[SQUEAKING]

[RUSTLING]

[CLICKING]

**ZACHARY ABEL:** Good morning. Welcome back. Today is Lecture 12. We are continuing our Graph Theory unit, this time talking about matchings and stable matchings. Matchings, in general, are a really useful concept in graph theory. They come up frequently when you're trying to match pairs of objects. You want everyone to have one partner or possibly zero partners. If they don't get a match, then they're sad. That's fine. But matchings are a really useful concept, so let's talk about what they mean, what they're good at, and a few ideas surrounding these.

So first of all, definition. A matching in a graph-- which, remember, has a set of vertices and a set of edges which are pairs of vertices-- is a subgraph, M-- and what subgraph means is we're just taking a subset of the vertices and a subset of the edges, so long as all the edges have their vertices in the subset-- in which every vertex in V has degree less equal to 1 in M. I'm sorry. This is vertex set of our subset.

Said differently-- a different definition which is just as good-- a matching is a subset of the edges of our graph where no two selected edges share an endpoint. So we're just choosing some of the edges of our graph, making sure that we never touch edges that have an endpoint in common.

For example, I can draw this bow-tie-looking graph, this, this, this, this, this, here, here, here, and here. And then vertex a, b, c, d, e, f, g, h. So if this is our graph, let me give you an example of a matching. So an example of a matching is M. This is going to be just the edges be and dg.

This is an alternative notation for edges, by the way. Instead of writing the set b comma e close curly brace, I can just write be next to each other and it's clear we're talking about those two vertices. So we're dealing with the edge, be-- so that's this edge-- and the edge, dg-- which is this edge. Is this a matching?

I see some head nods. Yeah, this is a matching because none of our selected edges have endpoints in common. If I had also chosen edge fd-- if I'd tried to put fd into there as well-- then these two edges would be in conflict and we would no longer have a matching. Does that make sense?

So said more similar to our first definition, if we look at just the subgraph that includes our chosen edges, all of the vertices in that subgraph need to have degree at most 1. So this has degree 1 and 1 and 1 and 1 and 0s for the others. If we have edges sharing endpoints, we'd have higher degree, which is not allowed in a matching.

Like I said, this has lots of applications anytime we're trying to pair things up, for example, online dating services. You're trying to recommend, say, this user to that user and you recommend them to each other and they go on a date or something like that. And maybe you want to try to match up a bunch of them at once so you can have lots of dates happening in parallel. So you want a matching that includes lots of edges, knowing that no one can go on multiple dates at once.

Sometimes you might have a bunch of tasks in an expensive computational project. You have lots of tasks you need to do and lots of servers that can help you out in parallel, but each server can only do one job at a time. So you want to find a matching that gets lots of servers working on different tasks at the same time. So you want to find a matching that includes lots of your servers and lots of your tasks and find a matching with lots of edges so you get lots done in parallel.

So in a lot of these examples, we're trying to maximize the number of edges in our matching, try to get lots and lots of pairings all at once. And there are a couple of different notions of big, a big matching, a matching with lots of edges that we want to define. We want to define a maximal matching and a maximum matching. And I'm sorry that those sound very similar. But let's talk about what they mean.

So we'll say that M is a maximal matching when it's not possible to find another edge that's in our set but not yet in our matching such that M with that new edge is matching. So maximal means you can't add new edges to the set of edges you already have.

So in this case, with these two edges, all of the other edges I might try to add to these two have an endpoint in common with one or the other. So I can't pick new edges to put into M while keeping the ones I already have. So that makes M, this matching is maximal. Does that make sense? This is a local notion where all you have to do is look at the matching you have and ask is there any single other edge that I can put in here as well.

And if there isn't, then it's maximal. And if it's not if there is another edge you can put in, well, go ahead and put it in, then search again. If there's yet another edge you can put in, put it in. Keep going until there isn't, and then you have a maximal matching. So it's very easy to find a maximal matching. You just keep adding edges until you can't.

The difference is that that maximal matching you find might not be the biggest matching possible ever. And that's the other notion we want-- a maximum matching. We'll say that M is a maximum matching when there does not exist a bigger matching, a matching M prime with more edges.

So this is a global measure of maximum. We'll say a matching is maximum when it's the biggest you can ever do. I don't care if you keep these particular edges. Just is there any matching at all anywhere that has more edges than this? So that's the difference between maximal and maximum. Maximal is about local search-- can you make a single improvement by adding a single thing here? Maximum is a global property-- is it the biggest possible ever? And as you can imagine, finding a maximum matching or proving that a matching you have is maximum is a lot harder than maximal.

This matching up here with dg and be, is that maximum? Why not? Someone shaking their head, can you tell me? Yes.

**AUDIENCE:** It may be bg and ce.

**ZACHARY ABEL:** All right, ad, bg you said, and ce?

**AUDIENCE:** Yes.

**ZACHARY ABEL:** All right. Is a matching. And is this a matching? Let's take a look. We've got ad here, bg here, ce here. None of those circled edges-- they kind of look like coffee beans to me-- none of those coffee beans share an endpoint, so that is a matching. And since this has more edges than M up there, M is not maximum. All right? Cool. So matchings can be maximal-- locally biggest-- even if they're not maximum-- globally biggest. Is this matching maximum, or can we do even better? Oh, there was someone raising their hands in the back, but it's Brynmor.

