Here Early?

Here for CSE 421?

Welcome! You’re early!

Want a copy of these slides to take notes?

- You can download them from the webpage cs.uw.edu/421

Stable Matchings

CSE 421 Fall 22
Lecture 1

Today

Logistics

What is this course?

Start of the content

Staff

Instructor: Robbie Weber

Ph.D. from UW CSE in theory
Third year as teaching faculty

Office: CSE2 311
Email: rtweber2@cs.washington.edu

TAs

Abhinav Bandari
Daniel Cheng
Kai Daniels
Airei Fukuzawa
Anna Goncharenko
Allie Pfleger
Alicia Stepin
Alice Wang
Jack Zhang
Muru Zhang

Classroom Logistics

We’re in-person :D

Masks are “strongly recommended” (but not required) by university policy.

Please speak loudly for questions

I’ll repeat questions into the microphone (so everyone can hear/for the recording).

TODO List

Make sure you’re on Ed! (check your spam folder for an invite, if not there send an email to Robbie).

Go to section tomorrow.

Homework 1 is out (should be able to start after section).

Syllabus

It’s all on the webpage:
https://courses.cs.washington.edu/courses/cse421/22au

In general, we’re designing lecture to be synchronous (taking questions live, time for student discussions).

But we’re trying to make sure if you need to stay home for a few days on short notice the effect will be minimal.

Textbook

Optional: Algorithm Design by Kleinberg & Tardos

It's a good introduction, and nice as a reference.

There are lots of other books:

- Introduction to algorithms by Cormen, Leiserson, Rivest, Stein
- One free reference: Algorithms by Jeff Erickson Algorithms.wtf

Logistics – Lecture Activities

I'm going to be teaching with **active learning** in this course.

Why? Because it works.
https://www.pnas.org/content/111/23/8410 a meta-analysis of 225 studies.

Just listening to me isn't as good for you as listening to me then trying problems on your own and with each other.

The answers live help me adjust explanations.

There aren't points associated with completing the activities.

Logistics – where to go?

Slides, homework problems, etc. go up on the webpage

Homework submission on gradescope

Live lecture activities on polleverywhere

Questions on Ed discussion board

Don’t trust canvas – we won’t be updating frequently. We’ll tell you when we’re using it for specific purposes.

Late Policy

A little different from what you're used to!

We're counting late ***problems*** instead of late ***days***.

You can use a "late problem" to turn in a problem up to 48 hours late.

- That applies to one problem not the other 3 on the assignment
- But you can use more than one per assignment

You have 5 to use.

We'll also drop a few of your lowest scores. Details on the syllabus.

Logistics – Work

8 (approximately) weekly homework assignments

A midterm exam

In the evening of Monday November 7th.

We'll have alternate exams for people celebrating Diwali, or other immovable conflicts, but please put that date on your calendar now!

A final exam

(as on the official schedule, Monday Dec. 12, 2:30 PM)

Hey, We’re in a pandemic

The staff is going to do our best to help you learn.

Real life is going to get in the way. If it does, tell us as soon as possible, and we’ll work with you.

I don’t need to know private details, just enough to know it’s an emergency and how to help.

We will endeavor not to make any substantial changes to the syllabus.

But if something extremely unexpected happens we reserve the right to make changes.

Generally prefer individual accommodations, rather than course-wide ones.

What is this course?

Algorithms

themes:

“Design Techniques” – not just “here’s an algorithm” but “here’s a way of thinking about a class of algorithms”

“Modeling” – In the real world, no one will say “I need you to run Prim’s algorithm on this graph” they will say “I need you to choose where to build electrical wires so every town is connected to the power plant as cheaply as possible

“Set realistic expectations” – there are some things we (think/know) computers can’t do efficiently. How do you recognize these problems?

“Reductions” – if you’ve already solved a problem, don’t solve it again (reuse ideas) and if you know you can’t solve a problem, what else can’t you solve.

What is this course **not**

NOT: A list of the fastest-known algorithms for common problems.

I’m not concerned with which library is best.

- The best library changes over time and by language.
- I’m not qualified to keep a list.
- I want you to find this course useful 5 years from now.

And the best theoretical algorithms probably aren’t practical…

- and when they are, it’s often clever combinations/complicated variants of big ideas that we’ll see.

