ORIE 5355: People, Data, & Systems

Lecture 7: Recommendations – from predictions to decisions

Nikhil Garg

Course webpage: https://orie5355.github.io/Fall_2021/

Last time: Prediction (filling in missing entries)

	Avatar	LOTR	Matrix	Pirates
Alice	1		0.2	
Bob		0.5		0.3
Carol	0.2		1	
David				0.4

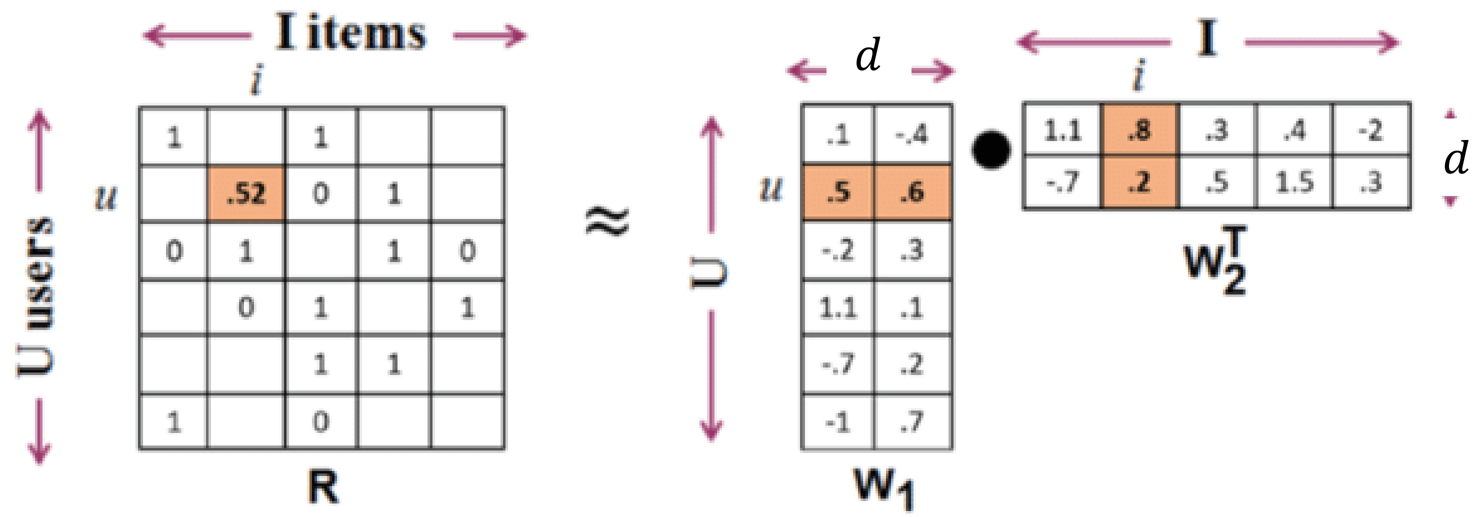
Matrix factorization – "Latent factor" models

