

[CS188 FA26] Lecture 4: CSPs I

Summary

Constraint satisfaction problems (CSPs) represent problems as variables, domains, and constraints, allowing specialized search methods to exploit their structure. The lecture introduces backtracking search, forward checking, and arc consistency as progressively stronger ways to prune doomed assignments early.

Key points

- A CSP consists of variables, each with a domain of possible values, and constraints that define valid assignments.
- CSPs can model map coloring, N-Queens, cryptarithmic, Sudoku, scheduling, and other problems.
- Constraint graphs show variables as nodes and constraints as edges or, for higher-order constraints, links among multiple variables.
- Naive BFS wastes effort exploring partial assignments; DFS reaches complete assignments sooner but still explores many invalid ones.
- Backtracking search fixes a variable order and abandons a branch as soon as a constraint is violated.
- Forward checking removes invalid values from unassigned neighboring variables' domains, detecting some failures early.
- Arc consistency checks support between values of connected variables, including unassigned ones, and AC-3 repeats these checks to propagate domain reductions.

Chapters

- 0:00 From Search to CSPs (highlight: slide 8)
- 8:12 Map Coloring as a CSP (highlight: slide 12)
- 12:42 Other CSP Formulations (highlight: slide 20)
- 30:06 Why Naive Search Falls Short (highlight: slide 34)
- 40:06 Backtracking Search (highlight: slide 51)
- 49:36 Forward Checking (highlight: slide 72)
- 1:02:18 Arc Consistency (highlight: slide 89)
- 1:13:54 AC-3, Efficiency, and Limits (highlight: slide 95)

— Chinese (Simplified) —

约束满足问题 (CSP) 将问题表示为变量、领域和约束，允许专门的搜索方法利用它们的结构。讲座介绍了回溯搜索、前向检查和弧一致性，作为逐步增强的方式来早期修

由变量组成，每个变量都有可能值的领域，以及定义有效分配的约束。

可以模型化地图着色、N皇后、密码算术、数独、调度和其他问题。

约束图将变量显示为节点，将约束显示为边，或者对于高阶约束，显示多个变量之间的链接。

朴素广度优先搜索浪费精力探索部分分配；深度优先搜索更早到达完整分配，但仍然会探索许多无效分配。

回溯搜索固定了变量顺序，并在约束被违反时立即放弃一个分支。

前向检查从未分配的邻近变量的领域中移除无效值，提前检测一些失败。

弧一致性检查连接变量的值之间的支持，包括未分配的变量，并且AC-3重复这些检查以传播领域缩减。

从搜索到CSP

作为CSP的地图着色

其他CSP表述

为什么朴素搜索不足

回溯搜索

前向检查

弧一致性

，效率和限制

speaker 1

Okay let's talk about CSP's. Okay Search you saw Search last time. Search made a lot of assumptions about the world in which we were in. For example, Search assumed that there was just one agent that was you.

CS 188: Artificial Intelligence

Constraint Satisfaction Problems



Fall 2026

University of California, Berkeley

were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at <http://ai.berkeley.edu>.

Slide 1 · 0:00

What is Search For?

- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space

There was no one else in the world doing adversarial or other actions that could affect you. Search also assumed that there was a deterministic world. In other words, once you come up with your master plan that says, I will take this action and this action, and then this action, it's expected that if you execute those actions, they are guaranteed to work and your plan will execute exactly in the way that you planned. This might not be true. Maybe you take an action and it doesn't do what you expect. Search also assumed that you could fully observe the state. In other words, you know exactly where you are, and once you take an action, you know exactly where you end up.

So Search made all of these assumptions, and what we'll do in the rest of the first half of the semester anyway, is we'll start relaxing these assumptions. What if this isn't true? This isn't true. What if this isn't true? By relaxing them, we'll be able to solve different types of problems.

What is Search For?

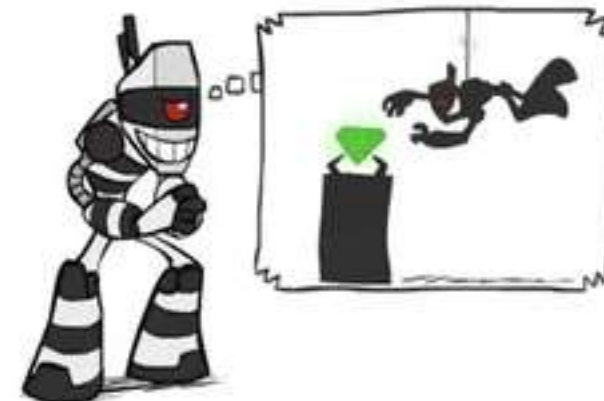
- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space



Slide 3 · 1:12

What is Search For?

- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space
- Planning: sequences of actions
 - The path to the goal is the important thing
 - Paths have various costs, depths
 - Heuristics give problem-specific guidance



That's sort of the sketch of what's gonna happen in the rest of this first half of this semester. And so this section, this lecture, and the next one, will be about a specific type of search problem. And so to motivate that, first, let me remind you that search problems that we've seen so far, you can think of them like planning problems. So for example, here we have this like thief robot, and the robot wants to get the gem and the robot's thinking, well, if I want to plan ahead, these are the sequences of actions I can take so that I can go and successfully steal this gem from its holding position. And so what we would think about is the path to the goal. In other words, what sequence of actions allows this robot to get the gem? And this path would have things like costs. For example, each action could have a different cost, and we want to find the least cost. We think about the depths of different paths and to help you solve these problems, what you would do is take your problem, like the robot and the Gem. You would formulate it as a specific instance of a search problem, and then you would give the problem to one of your many general purpose algorithms like Breadth First Search, Uniform Cost Search, a Star Search and along the way, you can also give this problem a special hint, like a heuristic that says, I'm getting closer to the solution. I'm getting further from the solution, takes it a state and outputs a real number telling. You, whether the state is good close to the goal or bad far from the goal, this is all stuff from last time.

What is Search For?

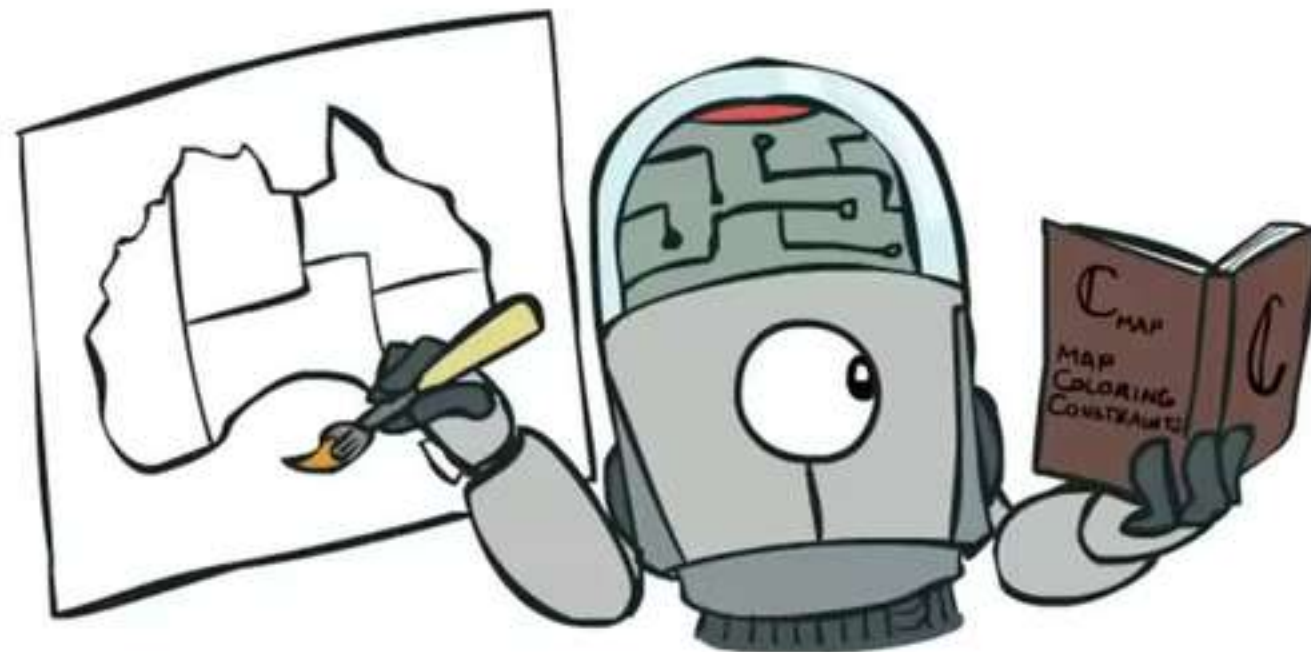
- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space
- Planning: sequences of actions
 - The path to the goal is the important thing
 - Paths have various costs, depths
 - Heuristics give problem-specific guidance
- Identification: assignments to variables
 - The goal itself is important, not the path
 - All paths at the same depth (for some formulations)
 - CSPs are a specialized class of identification problems



Okay, and the reason why I'm telling you this is because today is more about a specific class of search problem called the identification problem. So here we're actually not too concerned about the path to the goal, and we're instead more concerned about a specific set of assignments to variables. So, as an example, let's say that here we have this detective robot. This robot comes in after the gem was stolen, and the detective robot wants to figure out what was what happened to the gem? And like, what happened here? Why is the gem gone? Why is it, ah, lying underneath this blanket? And so here we don't really care how, the detective. Gets to the answer We just care about the assignments to the variables. In other words, what's the explanation for why this thing happened? We don't really care about how the detective gets there, we just care that there is an answer. And so this is a specialized class of search problems that we'll look at, and because it's a specialized class, we'll have specialized tools to solve it. It's all kind of abstract I will make it very concrete.

Okay, but interrupt me if there are problems, or tech problems, because it's a new building, I still don't understand how all the buttons work. Okay cool. It's all like about constraint satisfaction problems, or CSPS, for short. So what are they? So first let's think back to.

Constraint Satisfaction Problems



Slide 6 · 3:42

Constraint Satisfaction Problems

- Standard search problems:
 - State is a “black box”: arbitrary data structure
 - Goal test can be any function over states
 - Successor function can also be anything

One more time, because CSP's are a special class of search problems. If you think back to the general purpose algorithms we developed, like uniform cost Search, a Star Search, Depth First Search, those search problems didn't care about what the specific underlying search problem was. In other words, you would just give it a state and you would say, here's the state, and what can you do with the state? You can't open it and look inside. Remember, if you're working on depth first search or uniform cost search, you don't get to look inside the state and say, oh, the state is Romania, or the state is pancakes. All you get is a black box. It's a state and you can only ask two questions. Question number one, are you a goal? And question number two, what are your successors? And that's it. You can't look inside the state. And this allowed for our algorithms to be very general purpose. Depth First Search can solve any search problem as long as you formulated using this formulation. But it was limited because you can't look inside the state and see what's going on. Okay.

Constraint Satisfaction Problems

- Standard search problems:
 - State is a “black box”: arbitrary data structure
 - Goal test can be any function over states
 - Successor function can also be anything



And so you can almost think of the search states as these black boxes, and the only thing you can do is present it to this judge, and the judge would say, yes, it's a goal, no it's not a goal, yes it's a goal, no it's not a goal. It would just give you a yes no answer, and that was it. So it's general purpose, but that also means we can't use any specialized techniques on it. Now CSP's are more specialized, so here I'm actually going to let you open the search state for the first time. So in general search problems like pancakes, Romania can't do that. But for CS P's? I'm going to let you open the state and look inside, and I will tell you that for this special subset of problems. When you open the state and you look inside. What you will see is a set of variables, and each variable has a value from a domain. And again, examples will hopefully make this more clear. But the idea is there's a bunch of variables, and each variable has like a list of possible values. So this variable can be one of these five values, this variable can be one of these five values, this variable can be one of these ten values, so on and so forth. And your goal test, which I'm also going to let you open up for the first time, so instead of a judge telling you, yes, no, yes no. I'm going to let you open up the goal Test, down inside the goal Test, you will see a set of rules, also called constraints, that tell you what values can and cannot be assigned. So it's still a search problem. There's still a state. There's still a goal test. Still a successor function, but the difference is now, for the first time, I'm letting you open up the state and open up the goal Test, and by doing so, you will get special purpose algorithms that solve this specific type of problem really well.

Constraint Satisfaction Problems

- **Standard search problems:**
 - State is a “black box”: arbitrary data structure
 - Goal test can be any function over states
 - Successor function can also be anything
- **Constraint satisfaction problems (CSPs):**
 - A special subset of search problems
 - State is defined by **variables X_i** , with values from a **domain D** (sometimes D depends on i)
 - Goal test is a **set of constraints** specifying allowable combinations of values for subsets of variables



speaker 1

Okay. So now the difference is, instead of a judge telling you, yes, yes, no, yes, we have like a little book, and it has a set of laws, and the laws are the constraints, telling you what's allowed and what's not. Okay. And so what we'll do in both search problems and CSP's is we will take general purpose problems, pancakes, pac-man eating all the dots. Today's gonna be about maps and colors, and you'll take these problems, and you'll always formulate them as a specific instance of search problem or the CSP problem, and taking your problem and formulating it as an instance of one of these problems. It's like your first job, and then your second job is once you have the problem formulated in this very specific way, what are the methods to solve it. So just like last time we'll see both of these things. How do you formulate the problems, and then how do you solve them? And as mentioned earlier, because CSP's are a more specific class of problems, they have more specific algorithms that work. Very, very, very well for these problems only, and not the other ones. Yeah question. Why isn't a heuristic? Yeah good question. Why is a heuristic part of a state? It doesn't have to be. So if you go back here heuristics don't have to be part of a state. In fact the heuristic is just a function. It says Here's a state. Give me a number that tells me something about the state.

Constraint Satisfaction Problems

- Standard search problems:
 - State is a “black box”: arbitrary data structure
 - Goal test can be any function over states
 - Successor function can also be anything
- Constraint satisfaction problems (CSPs):
 - A special subset of search problems
 - State is defined by **variables X_i** , with values from a **domain D** (sometimes D depends on i)
 - Goal test is a **set of constraints** specifying allowable combinations of values for subsets of variables
- Simple example of a *formal representation language*
- Allows useful general-purpose algorithms with more power than standard search algorithms



speaker 1

Sorry can you say that one more time? Yeah, so you're using the state. You're calculating a state. Yeah, so the question was about how states and heuristics are related, and the answer is a heuristic is a function. It's a piece of code and it takes in one. Argument which is the state? And it outputs one value, which is a number, telling you something about the state. Yeah good question. Anything else? Last one. Okay cool so this is the thing we're gonna formulate.

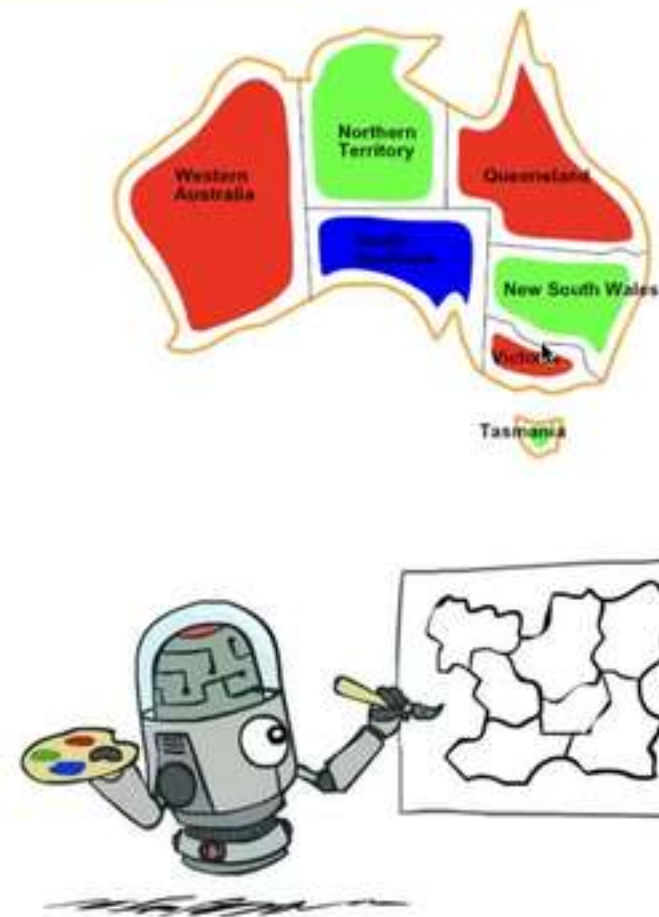
CSP Examples



It's kind of abstract now. Let me show you some examples. Okay, so the running example for this lecture, and also in your book, is map coloring. So I hope you're all really familiar with the geography of Australia. So we're about to become really familiar. This is basically what you need. It's a set of states. So Australia has 1234567 states, and our goal is to assign colors to each of the states. And when you color maps, you ever look at a map with colors on the states in the country. It'd be really nice if two adjacent territories or states don't have the same color. Like this will be really ugly. If Western Australia and Northern Territory were both red, that's really gross. We want all the neighboring states to have different colors now. This is not the only CSP example, so don't go out into the real world and start thinking every CSP is maps. In Australia not the case, so there's lots of other CSP's, and I'll show you some examples later. But the running example for us, just to see how to solve them and how to formulate them, will be Australia Map. And the rules are we need to assign colors to all the states so that no two adjacent states have the same color. Same color. Simple enough. Okay.

Example: Map Coloring

- Variables:



Slide 12 · 9:24

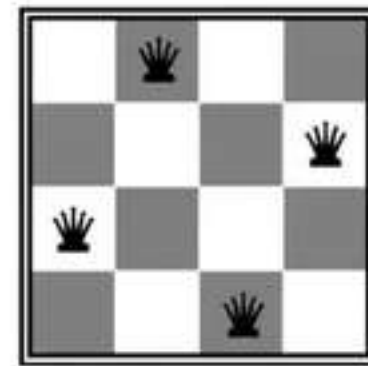
So we can formulate this. We can take this problem that I just described to you in English and make it into a c s p, which has very specific mathematical definitions. So the CSP asks What are the variables. Well, in this case, I could say the variables are the states. So look, there's a variable for w a, that's Western Australia, one for the northern territories, one for Queensland, New South Wales, what is this? Victoria, South Australia and Tasmania all by itself in a little island. So there are seven variables. Then I have to specify the domains. What can I assign to each of these variables. So there are seven variables each one gets a value. And what's the pool of possible values I can pick from? Those are the domains, and so I can say, well, each of the variables can be assigned to either red or green or blue. So I have now taken my English language problem and formulated it as a specific instance of a c s p, because I told you what the variables are, I told you what the domains are. Now I'm not quite done, because if I stop right here, you can just say I'll just assign everything red and I win, but can't do that, so I'm going to introduce constraints, rules that say there are certain things you can't do. So I could say the constraints in English, like adjacent regions, must have different colors, that's what I just said to you in English. But you can also express these constraints mathematically. So here are two possible ways to do it. One is, you can write an implicit constraint, and there's like a little piece of code that you can run to check whether or not the constraint has been satisfied. So this little piece of code says, the value in w a, which might be red or green or blue, cannot be equal to the value in NT, which might be red or green or blue. You can run this little piece of code to check whether or not they're the same. And if they are the same, then, well, your constraints have not been satisfied. But if they're different then the constraint is satisfied. So you can write a little piece of code like this, or you can write it out explicitly, and you can say, well, if I'm assigning w a and n t i can assign w a equals red and NT equals green, or I can assign. W a equals red and NT equals blue, or I can assign w a equals green and NT equals blue, I can write out the entire set of possible assignments. So you can write a little piece of code, or you can list out all the possible assignments either, one's fine, just depends on how you want to specify your constraints, but here you can see, this is like a mathematical version of the thing I wrote in English here. Cool. So what's the answer? How do you solve one of these problems? And the solution you're looking for? Here is an assignment, so you want every variable to have a value, in other words, every state must be colored red or green or blue, and you need to satisfy all ... continued on the next page

Slide 12 (continued)

the constraints, so, for example, the one I'm showing. Here, this is a valid solution. It's not the only valid solution. In fact, CSP's often have many solutions, but here's just one example of a solution. You can check for yourself. No neighboring states have the same color. So this is a good solution. Other ones exist too. But this is our goal. We want to take our problems, specify it in this specific way, and then out pops a solution that is an assignment of values to variables that satisfies all the constraints. Haven't told you how to come up with these solutions yet. So far, we're just taking English language problems and turning them into math. And once we have the math, we can go solve the math later, which we will do. Okay, thoughts on map coloring and Australia. Okay we have Romania we have Australia.

