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.
Issues in algorithm design

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.
How to describe algorithms, II

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.
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.
Weighted Interval Scheduling

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.)

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 $M$.

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.)
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?
Stable matching problem

Rather than students & companies, we can frame the question in terms of men & women getting married:

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

- A matching is a set of pairs from $M \times W$ such that each man and each woman appears at most once.
- A perfect matching is a set of pairs from $M \times W$ that includes each man and each woman 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?
The pair \((m, w')\) is an instability with respect to the shown matching (solid lines).
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:

\[
\begin{align*}
  m & \text{ prefers } w \text{ to } w' \\
  m' & \text{ prefers } w' \text{ to } w' \\
  w & \text{ prefers } m \text{ to } m' \\
  w' & \text{ prefers } m \text{ to } m'
\end{align*}
\]

\((m, w), (m', w')\)

Example 2:

\[
\begin{align*}
  m & \text{ prefers } w \text{ to } w' \\
  m' & \text{ prefers } w' \text{ to } w \\
  w & \text{ prefers } m' \text{ to } m \\
  w' & \text{ prefers } m \text{ to } m'
\end{align*}
\]
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
Endwhile
Return set \( S \) of engaged pairs
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.
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 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?  
  Next $[m] =$ 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 $m'$?

\[
w : m_2, m_3, m_4, m_1\\Rank[w] = [4, 1, 2, 3]\\\text{Can test } \text{Rank}[w, m] < \text{Rank}[w, m']\]
Some notes

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

2. A women’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?

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

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

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

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

See page 11 of your book.