Once we have $u_i \in R^d$ for each user, $\mathrm{v_j} \in R^d$ for each item

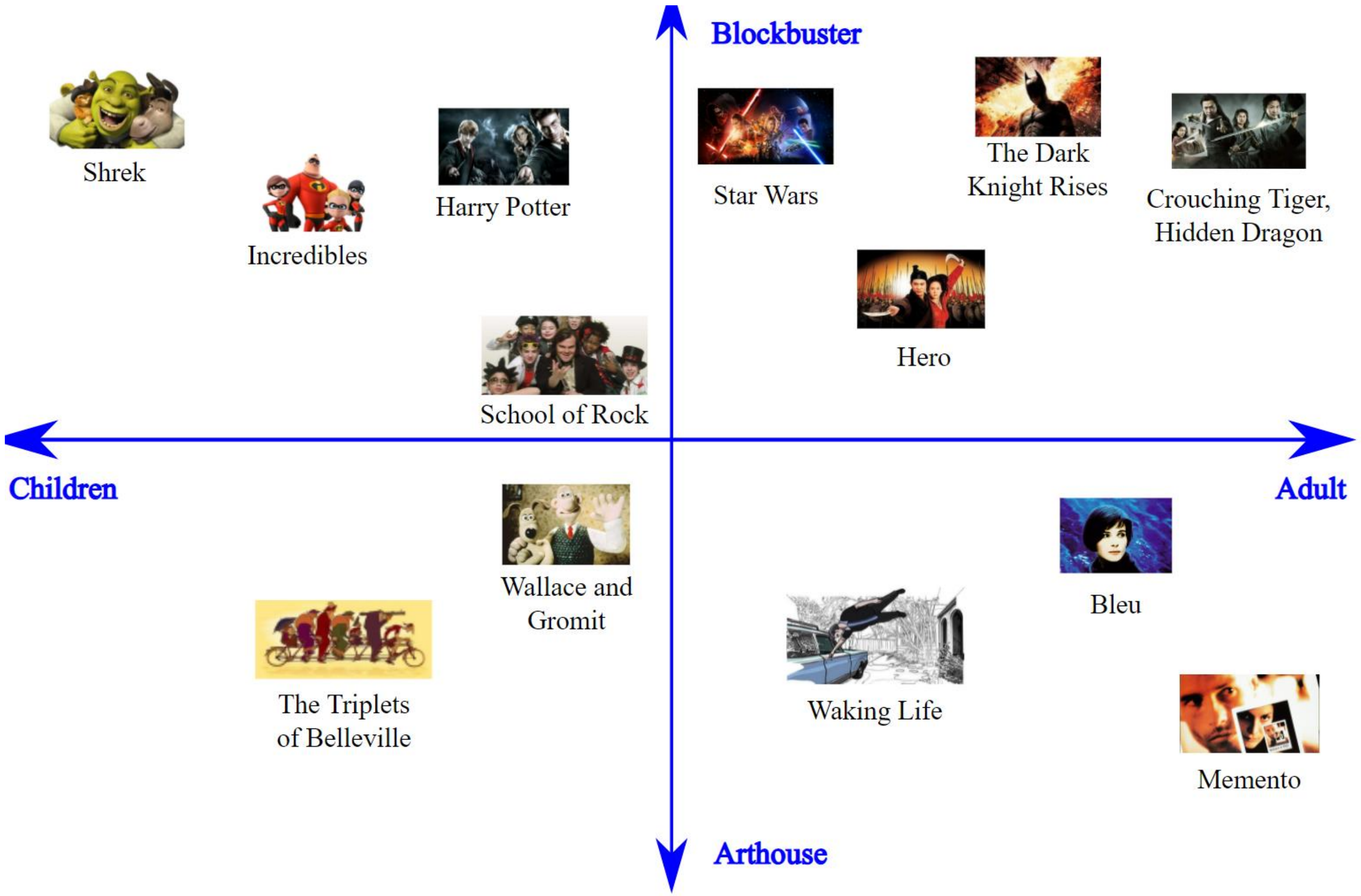
Such that $u_i \cdot v_j \approx \widehat{r_{ij}}$ (the rating user gave to the item in the past)

Then, for every pair of items and users that have not been rated:

Set predicted rating $\mathrm{r_{ij}} = u_i \cdot v_j$



Example vectors with d=2



Embeddings | Machine Learning Crash Course | Google Developers

Matrix factorization: Pros and Cons

+: Don’t need to guess at what features matter

–: Need historical data about each item and user

–: Hard to provide explanations

In practice, matrix-factorization-based methods (and modern deep learning successors) are used when you have enough data

“Cold start” with matrix factorization

- Chief challenge in many settings: you don’t have (a lot of) historical data on some new users or new items
- Idea: Combine matrix factorization with content- and user- similarity based approaches

  Step 1: Train matrix factorization model with dataset

  Step 2: For new users [items] find “nearby” users [items] to them and *initialize* their vector using the nearby users [items]

  i.e., pretend their vector is the same as those of nearby users

  Step 3: Over-time, *update* their vectors using their own history

- Determining “nearby” items: must use data like genre and demographics
- Key idea in many settings: At first without individual data, pretend someone is like the “average” user. Then with more data, start doing personalized things