[LAUGHTER]

There was someone else raising their hand, but I think they were just adjusting their glasses. All these fake-outs playing jokes on me. OK, so is this a maximum matching, or can we do better? Yes.

**AUDIENCE:** For it to be better, we'd have to add vertices, either f or h. And since they both share d with a and c, it makes sense that we can't do that because the degree of d and c would be bigger than 1.

**ZACHARY ABEL:** All right. So let me say that so everyone else can hear. So if we had a bigger matching, we'd have to include f or h. Is that what you said?

**AUDIENCE:** Yes.

**ZACHARY ABEL:** All right. And if we try to include f, then it's going to share an endpoint with ad, which is not great. So yes, that's almost exactly the logic I'm looking for. Just a slight tweak to that. When we're asking if a matching is maximum, we're not really talking about adding to this specific matching. That's the maximal side of things.

So instead of comparing to what we have to this particular matching, I just want to be able to prove that there is no bigger matching at all. So what would it mean for there to be a bigger matching? A bigger matching would include, well, at least four edges, and, therefore, at least eight vertices. But we only have eight vertices, right? 1, 2, 3, 4, 5, yep. So it would include all the vertices.

And any matching that includes all the vertices, in particular, includes a and f. And we can't use a and f because they would both have edges to d. There's only one way to use a. There's only one way to use f. And they both use vertex d in the middle, and we can't do that. Can't use a and f at the same time. And this is our contradiction. So if there's a bigger matching, bad stuff happens. So there doesn't exist a bigger matching.

That doesn't mean these three coffee beans are the only maximum matching. There are a couple more. For example, we could replace ad with fd. That would be fine. That would be a different maximum matching. So there can be multiple maximum matchings that all necessarily have to have the same size. Different maximal matchings don't all have to have the same size. As we saw, this two-edge matching is maximal, and also this three-edge matching is maximal. Does that distinction make sense? Wonderful. All right.

Let's see. So next definition. A matching is perfect if and only if it includes all vertices. So a perfect matching gives everyone a partner and, therefore, exactly one partner. Does this graph have a perfect matching?

No. A perfect matching would need all eight vertices and, therefore, four edges. But we already saw there are no matchings with four edges in this graph. So not all graphs are going to have perfect matchings. But often in your task allocation problems where you're trying to match everyone to another resource, ideally, you'll end up with a perfect matching where everyone gets a partner or every task gets a server. And so perfect matchings are very commonly the goal.

Now, so far we've been talking about matchings in general graphs. But matchings are very common, especially in bipartite graphs, so bipartite matchings. So when we're trying to match tasks to servers, we have a bipartite graph. All the edges connect a task to a server when it's possible for that task to be done on that server. And then we're looking for a matching, which tells you which servers are doing which tasks. So a lot of problems that involve matchings are naturally bipartite.

Let's see. And to emphasize, remember from last time, bipartite graphs don't need the same size on both sides. As we saw with the Harvard versus MIT bipartite graph, we had 7,000 on one side and 4,000 on the other side.

Still a perfectly valid bipartite graph, however perfect matching can't possibly exist in a bipartite graph when the two sides have different sizes. There can never be a perfect matching between Harvard undergrads and MIT undergrads because there are more Harvard undergrads than MIT undergrads.

Every edge includes one of each. So even if we max out all of the MIT students and give them a Harvard buddy, there are still going to be 3,000 Harvard students that lose out. So if there's a perfect matching in your bipartite graph, you have to have the same size. Doesn't mean there will always be a perfect matching when the two sides have the same size, but it's at least necessary. All right?

Sometimes when you're trying to make these matchings, not only are you trying to maximize the number of pairs you get, or possibly get a perfect matching and get everyone a pair, but often edges are weighted. So each edge comes with a number. So we have a function w from the set of edges to real numbers.

And each edge comes with a number. You can imagine in the task versus servers graph, each edge represents how long that task would take on that server. And maybe certain tasks are going to be faster in different servers maybe because that server is more powerful, or maybe because it's closer to other resources it might need and it can access them more quickly. But often when we have a weighted edge like this, we want a perfect matching with minimum total weight.

As an example of that, here's a nice bipartite graph. And maybe this edge has weight 3. This has weight 1. This has weight 5. And this has weight 2. And I can see two different perfect matchings in this graph. There's the one that does this. And there's the one that does this. I think that's all of them.

And if we look at the weights, well, this has weight 3 and 2, so this has total weight 5. And this has weight 1 and 5, so this has total weight 6. So you'd prefer this one to that one because 5 is smaller than 6. The total weight in this perfect matching is better than the total weight in this perfect matching. So these are just very common kinds of questions that you find yourself asking when you're modeling something in terms of matchings or perfect matchings. Often, there's more to optimize than just finding a perfect matching.

And I mentioned all of these variations on these problems because unlike last time where we saw that graph coloring is a useful abstraction but hard to compute exactly or hard to compute optimally, all of the matching problems I've mentioned, we do have efficient algorithms for. So we have we can find maximum matchings in graphs, bipartite or otherwise. Yes. Question.

**AUDIENCE:** A question about the weights. Are they just arbitrary? Like did you just pick those, or did they come from somewhere?

