CMSC 451: Introduction & Stable Marriage

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Chapter 1 of Algorithm Design by Kleinberg & Tardos.
Course overview

Objective

Study algorithms for interesting computational problems, focusing on principles used to design those algorithms.

- Not as focused on recurrence relations as 351.
- Nor on getting the best, smallest runtime.
- Usually interested in proving correctness and efficiency (polynomial time)
• Basically cover the entire book (some chapters more than others)
• We’ll skip chapters 9 and 10, although there may be an extra credit assignment related to them.
• Chapters 1–3 should mostly be review; we’ll cover these over the next two classes.
• Then we’ll discuss several general techniques for designing algorithms:
  1. Greedy algorithms
  2. Divide and conquer
  3. Dynamic programming
  4. Linear programming
  5. Network flow

A surprisingly large number of algorithms are based on one of these techniques.
• We’ll cover the theory of NP-completeness and the $P=NP$ problem.

• Finally, we’ll talk about how to deal with problems for which there is no known good solution (Approximation algorithms, local search).
Course work:

- 2 midterm exams (dates are on the syllabus; each 25% of grade)
- 1 comprehensive final exam (40% of grade)
- ~12 homeworks (10% of grade)

**Reading** for the semester is on the syllabus: I *strongly* encourage you to keep up with the reading. Nothing will make the class easier than that.
Homework

Goal with the homeworks is to get some practice designing algorithms and writing up clear solutions.

- Some problems are easy, some are challenging.
- Exams will be closed book, closed note, but you may use your graded homework.
- Messy or poorly written homeworks will not be graded.
- You can work together on the homeworks, but you must write up your own solutions independently.
- Homeworks for the first half of the semester are on the handout.
Why study algorithms?

It’s true that usually a very small portion of “real” programs deal with the kinds of algorithms we will study.

But often it’s a very important part: choosing the right algorithm can often lead to a dramatic increase in performance.

Changing the algorithm can be far more effective than simply tweaking the code.

Finally, algorithms are fun and interesting in their own right.
Our focus:

**Correctness:** Does the algorithm do what it is supposed to do? Can we prove it?

**Efficiency:** Does the algorithm have a runtime that is polynomially bounded? Is it as fast as possible?
Describing algorithms:

- Keep it as simple as possible, but no simpler. Difficult algorithms require more detail than intuitively obvious ones. No need to “write assembly code”: high-level statements that can obviously be implemented are fine.

- Though this can depend on the context:
  
  If $S$ is a set, we can generally assume we can iterate its elements, test if $p$ is in $S$, etc. In some contexts we can assume calculating $S_1 \cap S_2$ is obviously easy. In others, we have to spell out how to do this. (E.g. suppose $S$ is the set of primes.)

- Knowing the level of detail to present is a bit of an art. The solved exercises in the textbook can help.
Prove the algorithm is correct:

- Again, keep it simple: briefly say why the algorithm works in general and then focus on the non-obvious parts.
- Assume you are trying to convince one of your classmates.

Analyze its efficiency:

- Ensure that it runs in polynomial time.
- Then try to give the best possible, worst-case upper bound on how many steps it will take.
Representative problems

Some of the problems we’ll discuss:

1. Interval Scheduling
2. Weighted Interval Scheduling
3. Bipartite Matching
4. Independent Set

These are “representative problems” that we’ll come back to over the semester.
Interval Scheduling

- You want to schedule jobs on a supercomputer.
- Requests take the form \((s_i, f_i)\) meaning a job that runs from time \(s_i\) to time \(f_i\).
- You get many such requests, and you want to process as many as possible, but the computer can only work on one job at a time.

Given a set \(J = \{(s_i, f_i) : i = 1, \ldots, n\}\) of job intervals, find the largest \(S \subset J\) such that no two intervals in \(S\) overlap.
Suppose you assign a weight $v_i$ to each job. This models the benefit to you of choosing that job. (Maybe it is the amount of money the customer will pay.)

Given a set $J = \{(s_i, f_i; v_i) : i = 1, \ldots, n\}$ of job intervals $(s_i, f_i)$ and their weights $v_i$, find the $S \subset J$ such that no two intervals in $S$ overlap and $\sum_{i \in S} v_i$ is maximized.

Interval Scheduling can be solved by a greedy algorithm. Weighted Interval Scheduling seems to require dynamic programming.
A graph is bipartite if its nodes can be partitioned into two sets $X$ and $Y$ so that every edge has one end in $X$ and one end in $Y$.

A matching is a set of edges $M$ such that no node appears in more than one edge in $M$. 

Bipartite Matching

Given a bipartite graph $G$, find a matching $M$ of maximum size.
Independent Set

Definition (Independent Set)
Given a graph $G = (V, E)$ an independent set is a set $S \subseteq V$ if no two nodes in $S$ are joined by an edge.

Maximum Independent Set
Given a graph $G$, find the largest independent set.

Apparently a computationally difficult problem. (No efficient algorithm known, and good reason to suspect that none exists.)
Independent Set

Definition (Independent Set)

Given a graph $G = (V, E)$ an independent set is a set $S \subseteq V$ if no two nodes in $S$ are joined by an edge.
Interval Scheduling can be written as an Independent Set problem. How?
Interval Scheduling can be written as an Independent Set problem. How?

- Define a graph $G$ with a node for each job request, and an edge if two requests overlap.
- The largest independent set $\equiv$ to the largest choice of jobs that do not overlap.
Independent Set is very general

Bipartite Matching can also be written as an independent set problem. How?

Hint: What are the constraints in choosing edges in a matching?
Independent Set is very general

Bipartite Matching can also be written as an independent set problem. How?

Hint: What are the constraints in choosing edges in a matching?

- Let $G$ be the graph we want to find a matching in.
- Define a new graph $G'$ with a node for every edge in $G$.
- Add an edge $(u, v)$ to $G'$ if the edges $u$ and $v$ in $G$ share an endpoint.
- The largest choice of independent nodes in $G' \equiv$ to largest choice of edges that do not share an endpoint in $G$. 
A representative problem

Stable Matching
A representative problem

Suppose three students are applying for jobs and have the following preferences for where they want to work:

Alice: Google, Yahoo, Microsoft
Bob: Microsoft, Google, Yahoo
Carl: Google, Microsoft, Yahoo

Each company also has its preferences for who to hire:

Google: Alice, Bob, Carl
Microsoft: Bob, Alice, Carl
Yahoo: Carl, Alice, Bob

After the hiring, everyone wants to avoid the situation where a student would rather work for, say, Microsoft, and Microsoft would rather have hired that student than the one they got.

Can we assign students to employers so that this doesn’t happen?
Rather than students & companies, we can frame the question in terms of men & women getting married:

- $M = \{m_1, m_2, \ldots, m_n\} =$ a set of $n$ men.
- $W = \{w_1, w_2, \ldots, w_n\} =$ a set of $n$ women.
- $M \times W =$ set of possible pairs of men and women.

- A matching is a set of pairs from $M \times W$ such that each man and each women appears at most once.
- A perfect matching is a set of pairs from $M \times W$ that includes each man and each women exactly once.

A perfect matching corresponds to one way to pair up the men and women.
Avoiding Instability

Each man ranks all the women; each women ranks all the men.

Want to avoid situation where some marriage is not stable:

**Definition (Instability)**

Let $S$ be a perfect matching. A pair $(m, w')$ is an instability with respect to $S$ if $m$ is not paired with $w'$ in $S$, but $m$ and $w'$ prefer each other to the people they are married to in $S$.

If a perfect matching has an instability, both $m$ and $w'$ would be happier with each other, and the marriage may break up.

Given the preference lists, can we find a perfect matching that has no instabilities?
A different view of the problem

The pair \((m, w')\) is an instability with respect to the shown matching (solid lines).
Examples

What are the stable matchings?

Example 1:

- $m$ prefers $w$ to $w'$
- $m'$ prefers $w$ to $w'$
- $w$ prefers $m$ to $m'$
- $w'$ prefers $m$ to $m'$

Example 2:

- $m$ prefers $w$ to $w'$
- $m'$ prefers $w'$ to $w$
- $w$ prefers $m'$ to $m$
- $w'$ prefers $m$ to $m'$
Examples

What are the stable matchings?

Example 1:

- $m$ prefers $w$ to $w'$
- $m'$ prefers $w$ to $w'$
- $w$ prefers $m'$ to $m'$
- $w'$ prefers $m$ to $m'$

$(m, w), (m', w')$

Example 2:

- $m$ prefers $w$ to $w'$
- $m'$ prefers $w'$ to $w$
- $w$ prefers $m'$ to $m$
- $w'$ prefers $m$ to $m'$
Examples

What are the stable matchings?

Example 1:
- $m$ prefers $w$ to $w'$
- $m'$ prefers $w$ to $w'$
- $w$ prefers $m$ to $m'$
- $w'$ prefers $m$ to $m'$

$(m, w), (m', w')$

Example 2:
- $m$ prefers $w$ to $w'$
- $m'$ prefers $w'$ to $w$
- $w$ prefers $m'$ to $m$
- $w'$ prefers $m$ to $m'$

$(m, w), (m', w')$
OR $(m', w), (m, w')$
The Gale-Shapley algorithm

**Idea:** men will successively propose to women, in order of decreasing preference.

- Initially, everyone is **free**.
- A free man $m$ proposes to the first women on his list that he has not proposed to yet.
- If she is free, she tentatively accepts his proposal and they become **engaged**.
- If she is engaged but prefers $m$ to her fiancé, she dumps her fiancé and becomes engaged to $m$. Otherwise, she tells $m$ “no”.
- Algorithm stops when no one is left **free**.

**Recap:** men propose in order of preference, women always take the best of the two choices available at any time.
Set all $m$ in $M$ and $w$ in $W$ to be free.
While there is a man who is free and hasn’t proposed to every women:

Choose such a man $m$
Let $w$ be the highest-ranked women on $m$’s list to whom $m$ has not yet proposed.

If $w$ is free, then:
  $(m, w)$ become engaged
Else $w$ is current engaged to $m'$:
  If $w$ prefers $m$ to $m'$:
    $(m, w)$ become engaged
    $m'$ becomes free
  Endif
Endif
Endwhile
Return set $S$ of engaged pairs
Theorem

The algorithm terminates after $\leq n^2$ iterations of the while loop.

Proof.

Let $\mathcal{P}(t)$ be the set of pairs $(m, w)$ such that $m$ has proposed to $w$ by the end of iteration $t$.

Because there are only $n^2$ possible pairs, $\mathcal{P}(t) \leq n^2$.

$\mathcal{P}(t)$ increases every iteration, however: $\mathcal{P}(t + 1) > \mathcal{P}(t)$.

So: there are at most $n^2$ iterations.
Implementation

To get $O(n^2)$ running time, each execution of the loop must take constant time.

- How would you find a free man?
- How would you find the highest-ranking women he hasn’t proposed to yet?
- How would you decide if a women prefers $m$ or $m'$?
To get $O(n^2)$ running time, each execution of the loop must take constant time.

- How would you find a free man? linked list
- How would you find the highest-ranking women he hasn’t proposed to yet?
- How would you decide if a woman prefers $m$ or $m'$?
Implementation

To get $O(n^2)$ running time, each execution of the loop must take constant time.

- How would you find a free man?  
  *linked list*

- How would you find the highest-ranking women he hasn’t proposed to yet?  
  \[ \text{Next}[m] = \text{index of next woman to propose to}. \]

- How would you decide if a women prefers $m$ or $m'$?
Implementation

To get $O(n^2)$ running time, each execution of the loop must take constant time.

- How would you find a free man? linked list
- How would you find the highest-ranking women he hasn’t proposed to yet? $\text{Next}[m] =$ index of next woman to propose to.
- How would you decide if a women prefers $m$ or $m'$?

$$w : m_2, m_3, m_4, m_1$$

$$\text{Rank}[w] = [4, 1, 2, 3]$$

Can test $\text{Rank}[w, m] < \text{Rank}[w, m']$
Some notes

1. A woman remains *engaged* starting from when she receives her first proposal.

2. A woman’s fiancé always improves (or stays the same).

3. A man may alternate between being *free* and being *engaged*. 
Theorem

The set $S$ returned is a perfect matching.

Proof.

The set $S$ is always a matching. Suppose it is not perfect. Then there is a free man $m$.

He must have proposed to every women, or else the algorithm would not terminate.

Because every women has been proposed to, each must be engaged (by 1 above).

Therefore there are $n$ engaged women, so there must be $n$ engaged men. This contradicts that $m$ was free.
Theorem

A set $S$ returned by the G-S algorithm is a stable matching.

Proof.

Suppose not. Then there is some instability in $S$. This means there are pairs $(m, w)$ and $(m', w')$ in $S$ such that:

- $m$ prefers $w'$ to $w$
- $w'$ prefers $m$ to $m'$

Therefore, $m$ must have proposed to $w'$ before $w$.

Because he ended up with $w$, he must have been rejected by $w'$. Since the fiancé of women always improves, $w'$ must prefer who she ended up with ($m'$) to the rejected $m$.

This contradicts the assumption that $w'$ prefers $m$ to $m'$. □
Example

Which matching will the G-S algorithm find?

$m$ prefers $w$ to $w'$
$m'$ prefers $w'$ to $w$

$w$ prefers $m'$ to $m$
$w'$ prefers $m$ to $m'$

$(m, w), (m', w')$
OR $(m', w), (m, w')$
Example

Which matching will the G-S algorithm find?

<table>
<thead>
<tr>
<th></th>
<th>m prefers</th>
<th>w to w'</th>
</tr>
</thead>
<tbody>
<tr>
<td>m' prefers</td>
<td>w' to w</td>
<td></td>
</tr>
<tr>
<td>w prefers</td>
<td>m' to m</td>
<td></td>
</tr>
<tr>
<td>w' prefers</td>
<td>m to m'</td>
<td></td>
</tr>
</tbody>
</table>

\[(m, w), (m', w')\] OR \[(m', w), (m, w')\]

\[(m, w), (m', w')\]: this is the best matching according to the men, but the worst matching according to the women.

This is in general true: the G-S algorithm finds the matching that makes the men happiest.
In general, we usually have many choices for which free man to look at next in the while loop:

While there is a man who is free and hasn’t proposed to every women:

Choose such a man m

Surprisingly, the order we consider the men in doesn’t make any difference: we’ll always get the same matching.
Every execution returns the same matching

No matter the order we consider the men in, we get the same stable matching.

- Call \( w \) a valid partner for \( m \) if \((m, w)\) appears in some stable matching.
- Let \( \text{best}(m) \) be the highest ranking, valid partner for \( m \).
- Let \( S^* = \{(m, \text{best}(m)) : m \in M\} \).
  In other words, \( S^* \) pairs \( m \) with the highest ranking women that is paired with \( m \) in some stable matching.

Theorem

Every execution of G-S returns \( S^* \).
Valid and best partners

\[ m_1 \quad m_2 \quad m_3 \quad m_4 \]

Theorem

*Every execution of G-S matches every \( m \) with \( \text{best}(m) \) — his highest ranked, best partner.*

See page 11 of your book.
CMSC 451: Algorithm Analysis Basics

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Chapter 2 of *Algorithm Design* by Kleinberg & Tardos.
A problem like Independent Set, Stable Marriage, etc. can be thought of as families of instances:

- An instance of Independent Set is specified by a particular graph \( G \).
- An instance of Stable Matching is given by the preference lists.

These instances often will have a natural way of encoding them into the computer:

- A graph with \( N \) nodes might be specified by an \( N \times N \) matrix or a list of edges.
- The preference lists could be encoded as an array of lists, or a matrix.
Instance Sizes

Size of the instance is the space used to represent it. Usually use $n$ to represent this size.

Generally, “larger” instances are more difficult than smaller ones.

Can often break the problem down into:

- **Search space**: the set of feasible solutions (every independent set or perfect matching)
- **Objective function**: a way of measuring how good the solution is (size of independent set, stability of perfect matching)
Central Dogma of Computer Science:

Difficulty is not necessarily proportional to size of the search space.
Efficient Algorithms

Definition (Efficiency)

An algorithm is efficient if its worst-case runtime is bounded by a polynomial function of the input size.

- There is a polynomial $p(n)$ such that for every instance of size $n$, the algorithm solves the instance in fewer than $p(n)$ steps.

Generally, the problems we consider will have huge search spaces.

How many perfect matchings are there in a stable matching problem with $n$ men and $n$ women?

Any polynomial is much smaller than the search space.
Efficient Algorithms

Definition (Efficiency)
An algorithm is efficient if its worst-case runtime is bounded by a polynomial function of the input size.

- There is a polynomial \( p(n) \) such that for every instance of size \( n \), the algorithm solves the instance in fewer than \( p(n) \) steps.

Generally, the problems we consider will have huge search spaces.

How many perfect matchings are there in a stable matching problem with \( n \) men and \( n \) women? \( n! \)

Any polynomial is much smaller than the search space.
This definition of efficiently is not perfect:

1. There are non-polynomial algorithms that are usually the best choice in practice (e.g. simplex method for linear programming).

2. There are polynomial time algorithms that are almost never used in practice (e.g. ellipsoid algorithm for linear programming).

3. An algorithm that takes $n^{100}$ steps would be “efficient” by this definition, but horrible in practice.

4. An algorithm that takes $n^{1+0.002\log n}$ is probably useful in practice, but it is not “efficient”.
But it has a lot of benefits:

1. It’s concrete and falsifiable — avoids “vague” arguments about which algorithm is better.

2. Average performance is hard to define.

3. The exceptions are apparently fewer than the cases where polynomial time corresponds to useful algorithms in practice.

4. There is normally a huge difference between polynomial time and other natural runtimes
   (If you can do 1 million steps per second and $n = 1,000,000$, then a $n^2$ algorithm would take 12 days, but a $1.5^n$ algorithm would take far more than $10^{25}$ years)
A running time of $n^2 + 4n + 2$ is usually too detailed. Rather, we’re interested in how the runtime grows as the problem size grows.

**Definition (O)**

A runtime $T(n)$ is $O(f(n))$ if there exist constants $n_0 \geq 0$ and $c > 0$ such that:

$$T(n) \leq cf(n) \quad \text{for all } n \geq n_0$$

What does this mean?

- for all large enough instances
- the running time is bounded by a constant multiple of $f(n)$
Asymptotic Lower Bounds

$O(\cdot)$ talks about the *longest* possible time an algorithm could take. $\Omega(\cdot)$ talks about the *shortest* possible time.

**Definition ($\Omega$)**

$T(n)$ is $\Omega(f(n))$ if there are constants $\epsilon > 0$ and $n_0 \geq 0$ so that:

$$T(n) \geq \epsilon f(n) \quad \text{for all } n \geq n_0$$
Tight bounds

Definition (Θ)

\( T(n) \) is \( \Theta(f(n)) \) if \( T(n) \) is \( O(f(n)) \) and \( \Omega(f(n)) \).

If we know that \( T(n) \) is \( \Theta(f(n)) \) then \( f(n) \) is the “right” asymptotic running time: it will run faster than \( O(f(n)) \) on all instances and some instances might take that long.
Asymptotic Limit

**Theorem**

If \( \lim_{n \to \infty} \frac{T(n)}{g(n)} \) equals some \( c > 0 \), then \( T(n) = \Theta(f(n)) \).

There is an \( n_0 \) such that \( c/2 \leq T(n)/g(n) \leq 2c \) for all \( n \geq n_0 \).

Therefore, \( T(n) \leq 2cg(n) \) for \( n \geq n_0 \) \( \Rightarrow \) \( T(n) = O(g(n)) \).

Also, \( T(n) \geq \frac{c}{2}g(n) \) for \( n \geq n_0 \) \( \Rightarrow \) \( T(n) = \Omega(g(n)) \).
Linear time usually means you look at each element a constant number of times.

- **Finding the maximum element in a list:**

```plaintext
max = a[1]
for i = 2 to n:
    if a[i] > max then
        set max = a[i]
Endfor
```

This does a constant amount of work per element in array a.

- **Merging sorted lists.**
$O(n \log n)$ time common because of sorting (often the slowest step in an algorithm).

Where does the $O(n \log n)$ come from?
$O(n \log n)$ time common because of sorting (often the slowest step in an algorithm).

Where does the $O(n \log n)$ come from?

\[ T(n) = 2T(n/2) + n \]
Quadratic Time — $O(n^2)$

Given a set of points, what is the smallest distance between them: 

$\binom{n}{2}$ pairs
Given $n$ sets $S_1, S_2, \ldots, S_n$ that are subsets of $\{1, \ldots, n\}$, is there some pair of sets that is disjoint?
Larger polynomials arise from exploring smaller search spaces exhaustively:

**Independent Set of Size $k$**

Given a graph with $n$ nodes, find an independent set of size $k$ or report none exists.

For every subset $S$ of $k$ nodes

If $S$ is an independent set then

return $S$  \hspace{1cm} (*)

Endfor

return Failure

How many subsets of $k$ nodes are there?
What if we didn't limit ourselves to independent sets of size $k$, and instead want to find the largest independent set?

**Brute force** algorithms search through all possibilities.

How many subsets of nodes are there in an $n$-node graph?

What's the runtime of the brute force search search for a largest independent set?
What if we didn't limit ourselves to independent sets of size $k$, and instead want to find the largest independent set?

**Brute force** algorithms search through all possibilities.

How many subsets of nodes are there in an $n$-node graph? $2^n$

What's the runtime of the brute force search search for a largest independent set?
Exponential Time

What if we didn't limit ourselves to independent sets of size $k$, and instead want to find the largest independent set?

**Brute force** algorithms search through all possibilities.

How many subsets of nodes are there in an $n$-node graph? $2^n$

What's the runtime of the brute force search search for a largest independent set? $O(n^2 2^n)$
Sublinear time means we don’t even look at every input.

Since it takes $n$ time just to read the input, we have to work in a model where we count how many queries to the data we make.

Sublinear usually means we don’t have to look at every element.

Example?
**Sublinear time**

Sublinear time means we don’t even look at every input.

Since it takes $n$ time just to read the input, we have to work in a model where we count how many queries to the data we make.

Sublinear usually means we don’t have to look at every element.

Example? Binary search — $O(\log n)$
CMSC 451: Graph Properties, DFS, BFS, etc.

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Chapter 3 of *Algorithm Design* by Kleinberg & Tardos.
Graphs specify pairwise relationships between objects.

An undirected graph $G = (V, E)$ is a pair of sets:

- $V$ is a set of nodes, aka vertices.
- $E$ is a set of two-elements subset of $V$.  
  An element of $E$ is of the form: $e = \{u, v\}$ with $u, v \in V$.

A graph is directed if $E$ is a set of ordered pairs $(u, v)$, $u, v \in V$.  

![Diagram of undirected and directed graphs](image-url)
Graph Terminology

- **Simple**: at most one edge between every pair of vertices.
- **Complete**: all possible edges are present (denoted: $K_n$)
- **Degree of a vertex**: number of incident edges.
- **Self-loop**: an edge $\{u, u\}$ — often disallowed.
- **Path**: a sequence $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $i = 1 \ldots, k$.
- **Simple path**: a path with no vertex repeated.
- **Closed path**: a path with $v_1 = v_k$. This path is a **cycle** if it is simple.
- **A graph is connected** if there is a path between any pair of vertices.
- **Digraph**: short name for directed graph.
A graph \( G = (V_G, E_G) \) is a **subgraph** of \( H = (V_H, E_H) \) if \( V_G \subseteq V_H \) and \( E_G \subseteq E_H \).