Step 2: Vectors from "nearby" users

Suppose we have a demographic vector for each new and old user:

[age, ethnicity, gender, income, …]

- Simple: K nearest neighbors
  - Define a distance function on the vector of demographics
  - For each new user, find the K closest old users and average their vectors
  - Challenge: defining the distance function!
- Also simple: train matrix factorization with known user vector
  - Instead of learning vector $u_i \in R^d$ for each user, $\mathrm{v_j} \in R^d$ for each item
  - Set u_i to the demographic vector, and just learn $\mathrm{v_j} \in R^d$ for each item
- Many other approaches:

  Train a model using the demographics to predict u_i^k , each dimension k of u_i, using all the old users

Questions on prediction?

What to *do* with predictions? Naïve method

Train a single matrix factorization model using some data (what data?)

→I have predictions for each item and each user

For example, predict $\mathrm{r_{ij}} = u_i \cdot v_j$

For each user i, simply recommend the best item

$$\mathrm{argmax}_j \, u_i \cdot v_j$$

(Or K best items):

$$\mathrm{argmax}_{j_1 \dots j_K} \sum_{\ell=1} u_i \cdot v_{j_\ell}$$

Issues with naïve method

- Capacities

  What if you only have 5 of item j, and everyone likes item j?

- Multi-sided preferences

  Recommendations in freelancing markets (workers matched with clients), dating apps, volunteer platforms, etc

- Challenges in recommending *sets* of items
  - *Diversity* of items recommended
  - Behavioral effects? Recommending one item makes another item more popular

Today: going from predictions → recommendations

Dealing with capacity constraints

Overview

- What’s the challenge, exactly?
- Solving an “easier” problem: “maximum weight matching in a bipartite graph”
- Insights from the easier problem to real-life applications

The challenge

- In many (non-online-media) settings, you are recommending “items” with capacity constraints:
  - You have a finite number of each item in your warehouse
  - An AirBnb can only be booked by one customer at a time
  - Workers can’t work for every client; a client can only hire 1 person
  - People on dating apps – can’t talk to everyone
- If you ignore these capacity constraints, then everyone may be recommended the same (limited) item

  Some people will be left out
- (How) should you factor in capacity in your recommendations?

The challenge, formally (simple version)

- You have N users and M items, but only 1 copy of each item
- You want to recommend 1 item $\mathrm{j(i)}$ to each user
- Each user i will consume the that you recommend them
- You want to maximize the sum of predicted ratings of consumed items

$$\sum_i \mathrm{r}_{ij(i)}$$

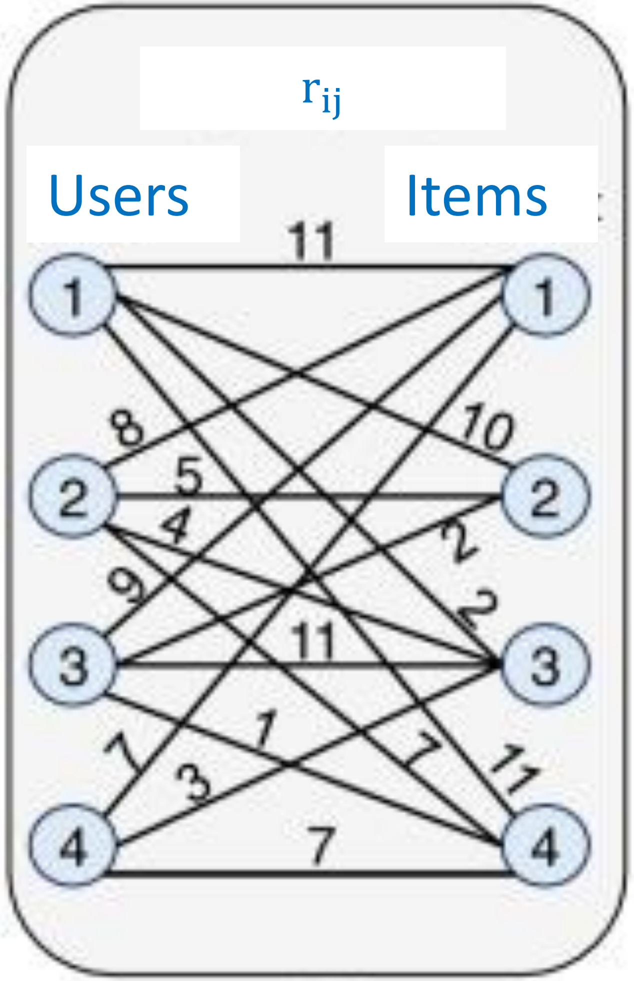
- However, each item can only be recommended once

