Recap: Stable Matching Problem

- Definition of Stable Matching Problem
- Gale-Shapley Algorithm
  - Unengaged men propose to the top remaining woman $w$ on their preference list
  - The proposal is (temporarily) accepted if the woman $w$ is currently unengaged or if the proposer $m$ is preferred to current fiancé $m'$
- Analysis of Gale-Shapley Algorithm
  - Proof of correctness
    - Everyone is matched when algorithm terminates
    - Gale-Shapley Matching is stable
  - Implementation + Running Time Analysis:
    - Runs in at most $O(n^2)$ steps
- Implies Stable Matching Always exists
  - (Contrast with Stable-Roommate Problem)
- Gale-Shapley Matching is Optimal for Men
Q. For a given problem instance, there may be several stable matchings. Do all executions of Gale-Shapley yield the same stable matching? If so, which one?

Def. Man m is a valid partner of woman w if there exists some stable matching in which they are matched.

Man-optimal assignment. Each man receives best valid partner.

Claim. All executions of GS yield man-optimal assignment, which is a stable matching!

- No reason a priori to believe that man-optimal assignment is perfect, let alone stable.
- Simultaneously best for each and every man.
Claim. GS matching $S^*$ is man-optimal.

Pf. (by contradiction)

- Suppose some man is paired with someone other than best partner. Men propose in decreasing order of preference $\Rightarrow$ some man is rejected by valid partner.

- Let $Y$ be first such man, and let $A$ be first valid woman that rejects him.

- Let $S$ be a stable matching where $A$ and $Y$ are matched.

- When $Y$ is rejected, $A$ forms (or reaffirms) engagement with a man, say $Z$, whom she prefers to $Y$.

- Let $B$ be $Z$'s partner in $S$.

- $Z$ not rejected by any valid partner at the point when $Y$ is rejected by $A$. Thus, $Z$ prefers $A$ to $B$.

- But $A$ prefers $Z$ to $Y$.

- Thus $A-Z$ is unstable in $S$. □
Stable Matching Summary

**Stable matching problem.** Given preference profiles of \( n \) men and \( n \) women, find a **stable** matching.

\[ \text{no man and woman prefer to be with each other than assigned partner} \]

**Gale-Shapley algorithm.** Finds a stable matching in \( O(n^2) \) time.

**Man-optimality.** In version of GS where men propose, each man receives best valid partner.

\[ \text{w is a valid partner of m if there exist some stable matching where m and w are paired} \]

**Q.** Does man-optimality come at the expense of the women?
**Woman Pessimality**

**Woman-pessimal assignment.** Each woman receives worst valid partner.

**Claim.** GS finds **woman-pessimal** stable matching $S^*$.  

**Pf.**
- Suppose $A-Z$ matched in $S^*$, but $Z$ is not worst valid partner for $A$.
- There exists stable matching $S$ in which $A$ is paired with a man, say $Y$, whom she likes less than $Z$.
- Let $B$ be $Z$'s partner in $S$.
- $Z$ prefers $A$ to $B$.  \(\leftarrow\) **man-optimality**
- Thus, $A-Z$ is an unstable in $S$.  \(\blacksquare\)
Extensions: Matching Residents to Hospitals

Ex: Men \approx hospitals, Women \approx med school residents.

Variant 1. Some participants declare others as unacceptable.

Variant 2. Unequal number of men and women.

Variant 3. Limited polygamy.

Gale-Shapley Algorithm Still Works. Minor modifications to code to handle variations!
Extensions: Matching Residents to Hospitals

**Ex:** Men ≈ hospitals, Women ≈ med school residents.

**Variant 1.** Some participants declare others as unacceptable.

**Variant 2.** Unequal number of men and women.

**Variant 3.** Limited polygamy.

**Def.** Matching $S$ unstable if there is a hospital $h$ and resident $r$ such that:

- $h$ and $r$ are acceptable to each other; and
- either $r$ is unmatched, or $r$ prefers $h$ to her assigned hospital; and
- either $h$ does not have all its places filled, or $h$ prefers $r$ to at least one of its assigned residents.
1.2 Five Representative Problems
Interval Scheduling

**Input.** Set of jobs with start times and finish times.

**Goal.** Find maximum cardinality subset of mutually compatible jobs. 

- Jobs don't overlap
Interval Scheduling

**Input.** Set of jobs with start times and finish times.

**Goal.** Find maximum cardinality subset of mutually compatible jobs.

Jobs don't overlap

**Greedy Choice.** Select job with earliest finish time and eliminate incompatible jobs.
Input. Set of jobs with start times and finish times.

