Stable Matchings

Stable Matching Problem

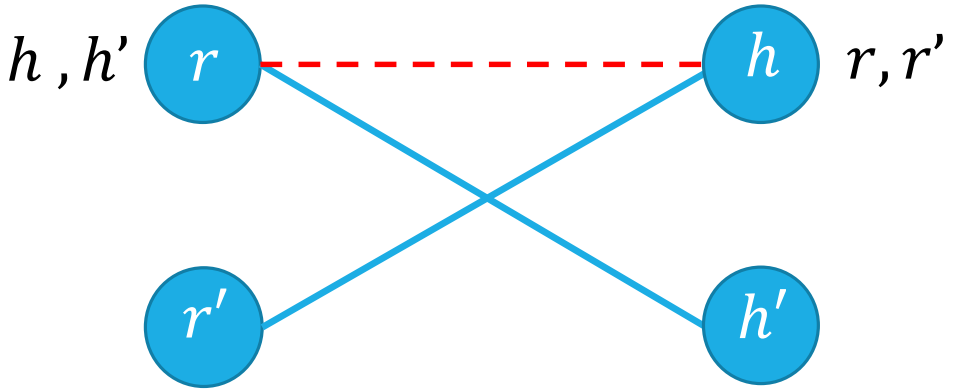
Given two sets $\mathrm{R} = \{r_1, \ldots, r_n\}, H = \{h_1, \ldots, h_n\}$

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.

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.

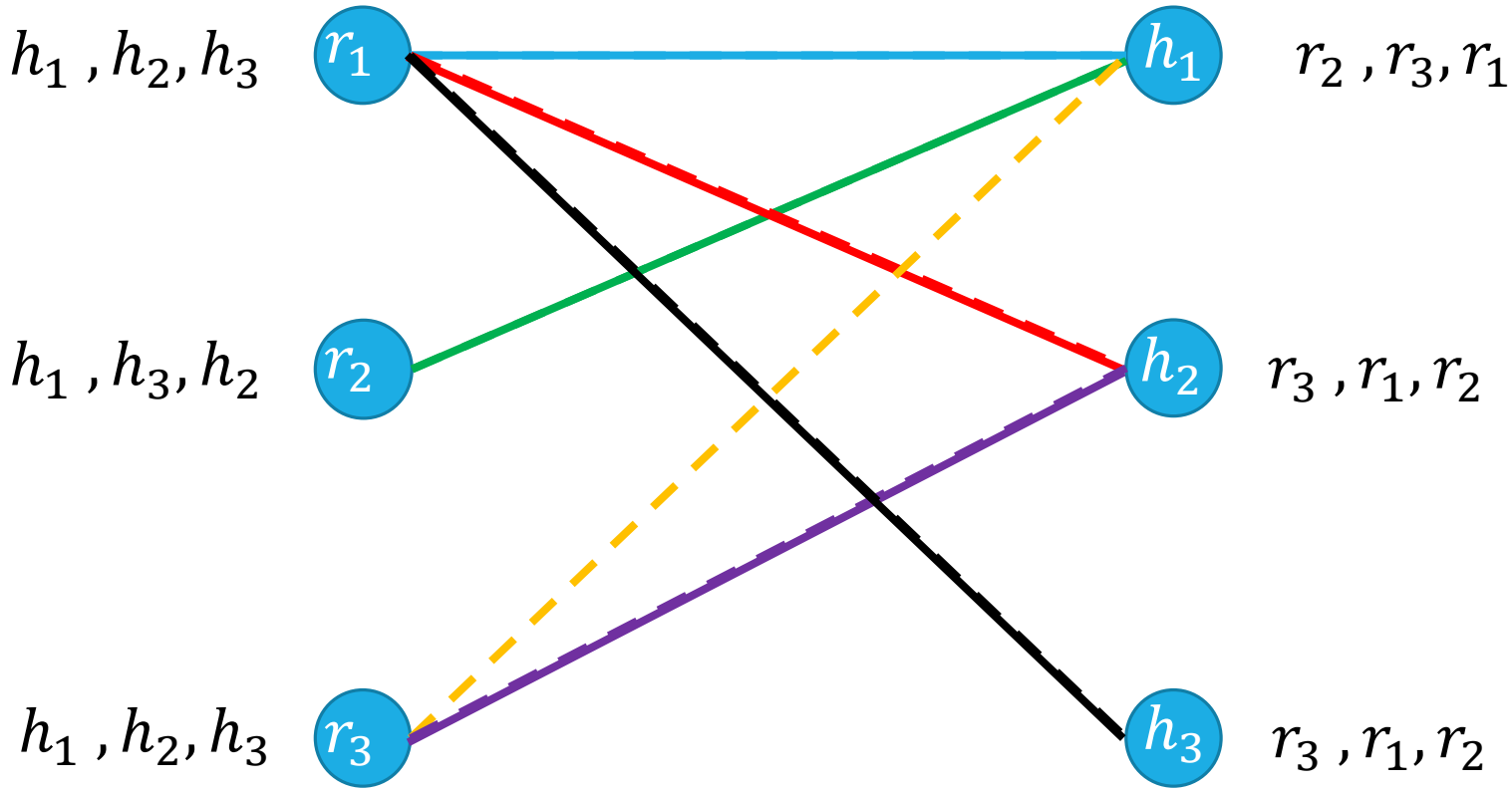
Rejected riders go back outside.