$$j(i) \neq j(i') \text{ unless } i = i'$$

Solving the simple case

It turns out that this simple case is called “maximum weight matching”

Draw a graph with users on one side and items on the other



[OSA | Simulation and FPGA-Based Implementation of Iterative Parallel Schedulers for Optical Interconnection Networks (osapublishing.org)]

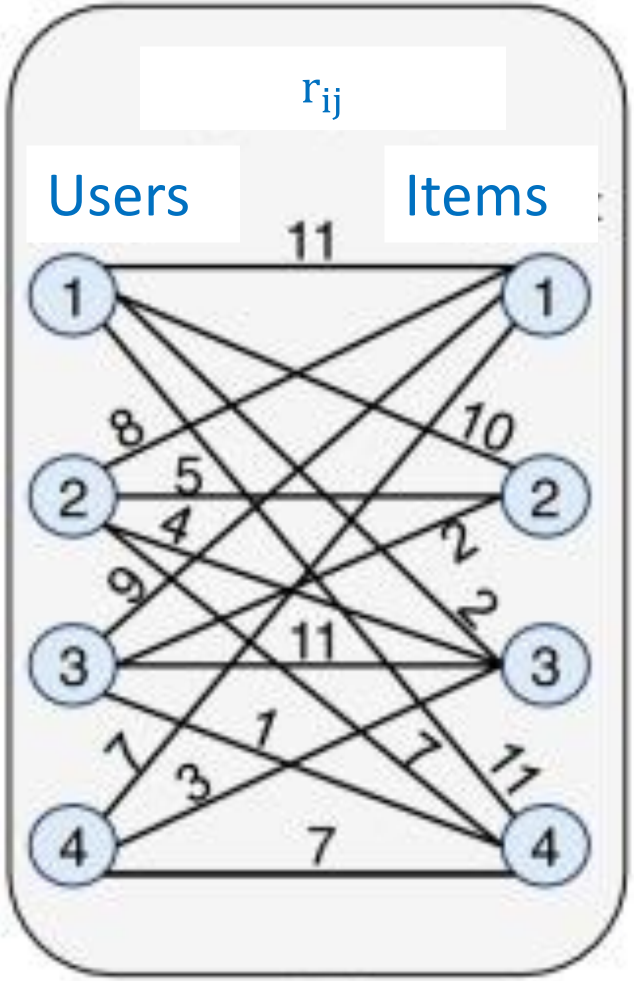
Solving the simple case

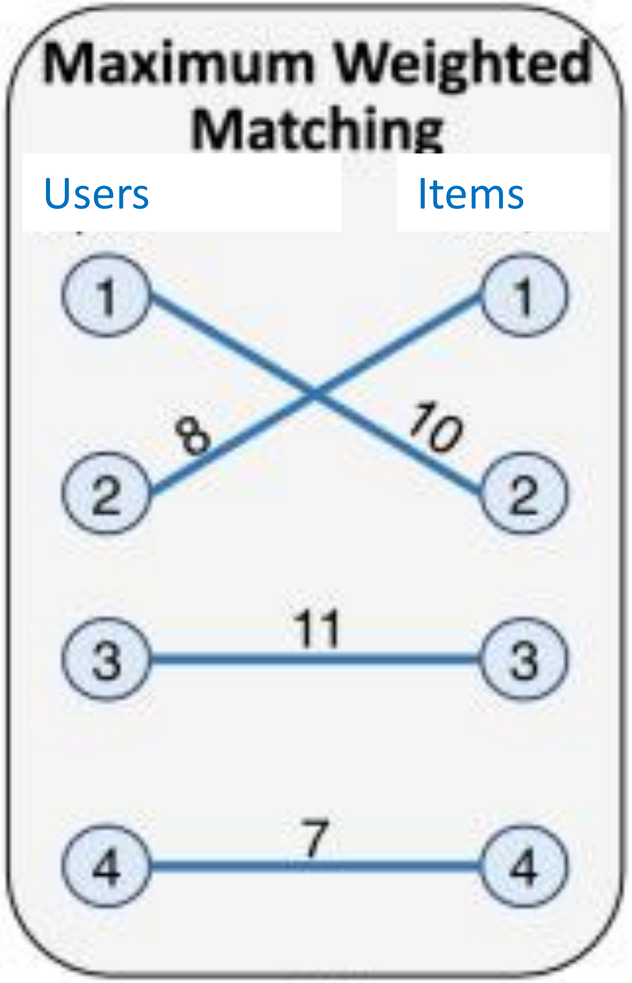
It turns out that this simple case is called "maximum weight matching"

Draw a graph with users on one side and items on the other

Find the "maximum weight matching"

[scipy.optimize.linear_sum_assignment — SciPy v1.7.1 Manual]