**Connected Component**: a maximal connected subgraph.

**Cut-edge** (aka **bridge**): an edge whose removal increases the number of connected components.
Graph Modeling Examples

Graphs model *many* concepts naturally:

1. Social networks
2. Geographic adjacency
3. Polyhedra
4. Chemical Molecules
5. Assigning jobs to applicants
6. Food webs
7. Finite-state machines
8. Markov processes
9. Project dependencies
10. World Wide Web
11. Telephone network
12. Roads
Graph Modeling Examples

Graphs model many concepts naturally:

1. Social networks
2. Geographic adjacency
3. Polyhedra
4. Chemical Molecules
5. Assigning jobs to applicants ← last week
6. Food webs
7. Finite-state machines
8. Markov processes
9. Project dependencies ← today
10. World Wide Web
11. Telephone network
12. Roads
When are two graphs “the same”? Often, we care only about the structure of the graph, and not any vertex labels.

**Definition (Graph isomorphism)**

Graphs $G$ and $H$ are **isomorphic** if there is a bijection $f : V_G \rightarrow V_H$ such that

$$\{u, v\} \in E_G \iff \{f(u), f(v)\} \in E_H.$$  

What’s an algorithm to test whether two graphs are isomorphic?
When are two graphs “the same”? Often, we care only about the structure of the graph, and not any vertex labels.

**Definition (Graph isomorphism)**

Graphs $G$ and $H$ are **isomorphic** if there is a bijection $f : V_G \rightarrow V_H$ such that

$$\{u, v\} \in E_G \iff \{f(u), f(v)\} \in E_H.$$  

What’s an algorithm to test whether two graphs are isomorphic?

No one knows a really good algorithm. This problem is not known to be NP-complete (unlike Independent Set).
Open Problems (Already!)

1. Is there a polynomial time algorithm for Independent Set?

2. Is there a polynomial time algorithm for Graph Isomorphism?
Representing Graphs

Adjacency matrix:

\[
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 \\
\end{pmatrix}
\]

Adjacency list:

```
1 <-> 2 <-> 3
2 <-> 1 <-> 5
3 <-> 4 <-> 5
4 <-> 3
5 <-> 2 <-> 3
```
Trees

Trees are a special type of graph that occur often in algorithms.

Definition (Tree)

A graph $G$ is a tree if it is connected and contains no cycles.
Theorem (Characterization of Trees)

The following statements are equivalent:

1. $T$ is a tree.
2. $T$ contains no cycles and $n - 1$ edges.
3. $T$ is connected and has $n - 1$ edges.
4. $T$ is connected and every edge is a cut-edge.
5. Any two nodes in $T$ are connected by exactly 1 path.
6. $T$ is acyclic, and adding any new edge creates exactly one cycle.
• A tree is rooted if it has a distinguished vertex called the root.
• A tree is ordered if it is a rooted tree where the children are assigned an order.
• In a binary tree, each node has at most 2 children.
• In a \textit{m-ary} tree, each node has at most \textit{m} children.
• A complete tree is a \textit{m-ary} tree for which each node has \textit{m} children and all leaves are at the same level.
A **binary search tree** is a binary tree where

1. a key $k(u)$ is associated with each node $u$, and
2. the keys in the subtree rooted at $\text{left}(u)$ are all $\leq k(u)$
3. the keys in the subtree rooted at $\text{right}(u)$ are all $> k(u)$.

**Balanced** binary search trees try to avoid long, stringy trees so that when searching we can eliminate about half the remaining elements at each step.

E.g. AVL trees let us find any element in $O(\log n)$ time, and we can efficiently update them.
Graph Traversals
Breadth-First Search

Breadth-first search explores the nodes of a graph in increasing distance away from some starting vertex $s$.

It decomposes the component into layers $L_i$ such that the shortest path from $s$ to each of nodes in $L_i$ is of length $i$.

**Breadth-First Search:**

1. $L_0$ is the set $\{s\}$.
2. Given layers $L_0, L_1, \ldots, L_j$, then $L_{j+1}$ is the set of nodes that are not in a previous layer and that have an edge to some node in layer $L_j$. 
A BFS traversal of a graph results in a breadth-first search tree:
A BFS traversal of a graph results in a **breadth-first search tree**: 

Can we say anything about the non-tree edges?
Choose \( x \in L_i \) and \( y \in L_j \) such that \( \{x, y\} \) is an edge in undirected graph \( G \). Then \( i \) and \( j \) differ by at most 1.

In other words, edges of \( G \) that do not appear in the tree connect nodes either in the same layer or adjacent layer.

Proof.
Suppose not, and that \( i < j - 1 \).
All the neighbors of \( x \) will be found by layer \( i + 1 \).
Therefore, the layer of \( y \) is less than \( i + 1 \), so \( j \leq i + 1 \), which contradicts \( i < j - 1 \).
Depth-First Search

DFS keeps walking down a path until it is forced to backtrack. It backtracks until it finds a new path to go down.

Think: Solving a maze.

It results in a search tree, called the depth-first search tree. In general, the DFS tree will be very different than the BFS tree.
Depth-First Search

We say anything about the non-tree edges.
Can we say anything about the non-tree edges?
Theorem

Let \(x\) and \(y\) be nodes in the DFS tree \(T_G\) such that \(\{x, y\}\) is an edge in undirected graph \(G\). Then one of \(x\) or \(y\) is an ancestor of the other in \(T_G\).

Proof.

Suppose, wlog, \(x\) is reached first in the DFS.

All the nodes that are marked explored between first encountering \(x\) and leaving \(x\) for the last time are descendants of \(x\) in \(T_G\).

When we reach \(x\), node \(y\) must not yet have been explored.

It must become explored before leaving \(x\) for the last time (otherwise, we should add \(\{x, y\}\) to \(T_G\)). Hence, \(y\) is a descendent of \(x\) in \(T_G\).
Implementing BFS & DFS

- **BFS**: When you see a new, unexplored node, put it on a queue. Will process nodes in the order you first see them.

- **DFS**: When you see a new, unexplored node, put it on a stack. Will immediately process a new node.

  When you have to backtrack, unexplored node closest to the top of the stack will be a child of the last node you visited where you had a choice.
We can think of BFS and DFS (and several other algorithms) as special cases of tree growing:

- Let $T$ be the current tree $T$, and
- Maintain a list of frontier edges: the set of edges of $G$ that have one endpoint in $T$ and one endpoint not in $T$:

 repeately choose a frontier edge (somehow) and add it to $T$. 
TreeGrowing(graph G, vertex v, func nextEdge):
    T = (v, ∅)
    S = set of edges incident to v
    While S is not empty:
        e = nextEdge(G, S)
        T = T + e  // add edge e to T
        S = updateFrontier(G, S, e)
    return T

- The function nextEdge(G, S) returns a frontier edge from S.
- updateFrontier(G, S, e) returns the new frontier after we add edge e to T.
These algorithms are all special cases of Tree Growing, with different versions of `nextEdge`:

1. Depth-first search
2. Breadth-first search
3. Prim’s minimum spanning tree algorithm
4. Dijkstra’s shortest path
What’s nextEdge for DFS?

What’s nextEdge for BFS?
BFS & DFS as Tree Growing

What’s nextEdge for DFS?

Select a frontier edge whose tree endpoint was discovered most recently.

Why? We can use a stack to implement DFS.

What’s nextEdge for BFS?
BFS & DFS as Tree Growing

What’s nextEdge for DFS?

Select a frontier edge whose tree endpoint was discovered most recently.

Why? We can use a stack to implement DFS.

What’s nextEdge for BFS?

Select a frontier edge whose tree endpoint was discovered earliest.

Why? We can use a queue to implement BFS.
An Application of BFS
Testing Bipartiteness

Problem

Determine if a graph $G$ is bipartite.

Bipartite graphs can’t contain odd cycles:
How can we test if $G$ is bipartite?
Bipartite Testing

How can we test if $G$ is bipartite?

- Do a BFS starting from some node $s$.
- Color even levels “blue” and odd levels “red.”
- Check each edge to see if any edge has both endpoints the same color.
One of two cases happen:

1. There is no edge of $G$ between two nodes of the same layer. In this case, every edge just connects two nodes in adjacent layers. But adjacent layers are oppositely colored, so $G$ must be bipartite.

2. There is an edge of $G$ joining two nodes $x$ and $y$ of the same layer $L_j$. Let $z \in L_i$ be the least common ancestor of $x$ and $y$ in the BFS tree $T$. $z - x - y - z$ is a cycle of length $2(j - i) + 1$, which is odd, so $G$ is not bipartite.
An Application of DFS
- A **directed, acyclic graph (DAG)** is a graph that contains no directed cycles. (After leaving any node $u$ you can never get back to $u$ by following edges along the arrows.)

- DAGs are very useful in modeling project dependencies: Task $i$ has to be done before task $j$ and $k$ which have to be done before $m$. 

```
    e       f       g       h       i       j       k       m
```


Given a DAG $D$ representing dependencies, how do you order the jobs so that when a job is started, all its dependencies are done?

Topological Sort

Given a DAG $D = (V, E)$, find a mapping $f$ from $V$ to $\{1, \ldots, |V|\}$, so that for every edge $(u, v) \in E$, $f(u) < f(v)$. 
Theorem

Every DAG contains a vertex with no incoming edges.
Theorem

Every DAG contains a vertex with no incoming edges.

Proof.

Suppose not.

Then keep following edges backward and in fewer than $n + 1$ steps you’ll reach a node you’ve already visited.

This is a directed cycle, contradicting that the graph is a DAG.
Theorem

Every DAG contains a vertex with no incoming edges.

Proof.

Suppose not.

Then keep following edges backward and in fewer than \( n + 1 \) steps you’ll reach a node you’ve already visited.

This is a directed cycle, contradicting that the graph is a DAG.

How can we turn this into an algorithm?
Topological Sort Algorithm

Topological sort:

1. Let $i = 1$
2. Find a node $u$ with no incoming edges, and let $f(u) = i$
3. Delete $u$ from the graph
4. Increment $i$

Implementation: Maintain

- $\text{Income}[w] =$ number of incoming edges for node $w$
- a list $S$ of nodes that currently have no incoming edges.

When we delete a node $u$, we decrement $\text{Income}[w]$ for all neighbors $w$ of $u$. If $\text{Income}[w]$ becomes 0, we add $w$ to $S$. 
DFS can be used to associate 2 numbers with each node of a graph $G$:

- **discovery time**: $d[u] = \text{the time at which } u \text{ is first visited}$
- **finishing time**: $f[u] = \text{the time at which all } u \text{ and all its neighbors have been visited.}$

Clearly $d[u] \leq f[u]$. 
Let \((u, v)\) be an edge of a DAG \(D\). What can we say about the relationship between \(f[u]\) and \(f[v]\)?
Let \((u, v)\) be an edge of a DAG \(D\). What can we say about the relationship between \(f[u]\) and \(f[v]\)?

Back edges and Left-Right edges cannot occur

\[ \implies f[v] < f[u] \] if \((u, v) \in D\).
Topological Sort Via Finishing Times:

Every edge \((u, v)\) in a DAG has \(f[v] < f[u]\).

If we list nodes from largest \(f[u]\) to smallest \(f[u]\) then every edge goes from left to right.

Exactly a topological sort.

So: as each node is finished, add it to the front of a linked list.
CMSC 451: Interval Scheduling

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 4.1 of Algorithm Design by Kleinberg & Tardos.
Interval Scheduling

- You want to schedule jobs on a supercomputer.
- Requests take the form \((s_i, f_i)\) meaning a job that runs from time \(s_i\) to time \(f_i\).
- You get many such requests, and you want to process as many as possible, but the computer can only work on one job at a time.

Given a set \(J = \{(s_i, f_i) : i = 1, \ldots, n\}\) of job intervals, find the largest \(S \subset J\) such that no two intervals in \(S\) overlap.
Greedy Algorithm

**Greedy Algorithms**

- Not easy to define what we mean by “greedy algorithm”
- Generally means we take little steps, looking only at our local choices
- Often among the first reasonable algorithms we can think of
- Frequently doesn’t lead to optimal solutions, but sometimes it does.
Example Greedy Algorithms

- TreeGrowing is an example of a greedy framework for most choices of nextEdge functions.

- Topological sort & testing bipartiteness were greedy algorithms

- Interval Scheduling turns out to have a nice greedy algorithm that works.
Ideas for Interval Scheduling

A greedy framework:

\[ S = \text{set of input intervals } (s_i, f_i) \]
While \( S \) is not empty:
\[ q = \text{nextInterval}(S) \]
Output interval \( q \)
Remove intervals that overlap with \( q \) from \( S \)

What are possible rules for \text{nextInterval}?
What are possible rules for nextInterval?

1. **Choose the interval that starts earliest.**
   Rationale: start using the resource as soon as possible.

2. **Choose the smallest interval.**
   Rationale: try to fit in lots of small jobs.

3. **Choose the interval that overlaps with the fewest remaining intervals.**
   Rationale: keep our options open and eliminate as few intervals as possible.
Rules That Don’t Work

1. Earliest start time

2. Shortest job

3. Fewest conflicts
Rules That Don’t Work

1. Earliest start time

2. Shortest job

3. Fewest conflicts
Rules That Don’t Work

1 Earliest start time

2 Shortest job

3 Fewest conflicts
Rules That Don’t Work

1. Earliest start time

2. Shortest job

3. Fewest conflicts
Optimal Greedy Algorithm

- Choose the interval with the *earliest finishing* time. 
  Rationale: ensure we have as much of the resource left as possible.

\[
S = \text{set of input intervals } \{(s[i], f[i])\}
\]

While \( S \) is not empty:

- \( q \) = a request in \( S \) that has the *soonest finishing* time

Output interval \( q \)

Remove intervals that overlap with \( q \) from \( S \)

- This algorithm chooses a compatible set of intervals.
How can we prove the schedule returned is optimal?

- Let $A$ be the schedule returned by this algorithm.
- Let $OPT$ be some optimal solution.

Might be hard to show that $A = OPT$, instead we need only to show that $|A| = |OPT|$.

Note the distinction: instead of proving directly that a choice of intervals $A$ is the same as an optimal choice, we prove that it has the same number of intervals as an optimal. Therefore, it is optimal.
Notation

Let these be the schedules of $A$ and $OPT$:

$A$: $i_1, i_2, \ldots, i_k$

$OPT$: $j_1, j_2, \ldots, j_m$

Let $f(i)$ be the finishing time of job $i$.

Theorem

For all $r \leq k$ we have $f(i_r) \leq f(j_r)$.

Proof.

By induction. True when $r = 1$ because we chose greedily.

Assume $f(i_{r-1}) \leq f(j_{r-1})$. Then we have

$$f(i_{r-1}) \leq f(j_{r-1}) \leq s(j_r),$$

where $s(j_r)$ is the start time of job $j_r$. So, job $j_r$ is available when the greedy algorithm makes its choice. Hence, $f(i_r) \leq f(j_r)$. 

\QED
Theorem

This greedy algorithm is optimal for Interval Scheduling.

Proof.

Suppose not. Then if \( A \) has \( k \) jobs, \( OPT \) has \( m > k \) jobs. By our lemma, after \( k \) jobs we have this situation:

So, job \( j_{k+1} \) must have been available to the greedy algorithm.
Implementation

1. Sort the intervals based on $f_i$ — takes $O(n \log n)$.

2. Scan down this list, output the first element that starts after the finishing time of the last item output — takes $O(n)$.

\[\begin{array}{cccccccc}
  f_1 & f_2 & f_3 & f_4 & f_5 & f_5 & f_6 & f_7 \\
  \hline
  s_1 & s_2 & s_3 & s_4 & s_5 & s_5 & s_6 & s_7 \\
\end{array}\]

LatestFinishingTime = finishing time of last scheduled interval
Extensions

• Online algorithms: What if you don’t know all the intervals at the start? Current active area of research.

• What if some intervals are more important than others? Weighted interval scheduling. — We’ll see this later.
Other rules of scheduling intervals also lead to nice greedy algorithms. For example:

Interval Partitioning Problem
Interval Partitioning Problem

Given intervals \((s_i, f_i)\) assign them to processors so that you schedule every interval and use the smallest \# of processors.

Now, we’re trying to minimize the \# of processors (or rooms) used.

**Another way to think of it:** Each processor corresponds to a color. We’re trying to color intervals with the fewest \# of colors so that no two overlapping intervals get the same color.
Interval Partitioning Problem

Definition
Depth is the maximum number of intervals passing over any time point.

Theorem
The number of processors needed is at least the depth of the set of intervals.

depth = 3
We need at least depth processors.

Can we find a schedule with no more than depth processors?
Greedy Alg for Interval Partitioning

Yes: Let \( \{1, \ldots, d\} \) be a set of labels, where \( d \) = depth.

---

Sort intervals by start time

For \( j = 1, 2, 3, \ldots, n \)

Let \( Q \) = set of labels that haven’t been assigned to a preceding interval that overlaps \( I_j \)

If \( Q \) is not empty,

Pick any label from \( Q \) and assign it to \( I_j \)

Else

Leave \( I_j \) unlabeled

Endfor
Every interval gets a label

No overlapping intervals get the same label because we exclude the labels that have already been used.

Every interval gets a label: The only way it wouldn’t is if we’ve run out of labels. That would mean, when coloring interval $I$ that $\geq d$ intervals with start times before $I$ overlap $I$:

So, when coloring $i$, there must be a color free.
Graph Coloring

Given a graph $G$ and a number $k$, color the nodes with $k$ colors such that no edge connects 2 vertices of the same color (or report that it can’t be done).
Graph Coloring

Given a graph $G$ and a number $k$, color the nodes with $k$ colors such that no edge connects 2 vertices of the same color (or report that it can’t be done).
When $k \geq 3$, Graph Coloring is hard in general.

We saw an algorithm for Graph Coloring when $k = 2$.

**Interval Graph Coloring**

Given a graph $G$ derived from a set of intervals and a number $k$, color the nodes with $k$ colors such that no edge connects 2 vertices of the same color (or report that it can’t be done).

Now we’ve seen an algorithm for solving Graph Coloring on the restricted class of graphs called “Interval Graphs.”
CMSC 451: Minimizing Lateness

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 4.3 of Algorithm Design by Kleinberg & Tardos.
A new kind of scheduling problem. Not given intervals with start and end times.

Instead, requests are of the form \((t_i, d_i)\), where:

- \(t_i\) is the job length, and
- \(d_i\) is the job deadline

We want to schedule these jobs on a single processor.

**Definition**

The **lateness** \(L_i\) of job \(i\) is \(\max\{0, f_i - d_i\}\), i.e. the length of time past its deadline that it finishes.

**Our goal:** minimize the maximum lateness.
Example
Possible Greedy Rules

- Short jobs first.
Possible Greedy Rules

- Short jobs first.

\[ t_1 = 10 \]
\[ t_2 = 100 \]
\[ d_2 = 100 \]
\[ d_1 = 110 \]
Possible Greedy Rules

- Short jobs first.

\[ t_1 = 10 \quad t_2 = 100 \]

- Smallest slack time: \( d_i - t_i \).
Possible Greedy Rules

- Short jobs first.

- Smallest slack time: \( d_i - t_i \).
**The Right Greedy Rule**

- **Earliest deadline first**: get the job with the most pressing deadline done first.

Surprisingly, don't need to consider the length of the job!

**The algorithm:**

Let \( d[1] \leq \ldots \leq d[n] \) be the jobs sorted by increasing deadline

Let \( f = s \)

For \( i = 1, \ldots, n \):

- Schedule job \( i \) starting from time \( f \) to \( f + t[i] \)
- Let \( f = f + t[i] \)
Idle time = gaps in the schedule.

**Lemma**

*There is an optimal schedule with no idle time.*

Note: *some* optimal solutions may have idle time.

Given an OPT schedule with gaps, closing the gaps can only decrease the maximum lateness.
The Exchange Argument

- There may be lots of optimal schedules.
- Let A be the schedule produced by our algorithm.
- We’ll start with some optimal solution OPT.
- We make local changes to OPT, trying to transform it into A.
- Each local change will preserve optimality.
An inversion is a pair of jobs $i, j$ such that $i$ is scheduled before $j$, but $d_j < d_i$. 
Inversions in Schedules

Lemma

All schedules with no inversions and no idle time have the same maximum lateness.

If there are no inversions, this means the jobs are sorted by increasing deadlines. Let $S_1$ and $S_2$ be two such schedules.

The only way these could differ is when several jobs $i_1, \ldots, i_k$ have the same deadline, which must be adjacent in the schedule.

The last of these jobs is the only one that matters for maximum lateness.

And it doesn’t matter what the order of these jobs are.
Proof Outline

Property P

A schedule has Property P if it has no idle time & no inversions.

1. Our solution has property P

2. All solutions with property P have the same lateness

3. An optimal solution has property P
Proof Outline

Property P
A schedule has Property P if it has no idle time & no inversions.

1. Our solution has property P
   True by construction.

2. All solutions with property P have the same lateness
   True by last lemma

3. An optimal solution has property P
Proof of (3), outline

Theorem

There is an optimal solution with no idle time & no inversions.

• **Idea:** Start with any optimal solution $OPT_0$.

• Flip two adjacent jobs that are an inversion.

• This will reduce the number of inversions without increasing the maximum lateness.

• Repeat until we have no inversions.
Proof of (3)

(a) If $\text{OPT}_0$ has an inversion, it must have an inversion $i, j$ where $i$ and $j$ are adjacent:

If $a, b$ is an inversion, then $d_a > d_b$, so there must be some step down.

(b) If we swap $i$ and $j$ we reduce the number of inversions by 1.
(c) Let $OPT_1$ be the schedule with $i$ and $j$ swapped. $OPT_1$ has maximum lateness $\leq OPT_0$. 

\[ \text{New max}\{L_i, L_j\} \]

\[ \text{Old max}\{L_i, L_j\} \]
So: If we keep swapping adjacent inversions in an optimum solution, we will eventually arrive a solution with no inversions without changing the maximum lateness.

Our greedy algorithm produced a solution without inversions.

Since all solutions without inversions have the same maximum lateness, our greedy algorithm (sort by deadline) must have the same maximum lateness as the optimum.
CMSC 451: Optimal Caching

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 4.4 of *Algorithm Design* by Kleinberg & Tardos.
Memory Hierarchies

- Registors
- Cache
- RAM
- disk

Faster

Larger, Slower
• We want to store $n > k$ items in a cache $C$ of size $k$
• We’re given a request order of items: 1, 5, 8, 9, ...
• There’s a hit if $i$ is in $C$ when it is requested, miss otherwise
• At each request, we can swap a cache item with a non-cache item
• **Our Goal:** Minimize the number of misses
Example

Suppose $k = 3$, with $n = 4$ items $\{a, b, c, d\}$.

Request order: a, b, c, a, d, d, c, d, a, b

Request:       Cache:
  (start)      abc
    a         abc
    d         abd         miss!
    d         abd
    c         acd         miss!
    d         acd
    a         acd
    b         abd         miss!
Belady’s Algorithm:

- When $d_i$ is requested but not in the cache, evict the cache item that will next be used farthest into the future.