**ZACHARY ABEL:** Oh. Yeah, where did these weights come from? These are just arbitrary in this example, I just picked them for demonstration. In a real-world scenario, the weights would represent how much work it takes to do the task corresponding to this pairing, or how expensive this is, or how desirable. So usually, these weights are going to come from whatever context you're modeling and trying to optimize for. But yeah, in this example, I just picked them.

But there is an algorithm to find maximum matchings in graphs, bipartite or not. Doesn't matter. And because we can find maximum matchings, we can find perfect matchings if they exist. If a perfect matching exists, it's certainly maximum because you can't use more than that number of edges. We can find min-weight perfect matchings in bipartite graphs.

We have algorithms to solve these kinds of problems. So if you ever find yourself modeling a question with matching, trying to find a matching with certain properties, very often, there will be algorithms to solve it quickly. We're not going to cover those algorithms in this class. You'll see more of them in the follow-up classes. 6.121 and 122. But just know that they're out there and know that this is a useful tool, both for modeling and for computing.

Are there questions about these kinds of things? All right. No questions. Then let's move on to the main topic for today, which is stable matchings. So the topic is stable matchings.

In this problem, we're going to be looking for certain kinds of perfect matchings in bipartite graphs. So in the specific scenario, let's imagine we have n applicants and n evaluators. You can think of these like n interns that are looking for managers, and n managers where we're going to match them up one to one-- each intern gets one manager and vice versa-- and we're trying to find a way to match them nicely.

You can think of these as the applicants are recently graduated medical school students looking for residencies. And I mentioned this example specifically because the algorithm we're about to talk about is used in precisely that way. In practice, in the real world, in the United States at least, all residency applicants submit their list of preferences about which hospitals they want to do their residencies at. And they do interviews, and hospitals submit their preferences for which candidates they prefer.

And they send this all to a national nonprofit organization who does a big global computation to find, in some sense, the best perfect matching it can. And then it notifies everyone on the same day. All the residency applicants are told on Match Day which was actually just a few weeks ago. It happens once a year. They're told where they're doing their residency. And it's a big party, and many of them are excited about going at their top school. Many of them don't get their top school, and they're still happy because they still get to be a doctor.

But they're going to use exactly the algorithm I'm about to describe with some tweaks because the real world is more complicated than the simple math we're about to describe. But this is a real algorithm with real consequences. It was invented in 1962 by Gale and Shapley. That's two people. I'm going to call it the Gale-Shapley algorithm.

It was also improved many years later by Al Roth. Together, they have found many real-world applications and improvements. And it won them the 2012 Nobel Prize in Economics for Al Roth and for Shapley, I don't know where Gale was in all of this. I don't think he got the Prize as well. Anyway, really important, useful, practical algorithm that actually has some fairly simple mathematical underpinnings, enough that we can describe it and analyze it and prove things about it in the next hour we have together.

So what is this algorithm? What is it for? So as I said before, we have n applicants and n evaluators. And each of them is going to give a preference ordering for all n of the opposite parties in order of their preference. So all applicants and evaluators give a full ranked list of their preferences.

So as a quick example-- trying not to run too fast for the cameramen back there-- so here's an example where we have five applicants and five evaluators. I haven't said, which because the relationship is pretty symmetric at this point. And so each person on the left gives a full ranked preference order for all of the people on the right. So this person A prefers H, then J, then F, then G, then I in this strict ordering.

So each of these gives some permutation of F through J. Each person over here gives some permutation of A through E with no restrictions on the kinds of rankings that are allowed, as long as they give a full strict ranking of all the people, no other restrictions. If all the applicants want to choose the same evaluator as their first choice, they can. They're not all going to get it, but they can all list it.

For example, if everyone really wants to come to MGH for their residency, let's live in the fiction where each hospital only gets one resident. So we're trying to match one hospital to one resident and vice versa. If everyone lists MGH as their first choice, that's fine. They can. Only one is going to get it. So no other restrictions on which permutations they give.

And we want a perfect matching that leaves everyone happy-- eh, happy enough. Like I said, we can't give everyone their top choice, especially if all their top choices collide. But we want to make sure everyone's happy enough. And so let's look at an example before we describe exactly what we mean by happy enough.

So let's imagine-- back to our favorite graph-- AB and CD. So we have applicants, AB, evaluators, CD. And A prefers C over D. So I'll list that as a 1 and a 2. A prefers C, and then after C, they prefer D. B likewise prefers C over D. C prefers A over B. And D also prefers A over B.

If we write it out more similar to the example over there, this is saying that A prefers C over D, and B prefers C over D, and also C prefers A over B, and D prefers A over B. So we're already in a scenario where not everyone can get their top choice. Someone is going to have to take their second choice.

And let's look at an example. What if we try to match AD and BC? So what's going to happen here? So A with D and B with C. So B gets C. So B is happy. B got their first choice. A with D, D is happy. D got their first choice, right? B and D have no reason to want to switch.

But what about A and C? A and C are not paired together, but they each prefer each other over their assigned partners. So if you think of this as a dating scenario, if we try to say that A and D are paired and B and C are paired, then what's going to happen is that A and C are going to run away with each other because they each prefer to be with each other than their assigned partner. We call that a rogue pair.

So we're going to say a pair-- so an applicant and an evaluator form a rogue pair in a matching M when E and A each prefer each other more than their assigned partner. And by the way, for the rest of lecture when I mention matching, I'm going to mean a perfect matching because we're trying to match all the applicants to all the evaluators. And if I forget to say perfect, I apologize. I mean perfect.