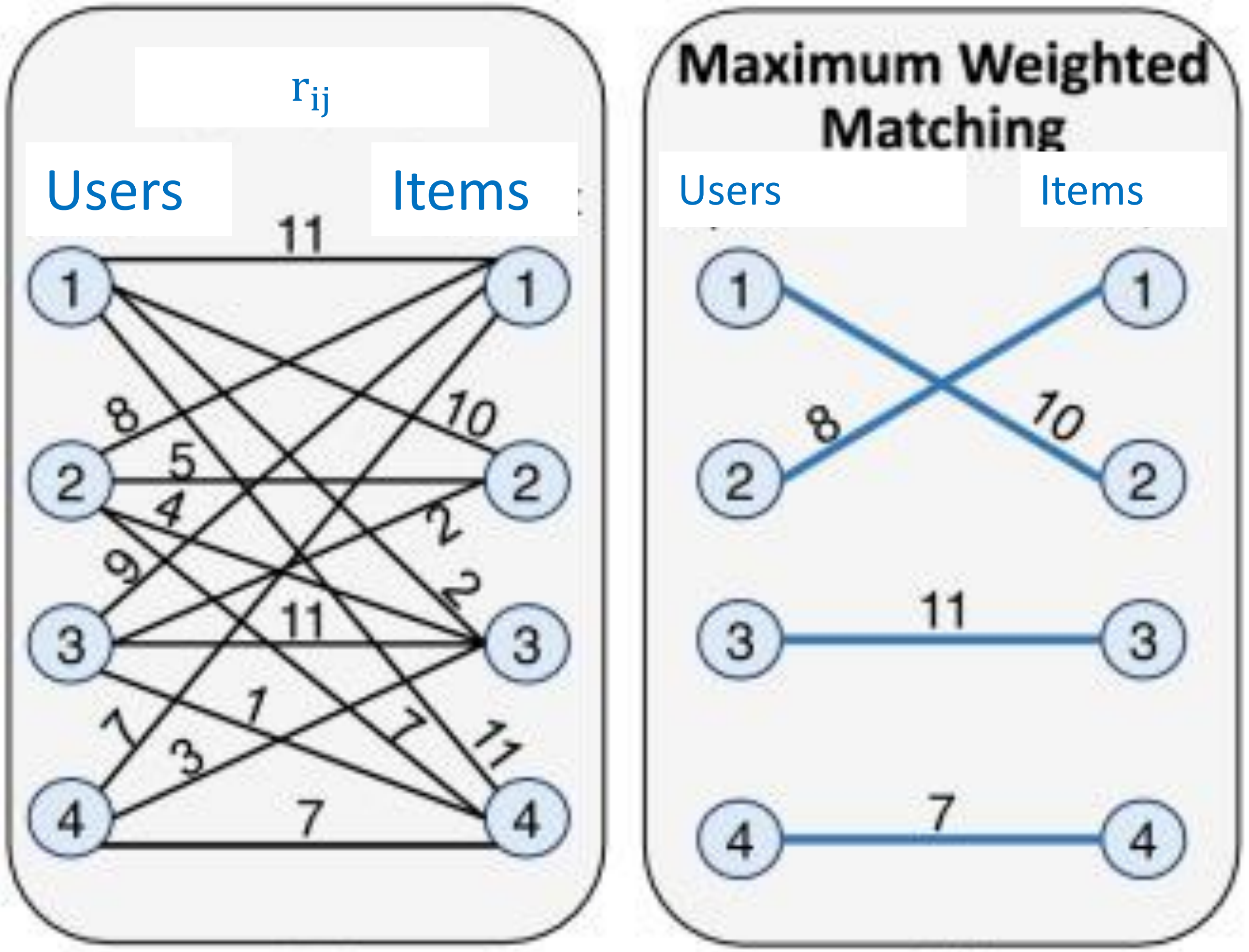


[OSA | Simulation and FPGA-Based Implementation of Iterative Parallel Schedulers for Optical Interconnection Networks (osapublishing.org)]

Insights from the simple case

In general, the actual solution might be combinatorial – a complex function of all the joint preferences

- Some *users* are not matched with their most preferred item!
- Some *items* are not matched with the user that likes it the most!
- If a user likes multiple items similarly, maybe they get their 2nd choice
- If only 1 user likes some item, make sure that item and user are matched



[OSA | Simulation and FPGA-Based Implementation of Iterative Parallel Schedulers for Optical Interconnection Networks (osapublishing.org)]

Challenges in using max weight matchings

- Everyone doesn't show up at once

  New users come in tomorrow – have to leave items for them
- You can't "match" people, only recommend them items

  Someone may not consume the item!
- "Capacity" constraints are also soft
  - New items are shipped to warehouse all the time
  - Maybe you can spend more money to expedite shipment
- Computational constraints in rerunning large scale max weight matchings with every new user

What to do in practice

- Finding an "great" solution requires a lot of careful data science + modeling work
- Some reasonable heuristics:

  "Batching": If you don't have to give recommendations immediately, wait for some number of users to show up and solve max weight matching (for example, every hour)