Goal. Find maximum cardinality subset of mutually compatible jobs.

Greedy Choice. Select job with earliest finish time and eliminate incompatible jobs.
**Input.** Set of jobs with start times and finish times.

**Goal.** Find maximum cardinality subset of mutually compatible jobs.

**Greedy Choice.** Select job with earliest finish time and eliminate incompatible jobs.

[Diagram showing intervals of jobs and the selection process for optimal scheduling]
Input. Set of jobs with start times and finish times.

Goal. Find maximum cardinality subset of mutually compatible jobs.

Greedy Choice. Select job with earliest finish time and eliminate incompatible jobs.
Interval Scheduling

**Input.** Set of jobs with start times and finish times.

**Goal.** Find **maximum cardinality** subset of mutually compatible jobs.

Chapter 4: We will prove that this greedy algorithm always finds the optimal solution!
Weighted Interval Scheduling

**Input.** Set of jobs with start times, finish times, and weights.

**Goal.** Find maximum weight subset of mutually compatible jobs.

**Greedy Algorithm No Longer Works!**
Weighted Interval Scheduling

**Input.** Set of jobs with start times, finish times, and weights.

**Goal.** Find **maximum weight** subset of mutually compatible jobs.

**Greedy Algorithm No Longer Works!**
Weighted Interval Scheduling

**Input.** Set of jobs with start times, finish times, and weights.

**Goal.** Find *maximum weight* subset of mutually compatible jobs.

Problem can be solved using technique called Dynamic Programming.
Bipartite Matching

**Input.** Bipartite graph.

**Goal.** Find maximum cardinality matching.

Different from Stable Matching Problem! How?
Bipartite Matching

Input. Bipartite graph.
Goal. Find maximum cardinality matching.

Problem can be solved using Network Flow Algorithms
Independent Set

Input. Graph.
Goal. Find maximum cardinality independent set.

subset of nodes such that no two joined by an edge

1
2
3
4
5
6
7

Brute-Force Algorithm: Check every possible subset. Running Time: $\geq 2^n$ steps

NP-Complete: Unlikely that efficient algorithm exists!

Positive: Can easily check that there is an independent set of size $k$
Input. Graph with weight on each node.

Game. Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

Goal. Select a maximum weight subset of nodes.

Second player can guarantee 20, but not 25.
Competitive Facility Location

**Input.** Graph with weight on each node.

**Game.** Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

**Goal.** Select a maximum weight subset of nodes.

---

Second player can guarantee 20, but not 25.
Competitive Facility Location

**Input.** Graph with weight on each node.

**Game.** Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

**Goal.** Select a maximum weight subset of nodes.

Second player can guarantee 20, but not 25.
Competitive Facility Location

**Input.** Graph with weight on each node.

**Game.** Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

**Goal.** Select a maximum weight subset of nodes.

Second player can guarantee 20, but not 25.
Competitive Facility Location

**Input.** Graph with weight on each node.

**Game.** Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

**Goal.** Select a **maximum weight** subset of nodes.

Second player can guarantee 20, but not 25.
Competitive Facility Location

**Input.** Graph with weight on each node.

**Game.** Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

**Goal.** Select a maximum weight subset of nodes.

PSPACE-Complete: Even harder than NP-Complete!

No short proof that player can guarantee value B. (Unlike previous problem)

Second player can guarantee 20, but not 25.
Five Representative Problems

Variations on a theme: independent set.

Interval scheduling: $n \log n$ greedy algorithm.
Weighted interval scheduling: $n \log n$ dynamic programming algorithm.
Bipartite matching: $n^k$ max-flow based algorithm.
Independent set: NP-complete.
Competitive facility location: PSPACE-complete.
Chapter 2

Basics of Algorithm Analysis

Algorithm Design

JON KLEINBERG • ÉVA TARDOS

Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.
As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - Charles Babbage
Polynomial-Time

**Brute force.** For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
- Typically takes $2^N$ time or worse for inputs of size $N$.
- Unacceptable in practice.

**Desirable scaling property.** When the input size doubles, the algorithm should only slow down by some constant factor $C$.

There exists constants $c > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $c N^d$ steps.

**Def.** An algorithm is **poly-time** if the above scaling property holds.

$n!$ for stable matching with $n$ men and $n$ women

choose $C = 2^d$
Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size \( N \).
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size \( N \).
- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
**Worst-Case Polynomial-Time**

**Def.** An algorithm is *efficient* if its running time is polynomial.

**Justification:** *It really works in practice!*
- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

**Exceptions.**
- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

Simplex method
Unix grep
Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>30</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>50</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>100</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>1,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>10,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
2.2 Asymptotic Order of Growth
Asymptotic Order of Growth

Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.
- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$. 
Slight abuse of notation. \( T(n) = O(f(n)) \).