How can we prove this results in the fewest # of misses possible?
• Normally, in practice, we don’t know the full sequence of requests.

• So we can’t calculate which item will be used farthest in the future.

• Heuristics such as least recently used are employed instead.

• Still important to know the OPT, however: we can compare to OPT to figure out how well our heuristic did.
Same basic approach as before: we transform some optimal schedule into our schedule without increasing the number of misses.

- Let $D$ be a sequence of requests.
- Let $A$ be the schedule for $D$ obtained by the “farthest in the future” algorithm.
- Let $S_j$ be a schedule for $D$ that makes the same decisions as $A$ up through the first $j$ steps.

- We’ll construct a schedule $S_{j+1}$ that agrees with $A$ up through the first $j+1$ steps and has the same number of misses as $S_j$. 
The Exchange Argument

- Start with a schedule $S_0$.

- Let $A$ be the schedule produced by our algorithm.

- We’ll start with some optimal solution $S_0$.

- We make local changes to $S_0$, trying to transform it into $A$.

- Each local change will keep the same $\#$ of misses.
If we let the first schedule \( S_0 = OPT \),
then this shows that \( A \) has the same number of misses as \( OPT \).
We’ll construct a schedule $S_{j+1}$ that agrees with $A$ up through the first $j + 1$ steps and has the same number of misses as $S_j$. 

Given:

$A \rightarrow e$  

$S_j \rightarrow f$  

$g \rightarrow e$  

$f \rightarrow h$  

Want to construct:

$S_{j+1} \rightarrow e$  

$g \rightarrow f$  

$e \rightarrow h$
After step $j$, the cache contents of $S_j$ and $S_{j+1}$ differ slightly:

$$S_j : - - - - - e$$
$$S_{j+1} : - - - - - f$$

When one of the following happens, we can make their cache contents agree:

1. **$S_j$ evicts $e$ because some $g \not\in \{f, e\}$ was requested**: We evict $f$ and now $S_j$ and $S_{j+1}$ have the same cache content.

2. **$S_j$ evicts some element $h$ because $f$ was requested**: $f$ is already in $S_{j+1}$'s cache, so we use the chance to bring $e$ in.
S_{j+1} will have the same \# of misses as S_j unless item e is accessed before we have a chance to make the caches the same.

But this can’t happen, because e was the item that is accessed farthest in the future (by our greedy rule).

Hence, before we access e next, we’ll have a chance to fix the cache up.
Greedy Scheduling Recap:

- **Interval Scheduling:** *Increasing Finishing Time.*
  Greedy stays ahead of OPT, can always do what OPT does.

- **Interval Partitioning:** *Increasing Start Time.*
  Limit on how much of the resource (colors) have been used.

- **Minimizing Lateness:** *Increasing Deadlines.*
  Violations of this rule can only increase the cost.
  (oversimplified)

- **Optimal Caching:** *Farthest in the Future.*
CMSC 451: Shortest Paths in a Graph

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Chapter 4.5 of Algorithm Design by Kleinberg & Tardos.
Given a directed graph $G$ with lengths $\ell_e$ on each edge $e$:

**Goal:** Find the shortest path from a given node $s$ to every other node in the graph.
Dijkstra’s algorithm (1959) solves this problem.

If we have an undirected graph, we can replace each undirected edge by 2 directed edges:

• If all the edge lengths are \( = 1 \), how can we solve this?
Shortest Paths

Given directed graph $G$ with $n$ nodes, and non-negative lengths on each edge, find the $n$ shortest paths from a given node $s$ to each $v_i$.

- Dijkstra’s algorithm (1959) solves this problem.
- If we have an undirected graph, we can replace each undirected edge by 2 directed edges:

![Diagram of edge replacement](image)

- If all the edge lengths are $= 1$, how can we solve this? BFS
There is some optimal set of shortest paths such that their union forms a tree.
Dijkstra’s algorithm is just a special case of tree growing:

- Let $T$ be the current tree $T$, and
- Maintain a list of frontier edges: the set of edges of $G$ that have one endpoint in $T$ and one endpoint not in $T$:

- Repeatedly choose a frontier edge (somehow) and add it to $T$. 
Tree Growing

TreeGrowing(graph G, vertex v, func nextEdge):
    T = (v, ∅)
    S = set of edges incident to v
    While S is not empty:
        e = nextEdge(G, S)
        T = T + e // add edge e to T
        S = updateFrontier(G, S, e)
    return T

- The function nextEdge(G, S) returns a frontier edge from S.
- updateFrontier(G, S, e) returns the new frontier after we add edge e to T.
nextEdge for Shortest Path

• Let \( u \) be some node that we’ve already visited (it will be in \( S \)).

• Let \( d(u) \) be the length of the path found for node \( u \in S \).

• \text{nextEdge}: \text{return the frontier edge } (u, v) \text{ for which } d(u) + \text{length}(u, v) \text{ is minimized.}
Example

\[d[s] = 0; \, d[u] = 1\]

(green gives frontier)

\[d[w] = 2\]
Theorem

Let \( T \) be the set of nodes explored at some point during the algorithm. For each \( u \in T \), the path to \( u \) found by Dijkstra’s algorithm is the shortest.

Proof.

By induction on the size of \( T \). **Base case:** When \( |T| = 1 \), the only node in \( T \) is \( s \), for which we’ve obviously found the shortest path.

**Induction Hypothesis:** Assume theorem is true when \( |T| \leq k \).

Let \( v \) be the \( k + 1 \) node added using edge \((u, v)\).

Let \( P_v \) be the path chosen by Dijkstra’s to \( v \) and let \( P \) be any other path from \( s \) to \( v \).

Then we have the situation on the next slide.
The path to $v$ chosen by Dijkstra’s is of length $\leq$ the alternative blue path.
CMSC 451: Minimum Spanning Trees & Clustering

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Sections 4.5–4.6 of *Algorithm Design* by Kleinberg & Tardos.
Network Design

You want to connect up several computers with a network, and you want to run as little wire as possible.

It is feasible to directly connect only some pairs of computers.
Minimum Spanning Tree Problem

Given

- undirected graph $G$ with vertices for each of $n$ objects
- weights $d(u, v)$ on the edges giving the distance $u$ and $v$,

Find the subgraph $T$ that connects all vertices and minimizes

$$\sum_{\{u,v\} \in T} d(u, v).$$

$T$ will be a tree. Why?
Minimum Spanning Tree Problem

Given

- undirected graph $G$ with vertices for each of $n$ objects
- weights $d(u, v)$ on the edges giving the distance $u$ and $v$,

Find the subgraph $T$ that connects all vertices and minimizes $\sum_{\{u, v\} \in T} d(u, v)$.

$T$ will be a tree. Why?

If there was a cycle, we could remove any edge on the cycle to get a new subgraph $T'$ with smaller $\sum_{\{u, v\} \in T'} d(u, v)$. 
MST History

- Studied as far back as 1926 by Borůvka.

- We’ll see algorithms that take $O(m \log n)$ time, where $m$ is number of edges.

- Best known algorithm takes time $O(m \alpha(m, n))$, where $\alpha(m, n)$ is the “inverse Ackerman” function (grows very slowly).

- Still open: Can you find a $O(m)$ algorithm?
We assume no two edges have the same edge cost.

If this doesn’t hold true, we can add a very small value $\epsilon_e$ to the weight of every edge $e$. 
Theorem

Let $S$ be a subset of nodes, with $|S| \geq 1$ and $|S| \leq n$. Every MST contains the edge $e = (v, w)$ with $v \in S$ and $w \in V - S$ that has minimum weight.
Suppose $T$ doesn’t contain $e$. Because $T$ is connected, it must contain a path $P$ between $v$ and $w$. $P$ must contain some edge $f$ that “crosses the cut.”

The subgraph $T' = T - f \cup e$ has lower weight than $T$. $T'$ is acyclic because the only cycle in $T' \cup f$ is eliminated by removing $f$. 

![Diagram](image-url)
Theorem (Cycle Property)

Let $C$ be a cycle in $G$. Let $e = (u, v)$ be the edge with maximum weight on $C$. Then $e$ is not in any MST of $G$.

Suppose the theorem is false. Let $T$ be a MST that contains $e$.

Deleting $e$ from $T$ partitions vertices into 2 sets:

$S$ (that contains $u$) and $V - S$ (that contains $v$).

Cycle $C$ must have some other edge $f$ that goes from $S$ and $V - S$.

Replacing $e$ by $f$ produces a lower cost tree, contradicting that $T$ is an MST.
Cycle Property, Picture
MST Property Summary

1. **Cut Property:** The smallest edge crossing any cut must be in all MSTs.

2. **Cycle Property:** The largest edge on any cycle is never in any MST.
Greedy MST Rules

All of these greedy rules work:

1. Add edges in increasing weight, skipping those whose addition would create a cycle. *(Kruskal’s Algorithm)*

2. Run TreeGrowing starting with any root node, adding the frontier edge with the smallest weight. *(Prim’s Algorithm)*

3. Start with all edges, remove them in decreasing order of weight, skipping those whose removal would disconnect the graph. *(“Reverse-Delete” Algorithm)*
Kruskal’s Algorithm: Add edges in increasing weight, skipping those whose addition would create a cycle.

**Theorem**

*Kruskal’s algorithm produces a minimum spanning tree.*

**Proof.** Consider the point when edge \( e = (u, v) \) is added:

\[ S = \text{nodes to which v has a path just before e is added} \]

\[ u \text{ is in } V - S \text{ (otherwise there would be a cycle)} \]
**Prim’s Algorithm**: Run TreeGrowing starting with any root node, adding the frontier edge with the smallest weight.

**Theorem**

*Prim’s algorithm produces a minimum spanning tree.*

S = set of nodes already in the tree when e is added
**Reverse-Delete Algorithm**: Remove edges in decreasing order of weight, skipping those whose removal would disconnect the graph.

**Theorem**

*Reverse-Delete algorithm produces a minimum spanning tree.*

Because removing edge $e$ won't disconnect the graph, there must be another path between $u$ and $v$.

Because we're removing in order of decreasing weight, $e$ must be the largest edge on that cycle.
Implementation: Prim’s & Dijkstra’s

- Store the nodes on the frontier in a priority queue, using key:

  **Prim’s:** \[ p(v) = \min_{(u,v): u \in S} d(u, v) \]

  **Dijkstra’s:** \[ s(v) = \min_{(u,v): u \in S} \text{dist}(s, u) + d(u, v) \]

- ExtractMin takes \( O(1) \) time, and we do \( O(n) \) of them.
- ChangeMin takes \( O(\log n) \) time, and we do \( O(m) \) of them.

**Total run time:** \( O(m \log n) \).

Can implement Kruskal’s algorithm in \( O(m \log n) \) time too, with more complicated data structures.
Clustering: an application of MST
You’re given \( n \) items and the distance \( d(u, v) \) between each of pair.

\( d(u, v) \) may be an actual distance, or some abstract representation of how dissimilar two things are.

(What’s the “distance” between two species?)

**Our Goal:** Divide the \( n \) items up into \( k \) groups so that the minimum distance between items in different groups is maximized.
**Our Goal:** Divide the $n$ items up into $k$ groups so that the minimum distance between items in different groups is maximized.

Quality of this partitioning is $\min\{a,b,c\}$.
Idea:

- Maintain clusters as a set of connected components of a graph.
- Iteratively combine the clusters containing the two closest items by adding an edge between them.
- Stop when there are $k$ clusters.
Maximum Minimum Distance

Idea:

- Maintain clusters as a set of connected components of a graph.
- Iteratively combine the clusters containing the two closest items by adding an edge between them.
- Stop when there are $k$ clusters.

This is exactly Kruskal’s algorithm.

The “clusters” are the connected components that Kruskal’s algorithm has created after a certain point.

Example of “single-linkage, agglomerative clustering.”
Proof of Correctness

Another way too look at the algorithm: delete the $k - 1$ most expensive edges from the MST.

The spacing $d$ of the clustering $C$ that this produces is the length of the $(k - 1)^{st}$ most expensive edge.

Let $C'$ be a different clustering. We’ll show that $C'$ must have the same or smaller separation than $C$. 
Proof of correctness, 2

Since $C \neq C'$, there must be some pair $p_i, p_j$ that are in the same cluster in $C$ but different clusters in $C'$.

Together in $C$ $\implies$ path $P$ between $p_i, p_j$ with all edges $\leq d$.

Some edge of $P$ passes between two different clusters of $C'$.

Therefore, separation of $C' \leq d$. 
Class So Far

6 lectures:

- Stable Marriage
- Topological Sort
- Detecting bipartite graphs
- Interval Scheduling
- Interval Partitioning
- Minimal Lateness Scheduling
- Optimal Caching
- Minimum Spanning Tree (3 Algs)
- Dijkstra’s algorithm (proof of correctness)
- Matriods
- Min cost aboresences
CMSC 451: Matroids, When Greed Works

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park
There’s a general argument that can be used to show why many greedy algorithms work.

We’ll describe a very general framework.

If you can write your problem in that framework,

and prove one simple property about your specific problem, then the greedy algorithm works.
A hereditary subset system is a pair $S = (E, I)$, where $E$ is a finite set and $I$ is a collection of subsets of $E$ closed under inclusion.

“Closed under inclusion” means that if $A \in I$ then all of the subsets of $A$ are also in $I$.

**Example 1:** If $G = (V, E)$ is a graph and $I$ is the collection of acyclic subgraphs of $G$, then $(E, I)$ is a hereditary subset system.

**Example 2:** Let $E$ be a set of intervals and $I$ be all subsets of compatible intervals. Then $(E, I)$ is a hereditary subset system.
Maximum Weight Problem

Given a hereditary subset system \((E, I)\) and a weight \(w(e)\) for every \(e \in E\), find the subset \(A \in I\) so that \(\sum_{e \in A} w(e)\) is maximized.

**Example 1:** *Maximum Spanning Tree*: Let \(E\) be edges of a graph \(G\), let \(I\) be the collection of acyclic subgraphs of \(G\), and \(w(e) = \) a weight on each edge.

**Example 2:** *Interval Scheduling*: Let \(E\) be intervals, \(I\) be all subsets of compatible intervals, and \(w(e) = 1\).
Greedy Algorithm

Greedy(E, I, w):
    S = ∅
    A = E
    While |A| > 0:
        Let e ∈ A be the element of largest weight w(e)
        A = A - e
        If S ∪ e ∈ I then S = S ∪ e
    Return S

Similar idea as TreeGrowing:
    • S is the current set,
    • A is the remaining elements to consider, and
    • I determines which sets are allowed.
**Augmentation Property:** If $A, B \in I$ and $|A| < |B|$, then there is an element $e \in B - A$ such that $A \cup \{e\} \in I$.

Given any two sets in $I$, one smaller than the other, there is some element in the larger one that we can add to the smaller one to form a set that is also in $I$.

**Definition (Matroid)**

A hereditary subset system $(E, I)$ is a matroid if it satisfies the augmentation property.
Why Matroids Are Interesting

**Theorem**

*Greedy will solve every instance of a maximum weight problem associated with \((E, I)\) if and only if \((E, I)\) is a matroid.*
There is a minimization version of this as well:

Minimum Weight Problem

Given a hereditary subset system \((E, I)\) and a weight \(w(e)\) for \(e \in E\), find the maximal subset \(S \subseteq E\) that minimizes \(\sum_{e \in S} w(e)\).

- We change Greedy to choose the element of \(A\) that has the smallest weight.
- This version of Greedy solves the Minimum Weight Problem if and only if it is applied to a matroid.
CMSC 451: Divide and Conquer

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Sections 5.1–5.3 of Algorithm Design by Kleinberg & Tardos.
Greedy Recap

Greedy algorithms are usually very natural.

Many problems have nice greedy solutions:

1. Topological Sorting (ch. 3)
2. Interval Scheduling (4.1)
3. Interval Partitioning (4.1)
4. Minimizing Lateness (4.2)
5. Optimal Scheduling (4.3)
6. Shortest Paths (Dijkstra’s) (4.4)
7. Minimum Spanning Tree (4.5)
8. Maximum Separation Clustering (4.7)
9. Matroids: Max-Weight
We’ve seen some general patterns for algorithms:

1. Sort and then scan (matroid greedy algorithm)
2. TreeGrowing

And for proof techniques:

1. **Greedy “stays ahead”:** any choice the OPT can make, the greedy can make and will
2. **Exchange:** we can transform an OPT solution into greedy with series of small changes.
3. **Matroid:** Hereditary Subset System with Augmentation Property
Divide and Conquer is a different framework.

Related to *induction*:

- Suppose you have a “box” that can solve problems of size $\leq k < n$
- You use this box on some subset of the input items to get partial answers
- You combine these partial answers to get the full answer.

But: you construct the “box” by recursively applying the same idea until the problem is small enough to be solved by brute force.
Merge Sort

MergeSort(L):
\[
\text{if } \mid L \mid = 2:
\]
\[
\quad \text{return } [\text{min}(L), \text{max}(L)]
\]

\[
\text{else:}
\]
\[
\quad L_1 = \text{MergeSort}(L[0, \mid L \mid /2])
\]
\[
\quad L_2 = \text{MergeSort}(L[\mid L \mid /2+1, \mid L \mid -1])
\]
\[
\text{return } \text{Combine}(L_1, L_2)
\]

• In practice, you sort in-place rather than making new lists.
• Combine(L1,L2) walks down the sorted lists putting the smaller number onto a new list. Takes $O(n)$ time
• Total time: $T(n) \leq 2T(n/2) + cn$. 
To Solve a Recurrence

Given a recurrence such as $T(n) \leq 2T(n/2) + cn$, we want a simple upper bound on the total running time.

Two common ways to “solve” such a recurrence:

1. Unroll the recurrence and see what the pattern is. Typically, you’ll draw the recursion tree.

2. Guess an answer and prove that it’s right.
Solving Recurrences

Draw the first few levels of the tree.

Write the amount of work done at each level in terms of the level.

Figure out the height of the tree.

Sum over all levels of the tree.

\[
T(n) \leq 2T(n/2) + cn
\]

Each level is \(cn\). There are \(\log n\) levels, so \(T(n)\) is \(O(n \log n)\).
Substitution method is based on induction. We:

1. Show $T(k) \leq f(k)$ for some small $k$.
2. Assume $T(k) \leq f(k)$ for all $k < n$.
3. Show $T(n) \leq f(n)$.

\[ T(n) \leq 2T(n/2) + cn \]

**Base Case:** $2c \log 2 = 2c \geq T(2)$

**Induction Step:**

\[
T(n) \leq 2T(n/2) + cn \\
\leq 2c(n/2) \log(n/2) + cn \\
= cn[(\log n) - 1] + cn \\
= cn \log n
\]
Counting Inversions
Comparing Rankings

Suppose two customers rank a list of movies.

similar

more different
A measure of distance

What’s a good measure of how dissimilar two rankings are?
A measure of distance

What’s a good measure of how dissimilar two rankings are?

We can count the number of inversions:

- Assume one of the rankings is 1, 2, 3, \ldots, n.
- Denote the other ranking by \(a_1, a_2, \ldots, a_n\).
- An inversion is a pair \((i, j)\) such that \(i < j\) but \(a_j < a_i\).

Two identical rankings have no inversions.

How many inversions do opposite rankings have?
What’s a good measure of how dissimilar two rankings are?

We can count the number of **inversions**:

- Assume one of the rankings is 1, 2, 3, \ldots, n.
- Denote the other ranking by \( a_1, a_2, \ldots, a_n \).
- An inversion is a pair \((i, j)\) such that \( i < j \) but \( a_j < a_i \).

Two identical rankings have no inversions.

How many inversions do opposite rankings have? \( \binom{n}{2} \)
How can we count inversions quickly?

• **Brute Force:** check every pair: $O(n^2)$.

• Some sequences might have $O(n^2)$ inversions, so you might think that it might take as much as $O(n^2)$ time to count them.

• In fact, with divide and conquer, you can count them in $O(n \log n)$ time.
Count the number of inversions in the sequence $a_1, \ldots, a_n$.

Suppose I told you the number of inversions in the first half of the list and in the second half of the list:

What kinds of inversions are not accounted for in $\text{Inv1} + \text{Inv2}$?
Half-Crossing Inversions

The inversions we have to count during the merge step:

\[
\begin{array}{|c|c|}
\hline
a_1, \ldots, a_{n/2} & a_{n/2+1}, \ldots, a_n \\
\hline
\end{array}
\]

\[a_i > a_j\]

The crux is that we have to count these kinds of inversion in \(O(n)\) time.
What if each of the half lists were sorted?

Suppose each of the half lists were sorted.

If we find a pair \( a_i > a_j \), then we can infer many other inversions:

Suppose \( a_i > b_j \); then all these are bigger than \( b_j \).

Each of the green items is an inversion with \( b_j \).
MergeAndCount(SortedList A, SortedList B):
    a = b = CrossInvCount = 0
    OutList = empty list
    While a < |A| and b < |B|:  // not at end of a list
        next = min(A[a], B[b])
        OutList.append(next)
        If B[b] == next:
            b = b + 1
            CrossInvCount += |A| - a  // inc by # left in A
        Else
            a = a + 1
    EndWhile
    Append the non-empty list to OutList
    Return CrossInvCount and OutList
Note that MergeAndCount will produce a sorted list as well as the number of cross inversions.
SortAndCount(List L):
    If |L| == 1: Return 0

    A, B = first & second halves of L

    invA, SortedA = SortAndCount(A)
    invB, SortedB = SortAndCount(B)

    crossInv, SortedL = MergeAndSort(SortedA, SortedB)
    Return invA + invB + crossInv and SortedL
Algorithm Schematic

Divide it into 2 parts

- $a_1, \ldots, a_{n/2}$
- $a_{n/2+1}, \ldots, a_n$

Compute the answer (and maybe some additional info) on each part separately

- Recursive Box
  - Inv1
    - sorted $a_1, \ldots, a_{n/2}$
  - Inv2
    - sorted $a_{n/2+1}, \ldots, a_n$

Merge

$\text{Inv1} + \text{Inv2} + \text{inversions that cross between the first half and the second half}$

$\text{sorted } a_1, \ldots, a_n$
What's the running time of SortAndCount?
What’s the running time of SortAndCount?

Break the problem into two halves.

Merge takes $O(n)$ time.

\[
T(n) \leq 2T(n/2) + cn
\]

⇒ Total running time is $O(n \log n)$. 
CMSC 451: Closest Pair of Points

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 5.4 of *Algorithm Design* by Kleinberg & Tardos.
Finding closest pair of points

Problem
Given a set of points \( \{p_1, \ldots, p_n\} \) find the pair of points \( \{p_i, p_j\} \) that are closest together.
• Brute force gives an $O(n^2)$ algorithm: just check every pair of points.

• Can we do it faster? Seems like no: don’t we have to check every pair?

• In fact, we can find the closest pair in $O(n \log n)$ time.

• What’s a reasonable first step?
Split the points with line $L$ so that half the points are on each side.

Recursively find the pair of points closest in each half.
Merge: the hard case

Let $d = \min\{d_{\text{left}}, d_{\text{right}}\}$.

- $d$ would be the answer, except maybe $L$ split a close pair!
If there is a pair \( \{p_i, p_j\} \) with dist\((p_i, p_j) < d\) that is split by the line, then both \( p_i \) and \( p_j \) must be within distance \( d \) of \( L \).

