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**

Understanding the Solution

**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 non-optimal assignment is perfect, let alone stable.
- Simultaneously best for each and every man.

Stable Matching Summary

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

- 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.

**Woman pessimality.** Each woman receives worst valid partner.

- 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 prefers less than $Z$.
  - 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 $S$. □

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

**Pf.** (by contradiction)

- Suppose $A-Z$ matched in $S^*$.
  - $Z$ is not worst valid partner for $A$.
  - There exists stable matching $S$ in which $A$ is paired with a man, say $Y$, whom she prefers less than $Z$.
  - Let $B$ be $Z$’s partner in $S$.
  - Thus, $Z$ prefers $A$ to $B$.
  - $A-Z$ is unstable in $S$. □
Extensions: Matching Residents to Hospitals

Ex: Men \sim hospitals, Women \sim 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!

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.

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.

*Jobs don’t overlap*

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

---

**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!*

*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?

Problem can be solved using Network Flow Algorithms

Independent Set

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

Diffie-Hellman 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 $k$

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.

---

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: $nm$ max-flow based algorithm.

Independent set: NP-complete.

Competitive facility location: PSPACE-complete.

---

2.1 Computational Tractability

"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - Francis Sullivan
Computational Tractability

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

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

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.

Why It Matters

<table>
<thead>
<tr>
<th>n</th>
<th>$O(n)$</th>
<th>$O(n^2)$</th>
<th>$O(2^n)$</th>
<th>$O(n!)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</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>
</tr>
<tr>
<td>100</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>1,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>1,000,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
</tr>
</tbody>
</table>

2.2 Asymptotic Order of Growth
Asymptotic Order of Growth

Upper bounds. \( T(n) = O(f(n)) \) 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. \( T(n) = \Omega(f(n)) \) 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. \( T(n) = \Theta(f(n)) \) 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 \( O(n) \), \( \Omega(n) \), \( \Theta(n) \), or \( \Theta(n^3) \).

Notation

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 \( \Theta \) for lower bounds.

Properties

Transitivity.
- \( \text{if } T(n) \in O(g(n)) \text{ and } g(n) \in O(h(n)) \text{ then } T(n) \in O(h(n)) \).
- \( \text{if } T(n) \notin O(g(n)) \text{ and } g(n) \notin O(h(n)) \text{ then } T(n) \notin O(h(n)) \).
- \( \text{if } T(n) \notin O(g(n)) \text{ and } g(n) \in O(h(n)) \text{ then } T(n) \notin O(h(n)) \).

Additivity.
- \( \text{if } T(n) \in O(f(n)) \text{ and } g(n) \in O(h(n)) \text{ then } T(n) + g(n) \in O(f(n)) \).
- \( \text{if } T(n) \notin O(f(n)) \text{ and } g(n) \in O(h(n)) \text{ then } T(n) + g(n) \notin O(f(n)) \).
- \( \text{if } T(n) \notin O(f(n)) \text{ and } g(n) \notin O(h(n)) \text{ then } T(n) + g(n) \notin O(f(n)) \).

Proof of All \( \text{if } T(n) \in O(f(n)) \text{ and } g(n) \in O(h(n)) \text{ then } T(n) + g(n) \in O(f(n)) \).
- \( f(n) \in O(g(n)) \) means that for some constants \( c_1, n_1 \) we have \( f(n) \leq c_1 \cdot g(n) \) for all \( n \geq n_1 \).
- \( g(n) \in O(h(n)) \) means that for some constants \( c_2, n_2 \) we have \( g(n) \leq c_2 \cdot h(n) \) for all \( n \geq n_2 \).
- Set \( c = c_1 + c_2 \) and \( n = \max(n_1, n_2) \) for all \( n \geq N \leq \max(n_1, n_2) \).

Asymptotic Bounds for Some Common Functions

Polynomials. \( a_0 + a_1n + \ldots + a_\ell n^\ell \) is \( \Theta(n^\ell) \) if \( a_\ell > 0 \).

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

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

Logarithms. For every \( x > 0 \), \( \log n = O(xn) \).
- \( \log \) grows slower than every polynomial even if \( x = 0.000000001 \).

Exponentials. For every \( r > 1 \) and every \( \ell > 0 \), \( n^\ell = O(r^n) \).
- 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 \( a_1, \ldots, a_n \).

```java
int max = a0;
for (i = 1 to n {
    if (ai > max)
        max = ai;
}
```
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.

