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

- **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 \( 1, 2, ..., p(j) \)

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

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v_j + OPT(p(j)), \, OPT(j-1) \} & \text{otherwise}
\end{cases}
\]
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
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 \(\Rightarrow\) 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.

- 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?
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: $E + cL$, for some constant $c > 0$. 

![Graph showing data points and fitted line segments.](image-url)
Dynamic Programming: Multiway Choice

Notation.

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

To compute \( \text{OPT}(j) \):

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

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\min_{1 \leq i \leq j} \left\{ e(i,j) + c + \text{OPT}(i-1) \right\} & \text{otherwise}
\end{cases}
\]
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.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

$W = 11$

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. $OPT(i, w) = \text{max profit subset of items } 1, \ldots, i \text{ with weight limit } w.$

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

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

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

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

W + 1

\( W = 11 \)
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.

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

How similar are two strings?

- occurance
- occurrence

- occurance
- occurrence

5 mismatches, 1 gap

- occ - ur r a nce
- occurrence

1 mismatch, 1 gap

- occ - ur r a nce
- occurrence

0 mismatches, 3 gaps
Applications.
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

- 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 $j > j'$.

\[
\text{cost}(M) = \sum_{(x_i, y_j) \in M} \alpha \, x_i y_j + \sum_{i : x_i \text{ unmatched}} \delta_i + \sum_{j : y_j \text{ unmatched}} \delta_j
\]

**Ex:** CTACCG vs. TACATG.

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

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

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

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

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 $w$. 

![Graph Diagram]
**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. $OPT(i, v) = \text{length of shortest } v \rightarrow t \text{ path } P \text{ using at most } i \text{ edges.}$

- **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 $w \rightarrow t$ path using at most $i-1$ edges

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

Remark. By previous observation, if no negative cycles, then $OPT(n-1, v) = \text{length of shortest } v \rightarrow t \text{ path.}$
Shortest Paths: Implementation

\[
\text{Shortest-Path}(G, t) \{ \\
\quad \text{foreach node } v \in V \\
\quad \quad M[0, v] \leftarrow \infty \\
\quad \quad M[0, t] \leftarrow 0 \\
\quad \text{for } i = 1 \text{ to } n-1 \\
\quad \quad \text{foreach node } v \in V \\
\quad \quad \quad M[i, v] \leftarrow M[i-1, v] \\
\quad \quad \text{foreach edge } (v, w) \in E \\
\quad \quad \quad M[i, v] \leftarrow \min \{ M[i, v], M[i-1, w] + c_{vw} \} \\
\}\]

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

Finding the shortest paths. Maintain a "successor" for each table entry.
Network Flow
Flow network.

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

Minimum Cut Problem
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 $s$.

$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)$$
**Flows and Cuts**

**Weak duality.** Let $f$ be any flow. Then, for any $s$-$t$ cut $(A, B)$ we have $v(f) \leq \text{cap}(A, B)$.

**Pf.**

\[
v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \leq \sum_{e \text{ out of } A} f(e) \leq \sum_{e \text{ out of } A} c(e) = \text{cap}(A, B).
\]
**Certificate of Optimality**

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

\[
\begin{align*}
\text{Value of flow} &= 28 \\
\text{Cut capacity} &= 28 \implies \text{Flow value} \leq 28
\end{align*}
\]
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 $v(f) = \text{cap}(A, B)$.
(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) ⇒ (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 $f$, $t \not\in A$.

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

$$= \sum_{e \text{ out of } A} c(e)$$

$$= \text{cap}(A, B)$$
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 $\nu(f^*) \leq nC$ 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 $t$. 
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. ≥**
- 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
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 \( X \equiv_p Y \).

\[ \uparrow \]
up to cost of reduction
Claim. \( \text{VERTEX-COVER} \equiv_p \text{INDEPENDENT-SET}. \)

Pf. We show \( S \) is an independent set iff \( V - S \) is a vertex cover.
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 $i$th piece of software provides the set $S_i \subseteq U$ of capabilities.
- Goal: achieve all $n$ capabilities using fewest pieces of software.

