overline{x_5}$
  - $x_0 = 1$ $\Rightarrow$ add 1 clause: $x_0$

- Final step: turn clauses of length < 3 into clauses of length exactly 3. ▪



# NP-Completeness

Observation. All problems below are NP-complete and polynomial reduce to one another!



by definition of NP-completeness

CIRCUIT-SAT

3-SAT

3-SAT reduces to INDEPENDENT SET

INDEPENDENT SET

DIR-HAM-CYCLE

GRAPH 3-COLOR

SUBSET-SUM

VERTEX COVER

HAM-CYCLE

PLANAR 3-COLOR

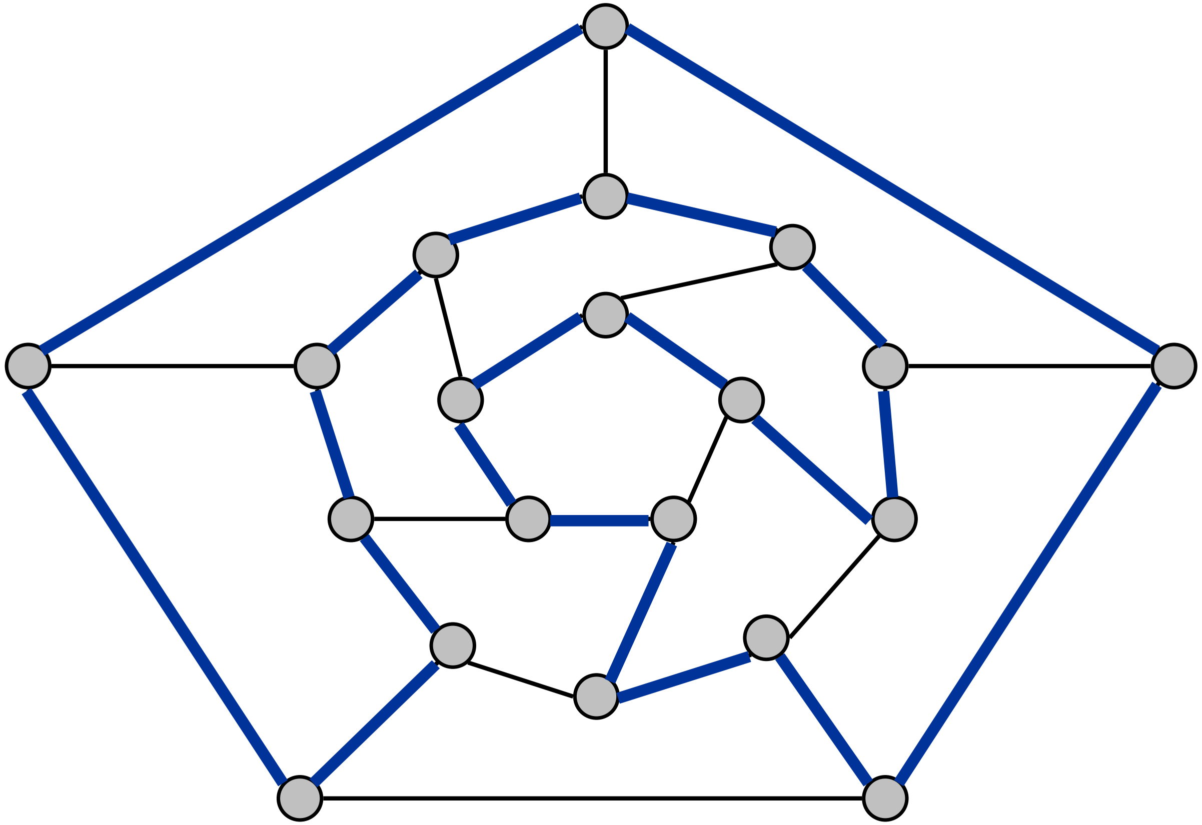
SCHEDULING

SET COVER

TSP

# Hamiltonian Cycle

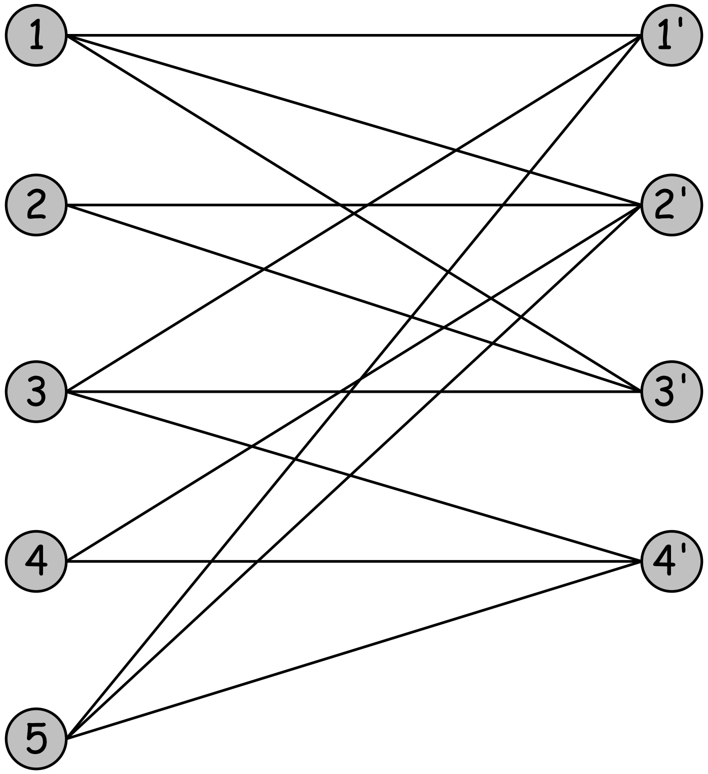
**HAM-CYCLE:** given an undirected graph G = (V, E), does there exist a simple cycle $\Gamma$ that contains every node in V.



YES: vertices and faces of a dodecahedron.

# Hamiltonian Cycle

HAM-CYCLE: given an undirected graph G = (V, E), does there exist a simple cycle $\Gamma$ that contains every node in V.