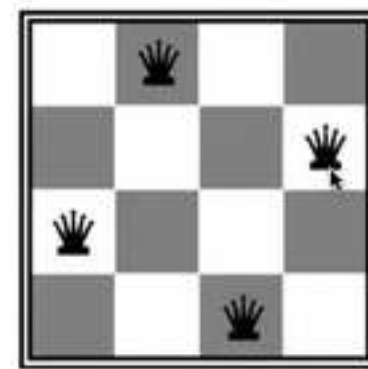
Okay we'll wonder what countries will come up next. Okay let's talk about N Queens. Have people seen the N Queens problem? Is anyone a frequent N Queens player? Okay maybe not.

Example: N-Queens



Slide 13 · 12:42

Example: N-Queens



Okay this is the N Queens problem. So this is a Queen, it's on a little chess board, and if you've played or played chess, you know that queens can attack in the up down direction. They can also attack in the left right direction. They can also attack along the diagonal. So what our goal here is is to place n Queens on the board, where n is some number you pick such that none of the Queens are attacking, and you can check this. So, for example, I'm not gonna drop, but like this Queen over here. Well, there's no one else in the same row, so it's not attacking any other Queen. There's no one in the same column, so it's not attacking any other Queen, and there's no one on the same diagonals, so it's not attacking any other queens. So you have all these queens and your goal is to place them on the board so that no two Queens are attacking, and you have a peaceful kingdom, and you don't have these nasty fights. Okay. So how do you take this problem that I just described in English and pictures and turn it into a mathematical CSP object? There's actually multiple ways to do it. So let me show you two possible ways. So, one possible way you can take this N Queens problem and make it into a c s p, with variables and values and constraints is, I can say that the variables touch panel is offline. I don't know what that means but if. Anything is broken you should tell me. Sorry, this screen just popped up, had some weird error message popped up. Okay it's back. I don't know what happened. Okay let's talk about ah n Queens again. So I can say the variables are the squares. So, for example, this square is a variable, this square is a variable, this square is a variable. So I'd have 16 variables here. So what values can I assign to the variables, like the square right here? What values can I assign? Well, this case, I could say this square only has two possible values. So either got a Queen or it doesn't have a Queen. So the domain might be something like zero and one, no Queen, or put a Queen on that square. So I have the variables I have the domains. I'm not quite done yet, because if I stop here, you might say, well, a valid solution is let me put a Queen on every square. Okay, but that doesn't work, so I need constraints to tell me what is and isn't allowed. And once again, you can write the constraints a lot, lots of different ways, here's an explicit way to write it. And so it would say, Okay, well, and again, don't read this too carefully, right? It's like math, you can go share it on your own time later, but it basically says something like, Okay, well, if the two queens are, if you have two squares in the same column, they could either both not have queens. Or one of them can have a Queen or the other one can have a Queen, but they can't both have a Queen. [afraid] So I can't have like one comma one, because if two queens are ... continued on the next page

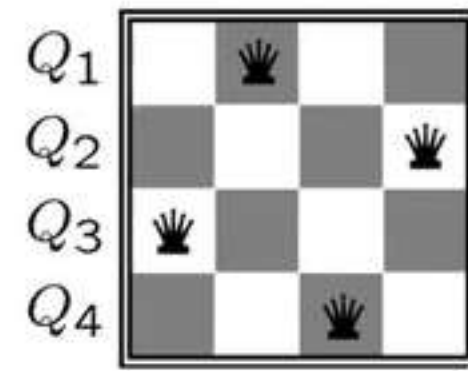
Slide 14 (continued)

in the same row, that would be a problem. And the same thing for all the columns. So it's a lot of writing, and these are for the diagonals. Oh, if I have this diagonal, I can have a Queen on one of the squares or a Queen on another one of the squares, but I can't have two queens on two squares, otherwise, they would be fighting, and I don't want fighting. So this might be like an explicit way to write out the rules that I just described in English for you, which is that the Queens can't attack. So this is one possible formulation, you can take this formulation, plug it into your favorite CSP solver and hopefully out pops, solution where all the Queens don't attack, and this board is an example where Queens are all nice and peaceful, Yeah, what's up. Yep. Yeah, you're absolutely right. So the last thing I was gonna mention, you you beat me to it, is if you stop right here, you've actually under defined the problem because there's a cheating solution as the one you brought up. The cheating solution is what? No queens anywhere. Just set every square to zero. That wouldn't solve my problem. So I need one more constraint that says something like, if you add up all the squares, there are actually N queens on the board. This is a good example of your formulations have to be very carefully specified. If you accidentally miss a rule like Oops, I forgot this. Well, then the CSP solver will just spit out and say, Look, I solved the problem. I just don't put any queens anywhere on the board. That doesn't solve your problem. You want n queens. So you have to add one more constraint that says, maybe like, add up all the. Squares and queens in total. Okay, this is a pretty good formulation, but it's not the only one.

Example: N-Queens

- Formulation 2:

- Variables: Q_k
- Domains: $\{1, 2, 3, \dots, N\}$
- Constraints:



So turns out, it's often the case in these problems and in the problems on your projects, that you can actually take the problems and rewrite them in different ways. So here's another formulation of the exact same problem. So here I'm actually building a little bit of the solution into the formulation. So me, as a human, I'm thinking, well, as a human, I know that, every row is going to have exactly one queen, because if it's a four by four board and I need four queens on the board, every queen has to be on its own row. I can't have two queens on the same row, they would be fighting. So I know for sure that every row has only one Queen. So maybe another. Way to formulate the same problem is to say my variables are actually the rows. So now, instead of 16 variables, one per square, now I have four variables and the first variable is row one, second variable is row two, row three, row four. So what are my domains? The domains might be like, Where's the Queen on row one? Is it in square number 123 or four? And Where's the Queen on row number two? Is it in square number 123 or four. So, for each of these variables, Q_1 through Q_4 , I would assign a number that tells me is the Queen? In column one, column two, column three, all the way up to column four. This is the same problem as before, I just specified it slightly differently, and here I'm sort of baking a bit of the solution into the formulation because I have a huge. Know that each Queen has to be on its own row. But I would still have to write constraints, and here the constraints are actually a little bit trickier to write. So I just got lazy and wrote a function that says somewhere in my code, there's a function called non threatening that lets me check are they on the same row, the same diagonal or the same column? And for all queens, they need to satisfy this non threatening function. This would be an example of an implicit constraint. You can also be explicit and write out, Okay, between one and two, they can both, they can be on columns one and two one and three one and four oh, but they can't be on one and one, that would be an explicit version of the same constraint. It's up to you, you can write whichever one you like, whichever one your solver demands. Okay, so far, so good, okay now. One powerful tool.

Because once you start looking at these constraints, they get kind of hard to look at, like, oh, what is this, what is that? Or like, oh, what is this? So one way to look at these constraints and really think through them is to draw a graph. And here the same warning from last time comes up.

Example: N-Queens

■ Formulation 2:

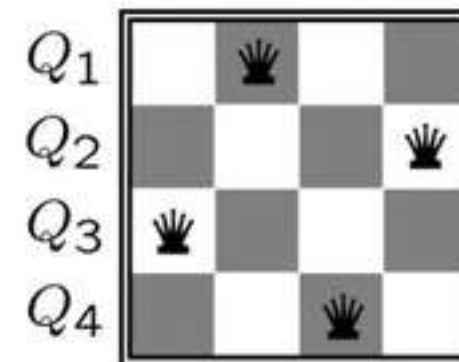
■ Variables: Q_k

■ Domains: $\{1, 2, 3, \dots, N\}$

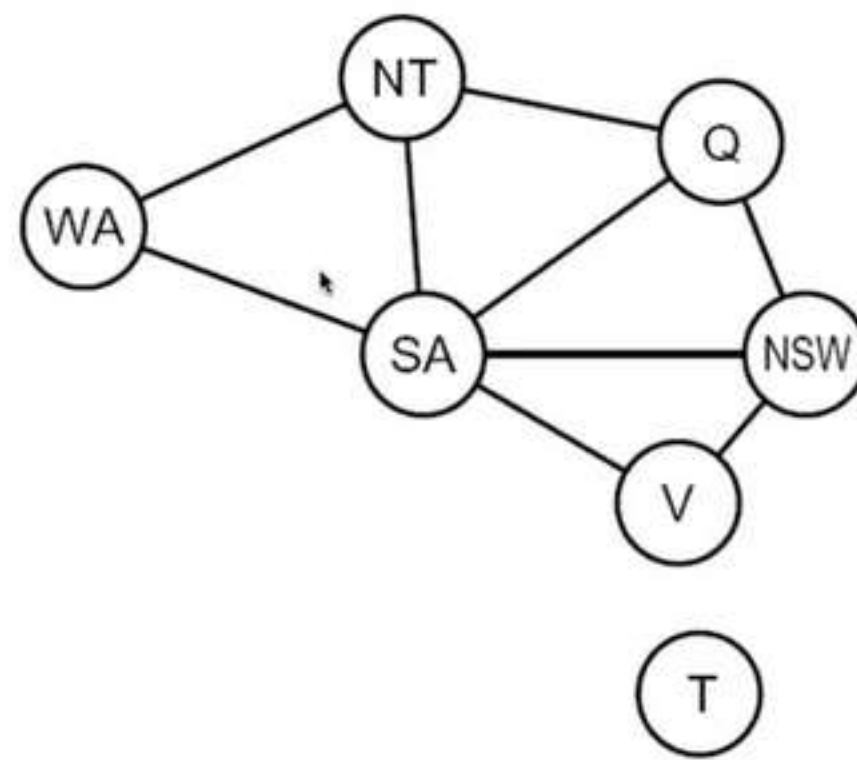
■ Constraints:

Implicit: $\forall i, j \text{ non-threatening}(Q_i, Q_j)$

Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \dots\}$
...



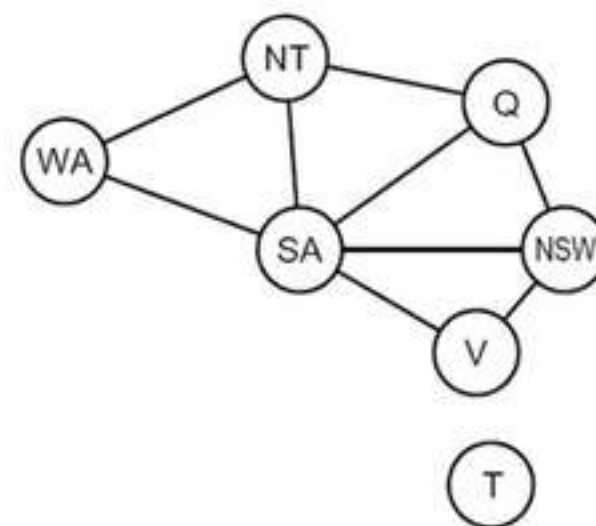
Constraint Graphs



It's a graph. This class is just graphs on graphs on graphs. So every time you see a graph, you should stop and ask yourself, What does this one mean? This is a totally different graph from the ones you've seen before, so in this graph, what are the nodes represent? Like, when I draw a circle, this node represents a variable like this is the Northern Territories variable, this is the Victoria variable, this is the New South Wales variable. So each These circles represents a variable in your CSP that you want to assign values to. What are the edges represent? The edges represent the fact that there is a constraint between these two variables. In other words, the value of V depends on the value of s a in some way. So for example, here, it would be the fact that South Australia and Victoria are neighbors, so they can't be the same color. So I draw this edge to remind myself, oh yeah, these two things can't be the same color. And so by doing this, you sort of get a structural picture of where the constraints are, which I think is nicer than looking at all the math. You can look at the math if you want, but I like looking at pictures. So, the nodes represent variables, and the edges represent constraints that live between the variables.

Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



[Demo: CSP applet (made available by aispace.org) -- n-queens]

Okay. Now there's all sorts of different constraints. So for example here, I'm showing a c s p where all the constraints are binary, which means they only relate to variables. So, for example, w a and NT you cannot be the same color, and that's it. I just thought there. I didn't have to say anything about this one or that one, or that one, or so on. Okay. But there are other constraints out there. There are unary constraints that say, okay, w a has to be blue for some reason, or it can only be blue or green. That would be a constraint on one variable. That'd be a unary constraint. You can also have higher order constraints. So for some reason, maybe you're like, oh, w a and South Australia and NSW all have to be different colors. That would be a higher. Order constraint but in this case, all the constraints are binary, they just relate two variables, and because all the constraints relate two variables, I can draw this nice picture where the nodes are variables and the arcs show constraints. Cool, and then once I have this, I can shove it into an algorithm, a general purpose algorithm that can solve any problem, as long as I formulate it in this special CSP way, where I say variables, values and constraints.

Example: Cryptarithmic

- Variables:
- Domains:
- Constraints:

$$\begin{array}{r} \text{TWO} \\ + \text{TWO} \\ \hline \text{FOUR} \end{array}$$



Cool, let me show you some other ones, so you're like, oh, everything is map colors, let me prove to you that not everything is map colors, we saw N Queens earlier, here's another example of a c s p that's not maps or colors, or Australia, so. Every one of these little puzzles is like, oh, you need to find a number, a digit that goes in T, and a digit for W and a digit for O, so that this equation makes sense. So how would you do that? Well, once again, I can formulate this Acsp A-C-S-P, so every one of these letters represents a digit. I wanna know what digit they represent, so I can say the variables are each of the letters, like F is a variable and T is a variable, and W is a variable, and what are the domains? Well, the domains are like zero through nine, because each of these is assigned to exactly one digit. What the heck are X one X two and X three? I don't see them anywhere in my problem. Now you have to think back to your arithmetic class. Why are there three more variables that I didn't ah write down here? Well, like, what if O was equal to? Let's say O was equal to nine? Okay, there's a nine really ugly nine, Let's say that was a nine, was nine plus nine, let's say 18. Ah, I need a carry bit, right? So the X one and X two and X three. Those are the carry bits that, like, live up here. Because when you do math the sum is greater than ten, you need some carry bits. So X one, X two, x three, they're the carry bits that live up here. Okay, again, not the most important thing. The sort of take away here is that you can take problems that have nothing to do with maps and formulate them as CSP's. And once again, you need constraints, and the constraints might look something. Like, well, if I add O and O, I'd better get R plus, maybe some carry bit, and if I add W and W, I'd better get U plus some carry bit. And if I add the carry bit and T and t i guess some letter the number O again, and so on, and so forth. And maybe one more constraint to make this problem interesting is, you want, you might want to say, they should all be different, because what if I don't have this constraint? That all the digits are different, you would just say 000 plus, 000 equals 0000. It's too boring. So when people write these little puzzles or whatever, they might add a constraint that says all the digits have to be different. Cool once again. You can draw it as a graph so you can say, what are the circles? They represent? My variables FTU. What are these squares? Well now suddenly you have constraints that are higher order. They link up more than two variables. For example look at this constraint right here. This says O, plus o equals R, and then there's a possibly a carry bit up here. Like nine plus nine equals 18 you gotta carry the one. So this is a constraint that relates three variables, and the way you draw, that is, you say, well, O and R and X one, ... continued on the next page

Slide 19 (continued)

they're all related by a constraint. So you draw a little square. This square is my constraint, and the constraint relates R and O and X one. So the circles are variables and the squares are constraints, and the lines tell me which constraints tied to which variables. What about this one up here? Why is there. A constraint that relates all the letters oh, because that's the all diff constraint that says f t u w r, and O, they all must be different, digits cannot reuse the same digit twice. So I took my problem and I formulated it in a very specific way so that my CSP solvers can give me answers.

Example: Sudoku

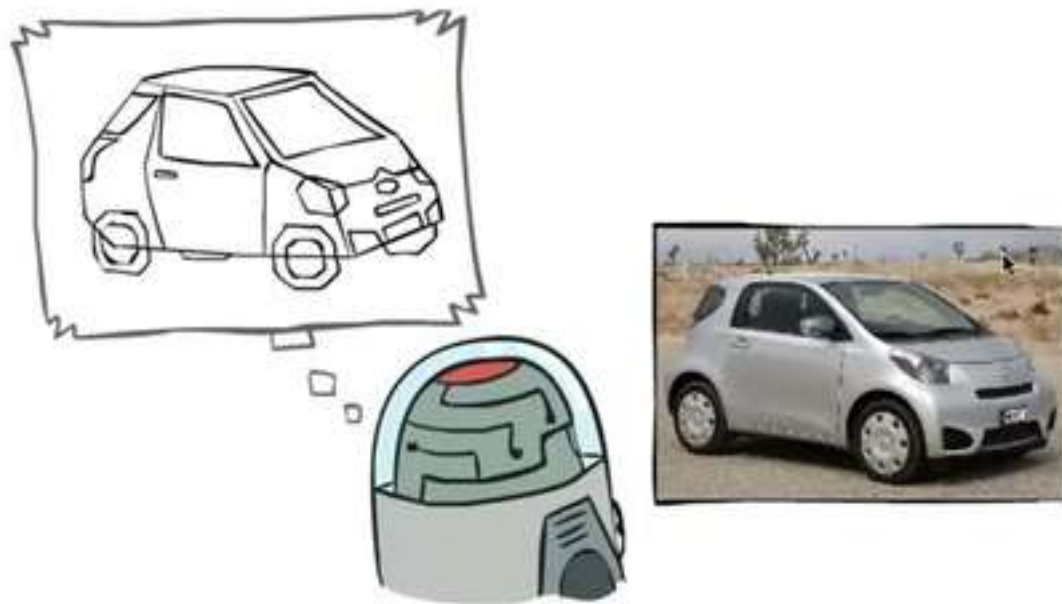
					8			4
	8	4		1	6			
			5			1		
1		3	8			9		
6		8				4		3
		2			9	5		1
		7			2			
			7	8		2	6	
2			3					

- Variables:
 - Each (open) square
- Domains:
 - {1,2,...,9}
- Constraints:

Cool. Sudoku Anyone play sudoku? Sudoku champion? Okay, so how does sudoku work? The idea is we have all these squares and you need to fill in numbers and there's a bunch of rules, right? It's like, well, within the column I can't have the same number twice, and within the rows I can't have the same number twice. And then within each of these little squares, I can have the same number twice. How would you formulate this Acsp A-C-S-P? You're starting to get the hang of it. Okay, I could say, well, each square is a variable, and each domain might be one through nine, because those are the only legal numbers I can write in Sudoku. And the constraints might be something like, well, within each column, I cannot have the same digit twice and within each row, I can have the same digit twice and then within each of these little squares, this like a three by three regions, I cannot have the same number twice. So if I formulate sudoku in this way, I can plug it into a c s p and solve it. By the way, when you play sudoku, have you ever noticed there are some problems that are really easy? You're like, oh, well, this number is nine, so this one must be one, so this one must be two and then you're done. And then there are some sudoku problems that are really hard because you have to. Go back and try this thing and then it doesn't work and you have to go back and try another thing. Turns out, the difficulty of that sudoku problem for you as a human is really closely related to how difficult these CSP, problems are for computers. So now, you know, there's easy sudoku problems and there's hard ones. And the difference between them is very closely related to how computers solve these things. Which is something we'll see soon. Okay, a sudoku. Okay, I'm not gonna keep going through all the examples, but I'll show you just one more.

