CSE 421: Introduction to Algorithms

Stable Matching

Shayan Oveis Gharan

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?

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.

Company Optimality Summary

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

> a is a valid partner of c if there exist some stable matching where c and a 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'. ← company-optimality of **S***

Thus, (c, a) is an unstable in **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, …, n.

Assume applicants are named n+1, …, 2n.

Data Structure:

Maintain a list of free company, e.g., in a queue.

Maintain two arrays **applicant[c]**, and **company[a]**.

- set entry to **0** if unmatched
- if **c** matched to **a** then **applicant[c]=a** and **company[a]=c**

Companies proposing:

For each company, maintain a list of applicants, ordered by preference.

Maintain an array **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(n²)** total reprocessing cost.

a_i	1st	2nd	3rd	4th	5th	6th	7th	8th
Pref	8	3	7	1	4	5	6	2

a_i	1	2	3	4	5	6	7	8
Inverse	4th	8th	2nd	5th	6th	7th	3rd	1st

```
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 $\mathbf{O(n^2)}$ time. ✓
- GS algorithm finds man-optimal woman pessimal matching ✓
- Q: How many stable matching are there?

Lessons Learned

- Powerful ideas learned in course.
  - Isolate underlying structure of problem.
  - Create useful and efficient algorithms.

- Potentially deep social ramifications. [legal disclaimer]
  - Always try to propose first!

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.

Induction: Intro 1

Prove that for all $n \geq 1$,

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

Def $P(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

Base Case: $P(1)$ holds: $1 = 1(1+1)/2$

IH: $P(n-1)$ holds.

IS: Goal to prove $P(n)$.

$$\begin{aligned} 1 + \cdots + n &= (1 + \cdots + n - 1) + n \\ &= \left(\frac{(n-1)n}{2}\right) + n \qquad \text{By IH} \\ &= \frac{n(n+1)}{2} \end{aligned}$$

Induction: Intro 2

Prove that if n+1 balls are placed into n bins then one bin has at least two balls.

Def: P(n): If n+1 balls are placed into n bins then one bin has at least two balls.

Base Case: P(1) holds. Two balls into one bin

IH: P(n-1) holds)

IS: Goal is to prove P(n). Suppose n+1 balls are placed into n bins. Need to show a bin has >=2 balls. Look at bin 1.

Case 1: Bin 1 has at least two balls. Then we are done.

Case 2: Bin 1 has 1 ball. Then. we have placed n balls into bins 2,..,n. So, by IH one bin has at least two balls.

Case 3: Bin 1 has 0 balls. Remove an arbitrary ball. Then, we have n balls into bins 2,..,n. So, by IH a bin has >=2 balls

Main Objective: Design Efficient Algorithms
that finds optimum solutions in the Worst Case

Measuring Efficiency

Time ≈ # 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

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: max # steps algorithm takes on any input of size **N**

This Couse

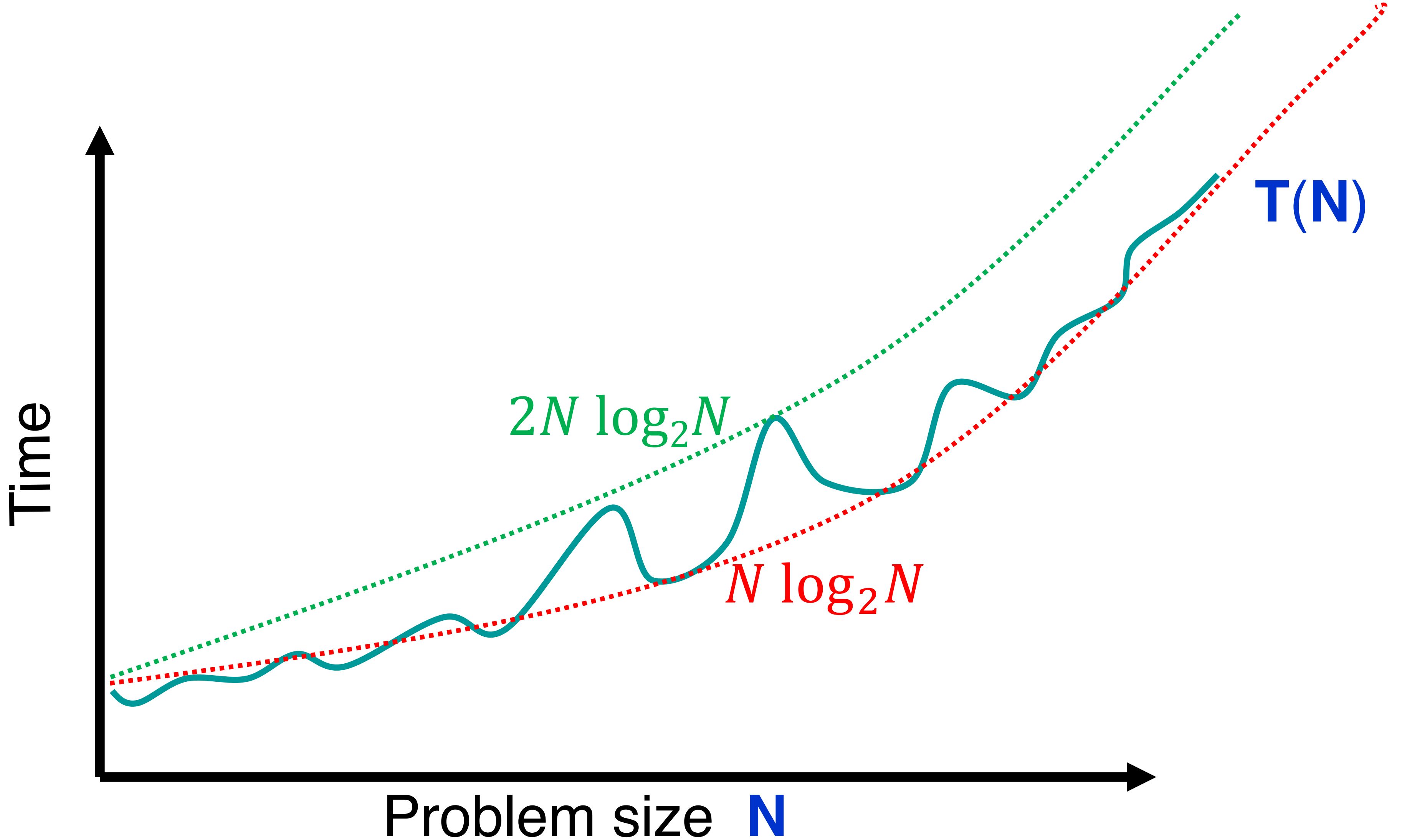
Average Case Complexity: avg # steps algorithm takes on inputs of size **N**

Best Case Complexity: 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, uber, …)
- 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, online retail, …
- Unlike average-case no debate about the right definition

Time Complexity on Worst Case Inputs



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 $\leq$ **$c\ g(N)$**

- **$f(N)$** is **$\Omega(g(N))$** iff there is a constant **$\varepsilon>0$** s.t.,
  $f(N)$ is $\geq \varepsilon$ **$g(N)$** 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:

  $a_0 + a_1 n + \cdots + a_d n^d$ is $O\left(n^d\right)$

- Logarithms:

  $\log_a n = O(\log_b n)$ 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. $\mathbf{T(2N) \leq c(2N)^k \leq 2^k(cN^k)}$
- Polynomial-time is exactly the set of running times that have this property

Typical running times are small degree polynomials, mostly less than $\mathbf{N^3}$, at worst $\mathbf{N^6}$, not $\mathbf{N^{100}}$

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.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

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