Stable Matchings

CSE 417 22AU
Lecture 2

Stable Matching Problem

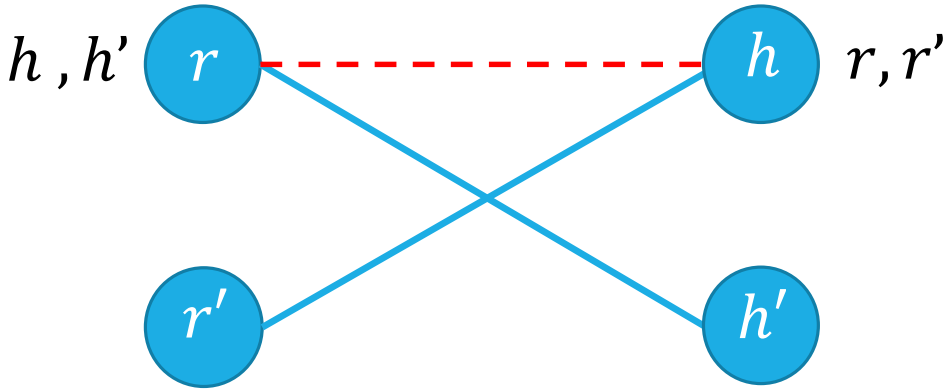
Given two sets $\mathrm{R} = \{r_1, \ldots, r_n\}, H = \{h_1, \ldots, h_n\}$

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

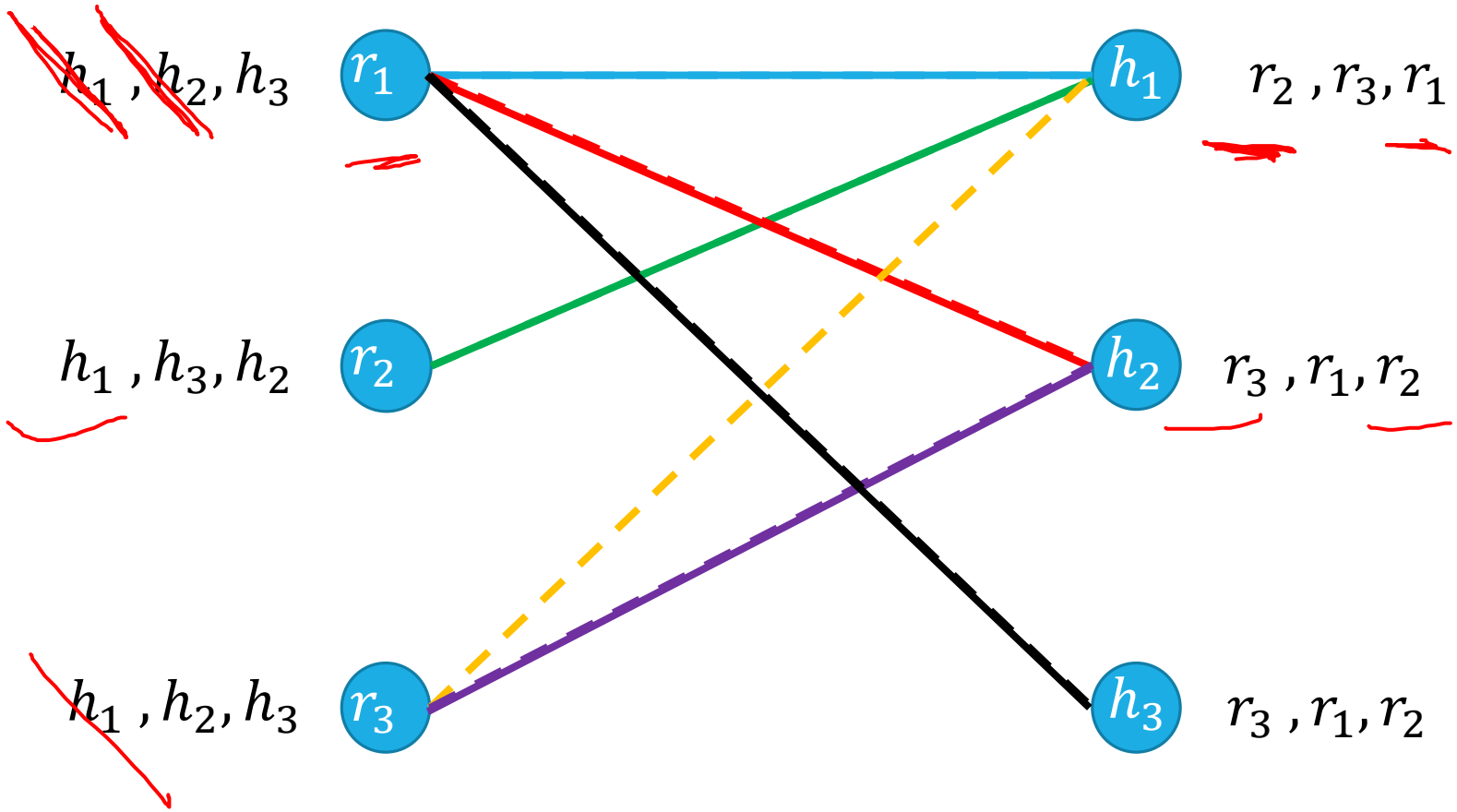
suppose (r', h) are matched

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?