Example: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects
- An early example of an AI computation posed as a CSP



- Approach:
 - Each intersection is a variable
 - Adjacent intersections impose constraints on each other
 - Solutions are physically realizable 3D interpretations

This is not how people do computer vision nowadays, but ah, one of the earlier approaches to computer vision, where you wanna, you know, see what the picture is is you would get this little line drawing, and in this line drawing, you have to stop and ask yourself, it's like, a. One of those like visual tricks that play on your brain, like, look at this corner right here. Like, is that corner in inY? Because it's, ah, facing inwards, or is like an outy? Because it like, sticks out? And so what you could do is there's some clever way you can formulate this Acsp A-C-S-P. And you can say, well, if this one points inwards, well, then this corner has to point out and this one has to point in. And you can somehow use this to detect which parts of this picture are like an iny or an outy, right? So this could be like something you can, ah, this could be like a little stage you can step on, or like this would be a box that you're looking at. So which of the two is it? A c s p can help you solve that again? This is not how people do it nowadays, but the point is CSP can solve things that are totally. Really different from maps and colors, even though we will spend most of today on maps and colors. Okay cool. Alright, so in this case, it was an INY all along, right? It was a INY because it's like a little corner that I cut out of the cube. But if you look here, maybe it was a box, you don't know, the visual illusion. Okay cool. Let me tell you a little bit about different types of CSPS. There are some CSPS we'll talk about, there's a whole class we will not talk about because we don't have time, but let me just remind you that they exist.

Varieties of CSPs

- Discrete Variables

- Finite domains
 - Size d means $O(d^n)$ complete assignments
 - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
- Infinite domains (integers, strings, etc.)
 - E.g., job scheduling, variables are start/end times for each job
 - Linear constraints solvable, nonlinear undecidable



So it turns out there's lots of different CSPs. So we will focus on CSPs, where the domains are finite, like the colors are red, green and blue, or the assignments are true or false, or the sudoku. Puzzle was digits one through nine, but there are also CS, p's where the domains are infinite. For example you can put any integer in the box. How many integers are there, like infinitely many, or what? You can put a real number in the box or a string. Well, then now it's like there's infinitely many values you can put in the box. Those require different approaches from what we're about to see today. And there are also CS, p's that have continuous domains, like for example, like scheduling, like when are we all free? Right? That could be a continuous ah, domain, because time is continuous, right? And the variables might be like, oh, when does this job start? When does this job end? When are you free? When am I free? We can schedule rooms. Like, who's teaching in which classroom, so that the whole campus works and we don't have two classes in the same room. So there's all these different problems, different approaches to solving them, if you've ever seen a linear program, that's a special class of CSPs as well that we will not talk about here. But point being, we're just looking at a small subset of them. I'm just reminding you that other ones exist.

Varieties of CSPs

Discrete Variables

- Finite domains
 - Size d means $O(d^n)$ complete assignments
 - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
- Infinite domains (integers, strings, etc.)
 - E.g., job scheduling, variables are start/end times for each job
 - Linear constraints solvable, nonlinear undecidable

Continuous variables

- E.g., start/end times for Hubble Telescope observations
- Linear constraints solvable in polynomial time by LP methods (see cs170 for a bit of this theory)



Right, so there's also other ones, like continuous variables, again, if you have linear programs, those are ways you can solve them, maybe instead of just three colors, you can use any color that you can possibly think of, well, now, there's too many variables to think about, infinitely many even. And so we will not think about this, but just do know that they exist. So there's infinite domains and there's also continuous domains. Okay.

Varieties of Constraints

- Varieties of Constraints

- Unary constraints involve a single variable (equivalent to reducing domains), e.g.:

$SA \neq \text{green}$

- Binary constraints involve pairs of variables, e.g.:

$SA \neq WA$

- Higher-order constraints involve 3 or more variables:
e.g., cryptarithmic column constraints

- Preferences (soft constraints):

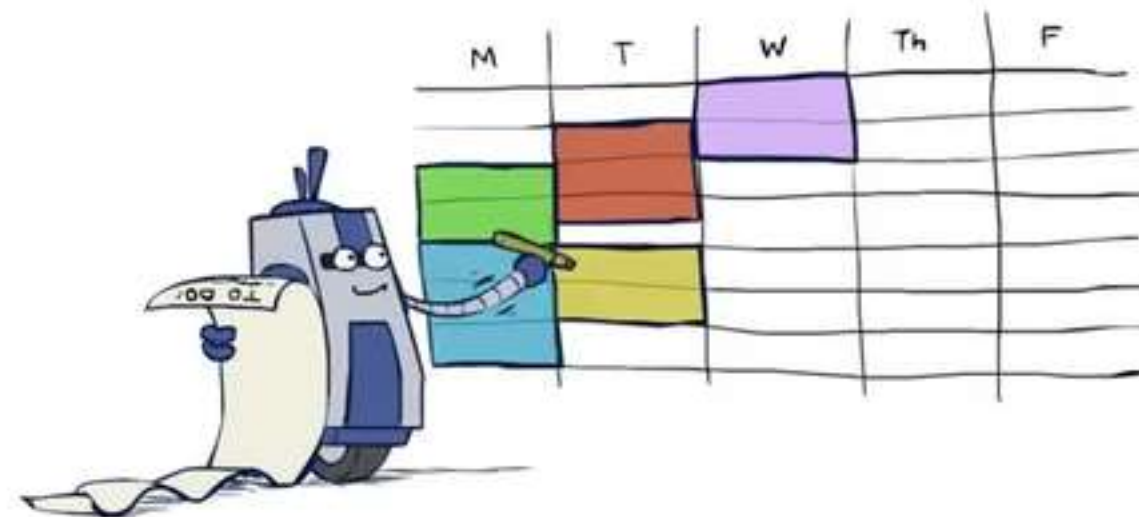
- E.g., red is better than green
- Often representable by a cost for each variable assignment
- Gives constrained optimization problems
- (We'll ignore these until we get to Bayes' nets)



And there's also different types of constraints. So we already talked about the ones we will cover, unary constraints, talk about a single variable, like South Australia cannot be green for some reason. Binary constraints are pairs like these two states cannot have the same color. And then higher order constraints are, like the crypt arithmetic from earlier, the two plus two equals four, where you have three variables that are related by a single constraint or more. And one thing we will not talk about is preferences. So maybe you want, you'll say, I want to color the Australia map, but I like red. Please use red if possible. And how would you do that? You'd have to add costs and that's not something we're gonna talk about, but it's something you can do. You can go check it out on your own time, but we will just. To constraints that are either satisfied or not, but do know that preferences are something, you could also bake into these problems if you want to, or if your problem demands it, because you really want red for some reason.

Real-World CSPs

- Scheduling problems: e.g., when can we all meet?
- Timetabling problems: e.g., which class is offered when and where?
- Assignment problems: e.g., who teaches what class
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- ... lots more!



- Many real-world problems involve real-valued variables...

Okay. Cool, and again, there's so many CSP's out there, right? So as mentioned earlier, right? There's like, when can we meet? Or you know, like, where do you put the classes? Or who teaches the classes? Or you know, how do you schedule? Airplanes so that they land and depart at the correct times? And they don't like bump into each other in the air and explode and things like that. So lots of different things can be solved using CSP's, and they all involve assigning values to variables. Okay. Just remember, it's not all maps, even though they're all maps today.

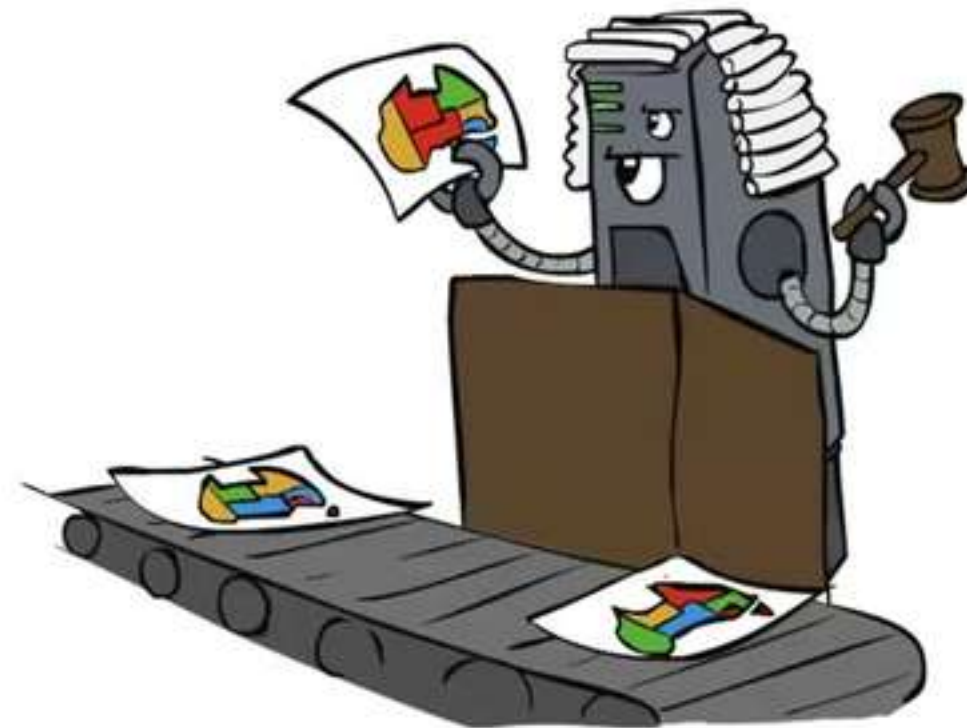
Solving CSPs



Okay, we're done with formulating, we've taken all sorts of different problems, the cryptarithmic, the sudoku, the n Queens, and we made them into CS P's, now we will build a general purpose algorithm for solving CS P's. You ready? Okay. Let's get end of first half and now let's do the second half, let's solve some CS p's. See if questions everything on Zoom okay. Okay no explosions very good. Let's talk about solving CS p's.

Standard Search Formulation

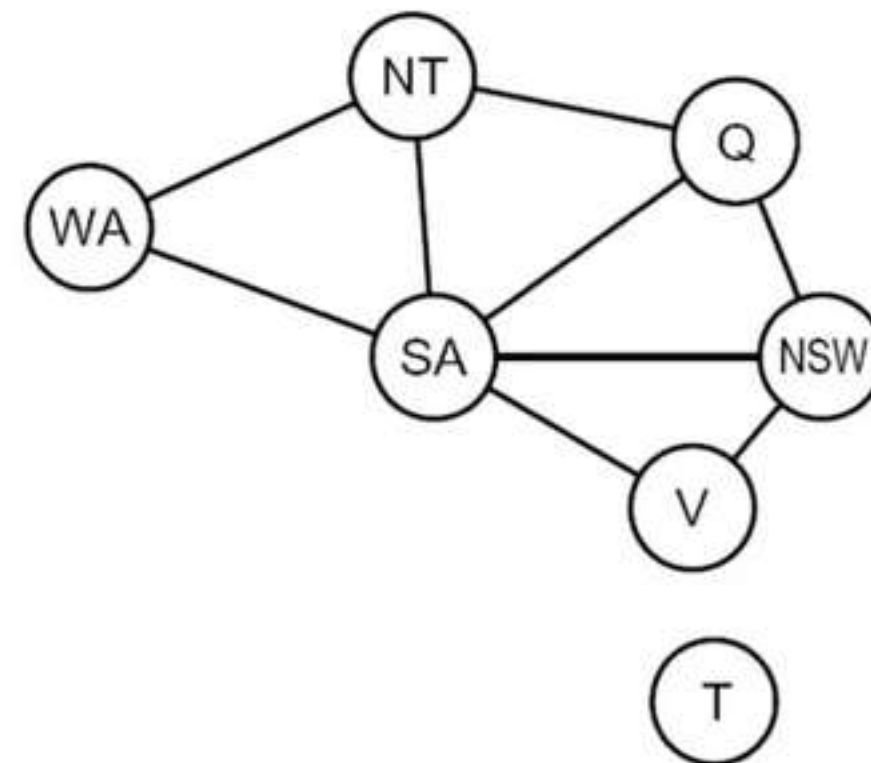
- Standard search formulation of CSPs
- States defined by the values assigned so far (partial assignments)
 - Initial state: the empty assignment, {}
 - Successor function: assign a value to an unassigned variable
 - Goal test: the current assignment is complete and satisfies all constraints
- We'll start with the straightforward, naïve approach, then improve it



Okay, so it turns out CS p's, as mentioned way at the start of today, it's just a special version of Search. We have states, we have a successor function, we have a goal test. So we can. Actually, just use basic search algorithms like d f s b f s to solve CSPS, but it turns out those will not be efficient, and so we'll come up with improvements that only exist for CSPS, where you can look inside the state and see values and variables. But let's do the dumb thing first. So the dumb thing would say What's my initial state? The initial state is an empty map. It's a blank map of Australia. It's a empty assignment where none of the variables have values. What color is Western Australia? Don't know yet. It's blank. What color is Tasmania? Don't know yet. It's blank. And the successor function, which takes a state and gives you all the successor states, would say, let's assign a value to one of the variables that is currently not assigned. So your successor function. I'd say, well, given this state right here. You can color Western Australia red, or you can color Western Australia blue, or you can color it green, or you can color Tasmania red, you can color Tasmania green. So it takes a single variable, assigns it a value. And now you had a new state, and then once you're at that new state, you can call Successor Function. And it will tell you what other assignments are available, and so on, and so forth. And what's the goal Test? Who's the judge? The judge looks at your map or looks at your CSP and asks, is the assignment complete? Did you assign every variable? And also did you satisfy all constraints? And the goal Test gives you a yes or no? Answer? Yes, it's a complete assignment that satisfies all constraints, or No, it is not complete or did not satisfy all constraints. So this is a simple yes no test that you use to check whether or not the CSP has been solved. So you can see, I've taken my CSP and I've made it into a search problem. So it's another case of taking a problem and formulating it as a specific instance of another problem. So this works and go around d s d f s b f s, you will get the answer, but it's really inefficient.

Search Methods

- What would BFS do?



[Demo: coloring -- dfs]

Let's see why. Okay here comes the annotation again. I will try and get this fixed in next week, but until then, let's think about BFS. So BFS, would say, let's start with the empty assignment. The empty assignment is there's no variable. And then we can ask for all of its successors. So what are your successors? Well, your successors are something like, ah, let's do. Queensland is blue, or maybe another successor might be like, Queensland is, or my other colors, red, okay, and then another successor might be

Queensland is green. Okay, very good. And then there's also other successors using the other states. But in the interest of not boring you with my drawing skills, I'll cut it here, okay, so these are the three successors.

Search Methods

What would BFS do?

[]

[Q = 5]

[6]

```
graph TD; WA((WA)) --- NT((NT)); WA --- SA((SA)); NT --- SA; SA --- Q((Q)); SA --- NSW((NSW)); SA --- V((V)); Q --- NSW; NSW --- V; V --- T((T))
```

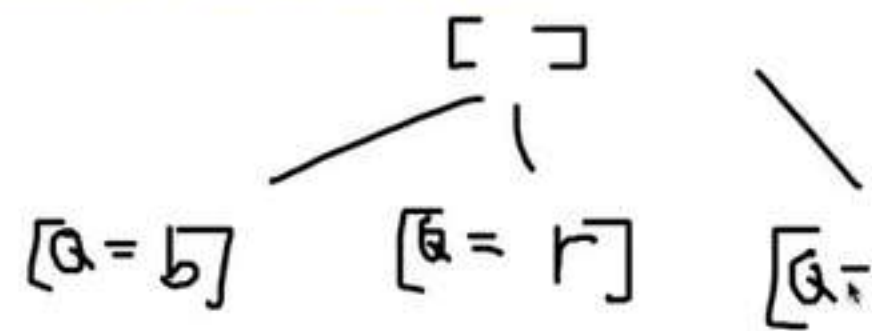
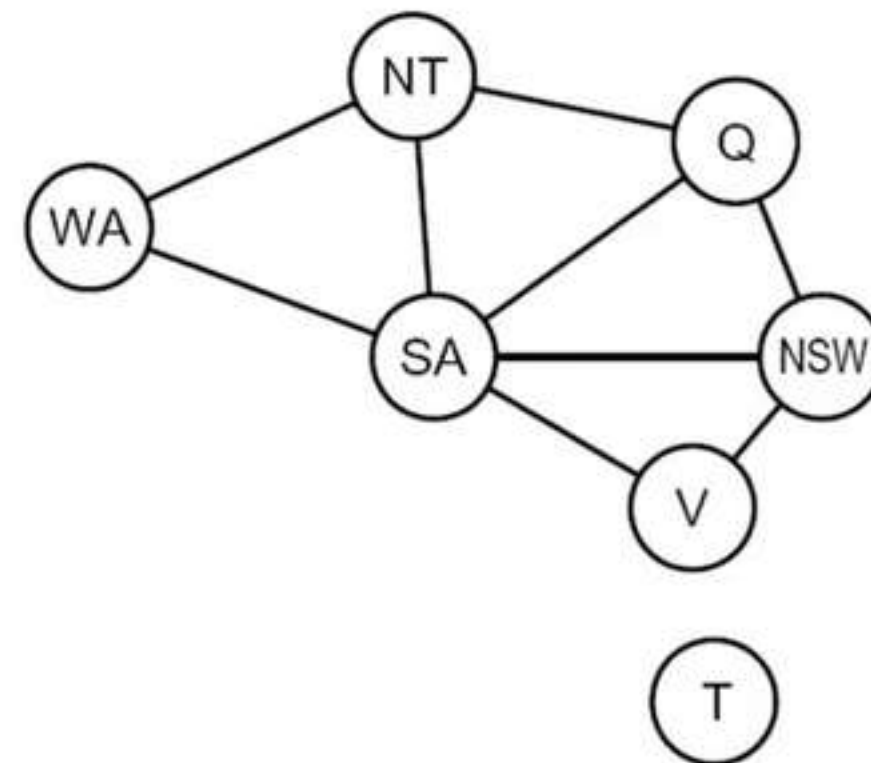
[Demo: coloring -- dfs]

Slide 29 · 32:54

Made by Deckcast. · <https://deckcast.org>

Search Methods

- What would BFS do?