NO: bipartite graph with odd number of nodes.

# Traveling Salesperson Problem

**TSP.** Given a set of n cities and a pairwise distance function d(u, v), is there a tour of length ≤ D?

**HAM-CYCLE:** given a graph G = (V, E), does there exists a simple cycle that contains every node in V?

**Claim.** HAM-CYCLE $\leq_P$ TSP.

**Pf.**

- Given instance G = (V, E) of HAM-CYCLE, create n cities with distance function

$$d(u,\ v) \ = \begin{cases} 1 & \text{if } (u,\ v) \in E \\ 2 & \text{if } (u,\ v) \notin E \end{cases}$$

- TSP instance has tour of length ≤ n iff G is Hamiltonian. ▪

**Remark.** TSP instance in reduction satisfies Δ-inequality.

# 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.

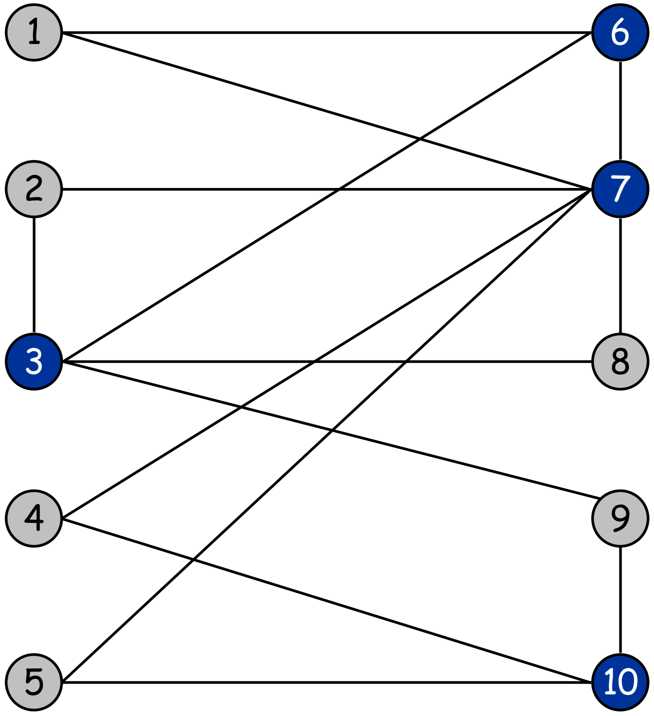
Must sacrifice one of three desired features.

- 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.