Let \( S_y \) be an array of the points in that region, sorted by decreasing \( y \)-coordinate value.
Slab Might Contain All Points

• Let $S_y$ be an array of the points in that region, sorted by decreasing $y$-coordinate value.

• $S_y$ might contain all the points, so we can’t just check every pair inside it.

**Theorem**

Suppose $S_y = p_1, \ldots, p_m$. If $\text{dist}(p_i, p_j) < d$ then $j - i \leq 15$.

In other words, if two points in $S_y$ are close enough in the plane, they are close in the array $S_y$. 
Divide the region up into squares with sides of length \( d/2 \):

How many points in each box?

At most 1 because each box is completely contained in one half and no two points in a half are closer than \( d \).
Proof, 1

Divide the region up into squares with sides of length \( d/2 \):

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

How many points in each box?

At most 1 because each box is completely contained in one half and no two points in a half are closer than \( d \).
Suppose 2 points are separated by $> 15$ indices.

Then, at least 3 full rows separate them (the packing shown is the smallest possible).

But the height of 3 rows is $> 3d/2$, which is $> d$.

So the two points are farther than $d$ apart.
Linear Time Merge

Therefore, we can scan $S_y$ for pairs of points separated by $< d$ in linear time.

ClosestPair(Px, Py):
    if $|Px| == 2$: return dist(Px[1],Px[2])  // base

    d1 = ClosestPair(FirstHalf(Px,Py))  // divide
    d2 = ClosestPair(SecondHalf(Px,Py))
    d = min(d1,d2)

    Sy = points in Py within d of L  // merge
    For i = 1,...,|Sy|:
        For j = 1,...,15:
            d = min(dist(Sy[i], Sy[j]), d )
    Return d
Total Running Time:

- Divide set of points in half each time: $O(\log n)$ depth recursion
- Merge takes $O(n)$ time.
- Recurrence: $T(n) \leq 2T(n/2) + cn$
- Same as MergeSort $\implies O(n \log n)$ time.
CMSC 451: Dynamic Programming

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Sections 6.1&6.2 of Algorithm Design by Kleinberg & Tardos.
Dynamic Programming

• Our 3rd major algorithm design technique

• Similar to divide & conquer
  • Build up the answer from smaller subproblems
  • More general than “simple” divide & conquer
  • Also more powerful

• Generally applies to algorithms where the brute force algorithm would be exponential.
Weighted Interval Scheduling

Recall the interval scheduling problem we’ve seen several times: choose as many non-overlapping intervals as possible.

What if each interval had a value?

Problem (Weighted Interval Scheduling)

*Given a set of $n$ intervals $(s_i, f_i)$, each with a value $v_i$, choose a subset $S$ of non-overlapping intervals with $\sum_{i \in S} v_i$ maximized.*
Example

Note that our simple greedy algorithm for the unweighted case doesn’t work.

This is because some interval can be made very important with a high weight.
Greedy Algorithm For Unweighted Case:

1. Sort by increasing finishing time

2. Repeat until no intervals left:
   1. Choose next interval
   2. Remove all intervals it overlaps with
Suppose for now we’re not interested in the actual set of intervals.

Only interested in the value of a solution (aka it’s cost, score, objective value).

This is typical of DP algorithms:

- You want to find a solution that optimizes some value.
- You first focus on just computing what that optimal value would be. E.g. what’s the highest value of a set of compatible intervals?
- You then post-process your answer (and some tables you’ve created along the way) to get the actual solution.
Another way to look at Weighted Interval Scheduling:

Assume that the intervals are sorted by finishing time and represent each interval by its value.

Goal is to choose a subset of the values of maximum sum, so that none of the chosen (√) intervals overlap:

<table>
<thead>
<tr>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$v_3$</th>
<th>$v_4$</th>
<th>$\cdots$</th>
<th>$v_{n-1}$</th>
<th>$v_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>√</td>
<td>X</td>
<td>√</td>
<td></td>
<td>√</td>
<td>X</td>
</tr>
</tbody>
</table>
Definition

\[ p(j) = \text{the largest } i < j \text{ such that interval } i \text{ doesn’t overlap with } j. \]

\[ p(1) = 0 \quad p(2) = 0 \quad p(3) = 1 \quad p(4) = 0 \quad p(5) = 3 \quad p(6) = 3 \]

\[ p(j) \text{ is the interval farthest to the right that is compatible with } j. \]
What does an OPT solution look like?

Let OPT be an optimal solution.

Let \( n \) be the last interval.

Does OPT contain interval \( n \)?

- Yes
  - \( \text{OPT} = n + \text{Optimal solution on } \{1, \ldots, p(n)\} \)

- No
  - \( \text{OPT} = \text{optimal solution on } \{1, \ldots, n-1\} \)
Definition

\[ \text{OPT}(j) = \text{the optimal solution considering only intervals } 1, \ldots, j \]

\[
\text{OPT}(j) = \max \begin{cases} 
 v_j + \text{OPT}(p(j)) & \text{if } j \text{ is in OPT solution} \\
 \text{OPT}(j - 1) & \text{if } j \text{ not in solution} \\
 0 & j = 0
\end{cases}
\]

This kind of recurrence relation is very typical of dynamic programming.
Implementing the recurrence directly:

**WeightedIntSched(j):**

- If $j = 0$: Return 0
- Else:
  - Return $\max(v[j] + \text{WeightedIntSched}(p[j]), \text{WeightedIntSched}(j-1))$

Unfortunately, this is exponential time!
Why is this exponential time?

Consider this set of intervals:

- What’s the shortest path from the root to a leaf?
- Total # nodes is $\geq 2^{n/2}$
- Each node does constant work $\implies \Omega(2^n)$
Why is this exponential time?

Consider this set of intervals:

\[ p(j) = j - 2 \text{ for all } j \geq 3 \]

- What’s the shortest path from the root to a leaf? \( n/2 \)
- Total \# nodes is \( \geq 2^{n/2} \)
- Each node does constant work \( \implies \Omega(2^n) \)
Problem: Repeatedly solving the same subproblem.

Solution: Save the answer for each subproblem as you compute it.

When you compute $OPT(j)$, save the value in a global array $M$. 
MemoizedIntSched(j):
    If j = 0: Return 0
    Else If M[j] is not empty:
        Return M[j]
    Else
        M[j] = max(
            v[j] + MemoizedIntSched(p[j]),
            MemoizedIntSched(j-1)
        )
    Return M[j]

• Fill in 1 array entry for every two calls to MemoizedIntSched. 
  \[\Rightarrow O(n)\]
Easier Algorithm

When we compute $M[j]$, we only need values for $M[k]$ for $k < j$:

ForwardIntSched(j):

```plaintext
M[0] = 0
for j = 1, ..., n:
    M[j] = \max(v[j] + M[p(j)], M[j-1])
```

**Main Idea of Dynamic Programming:** solve the subproblems in an order that makes sure when you need an answer, it’s already been computed.
Example

$v_j + M[p(j)]$

$M[j-1]$
Example

\[ v_j + M[p(j)] \]

\[ M[j-1] \]

\[ 10 \]

\[ 0 \]
Example

\[ v_j + M[p(j)] \]

\[ M[j-1] \]

10 10
20 20
5 15
15 5

0 10 20
1 2 3
4 5
Example

\[ v_j + M[p(j)] \]

\[ M[j-1] \]

\[
\begin{array}{c c c c c}
0 & 10 & 20 & 20 & \\
1 & 2 & 3 & 4 & 5
\end{array}
\]
Example

\[ v_j + M[p(j)] \]

\[ M[j-1] \]

\[
\begin{array}{ccccc}
0 & 10 & 20 & 20 & 30 \\
1 & 2 & 3 & 4 & 5 \\
10 & 20 & 5 & 20 & 15 & 20 & 30 \\
15 & 20 & 30 \\
\end{array}
\]
Example

\[ v_j + M[p(j)] \]

\[ M[j-1] \]
General DP Principles

1. Optimal value of the original problem can be computed easily from some subproblems.

2. There are only a polynomial number of subproblems.

3. There is a “natural” ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems.
General DP Principles

1. Optimal value of the original problem can be computed easily from some subproblems. \( \text{OPT}(j) = \max \text{ of two subproblems} \)

2. There are only a polynomial \( \# \) of subproblems. \( \{1, \ldots, j\} \) for \( j = 1, \ldots, n \).

3. There is a “natural” ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems. \( \{1, 2, 3\} \) is smaller than \( \{1, 2, 3, 4\} \)
Getting the actual solution

We now have an algorithm to find the value of OPT. How do we get the actual choices of intervals?

Interval \( j \) is in the optimal solution for the subproblem on intervals \( \{1, \ldots, j\} \) only if

\[
v_j + OPT(p(j)) \geq OPT(j - 1)
\]

So, interval \( n \) is in the optimal solution only if

\[
v[n] + M[p[n]] \geq M[n - 1]
\]

After deciding if \( n \) is in the solution, we can look at the relevant subproblem: either \( \{1, \ldots, p(n)\} \) or \( \{1, \ldots, n - 1\} \).
Example

\[ v_j + M[p(j)] \]

\[ M[j-1] \]
Example

\[ v_j + M[p(j)] \]

\[ \begin{array}{cccccc}
10 & 20 & 20 & 30 & 35 \\
1 & 2 & 3 & 4 & 5 \\
\end{array} \]

\[ \begin{array}{cccccc}
10 & 20 & 15 & 30 & 35 \\
0 & 10 & 20 & 20 & 30 \\
\end{array} \]
Example

$v_j + M[p(j)]$

\begin{align*}
10 & 10 \\
20 & 20 \\
15 & 15 \\
20 & 20 \\
30 & 30 \\
35 & 35 \\
\end{align*}

$M[j-1]$

\begin{align*}
0 & 0 \\
10 & 10 \\
20 & 20 \\
20 & 20 \\
30 & 30 \\
30 & 30 \\
\end{align*}
Example
BacktrackForSolution(M, j):
    If j > 0:
        If v[j] + M[p[j]] \geq M[j-1]:  // find the winner
            Output j  // j is in the soln
            BacktrackForSolution(M, p[j])
        Else:
            BacktrackForSolution(M, j-1)
        EndIf
    EndIf
EndIf
Running Time

Time to sort by finishing time: $O(n \log n)$

Time to compute $p(n)$: $O(n^2)$

Time to fill in the $M$ array: $O(n)$

Time to backtrack to find solution: $O(n)$
CMSC 451: Subset Sum & Knapsack

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 6.4 of Algorithm Design by Kleinberg & Tardos.
Subsetsum

**Subset Sum**

Given:

- an integer bound $W$, and
- a collection of $n$ items, each with a positive, integer weight $w_i$,

find a subset $S$ of items that:

\[ \sum_{i \in S} w_i \text{ maximizes} \quad \text{while keeping} \quad \sum_{i \in S} w_i \leq W. \]

**Motivation:** you have a CPU with $W$ free cycles, and want to choose the set of jobs (each taking $w_i$ time) that minimizes the number of idle cycles.
We assume $W$ and each $w_i$ is an integer.
Optimal Notation

**Notation:**

- Let $S^*$ be an optimal choice of items (e.g. a set $\{1, 4, 8\}$).
- Let $OPT(n, W)$ be the value of the optimal solution.
- We design a dynamic programming algorithm to compute $OPT(n, W)$.

**Subproblems:**

- To compute $OPT(n, W)$: We need the optimal value for subproblems consisting of the first $j$ items for every knapsack size $0 \leq w \leq W$.
- Denote the optimal value of these subproblems by $OPT(j, w)$. 
Recurrence: How do we compute $OPT(j, w)$ given solutions to smaller subproblems?

$$OPT(j, W) = \max \begin{cases} OPT(j - 1, W) & \text{if } j \not\in S^* \\ w_j + OPT(j - 1, W - w_j) & \text{if } j \in S^* \end{cases}$$

Special case: if $w_j > W$ then $OPT(j, W) = OPT(j - 1, W)$. 
Another way to write it...

$$OPT(j, W) = \begin{cases} 
OPT(j - 1, W) & \text{if } w_j > W \\
\max \left\{ OPT(j - 1, W) \right\} & \text{if } j \not\in S^* \\
w_j + OPT(j - 1, W - w_j) & \text{if } j \in S^* 
\end{cases}$$

Note: Because we don’t know the answer to the blue questions, we have to try both.
The table of solutions

![Table of solutions](image_url)
Filling in a box using smaller problems

\[ \text{OPT}(j, w) = \begin{cases} \text{OPT}(j-1, W-w_j) & \text{if } w_j \leq w \\ \text{OPT}(j, w) & \text{otherwise} \end{cases} \]
Filling in a box using smaller problems

OPT(j, w)

OPT(j-1, W-w_j)

OPT(j-1, W)
When we fill in the gray box, we also record which subproblem was chosen in the maximum:

\[
OPT(j, w) = \max \{ OPT(j-1, W) \text{ if } j \text{ is not chosen, } \}
\]

\[
OPT(j-1, W-w_j) \text{ if } j \text{ is chosen.}
\]
Filling in the Matrix

Fill matrix from bottom to top, left to right.

When you are filling in box, you only need to look at boxes you’ve already filled in.
Pseudocode

SubsetSum(n, W):
    Initialize M[0,w] = 0 for each w = 0,...,W
    Initialize M[i,0] = 0 for each i = 1,...,n

    For i = 1,...,n:
        For w = 0,...,W:
            If w[i] > w:
                M[i,w] = M[i-1,w]

            M[i,w] = max( M[i-1,w], w[j] + M[i-1, W-w[j]] )

    Return M[n,W]
Finding The Choice of Items

Follow the arrows backward starting at the top right:

Which items does this path imply?

8, 5, 4, 2
Follow the arrows backward starting at the top right:

Which items does this path imply? 8, 5, 4, 2
Runtime:

- $O(nW)$ to fill in the matrix.
- $O(n)$ time to follow the path backwards.
- Total running time is $O(nW)$.

This is pseudo-polynomial because it depends on the size of the input numbers.
Knapsack

Given:
- a bound $W$, and
- a collection of $n$ items, each with a weight $w_i$,
- a value $v_i$ for each weight

Find a subset $S$ of items that:

$$\text{maximizes } \sum_{i \in S} v_i \text{ while keeping } \sum_{i \in S} w_i \leq W.$$ 

**Difference from Subset Sum:** want to maximize value instead of weight.
How can we solve Knapsack?
Knapsack

Subset Sum:

\[
OPT(j, W) = \max \begin{cases} 
OPT(j - 1, W) & \text{if } j \notin S^* \\
 w_j + OPT(j - 1, W - w_j) & \text{if } j \in S^*
\end{cases}
\]
**Subset Sum:**

$$OPT(j, W) = \max \begin{cases} OPT(j - 1, W) & \text{if } j \not\in S^* \\ w_j + OPT(j - 1, W - w_j) & \text{if } j \in S^* \end{cases}$$

---

**Knapsack:**

$$OPT(j, W) = \max \begin{cases} OPT(j - 1, W) & \text{if } j \not\in S^* \\ v_j + OPT(j - 1, W - w_j) & \text{if } j \in S^* \end{cases}$$
0-1 Knapsack

You’re presented with \( n \), where item \( i \) has value \( v_i \) and size \( w_i \). You have a knapsack of size \( W \), and you want to take the items \( S \) so that

- \( \sum_{i \in S} v_i \) is maximized, and
- \( \sum_{i \in S} w_i \leq W \).

This is a hard problem. However, if we are allowed to take fractions of items we can do it with a simple greedy algorithm:

- Value of a fraction \( f \) of item \( i \) is \( f \cdot v_i \);
- Weight of a fraction \( f \) is \( f \cdot w_i \).
**Idea:** Sort the items by $p_i = v_i/w_i$
Larger $v_i$ is better, smaller $w_i$ is better.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$30$</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>$40$</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>$45$</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>$100$</td>
<td>4</td>
<td>25</td>
</tr>
</tbody>
</table>

**Knapsack size = 6**

$$30 + 100 + (1/2) \times 40 = 150$$
This greedy algorithm doesn’t work for 0-1 knapsack, where we can’t take part of an item:

knapsack size = 6
Greedy Choice:

$30 + $100 = $130

A better choice:
Given: Two strings

\[ a = a_1 a_2 a_3 a_4 \ldots a_m \]
\[ b = b_1 b_2 b_3 b_4 \ldots b_n \]

\[ a_i, b_i \in L \] for some alphabet \( L \) like \{A, C, G, T\}.

Compute how “similar” the two strings are.

What do we mean by similarity between two strings?
Alignment Examples

prin-ciple

prinncipal

(1 gap, 2 mm)

misspell

mis-pell

(1 gap)

aa-bb-ccaabb

ababbbbc-a-b-

(5 gaps, 1 mm)

prin-cip-le

prinnccipale-

(3 gaps, 0 mm)

prehistoric

---historic

(3 gaps, 0 mm)

al-go-rithm-

alKhwariz-mi

(4 gaps, 3 mm)
Motivation

- Alignment is used extensively in molecular biology, where $a$ and $b$ are the DNA sequences of two genes (see NCBI BLAST)

- Spell checkers
Parameters:

- “gap” is the cost of leaving a gap.
- \( \text{cost}(x, y) \) is the cost of matching character \( x \) with character \( y \).

Objective function:

- Cost of an alignment is sum of the costs of the matches plus the number of gaps.
The Problem as a Matching

Each sequence is a set of nodes (one for each character).

Looking for a low-cost matching between the two sequences:

\[ \begin{align*}
\text{a=} & \quad x \quad x \quad y \quad z \quad x \quad x \quad x \\
\text{b=} & \quad y \quad y \quad x \quad x \quad y
\end{align*} \]

Cost of the matching is:

\[ \text{gap} \times \# \text{unmatched} + \sum_{(a_i, b_j)} \text{cost}(a_i, b_j) \]

“No-crossing” rule!
Consider the last character in each string:

\[ a = a_1 a_2 a_3 a_4 \ldots a_m \]
\[ b = b_1 b_2 b_3 b_4 \ldots b_n \]

Consider the 4 possibilities:

1. \((a_m, b_n)\) are matched to each other
2. \(a_m\) is not matched at all
3. \(b_n\) is not matched at all
4. \(a_m\) is matched to some \(b_j\) \((j \neq n)\), and \(b_n\) is matched to some \(a_k\) \((k \neq m)\).
The Three Possibilities

This last possibility can’t happen:

\[ a_m \text{ is matched to some } b_j, \text{ and } b_n \text{ is matched to some } a_k. \]

If that occurred, then there would be a crossing in the matching:

So, only possibilities are:

1. \((a_m, b_n)\) is in the matching
2. \(a_m\) is not matched
3. \(b_n\) is not matched
Recurrence

Turn these possibilities into a recurrence:

$$OPT(i, j) = \max \begin{cases} 
\text{cost}(a_i, b_j) + OPT(i - 1, j - 1) & \text{match } a_i, b_j \\
gap + OPT(i - 1, j) & a_i \text{ is not matched} \\
gap + OPT(i, j - 1) & b_j \text{ is not matched}
\end{cases}$$

Base case: \( OPT(i, 0) = i \times \text{gap} \) and \( OPT(0, j) = j \times \text{gap} \).

(matching \( i \) characters to 0 characters must use \( i \) gaps.)
Align(X,Y):
    For i = 1,...,m: A[i,0] = i*gap
    For j = 1,...,n: A[0,j] = j*gap

    For i = 1,...,m:
        For j = 1,...,n:
            A[i,j] = max(
                cost(a[i],b[j]) + A[i-1,j-1],
                gap + A[i-1,j],
                gap + A[i,j-1]
            )

    EndFor
    EndFor
Return A[m,n]
Running Time

Number of subproblems $= m \times n$.

Each subproblem takes constant time to solve.

Total running time $= O(mn)$. 
Recasting as a Graph

The diagram illustrates a graph with nodes labeled from $(0,0)$ to $(m,n)$, representing elements $a_1, a_2, a_3, a_4, a_5$ on the horizontal axis and $b_1, b_2, b_3, b_4$ on the vertical axis. Each edge from $(i-1,j-1)$ to $(i,j)$ has weight $\text{cost}(a_i, b_j)$. The term "gap" indicates a missing edge or a point in the graph where no connection exists.
Theorem

Let $f(i, j)$ be the cost of the shortest path from $(0, 0)$ to $(i, j)$. Then $\text{OPT}(i, j) = f(i, j)$ for all $i, j$. 
CMSC 451: RNA Folding

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 6.5 of Algorithm Design by Kleinberg & Tardos.
RNA Folding

**RNA:** ACGGGGUUUAAAUUUCCCUUAUAT

In the cell, RNA folds up:

- G and C stick together
- A and U stick together
RNA Folding Rules:

1. If two bases are closer than 4 bases apart, they cannot pair.
2. Each base is matched to at most one other base.
3. The allowable pairs are \{U, A\} and \{G, C\}.
4. Pairs cannot “cross.”
This is not allowed:

In other words, if \((i, j)\) and \((k, \ell)\) are paired, then we must have \(i < k < \ell < j\).

Paired bases have to be nested.
RNA Folding Problem

**RNA Folding**

Given: a string \( r = b_1, b_2, b_3, \ldots, b_n \), with \( b_i \in \{A, C, U, G\} \)

Find: the largest set of pairs \( S = \{(i,j)\} \), where

\( i, j \in \{1, 2, \ldots, n\} \), that satisfies the RNA Folding Rules.

**Goal:** match as many pairs of bases as possible.
Subproblems

j is not paired with anything

\[ j \text{ is not paired with anything} \]

OPT(1, j-1)

j is paired with some \( t \leq j - 4 \)

\[ j \text{ is paired with some } t \leq j - 4 \]

OPT(1, t-1)

OPT(t+1, j-1)
We have a subproblem for every interval $(i, j)$.

How many subproblems are there?
We have a subproblem for every interval \((i, j)\).

How many subproblems are there?

\[
\binom{n}{2} = O(n^2)
\]
Recurrence

If $j - i \leq 4$: \( OPT(i,j) = 0 \)

If $j - i > 4$:

\[
OPT(i,j) = \max \left\{ \begin{array}{l}
OPT(i,j - 1) \\
\max_t \{ 1 + OPT(i,t - 1) + OPT(t + 1,j - 1) \}
\end{array} \right. 
\]

In the 2nd case above, we try all possible $t$ to pair with $j$. That is $t$ runs from $i$ to $j - 4$. 
Order to Solve the Problems

- In what order should we solve the subproblems?
- What problems do we need to solve $OPT(i, j)$?

Subproblem Ordering
We solve $OPT(i, j)$ in order of increasing value of $j - i$.
Order to Solve the Problems

• In what order should we solve the subproblems?

• What problems do we need to solve $OPT(i, j)$?

• $OPT(i, t - 1)$ and $OPT(t + 1, j - 1)$ for every $t$ between $i$ and $j$.

• In what sense are these problems “smaller?”
Order to Solve the Problems

• In what order should we solve the subproblems?

• What problems do we need to solve $OPT(i, j)$?

  - $OPT(i, t − 1)$ and $OPT(t + 1, j − 1)$ for every $t$ between $i$ and $j$.

• In what sense are these problems “smaller?”

• They involve smaller intervals of the string.