[Demo: coloring -- dfs]

And then there's also other successors using the other states. But in the interest of not boring you with my drawing skills, I'll cut it here, okay, so these are the three successors. What does Breadth First Search do? Well, it picks one of these and expands it, so let's pick this one. We'll expand it. Okay what are the successors of this one? Well, the successor would be something like, Queensland equals blue, and let's do another one, ah, Tasmania equals blue.

That's another successor, and another successor here might be okay, Queensland equals blue and Tasmania equals green. Okay, maybe you're starting to see a problem here, because what will BFS do? BFS will continue expanding and it'll expand all these. Okay good we'll expand these and then we'll expand these.

Search Methods

What would BFS do?

[]

[Q = b]

[Q = r]

[Q = g]

[Q = b]

[T = b]

[Q = r]

WA

NT

Q

SA

NSW

V

T

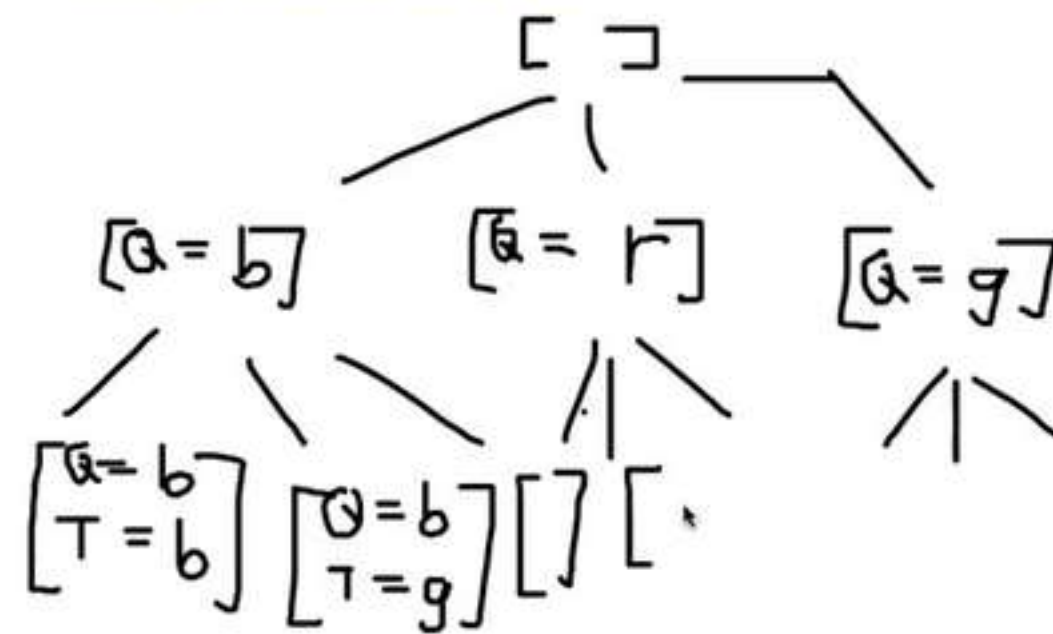
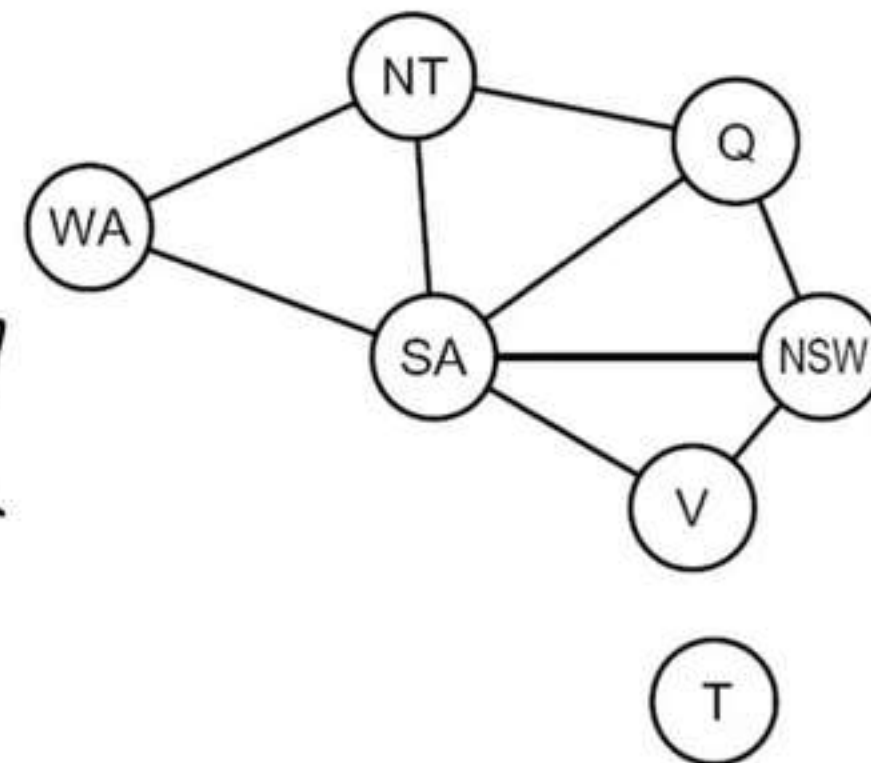
[Demo: coloring -- dfs]

Slide 31 · 33:36

Made by Deckcast. · <https://deckcast.org>

Search Methods

What would BFS do?

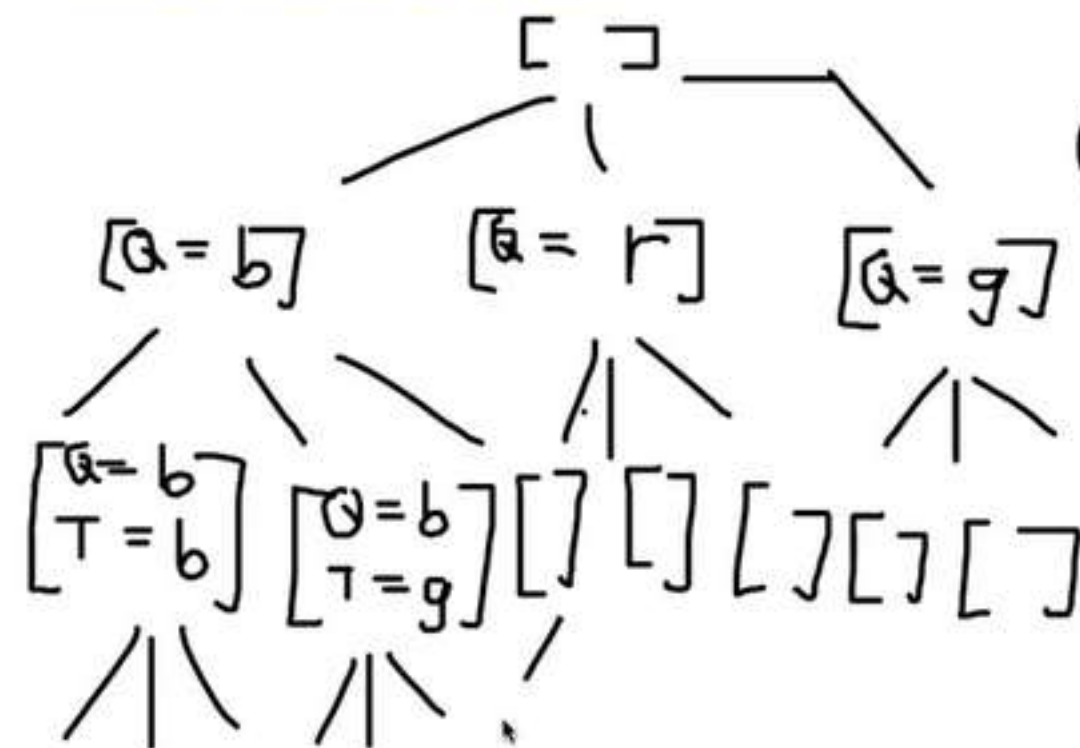
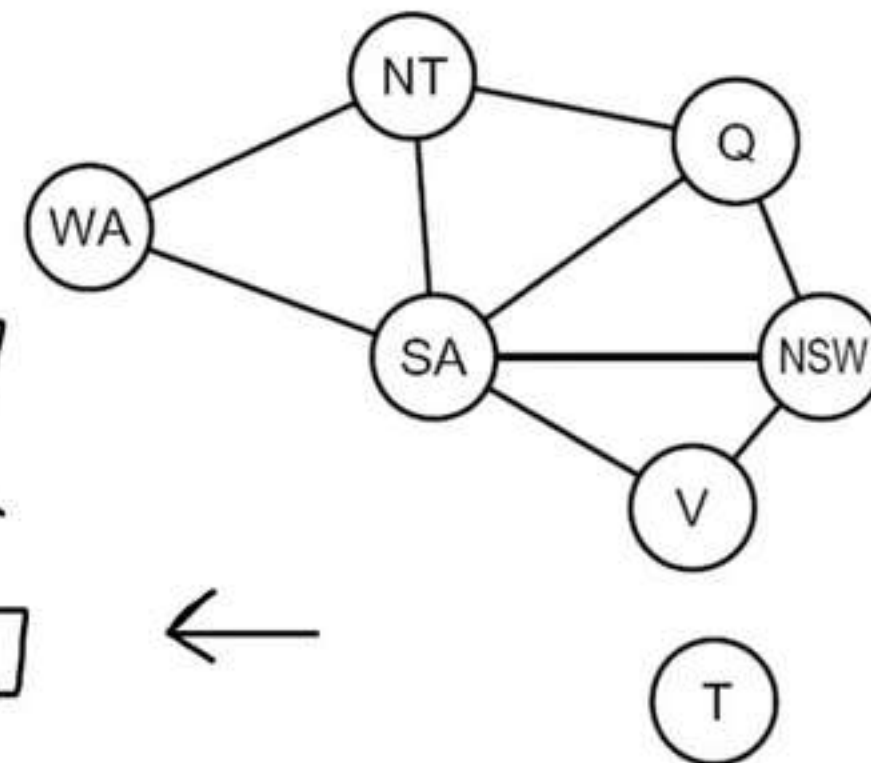


[Demo: coloring -- dfs]

I'm not gonna draw them because I don't have time, but each of these will be an assignment of two variables. Okay what does BFS do next? BFS will pick one of these up. Q and then expand. So remember BFS sort of stripes through, ah, from top to bottom, going layer by layer, right? So we do the first layer, we did the second layer, we did the third layer. Okay where are the solutions to this problem? Are there any solutions at this layer? No, because at this layer, only two of the variables are assigned. So I can guarantee you there's no answers here. A solution requires every state to be colored, and here only two states are colored, only two variables have values to use the general term. Okay so we'll go and expand all these.

Search Methods

- What would BFS do?

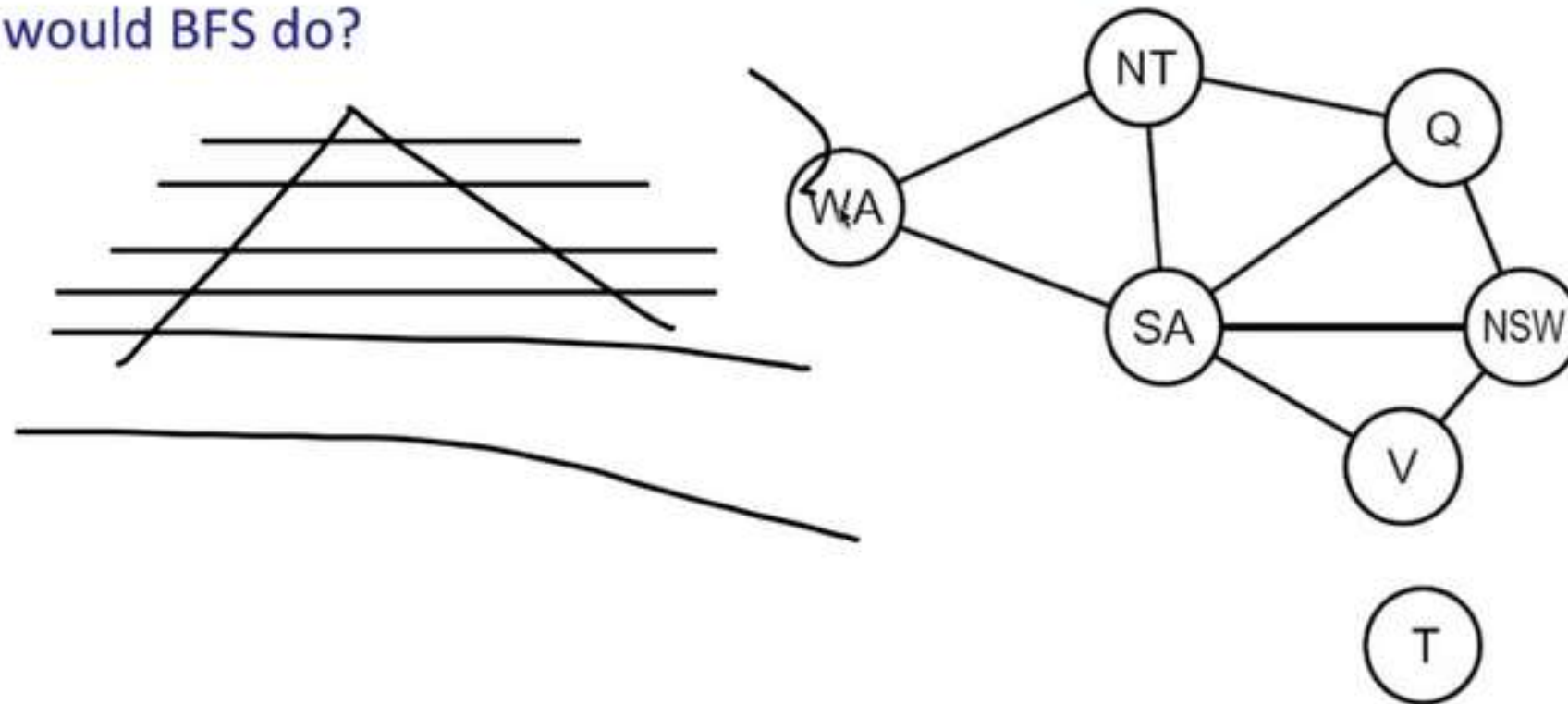


[Demo: coloring -- dfs]

Do you think we'll find any solutions at the next layer? Any solutions here? Nope because the solution or the next. Layer will only have states with three variables assigned. Where do all the solutions live in the BFS search tree? They live at the bottom because that's the place where all the variables get assigned. So if you think about your BFS cartoon, go away. Okay if you think about your big BFS tree, you're stripping through layer by layer by layer. Okay no solutions, no solutions. Where are all the solutions? All the solutions live down here.

Search Methods

- What would BFS do?



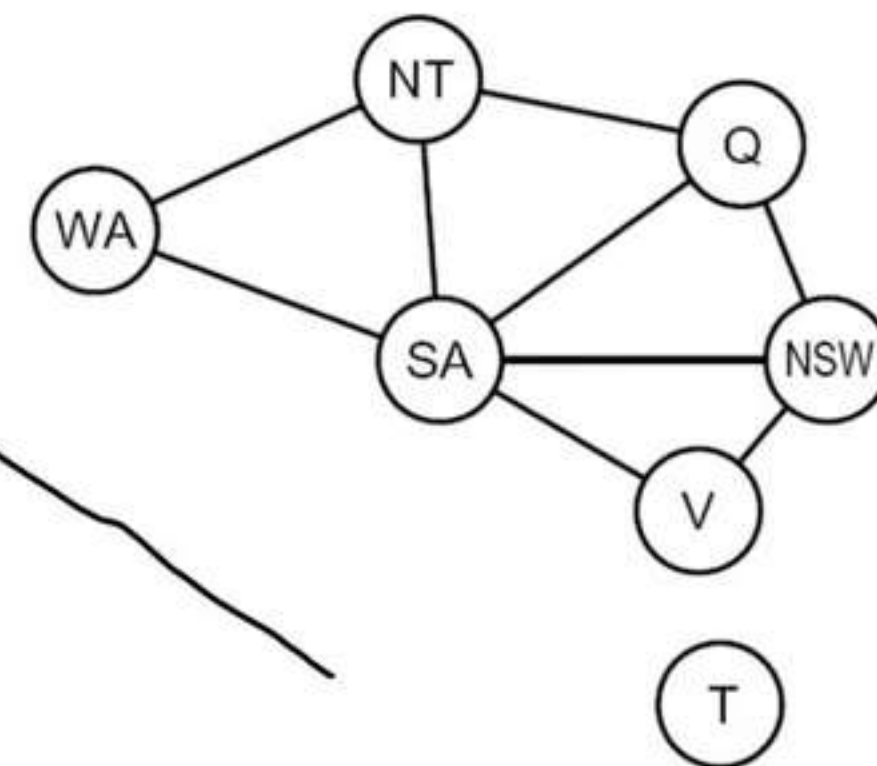
[Demo: coloring -- dfs]

So BFS just wasted all this time looking everywhere where the solution is, not before ever looking at a single solution. So, BFS, at first, when we saw Search last time, I felt pretty good for driving directions or something. For CSPs, oh no, BFS is like the worst possible choice, because you're gonna look everywhere where there's no solutions before you ever start looking at this bottom layer, where some of those solutions live. So it turns out for this one case, ah, where's my next slide? For this one case? It's actually the case that DFS is a little bit better. Why is that?

Search Methods

- What would BFS do?

- What would DFS do?