Ex:

<table>
<thead>
<tr>
<th>$U$ = ${1, 2, 3, 4, 5, 6, 7}$</th>
<th>$k$ = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1 = {3, 7}$</td>
<td>$S_4 = {2, 4}$</td>
</tr>
<tr>
<td>$S_2 = {3, 4, 5, 6}$</td>
<td>$S_5 = {5}$</td>
</tr>
<tr>
<td>$S_3 = {1}$</td>
<td>$S_6 = {1, 2, 6, 7}$</td>
</tr>
</tbody>
</table>
**Vertex Cover Reduces to Set Cover**

**Claim.** VERTEX-COVER $\leq_p$ 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:
  - $k = k$, $U = E$, $S_v = \{e \in E : e$ incident to $v\}$
- Set-cover of size $\leq k$ iff vertex cover of size $\leq k$.

**Example Diagram**

**VERTEX COVER**

- Vertices: $a, b, c, e, d, f$
- Edges: $e_1, e_2, e_3, e_4, e_5, e_6, e_7$
- $k = 2$

**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. \( C_j = x_1 \lor \overline{x}_2 \lor x_3 \)

Conjunctive normal form: A propositional formula \( \Phi \) that is the conjunction of clauses. \( \Phi = C_1 \land C_2 \land C_3 \land C_4 \)

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: \((\overline{x}_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2 \lor x_3) \land (x_2 \lor x_3) \land (\overline{x}_1 \lor \overline{x}_2 \lor \overline{x}_3)\)

Yes: \( x_1 = \text{true}, x_2 = \text{true} x_3 = \text{false} \).
Claim. 3-SAT ≤ₚ INDEPENDENT-SET.
Pf. Given an instance Φ of 3-SAT, we construct an instance (G, k) of INDEPENDENT-SET that has an independent set of size k iff Φ 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.

\[
\Phi = \left( \overline{x_1} \lor x_2 \lor x_3 \right) \land \left( x_1 \lor \overline{x_2} \lor x_3 \right) \land \left( \overline{x_1} \lor x_2 \lor x_4 \right)
\]
Review

Basic reduction strategies.

- Simple equivalence: $\text{INDEPENDENT-SET} \equiv_p \text{VERTEX-COVER}$.
- Special case to general case: $\text{VERTEX-COVER} \leq_p \text{SET-COVER}$.
- Encoding with gadgets: $3\text{-SAT} \leq_p \text{INDEPENDENT-SET}$.

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

Pf idea. Compose the two algorithms.

Ex: $3\text{-SAT} \leq_p \text{INDEPENDENT-SET} \leq_p \text{VERTEX-COVER} \leq_p \text{SET-COVER}$.
Decision Problems

Decision problem.
- X is a set of strings.
- Instance: string s.
- Algorithm A solves problem X: $A(s) = \text{yes}$ iff $s \in X$.

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

Def. Algorithm $C(s, t)$ is a certifier for problem X if for every string $s$, $s \in X$ iff there exists a string $t$ such that $C(s, t) = \text{yes}$.

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.**

\[
(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x}_2 \lor x_3) \land (x_1 \lor x_2 \lor x_4) \land (\overline{x}_1 \lor \overline{x}_3 \lor \overline{x}_4)
\]

instance $s$

\[
\begin{align*}
x_1 &= 1, & x_2 &= 1, & x_3 &= 0, & x_4 &= 1
\end{align*}
\]

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 ⊆ 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 ⊆ 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.

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.

If P ≠ NP

If P = NP

would break RSA cryptography
(and potentially collapse economy)
**NP-Complete**

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

**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 $X \leq_p Y$, 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.

Graph $G = (V, E)$, $n = 3$
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 $X \leq_P Y$.

