CSE 421: Introduction to Algorithms

Stable Matching

Shayan Oveis Gharan
Administrativia Stuffs

HW1 is due Wednesday April 03 at 11:59PM
Please submit to Gradescope

Late Submission: Fill out an extension request in edstem.

How to submit?
• Double check your submission before the deadline!!
• Please typeset your solution if possible

Guidelines:
• Always justify your answer
• You can collaborate, but you must write solutions on your own
• Read guidelines in the website
• Sanity Check: Spell out when you use assumptions of the problem
Summary

Stable matching problem: Given \( n \) companies and \( n \) applicants, and their preferences, find a stable matching if one exists.

- Gale-Shapley algorithm: Guarantees to find a stable matching for any problem instance.

- Q: If there are multiple stable matchings, which one does GS find?
- Q: How to implement GS algorithm efficiently?
- Q: How many stable matchings are there?

3
Company Optimal Assignments

Definition: Company $c$ is a valid partner of applicant $a$ if there exists some stable matching in which they are matched.

Company-optimal matching: Each company receives the best valid partner (according to its preferences).
- Not that each company receives its most favorite applicant.

Claim: All executions of GS yield a company-optimal matching, which is a stable matching!
- So, output of GS is unique!!
- No reason a priori to believe that company-optimal matching is perfect, let alone stable.
Claim: GS matching $S^*$ is company-optimal.
Proof: (by contradiction)
Suppose some company is paired with someone other than its best valid partner. Companies propose in decreasing order of preference $\Rightarrow$ some company is rejected by a valid partner.

Let $c$ be the first such rejection, and let $a$ be its best valid partner.

Let $S$ be a stable matching where $c$ and $a$ are matched.
In building $S^*$, when $c$ is rejected, $a$ is assigned to a company, say $c'$ whom she prefers to $c$.

Let $c'$ be $a'$ partner in $S$.
In building $S^*$, $c'$ is not rejected by any valid partner at the point when $c$ is rejected by $a$. Thus, $c'$ prefers $a$ to $a'$.

But $a$ prefers $c'$ to $c$.
Thus $(c', a)$ is unstable in $S$. since this is the first rejection by a valid partner
Company Optimality Summary

**Company-optimality:** In version of GS where companies propose, each company receives the best valid partner.

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

**Q:** Does company-optimality come at the expense of the applicants?
Applicant Pessimality

**Applicant-pessimal assignment:** Each applicant receives the worst valid partner.

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

**Proof.**

Suppose $(c, a)$ matched in $S^*$, but $c$ is not the worst valid partner for $a$. There exists stable matching $S$ in which $a$ is paired with a company, say $c'$, whom she likes less than $c$.

Let $a'$ be $c$ partner in $S$.

$c$ prefers $a$ to $a'$.  

Thus, $(c, a)$ is an unstable in $S$.  

company-optimality of $S^*$
Efficient Implementation

We describe $O(n^2)$ time implementation. This is linear in input size.

Representing company and applicant:
Assume companies are named $1, \ldots, n$.
Assume applicants are named $n+1, \ldots, 2n$.

Data Structure:
Maintain a list of free company, e.g., in a queue.
Maintain two arrays $\text{applicant}[c]$, and $\text{company}[a]$.
- set entry to 0 if unmatched
- if $c$ matched to $a$ then $\text{applicant}[c]=a$ and $\text{company}[a]=c$

Companies proposing:
For each company, maintain a list of applicants, ordered by preference.
Maintain an array $\text{count}[c]$ that counts the number of proposals made by company $c$. 

Efficient Implementation

Applicants rejecting/accepting.

Does applicant \( a \) prefer \( c \) to \( c' \)?

For each applicant, create inverse of preference list of companies.

Constant time access for each query after \( O(n) \) preprocessing per applicant. \( O(n^2) \) total reprocessing cost.

For \( i = 1 \) to \( n \)
for \( j = 1 \) to \( n \)
inverse[\( i \)][pref[\( i \)][\( j \)]\] = \( j \)

\( a_i \) prefers company \( 3 \) to \( 6 \)
since inverse[\( i \)][3]=2 < 7=inverse[\( i \)][6]
Summary

• **Stable matching problem:** Given $n$ men and $n$ women, and their preferences, find a stable matching if one exists.

• **Gale-Shapley algorithm** guarantees to find a stable matching for any problem instance.

• **GS algorithm** finds a stable matching in $O(n^2)$ time.

• **GS algorithm** finds man-optimal woman pessimal matching

• **Q:** How many stable matching are there?
How many stable Matchings?

We already show every instance has at least 1 stable matchings.