All right, so a pair forms a rogue pair in some matching if they would each prefer each other than the partners they're assigned in that matching. And what we're going to look for, so the goal of this Gale-Shapley algorithm, we want a perfect matching with no rogue pairs.

So happy enough-- we want a matching that makes everyone happy enough that you're never going to get two people running away with each other, rather than staying with their own partners. For every pair that you match up, at least one of them prefers to be with the partner they've been assigned than anyone else who would prefer to be with them. Such a matching, we call stable.

So a perfect matching with no rogue pairs is what we mean by a stable matching. And that's going to be our goal. Does there always exist a stable matching? How can we find out when there does exist one? Can we come up with an algorithm that will always find one if it can? In this example, does there exist a stable matching? Well, there are only two matchings in this graph. So let's check the other one. So we saw that AD and BC is not stable. What about AC with BD?

OK. So A with C, well, A and C each got their top choice. There's no way they would ever go rogue, right? And then if A and C can never go rogue, then no one can go rogue with them. So B and D isn't going to break up because B and C aren't going to run away together. C is already happy. A and D aren't going to run away together. A is already happy. So this matching is stable because none of the possible pairs are a rogue pair. Does this example make sense? Cool.

So this particular example, 2 people with 2 people, each with this preference order, has a stable matching. Will there will always be a stable matching? Well, if we relax our conditions a little bit and we decide let's no longer require bipartite. Let's just let everyone rank everyone and then hope to find a stable matching in that non-bipartite graph.

And here's an example of that, so non-bipartite example. And make sure I'm erasing well enough. All right, so let's imagine we have four people. They're going to be A, B, C, and D, and everyone's ranking everyone. So everyone has three other people to rank. And in this example, A prefers B, then prefers C, then prefers D. B prefers C, then prefers A, then prefers D. Let's see. So C prefers A, then B, then D. And I don't even care what D's preferences are.

So let me write this out in the other way. So A prefers B, then C, then D. B prefers C, then A, then D. C prefers A, then B, then D. And D, I don't care. D can have any preference order. This example is already not going to have a stable matching because this graph is, in some sense, symmetric. So A, B, and C are all doing the same thing.

They each prefer the next in order among A, B, and C, then the one after that, and then D is last. B prefers the next in order, then the one after that among A, B, and C, then D is last. C prefers the next in cyclic order, then the next, then D is last. So there are only three different matchings that we can make. D has to be matched with one of these three people, and then the other two are paired together. But since these three people are symmetric in this way, all three of those matchings, basically, behave the same way.

So without loss of generality, we can assume that the matching looks like, let's say, A goes with D and B goes with C. The other two matchings are going to do something similar. Is this matching stable?

And the answer if you look at it carefully, if you check all the pairs, the answer is that AC is rogue. According to that matching, AC is rogue. And let's check that. What does it mean for AC to be rogue? Well, it means A prefers C more than their assigned partner. So A is supposed to prefer C more than their assigned partner. And C is supposed to prefer A over their assigned partner which is B.

So are these two things true? Well, does A prefer C over D? Well, yeah. A prefers everyone to D, so that's true. Does C prefer A over B? Well, yeah, that's the first thing we said. So A and C prefer each other more than their assigned partners, and are, therefore, rogue. So M is not stable.

So this is an example where if you have a bipartite graph and preferences among everyone, then there won't always exist a stable matching, which is why it's so amazing that the Gale-Shapley algorithm which is looking only at bipartite graphs, it turns out-- and we're going to prove-- it always returns a stable matching. And in particular, there always exists a stable matching.

So let's talk about the Gale-Shapley algorithm. This is, again, just for bipartite graphs as we set up over there. We're going to prove it always returns a stable matching, and, therefore, stable matchings always exist in bipartite graphs.

Frequently, there will be many, many stable matchings for a given set of preferences. Gale-Shapley is only going to show us one of them. It's certainly going to find a stable matching, but not find all of them. All right. So let me describe this algorithm. So the algorithm. On each day, we do the following two steps.

So step one, all applicants approach their favorite evaluator who hasn't yet rejected them. So all the applicants walk up and say, hey, I want to go to your hospital for my residency. And maybe Mass General gets seven applicants that morning. At step two, all evaluators reject all but their favorite that approached them that day.

So seven people all approach MGH. MGH is going to look at all seven and say out of the seven of you, I prefer Dr. Evil. And the other six, you can leave forever. I never want to see your face again. Kind of harsh, but that's how the algorithm works. So you're going to do step one and step two every day, and stop when there are no more rejections.

If a day ends and no one was rejected, then that's the end of the algorithm. Now, what I haven't said is-- so all applicants approach their favorite evaluator who hasn't yet rejected them-- if everyone has rejected them, then there's nothing for them to do. They go home and cry if possible. So if they get through their whole list and everyone has said no and they have no one else they can approach, there's nothing else for them to do. They just drop out. All right? Yes.

**AUDIENCE:** For step two, are they allowed to have multiple favorites? Like if the favorite seems like the same for two different people, do they accept multiples?

**ZACHARY ABEL:** Good question. So if an evaluator has two different favorites that are equally preferable, are they allowed to save both of them? So, no. Because of the way we set up the problem, there are no ties. They give a full strict ordering. So among any two people, they know which one they prefer. They have just a full list. So you'll never have ties for preferences. Out of however many applicants you get that day, there will be one clear favorite. And you temporarily hold on to them, and everyone else gets the boot. Yes.