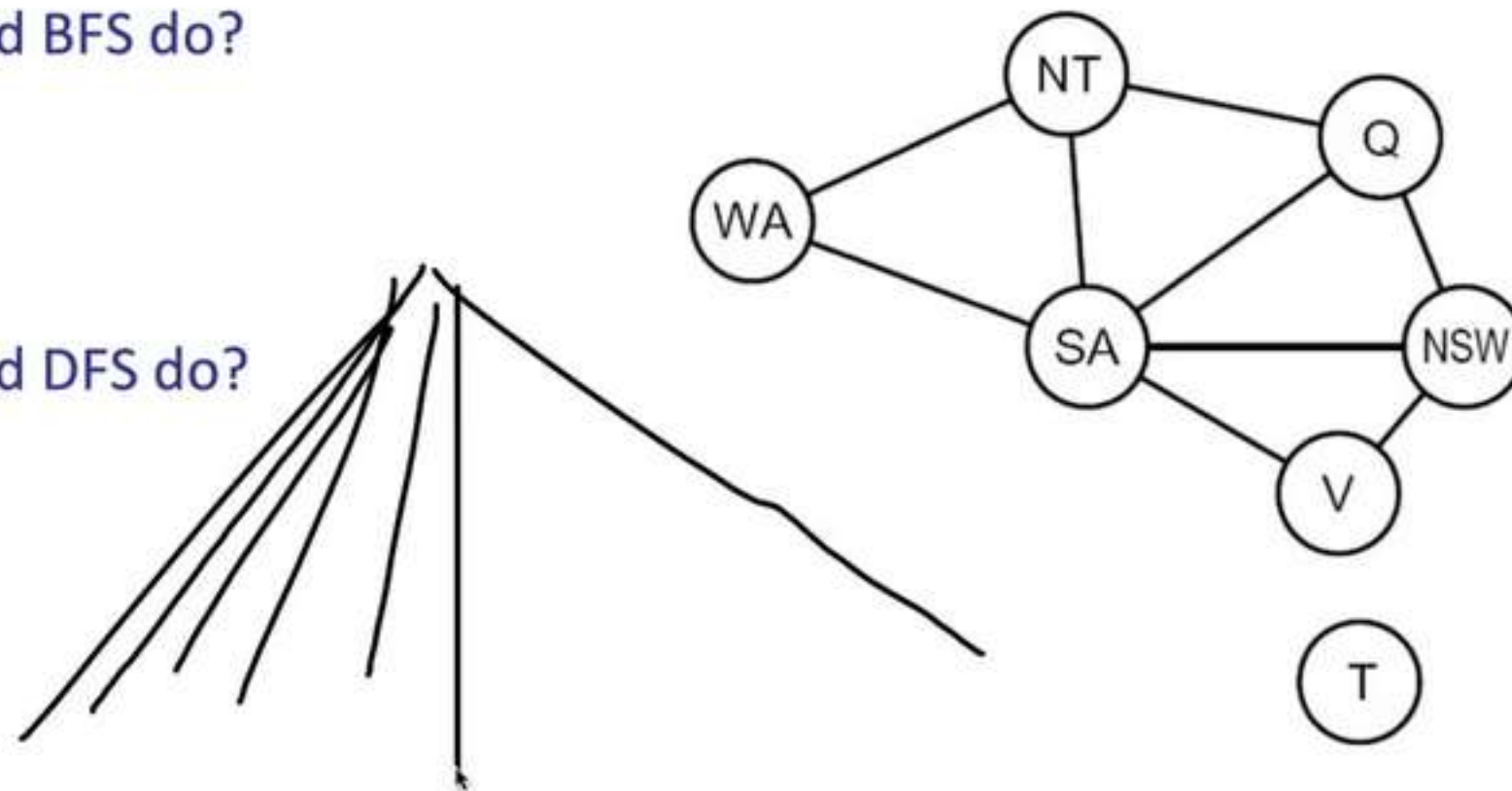
[Demo: coloring -- dfs]

Because if you think about DFS. And the search tree, what Dfs D-F-S. Do? At least it dives down here and tries to find some solutions first, because you know, the solutions live at the bottom. That's the place where all the variables get assigned. So at least DFS.

Search Methods

- What would BFS do?

- What would DFS do?

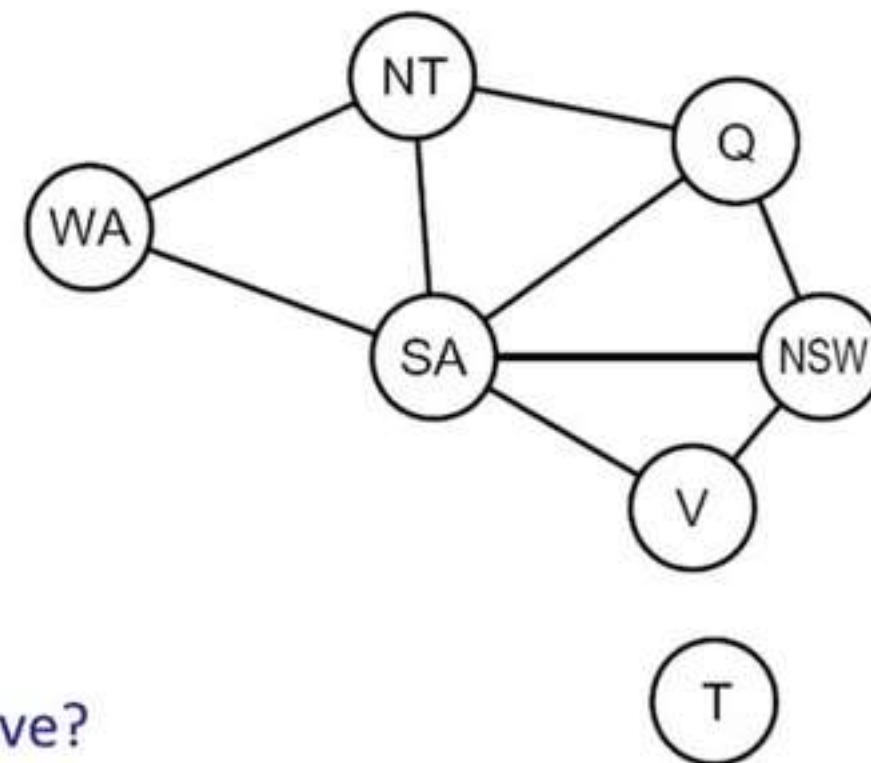


[Demo: coloring -- dfs]

Comes to the bottom and tries out solutions. Whereas BFS. Tried everywhere where it's not before looking. For solutions. So this is actually a case where DFS is better. So last time we were laughing at DFS being like, oh, you give us all these crazy solutions, but this is actually a case where DFS, does the right thing. It at least goes down and tries to find solutions that work.

Search Methods

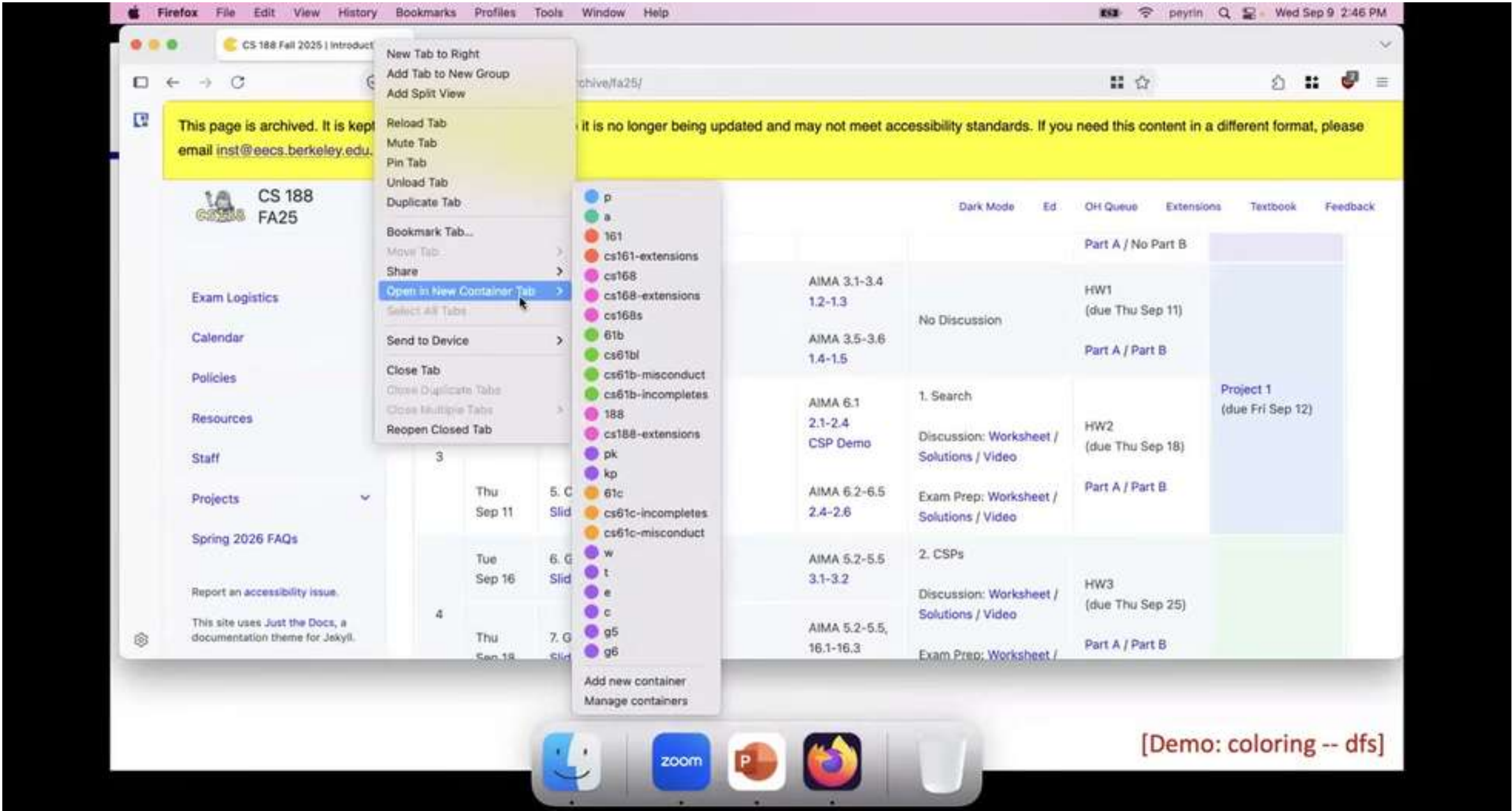
- What would BFS do?
- What would DFS do?
- What problems does naïve search have?



[Demo: coloring -- dfs]

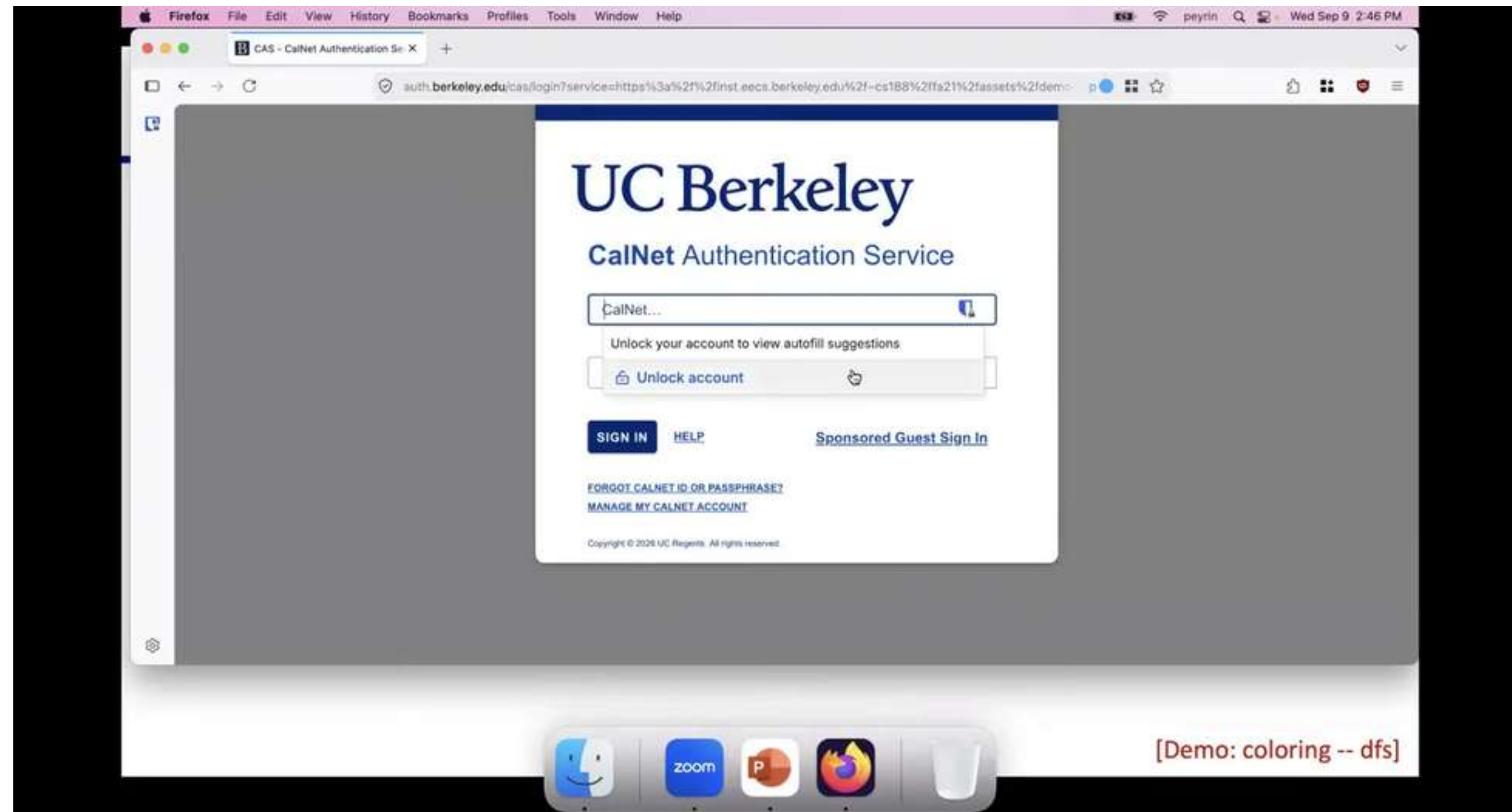
Okay great. So let's try using, let's try using DFS to solve this c s p. I have a demo. I forgot to pull it up earlier, so you're gonna have to bear with me while I pull it up. Okay. No Firefox. I don't care about what's new. I just wanna see my CSP demo. I'll put this on the website at some point. I promise.

Okay. But point being ah I have a c s p demo. It's basically. The map coloring problem from earlier, even though, remember, not all CSPs are maps. Where are you? CSP demo loading?



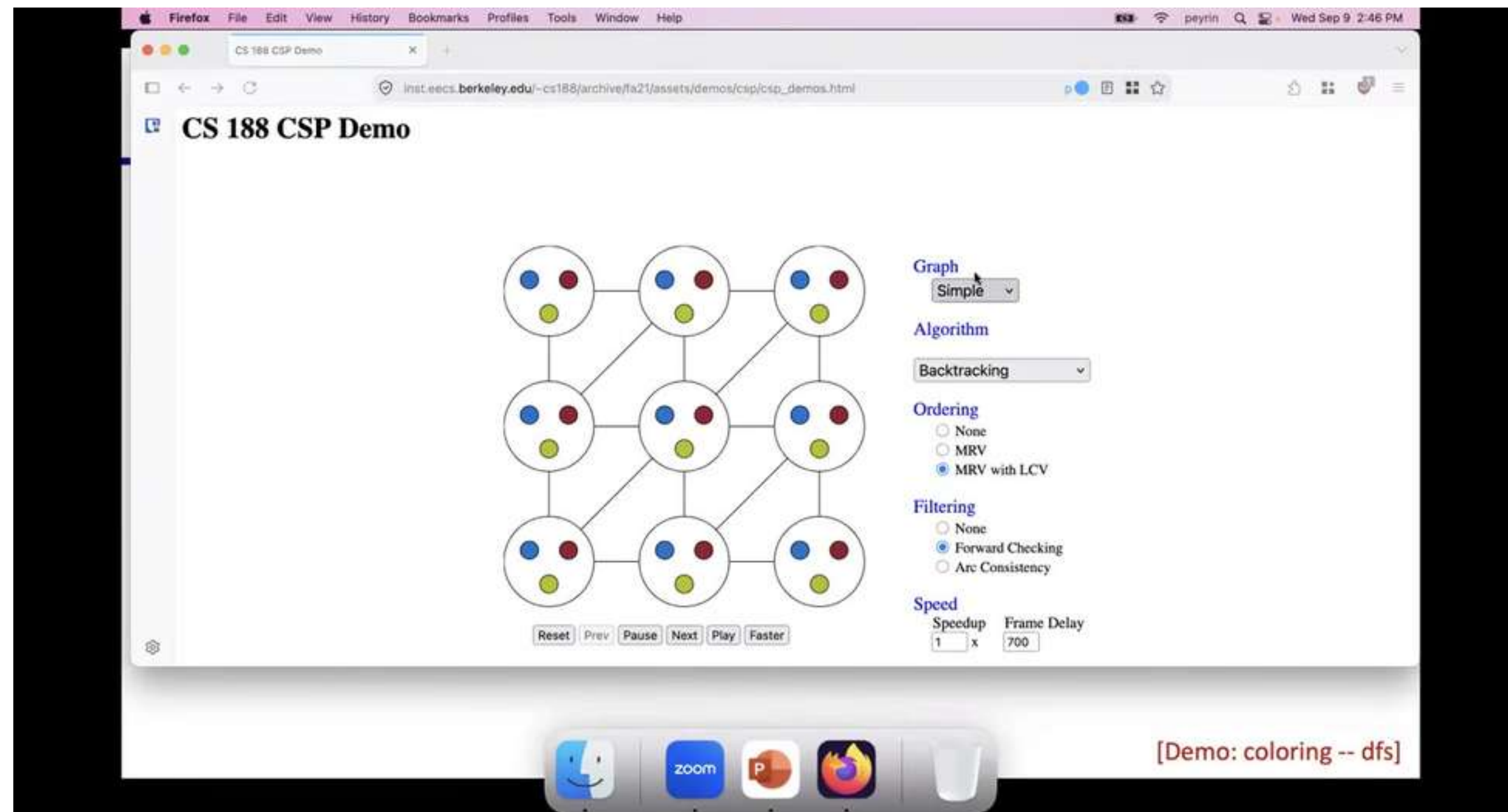
Slide 38 · 36:36

No, I don't want to login, I have to login, ah, what do I do.
Don't look at my password. No, I don't want to do two factor.
Auth. Okay, but I'll do it anyway. Okay, very good.