Course Topics (Tentative)

Stable Matchings

Graph Algorithms from 332 (BFS/DFS)

Greedy Algorithms

Divide & Conquer

Dynamic Programming

Network Flow

Linear Programming

P/NP

What’s Coming Up

This week: An extremely useful algorithm.

That has had lots of effect on the real world.

Along the way:

Examples of what we mean by an algorithm description

We won’t use Java

How do we prove algorithms correct? Resetting expectations on proofs.

Stable Matchings

Stable Matchings

Motivation:

You have to assign TAs to instructors.

Two groups of people you need to pair off, with preferences about their matches.
You can't make everyone happy…so at least ensure that everyone listens to you.

There are lots of other similar applications

Assign doctors to the hospitals where they do residency.

Assign high schoolers to magnet schools.

Among many, many other applications.

Motivation

The real world is complicated.

- Students shouldn't TA a course they haven't taken.
- Instructors need varying numbers of TAs.
- There are more TA applicants than positions.
- Doctors might want to be in the same city as their partner.

We're going to simplify away the real world constraints.

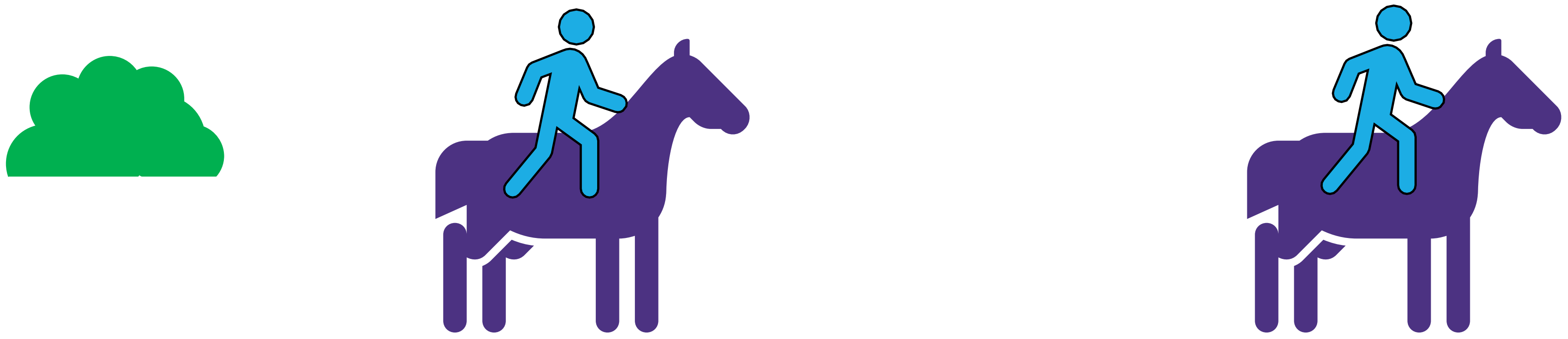
- The core ideas have been adapted to all of these scenarios.

Stable Matching Problem

To simplify. We have two sets:
A set of n horses, and a set of n riders.

Every rider can ride any horse, and vice versa.

We just need to pair them off. What could go wrong?



Stable Matching Problem

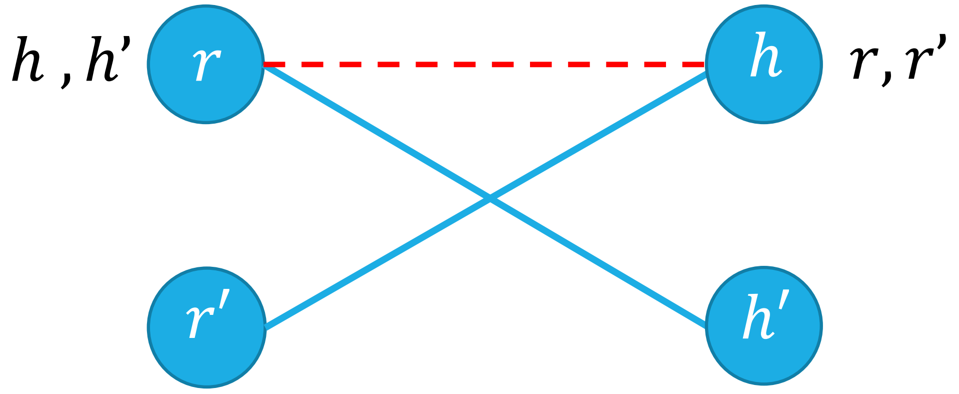
Given two sets $R = \{r_1, \dots, r_n\}, H = \{h_1, \dots, h_n\}$

each agent ranks **every** agent in the other set.

