More Stable Matchings

CSE 421 Winter 2023
Lecture 2

Announcements

HW1 is out! Due Wednesday.

Office hours start today, see last night's email for today's schedule.

Regular OH schedule will appear on the calendar soon.

Where Were We?

Last time:

Introduced what a stable matching is

Today:

How do we find a stable matching?

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.

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 rs 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)
```

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 never becomes free.
- Observation C: h's partners get better each time it changes.

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 Lecture 1's slides.

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.

Claim 4: The matching has no blocking pairs.

We want to prove a negative

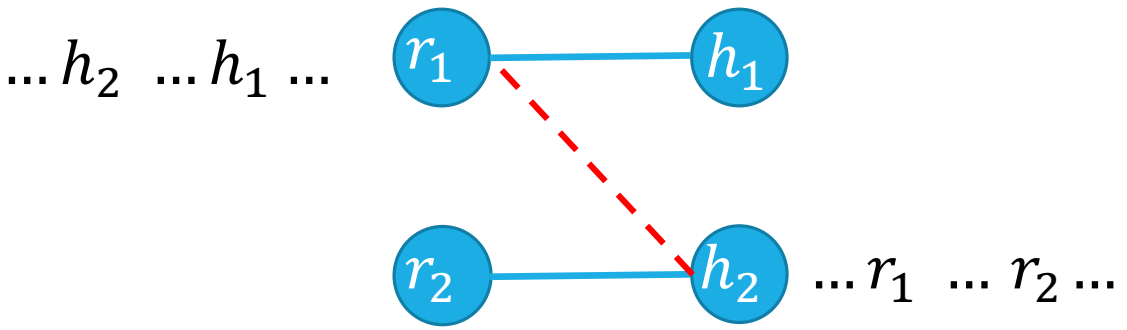
there is no blocking pair.

That's a good sign for proof by contradiction.

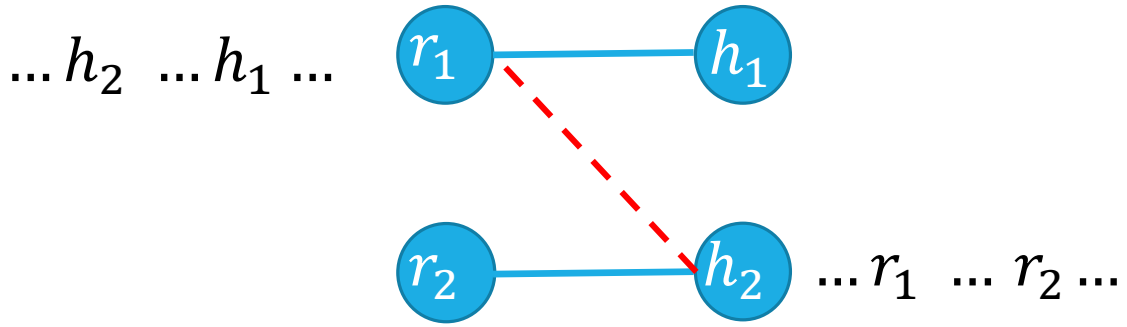
Suppose (for contradiction) that (r_1, h_1) and (r_2, h_2) are matched, but

r_1 prefers h_2 to h_1 and

h_2 prefers r_1 to r_2

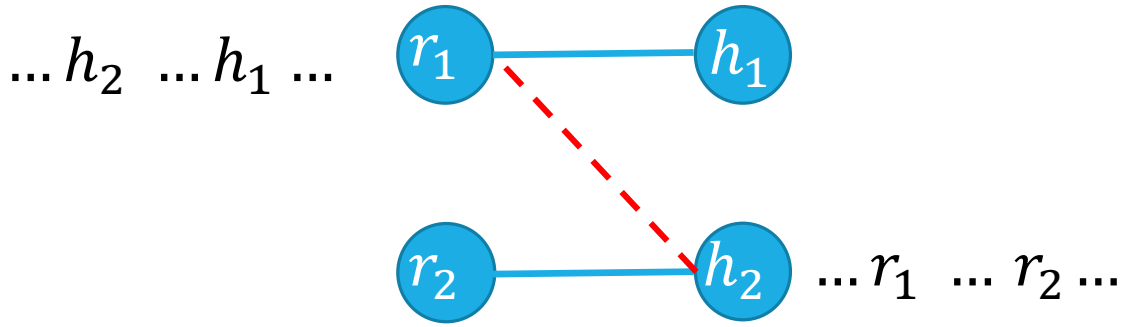


Claim 4: The matching has no blocking pairs.



How did r_1 end up matched to h_1?

Claim 4: The matching has no blocking pairs.



How did r_1 end up matched to h_1?

They must have proposed to and been rejected by h_2 (since riders propose down their list in order – Observation A).

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

If h_2 ever changed matches after that, the match was only better for it, (since horse's partners can only get better for them -- Observation C) so it must prefer r_2 (its final match) to r_1.

But r_1 is before r_2 on h_2's list. That's 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

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.

We say that h is a feasible partner for r if there is at least one stable matching where r and h are matched.

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

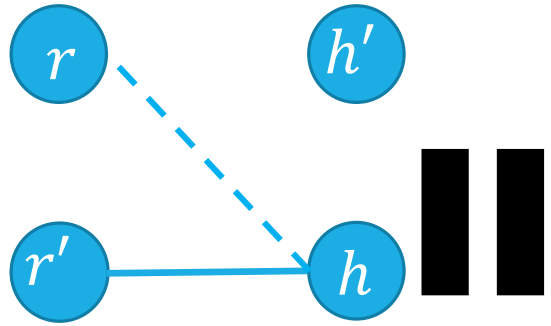
Intuition

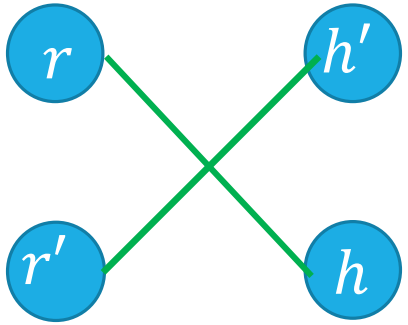
This isn't a full proof (coming in a second)

Suppose r is not matched to their favorite feasible partner, h. It has to be rejected by h during the algorithm (otherwise r matched to h or better). How did that happen? Well h had to have a better offer. But what's the source of that better offer? Call them r'.