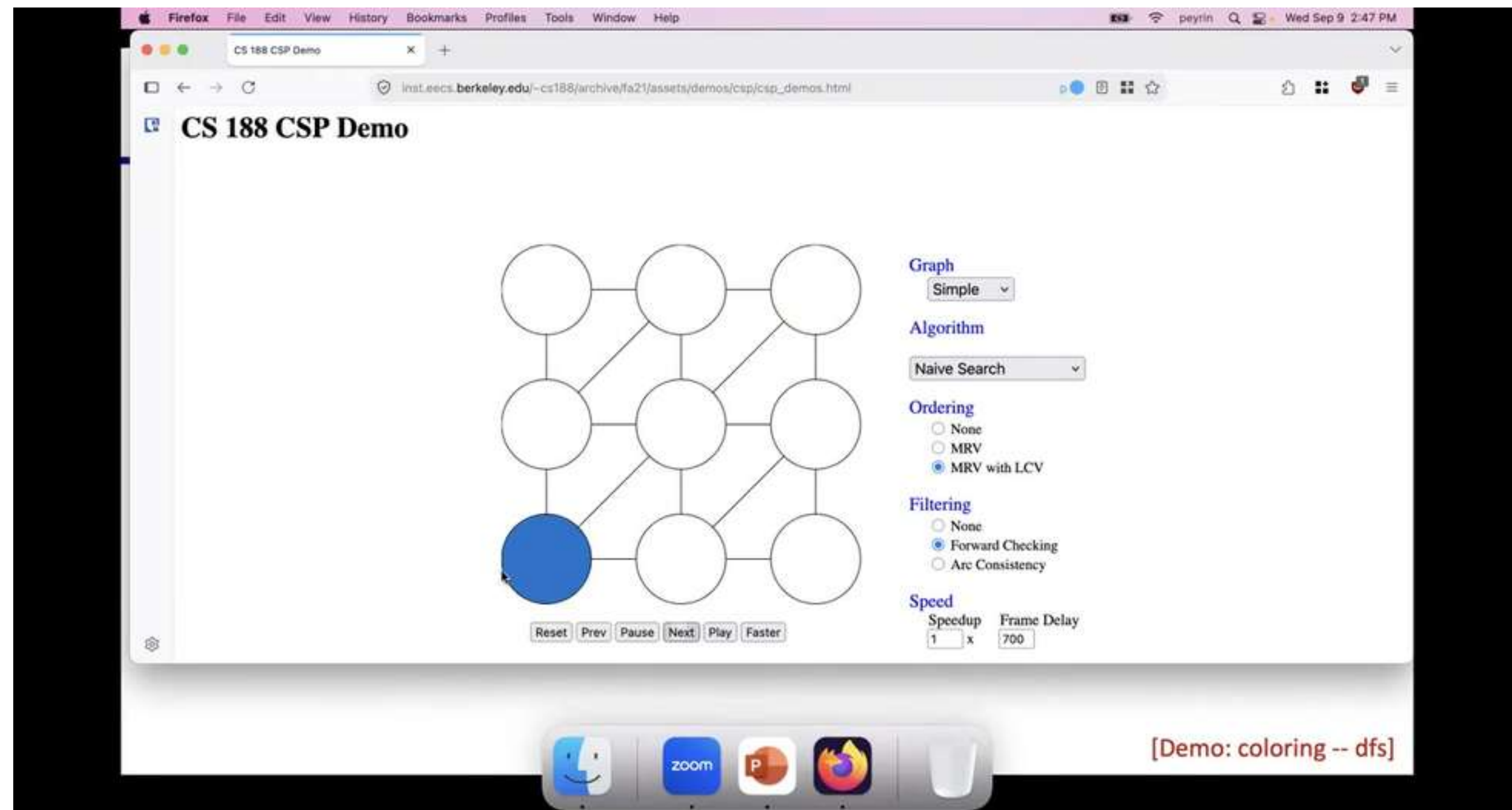


Slide 39 · 36:48

CSP. Demo, Great, that wasn't as bad as I thought. Okay, so let's do the basic thing, let's do naive search. Okay, so starting with naive search. And people on Zoom can see this, people here can see it. Okay, very good. Okay, so let's start assigning. Variables. This is the starting state. This is the state where everything is unassigned. So let's go on the fringe. Let's pick a state and let's expand it. Let's look at one of its successors.



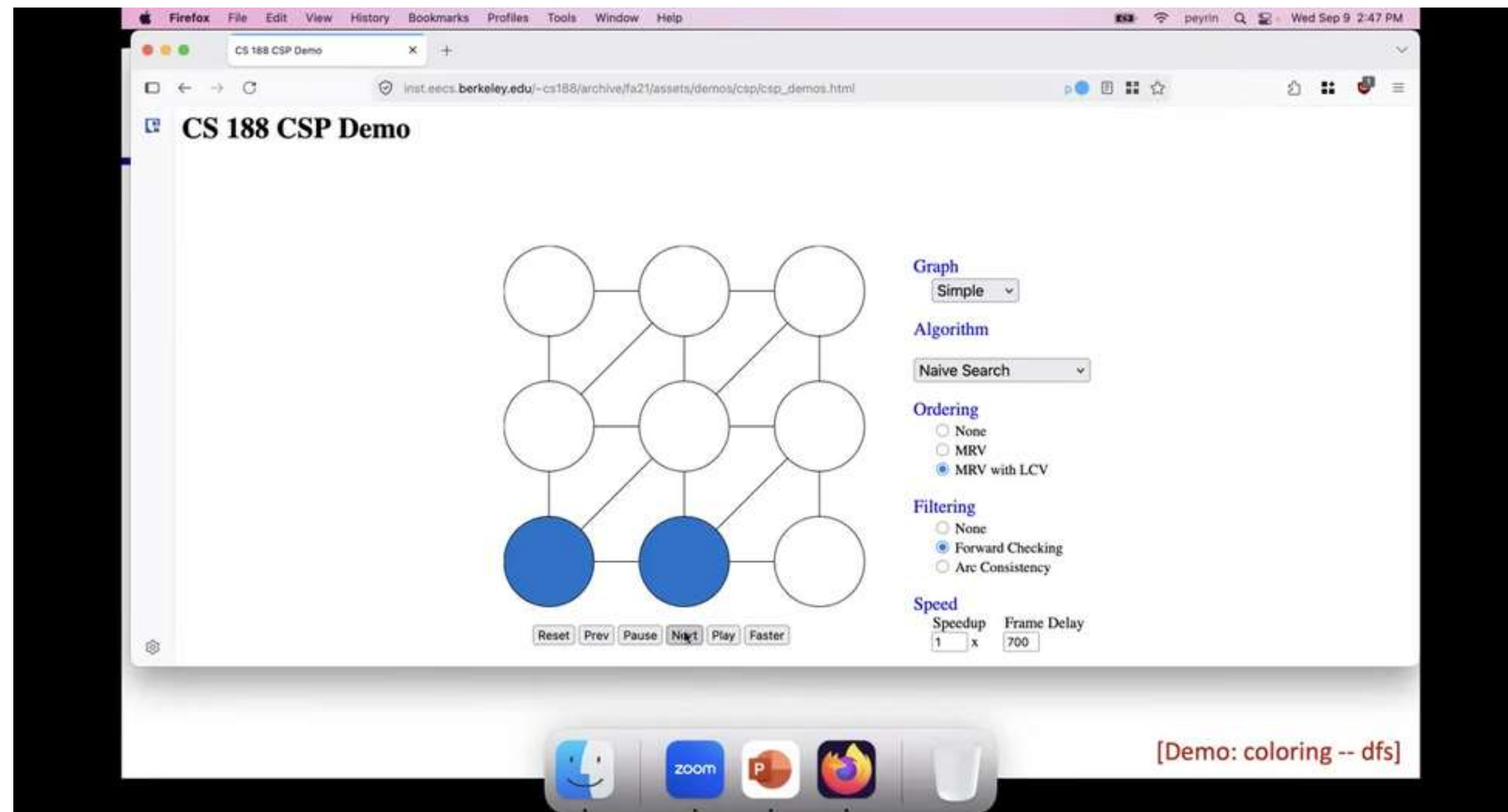
Slide 40 · 37:06



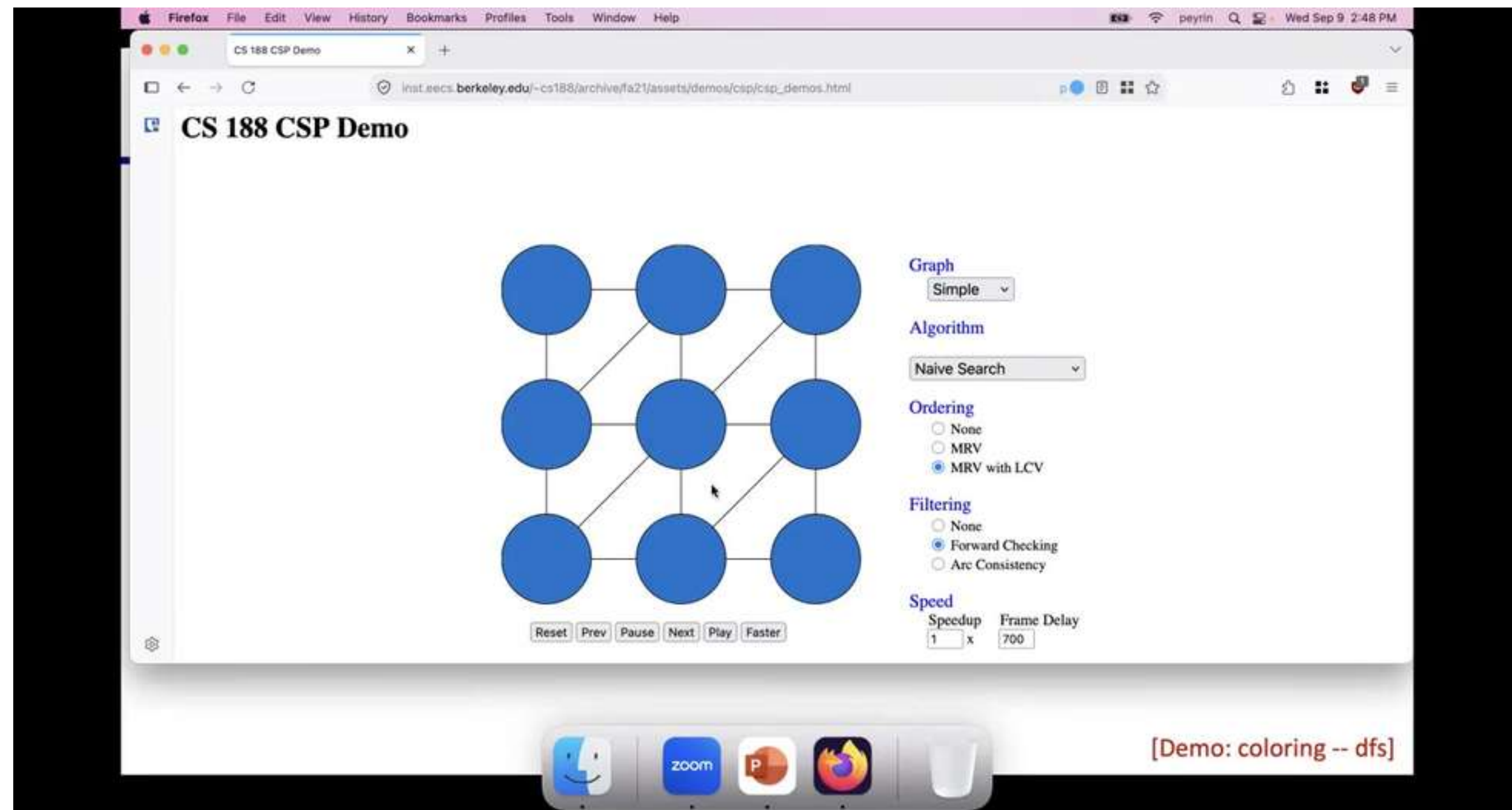
So the successor might be something like blue. I colored one of the nodes blue. I took a variable and I assigned it a value. So now I have this on the fringe and a bunch of other stuff on the fringe. Remember, the thing I'm not showing here is that in a search problem, there's a big fringe of unexplored answers. But let's pick one of them off the fringe. Let's say I pick this one. I pick it up. Are you the goal? Have I assigned all variables and met all constraints? No. This is not a complete assignment. So I have to expand this and look at one of its successors. So let's go and get one of its successors. So one of its success. Would be coloring one of these other nodes with a color assigning another value to one of the variables.

Slide 41 · 37:30

So let's do it. You ready? Blue. Uh-oh. Okay so now we have some other stuff on the fringe. We go back on the fringe. We pick the deepest thing, which means it has the most assignments so far, most actions taken. So I picked this up. Are you the goal? No. So I will look for one of your successors, or I'll expand you. I'll put all your successors on the fringe, and one of your successors will be blue. Okay. Blue, blue, blue because DFS goes to the deepest thing next. Blue, blue, blue.



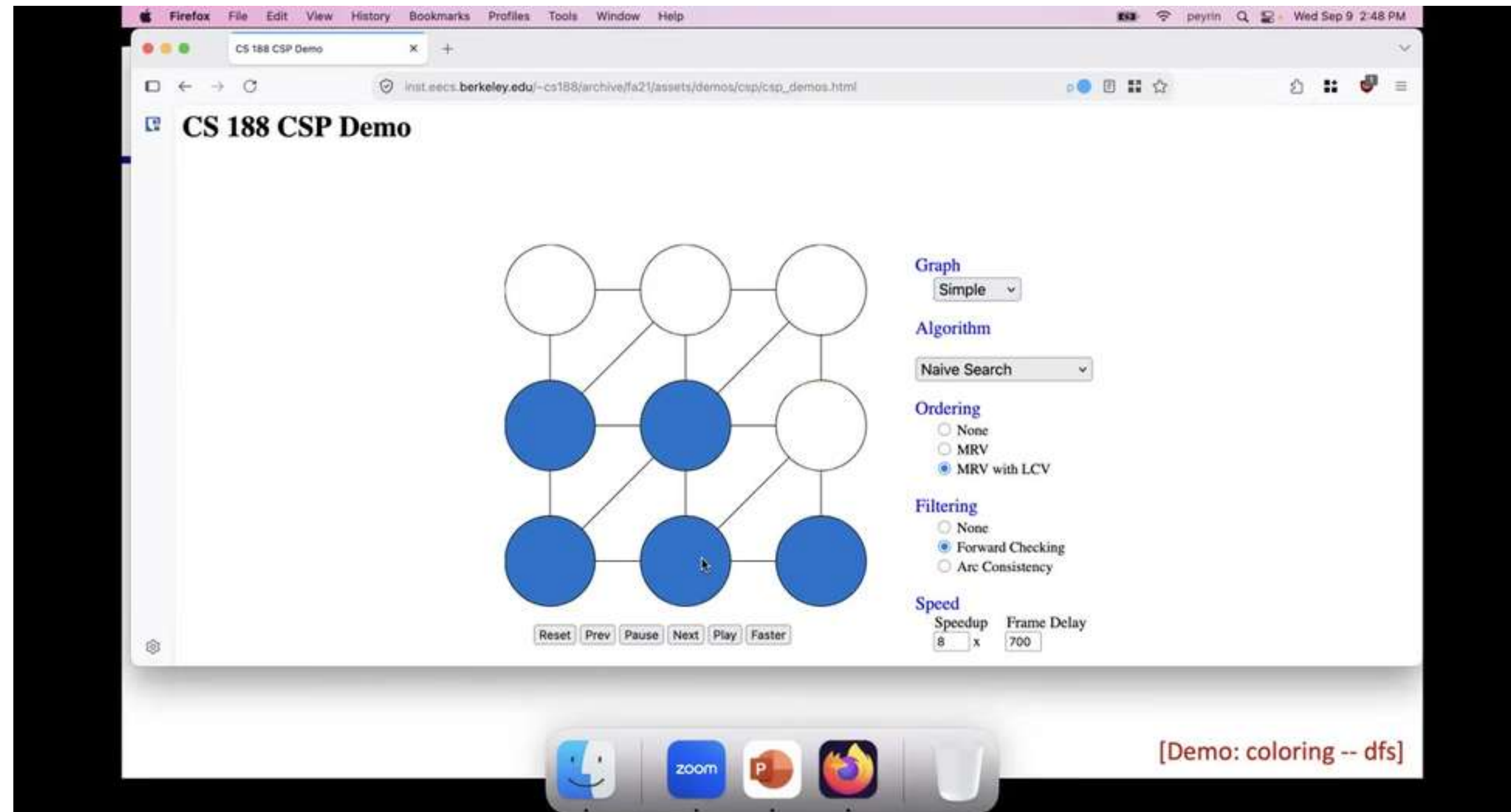
Slide 42 · 38:06



Okay, now I pick this up and I check, are you the goal? No you're not the goal because these. Lines, they represent the constraints, and this is clearly not a valid map coloring because a lot of these adjacent nodes connected by lines have the same color. So what do I do? I go back on the queue. So I go back on the queue. This is another state that was on the queue. Can I check this one? Are you the goal? No. Okay. Are you the goal? No. Is all go back check this one. Are you the goal? No. Go over this. Okay. So we could do this. Okay. In the interest of doing anything else today, I'm gonna stop this. But if you look at this, it's kind of ridiculous.

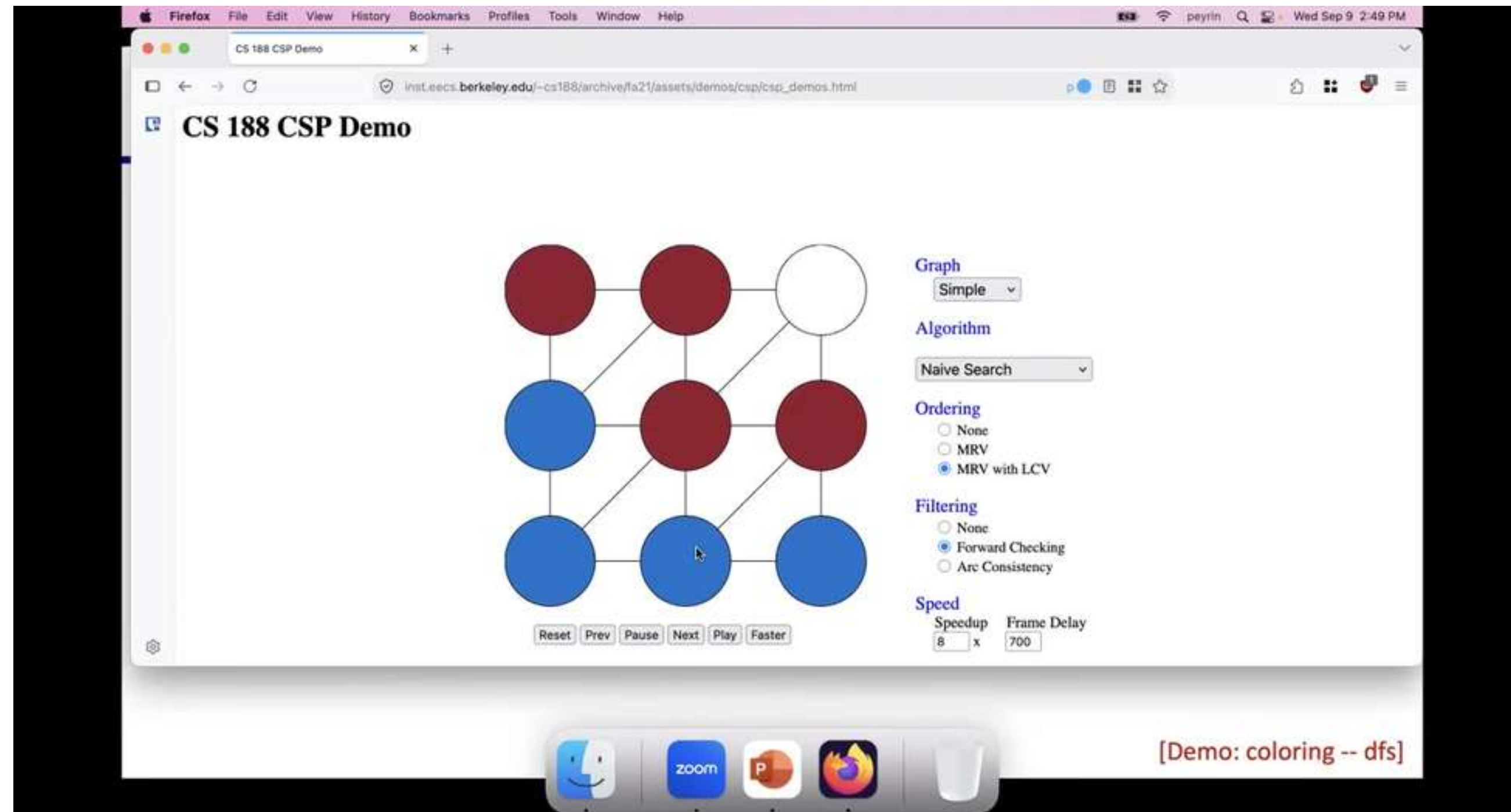
Slide 43 · 38:36

Like why would you do this? Why would you do this when you know that you were already busted right here. So this is the problem with Naive Search. Naive Search will work. I run this for like ten years. It will get to the solution, still not there yet, not even close, right? But this will eventually get to the solution.