**AUDIENCE:** Does this only work for one-to-one ratios?

**ZACHARY ABEL:** I'm sorry.

**AUDIENCE:** Can you tell me if this only works for the one-to-one ratios?

**ZACHARY ABEL:** Good question. Does this only work for one-to-one ratios? So, yes. Still, in this algorithm, we're only talking about n applicants and n evaluators. And each evaluator is going to end up with one applicant. And each applicant is going to end up with one evaluator. So we have the same number on each side and we're trying to find a perfect matching. Those are the rules of the game. Does that answer your question? Cool. It can be useful to see this algorithm running. It can be-- yes, awesome.

So here's a 4x4 example. These preferences are just whatever preferences they have. We have the evaluators on the top. So imagine those are the med schools on the top and the doctors on the bottom that are doing the applications. So on day one, each of the applicants is going to approach the evaluator that they most prefer. Because there have been no rejections yet, so they all just go to their top choice.

Evaluator 1 says, OK, well, A, you're my only applicant today, so I guess I'll hold on to you for now. Hospital 3 says, oh, no, I have three applicants Among those, I prefer D, so I'm going to reject B and C. So B and C are now gone forever. 3 will never talk to B and C again. And they get sent back down to the bottom. And that's day one. Did that make sense? Awesome.

Now in day two, well, A and D still haven't been rejected by their top choice, so they're just going to stay there for the next day. But B and C who got rejected have to go find someone new. So they're going to go to the next one down their list. And it so happens that they're applying to the same place again this time. And now they're in conflict. So Hospital 2 says, I prefer B, so, C, get away from me. And that's the end of day two. Did that make sense? Nice.

Who can tell me what's going to happen on day three? At least the first step of day three. Yep, up there.

**AUDIENCE:** C is going to go to 1.

**ZACHARY ABEL:** Yeah, C looks through their list and says, OK. Evaluator 1 is my next favorite that hasn't rejected me, so I'm going to go over here to 1. All right? And now Evaluator 1 says, oh, there's a contest. Between A and C, looks like I prefer C this time. So now A goes away. So even though A had been held on for a couple days, someone better came along, so A gets kicked out. It's a cutthroat game.

And let's keep going. Next day, A says, OK, well my next favorite is 3, next down my list. And I want to emphasize it's now day four, but we're not looking at A's fourth choice. We're only looking at A's second choice because A has only been rejected once so far. So we're always going to the next choice for that person, which might be at a different place for each person on that day.

So A goes to a Hospital 3. Hospital 3 says, nuh-uh, and sends A away again. Now A goes to hospital 2. Hospital 2 says, oh yeah, OK, I like that one, so B can go away. And now B has to go fend for themselves somewhere else. So B goes to Hospital 1 and gets rejected. B goes to Hospital 4. And finally, there's nothing else to do. No one is rejected, so the algorithm is done. Did that make sense? Wonderful.

Let's make it make even more sense by doing an algorithm ourselves by hand. So we've seen it run once. Now let's run it ourselves once. So let's do this algorithm. So on day one-- so let's see. Let me make sure I'm matching my notes. Yeah, so I want these on the left to be the evaluators, and these on the right to be the applicants.

So every day, F through J are going to apply to their best remaining choice. And A through E are going to make decisions. So A, B, C, D, E. What happens on day one? F through J are each going to apply to their top choice, OK? So F applies to C, G to A, H to D, I to A, and J to A. OK. So F goes to C, G goes to A, H goes to D, I goes to A, and J goes to A.

So these are the applications on day one. And then Evaluator A says between G, I, and J-- between G I and J, I prefer J. So G and I are rejected. G and I are rejected. G and I are rejected. Sound good? I see some head shake. Was that a question or is it just funny that we're being so aggressive? Wonderful. All right.

So that's what happens on day one. What about day two? Well, J, F, and H weren't rejected, so they stay where they are. So J stays here. F stays here. H stays here. But G and I have to go somewhere new. So G is now going over to B. And I is going over to C. All right. And now C has a choice. C can choose between F and I.

C can choose between F and I. Oh, C is definitely choosing I. F is way down at the bottom. So F is sad. And I believe that's the end of day two. Stop me if I make a mistake. I very well might make a mistake here. Day three. Well, the only one that got rejected was F. So they're the only one that needs to go somewhere new. So we can just copy over J, G, I, H.

And F, where is F going? F was rejected from C, so they're going to B. Let's hope F has a better day today than yesterday. So B is now choosing between G and F. And looks like F wins this time, so G is rejected. Whew, OK. So G was rejected from B. And tomorrow needs to go to E. So day four, G is going down here to E. H, I, F, and J were not rejected, so they stay put for the next day. And finally, there's no contest, so there's no rejection. And finally, we have a matching-- A with J, B with F, C with I, D with H, and E with G. Did that make sense? Question?

**AUDIENCE:** If you flip the evaluators and the applicants, do you get the same matching?

**ZACHARY ABEL:** Excellent question. If we flip the evaluators and the applicants, do we get the same matching? So as we said earlier, the relationship is kind of symmetric, right? They're all giving equivalent data. We could have run this in the opposite direction where F through J are the evaluators and A through E are the applicants. And I have claimed so far that we're going to end up with a perfect matching. Turns out, usually what you're going to get is a different perfect matching.