There's some stable matching where (r, h) are matched, and r' is matched to some h'. For stability h prefers r to r' or r' prefers h' to h.

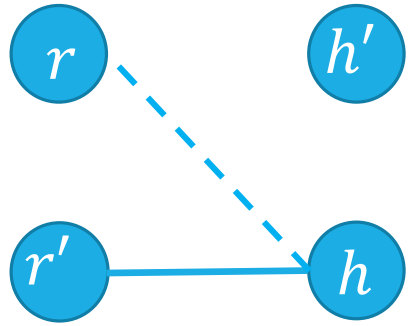
Must be the second, so r' was also already rejected by a feasible partner.





Proposer-Optimality

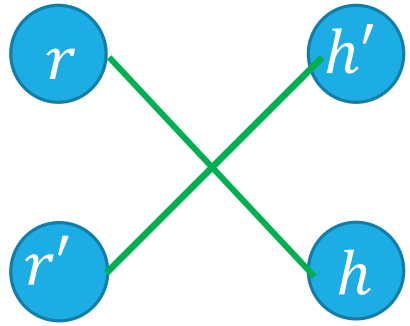
Every member of the proposing side is matched to the favorite of their feasible partners.



Intuition:

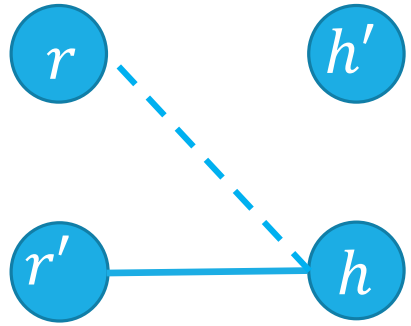
The riders start at the top of their lists. For the claim to be false, some rider r has to be the first to be rejected by their favorite feasible horse, h.

When that happens, h says it prefers some r' (and it does that while r' is still in the "favorite feasible partner" or "too good for you" sections of their list). So r' and h would block any matching



Proposer-Optimality

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



Let's prove it – again by contradiction

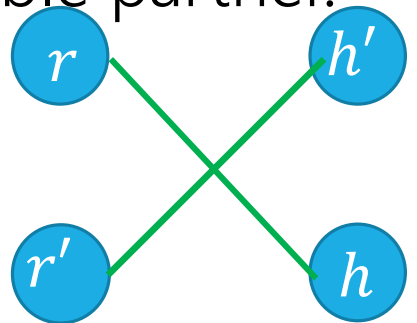
Suppose some rider is not matched to their favorite feasible partner.
Then some r must have been the **first** to be rejected by their favorite feasible partner, h. (Observation A)
And there is an r' that h (temporarily) matched to causing that rejection.

Since r and h are feasible for each other, there is some stable matching (call it M') where (r, h) are matched. The rider r' is matched to some horse 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'!



Contradiction With Extremality

We used a trick to make the proof nicer.

Instead of saying "r' was already rejected by a feasible partner, let's go back and analyze **that** rejection." And repeat the argument over and over, we jump straight back to the first rejection.

If in your proofs, you're saying "repeat this step until…" you can probably make a cleaner proof with this trick.

Another example of this trick is coming in section 2.

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 in our pseudocode 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 another tricky proof-by-contradiction, but very similar to proposer-optimality), 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, or choosing which parts of a problem to model and which ones to ignore, think about everyone involved, not just the people you’re optimizing for; you might not be able to have it all.

Takeaways

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.

When doing a proof by contradiction, it sometimes helps to analyze the first time something happens instead of just some time where it happens.