- Not transitive:
  - \( f(n) = 5n^3; \ g(n) = 3n^2 \)
  - \( f(n) = O(n^3) = g(n) \)
  - but \( f(n) \neq g(n) \).

- Better notation: \( T(n) \in O(f(n)) \).

**Meaningless statement.** Any comparison-based sorting algorithm requires at least \( O(n \log n) \) comparisons.

- Statement doesn't "type-check."
- Use \( \Omega \) for lower bounds.
Properties

Transitivity.
- If \( f \in O(g) \) and \( g \in O(h) \) then \( f \in O(h) \).
- If \( f \in \Omega(g) \) and \( g \in \Omega(h) \) then \( f \in \Omega(h) \).
- If \( f \in \Theta(g) \) and \( g \in \Theta(h) \) then \( f \in \Theta(h) \).

Additivity.
- If \( f \in O(h) \) and \( g \in O(h) \) then \( f + g \in O(h) \)
- If \( f \in \Omega(h) \) and \( g \in \Omega(h) \) then \( f + g \in \Omega(h) \).
- If \( f \in \Theta(h) \) and \( g \in \Theta(h) \) then \( f + g \in \Theta(h) \).

Proof of \( A1 \) (If \( f \in O(h) \) and \( g \in O(h) \) then \( f + g \in O(h) \) )
- \( f \in O(h) \) means that for some constants \( c_1, N_1 \) we have \( f(n) \leq c_1 \times h(n) \) for all \( n \geq N_1 \)
- \( g \in O(h) \) means that for some constants \( c_2, N_2 \) we have \( g(n) \leq c_2 \times h(n) \) for all \( n \geq N_2 \)
- Set \( c = c_1 + c_2 \) and \( N := \max\{N_1, N_2\} \) for all \( n \geq N := \max\{N_1, N_2\} \)
Asymptotic Bounds for Some Common Functions

**Polynomials.** \( a_0 + a_1 n + \ldots + a_d n^d \) is \( \Theta(n^d) \) if \( a_d > 0 \).

**Polynomial time.** Running time is \( O(n^d) \) for some constant \( d \) independent of the input size \( n \).

**Logarithms.** \( O(\log_a n) = O(\log_b n) \) for any constants \( a, b > 0 \).

**Logarithms.** For every \( x > 0 \), \( \log n = O(n^x) \).

**Exponentials.** For every \( r > 1 \) and every \( d > 0 \), \( n^d = O(r^n) \).

- can avoid specifying the base
- log grows slower than every polynomial (even if \( x=0.000000001 \))
- every exponential grows faster than every polynomial
2.4 A Survey of Common Running Times
Linear Time: $O(n)$

Linear time. Running time is proportional to input size.

**Computing the maximum.** Compute maximum of $n$ numbers $a_1, \ldots, a_n$.

```
max ← a_1
for i = 2 to n {
    if (a_i > max)
        max ← a_i
}
```
**Linear Time: O(n)**

**Merge.** Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.

$$i = 1, j = 1$$

while (both lists are nonempty) {

  if ($a_i \leq b_j$) append $a_i$ to output list and increment $i$  

  else append $b_j$ to output list and increment $j$

} append remainder of nonempty list to output list

**Claim.** Merging two lists of size $n$ takes $O(n)$ time.

**Pf.** After each comparison, the length of output list increases by 1.
O(n log n) Time

**O(n log n) time.** Arises in divide-and-conquer algorithms.

also referred to as linearithmic time

**Sorting.** Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

**Largest empty interval.** Given n time-stamps \(x_1, \ldots, x_n\) on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

**O(n log n) solution.** Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.
Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

\[
\text{min} \leftarrow (x_1 - x_2)^2 + (y_1 - y_2)^2
\]
\[
\text{for } i = 1 \text{ to } n \{ \\
\quad \text{for } j = i+1 \text{ to } n \{ \\
\quad\quad d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2 \\
\quad\quad \text{if } (d < \text{min}) \\
\quad\quad\quad \text{min} \leftarrow d \\
\quad } \\
\}
\]

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. 

see chapter 5
Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given $n$ sets $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```plaintext
foreach set $S_i$ {
    foreach other set $S_j$ {
        foreach element $p$ of $S_i$ {
            determine whether $p$ also belongs to $S_j$
        }
        if (no element of $S_i$ belongs to $S_j$)
            report that $S_i$ and $S_j$ are disjoint
    }
}
```
Polynomial Time: $O(n^k)$ Time

**Independent set of size $k$.** Given a graph, are there $k$ nodes such that no two are joined by an edge?

$k$ is a constant

**$O(n^k)$ solution.** Enumerate all subsets of $k$ nodes.