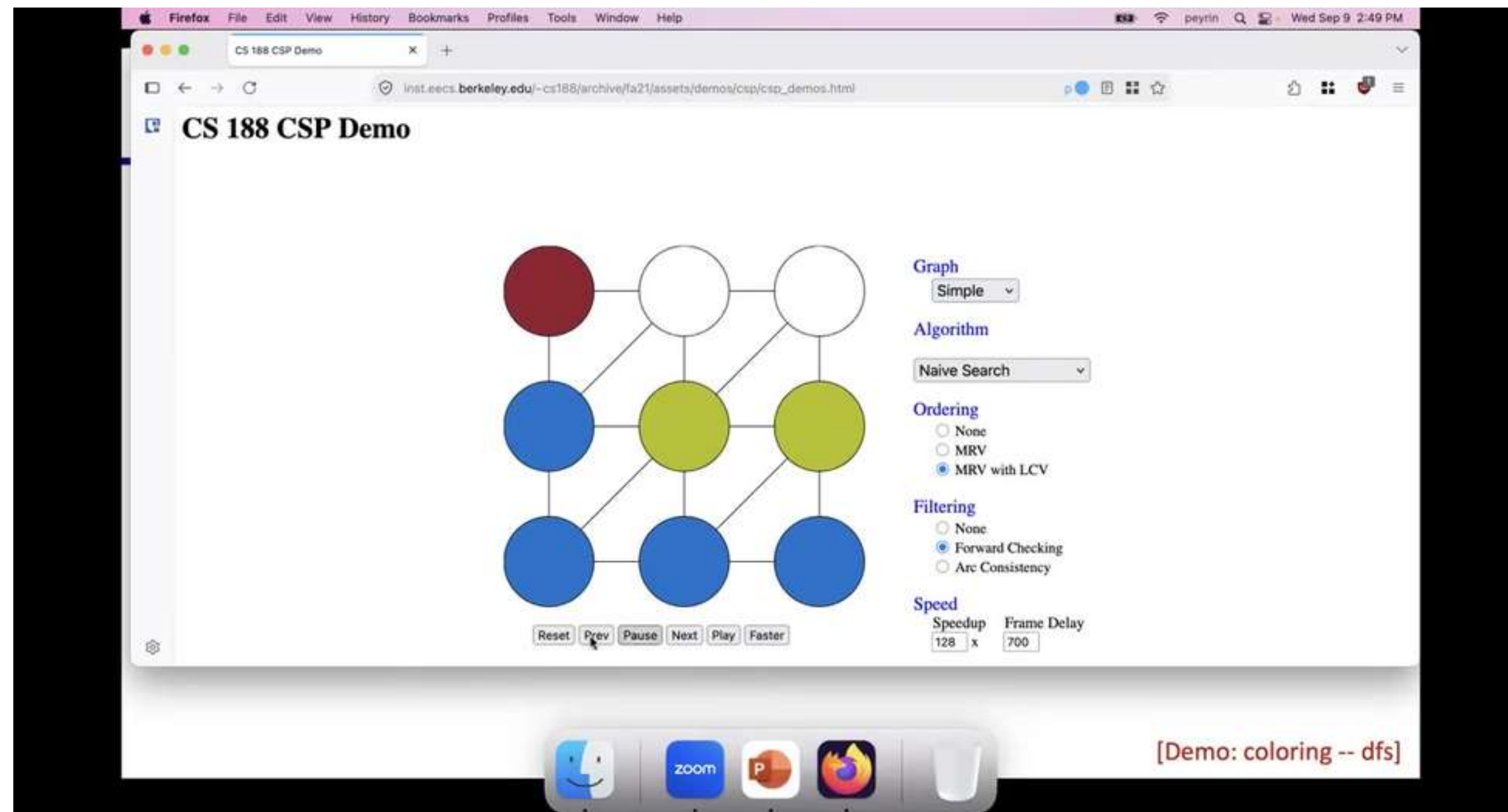
Slide 44 · 39:12

But the problem with why this isn't working is because Naive Search isn't looking inside the states, it's treating the states as a black box. It doesn't know that there are values and variables and assignments and constraints, but by decomposing the problem into these smaller pieces, like, there's a constraint and there's a variable and there's a value. I can get more intelligent solutions than whatever this is doing faster.



Slide 45 · 39:30

Okay, still, no good. Okay, so that didn't work, so that's naive search. It doesn't work because it doesn't think about the problem at all, it just takes your successor function.

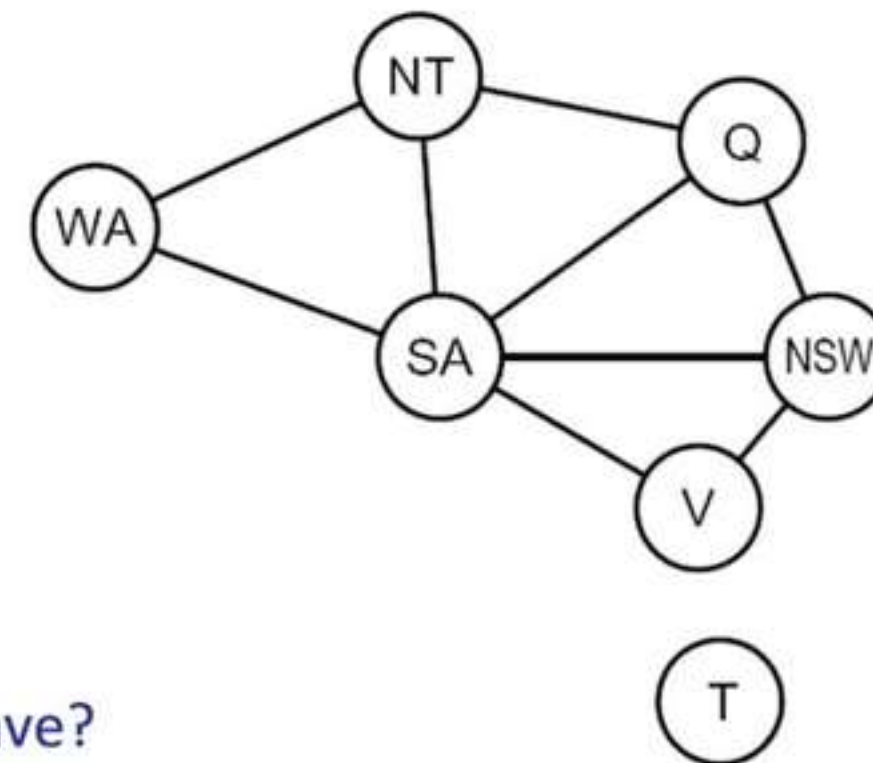


Slide 46 · 39:48

It doesn't work because it doesn't think about the problem at all, it just takes your successor function. And your state space and just runs it. Let's do better. It's a good start.

Search Methods

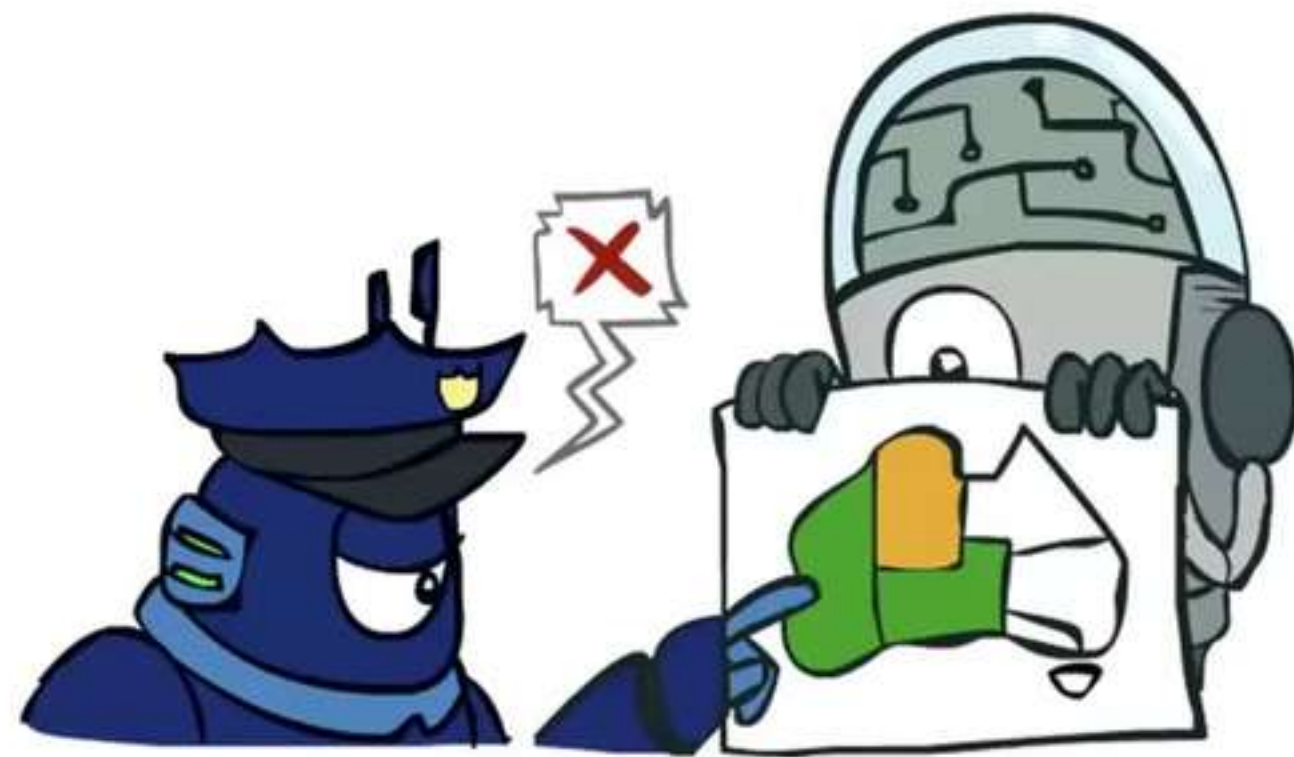
- What would BFS do?
- What would DFS do?
- What problems does naïve search have?



[Demo: coloring -- dfs]

We'll solve the problem, but let's do better. Okay. So to do better, we'll introduce something called backtracking search. Now honestly, I don't really like this name. I don't know who came up with it. I think it's a little bit of a misnomer, but that's what they called it. So that's what I have to call it.

Backtracking Search



Slide 48 · 40:06

Oh well. Okay. But this is a better algorithm. So we're gonna make two improvements. So we saw DFS earlier is terrible. Even though it technically works. Let's make two very big improvements.

Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs

Slide 49 · 40:18

Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
 - Variable assignments are commutative, so fix ordering
 - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
 - Only need to consider assignments to a single variable at each step

So the first big improvement is I will actually fix an ordering. So I will say, I'm going to assign this node first, and then this node first, and then this node first. This reduces the fan out of my. Search Tree. Because if you think back to the original naive version, the successors might be. I'm gonna color Western Australia, or I'm gonna color Queensland, or I'm gonna color Tasmania. Here, I'm gonna say, you have to color w a first, then you have to color NT first, and so on, and so forth. And this is fine. Because, remember, we don't care how you get to the answer. Remember the Detective Gem, the Detective Robot and the Gem from earlier. We don't care how you get to the answer, we just care that you get there. So I can fix an ordering and that's fine. The order of how I get to the answer doesn't matter. That'll save me some time. Because I don't have to consider twelve different ways to get to the same answer, I can just consider one way to get to the answer.

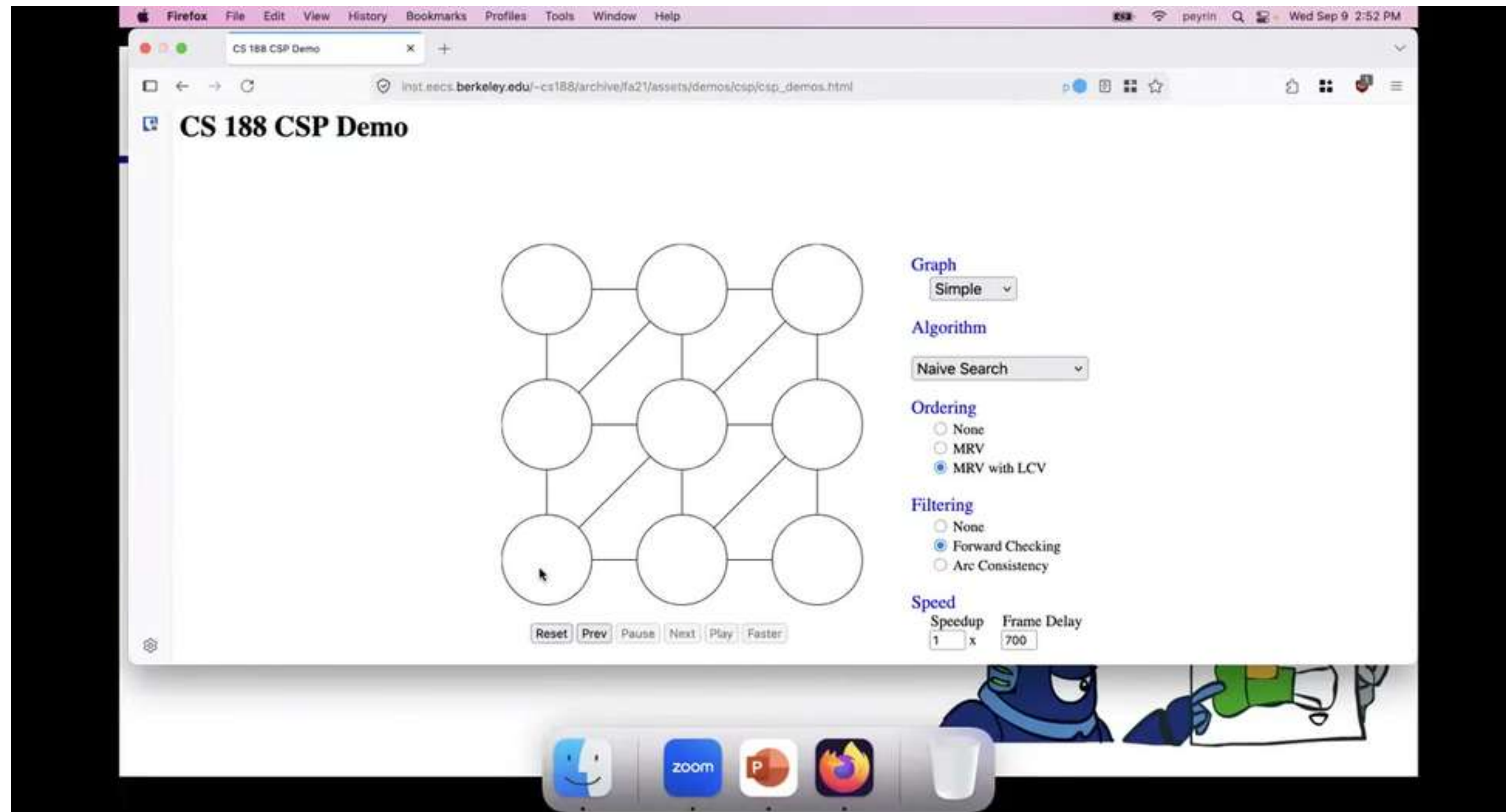
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
 - Variable assignments are commutative, so fix ordering
 - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
 - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
 - I.e. consider only values which do not conflict with previous assignments
 - Might have to do some computation to check the constraints
 - "Incremental goal test"