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

Pf. Let $W$ be any problem in NP. Then $W \leq_P X \leq_P Y$.

- By transitivity, $W \leq_P Y$.
- Hence $Y$ is NP-complete. ◦
Theorem. 3-SAT is NP-complete.

Pf. Suffices to show that CIRCUIT-SAT $\leq_p$ 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:
  - $x_2 = \neg x_3 \implies$ add 2 clauses: $x_2 \lor \overline{x_3}, \overline{x_2} \lor \overline{x_3}$
  - $x_1 = x_4 \lor x_5 \implies$ add 3 clauses: $x_1 \lor \overline{x_4}, x_1 \lor \overline{x_5}, x_1 \lor x_4 \lor x_5$
  - $x_0 = x_1 \land x_2 \implies$ add 3 clauses: $\overline{x_0} \lor x_1, \overline{x_0} \lor x_2, x_0 \lor \overline{x_1} \lor \overline{x_2}$

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

- Final step: turn clauses of length < 3 into clauses of length exactly 3. ·
Observation. All problems below are NP-complete and polynomial reduce to one another!

NP-Completeness

CIRCUIT-SAT

3-SAT

3-SAT reduces to INDEPENDENT SET

INDEPENDENT SET

VERTEX COVER

SET COVER

DIR-HAM-CYCLE

HAM-CYCLE

TSP

GRAPH 3-COLOR

PLANAR 3-COLOR

SUBSET-SUM

SCHEDULING

by definition of NP-completeness
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 \( \leq D \)?

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

**Claim.** \( \text{HAM-CYCLE} \leq_p \text{TSP} \).

**Pf.**
- Given instance \( G = (V, E) \) of \( \text{HAM-CYCLE} \), 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 \( \leq n \) iff \( G \) is Hamiltonian.

**Remark.** TSP instance in reduction satisfies \( \Delta \)-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 \( \binom{n}{k} = O(n^k) \) 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 $O(2^k \cdot kn)$ time.

```java
boolean Vertex-Cover(G, k) {
    if (G contains no edges) return true
    if (G contains $\geq kn$ 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

\[ T(n, k) \leq \begin{cases} 
  cn & \text{if } k = 1 \\
  2T(n, k-1) + ckn & \text{if } k > 1
\end{cases} \Rightarrow T(n, k) \leq 2^k c k n \]
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.
\[ \text{degree} = 1 \]

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).

```plaintext
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 $\sum_{v \in S} w_v$.

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 $u$.

\[
OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v)
\]

\[
OPT_{out}(u) = \sum_{v \in \text{children}(u)} \max \{OPT_{in}(v), OPT_{out}(v)\}
\]
Independent Set on Trees: Greedy Algorithm

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

```plaintext
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 {
         $M_{in}[u] = \sum_{v \in \text{children}(u)} M_{out}[v] + w_v$
         $M_{out}[u] = \sum_{v \in \text{children}(u)} \max(M_{out}[v], M_{in}[v])$
      }
   }
   return $\max(M_{in}[r], M_{out}[r])$
}

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

**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.

```plaintext
List-Scheduling(m, n, t_1, t_2, ..., t_n) {
    for i = 1 to m {
        L_i ← 0 ← load on machine i
        J(i) ← ∅ ← jobs assigned to machine i
    }

    for j = 1 to n {
        i = argmin_k L_k ← machine i has smallest load
        J(i) ← J(i) ∪ {j} ← assign job j to machine i
        L_i ← 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 $\sum_j t_j$.
- 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? ·
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 $1 \leq k \leq m$. 

![Diagram showing load balancing with blue jobs scheduled before job $j$ and machine $i$.]
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 $m$:

$$L_i - t_j \leq \frac{1}{m} \sum_k L_k \leq \frac{1}{m} \sum_j t_j \leq L^*$$

**Lemma 2**

- Now $L_i = (L_i - t_j) + t_j \leq 2L^*$.

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