# 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: Algorithm

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

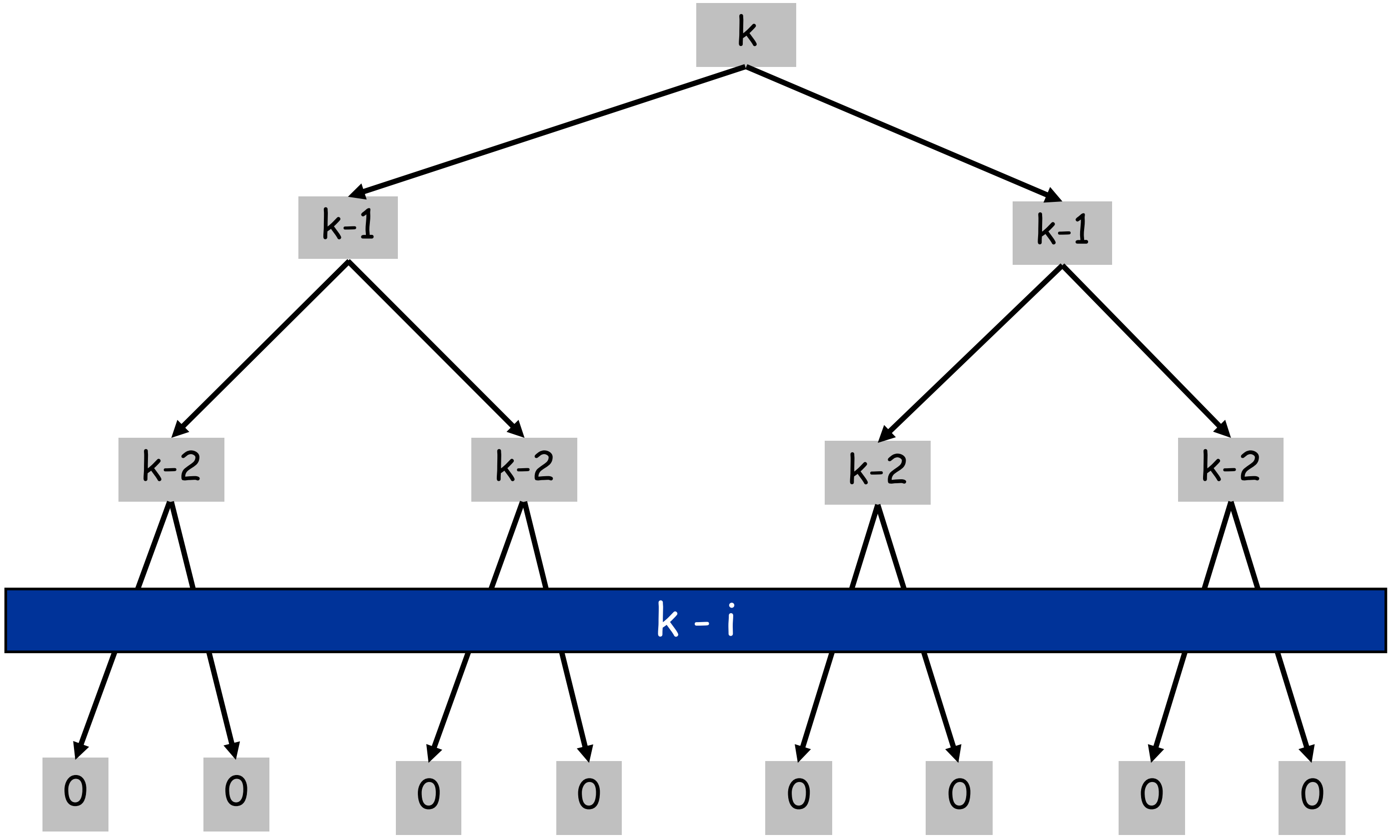
```
boolean Vertex-Cover(G, k) {
   if (G contains no edges)    return true
   if (G contains ≥ kn edges) return false

   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. ▪

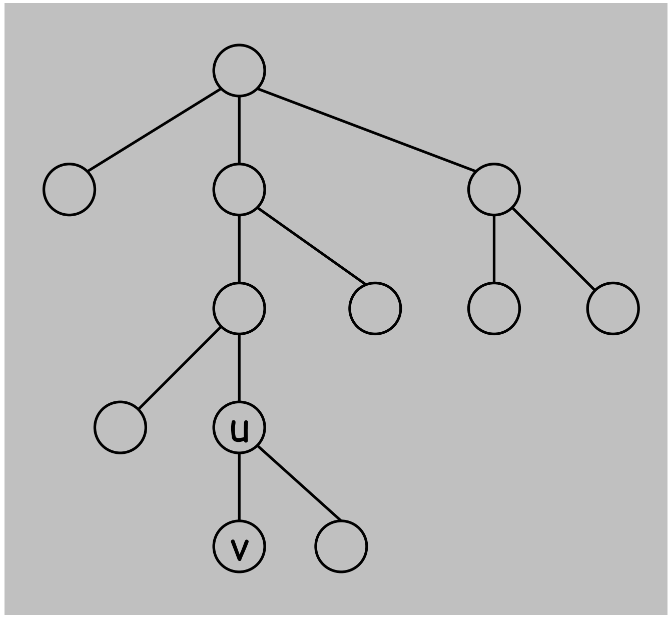
# Finding Small Vertex Covers: Recursion Tree

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



# 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

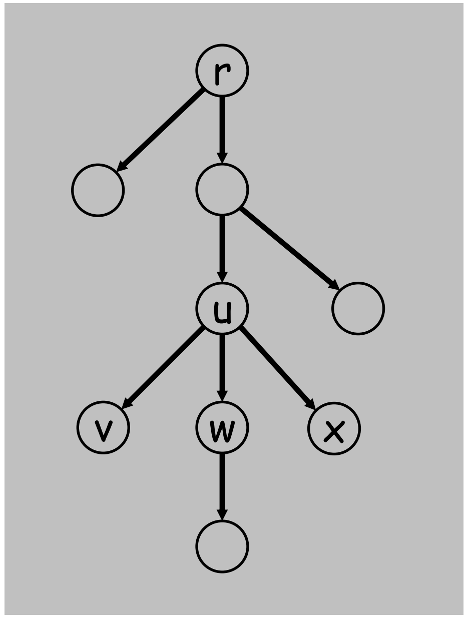
Weighted independent set on trees. Given a tree and node weights $w_v > 0$, find an independent set S that maximizes $\Sigma_{v \in S}\, w_v$.

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] = $\Sigma_{v \in children(u)}$ $M_{out}$[v]  +  $w_v$
         $M_{out}$[u] = $\Sigma_{v \in children(u)}$ 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. ▪