And we're going to say very precisely why that is a little later on. So a really good question. But yeah, just always keep that in the back of your mind. You could have run the algorithm in the opposite direction and you'll probably end up getting a different stable matching. Yes.

**AUDIENCE:** Similarly, does the order matter? Because of the way you assigned them the first day kind of goes based on the ordering [INAUDIBLE].

**ZACHARY ABEL:** Does the order of what matter?

**AUDIENCE:** The applicants.

**ZACHARY ABEL:** The order of the applicants. So the order of the applicants doesn't matter. The only thing that matters for the algorithm is the individual rankings for each person. But the fact that like G, I, and J approached A in that order on day one didn't matter. All that matters is that these three approached, and J is the best out of these three. Cool. Good question.

All right. So we got a matching out of this example. Is it a stable matching? If my theorem is to be believed, or if Gale and Shapley's theorem is to be believed, then, yes, this should be a stable matching. And in this case, if you check all the possible rogue pairs, turns out, yes, this is stable for this example. So that's at least one point in favor of this algorithm producing correct results. Question?

**AUDIENCE:** If you were to weight these as like the first choice as one, second choice as two, third-- is this necessarily like the smallest weight as well, or is it--

**ZACHARY ABEL:** Interesting question. If we give a weight for who gets which preference-- so first choice gets one point, second choice gets two points, and all the way down-- does this algorithm give a matching with lowest total weight? That's a really great question. At least if you're counting it on one side, I'm pretty sure the answer is yes.

If you're counting it on the other side, I'm pretty sure it gives a matching with the maximum total weight, which is probably not the side that you want. You want lower ranks. But, yeah, really great question. So there are lots of ways you can see this as, in some sense, some optimization problem. We want to maximize happiness, which means minimizing how far along your ranking list you have to go before you find a match. Cool.

So let's try to prove why this algorithm does, in fact, give us stable matchings. And there are a couple of things to prove here. Let's see. Where did I leave off? OK. So first question, so we've given an algorithm. In this class, usually, when we're talking about algorithms, we're talking about state machines.

And so we can think of this algorithm as a state machine where each state is the set of rejections that have happened so far.

So to know where you are in the algorithm and where you have to resume to start the next day, all you need to know is which things have been crossed off from which people's lists. And then what happens the next day, because each of the applicants just goes to the next one in their list. All right? And transitions correspond to run one day of the algorithm. This is a way to translate to transform one set of rejections into probably a slightly bigger set of rejections. Whatever rejections happen on that day are now also part of the new state.

And now we can ask state machine questions like does the Gale-Shapley state machine terminate or might it run forever? Who can tell me? Yes.

**AUDIENCE:** It terminates.

**ZACHARY ABEL:** It terminates. Why?

**AUDIENCE:** It says so in the lecture notes.

**ZACHARY ABEL:** It says so in the lecture notes. Ooh, appeal to authority.

[LAUGHTER]

I say it terminates, and therefore, it terminates. Ooh! Unfortunately, I can't use that justification for myself because if I wrote the lecture notes, then I am that authority. So that's not a valid proof for me to use. So I would ask that you not use it either. Yes, down here.

**AUDIENCE:** I mean, there are only a limited number of rejections that can happen and then they run out.

**ZACHARY ABEL:** Awesome. There are only a limited number of rejections that can happen. Exactly. So every day that the algorithm continues, at least one rejection happens. But there are only n squared rejections that can happen in total. Right. So there are, at most, n squared rejections possible. We use at least one for every transition. So we can follow, at most, n squared transitions before terminating.

Yeah, we terminate just because we constantly use up a limited resource. And we have to finish at some point before we eat up that resource, or in the worst case, when we finish going through all of that resource. So worst case, everyone gets rejected from everywhere and then we stop. Yes.

**AUDIENCE:** Is n the number of applicants?

**ZACHARY ABEL:** Yes, n is the number of applicants and the number of evaluators.

**AUDIENCE:** Oh, they're the same.

**ZACHARY ABEL:** They're the same, yes. All right, so it terminates. Yes. Awesome. Next important question. When it terminates, does everyone have a partner? We were saying earlier that if someone gets rejected from everyone, they go home crying. So does anyone go home crying? We'd like that answer to be no. We'd like everyone to have a partner at the end. All right.

In the examples we've seen on the projector and on the board over there, we did end with an actual perfect matching. No one got rejected from everywhere. So is that true in general? And to talk about this, and also to understand the algorithm a little better in general, let me give us some invariants of this algorithm. This algorithm is a state machine, so let's use invariants and preserve properties to talk about it.

So let's see-- invariant. Each applicant has been rejected from a prefix of their preference list. So on any particular day, if you look at one of these blue guys and ask who they've been rejected from, it's always from the top of their list down to some point. And that's just because of how the algorithm runs.

Every day, an applicant goes to the top choice that hasn't rejected them yet, and either they stay with them for another day, or that evaluator rejects them and the prefix gets one longer. But either way, they're always rejected from a prefix, in particular, each applicant approaches worse evaluators over time since applicants always go down their list from the top.

They're always approaching their favorite evaluators every time they can. And if they're ever rejected, they have to go to someone worse. And if they're rejected, they have to go to someone worse again. So the evaluators that they're approaching only ever get worse over time or stay constant if they're not rejected that day. But they never get better. So applicants' prospects only ever get worse.