Subproblem Ordering

We solve $OPT(i, j)$ in order of increasing value of $j − i$. 
Initialize $OPT[i,j]$ to 0 for $1 \leq i,j \leq n$

For $k = 5, 6, \ldots, n-1$  // interval length
    For $i = 1, 2, \ldots, n-k$  // interval start
        Set $j = i + k$  // interval end

    // find the best $t$
    best_t = 0
    For $t = i, \ldots, j-1$:
        best_t = max(best_t, 1 + $OPT[i,t-1]+OPT[t+1,j-1]$)

    // Either pair $j$ with $t$ or nothing
    $OPT[i,j] = \max(best_t, OPT[i,j-1])$

EndFor
EndFor
Return $OPT[1,n]$
Running Time

1. $O(n^2)$ subproblems.

2. Now, it takes $O(n)$ time to solve each subproblem. (have to search over all possible choices for $t$.)

3. Total running time is $O(n^3)$.
CMSC 451: Shortest Paths with Negative Weights

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 6.8 of *Algorithm Design* by Kleinberg & Tardos.
Dynamic Programming Pattern:

1. Decompose the problem into subproblems.

2. Recursively define the value of a solution of a subproblem by the value of solutions of smaller subproblems.

3. Compute the value of the solutions for subproblems from smaller to larger.

4. Use the choices made (arrows) to reconstruct an actual solution.
Principle of Optimality: A problem obeys this principle if an optimal solution to the problem contains within it optimal solutions to subproblems.

- Solution to the problem requires making a choice e.g. include the last interval or not?

- This choice leaves 1 or more subproblems unsolved.

- Assuming you’re given the optimal solution to these subproblems, you show how to construct the optimal solution to the larger subproblem.
Shortest Path Problem

Shortest Path with Negative Weights

Given directed graph $G$ with weighted edges (weights may be positive or negative), find the shortest path from $s$ to $t$. 
**Negative cycles:** If some cycle has a negative total cost, we can make the \( s \rightarrow t \) path as low cost as we want:

Go from \( s \) to some node on the cycle, and then travel around the cycle many times, eventually leaving to go to \( t \).

Assume, therefore, that \( G \) has no negative cycles.
Let’s just add a big number!

- Adding a large number $M$ to each edge doesn’t work!

- The cost of a path $P$ will become $M \times \text{length}(P) + \text{cost}(P)$.

- If $M$ is big, the number of hops (length) will dominate.
**Definition**

$OPT(v, i)$ is minimum cost of a path from $s$ to $v$ that uses at most $i$ edges.

1. If best $s - v$ path uses at most $i - 1$ edges, then
   
   $OPT(v, i) = OPT(v, i - 1)$.

2. If best $s - v$ uses $i$ edges, and the last edge is $(w, v)$, then
   
   $OPT(v, i) = c_{wv} + OPT(w, i - 1)$. 
Subproblems, picture

\[
\text{OPT}(w_1, i-1) \\
\text{OPT}(w_2, i-1)
\]
Recurrence

Let $N(w)$ be the neighbors of $w$.

$OPT(v, i) =$ cost of best path from $s$ to $v$ using at most $i$ edges.

**Recurrence:**

$$OPT(v, i) = \min \begin{cases} 
OPT(v, i - 1) \\
\min_{w \in N(v)} OPT(w, i - 1) + c_{wv}
\end{cases}$$

**Goal:** Compute $OPT(t, n - 1)$. 
What do we need $i$?

Why do we introduce the variable $i$?
What do we need $i$?

Why do we introduce the variable $i$?

- $i$ gives us a natural ordering on the problems from larger to smaller.

- To solve $OPT(v, i)$ we need to know only $OPT(w, k)$ for $k < i$.

⇒ by expanding our class of subproblems, ordering them can become simpler.
ShortestPath(G, s, t):
   For i = 1,...,n-1:
      For v in V:
         // try all possible w’s:
         best_w = None
         for w in N(v):
            best_w = min(best_w, OPT[i-1,w] + c[w,v])

         M[v,i] = max(best_w, OPT[v, i-1])
      EndFor
   EndFor
Return M[t, n-1]
Simple Analysis:

- $O(n^2)$ subproblems
- $O(n)$ time to compute each entry in the table (have to search over all possible neighbors $w$).
- Therefore, runs in $O(n^3)$ time.

A better analysis:

- Let $n_v$ be the number of edges entering $v$.
- Filling in each entry actually only takes $O(n_v)$ time.
- Total time $= O \left( n \sum_{v \in V} n_v \right) = O(nm)$. 
CMSC 451: Network Flows

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Sections 7.1 & 7.2 of *Algorithm Design* by Kleinberg & Tardos.
• Our 4th major algorithm design technique (greedy, divide-and-conquer, and dynamic programming are the others).

• A little different than the others: we’ll see an algorithm for one problem (and minor variants) that is so useful that we can apply to many practical problems.

• Called network flow.
Network flow problem, e.g.

Suppose you want to ship natural gas from Alaska to Texas.

There are pipes, each with a capacity.

How can you send as much gas as possible?
A flow network is a connected, directed graph $G = (V, E)$.

- Each edge $e$ has a non-negative, integer capacity $c_e$.
- A single source $s \in V$.
- A single sink $t \in V$.
- No edge enters the source and no edge leaves the sink.
Assumptions

To repeat, we make these assumptions about the network:

1. Capacities are integers.
2. Every node has one edge adjacent to it.
3. No edge enters the source and no edge leaves the sink.

These assumptions can all be removed.
**Def.** An **s-t flow** is a function \( f : E \rightarrow \mathbb{R}^{\geq 0} \) that assigns a real number to each edge.

Intuitively, \( f(e) \) is the amount of material carried on the edge \( e \).
Flow constraints

Constraints on $f$:

1. $0 \leq f(e) \leq c_e$ for each edge $e$. (capacity constraints)

2. For each node $v$ except $s$ and $t$, we have:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ leaving } v} f(e).$$

(balance constraints: whatever flows in, must flow out).

![Diagram of a flow network with capacities on edges.]
The **value** of flow $f$ is:

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

This is the amount of material that $s$ is able to send out.

**Notation**:

- $f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$

- $f^{\text{out}}(v) = \sum_{e \text{ leaving } v} f(e)$

Balance constraints becomes: $f^{\text{in}}(v) = f^{\text{out}}(v)$ for all $v \not\in \{s, t\}$
Maximum Flow Problem

Definition (Value)
The value \( v(f) \) of a flow \( f \) is \( f^{\text{out}}(s) \).

That is: it is the amount of material that leaves \( s \).

Maximum Flow Problem
Given a flow network \( G \), find a flow \( f \) of maximum possible value.
A Greedy Start:

1. Suppose we let $f(e) = 0$ for all edges (no flow anywhere).

2. Choose some $s - t$ path and “push” flow along it up to the capacities. Repeat.

3. When we get stuck, we can erase some flow along certain edges.

How do we make this more precise?
Example

After 1 path, we've allocated 20 units

Want to send the blue path
We define a **residual graph** $G_f$. $G_f$ depends on some flow $f$:

1. $G_f$ contains the same nodes as $G$.

2. **Forward edges**: For each edge $e = (u, v)$ of $G$ for which $f(e) < c_e$, include an edge $e' = (u, v)$ in $G_f$ with capacity $c_e - f(e)$.

3. **Backward edges**: For each edge $e = (u, v)$ in $G$ with $f(e) > 0$, we include an edge $e' = (v, u)$ in $G_f$ with capacity $f(e)$. 
- If $f(e) < c_e$, add edge $e$ to $G_f$ with capacity $c_e - f(e)$. (remaining capacity left)
- If $f(u, v) > 0$, add reverse edge $(v, u)$ with capacity $f(e)$. (can erase up to $f(e)$ capacity)
Augmenting Paths

- Let $P$ be an $s - t$ path in the residual graph $G_f$.

- Let $\text{bottleneck}(P, f)$ be the smallest capacity in $G_f$ on any edge of $P$.

- If $\text{bottleneck}(P, f) > 0$ then we can increase the flow by sending $\text{bottleneck}(P, f)$ along the path $P$. 
If \( \text{bottleneck}(\mathcal{P}, f) > 0 \) then we can increase the flow by sending \( \text{bottleneck}(\mathcal{P}, f) \) along the path \( \mathcal{P} \):

\[
\text{augment}(f, \mathcal{P}): \\
b = \text{bottleneck}(\mathcal{P}, f) \\
\text{For each edge } (u, v) \in \mathcal{P}: \\
\quad \text{If } e = (u, v) \text{ is a forward edge:} \\
\qquad \text{Increase } f(e) \text{ in } G \text{ by } b \quad //add \text{ some flow} \\
\quad \text{Else:} \\
\qquad e' = (v, u) \\
\qquad \text{Decrease } f(e') \text{ in } G \text{ by } b \quad //erase \text{ some flow} \\
\quad \text{EndIf} \\
\text{EndFor} \\
\text{Return } f
Ford-Fulkerson Algorithm

MaxFlow(G):

// initialize:
Set f[e] = 0 for all e in G

// while there is an s-t path in Gf:
While P = FindPath(s, t, Residual(G, f)) != None:
    f = augment(f, P)
    UpdateResidual(G, f)
EndWhile
Return f
After augment, we still have a flow

After \( f' = \text{augment}(P, f) \), we still have a flow:

**Capacity constraints:** Let \( e \) be an edge on \( P \):

- if \( e \) is forward edge, it has capacity \( c_e - f(e) \). Therefore,
  \[
  f'(e) = f(e) + \text{bottleneck}(P, f) \leq f(e) + c_e - f(e) \leq c_e
  \]

- if \( e \) is a backward edge, it has capacity \( f(e) \). Therefore,
  \[
  f'(e) = f(e) - \text{bottleneck}(P, f) \geq f(e) - f(e) = 0
  \]
**Balance constraints:** An s-t path in $G_f$ corresponds to some set of edges in $G$:

```
\[ s \rightarrow +b \rightarrow +b \rightarrow -b \rightarrow -b \rightarrow +b \rightarrow t \]
```

In other pictures,
Running Time

1. At every step, the flow values \( f(e) \) are integers.

2. At every step we increase the amount of flow \( v(f) \) sent by at least 1 unit.

3. We can never send more than \( C := \sum_{e \text{ leaving } s} c_e \).

Theorem

*The Ford-Fulkerson algorithm terminates in \( C \) iterations of the *While* loop.*
Running Time

1. At every step, the flow values $f(e)$ are integers. Start with ints and always add or subtract ints.

2. At every step we increase the amount of flow $\nu(f)$ sent by at least 1 unit.

3. We can never send more than $C := \sum_{e \text{ leaving } s} c_e$.

Theorem

*The Ford-Fulkerson algorithm terminates in $C$ iterations of the While loop.*
Time in the While loop

1. If $G$ has $m$ edges, $G_f$ has $\leq 2m$ edges.

2. Can find an $s - t$ path in $G_f$ in time $O(m + n)$ time with DFS or BFS.

3. Since $m \geq n/2$ (every node is adjacent to some edge), $O(m + n) = O(m)$.

Theorem

The Ford-Fulkerson algorithm runs in $O(mC)$ time.
Note this is **pseudo-polynomial** because it depends on the size of the integers in the input.

You can remove this with slightly different algorithms. E.g.:

- $O(nm^2)$: Edmonds-Karp algorithm (use BFS to find the augmenting path)
- $O(m^2 \log C)$ (see section 7.3), or
- $O(n^2 m)$ or $O(n^3)$ (see section 7.4).
Proof of correctness

How do we know the flow is maximum?
Definition

The \textbf{capacity}(A, B) of an s-t cut \((A, B)\) is the sum of the capacities of edges leaving \(A\).
A Cut Theorem

Theorem

Let \( f \) be an \( s-t \) flow and \((A, B)\) be an \( s-t \) cut. Then \( \nu(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \).
Theorem

Let $f$ be any $s$-$t$ flow and $(A, B)$ be any $s$-$t$ cut. Then $v(f) \leq \text{capacity}(A, B)$.

Proof.

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$$

prev. thm

$$\leq f^{\text{out}}(A)$$

$f^{\text{in}}(A)$ is $\geq 0$

$$= \sum_{e \text{ leaving } A} f(e)$$

by definition

$$\leq \sum_{e \text{ leaving } A} c_e$$

by capacity constraints

$$= \text{capacity}(A, B)$$

by definition
This theorem:

**Theorem**

Let $f$ be any $s$-$t$ flow and $(A, B)$ be any $s$-$t$ cut. Then $\nu(f) \leq \text{capacity}(A, B)$.

Says that any cut is bigger than any flow.

Therefore, cuts constrain flows. The minimum capacity cut constrains the maximum flow the most.

In fact, the capacity of the minimum cut always equals the maximum flow value.
Max-Flow = Min-Cut

minimum capacity of a cut = maximum value of a flow
Let $f^*$ be the flow returned by our algorithm. Look at $G_{f^*}$ but define a cut in $G$:

$A^* = \text{nodes reachable from } s \text{ in residual graph } G_{f^*}$

$\text{Cut} = (A^*, B^*)$

**Blue** edges must be saturated.
**Red** edges must have 0 flow $\implies \nu(f^*) = \text{capacity}(A^*, B^*)$. 
Max-Flow = Min-Cut, 3

- \((A^*, B^*)\) is an s-t cut because there is no path from \(s\) to \(t\) in the residual graph \(G_{f^*}\).

- **Edges** \((u, v)\) from \(A^*\) to \(B^*\) must be saturated — otherwise there would be a forward edge \((u, v)\) in \(G_{f^*}\) and \(v\) would be part of \(A^*\).

- **Edges** \((v, u)\) from \(B^*\) to \(A^*\) must be empty — otherwise there would be a backedge \((u, v)\) in \(G_{f^*}\) and \(v\) would be part of \(A^*\).
Therefore,

- $\nu(f^*) = \text{capacity}(A^*, B^*)$.
- No flow can have value bigger than $\text{capacity}(A^*, B^*)$.
- So, $f^*$ must be an maximum flow.
- And $(A^*, B^*)$ has to be a minimum-capacity cut.

**Theorem (Max-flow = Min-cut)**

*The value of the maximum flow in any flow graph is equal to the capacity of the minimum cut.*
Finding the Min-capacity Cut

Our proof that maximum flow = minimum cut can be used to actually **find** the minimum capacity cut:

1. Find the maximum flow $f^*$.
2. Construct the residual graph $G_{f^*}$ for $f^*$.
3. Do a BFS to find the nodes reachable from $s$ in $G_{f^*}$. Let the set of these nodes be called $A^*$.
4. Let $B^*$ be all other nodes.
5. Return $(A^*, B^*)$ as the minimum capacity cut.
To Summarize

**Summary:**

- Ford-Fulkerson algorithm can find max flow in $O(mC)$ time.

- **Algorithm idea:** Send flow along some path with capacity left, possibly “erasing” some flow we’ve already sent. Use residual graph to keep track of remaining capacities and flow we’ve already sent.

- We can eliminate $C$ to get a true polynomial algorithm by using BFS to find our augmenting paths.

- All cuts have capacity $\geq$ the value of all flows.

- Know the flow is maximum because its value equals the capacity of some cut.
CMSC 451: Maximum Bipartite Matching

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 7.5 of Algorithm Design by Kleinberg & Tardos.
The network flow problem is itself interesting.

But even more interesting is how you can use it to solve many problems that don’t involve flows or even networks.
• Suppose we have a set of people $L$ and set of jobs $R$.

• Each person can do only some of the jobs.

• Can model this as a bipartite graph →
• A **matching** gives an assignment of people to tasks.

• Want to get as many tasks done as possible.

• So, want a **maximum matching**: one that contains as many edges as possible.

• (This one is not maximum.)
Maximum Bipartite Matching

Given a bipartite graph $G = (A \cup B, E)$, find an $S \subseteq A \times B$ that is a matching and is as large as possible.

Notes:

- We’re given $A$ and $B$ so we don’t have to find them.
- $S$ is a **perfect matching** if every vertex is matched.
- *Maximum* is not the same as *maximal*: greedy will get to maximal.
Given an instance of bipartite matching,

Create an instance of network flow.

Where the solution to the network flow problem can easily be used to find the solution to the bipartite matching.
Reducing Bipartite Matching to Net Flow

```
People          Tasks
```

```
a 1
b 2
c 3
d 4
e 5
```

L R
Reducing Bipartite Matching to Net Flow

People

- a
- b
- c
- d
- e

Tasks

- 1
- 2
- 3
- 4
- 5

Source: L
Sink: R
Reducing Bipartite Matching to Net Flow
Using Net Flow to Solve Bipartite Matching

To Recap:

1. Given bipartite graph $G = (A \cup B, E)$, direct the edges from $A$ to $B$.
2. Add new vertices $s$ and $t$.
3. Add an edge from $s$ to every vertex in $A$.
4. Add an edge from every vertex in $B$ to $t$.
5. Make all the capacities 1.

The edges used in the maximum network flow will correspond to the largest possible matching!
• Because the capacities are integers, our flow will be integral.

• Because the capacities are all 1, we will either:
  • use an edge completely (sending 1 unit of flow) or
  • not use an edge at all.

• Let $M$ be the set of edges going from $A$ to $B$ that we use.

• We will show that
  1. $M$ is a matching
  2. $M$ is the largest possible matching
M is a matching

We can choose at most one edge leaving any node in A. We can choose at most one edge entering any node in B.

If we chose more than 1, we couldn’t have balanced flow.
• If there is a matching of \( k \) edges, there is a flow \( f \) of value \( k \).

• If there is a flow \( f \) of value \( k \), there is a matching with \( k \) edges.

\[
v(f) = \text{out}(A) - \text{in}(A)
\]
If there is a matching of \( k \) edges, there is a flow \( f \) of value \( k \).

- \( f \) has 1 unit of flow across each of the \( k \) edges.
- \( \leq 1 \) unit leaves & enters each node (except \( s, t \))

If there is a flow \( f \) of value \( k \), there is a matching with \( k \) edges.

\[
v(f) = f^{\text{out}}(A) - f^{\text{in}}(A)
\]
• We find the \textbf{maximum} flow $f$ (say with $k$ edges).

• This corresponds to a matching $M$ of $k$ edges.

• If there were a matching with $> k$ edges, we would have found a flow with value $> k$, contradicting that $f$ was maximum.

• Hence, $M$ is maximum.
Running Time

• How long does it take to solve the network flow problem on $G'$?

• The running time of Ford-Fulkerson is $O(m' C)$ where $m'$ is the number of edges, and $C = \sum_{e \text{ leaving } s} c_e$.

• $C = |A| = n$.

• The number of edges in $G'$ is equal to number of edges in $G$ $(m)$ plus $2n$.

• So, running time is $O((m + 2n)n) = (mn + n^2) = O(mn)$

Theorem

We can find maximum bipartite matching in $O(mn)$ time.
• Fold-Fulkerson can find a maximum matching in a bipartite graph in $O(mn)$ time.

• We do this by reducing the problem of maximum bipartite matching to network flow.
CMSC 451: Edge-Disjoint Paths

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 7.6 of Algorithm Design by Kleinberg & Tardos.
Suppose you want to send $k$ large files from $s$ to $t$ but never have two files use the same network link (to avoid congestion on the links).

Leads naturally to the Edge-Disjoint Paths problem:

$k$ Edge-disjoint Paths

Given directed graph $G$, and two nodes $s$ and $t$, find $k$ paths from $s$ to $t$ such that no two paths share an edge.
Again a Reduction!

- Given an instance of \textit{k-Edge-Disjoint Paths},
- Create an instance of \textit{Maximum Network Flow}.
- The maximum flow will be used to find the \textit{k} edge-disjoint paths.

Are there \textit{k} edge-disjoint paths?

What is the maximum flow in the graph?
There is a nice correspondence between paths and flows in unit capacity networks.

Suppose we had $k$ edge-disjoint $s - t$ paths.

We could sent 1 unit of flow along each path without violating the capacity constraints.

**Lemma (Paths $\implies$ Flow)**

*If there are $k$ edge-disjoint $s - t$ paths in directed, unit-weight graph $G$, then the maximum $s - t$ flow is $\geq k$.***
Flow $\implies$ Paths

Theorem (Flow $\implies$ Paths)

If there is a flow of value $k$ in a directed, unit-weight graph $G$, then there exist at least $k$ edge-disjoint $s-t$ paths.

In other words: if we can find a flow of value $k$, then we know it’s possible to “pack” at least $k$ edge-disjoint paths into the graph.

If we can prove this, then we know how to check whether the $k$ disjoint paths exist. The proof will also show how we can find the $k$ disjoint paths.

Note: by our previous discussion, we can assume that flow $f$ is a 0-1 flow: each edge contains either no flow, or 1 unit.
Theorem

If $f$ is a 0-1 flow of value $k$, then the set of edges where $f(e) = 1$ contains set of $k$ edge-disjoint paths.

Proof: By induction on the number of edges with $f(e) = 1$.

IH: Assume the thm holds for flows with fewer edges used than $f$.

Let $(s, u)$ be an edge that carries flow. Then by conservation we can find some edge leaving $u$ that also has 1 unit of flow.

Repeating this, either (1) we reach $t$ or (2) we loop around. We look at each of those cases on the next slides.
(1) Reach $t$: $k=2$
(1) Reach $t$: $k=1$
We find an $s - t$ path, reduce the flow along it to 0, creating new flow $f'$.

Value of new flow is $k - 1$.

And fewer edges have flow, so we apply our induction hypothesis: there are $k - 1$ edge-disjoint paths in flow $f'$.

Hence, in this case, there are $1 + k - 1 = k$ edge-disjoint paths.

Suppose, instead we loop back to some node we’ve already visited:
(2) Create a cycle:

Sending 3 units of flow
(2) Create a cycle:

Sending 3 units of flow
(2) Create a cycle:

Sending 3 units of flow
So,

We find a cycle, reduce the flow around it to 0, creating a new flow $f'$. Value of new flow is still $k$.

BUT there are fewer edges that have flow: so we can still apply our induction hypothesis: there are $k$ edge-disjoint paths in flow $f'$.

Hence, in either case, there are $k$ edge-disjoint paths.

**Base case:** When $k = 1$ there is clearly 1 edge disjoint path.
Path Decomposition Algorithm

The proof gives us a way to actually find the paths:

1. Find the maximum flow in $G$.

2. Start walking from $s$.

3. If you create a cycle, eliminate the flow around the cycle.

4. If you reach $t$, output the path you used to reach $t$. 
We can use a maximum flow algorithm to find $k$ edge-disjoint, s-t paths in a graph.

Embedded within any flow of value $k$ on a unit-capacity graph there are $k$ edge-disjoint paths.

In other words, the value of the flow gives us the number of edge disjoint paths.
Menger’s Theorem

**Theorem (Menger)**

*Given a directed graph* $G$ *with nodes* $s, t$ *the maximum number of edge-disjoint* $s$-$t$ *paths equals the minimum number of edges whose removal separates* $s$ *from* $t$.

**Useful:** Suppose you are a hacker who wants to disrupt communications between the US and Russia. You know the network. How many edges must you knock out?
CMSC 451: Image Segmentation

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 7.10 of Algorithm Design by Kleinberg & Tardos.
Image Segmentation

• Given an image, what is foreground and what is background?

• E.g. Hockey puck against the ice, football against the field, missiles against the sky.

• If \((x, y)\) is a foreground pixel, then nearby pixels are also likely foreground.
Given an image, what is foreground and what is background?

E.g. Hockey puck against the ice, football against the field, missiles against the sky.