Goal: Match each agent to **exactly one** agent in the other set, respecting their preferences.

How do we "respect preferences"?

Avoid blocking pairs: unmatched pairs (r, h) where r prefers h to their match, and h prefers r to its match.



Stable Matching, More Formally

Perfect matching:

- Each rider is paired with exactly one horse.
- Each horse is paired with exactly one rider.

Stability: no ability to exchange

an unmatched pair r-h is blocking if they both prefer each other to current matches.

Stable matching: perfect matching with no blocking pairs.

Stable Matching Problem

Given: the preference lists of n riders and n horses.
Find: a stable matching.

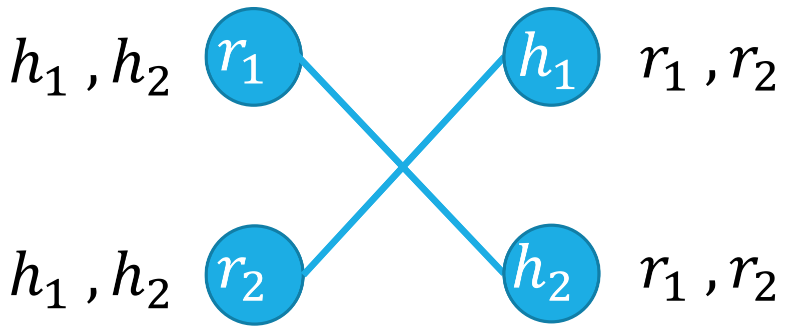
Lecture Activity

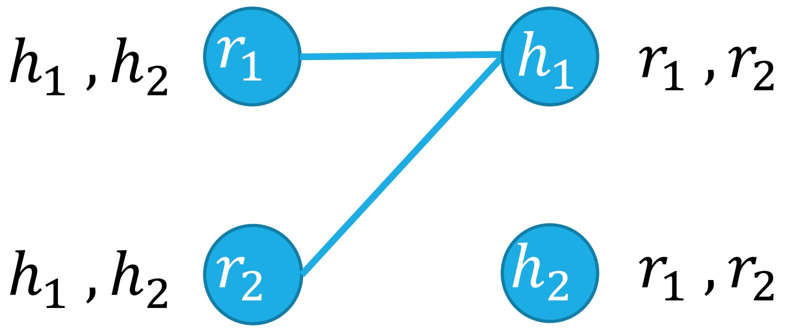
To make sure you've got the definition:

1. Download the activity pdf from the webpage (it's just the next slide in this slide deck) or look at the physical handout.
2. Introduce yourself to those around you.
3. Try the problem.
4. Fill out the polleverywhere

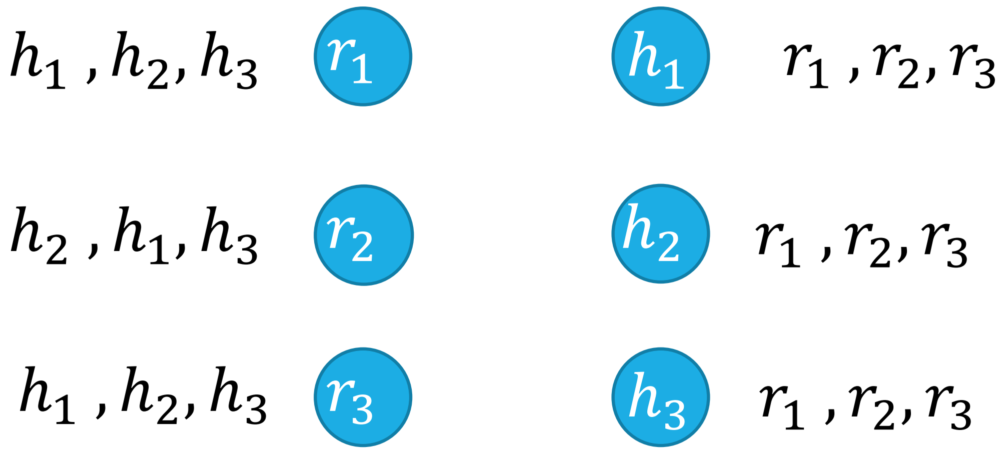
Try it!