And the opposite happens for the evaluators. I claim that each evaluator-- ooh, I can't spell evaluators-- picks from day to day only get better.

And if you think about this from the perspective of an evaluator, well, if no one has ever approached them, then there's nothing for them to do that day. But if someone has approached them and they have some number of people in front of them, they're always going to pick the best from whoever's there. When the next day happens, they at least have that same person again and possibly others. And if there's someone better, then they'll trade up and take someone better. But they never trade down. The evaluators always only keep who they had yesterday or trade up to someone better for the next day. So evaluators' prospects only get better and better from day to day.

We can be a little more precise about this. Let's see. So invariant. So for all applicants and for all evaluators, if the evaluator has ever rejected that applicant, then the evaluator has a favorite-- I'm going to call them a candidate-- has a candidate better than A.

So if Dr. Jekyll approaches Brigham and Women's and is rejected on day 12, then for all future days, Brigham and Women's is going to have some candidate that they finish the day with that's better than Dr. Jekyll. And that's just a more precise way of saying this. Every day they either keep who they had yesterday or trade up to someone strictly better. So when they rejected Dr. Jekyll, they finished that day with someone better than Dr. Jekyll because that's the only reason you would reject Dr. Jekyll.

And then from then on, they either keep or trade up. They only ever get better. Does this invariant make sense? Awesome. So if evaluator has rejected applicant, then evaluator is always going to have a candidate better than that applicant from then on.

So now, let's answer this question. Does everyone have a partner at the end? And let's see what happens if they don't.

So by way of contradiction, let's pick one particular applicant, say, Dr. Jekyll. So by way of contradiction, assume Dr. Jekyll was rejected by everyone-- was rejected by all of the evaluators. All right? And let's see what goes wrong here.

Well, at the end of the algorithm-- at the end of the algorithm, since all of the evaluators rejected Dr. Jekyll, well, what we wrote up here, this invariant says that all of the evaluators have a candidate that's better than Dr. Jekyll. It was true when they rejected Dr. Jekyll and it remains true from then on. All evaluators have a candidate better than Dr. J.

So we know we finish, and when we finish, all of the evaluators end up paired with someone better than Dr. J. But there aren't enough people for that. There are n evaluators and n applicants. And if Dr. Jekyll isn't paired with anyone, then there are only n-1 other applicants that can possibly have partners.

So it can't be that all n of the evaluators finish with a partner better than Dr. J because all n of the evaluators, they can't all finish with anyone at all. Right? But there are only n-1 other applicants, so the n evaluators can't all have partners better than Dr. J. And there's our contradiction.

So if we assume Dr. J. has no partner at the end, bad things happen. We find a contradiction. So that doesn't happen. So Dr. Jekyll does end up with a partner at the end, and, therefore, everyone does. So it returns a perfect matching. It returns a perfect matching. Everyone gets some partner. But is it stable? This is the part we really care about. Is it stable at the end? This is the part we really, truly care about. And let's prove that it is.

So to see why, let's imagine what could possibly result in a rogue pair. Who can be rogue? The algorithm returns some perfect matching at the end. What does it mean for some pair of entities to be rogue? Well, I claim there are three kinds of applicants in relation to one of the particular evaluators.

So if we think of Tufts as one of the evaluators-- Tufts Medical Center-- then some of the applicants, Tufts rejected at some point in the algorithm. Some of the applicants, Tufts never spoke to. And then one applicant, Tufts keeps until the end and ends up paired with. So let's consider each of those three kinds of pairings and show that Tufts can't be rogue with any of them.

So let's assume Tufts rejected-- let's pick a generic example-- Tufts rejected. Dr. No. Let's say Tufts never spoke to Dr. Who? Who's that? I never spoke to them. And finally, Tufts is paired with the Good Doctor.

These are the three kinds of people that can exist relative to Tufts. And let's show that none of these can be rogue. First of all, Tufts is paired with the Good Doctor. They're in our perfect matching. Rogue pairs are never in our perfect matching. They're pairs that are mutually preferred to the perfect matching. So this is never rogue by definition of rogue. If you're in the matching, you're not rogue. All right? So it's these other two cases that we care about.

So first, let's look at Dr. No. All right. So Tufts rejected Dr. No at some point in the algorithm. And because of this invariant that we found, we know that Tufts ends up with someone better than Dr. No-- is paired with someone better than Dr. No. Because the moment an evaluator rejects someone, they know, they're always going to have someone better from then on.

So Tufts and Dr. No can't possibly be rogue because Tufts is happier with who they end up with than they would be with Dr. No. Remember, rogue pair requires both members of the pair to prefer each other rather than their assigned partner. So since Tufts is happier with their assigned partner than with Dr. No, Tufts and Dr. No cannot be rogue. Does that make sense? All right.

So we just use that invariant to say that since Tufts rejected Dr. No, their eventual partner is better than Dr. No. That's all that that is saying. What about the other side? What about Tufts with Dr. Who? Dr. Who never spoke with Tufts during the algorithm. All right? And since Dr. Who never spoke with Tufts-- well, remember what the applicants are doing. They're going down their list from the top.

And since they stopped before they got to Tufts, wherever they stopped, they like better than they like Tufts. So Dr. Who likes their partner better than they like Tufts. And, therefore, Tufts with Dr. Who can't be rogue-- not rogue-- because this time, it's Dr. Who who's happier with their assignment then they would be with Tufts.