If \((x, y)\) is a foreground pixel, then nearby pixels are also likely foreground.
Different applications will have different rules for identifying foreground and background pixels, e.g.:

If a pixel is brown, it is more likely to be the football.

Abstract those rules away, and suppose we have 2 nonnegative numbers for each pixel $i$:

1. $a_i = \text{likelihood that pixel } i \text{ is in foreground.}$
2. $b_i = \text{likelihood that pixel } i \text{ is in background.}$

How these values are generated depends on the application.
All else equal, if $a_i > b_i$, then we should make $i$ a foreground pixel.

**But:** we noted that foreground pixels tend to be near one another, and background pixels tend to be near one another.

To represent neighboring pixels, we encode an image by an undirected graph $G = (V, E)$.

- $V$ contains a vertex for each pixel
- $E$ contains an edge between pixels $i$ and $j$ if $i$ and $j$ neighbor each other.
Image Graph Example
Image Graph Example
We add new parameters to model separating neighboring pixels. For every neighboring pair of pixels \( \{i, j\} \), we have a parameter:

\[ p_{ij} = \text{the penalty for putting one of } i, j \text{ in foreground and the other in the background.} \]

Partition the set of pixels into two sets \( A \) and \( B \) to maximize:

\[
q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\text{sep}} p_{ij}.
\]
Image Graph Example
Min Cut?

**Image Segmentation:** Partition the vertices of the image graph into 2 sets $A, B$ to maximize $q(A, B)$.

**Minimum Cut:** Partition the vertices of the a directed graph into 2 sets $A, B$, with $s \in A$, $t \in B$, to minimize weight of edges crossing from $A$ to $B$.

Seem similar, but with some differences:

- Maximization vs. minimization
- Image segmentation has no source or sink
- Image segmentation has a more complicated objective function $q(A, B)$ with weights on the nodes
- Undirected vs. directed
• Given an instance of **Image Segmentation**, 

• Create an instance of **Minimum Capacity Cut**.

• Where the minimum cut can trivially be used to find the solution to the image segmentation.
Missing Source and Sink?

We add:

- a source $s$ with an edge $(s, u)$ for every vertex $u$.
- a sink $t$ with an edge $(u, t)$ for every vertex $u$.

**We will see:** $s$ will represent the foreground, and $t$ will represent the background.
Source and Sink Example
Directed Edges

We convert the currently **undirected** graph into a **directed** graph:

- Edges adjacent to $s$ are directed so they leave $s$.
- Edges adjacent to $t$ are directed so they enter $t$.
- All other edges are replaced by 2, anti-parallel edges:

![Diagram](image_url)
What have we done?

Given an image graph $G = (V, E)$ where $V$ represent the pixels and $E$ connect neighboring pixels, we produced a new graph $G' = (V', E')$ by:

1. $V' = V \cup \{s, t\} \leftarrow$ added source and sink.

2. $E'$ edges from $s$ to every pixel and from every pixel to $t$ plus edges $(i, j)$ and $(j, i)$ between each pair of pixels.

Last issue: how do we handle the parameters $a_i, b_i, p_{ij}$ and minimization vs. maximization?
Maximization vs. minimization

Let \( Q = \sum_i (a_i + b_i) \).

Note that:

\[
\sum_{i \in A} a_i + \sum_{j \in B} b_j = Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j
\]

Our old objective function was to maximize:

\[
q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{(i, j) \in E, i, j \text{ sep}} p_{ij}
\]

By above, this equals:

\[
q(A, B) = Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j - \sum_{(i, j) \in E, i, j \text{ sep}} p_{ij}
\]
Maximization vs. Minimization

We want to maximize:

\[ q(A, B) = Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j - \sum_{(i, j) \in E} p_{ij} \]

This is the same as \textbf{minimizing}:

\[ q'(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{(i, j) \in E} p_{ij} \]
We use the parameters $a_i$, $b_i$ and $p_{ij}$ as weights on the various edges:

- Edges between two pixels $i$ and $j$ get weight $p_{ij}$.
- Edge $(s, i)$ gets weight $a_i$.
- Edge $(j, t)$ get weight $b_j$.

The intuition here is that the capacity of an $s - t$ cut $(A, B)$ will equal the quantity we are trying to minimize. *We’ll see why.*

Therefore, if we find the min cut, we will find the best partition into foreground and background.
Weights Example
Capacity of a cut = quality of partition

We’ve designed the graph so that the capacity of an s-t cut (A,B) equals the quality of the partition defined by taking:

- A to be the set of foreground pixels (plus s)
- B to be the set of background pixels (plus t)

Why is this?
This is why:

Capacity of cut $(A, B) = ?$
minimize $q'(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{(i,j) \in E, i,j \text{ sep}} p_{ij}$

The edges of any cut $(A, B)$ of our graph can be divided into 3 groups:

- **Edges $(i, t)$, where $i \in A$:** this edge contributes $b_i$ to the capacity (and putting $i$ into the foreground costs us $b_i$).

- **Edges $(s, j)$, where $j \in B$:** this edge contributes $a_j$ to the capacity (and putting $j$ into the background costs us $a_j$).

- **Edges $(i, j)$, where $i \in A, j \in B$:** contributes $p_{ij}$ to the capacity (and separating $i$ and $j$ costs us $p_{ij}$).
We’ve shown that for a cut \((A, B)\) in \(G'\):

\[
\text{capacity}(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{(i, j) \in E_{i, j \text{ sep}}} p_{ij} = q'(A, B)
\]

Hence, finding the minimum capacity cut in our modified graph gives us the partition into foreground \(A\) and background \(B\) that maximizes \(q(A, B)\).

We can find that best partition using a minimum-cut algorithm on the modified graph \(G'\) and deleting \(s\) and \(t\) from the cut it returns.
CMSC 451: Max-Flow Extensions

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 7.7 of Algorithm Design by Kleinberg & Tardos.
Circulations with Demands

• Suppose we have multiple sources and multiple sinks.

• Each sink wants to get a certain amount of flow (its demand).

• Each source has a certain amount of flow to give (its supply).

• We can represent supply as negative demand.
Demand Example

supply $d_1 = -4$

supply $d_2 = -7$

demand $d_5 = 3$

demand $d_8 = 8$

1

3

2

5

4

6

7

8

9
Goal: find a flow $f$ that satisfies:

1. **Capacity constraints:** For each $e \in E$, $0 \leq f(e) \leq c_e$.

2. **Demand constraints:** For each $v \in V$,

$$f^{\text{in}}(v) - f^{\text{out}}(v) = d_v.$$

The demand $d_v$ is the excess flow that should come into node.
Sources and Sinks

Let $S$ be the set of nodes with **negative** demands (supply).

Let $T$ be the set of nodes with **positive** demands (demand).

In order for there to be a feasible flow, we must have:

$$
\sum_{s \in S} -d_s = \sum_{t \in T} d_t
$$

Let $D = \sum_{t \in T} d_t$. 
How can we turn the **circulation with demands** problem into the maximum flow problem?
Reduction

How can we turn the **circulation with demands** problem into the maximum flow problem?

1. Add a new source $s^*$ with an edge $(s^*, s)$ from $s^*$ to every node $s \in S$.

2. Add a new sink $t^*$ with an edge $(t, t^*)$ from $t^*$ to every node $t \in T$. 
How can we turn the **circulation with demands** problem into the maximum flow problem?

1. Add a new source $s^*$ with an edge $(s^*, s)$ from $s^*$ to every node $s \in S$.

2. Add a new sink $t^*$ with an edge $(t, t^*)$ from $t^*$ to every node $t \in T$.

The capacity of edges $(s^*, s) = -d_s$ (since $d_s < 0$, this is $+ve$)

The capacity of edges $(t, t^*) = d_t$. 
Feasible circulation if and only if there is a flow of value
\[ D = \sum_{t \in T} d_t. \]
Intuition:

- Capacity of edges \((s^*, s)\) limit the supply for source nodes \(s\).

- Capacity of edges \((t, t^*)\) require that \(d_t\) flow reaches each \(t\).

Hence, we can use max-flow to find these circulations.
Another extension: what if we want lower bounds on what flow goes through some edges?

In other words, we want to require that some edges are used.

**Goal: find a flow $f$ that satisfies:**

1. **Capacity constraints:** For each $e \in E$, $l_e \leq f(e) \leq c_e$.

2. **Demand constraints:** For each $v \in V$,

$$f^{\text{in}}(v) - f^{\text{out}}(v) = d_v.$$
Suppose we defined an initial flow $f_0$ by setting the flow along each edge equal to the lower bound. In other words: $f_0(e) = \ell_e$.

This flow satisfies the capacity constraints, but not the demand constraints.

Define: $L_v = f_0^{\text{in}}(v) - f_0^{\text{out}}(v)$.

Recall that the demand constraints say that $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$. Hence, $L_v$ is equal to the amount of the demand that $f_0$ satisfies at node $v$. 
For each node, our flow $f_0$ satisfies $L_v$ of its demand, hence we have:

**New demand constraints:**

$$f^\text{in}(v) - f^\text{out}(v) = d_v - L_v$$

Also, $f_0$ uses some of the edge capacities already, so we have:

**New capacity constraints:**

$$0 \leq f(e) \leq c_e - \ell_e$$

These constraints give a standard instance of the circulation problem.
(a) Small instance where one edge has a lower bound. This makes the most obvious flow not feasible.

(b) After transformation, we have an equivalent instance with no lower bounds.
Reduction:

Given a circulation instance $G$ with lower bounds, we:

1. subtract $l_e$ from the capacity of each edge $e$, and

2. subtract $L_v$ from the demand of each node $v$. 
   (This may create some new "sources" or "sinks".)

We then solve the circulation problem on this new graph to get a flow $f'$.

To find the flow that satisfies the original constraints, we add $l_e$ to every $f'(e)$. 
We can efficiently find a feasible flow for the following general problem:

**Circulations with demands and lower bounds**

**Given:**
- a directed graph $G$
- a nonnegative lower bound $l_e$ for each edge $e \in G$
- a nonnegative upper bound $c_e \geq l_e$ for each edge $e \in G$
- and a demand $d_v$ for every node

**Find:** a flow $f$ such that
- $l_e \leq f(e) \leq c_e$ for every $e$, and
- $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ for every $v$. 
We designed the algorithm for this general problem by reducing the Circulation with Lower Bounds problem to the Circulation without Lower Bounds problem. We in turn reduced that problem to the Max Flow problem.
Linear Programming

Suppose you are given:

- A matrix $A$ with $m$ rows and $n$ columns.
- A vector $\vec{b}$ of length $n$.
- A vector $\vec{c}$ of length $n$.

Find a length-$n$ vector $\vec{x}$ such that

$$A\vec{x} \leq \vec{b}$$

and so that

$$\vec{c} \cdot \vec{x} := \sum_{j=1}^{n} c_j x_j$$

is as large as possible.
The matrix inequality:

\[ A\vec{x} \leq \vec{b} \]

in pictures:

Each row of \( A \) gives coefficients of a linear expression: \( \sum_j a_{ij}x_j \).

Each row of \( A \) along with an entry of \( b \) specifies a linear inequality: \( \sum_j a_{ij}x_j \leq b_i \).
A little more general

\[
\text{maximize } \sum_{j} c_j x_j \\
\text{subject to } A\vec{x} \leq b
\]

What if you want to minimize?

What if you want to include a “≥” constraint \( \vec{a}_i \cdot \vec{x} \geq b_i \)?

What if you want to include a “=” constraint?
A little more general

\[
\text{maximize} \quad \sum_j c_j x_j \\
\text{subject to} \quad A\vec{x} \leq b
\]

What if you want to minimize? Rewrite to maximize \( \sum_j (-c_j)x_j \).

What if you want to include a “\( \geq \)” constraint \( \vec{a}_i \cdot \vec{x} \geq b_i \)?

What if you want to include a “\( = \)” constraint?
A little more general

\[
\text{maximize} \quad \sum_j c_j x_j \\
\text{subject to} \quad A\vec{x} \leq b
\]

What if you want to minimize? Rewrite to maximize \(\sum_j (-c_j)x_j\). 

What if you want to include a “\(\geq\)" constraint \(\vec{a}_i \cdot \vec{x} \geq b_i\)? Include the constraint \(-\vec{a}_i \cdot \vec{x} \leq -b_i\) instead.

What if you want to include a “\(=\)" constraint?
A little more general

\[
\text{maximize } \sum_j c_j x_j
\]

subject to \( A\vec{x} \leq b \)

What if you want to minimize? Rewrite to maximize \( \sum_j (−c_j)x_j \).

What if you want to include a \( \geq \) constraint \( \vec{a}_i \cdot \vec{x} \geq b_i \)? Include the constraint \( −\vec{a}_i \cdot \vec{x} \leq −b_i \) instead.

What if you want to include a \( = \) constraint? Include both the \( \geq \) and \( \leq \) constraints.

Hence, we can use \( = \) and \( \geq \) constraints and maximize if we want.
History of LP Algorithms

The Simplex Method:

- Oldest method.
- **Not** a polynomial time algorithm: for all proposed variants, there are examples LPs that take exponential time to solve.
- Still very widely used because it is fast in practice.

The Ellipsoid Method:

- Discovered in the 1970s.
- First polynomial time algorithm for linear programming.
- Horribly slow in practice, and essentially never used.

Interior Point Methods:

- Polynomial.
- Practical.
There is *lots* of software to solve linear programs:

- **CPLEX** — commercial, seems to be the undisputed winner.
- **GLPK** — GNU Linear Programming Solver
  (this is what we will use).
- **COIN-OR (CLP)** — Another open source solver.
- **...**
- **NEOS server** — http://www-neos.mcs.anl.gov/

Even Microsoft Excel has a built-in LP solver (though may not be installed by default).
What is Linear Programming Good For?
Maximum Flow

Given a directed graph $G = (V, E)$, capacities $c(e)$ for each edge $e$, and two vertices $s, t \in V$, find a flow $f$ in $G$ from $s$ to $t$ of maximum value.

What does a valid flow $f$ look like?

- $0 \leq f(e) \leq c(e)$ for all $e$.

- $\sum_{(u, v) \in E} f(u, v) = \sum_{(v, w) \in E} f(v, w)$
  for all $v \in V$ except $s, t$. 
Maximum Flow as LP

Create a variable $x_{uv}$ for every edge $(u, v) \in E$. The $x_{uv}$ values will give the flow: $f(u, v) = x_{uv}$.

Then we can write the maximum flow problem as a linear program:

\[
\text{maximize} \quad \sum_{(u,t) \in E} x_{uv} \\
\text{subject to} \quad 0 \leq x_{uv} \leq c_{uv} \quad \text{for every } (u, v) \in E \\
\quad \sum_{(u,v) \in E} x_{uv} = \sum_{(v,w) \in E} x_{vw} \quad \text{for all } v \in V \setminus \{s, t\}
\]

The first set of constraints ensure the capacity constraints are obeyed. The second set of constraints enforce flow balance.
Maximum Flow as MathProg

```plaintext
set V;               # rep vertices
set E within V cross V;  # rep edges
param C {(u,v) in E} >= 0;  # capacities
param s in V;       # source & sink
param t in V;

var X {(u,v) in E} >= 0, <= C[u,v];  # var for each edge

maximize flow: sum {(u,t) in E} X[u,t];

subject to balance {v in (V setminus {s,t})}:
   sum {(u,v) in E} X[u,v] = sum {(v,w) in E} X[v,w];

solve;
printf {(u,v) in E : X[u,v] > 0}: "%s %s %f", u,v,X[u,v];
end;
```
General MathProg Organization

- Declarations
- Objective Function
- Constraints
- Output
The “model” on the previous slide can work any graph and capacities.

The “data” file of the MathProg program gives the specific *instance* of the problem.

Support your graph was this:
data;
set V := 1..7;
set E := (1,2) (1,3) (1,4) (2,4) (2,7) (3,5) (4,6)
        (4,5) (5,7) (6,7) ;
param C : 1 2 3 4 5 6 7 :=
    1 . 3 1 7 . . .
    2 . . . 2 . . 6
    3 . . . . 9 . .
    4 . . . . . 1 .
    5 . . . . . . 4
    6 . . . . . . 4
    7 . . . . . . . ;
param s := 1;
param t := 7;
end;
Maximum Bipartite Matching

Given a bipartite graph \( G = (V, E) \), choose as large a subset of edges \( M \subseteq E \) as possible \( \text{that forms a matching} \).

The red text gives an objective function. The blue text gives constraints.
set A;
set B;
set E within A cross B;  # a bipartite graph

var X {e in E} >= 0, <= 1;  # variable for each edge

maximize numedges: sum {(u,v) in E} X[u,v];

s.t. matchA {u in A}: sum {(u,v) in E} X[u,v] <= 1;
s.t. matchB {v in B}: sum {(u,v) in E} X[u,v] <= 1;
end;
data;
set A := a b c d e f;
set B := 1..5;
set E : 1 2 3 4 5 :=
   a ++---
   b --+++ 
   c +-+-- 
   d +---+-
   e ----+ 
   f +-+--; 
end;
If we add one more kind of constraint, we get an integer linear program (ILP):

\[
\text{maximize } \sum_j c_j x_j \\
\text{subject to } A\vec{x} \leq b \\
x_i \in \{0, 1\} \quad \text{for all } i = 1, \ldots, n
\]

ILPs seem to be much more powerful and expressive than just LPs.

In particular, solving an ILP is NP-hard and there is no known polynomial time algorithm (and if P\(\neq\)NP, there isn’t one).

However: because of its importance, lots of optimized code and heuristics are available. CPLEX and GLPK for example provide solvers for ILPs.
Minimum Vertex Cover

Given graph $G = (V, E)$ choose a subset of vertices $C \subseteq V$ such that every edge in $E$ is incident to some vertex in $C$.

Why is this useful?

- In a social network, choose a set of people so that every possible friendship has a representative.
- On what nodes should you place sensors in an electric network to make sure you monitor every edge?
Vertex Cover as an ILP

Create a variable $x_u$ for every vertex $u$ in $V$.

We can then model the vertex cover problem as the following linear program:

$$\text{minimize} \quad \sum_{v \in V} x_v$$

subject to

$$x_u + x_v \geq 1 \quad \text{for every } \{u, v\} \in E$$

$$x_u \in \{0, 1\} \quad \text{for all } u \in V$$

The constraints “$x_u \in \{0, 1\}$” are called integrality constraints. They require that the variables be either 0 or 1, and they make the ILP difficult to solve.
# Declarations
set V;
set E within V cross V;
var x {v in V} binary;  # integrality constraints.

# Objective Function
minimize cover_size: sum { v in V } x[v];

# Constraints
subject to covered {(u,v) in E}: x[u] + x[v] >= 1;

solve;

# Output
printf "The Vertex Cover:";
printf {u in V : x[u] >= 1}: "%d ", u;
end;
Many problems can be modeled as linear programs (LPs).

If you can write your problem as an LP, you can use existing, highly optimized solvers to give polynomial time algorithms to solve them.

It seems even more problems can be written as integer linear programs (ILP).

If you write your problem as an ILP, you won’t have a polynomial-time algorithm, but you may be able to use optimized packages to solve it.
CMSC 451: The classes P and NP

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 8.3 of *Algorithm Design* by Kleinberg & Tardos.
• We’ve seen algorithms for lots of problems, and the goal was always to design an algorithm that ran in polynomial time.

• Sometimes we’ve claimed a problem is NP-hard as evidence that no such algorithm exists.

• Now, we’ll formally say what that means.
Decision Problems:

- Usually, we’ve considered optimization problems: given some input instance, output some answer that maximizes or minimizes a particular objective function.

- Most of computational complexity deals with a seemingly simpler type of problem: the decision problem.

- A decision problem just asks for a yes or a no.

- We phrased Circulation with Demands as a decision problem.
Recall this problem from a few weeks ago:

<table>
<thead>
<tr>
<th>Weighted Interval Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a set of intervals ( { I_i } ) each with a nonnegative weight ( w_i ), find a subset of intervals ( S ) such that ( \sum_{i \in S} w_i ) is maximized.</td>
</tr>
</tbody>
</table>

We can change this into a decision problem by asking:

<table>
<thead>
<tr>
<th>Weighted Interval Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a set of intervals ( { I_i } ) each with a nonnegative weight ( w_i ), is there a subset of intervals ( S ) such that ( \sum_{i \in S} w_i ) is greater than ( C ).</td>
</tr>
</tbody>
</table>
Decision is no harder than Optimization

The decision version of a problem is easier than (or the same as) the optimization version.

Why, for example, is this true of Weighted Interval Scheduling?
The decision version of a problem is easier than (or the same as) the optimization version.

Why, for example, is this true of Weighted Interval Scheduling?

- If you could solve the optimization version and got a solution of value $M$, then you could just check to see if $M > C$.
- If you can solve the optimization problem, you can solve the decision problem.
- If the decision problem is hard, then so is the optimization version.
We can often also go from decision to optimization:

**Independent Set**
Given graph $G$ and a number $k$, does $G$ contain a set of at least $k$ independent vertices?

How can we find the largest $k$ for which $G$ contains an independent set of size $k$?
We can often also go from decision to optimization:

**Independent Set**

Given graph $G$ and a number $k$, does $G$ contain a set of at least $k$ independent vertices?

How can we find the largest $k$ for which $G$ contains an independent set of size $k$?

Try every possible value of $k$ — there are only $n$ of them. Or even better use binary search.
We can **encode** an instance of a decision problem as a string.

**Example.** The encoding of a *Weighted Interval Set* instance with 3 intervals might be:

\[ s_1, e_1, w_1; s_2, e_2, w_2; s_3, e_3, w_3; \text{;} C \]

More explicitly,

\[ 1, 10, 5; 3, 7, 20; 12, 15, 1; \text{;} 10 \]

How do we “know” intuitively that all of the problems we’ve considered so far can be encoded as a single string?
We can **encode** an instance of a decision problem as a string.

**Example.** The encoding of a *Weighted Interval Set* instance with 3 intervals might be:

\[
s_1, e_1, w_1; s_2, e_2, w_2; s_3, e_3, w_3; ; C
\]

More explicitly,

\[
1,10,5;3,7,20;12,15,1; ;10
\]

How do we “know” intuitively that all of the problems we’ve considered so far can be encoded as a single string?

**Because we can represent them in RAM as a string of bits!**
Decision Problems and Languages

A decision problem $X$ is really just sets of strings:

| String                                                | $\in X$?
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,10,5;3,7,20;12,15,1;;10</td>
<td>Yes</td>
</tr>
<tr>
<td>1,10,5;3,7,20;12,15,1;;100</td>
<td>No</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Def.** A language is a set of strings.
(Analogy: English is the set of valid English words.)

Hence, any decision problem is equivalent to deciding membership in some language.

We talk about “decision problems” and “languages” pretty much interchangeably.
Computational complexity primarily deals with decision problems.

A decision problem is no harder than the corresponding optimization problem.

A decision problem can be thought of as a set of the strings that encode “yes” instances.

Such sets are called languages.

How can we say a decision problem is hard?
A Model of Computation

Ultimately, we want to say that “a computer can’t recognize some language efficiently.”

To do that, we have to decide what we mean by a computer.

We will mean a Turing Machine.