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, but we need a detour first…

How do we convince someone of a fact about our algorithm?

In 373 we usually used tests to convince ourselves our algorithm worked. But that's only helpful if we actually write the code.

You'll write code in this class, but you'll do at least as much writing of algorithms in "pseudocode." There's no auto-runnable tests for pseudocode.

Besides tests don't test everything.

What we want is to convince another person "if you implemented this algorithm, it would pass the tests you write."

Even if they make slightly different decisions than you would…

Different Decisions?

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)

Our justification needs to work regardless of what data structure keeps track of free r's.

Our justification needs to work for any language or design choice (are r, h objects?)

Our justification needs to work no matter what preference lists the r and h have. That would be a lot of tests.

For this class, our hope is that the algorithm ideas will be useful in other contexts (with doctors and hospitals or TAs and courses not just horses and riders). The more general we keep our analysis, the more likely it can be applied to new contexts!

How do we show facts about our algorithms?

Usually we'll formulate our facts as implications:

An implication is a promise – a statement of the form "if X, then Y"

How do we explain why a fact is true?
Method 1 (direct proof): Assume X is true, then explain, step-by-step, by Y must be true too.

You can also assume the input is valid (e.g., we have n horses and n riders, all the lists are valid, no `null`s,…)

But you shouldn't assume anything else – don't (just) check a particular example. We want this explanation to work for people in lots of contexts!

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)

How many iterations? Well when is there not a free r anymore?

In 373 we'd figure out:
How many steps there are per iteration
How many iterations we need
Get the running time (with a summation or multiplying)

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.

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.
`Horse[i][j]` gives the identity of the j^{th} rider on horse i's list.

What data structures should you use?

Initially all r in R and h in H are free

stack

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
Go to pollev.com/robbie and login
with your UW identity

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!

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 .

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

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

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

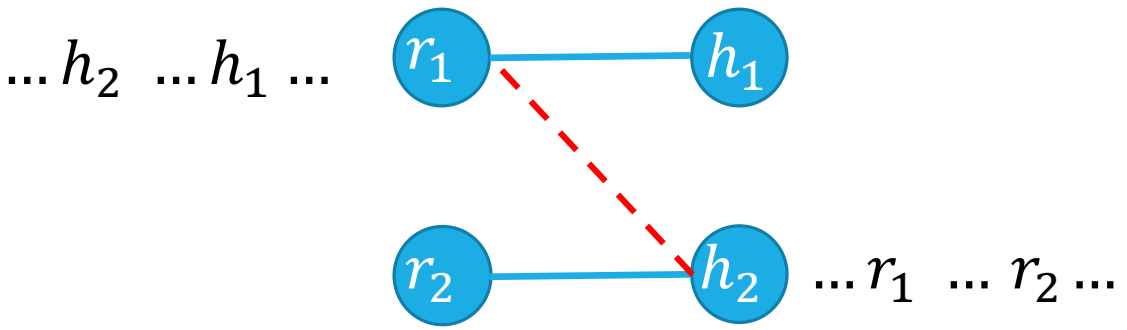
there is no blocking pair.

That's a good sign for proof by contradiction.

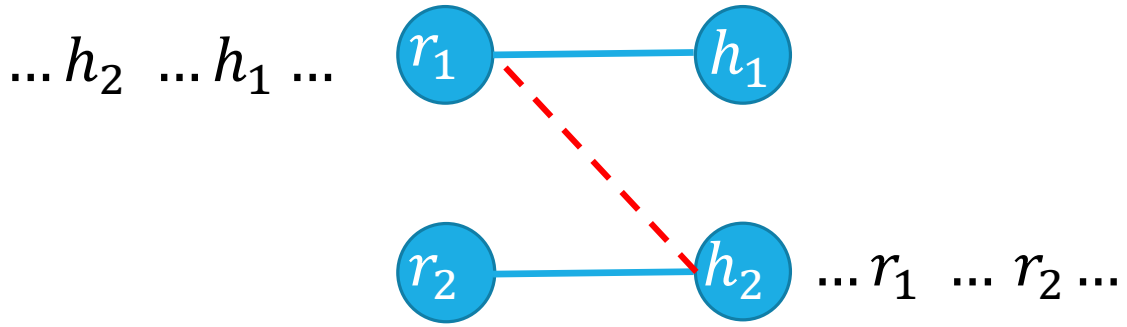
Suppose (for contradiction) that (r_1, h_1) and (r_2, h_2) are matched, but

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.