[Knuth’76] There are instances with about $2.24^n$ stable matchings for

[Karlin-O-Weber’17]: Every instance has at most $131072^n$ stable matchings

[Palmer-Palvolgyi’20]: Every instance has at most $4.47^n$ stable matchings

[Research-Question]:
Is there an “efficient” algorithm that chooses a uniformly random stable matching of a given instance.
Main Objective: Design Efficient Algorithms that finds optimum solutions in the Worst Case
Measuring Efficiency

Time $\approx$ # of instructions executed in a simple programming language

- Only simple operations (+, *, -, =, if, call, …)
- Each operation takes one time step each memory access takes one time step
- No fancy stuff (add these two matrices, copy this long string, …) built in; write it/charge for it as above
- Hashing: A hash function takes $O(n)$ to search for an element in the worst case.
Time Complexity

Problem: An algorithm can have different running time on different inputs

Solution: The complexity of an algorithm associates a number $T(N)$, the “time” the algorithm takes on problem size $N$.

On which inputs of size $N$?

Mathematically,

$T$ is a function that maps positive integers giving problem size to positive integers giving number of steps
Time Complexity (N)

Worst Case Complexity: \( \text{max} \) # steps algorithm takes on any input of size \( N \)

Average Case Complexity: \( \text{avg} \) # steps algorithm takes on inputs of size \( N \)

Best Case Complexity: \( \text{min} \) # steps algorithm takes on any input of size \( N \)
Why Worst-case Inputs?

- Analysis is typically easier
- Useful in real-time applications
e.g., space shuttle, nuclear reactors)
- Worst-case instances kick in when an algorithm is run as a module many times
e.g., geometry or linear algebra library
- Useful when running competitions
e.g., airline prices
- Unlike average-case no debate about the right definition
Time Complexity on Worst Case Inputs

\[ T(N) = 2N \log_2 N \]

\[ T(N) = N \log_2 N \]
O-Notation

Given two positive functions $f$ and $g$

• $f(N)$ is $O(g(N))$ iff there is a constant $c > 0$ s.t., $f(N)$ is eventually always $\leq c \cdot g(N)$

• $f(N)$ is $\Omega(g(N))$ iff there is a constant $\varepsilon > 0$ s.t., $f(N)$ is $\geq \varepsilon \cdot g(N)$ for infinitely

• $f(N)$ is $\Theta(g(N))$ iff there are constants $c_1, c_2 > 0$ so that eventually always $c_1 g(N) \leq f(N) \leq c_2 g(N)$
Asymptotic Bounds for common fns

• **Polynomials:**
  \[ a_0 + a_1 n + \cdots + a_d n^d \text{ is } O(n^d) \]

• **Logarithms:**
  \[ \log_a n = O(\log_b n) \text{ for all constants } a, b > 0 \]

• **Logarithms:** log grows slower than every polynomial
  For all \( x > 0 \), \( \log n = O(n^k) \)

• \( n \log n = O(n^{1.01}) \)
Efficient = Polynomial Time

An algorithm runs in polynomial time if $T(n) = O(n^d)$ for some constant $d$ independent of the input size $n$.

Why Polynomial time?

If problem size grows by at most a constant factor then so does the running time

- E.g. $T(2N) \leq c(2N)^k \leq 2^k(cN^k)$
- Polynomial-time is exactly the set of running times that have this property

Typical running times are small degree polynomials, mostly less than $N^3$, at worst $N^6$, not $N^{100}$
Why it matters?

- #atoms in universe < $2^{240}$
- Life of the universe < $2^{54}$ seconds
- A CPU does < $2^{30}$ operations a second

If every atom is a CPU, a $2^n$ time ALG cannot solve $n=350$ if we start at Big-Bang.

<table>
<thead>
<tr>
<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>$n = 10$</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>
</tr>
<tr>
<td>$n = 30$</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>
</tr>
<tr>
<td>$n = 50$</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>
</tr>
<tr>
<td>$n = 100$</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>
</tr>
<tr>
<td>$n = 1,000$</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>
</tr>
<tr>
<td>$n = 10,000$</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>
</tr>
<tr>
<td>$n = 100,000$</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>
</tr>
<tr>
<td>$n = 1,000,000$</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>
</tr>
</tbody>
</table>

Not only get very big, but do so *abruptly*, which likely yields erratic performance on small instances.
Why “Polynomial”?

Point is not that $n^{2000}$ is a practical bound, or that the differences among $n$ and $2n$ and $n^2$ are negligible. Rather, simple theoretical tools may not easily capture such differences, whereas exponentials are qualitatively different from polynomials, so more amenable to theoretical analysis.

• “My problem is in P” is a starting point for a more detailed analysis
• “My problem is not in P” may suggest that you need to shift to a more tractable variant