**Church-Turing Thesis**
Everything that is efficiently computable is efficiently computable on a Turing Machine.
Turing Machine

At each time step, the TM:
• reads the symbol at the current position
• Depending on that symbol, and the current state of the TM, it:
  - Writes a new symbol $x$
  - Moves Left or Right
  - Changes to a new state $s$
The class $\mathbf{P}$

$\mathbf{P}$ is the set of languages whose memberships are decidable by a Turing Machine that makes a polynomial number of steps.

By the Church-Turing thesis, this is the “same” as:

$\mathbf{P}$ is the set of decision problems that can be decided by a computer in a polynomial time.

From now on, you can just think of your normal computer as a Turing Machine — and we won’t worry too much about that formalism.
The Class NP

Now that we have a different (more formal) view of P, we will define another class of problems called NP.

We need some new ideas.
Recall the independent set problem (decision version):

**Independent Set**
Given a graph $G$, is there set $S$ of size $\geq k$ such that no two nodes in $S$ are connected by an edge?

Finding the set $S$ is hard (we will see).

But if I give you a set $S^*$, checking whether $S^*$ is the answer is easy: check that $|S| \geq k$ and no edges go between 2 nodes in $S^*$.

$S^*$ acts as a certificate that $\langle G, k \rangle$ is a yes instance of Independent Set.
Efficient Certification

**Def.** An algorithm $B$ is an efficient certifier for problem $X$ if:

1. $B$ is a polynomial time algorithm that takes two input strings $I$ (instance of $X$) and $C$ (a certificate).
2. $B$ outputs either yes or no.
3. There is a polynomial $p(n)$ such that for every string $I$:

$$I \in X \text{ if and only if there exists string } C \text{ of length } \leq p(|I|) \text{ such that } B(I, C) = yes.$$

$B$ is an algorithm that can decide whether an instance $I$ is a yes instance if it is given some “help” in the form of a polynomially long certificate.
Certifiers and Brute Force

Let’s say you had an efficient certifier $B$ for the Independent Set problem.

How could you use it in a brute force algorithm on instance $I$?
Certifiers and Brute Force

Let’s say you had an efficient certifier $B$ for the Independent Set problem.

How could you use it in a brute force algorithm on instance $I$?

Try every string $C$ of length $\leq p(|I|)$ and ask is $B(I, C) = yes$?
The class NP

**NP** is the set of languages for which there exists an efficient certifier.
The class NP

**NP** is the set of languages for which there exists an efficient certifier.

**P** is the set of languages for which there exists an efficient certifier that ignores the certificate.

That’s the difference:

A problem is in **P** if we can decided them in polynomial time. It is in **NP** if we can decide them in polynomial time, if we are given the right certificate.
Do we have to find the certificates?

- User provides instance as usual
- Certificate is magically guessed
**Theorem**

$P \subseteq NP$

**Proof.** Suppose $X \in P$. Then there is a polynomial-time algorithm $A$ for $X$.

To show that $X \in NP$, we need to design an efficient certifier $B(I, C)$.

Just take $B(I, C) = A(I)$. □

Every problem with a polynomial time algorithm is in $NP$. 
The big question:

\[ P \neq NP? \]

We know \( P \subseteq NP \). So the question is:

Is there some problem in \( NP \) that is not in \( P \)?

Seems like the power of the certificate would help a lot.
But no one knows...
CMSC 451: Reductions & NP-completeness

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Section 8.1 of Algorithm Design by Kleinberg & Tardos.
We want prove some problems are computationally difficult.

As a first step, we settle for relative judgements:

Problem $X$ is at least as hard as problem $Y$

To prove such a statement, we reduce problem $Y$ to problem $X$:

*If you had a black box that can solve instances of problem $X$, how can you solve any instance of $Y$ using polynomial number of steps, plus a polynomial number of calls to the black box that solves $X$?*
Polynomial Reductions

• If problem $Y$ can be reduced to problem $X$, we denote this by $Y \leq_p X$.

• This means “$Y$ is polynomial-time reducible to $X$.”

• It also means that $X$ is at least as hard as $Y$ because if you can solve $X$, you can solve $Y$.

• Note: We reduce to the problem we want to show is the harder problem.
Suppose:

- $Y \leq_P X$, and
- there is a polynomial time algorithm for $X$.

Then, there is a polynomial time algorithm for $Y$.

Why?
Suppose:

- $Y \leq_P X$, and

- there is a polynomial time algorithm for $X$.

Then, there is a polynomial time algorithm for $Y$.

Why? Because polynomials compose.
Examples of Reductions:

- **Max Bipartite Matching** $\leq_P$ **Max Network Flow**.

- **Image Segmentation** $\leq_P$ **Min-Cut**.

- **Survey Design** $\leq_P$ **Max Network Flow**.

- **Disjoint Paths** $\leq_P$ **Max Network Flow**.
Reductions for Hardness

**Theorem**

If \( Y \leq_P X \) and \( Y \) cannot be solved in polynomial time, then \( X \) cannot be solved in polynomial time.

Why? If we *could* solve \( X \) in polynomial time, then we’d be able to solve \( Y \) in polynomial time using the reduction, contradicting the assumption.

So: If we could find one hard problem \( Y \), we could prove that another problem \( X \) is hard by reducing \( Y \) to \( X \).
**Def.** A *vertex cover* of a graph is a set $S$ of nodes such that every edge has at least one endpoint in $S$.

In other words, we try to “cover” each of the edges by choosing at least one of its vertices.

**Vertex Cover**

Given a graph $G$ and a number $k$, does $G$ contain a vertex cover of size at most $k$. 
Independent Set to Vertex Cover

**Independent Set**
Given graph $G$ and a number $k$, does $G$ contain a set of at least $k$ independent vertices?

**Vertex Cover**
Given a graph $G$ and a number $k$, does $G$ contain a vertex cover of size at most $k$?

**Can we reduce independent set to vertex cover?**
Relation btw Vertex Cover and Indep. Set

**Theorem**

If $G = (V, E)$ is a graph, then $S$ is an independent set $\iff V - S$ is a vertex cover.

**Proof.** $\implies$ Suppose $S$ is an independent set, and let $e = (u, v)$ be some edge. Only one of $u, v$ can be in $S$. Hence, at least one of $u, v$ is in $V - S$. So, $V - S$ is a vertex cover.

$\impliedby$ Suppose $V - S$ is a vertex cover, and let $u, v \in S$. There can’t be an edge between $u$ and $v$ (otherwise, that edge wouldn’t be covered in $V - S$). So, $S$ is an independent set. $\square$
Independent Set $\leq_P$ Vertex Cover

To show this, we change any instance of Independent Set into an instance of Vertex Cover:

- Given an instance of Independent Set $\langle G, k \rangle$,
- We ask our Vertex Cover black box if there is a vertex cover $V - S$ of size $\leq |V| - k$.

By our previous theorem, $S$ is an independent set iff $V - S$ is a vertex cover. If the Vertex Cover black box said:

**yes:** then $S$ must be an independent set of size $\geq k$.

**no:** then there is no vertex cover $V - S$ of size $\leq |V| - k$, hence there is no independent set of size $\geq k$. 
Actually, we also have:

**Vertex Cover \( \leq_p \) Independent Set**

*Proof.* To decide if \( G \) has an vertex cover of size \( k \), we ask if it has an independent set of size \( n - k \). \( \square \)

So: **Vertex Cover and Independent Set** are equivalently difficult.
**Def.** We say $X$ is NP-complete if:

- $X \in \text{NP}$
- for all $Y \in \text{NP}$, $Y \leq_P X$.

If these hold, then $X$ can be used to solve every problem in NP.

Therefore, $X$ is definitely at least as hard as every problem in NP.
NP-completeness and P=NP

**Theorem**

*If X is NP-complete, then X is solvable in polynomial time if and only if \( P = NP \).*

**Proof.** If \( P = NP \), then \( X \) can be solved in polytime.

Suppose \( X \) is solvable in polytime, and let \( Y \) be any problem in \( NP \). We can solve \( Y \) in polynomial time: reduce it to \( X \).

Therefore, every problem in \( NP \) has a polytime algorithm and \( P = NP \).
Theorem

If $Y$ is NP-complete, and

1. $X$ is in NP
2. $Y \leq_P X$

then $X$ is NP-complete.

In other words, we can prove a new problem is NP-complete by reducing some other NP-complete problem to it.

Proof. Let $Z$ be any problem in $\textbf{NP}$. Since $Y$ is NP-complete, $Z \leq_P Y$. By assumption, $Y \leq_P X$. Therefore: $Z \leq_P Y \leq_P X$. □
Some First NP-complete problem

We need to find some first NP-complete problem.

Finding the first NP-complete problem was the result of the Cook-Levin theorem.

We’ll deal with this later. For now, trust me that:

• Independent Set is a packing problem and is NP-complete.
• Vertex Cover is a covering problem and is NP-complete.
Another very general and useful covering problem:

**Set Cover**

Given a set $U$ of elements and a collection $S_1, \ldots, S_m$ of subsets of $U$, is there a collection of at most $k$ of these sets whose union equals $U$?

We will show that

**Set Cover** $\in$ **NP**

**Vertex Cover** $\leq_P$ **Set Cover**

And therefore that **Set Cover** is NP-complete.
Set Cover, Figure
Set Cover, Figure
**Thm.** Vertex Cover $\leq_P$ Set Cover

**Proof.** Let $G = (V, E)$ and $k$ be an instance of VERTEX COVER. Create an instance of SET COVER:

- $U = E$
- Create a $S_u$ for each $u \in V$, where $S_u$ contains the edges adjacent to $u$.

$U$ can be covered by $\leq k$ sets iff $G$ has a vertex cover of size $\leq k$.

Why? If $k$ sets $S_{u_1}, \ldots, S_{u_k}$ cover $U$ then every edge is adjacent to at least one of the vertices $u_1, \ldots, u_k$, yielding a vertex cover of size $k$.

If $u_1, \ldots, u_k$ is a vertex cover, then sets $S_{u_1}, \ldots, S_{u_k}$ cover $U$. □
Last Step:

We still have to show that Set Cover is in $\textbf{NP}$!

The certificate is a list of $k$ sets from the given collection.

We can check in polytime whether they cover all of $U$.

Since we have a certificate that can be checked in polynomial time, Set Cover is in $\textbf{NP}$. 
You can prove a problem is NP-complete by reducing a known NP-complete problem to it.

We know the following problems are NP-complete:

- Vertex Cover
- Independent Set
- Set Cover

**Warning:** You should reduce the *known* NP-complete problem to the problem you are interested in. (You *will* mistakenly do this backwards sometimes.)
CMSC 451: SAT, Coloring, Hamiltonian Cycle, TSP

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Sects. 8.2, 8.7, 8.5 of Algorithm Design by Kleinberg & Tardos.
Boolean Formulas:

Variables: $x_1, x_2, x_3$ (can be either true or false)

Terms: $t_1, t_2, \ldots, t_\ell$: $t_j$ is either $x_i$ or $\bar{x}_i$ (meaning either $x_i$ or not $x_i$).

Clauses: $t_1 \lor t_2 \lor \cdots \lor t_\ell$ ($\lor$ stands for “OR”)

A clause is true if any term in it is true.

Example 1: $(x_1 \lor \bar{x}_2), (\bar{x}_1 \lor \bar{x}_3), (x_2 \lor \bar{v}_3)$

Example 2: $(x_1 \lor x_2 \lor \bar{x}_3), (\bar{x}_2 \lor x_1)$
Def. A truth assignment is a choice of true or false for each variable, ie, a function $v : X \rightarrow \{\text{true}, \text{false}\}$.

Def. A CNF formula is a conjunction of clauses:

$$C_1 \land C_2, \land \cdots \land C_k$$

Example: $(x_1 \lor \bar{x}_2) \land (\bar{x}_1 \lor x_3) \land (x_2 \lor \bar{v}_3)$

Def. A truth assignment is a satisfying assignment for such a formula if it makes every clause true.
**Satisfiability (SAT)**

Given a set of clauses $C_1, \ldots, C_k$ over variables $X = \{x_1, \ldots, x_n\}$ is there a satisfying assignment?

**Satisfiability (3-SAT)**

Given a set of clauses $C_1, \ldots, C_k$, each of length 3, over variables $X = \{x_1, \ldots, x_n\}$ is there a satisfying assignment?
Cook-Levin Theorem

**Theorem (Cook-Levin)**

3-SAT is NP-complete.

Proven in early 1970s by Cook. Slightly different proof by Levin independently.

**Idea of the proof:** encode the workings of a Nondeterministic Turing machine for an instance $I$ of problem $X \in \textbf{NP}$ as a SAT formula so that the formula is satisfiable if and only if the nondeterministic Turing machine would accept instance $I$.

We won’t have time to prove this, but it gives us our first hard problem.
Reducing 3-SAT to Independent Set

**Thm.** 3-SAT \( \leq_P \) Independent Set

**Proof.** Suppose we have an algorithm to solve Independent Set, how can we use it to solve 3-SAT?

To solve 3-SAT:

- you have to choose a term from each clause to set to **true**,  
- but you can’t set both \( x_i \) and \( \bar{x}_i \) to **true**.

How do we do the reduction?
$3$-SAT $\leq_{P}$ Independent Set

$$(x_1 \lor x_2 \lor \overline{x_3}) \land (x_2 \lor x_3 \lor \overline{x_4}) \land (x_1 \lor \overline{x_2} \lor x_4)$$
This graph has an independent set of size $k$ iff the formula is satisfiable.

Proof. $\implies$ If the formula is satisfiable, there is at least one true literal in each clause. Let $S$ be a set of one such true literal from each clause. $|S| = k$ and no two nodes in $S$ are connected by an edge.

$\implies$ If the graph has an independent set $S$ of size $k$, we know that it has one node from each “clause triangle.” Set those terms to true. This is possible because no 2 are negations of each other. $\square$
Graph Coloring
Graph Coloring Problem

Given a graph $G$, can you color the nodes with $\leq k$ colors such that the endpoints of every edge are colored differently?

**Notation:** A $k$-coloring is a function $f : V \rightarrow \{1, \ldots, k\}$ such that for every edge $\{u, v\}$ we have $f(u) \neq f(v)$.

If such a function exists for a given graph $G$, then $G$ is $k$-colorable.
How can we test if a graph has a 2-coloring?
Special case of $k = 2$

How can we test if a graph has a 2-coloring?

Check if the graph is bipartite.

Unfortunately, for $k \geq 3$, the problem is NP-complete.

**Theorem**

3-Coloring is NP-complete.
Graph Coloring is NP-complete

3-Coloring $\in \textbf{NP}$: A valid coloring gives a certificate.

We will show that:

$$3\text{-SAT} \leq_P 3\text{-Coloring}$$

Let $x_1, \ldots, x_n, C_1, \ldots, C_k$ be an instance of 3-SAT.

We show how to use 3-Coloring to solve it.
Reduction from 3-SAT

We construct a graph $G$ that will be 3-colorable iff the 3-SAT instance is satisfiable.

For every variable $x_i$, create 2 nodes in $G$, one for $x_i$ and one for $\bar{x}_i$. Connect these nodes by an edge:

Create 3 *special nodes* $T$, $F$, and $B$, joined in a triangle:
Connecting them up

Connect every variable node to B:
Properties:

- Each of $x_i$ and $\overline{x}_i$ must get different colors
- Each must be different than the color of B.
- B, T, and F must get different colors.

Hence, any 3-coloring of this graph defines a valid truth assignment!

Still have to constrain the truth assignments to satisfy the given clauses, however.
Connect Clause \((t_1, t_2, t_3)\) up like this:
Suppose Every Term Was False

What if every term in the clause was assigned the \texttt{false} color?
Connect Clause \((t_1, t_2, t_3)\) up like this:
Connect Clause \((t_1, t_2, t_3)\) up like this:
Connect Clause \((t_1, t_2, t_3)\) up like this:
Connect Clause \((t_1, t_2, t_3)\) up like this:
Suppose there is a 3-coloring.

We get a satisfying assignment by:

- Setting $x_i = \text{true}$ iff $v_i$ is colored the same as $T$

Let $C$ be any clause in the formula. At least 1 of its terms must be true, because if they were all false, we couldn't complete the coloring (as shown above).
Suppose there is a satisfying assignment.

We get a 3-coloring of $G$ by:

- Coloring $T$, $F$, $B$ arbitrarily with 3 different colors
- If $x_i = \text{true}$, color $v_i$ with the same color as $T$ and $\overline{v}_i$ with the color of $F$.
- If $x_i = \text{false}$, do the opposite.
- Extend this coloring into the clause gadgets.

Hence: the graph is 3-colorable iff the formula it is derived from is satisfiable.
General Proof Strategy for Proving Something is NP-complete:

1. Must show that $X \in \mathbf{NP}$. Do this by showing there is an certificate that can be efficiently checked.

2. Look at some problems that are known to be NP-complete (there are thousands), and choose one $Y$ that seems “similar” to your problem in some way.

3. Show that $Y \leq_P X$. 
One strategy for showing that \( Y \leq_P X \) often works:

1. Let \( I_Y \) be any instance of problem \( Y \).

2. Show how to construct an instance \( I_X \) of problem \( X \) in polynomial time such that:
   - If \( I_Y \in Y \), then \( I_X \in X \)
   - If \( I_X \in X \), then \( I_Y \in Y \)
Hamiltonian Cycle
Hamiltonian Cycle Problem

Hamiltonian Cycle

Given a directed graph $G$, is there a cycle that visits every vertex exactly once?

Such a cycle is called a Hamiltonian cycle.
Theorem

Hamiltonian Cycle is NP-complete.

Proof. First, HamCycle ∈ NP. Why?

Second, we show 3-SAT ≤ₚ Hamiltonian Cycle.

Suppose we have a black box to solve Hamiltonian Cycle, how do we solve 3-SAT?

In other words: how do we encode an instance $I$ of 3-SAT as a graph $G$ such that $I$ is satisfiable exactly when $G$ has a Hamiltonian cycle.

Consider an instance $I$ of 3-SAT, with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$. 
Reduction Idea (very high level):

- Create some graph structure (a “gadget”) that represents the variables
- And some graph structure that represents the clauses
- Hook them up in some way that encodes the formula
- Show that this graph has a Hamiltonian cycle iff the formula is satisfiable.
Gadget Representing the Variables

Direction we travel along this chain represents whether to set the variable to **true** or **false**.
Hooking in the Clauses

Add a new node for each clause:

$x_i$

Direction we travel along this chain represents whether to set the variable to true or false.

false → true
Connecting up the paths
Connecting up the paths

\[ x_1 \rightarrow C_1 \rightarrow C_2 \rightarrow C_k \rightarrow x_n \]
A Hamiltonian path encodes a truth assignment for the variables (depending on which direction each chain is traversed)

For there to be a Hamiltonian cycle, we have to visit every clause node

We can only visit a clause if we satisfy it (by setting one of its terms to true)

Hence, if there is a Hamiltonian cycle, there is a satisfying assignment
**Hamiltonian Path**: Does $G$ contain a path that visits every node exactly once?

How could you prove this problem is NP-complete?
Hamiltonian Path: Does $G$ contain a path that visits every node exactly once?

How could you prove this problem is NP-complete?

Reduce Hamiltonian Cycle to Hamiltonian Path.

Given instance of Hamiltonian Cycle $G$, choose an arbitrary node $v$ and split it into two nodes to get graph $G'$:

Now any Hamiltonian Path must start at $v'$ and end at $v''$. 
Hamiltonian Path

\[ G'' \text{ has a Hamiltonian Path} \iff G \text{ has a Hamiltonian Cycle.} \]

\[ \implies \text{ If } G'' \text{ has a Hamiltonian Path, then the same ordering of nodes (after we glue } v' \text{ and } v'' \text{ back together) is a Hamiltonian cycle in } G. \]

\[ \impliedby \text{ If } G \text{ has a Hamiltonian Cycle, then the same ordering of nodes is a Hamiltonian path of } G' \text{ if we split up } v \text{ into } v' \text{ and } v''. \]

Hence, Hamiltonian Path is NP-complete.
## Traveling Salesman Problem

Given $n$ cities, and distances $d(i, j)$ between each pair of cities, does there exist a path of length $\leq k$ that visits each city?

### Notes:

- We have a distance between every pair of cities.
- In this version, $d(i, j)$ doesn’t have to equal $d(j, i)$.
- And the distances don’t have to obey the triangle inequality ($d(i, j) \leq d(i, k) + d(k, j)$ for all $i, j, k$).
TSP large instance

- TSP visiting 24,978 (all) cities in Sweden.
- Solved by David Applegate, Robert Bixby, Vašek Chvátal, William Cook, and Keld Helsgaun
- [http://www.tsp.gatech.edu/sweden/index.html](http://www.tsp.gatech.edu/sweden/index.html)
- Lots more cool TSP at [http://www.tsp.gatech.edu/](http://www.tsp.gatech.edu/)
Thm. Traveling Salesman is NP-complete.

TSP seems a lot like Hamiltonian Cycle. We will show that

\[ \text{Hamiltonian Cycle} \leq_P \text{TSP} \]

To do that:

- Given: a graph \( G = (V, E) \) that we want to test for a Hamiltonian cycle,
- Create: an instance of TSP.
Creating a TSP instance

A TSP instance $D$ consists of $n$ cities, and $n(n - 1)$ distances.

**Cities** We have a city $c_i$ for every node $v_i$.

**Distances** Let $d(c_i, c_j) = \begin{cases} 1 & \text{if edge } (v_i, v_j) \in E \\ 2 & \text{otherwise} \end{cases}$
Theorem

\[ G \text{ has a Hamiltonian cycle } \iff D \text{ has a tour of length } \leq n. \]

Proof. If \( G \) has a Ham. Cycle, then this ordering of cities gives a tour of length \( \leq n \) in \( D \) (only distances of length 1 are used).

Suppose \( D \) has a tour of length \( \leq n \). The tour length is the sum of \( n \) terms, meaning each term must equal 1, and hence cities that are visited consecutively must be connected by an edge in \( G \). \( \square \)

Also, \( \text{TSP} \in \text{NP} \): a certificate is simply an ordering of the \( n \) cities.
Hence, TSP is NP-complete.

Even TSP restricted to the case when the $d(i,j)$ values come from actual distances on a map is NP-complete.
CMSC 451: More NP-completeness Results

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Sect. 8.5,8.7,8.9 of Algorithm Design by Kleinberg & Tardos.
Three-Dimensional Matching
Recall ‘2-d matching’:

**Given** sets $X$ and $Y$, each with $n$ elements, and a set $E$ of pairs $\{x, y\}$,

**Question:** is there a choice of pairs such that every element in $X \cup Y$ is paired with some other element?

Usually, we thought of *edges* instead of *pairs*: $\{x, y\}$, but they are really the same thing.
Three-Dimensional Matching

**Given:** Sets $X$, $Y$, $Z$, each of size $n$, and a set $T \subseteq X \times Y \times Z$ of order triplets.

**Question:** Is there a set of $n$ triplets in $T$ such that each element is contained in exactly one triplet?
Three-dimensional matching (aka 3DM) is NP-complete

Proof. 3DM is in NP: a collection of \( n \) sets that cover every element exactly once is a certificate that can be checked in polynomial time.

Reduction from 3-SAT. We show that:

\[
3\text{-SAT} \leq_P 3\text{DM}
\]

In other words, if we could solve 3DM, we could solve 3-SAT.
3-SAT instance: \( x_1, \ldots, x_n \) be \( n \) boolean variables, and \( C_1, \ldots, C_k \) clauses.