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, then match (r, h)
    else //h is not free
    suppose (r', h) are matched
        if h prefers r to 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.

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 them.**

Observation B: **Once h is matched, h stays matched.**

Observation C: **h's partners get better.**

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?

By Observation B, once a horse receives any proposal it is not free for the rest of the algorithm.

So every horse on r's list must be matched.

And every horse is on r's list!

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

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.

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.

Question 1 answered: The algorithm halts (quickly)!

Now question 2: does it produce a stable matching?

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, then match (r, h)
    else //h is not free suppose (r', h) are matched
        if h prefers r to 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, then match (r, h)
    else //h is not free suppose (r', h) are matched
        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?

Maintain partial matching

Each r should know where it is on its list.

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.

What data structures should you use?

Initially all r in R and h in H are free

While there is a free r

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

Let h be highest on r's list that r has not proposed to

if h is free, then match (r, h)

Maintain partial matching

Each r should know where it is on its list.

else //h is not free suppose (r', h) are matched

if h prefers r to r′

Given two riders, which horse is preferred?

unmatch (r', h)

match (r, h)

Maintain partial matching

Fill out the poll everywhere for Activity Credit!
Go to pollev.com/cse417 and login with your UW identity

Introduce yourselves!

If you can turn your video on, please do.
If you can't, please unmute and say hi.
If you can't do either, say "hi" in chat.

Choose someone to share screen, showing this pdf.

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)

But tl;dr: You really can get $O(n^2)$ time!

Analyzing Gale-Shapley

Efficient?
Halts in $O(n^2)$ steps. ✔

Works?
Need a matching that's:

- Perfect
- Has no blocking pairs

Claim 3: The algorithm identifies a perfect matching.

Why?

We know the algorithm halts. Which means when it halts every rider is matched.

But we have the same number of horses and riders, and we matched them one-to-one.

Hence, the algorithm finds a perfect matching.

Claim 4: The matching has no blocking pairs.

We want to prove a negative

there is no blocking pair.

That's a good sign for proof by contradiction.

What's proof by contradiction?

I want to know p is true.

Imagine, p were false. Study the world that would result.

Realize that world makes no sense (something false is true)

But the real world does make sense! So p must be true.

Claim 4: The matching has no blocking pairs.

We want to prove a negative

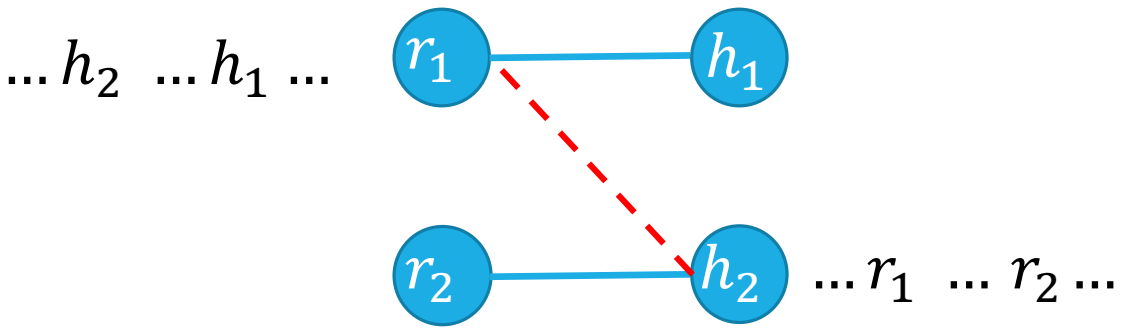
there is no blocking pair.

That's a good sign for proof by contradiction.