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

Proof. After each comparison, the length of output list increases by 1.

```java
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
```

$O(n \log n)$ Time

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

**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.

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

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

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 pair of sets, determine if they are disjoint.

```java
foreach set $S_i$ {
  foreach other set $S_j$ {
    foreach element $p$ of $S_i$ {
      if $p$ also belongs to $S_j$
    }
    if no element of $S_i$ belongs to $S_j$
      report that $S_i$ and $S_j$ are disjoint
  }
}
```

Polylogarithmic Time: $O(n \log n)$

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

Polynomial Time: $O(n^k)$

**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.

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

Remark. $O(n^k)$ seems inevitable, but this is just an illusion.

Exponential Time

**Independent set.** Given a graph, what is maximum size of an independent set?

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

```java
S* \leftarrow \emptyset
foreach subset $S$ of nodes {
  check whether $S$ is in an independent set
  if ($S$ is largest independent set seen so far)
    update $S^* \leftarrow S$
}
```
Review: Heap Data Structure

Heap Insertion

Min Heap Order: For each node v in the tree
Parent(v).Value ≤ v.Value

Max Heap Order: For each node v in the tree
Parent(v).Value ≥ v.Value

Heap Insertion

Heap.Insert(3)

Min Heap Order: For each node v in the tree
Parent(v).Value ≤ v.Value

Max Heap Order: For each node v in the tree
Parent(v).Value ≥ v.Value

Heap Insertion

Heap.Insert(3)

Min Heap Order: For each node v in the tree
Parent(v).Value ≤ v.Value

Max Heap Order: For each node v in the tree
Parent(v).Value ≥ v.Value

Heap Insertion

Heap.Insert(3)

Min Heap Order: For each node v in the tree
Parent(v).Value ≤ v.Value

Max Heap Order: For each node v in the tree
Parent(v).Value ≥ v.Value

Theorem 2.18 [KJ]: The procedure Heapify-up fixed 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

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

Theorem 2.19 (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?

3.1 Basic Definitions and Applications

Undirected Graphs

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

Some Graph Applications

Graph | Nodes | Edges
--- | --- | ---
- transportation: street intersections, highways
- communication: computers, fiber optic cables
- World Wide Web: web pages, hyperlinks
- social: people, relationships
- food web: species, predator-prey
- software systems: functions, function calls
- scheduling: tasks, precedence constraints
- circuit: gates, wires

World Wide Web

- Web graph
  - Node: web page
  - Edge: hyperlink from one page to another

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
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.

$\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
3 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
4 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
5 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
6 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
7 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
8 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
\end{array}$

Graph Representation: Adjacency List

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.

$\begin{array}{cccccccc}
1 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 1 & 2 & 3 & 4 & 5 & 8 \\
3 & 1 & 2 & 4 & 7 & 8 \\
4 & 2 & 3 & 5 & 6 & 8 \\
5 & 2 & 4 & 6 & 8 \\
6 & 5 & 8 \\
7 & 3 & 8 \\
8 & 2 & 4 & 7 \\
\end{array}$

Paths and Connectivity

Def. A path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, ..., v_k$ 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 $v_1, v_2, ..., v_k, v_1$ in which $v_i \neq v_0, k \geq 2$, and the first $k-1$ nodes are all distinct.

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

Definition. 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.

Phylogeny Trees

Phylogeny trees. Describe evolutionary history of species.

GUI Containment Hierarchy

GUI containment hierarchy. Describe organization of GUI widgets.

3.2 Graph Traversal

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

$s$-$t$ shortest path problem. Given two nodes $s$ 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.
- $L_0 = \{ s \}$
- $L_1 = \text{all neighbors of } L_0$
- $L_i = \text{all nodes that do not belong to } L_0 \text{ or } L_{i-1}, \text{and that have an edge to a node in } L_{i-1}$
- $L_{i+1} = \text{all nodes that do not belong to an earlier layer, and that have an edge to a node in } L_i$

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.

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.

Breadth First Search: Analysis

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 $\deg(u)$ incident edges $(u, v)$
  - total time processing edges is $\sum_{u \in V} \deg(u) = 2m$
  - each edge $(u, v)$ is counted exactly twice in sum since $\deg(u)$ and once in $\deg(v)$

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

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

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
**Connected Component**

**Connected component.** Find all nodes reachable from \( s \).

- \( R \) 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 \( R \).
  - It's safe to add \( v \).

**Theorem.** Upon termination, \( R \) is the connected component containing \( s \).
- BFS = explore in order of distance from \( s \).
- DFS = explore in a different way.