# 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!

# 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($m$, $n$, $t_1,t_2,\ldots,t_n$) {
    for $i$ = 1 to $m$ {
        $L_i \leftarrow 0$                ⟵ load on machine i
        $J(i) \leftarrow \phi$             ⟵ jobs assigned to machine i
    }

    for $j$ = 1 to $n$ {
        $i = \text{argmin}_k \ L_k$               ⟵ machine i has smallest load
        $J(i) \leftarrow J(i) \cup \{j\}$        ⟵ assign job j to machine i
        $L_i \leftarrow L_i + t_j$                ⟵ 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.

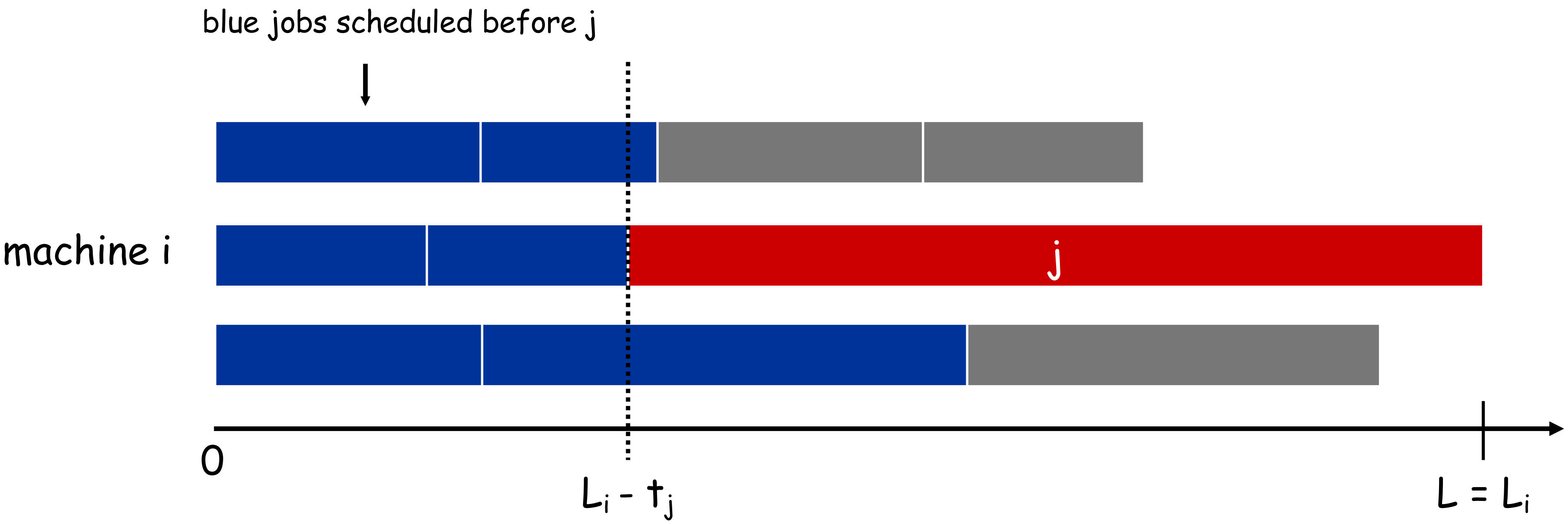
Not very strong lower bound. What if one job is very big and others are small jobs ? ▪

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

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

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

$\uparrow$

Lemma 1

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