Why are these not stable matchings?





Find a stable matching for this instance.



pollev.com/robbie

Questions

Does a stable matching always exist?

Can we find a stable matching efficiently?

We'll answer both of those questions in the next few lectures.

Let's start with the second one.

Idea for an Algorithm

Key idea

Unmatched riders “propose” to the highest horse on their preference list that they have not already proposed to.

Send in a rider to walk up to their favorite horse.

Everyone in front of a different horse? Done!

If more than one rider is at the same horse, let the horse decide its favorite.

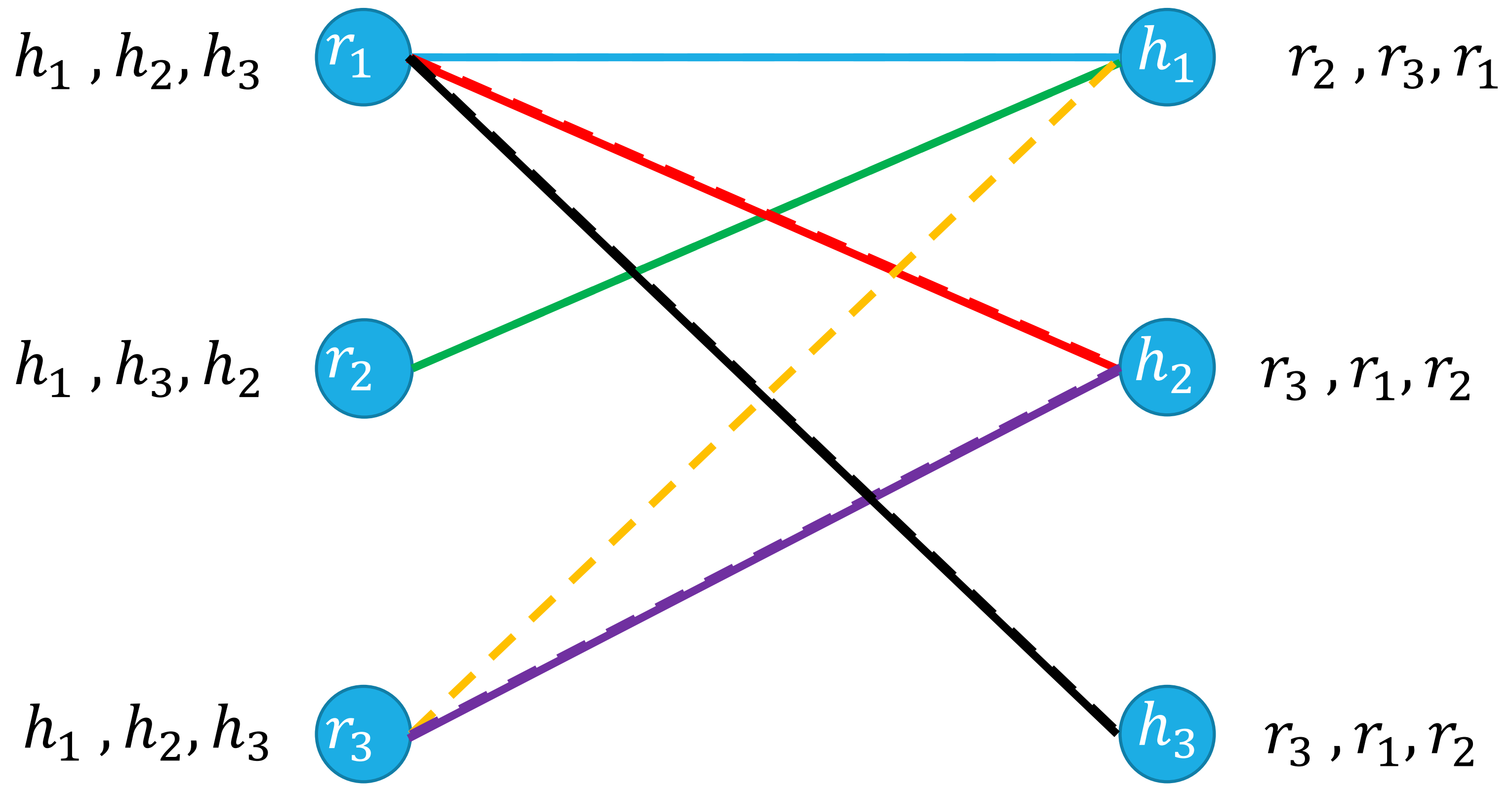
Rejected riders go back outside.