  "Index" policies: For each user, create a "score" for each item and just choose recommend the item(s) with the highest score(s)

Index policies

- We want a score (index) between each item j and user i: $\mathrm{s_{ij}}$
- Then, for each item, pick the item with the max score: $\mathrm{argmax}_j\ \mathrm{s_{ij}}$
- We've already seen an example: if the only thing that matters is predicted rating, then $\mathrm{s_{ij}} = \mathrm{r_{ij}}$
- Why index policies?
  - They're efficient: for each user, only need to consider their scores
  - They can be *explained* to users
  - All information about other users is contained in how score is constructed

Constructing index policies

What matters in constructing an index policy?

- The higher the ratings by other users for an item, the smaller $\mathrm{s_{ij}}$ should be
- The less capacity C_j left for the item, the smaller $\mathrm{s_{ij}}$ should be

An *example* score function

$$\mathrm{s_{ij}} = \alpha_j \left[\frac{r_{ij}}{\overline{r_j}}\right] C_j^{\beta}$$

where α_j, β are some (learned) parameters over time

α_j: Item is "special" and should be over-recommended

β : Relative importance of capacity. ($\beta = 0$ means ignore capacity)

Many possible score functions! Should be application specific

Capacity constraints lessons

- If you just recommend each user their highest predicted scores, then you might not be *globally* efficient
- Even if you can’t implement it, taking intuition from the “optimal” solution is often valuable
- Index policies: even if “optimal” solution requires combinatorial constraints, “practical” solution can decompose the problem