```
foreach subset $S$ of $k$ nodes {
    check whether $S$ in an independent set
    if (S is an independent set)
        report $S$ is an independent set
}
```

- Check whether $S$ is an independent set = $O(k^2)$.
- Number of $k$ element subsets =
  \[
  \binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}
  \]

poly-time for $k=17$, but not practical
Independent set. Given a graph, what is maximum size of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```plaintext
S* ← φ
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}
```
Min Heap Order: For each node v in the tree
\[ \text{Parent}(v).\text{Value} \leq v.\text{Value} \]

Max Heap Order: For each node v in the tree
\[ \text{Parent}(v).\text{Value} \geq v.\text{Value} \]
Heap Insertion

Min Heap Order: For each node $v$ in the tree

\[ \text{Parent}(v).\text{Value} \leq v.\text{Value} \]
Heap Insertion

Min Heap Order: For each node \( v \) in the tree

\[
\text{Parent}(v).\text{Value} \leq v.\text{Value}
\]
Heap Insertion

Min Heap Order: For each node v in the tree

\[
\text{Parent}(v).\text{Value} \leq v.\text{Value}
\]
Heap Insertion

Heap.Insert(3)

Min Heap Order: For each node v in the tree

\[ \text{Parent}(v).\text{Value} \leq v.\text{Value} \]
Heap Insertion

Min Heap Order: For each node $v$ in the tree

$$\text{Parent}(v).\text{Value} \leq v.\text{Value}$$

**Theorem 2.12 [KT]:** The procedure Heapify-up fixes the heap property and allows us to insert a new element into a heap of $n$ elements in $O(\log n)$ time.
Heap Extract Minimum

_heap_.ExtractMin()

Min Heap Order: For each node \( v \) in the tree

\[ \text{Parent}(v).\text{Value} \leq v.\text{Value} \]

**Theorem 2.13 [KT]:** The procedure Heapify-down fixes the heap property and allows us to delete an element in a heap of \( n \) elements in \( O(\log n) \) time.
Min Heap Order: For each node $v$ in the tree
\[ \text{Parent}(v).Value \leq v.\text{Value} \]

Theorem 2.13 [KT]: The procedure Heapify-down fixes the heap property and allows us to delete an element in a heap of $n$ elements in $O(\log n)$ time.
Min Heap Order: For each node $v$ in the tree

$\text{Parent}(v).\text{Value} \leq v.\text{Value}$

**Theorem 2.13 [KT]:** The procedure Heapify-down fixes the heap property and allows us to delete an element in a heap of $n$ elements in $O(\log n)$ time.
Heap Extract Minimum

Min Heap Order: For each node \( v \) in the tree

\[
\text{Parent}(v).\text{Value} \leq v.\text{Value}
\]

**Theorem 2.13 [KT]:** The procedure Heapify-down fixes the heap property and allows us to delete an element in a heap of \( n \) elements in \( O(\log n) \) time.
Heap Extract Minimum

Min Heap Order: For each node \( v \) in the tree
\[
\text{Parent}(v).\text{Value} \leq v.\text{Value}
\]

**Theorem 2.13 [KT]:** The procedure Heapify-down fixes the heap property and allows us to delete an element in a heap of \( n \) elements in \( O(\log n) \) time.
Heap Extract Minimum

Heap.ExtractMin()

Min Heap Order: For each node \( v \) in the tree
\[
\text{Parent}(v).\text{Value} \leq v.\text{Value}
\]

**Theorem 2.13 [KT]:** The procedure Heapify-down fixes the heap property and allows us to delete an element in a heap of \( n \) elements in \( O(\log n) \) time.
Heap Summary

Insert: $O(\log n)$
FindMin: $O(1)$
Delete: $O(\log n)$ time
ExtractMin: $O(\log n)$ time

Thought Question: $O(n \log n)$ time sorting algorithm using heaps?
Graphs
3.1 Basic Definitions and Applications
Undirected graph. $G = (V, E)$

- $V$ = nodes.
- $E$ = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|$, $m = |E|$.