Repeat until you have a perfect matching.

Gale-Shapley Algorithm

```
Initially all $r$ in $R$ and $h$ in $H$ are free
while there is a free $r$
    Let $h$ be highest on $r$s list that $r$ has not proposed to
    if $h$ is free
        match $(r, h)$
    else //$h$ is not free
        Let $r'$ be the current match of $h$.
        if $h$ prefers $r$ to $\mathrm{r}'$
            unmatch $(r', h)$
            match $(r, h)$
```

Algorithm Example



Proposals: $r_1, r_2, r_1, r_3, r_3, r_1$

Does this algorithm work?

Does it run in a reasonable amount of time?

Is the result correct (i.e. a stable matching)?

Begin by identifying invariants and measures of progress

- Observation A: r's proposals get worse for them.
- Observation B: Once h is matched, h stays matched.
- Observation C: h's partners get better.

How do we justify these? A one-sentence explanation would suffice for each of these on the homework.
How did we know these were the right observations? Practice. And editing – we wouldn't have found these the first time, but after reading through early proof attempts.

Does this algorithm work?

Want to show two things:

1. The code produces the right output (i.e., you get a stable matching)

2. The code runs in a reasonable amount of time.

We’ll start with question 2.

Claim 1: If r proposed to the last horse on their list, then all the horses are matched.

Claim 1: If r proposed to the last horse on their list, then all the horses are matched.

Try to prove this claim, i.e. clearly explain why it is true. You might want some of these observations:

Observation A: r's proposals get worse (for r).

Observation B: Once h is matched, h never becomes free again.

Observation C: h's partners cannot get worse (for h).

Hint: r must have been rejected a lot – what does that mean?

Claim 1: If r proposed to the last horse on their list, then all the horses are matched.

Hint: r must have been rejected a lot – what does that mean?

Since we immediately match any horse we un-match in the algorithm, once a horse receives any proposal it is not free for the rest of the algorithm. (Observation B).

Since r proposes to horses on its list in order, every horse on r's list must be matched.

And every horse is on r's list! So once a rider proposes to the last horse on their list, all horses are matched.

Claim 2: The algorithm stops after $O(n^2)$ iterations.

Hint: When do we exit the loop? (Use claim 1).

If every horse is matched, every rider must be matched too.

- Because each horse is matched to exactly one rider and there are an equal number of riders and horses.

Since we don't repeat a proposal, and each of the n riders have lists of length n, It takes at most $O(n^2)$ proposals to get to the end of some rider's list.

Claim 2 now follows from Claim 1.

That's the number of iterations. What about time per iteration?

Wrapping up the running time

We need $O(n^2)$ proposals. But how many steps does the full algorithm execute?

Depends on how we implement it…we're going to need some data structures.

With the right data structures the running time really is $O(n^2)$. More details in the optional slides at the end of this deck.

TODO List

Make sure you’re on Ed! (check your spam folder for an invite, if not there send an email to Robbie).

Go to section tomorrow

Start on HW1

Optional Content

Data Structure Tricks to run Gale-Shapley in $O(n^2)$ time.

Gale-Shapley Algorithm

```
Initially all $r$ in $R$ and $h$ in $H$ are free
while there is a free $r$
    Let $h$ be highest on $r'$s list that $r$ has not proposed to
    if $h$ is free
        match $(r,h)$
    else //$h$ is not free
        Let $r'$ be the current match of $h$.
        if $h$ prefers $r$ to $\mathrm{r}'$
            unmatch $(r',h)$
            match $(r,h)$
```

Are each of these operations really $O(1)$?
Assume that you get two `int[][]` with the preferences.