Multi-sided preferences

Multi-sided preferences

- In many modern online markets, both sides have preferences
  - Freelancing markets (workers matched with clients), dating apps, volunteer platforms, etc
- A match only happens if *both* sides like each other
  - And have capacity…

The challenge, formally (simple version)

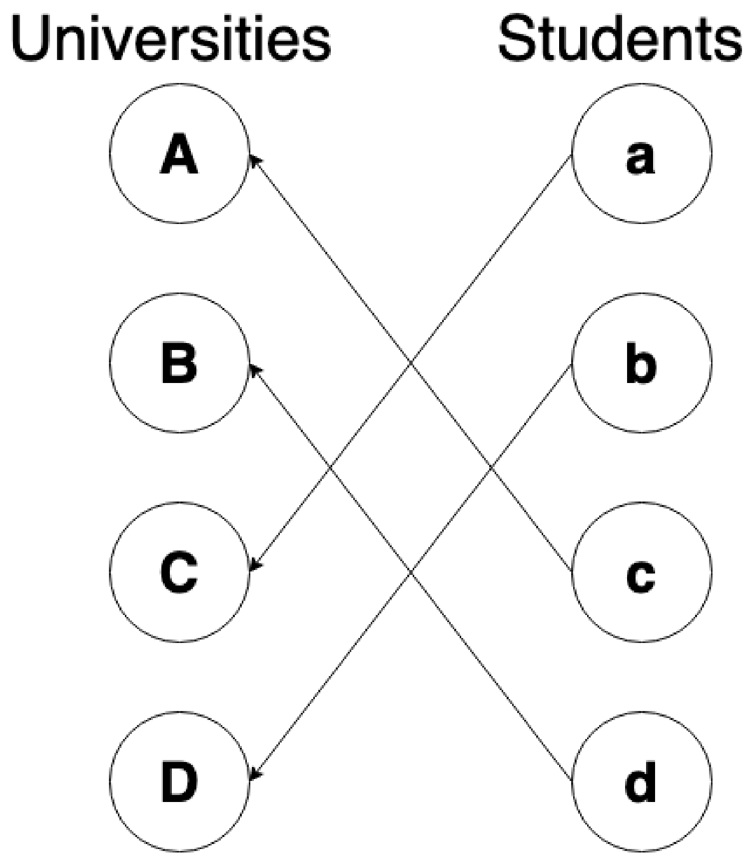
- You have N workers and N clients
  - Each worker can only work with 1 client; each client only hires 1 worker
- Each side has preferences (predicted ratings) over the other side
- You want to create “good” matches
  - Good for who? Workers? Clients? Some combination?
- Easier goal: create “stable” matches

“Stable matching” in 1 slide

- Stable matching:
  - Given rank order preferences from each person on each side
  - Match the sides such that matches are “stable”: No potential pair wants to abandon their current partners for each other.
- Efficient to find: “Gale-Shapley algorithm”
- Used to allocate:

  Medical students to residencies

  Students in NYC to high schools



Challenges in using stable matching

Same as from using maximum weight matchings

- Everyone doesn't show up at once

  New users come in tomorrow – have to leave items for them
- You can't "match" people, only recommend them items

  Someone may not consume the item!
- "Capacity" constraints are also soft
  - New items are shipped to warehouse all the time
  - Maybe you can spend more money to expedite shipment
- Computational constraints in rerunning large scale stable matchings with every new user

Just more complicated with both sides now having preferences

Intuition from stable matching to recommendations

What matters in constructing an index policy?

- The higher the ratings by other workers/clients, the smaller s_{ij} should be
- If either worker i or client j has been recommended to many other people in the past, the smaller s_{ij} should be
  Equivalent of "capacity"
- Now, *both* i's rating for j and j's rating for i matter

An *example* score function

$$s_{ij} = \alpha_j \alpha_i \left[\frac{\min(r_{ij}, r_{ji})}{\overline{r_j}\,\overline{r_i}} \right] C_j^{\beta} C_i^{\beta}$$

Announcements

- Guest lecture Amy Zhang on Monday 9/27
  - Regular class-time
  - **Remote only – please log in using zoom [Not from classroom]**

Questions?