We create a gadget for each variable \( x_i \):

\[
A_i = \{a_{i1}, \ldots, a_{i,2k}\} \quad \text{core}
\]
\[
B_i = \{a_{i1}, \ldots, a_{i,2k}\} \quad \text{tips}
\]
\[
t_{ij} = (a_{ij}, a_{i,j+1}, b_{ij}) \quad \text{TF triples}
\]
Gadget Encodes True and False
Gadget Encodes True and False
Gadget Encodes True and False
How “choice” is encoded

- We can only either use the even or odd “wings” of the gadget.

- In other words, if we use the even wings, we leave the odd tips uncovered (and vice versa).

- Leaving the odd tips free for gadget $i$ means setting $x_i$ to false.

- Leaving the odd tips free for gadget $i$ means setting $x_i$ to true.
Need to encode constraints between the tips that ensure we satisfy all the clauses.

We create a gadget for each clause $C_j = \{ t_1, t_2, t_3 \}$

$$P_j = \{ c_j, c'_j \} \quad \text{Clause core}$$

We will hook up these two clause core nodes with some tip nodes depending on whether the clause asks for a variable to be true or false.

See the next slide.
Clause Gadget Hookup

Add tuple \((c_1, c'_1, b_{i,2})\) if \(x_i\) in clause

Add tuple \((c_1, c'_1, b_{i,1})\) if \(\overline{x_i}\) in clause

\[
C_1 = x_1 \lor \overline{x_3} \lor \overline{x_5}
\]
Clause Gadgets

Since only clause tuples (brown) cover $c_j, c'_j$, we have to choose exactly one of them for every clause.

We can only choose a clause tuple $(c_j, c'_j, b_{ij})$ if we haven’t chosen a TF tuple that already covers $b_{ij}$.

Hence, we can satisfy (cover) the clause $(c_j, c'_j)$ with the term represented by $b_{ij}$ only if we “set” $x_i$ to the appropriate value.

That’s the basic idea. Two technical points left...
Need to cover all the tips:

Even if we satisfy all the clauses, we might have extra tips left over. We add a clean up gadget \((q_i, q'_i, b)\) for every tip \(b\).

Can we partition the sets?

\[
X = \{a_{ij} : j \text{ even}\} \cup \{c_j\} \cup \{q_i\} \\
Y = \{a_{ij} : j \text{ odd}\} \cup \{c'_j\} \cup \{q'_i\} \\
Z = \{b_{ij}\}
\]

Every set we defined uses 1 element from each of \(X, Y, Z\).
If there is a satisfying assignment,

We choose the odd / even wings depending on whether we set a variable to true or false. At least 1 free tip for a term will be available to use to cover each clause gadget. We then use the clean up gadgets to cover all the rest of the tips.

If there is a 3D matching,

We can set variable $x_i$ to true or false depending on whether it’s even or odd wings were chosen. Because $\{c_j, c'_j\}$ were covered, we must have correctly chosen one even/odd wing that will satisfy this clause.
Subset Sum
Subset Sum Problem

Given $n$ natural numbers $w_1, \ldots, w_n$ and a number $W$, is there a subset of $w_1, \ldots, w_n$ that adds up exactly to $W$?

We saw a $O(nW)$ dynamic programming algorithm for this problem earlier in the semester.

But this is pseudo-polynomial! Even problems with pseudo-polynomial algorithms can be $\textbf{NP}$-complete.

**Reason:** $W$ is actually exponential in the input size, $O(\log W)$. 
Subset Sum is **NP**-complete.

**Theorem**

*Subset Sum is NP-complete.*

**Proof.** (1) Subset Sum is in **NP**: a certificate is the set of numbers that add up to $W$.

(2) $3$-DM $\leq_P$ Subset Sum.

**Instance of 3-DM:** Let $X, Y, Z$ be sets of size $n$ and let $T \subseteq X \times Y \times Z$ be a set of tuples.

We encode this 3-DM instance into an instance of Subset Sum.
Encode each tuple \((x, y, z) \subseteq X \times Y \times Z\) as a bit vector:

Each tuple \(t \in T\) corresponds to a number

\[
w_t = d^{i-1} + d^{n+j-1} + d^{2n+k-1}\]

for some base \(d\).
For 3DM we want to choose a set of tuples that includes every element exactly once.

\[ t_1 \cup t_2 \] corresponds to \[ w_{t_1} + w_{t_2} \]:

\[
\begin{align*}
t_1 &= 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0 \\
t_2 &= 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0 \\
t_1 + t_2 &= 0\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 2\ 0\ 0\ 0\ 0
\end{align*}
\]
Goal: all ones

Set $W$ equal to the number represented by the all 1s vector:

$$W = \sum_{i=0}^{3n-1} d^i$$

What base $d$ should we use?

Want to avoid carries. Let $m$ be the number of tuples in $T$.

Set $d$ equal to $1 + m \implies$ Can’t have any carries.
Proof

If \( T \) contains a 3-dimensional matching,

Then \( t_1, \ldots, t_n \) then \( w_{t_1} + \cdots + w_{t_n} \) contains a 1 in every position and equals \( W \).

If \( w_{t_1} + \cdots + w_{t_k} = W \),

Then \( k = n \), and each of the \( 3n \) positions is covered by one 1 digit, and hence each element is covered by exactly 1 tuple.
If $W$ is bounded by a polynomial function of $n$, then we can solve Subset Sum in polynomial time: $O(nW)$. 

Polynomially bounded numbers
Other Complexity Classes
Suppose $B$ is an efficient certifier for an NP problem.

Problems in NP have yes-instances with efficient certifiers:

Instance $I$ is a yes instance $\iff$ there is a short certificate $C$ such that $B(I, C) = \text{yes}$.

Negation:

Instance $I$ is a no instance $\iff$ for all short $C$, we have $B(I, C) = \text{no}$.

I.e. we have short proofs for yes-instances, but not necessarily for no-instances.
Example

How would you convince me that \( G \) does \textbf{not} have an Hamiltonian cycle?
Recall that decision problems are really sets of strings.

For every decision problem $X$ there is a complementary problem $\bar{X}$:

$$I \in \bar{X} \iff I \notin X.$$  

That is, $\bar{X}$ contains those instances that $X$ does not.

Characterization of $\bar{X}$:

Instance $I \in \bar{X} \iff$ for all short certificates $C$, $B(I, C) = \text{no}$. 
**Def.** A problem $\overline{X}$ is in co-NP iff the complementary problem $X$ belongs to NP.

- These are the problems that have efficient “no” certificates.
- Does $\text{NP} = \text{co-NP}$? We don’t know.

---

**Theorem**

*If $\text{NP} \neq \text{co-NP}$, then $\text{P} \neq \text{NP}$.***

*Proof.* Contrapositive: $\text{P} = \text{NP} \implies \text{NP} = \text{co-NP}$. 

Since P is closed under complementation, if $\text{P} = \text{NP}$, then $\text{NP} = \text{co-NP}$. 
Consider the set: $\text{NP} \cap \text{co-NP}$.

These are the problems that have short “yes” proofs and short “no” proofs.

Any problem in $\text{P}$ is in both $\text{NP}$ and $\text{co-NP}$, so $\text{P} \subseteq \text{NP} \cap \text{co-NP}$.

Open Question: Does $\text{P} = \text{co-NP}$?
Summary of NP-complete problems

We’ve seen NP-completeness proofs for many problems:

• Independent Set
• Vertex Cover
• Set Cover
• 3-Dimensional matching
• Graph Coloring and 3-Coloring
• SAT and 3-SAT
• Hamiltonian Path and Cycle
• Traveling Salesman
• Subset Sum
CMSC 451: Approximation Algorithms

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park
If you like the **NP**-completeness topics, you’ll really like *CSMC 452*, taught next semester by Prof. Kruskal.

Topics may include:

- Finite Automata
- Context-free languages
- Turing machines
- Undecidability
- Computational complexity
Approximation Algorithms

- How do we deal with NP-complete (or other hard) problems?

- One option: heuristics

- But we’d like some guarantee: the answer we get should never be too far from the optimal.

- → Approximation Algorithms
Minimizing Makespan

Problem
Given \( n \) jobs of length \( t_1, \ldots, t_n \), and \( m \) machines \( M_1, \ldots, M_m \), schedule the jobs on the machines to minimize the makespan.

Def. The makespan is the total time to complete the jobs, running in parallel:

```
M_1
  \( t_1 \)
  \( t_2 \)

M_2
  \( t_3 \)

M_3
  \( t_4 \)
  \( t_5 \)
  \( t_6 \)

M_4
  \( t_7 \)
  \( t_8 \)
  \( t_9 \)

M_5
  \( t_7 \)
```
Minimizing Makespan is NP-complete (try to prove it!).

How would you solve it if you were asked to code it up?
Minimizing Makespan is NP-complete (try to prove it!).

How would you solve it if you were asked to code it up?

**Greedy Algorithm:**

- Order the items arbitrarily.

- Go down the list of items, scheduling item $t_i$ on the machine that has been used the least so far.
Greedy Example

\[
\begin{array}{cccc}
2 & 3 & 4 & 6 \\
M_1 & M_2 & M_3
\end{array}
\]
Greedy Example

3 4 6 2 2

M_1 M_2 M_3

2
Greedy Example
Greedy Example
Greedy Example
Greedy Example
Greedy Example

\[
\begin{align*}
M_1 & \quad 6 \quad 2 \\
M_2 & \quad 3 \quad 2 \\
M_3 & \quad 4 \\
\end{align*}
\]
The desired statement

Definitions:

- Let $G(I)$ be the makespan obtained by this greedy algorithm for instance $I$.
- Let $OPT(I)$ be the smallest possible makespan for instance $I$.

Goal: We want to prove something like:

For any instance $I$, $G(I) \leq \alpha OPT(I)$.

for some $\alpha$. 
Approximation Guarantee

Approximation Guarantee:

For any instance $I$, $G(I) \leq \alpha OPT(I)$.

Clearly, $\alpha \geq 1$ (for our minimization problem) because we can’t have a smaller makespan than the optimal.

Want $\alpha$ to be as small as possible.

For example, if $\alpha = 2$, we have the statement that the solution returned by our greedy algorithm is never more than twice as large as the optimal.
**Problem:** we don’t know the optimal, so how do we compare against it?

**Insight:** A lower bound on the optimal works almost as well:

- Suppose we know that $B(I) \leq OPT(I)$ for some function $B$.

- If we can prove $G(I) \leq \alpha B(I)$, then that immediately implies that $G(I) \leq \alpha OPT(I)$.
What are some lower bounds on the optimal makespan?
What are some lower bounds on the optimal makespan?

- Someone must do more work than the average:
  \[ OPT(I) \geq \frac{1}{m} \sum_j t_j, \]
  where \( m \) is the number of jobs.

- Someone must do the longest job: \( OPT(I) \geq \max_j t_j \).
Theorem

The greedy assignment produces a solution such that \( G(I) \leq 2OPT(I) \).

Proof. Definitions:

- Let \( M \) be the machine with the maximum load in the greedy solution.
- Let \( t_j \) be the last job assigned to \( M \).
- Let \( L_i \) be the load the greedy algorithm places on machine \( i \). (Note: \( G(I) = L_M \).)

When \( t_j \) was placed on \( M \), \( M \) had the lowest load. Why?

Every other machine had load \( \geq G(I) - t_j \). Why?
Proof, continued.

So every other machine had load $\geq G(I) - t_j$. That is:

$$m(G(I) - t_j) \leq \sum_k L_k$$

$$G(I) - t_j \leq \frac{1}{m} \sum_k L_k = \frac{1}{m} \sum_j t_j \leq OPT(I).$$

Just before we placed $t_j$, we had a load of $\leq$ the optimal on $M$.

We know too that $t_j \leq OPT(I)$. (second lower bound)

So: $G(I) = (G(I) - t_j) + t_j \leq OPT(I) + OPT(I) = 2OPT(I)$. 

To Summarize

- We split the solution up into two parts: (1) the contribution of the last item on $M$, and (2) the rest of the load on $M$.

- We used one lower bound to replace (1) by “$OPT(I)$”, and another lower bound to replace (2) by “$OPT(I)$”.

- There are examples where this greedy algorithm finds a solution nearly 2 times as bad as the optimal.
What if we sorted the jobs by decreasing length? (and apply our greedy algorithm.)

**Intuition:** Place the big jobs first, and then fit in the little ones wherever.

**Theorem**

*The Sorted-Greedy algorithm finds a solution of makespan $G(I)$ that is $\leq \frac{3}{2} \text{OPT}(I)$.*

Let $t_1, \ldots, t_n$ be the jobs, sorted by decreasing length.
Another Lower Bound

Wherever $t_{m+1}$ is placed, there will already be a job there of size $\geq t_{m+1}$.

Therefore, $\text{OPT}(I) \geq 2t_{m+1}$. 
Consider machine $M$ that has the largest load $L_M$ at the end of the algorithm.

Let $t_j$ be the last job placed on $M$.

$t_j \leq t_{m+1}$ because the first $m$ jobs all get placed on a different machine.

And $t_{m+1} \leq \frac{1}{2} OPT(I)$. By the new lower bound.

So: $L_M = (L_M - t_j) + t_j \leq OPT(I) + \frac{1}{2} OPT(I) = \frac{3}{2} OPT(I)$. 


Simple Approximation Algorithm for Vertex Cover
Vertex Cover

Given undirected graph $G$, find the smallest set $S$ of vertices so that every edge has an endpoint in $S$.

We saw this problem is $\text{NP}$-complete.

How could you try to solve it approximately?
Vertex Cover Approximation:

\[ S = \emptyset \]

While there are edges left:

  Let \((u,v)\) be any edge
  \[ S = S \cup \{u,v\} \quad \text{// Add both } u \text{ and } v \text{ to } S \]
  Remove every edge adjacent to either \(u\) or \(v\)

In other words: pick an edge, add its endpoints to the cover, and remove any edges that become covered.
2-Approximation

Theorem

This algorithm returns a vertex cover of at size $\leq$ twice the optimal.

The $m$ edges chosen during the algorithm.

None of these edge share an endpoint.

Therefore, the minimum vertex cover must be $\geq m$.

Our greedy cover is of size $2m$. 
Euclidean Traveling Salesman
Euclidean TSP

Given \( n \) cities, with distances \( d(u, v) \) between them (that satisfy the triangle inequality), find the order to visit them that minimizes the length of the route.

Euclidean TSP Approximation Algorithm:

1. Compute a minimum spanning tree \( T \) connecting the cities.
2. Visit the cities in order of a preorder traversal of \( T \).
Example
Example
Example
Notation:

- Let \( \text{cost}(A) \) be the total length of the edges in some set \( A \).
- Let \( A^* \) be the edges visited on the optimal tour.
- Let \( A \) be the edges visited on the tour found by our algorithm.

Theorem

\[
\text{cost}(A) \leq 2\text{cost}(A^*).
\]

(The algorithm gives a 2-approximation to the optimal TSP.)
Proof. The cost of a minimum spanning tree $T$ is less than the cost of the optimal tour: $\text{cost}(T) \leq \text{cost}(A^*)$. Why?

A full walk $W$ that “traces” the MST is of length $2\text{cost}(T)$ because every edge is crossed twice.

So: $\text{cost}(W) = 2\text{cost}(T) \leq 2\text{cost}(A^*)$.

$W$ isn’t a tour because it visits cities more than once. We can shortcut all but the first visit to a city. By the triangle inequality, this only reduces the cost of the tour.
Approximation algorithms for

- Minimizing makespan
- Vertex Cover
- Metric Traveling Salesman
CMSC 451: Local Search & Randomized Algorithms

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on §12.1,12.2,13.2 of Algorithm Design by Kleinberg & Tardos.
What if we have a hard problem but we can’t find an approximation algorithm for it?

**Local search** is a general class of algorithms that is often useful in practice.

Unfortunately, we can almost never prove that they will return a good solution.
Optimization problems

- A set $C$ of possible solutions
- A cost $\text{cost}(c)$ for each $c \in C$.
- We’re looking for a minimum/maximal cost $c \in C$. 
Energy Landscapes

Better

Energy landscapes
Gradient Descent

Take a series of "steps," improving the solution a little bit each time.

No guarantee you'll end up at the best solution...
Local Search:

- A set of “feasible solutions”: \( \mathcal{C} \).

- A \textit{neighbor} relation between some of these solutions: \( S \sim S' \) for some pairs \( S, S' \in \mathcal{C} \).

- \( \mathcal{N}(S) = \{S' : S \sim S'\} \): the \textit{neighbors} of solution \( S \).
Local Search Algorithm Schema:

1. Define a set of feasible solutions $\mathcal{C}$.

2. Define a neighbor relation $\sim$ on these sets.

3. Let $S_0$ be some feasible solution.

4. Let $S = S_0$.

5. Repeatedly choose some $S' \in \mathcal{N}(S)$, and let $S = S'$. 
Example: Vertex Cover

Find the minimum size vertex cover for graph $G$.

Define a state $S$ as a set of vertices that is a vertex cover.

Two states are neighbors if they differ by adding or deleting a vertex.

Algorithm:

While there is a neighbor $S'$ of $S$ with lower cost, let the new $S$ be the lowest cost neighbor.
Empty Graph:
Empty Graph:
Gradient Descent: Vertex Cover

Empty Graph:
Gradient Descent: Vertex Cover

Empty Graph:
Gradient Descent: Vertex Cover 2
If the center node is removed, then we have to stop.
Physical systems also typically have an “energy.”

The Gibbs-Boltzmann function says the probability of a system of being in a state of energy $E$ is:

$$e^{-E/(kT)}$$

where $T > 0$ is the temperature of the system.
Gibbs-Boltzmann
Metropolis Algorithm:

S = an initial solution
While not done:
    Choose $S' \in \mathcal{N}(S)$
    If $\text{cost}(S') \leq \text{cost}(S)$:
        Set $S = S'$
    Else:
        Delta = $\text{cost}(S') - \text{cost}(S)$
        With probability $\exp(-\text{Delta} / (kT))$:
            Let $S = S'$
    EndIf
EndWhile
Simulated Annealing

High values of \( T \) means you jump around a lot.

Low values of \( T \) mean you never increase the cost.

**Simulated Annealing: Idea:** Start with high \( T \) and reduce it as time progresses.

Some **cooling schedule** determines how to change \( T \).

Lot of work into finding cooling schedules that work in practice...
Simulated annealing used all the time in practice

No guarantees, but often gets good solutions

(Often does not, too, though.)
Randomized Algorithms
Randomized Algorithms

- Allow our algorithms to flip some random coins to make their choices.

- May require that the optimum solution is found with expected good runtime.

- Or may require that we always run in polynomial time, and we find the optimum solution with high probability.

- Often run in expectation faster than
Quicksort

You’ve probably seen probabilistic algorithms of the first type: quicksort.

When the list of numbers is already sorted, a naive deterministic algorithm performs very bad \(O(n^2)\).

A solution to this: randomly permute the input numbers.

Then the chance that you are in a “bad” case is small.
Global Minimum Cut

Given an undirected graph $G$ find a partition of the nodes of $G$ into two non-empty sets $A$ and $B$ such that the number of edges that have 1 endpoint in $A$ and 1 endpoint in $B$ is minimized.

Like minimum cut, but in an undirected graph, and we don’t specify $s$ and $t$. 
• Let $s$ be some node in $G$.

• In the global minimum cut, $s$ must be separated from something.

• Try all $n - 1$ choices for that something as $t$, and use network flow to compute the minimum $s - t$ cut.

• (Replace each undirected edge by two, anti-parallel, directed edges.)

• Cost is $n - 1$ network flow computations.
Contraction

Contraction step: choose a edge and merge its endpoints.
Contraction Algorithm:

While G contains more than 2 nodes:
Choose an edge e uniformly at random
Contract e, replacing its endpoints
with a new node w

Each new node is really a supernode that “contains” a number of original nodes.

Once we have a graph with only 2 supernodes, the supernodes define the cut.
Let $F$ be a global minimum cut.

Suppose $|F| = k$.

Every node in $G$ must have degree $\geq k$. Why?

Therefore, $|E| \geq \frac{1}{2}kn$.

The chance that we contract an edge in $F$ in the first step is at most:

$$\frac{k}{\frac{1}{2}kn} = \frac{2}{n}$$
After $j$ contractions there are $n - j$ supernodes.

Each super node has degree $\geq k$. Why?

There are at least $\frac{1}{2} k(n - j)$ edges, and the probability that we choose one from $F$ to contract is:

$$
\frac{k}{\frac{1}{2} k(n - j)} = \frac{2}{n - j}.
$$
The contraction algorithm stops after \( n - 2 \) iterations.

It will return the global minimum cut if none of the \( n - 2 \) contractions picked one of the edges in \( F \).

**Def.** \( \mathcal{E}_i = \text{even that an edge of } F \text{ was not contracted in step } i \).

- \( \Pr[\mathcal{E}_1] \geq 1 - \frac{2}{n} \)
- \( \Pr[\mathcal{E}_{j+1} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cap \cdots \cap \mathcal{E}_j] \geq 1 - \frac{2}{n-j} \)

**Probability of Success:** \( \Pr[\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}] \)
Theorem

The probability that the contraction algorithm returns the minimum cut is \( \geq \frac{1}{\binom{n}{2}} \).

\[
\begin{align*}
\Pr[\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}] & = \Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \cdots \Pr[\mathcal{E}_{j+1} \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_j] \cdots \\
& \geq \left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \cdots \left(1 - \frac{2}{n-j}\right) \cdots \left(1 - \frac{2}{3}\right) \\
& = \left(\frac{n-2}{n}\right) \cdots \left(\frac{n-3}{n-1}\right) \cdots \left(\frac{1}{3}\right) \\
& = \frac{2}{n(n-1)} = \binom{n}{2}^{-1} \quad \Box
\end{align*}
\]
Repeating the contraction algorithm

Repeat the algorithm \( \binom{n}{2} \ln n \) times.

The probability that we fail to find the global minimum cut every time is:

\[
\left(1 - \frac{1}{\binom{n}{2}}\right)^{\binom{n}{2} \ln n} \leq \frac{1}{n}.
\]
Summary

- Local Search often simple and works well in practice, despite it being hard to prove anything about.

- Randomization often yields simpler, faster algorithms.
Class Summary

**Basic Graph Algorithms:** BFS, DFS, TreeGrowing, testing bipartiteness, topological sort, stable marriage

**Greedy Methods:** exchange argument, stays-ahead argument, interval scheduling, minimizing lateness, optimal caching, shortest path, MST, interval partitioning

**Divide and Conquer:** counting inversions, closest pair of points, integer multiplication

**Dynamic Programming:** Weighted interval scheduling, memoization, subset sum, knapsack, RNA folding, sequence alignment, shortest paths

**Network Flow:** Ford-Fulkerson, Max-flow=Min-cut, Image segmentation, edge disjoint paths, circulation with demands, bipartite matching, linear programming

**NP-completeness:** NP, 3-SAT, Vertex Cover, Independent Set, Graph Coloring, 3-dimensional matching, Set Cover, Hamiltonian Cycle, TSP, Subset Sum, co-NP

**Other:** Approximation algorithms (vertex cover, TSP, makespan), local search (simulated annealing), randomized algorithms