Wrapping up the running time

We need $O(n^2)$ proposals. But how many steps does the full algorithm execute?

Depends on how we implement it…we're going to need some data structures.

Gale-Shapley Algorithm

```
Initially all $r$ in $R$ and $h$ in $H$ are free
while there is a free $r$
    Let $h$ be highest on $r'$s list that $r$ has not proposed to
    if $h$ is free
        match $(r,h)$
    else //$h$ is not free
        Let $r'$ be the current match of $h$.
        if $h$ prefers $r$ to $\mathrm{r}'$
            unmatch $(r',h)$
            match $(r,h)$
```

Are each of these operations really $O(1)$?
Assume that you get two `int[][]` with the preferences.

Gale-Shapley Algorithm

```
Initially all $r$ in $R$ and $h$ in $H$ are free
while there is a free $r$
    Let $h$ be highest on $r'$s list that $r$ has not proposed to
    if $h$ is free
        match $(r, h)$
    else //$h$ is not free
        Let $r'$ be the current match of $h$.
        if $h$ prefers $r$ to r'
            unmatch $(r', h)$
            match $(r, h)$
```

Need to maintain free r. What can insert and remove in $O(1)$ time?

Each r should know where it is on its list.

Maintain partial matching

Given two riders, which horse is preferred?

Maintain partial matching

Are each of these operations really $O(1)$?
Assume that you get two `int[][]` with the preferences.
`Horse[i][j]` gives the identity of the j^{th} rider on horse i's list.

What data structures?

Need to maintain free r. What can insert and remove in $O(1)$ time?

Queue, stack, or list (inserting at end) all would be fine.

Maintain partial matching

Two arrays (index `i` has number for partner of agent `i`.

Each r should know where it is on its list.

`int` for each rider (likely store in an array)

Given two riders, which is preferred?

Lookup in `int[][]` takes…$O(n)$ in the worst case. Uh-oh.
Better idea: build "inverse" arrays (given rider, what is their **rank** for horse?).
One time cost of $O(n^2)$ time and space to build, but comparisons $O(1)$.

What data structures?

Need to maintain free r. What can insert and remove in $O(1)$ time?

Queue, stack, or list (inserting at end) all would be fine.

Maintain partial matching

Two arrays (index `i` has number for partner of agent `i`).

Each r should know where it is on its list.

`int` for each rider (likely store in an array)

Given two riders, which is preferred?

Lookup in `int[][]` takes…$O(n)$ in the worst case. Uh-oh.
Better idea: build "inverse" arrays (given rider, what is their **rank** for horse?).
One time cost of $O(n^2)$ time and space to build, but comparisons $O(1)$.

These aren't the only options – you might decide on an object-based approach (can meet same time bounds up to constant factors)

How do we handle the lookup?

Changing The Question

`Horse[i][j]` asks horse i who their j^{th} favorite rider is.

To ask Horse i, "do you prefer rider 3 or rider 5" we'd have to iterate through `Horse[i][k]` as k goes from 0 to $n-1$, until we see 3 or 5.

It would be better if we could ask "horse i, where is rider 3 on your list?" and "horse i, where is rider 5 on your list?" have it say "2nd on my list" and "8th on my list" to let us say "well $2 < 8$, so you would prefer rider 3."

Let's make a data structure to answer the other question.

Inverse Array

`Inv[i][j]` answers the question "Hey, horse i, what position in your list is rider j?

```
for(int k=0; k<n; k++){
    int riderNum = horse[i][k];
    Inv[i][riderNum]=k;
}
```

`riderNum` is the identity of the rider who is in position k. So when we ask about `riderNum`, the answer should be k.

Inverse Array

Repeat that code for every i (probably making a 2-D inverse array), and you'll have $O(1)$ access to answer the question "Rider 7, Do you prefer horse 3 or horse 5?

How long does it take to create? $O(n^2)$ time total.

So as a pre-computation step, create the inverse arrays. Then run Gale-Shapley as shown a few slides ago.

Every other step was $O(1)$ time, so…

So the final running time of Gale-Shapley will be $O(n^2)$