So in this case, Tufts is happier with their current partner. In this case, Dr. Who is happier with their current partner. In both of these cases, Tufts with this person on the right cannot be rogue. So Tufts can't be rogue with anyone. So there are no rogue pairs. No rogue pairs can exist. The matching we get at the end is, in fact, stable. Did that make sense? Awesome.

OK, let's return to this question of how good is this matching? Is this the one that makes everyone maximally happy or minimally happy or somewhere in the middle? Does anyone have an advantage? If you were participating in this algorithm, would you prefer to be an applicant or an evaluator? Well, it can go both ways. We already said that applicants' prospects only get worse over time, and evaluators' prospects only get better over time. So maybe it's better to be an evaluator.

On the other hand, applicants start at the top and they stay as close to the top as they can until someone grudgingly says, OK, I guess I'll keep you. So maybe it's better to be an applicant because you're always asking the best partner you can possibly have, rather than the evaluators who have to wait around for people to come to them. So it's maybe not clear which side is preferable. So let me give a term so we can measure what we mean by preferable.

First of all, let's say, so if E is an evaluator and A is an applicant, let's say that E and A are feasible partners if and only if it's possible. to put E and A together in a stable matching.

Not necessarily the stable matching that the algorithm gives. But if there's any stable matching that pairs E with a together. Then we'll say that they are feasible partners. All right? And now we have the following theorems. The Gale-Shapley algorithm pairs every applicant with their most preferred feasible partner.

So each applicant doesn't get their top choice. We know they can't each get their first choice because there might be conflicts there. But each applicant, it turns out, gets the best possible choice they could ever hope to get among all possible stable matchings. So up here on the board, we see that Applicant A is paired with Evaluator 2 and not with 1 or 3.

And what this theorem says is that A can never be paired with 1 and can never be paired with 3 in any stable matching. There is no stable matching that pairs A with 1. There is no stable matching that pairs A with 3. A paired with 2 is the highest-preference partner that A has that is feasible. And Gale-Shapley gives him that partner. So every evaluator gets the best partner they could possibly get in any stable matching.

I personally find this surprising. Like often when we have some optimization problem but we have lots of different metrics that we're trying to optimize-- like Candidate A wants to get their best possible partner, Candidate B wants to get their best possible partner-- and to me, it feels intuitive that maybe these desires might conflict with each other.

Like maybe A and B both have Evaluator 3 as their optimal partner, and there are just two different stable matchings, one that pairs A with 3, one that pairs B with 3. So it's kind of weird to me that each candidate can be given their optimal feasible partner all at the same time and there's no conflict among them. So I find this a little bit counterintuitive for that reason. Beautiful fact, but a little counterintuitive.

And theorem-- Gale-Shapley pairs every evaluator with their, this time, least-preferred feasible partner. So exactly the same idea, but this time, the evaluators, instead of coming out as best as they possibly can, they come out as worst as they possibly can.

Let's say because Evaluator 1 ended up with Partner C, there is no stable matching that pairs 1 with anyone worse than C. There is no stable matching that pairs 1 with B. There is no stable matching that pairs 1 with A. So it's best for all of the applicants at the same time, and, simultaneously, worst for all of the evaluators at the same time. This single matching that Gale-Shapley returns has all of these properties in it.

Which brings us back to the question earlier, if we were to run this algorithm in reverse by swapping the roles of the evaluators and the applicants, we're going to move from the matching that is best for this set to the one that's best for the other set, and the one that's worst for the first set, to the one that's best for the other set. And that's why usually, you're going to get two different matchings out of it. Because if there are multiple matchings and multiple stable matchings at all for this set of preferences, the worst and the best are going to be different from each other.

Do these ideas make sense? We haven't proven them yet, but do the ideas make sense? All right. Let me give you just a brief hint at how we might go about proving this. We'll prove it more precisely in recitation tomorrow. But the basic idea-- pause for dramatic effect-- the basic idea is going to be-- oh, I'm holding chalk. I found it-- one more key invariant, so invariant for our Gale-Shapley algorithm.

So if an evaluator ever rejects an applicant A, then EA are not feasible partners. So if E rejects A at any time during the Gale-Shapley algorithm, then there does not exist any stable matching at all that puts E with A together.

It turns out we can prove this as an invariant of our state machine. Very briefly, let's see if we can do that. OK, so, whew-- also this is true for all evaluators and for all applicants. If any evaluator rejects any applicant at any time, then they're not feasible partners.

So the heart of the proof is let's assume that an Evaluator E rejects an Applicant A today, so we've newly rejected someone. And so now we're going to show that E and A can't be feasible partners in favor of A prime. So if E rejects this applicant in favor of that applicant, well then we know that E prefers the new applicant over the old applicant because that's the only reason E would reject one for the other.

And then the main observation is that in any hypothetical stable matching-- not necessarily the one that comes from Gale-Shapley but in any possible stable matching you try to set up where A matches with E, and then necessarily, A prime matches with someone else-- you can prove that E and A prime are rogue.

So because E rejected A prime in favor of A on some day in the algorithm, you can show that any time you try to pair A with E together in a stable matching, actually, it's not stable because E and A prime are rogue. They each prefer each other. You'll see the details of that in recitation tomorrow. Thank you so much.