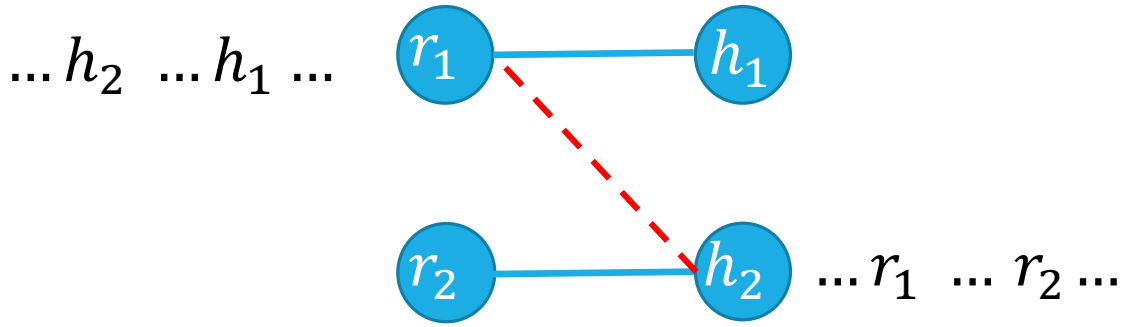
Suppose (for contradiction) that (r_1, h_1) and (r_2, h_2) are matched, but

r_1 prefers h_2 to h_1 and

h_2 prefers r_1 to r_2



Claim 4: The matching has no blocking pairs.



How did r_1 end up matched to h_1?

He must have proposed to and been rejected by h_2 (Observation A).

Why did h_2 reject r_1? It got a better offer from some r'.

If h_2 ever changed matches after that, the match was only better for it, (Observation C) so it must prefer r_2 (its final match) to r_1.

A contradiction!

Result

Simple, $O(n^2)$ algorithm to compute a stable matching

Corollary
A stable matching always exists.

The corollary isn't obvious!

The "stable roommates problem" doesn't always have a solution:
$2n$ people, rank the other $2n - 1$
Goal is to pair them without any blocking pairs.

Multiple Stable Matchings

Suppose we take our algorithm and let the horses do the "proposing" instead.

h_1 , h_2 r_1 h_1 r_2 , r_1

h_2 , h_1 r_2 h_2 r_1 , r_2

We got a different answer…

What does that mean?

Proposer-Optimality

Some agents might have more than one possible match in a stable matching. Call these people the "feasible partners."

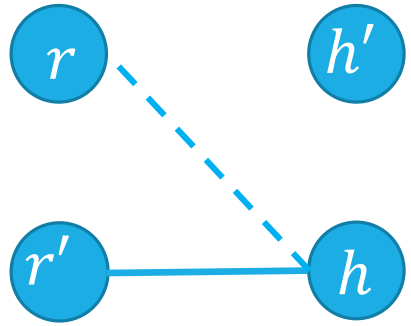
When there's more than one stable matching, there is a tremendous benefit to being the proposing side.

Proposer-Optimality

Every member of the proposing side is matched to their favorite of their feasible partners.

Proposer-Optimality

Every member of the proposing side is matched to the favorite of their feasible partners.



Let's prove it – again by contradiction

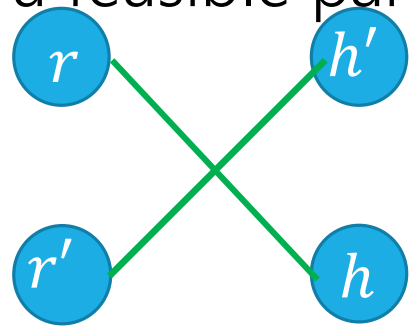
Suppose some rider is not matched to their favorite feasible partner.
Then some r must have been the **first** to be rejected by their favorite feasible partner, h. (Observation A)
And there is an r' that h (temporarily) matched to causing that rejection.

Let M' be a matching where (r, h) are matched. The rider r' is matched to some h'.

What can we say about r'? They had never been rejected by a feasible partner. So they prefer h to h'.

And h prefers r' to r (by the run of the algorithm).

But then (r', h) are a blocking pair in M'!



Implications of Proposer Optimality

Proposer-Optimality

Every member of the proposing side is matched to their favorite of their feasible partners.

We didn't specify which rider proposes when more than one is free

Proposer-optimality says it doesn't matter! You always get the proposer-optimal matching.

So what happens to the other side?

Chooser-Pessimality

A similar argument (it's a good exercise!), will show that choosing among proposals is a much worse position to be in.

Chooser-Pessimality

Every member of the choosing (non-proposing) side is matched to their least favorite of their feasible partners.

Some More Context and Takeaways

Stable Matching has another common name: “Stable Marriage”

The metaphor used there is “men” and “women” getting married.

When choosing or analyzing an algorithm think about everyone involved, not just the people you’re optimizing for; you might not be able to have it all.

Stable Matchings always exist, and we can find them efficiently.

The GS Algorithm gives proposers their best possible partner

At the expense of those receiving proposals getting their worst possible.