$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$E = \{1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6\}$
$n = 8$
$m = 11$
Some **Graph Applications**

<table>
<thead>
<tr>
<th><strong>Graph</strong></th>
<th><strong>Nodes</strong></th>
<th><strong>Edges</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersections</td>
<td>highways</td>
</tr>
<tr>
<td>communication</td>
<td>computers</td>
<td>fiber optic cables</td>
</tr>
<tr>
<td>World Wide Web</td>
<td>web pages</td>
<td>hyperlinks</td>
</tr>
<tr>
<td>social</td>
<td>people</td>
<td>relationships</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey</td>
</tr>
<tr>
<td>software systems</td>
<td>functions</td>
<td>function calls</td>
</tr>
<tr>
<td>scheduling</td>
<td>tasks</td>
<td>precedence constraints</td>
</tr>
<tr>
<td>circuits</td>
<td>gates</td>
<td>wires</td>
</tr>
</tbody>
</table>
Web graph.
- **Node:** web page.
- **Edge:** hyperlink from one page to another.
9-11 Terrorist Network

Social network graph.

- **Node:** people.
- **Edge:** relationship between two people.

Ecological Food Web

Food web graph.

- Node = species.
- Edge = from prey to predator.

Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.

- Two representations of each edge.
- Space proportional to $n^2$.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.
**Adjacency list.** Node indexed array of lists.

- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if $(u, v)$ is an edge takes $O(\text{deg}(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.
Def. A path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, ..., v_{k-1}, v_k$ with the property that each consecutive pair $v_i, v_{i+1}$ is joined by an edge in $E$.

Def. A path is simple if all nodes are distinct.

Def. An undirected graph is connected if for every pair of nodes $u$ and $v$, there is a path between $u$ and $v$. 
**Cycles**

**Def.** A *cycle* is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct.

![Diagram of a graph with labeled nodes](image)

*cycle $C = 1-2-4-5-3-1$*
Trees

**Def.** An undirected graph is a **tree** if it is connected and does not contain a cycle.

**Theorem.** Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.
- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
Rooted Trees

**Rooted tree.** Given a tree $T$, choose a root node $r$ and orient each edge away from $r$.

**Importance.** Models hierarchical structure.

![Diagram of a tree and the same tree rooted at 1]
Phylogeny trees. Describe evolutionary history of species.
GUI Containment Hierarchy

GUI containment hierarchy. Describe organization of GUI widgets.

Reference: http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html
3.2 Graph Traversal
Connectivity

s-t connectivity problem. Given two node s and t, is there a path between s and t?

s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Applications.

- Friendster.
- Maze traversal.
- Kevin Bacon number.
- Fewest number of hops in a communication network.
Breadth First Search

**BFS intuition.** Explore outward from \( s \) in all possible directions, adding nodes one "layer" at a time.

**BFS algorithm.**
- \( L_0 = \{ s \} \).
- \( L_1 \) = all neighbors of \( L_0 \).
- \( L_2 \) = all nodes that do not belong to \( L_0 \) or \( L_1 \), and that have an edge to a node in \( L_1 \).
- \( L_{i+1} \) = all nodes that do not belong to an earlier layer, and that have an edge to a node in \( L_i \).

**Theorem.** For each \( i \), \( L_i \) consists of all nodes at distance exactly \( i \) from \( s \). There is a path from \( s \) to \( t \) iff \( t \) appears in some layer.
Breadth First Search

**Property.** Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
Theorem. The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Pf.

- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node $u$, there are $\leq n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\text{deg}(u)$ incident edges $(u, v)$
  - total time processing edges is $\sum_{u \in V} \text{deg}(u) = 2m$  

  $\uparrow$

  each edge $(u, v)$ is counted exactly twice in sum: once in $\text{deg}(u)$ and once in $\text{deg}(v)$
Connected Component

Connected component. Find all nodes reachable from s.

Connected component containing node 1 = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}. 
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- **Node**: pixel.
- **Edge**: two neighboring lime pixels.
- **Blob**: connected component of lime pixels.

recolor lime green blob to blue
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- **Node**: pixel.
- **Edge**: two neighboring lime pixels.
- **Blob**: connected component of lime pixels.

![Image with Flood Fill demonstration](image-url)
**Connected component.** Find all nodes reachable from s.

---

**Theorem.** Upon termination, $R$ is the connected component containing $s$.

- **BFS** = explore in order of distance from $s$.
- **DFS** = explore in a different way.

---

$R$ will consist of nodes to which $s$ has a path

- Initially $R = \{s\}$
- While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  - Add $v$ to $R$
- Endwhile