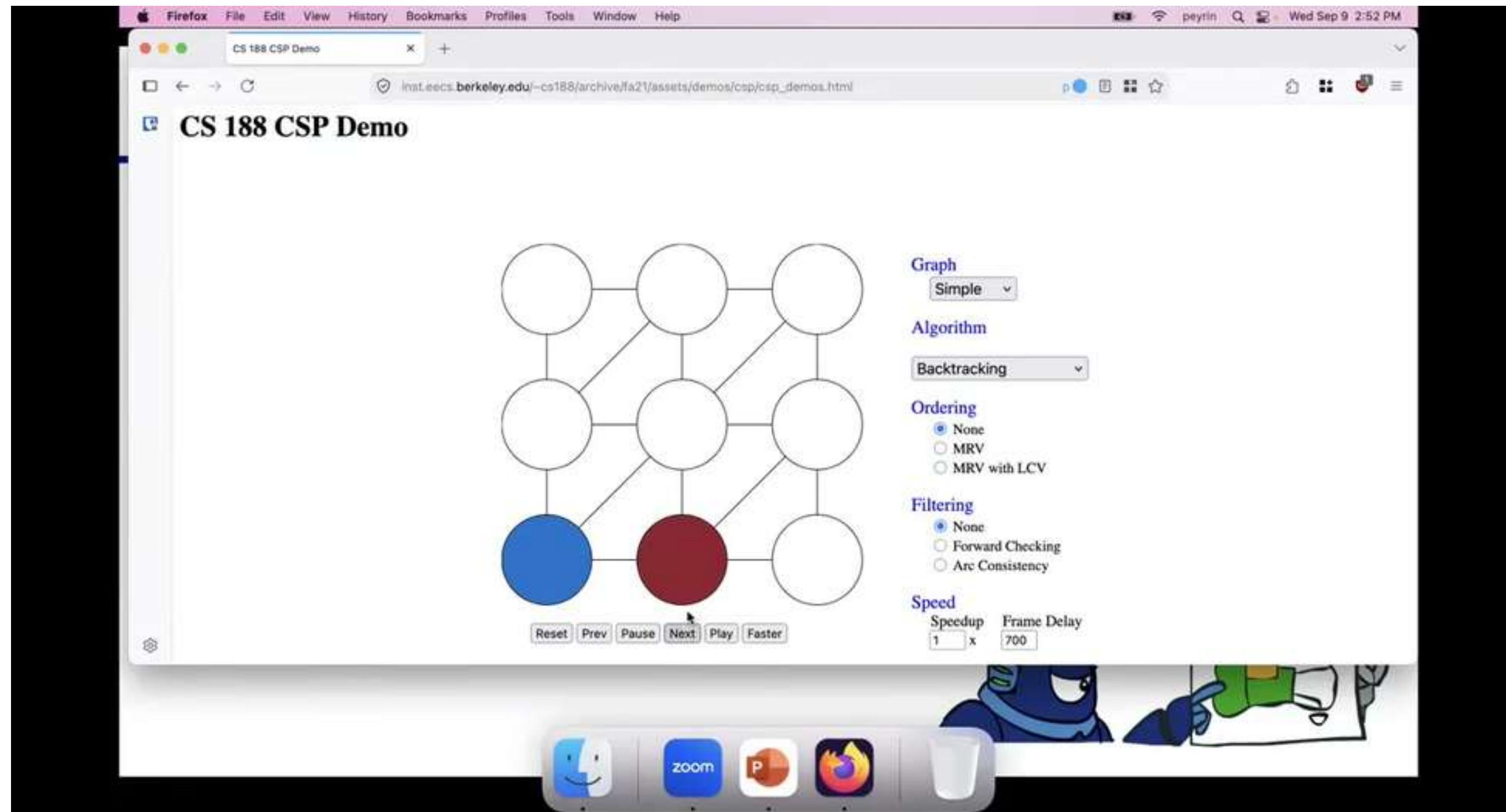
And the second, big improvement, and this is where we're going to start opening up the problem. In a way that search could not is, we're going to actually check the constraints as you go. So instead of waiting until the end, where everything is blue and say, are you a goal? Oh no, you're not a goal, we're gonna check as we go. And so now, unlike the sort of dummy algorithm you saw from earlier, you can actually check the constraints as you go and check, because if you see a constraint that's been failed, you know that beneath that search tree, all of its children are not going to be valid paths to goal. And this is useful because in a c s p, once you violate a constraint, you cannot unviolate it, there's no way to go back. Once you color two nodes blue in your map, if they're next to each other, there's no way to fix that. You've already colored them blue. [other] So you know that everything underneath that tree is doomed to failure, and you can fail earlier and go explore other parts of your tree. So what we've done here is it's still DFS. We didn't change that at all. Still a successor function, still a goal test. But the difference is, instead of the goal test being this judge, the black box that either says yes or no, the Goal Test has now been decomposed into a bunch of little rules. And you can check through all like Rule one's good, Rule two's good oh, Rule three is not good and I will never fix rule three underneath this tree. [afraid] So I can stop searching this tree and go search other parts of my big search tree. So it's kinda like a goal test, but you're checking the goal test little by little instead of waiting until the very end. And this will make a big improvement. So let's go watch it. So it's not the best name, even the slides agree with me, it's not the best name. So depth first search, if you add these two ideas, become something called backtracking search.

Let's see if it does any better. So no more naive. Let's do backtracking search. Okay. I do not want any of these fancy things. So we'll start with a totally blank grid. Here we go. Okay blue. So far so good. This is still depth first search. We're still building a little fringe.



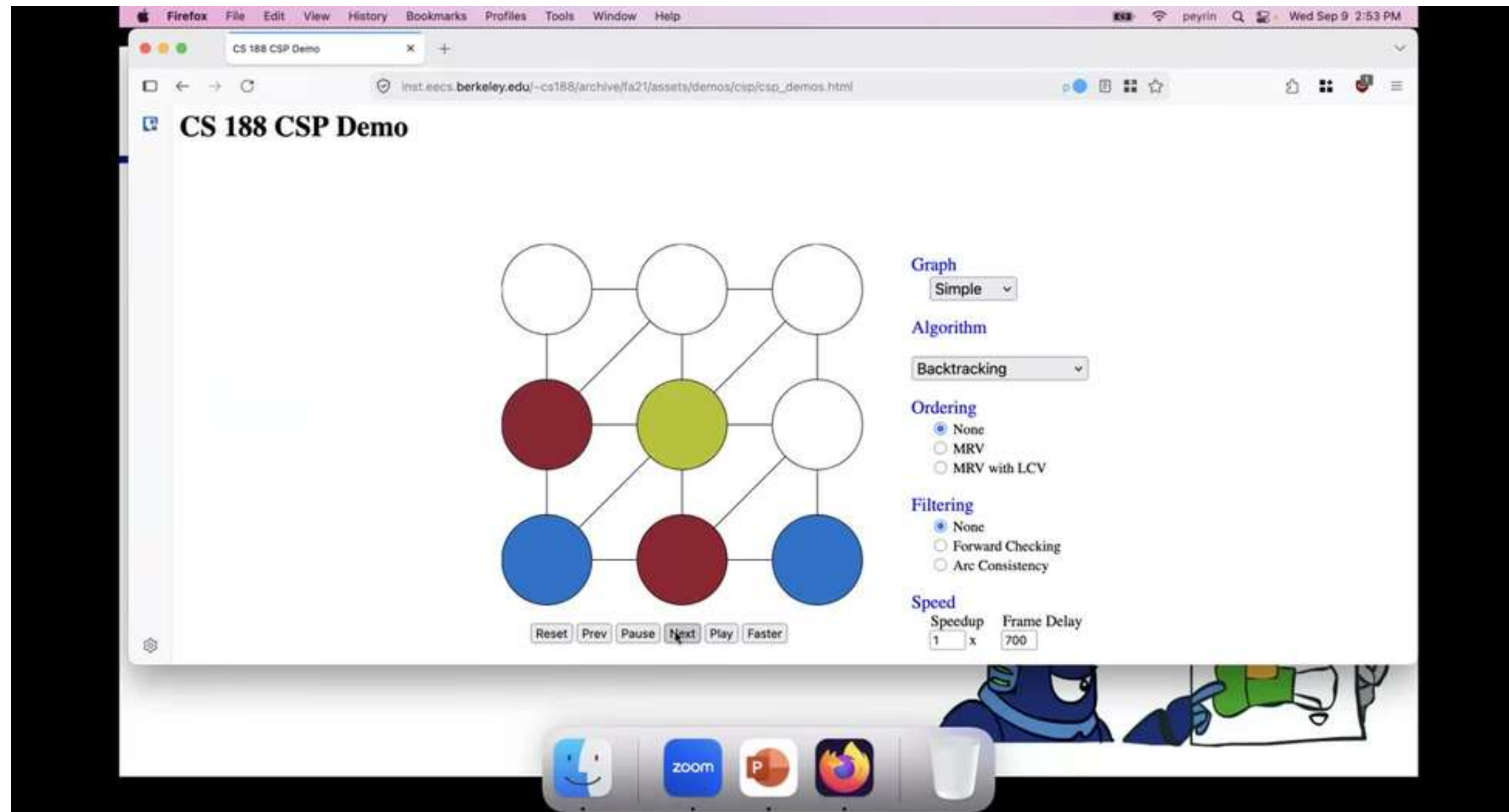
Slide 52 · 42:48

Okay what's coming next? Red. Great. Why did we pick red here? Because if we had picked blue, that would have caused a constraint violation, and once you violate the constraint, you can't go back. So Backtracking Search will notice. That entire search tree is doomed to failure, and unlike Naive Search, which would have checked the whole tree because it didn't know we can now fail earlier, and say this sub tree, it sucks, let's go check this sub tree where the second node is red. Okay, what's next blue?



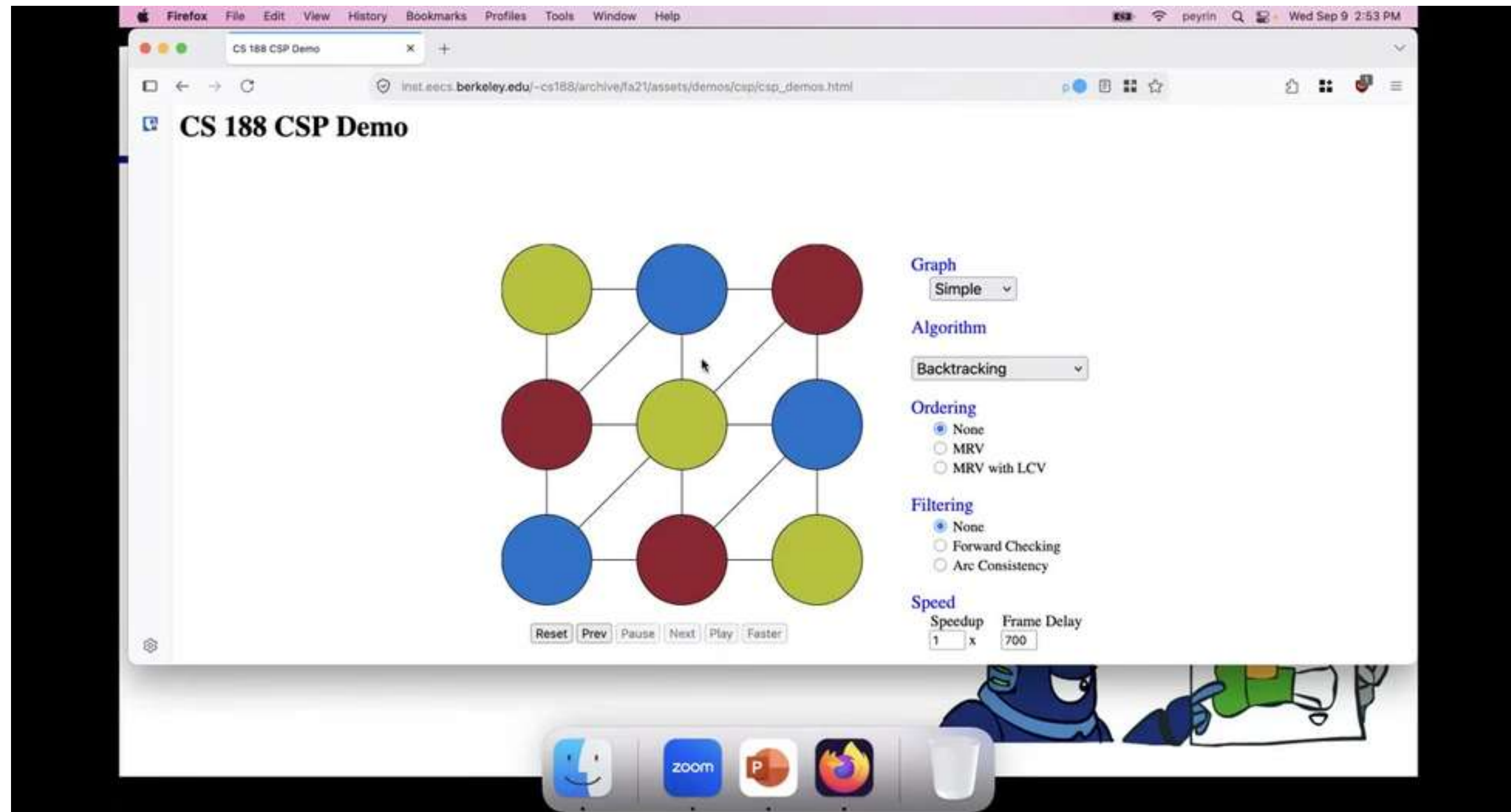
Slide 53 · 43:06

Okay, what's next red green? Why is this green? Because we checked blue here, that doesn't work. The entire sub tree with blue here is doomed to failure. Because of this constraint here, the entire sub tree with red here is also doomed to failure. So I'm only going to check the sub tree with green. Okay, now, keep in mind, this is still backtracking search, so we're not going to get an answer on the first try, and, in fact, right. Once I get here, there's actually no valid successor, but there's no color you can put here. So you're still going to have to go back and pick other things on the fringe. So this is not a perfect algorithm, right?



Slide 54 · 43:36

You're still gonna have to go back and try things on the fringe, just like Depth First Search, but with some struggling we get to the problem. So this is better. We solved the problem and we did it by adding those two improvements. Pick an ordering like Go in this order and fail as soon as you see a constraint that's been violated.



Slide 55 · 44:12

Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
 - Variable assignments are commutative, so fix ordering
 - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
 - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
 - I.e. consider only values which do not conflict with previous assignments
 - Might have to do some computation to check the constraints
 - "Incremental goal test"
- Depth-first search with these two improvements is called *backtracking search* (not the best name)



That's called backtracking search, and it's the core of all the algorithms we're about to see. We'll just make improvements to this for the rest of today. Yeah. Mm-hmm, yeah, that was a great question. What is on the fringe? At any given time? The answer is exactly the same as naive search, the only difference is that some sub trees don't get explored. If I had a pencil, I would show it to you and draw it.

Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
 - Variable assignments are commutative, so fix ordering
 - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
 - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
 - I.e. consider only values which do not conflict with previous assignments
 - Might have to do some computation to check the constraints
 - "Incremental goal test"
- Depth-first search with these two improvements is called *backtracking search* (not the best name)



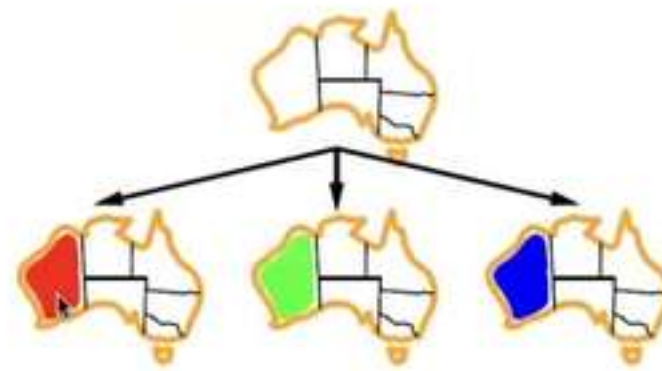
But basically, the idea is still that you're gonna have a big search tree. We'll try, okay, so you're still gonna have a big search tree, the only. And you're still gonna be expanding stuff. Just like Depth First Search does, and as you do, you're still gonna have a big fringe, right? Like, I don't know what's underneath here or here Changed at all, which is why I don't really like this name. The only difference is that if you see that one of these states, like this one, has a constraint violation, because it has two blues next to each other. You know, this entire search tree is doomed to failure, so you don't have to check it. But the fringe idea is still there. Does that help a little bit with my crappy drawing? Very good. The drawings will become less crappy. I hope. Okay. Cool. So that's backtracking search. So we're doing a little bit better. If you think back to the N Queens problem, we can now solve, N equals 25. We can do a 25 by 25 grid just using this naive approach. But I'm gonna do even better by introducing some more improvements unless you have questions.

Backtracking Example



Okay, let me show you an example. Oh, I do have this slide. Okay, hopefully this slide helps you a little bit by showing you that. It's still DFS with a fringe, even though the demo, I think, doesn't make it super clear. So here's your starting state. None of the variables have values. So what do you do? You pick it up and you ask, are you the goal? No, you're not the goal.

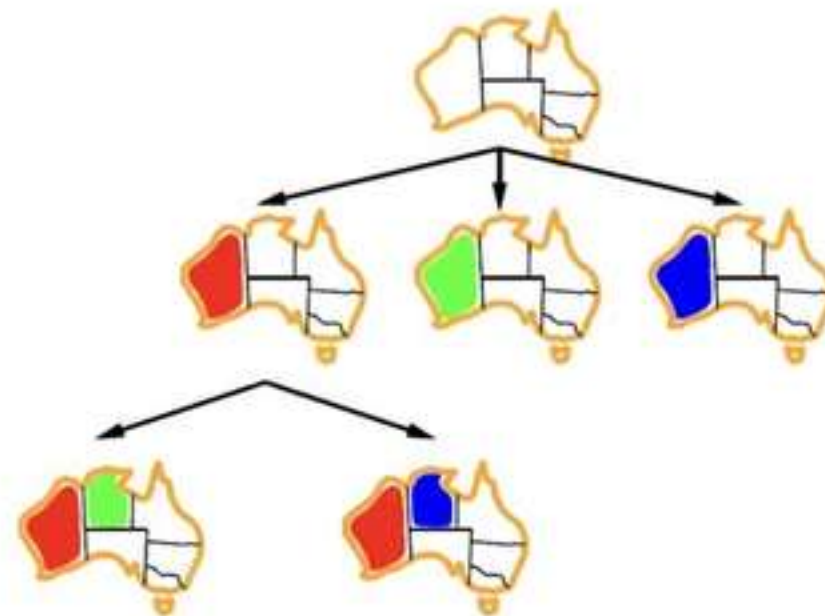
Backtracking Example



So we're gonna expand you, and the way we're gonna expand you is we're gonna look at your three successor states and your successor states. Are this one's red, this one's green or this one's blue? Okay, good, what do I do now? I pick one of these up, they're all the same depth, so I can pick any of them. Depth first Search says, pick the deepest one, but here they're all depth one. So I'll pick one of you up and I'll say, are you the goal?

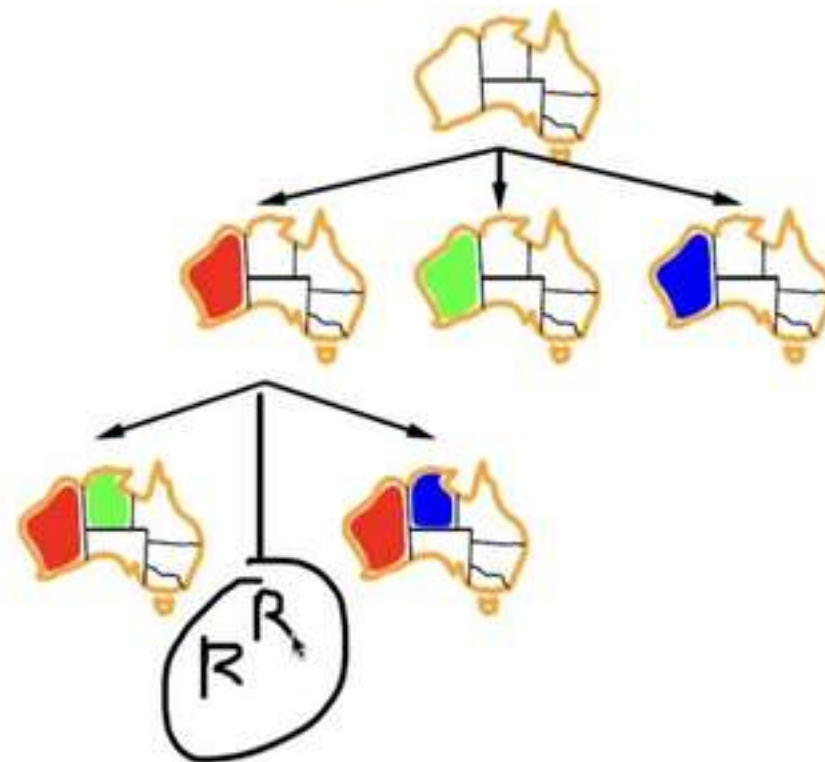
No, and this is where backtracking search differs from naive search. Because if this was naive search, there would be three children and the third child here would be the one where there's like a red and then there's another red.

Backtracking Example

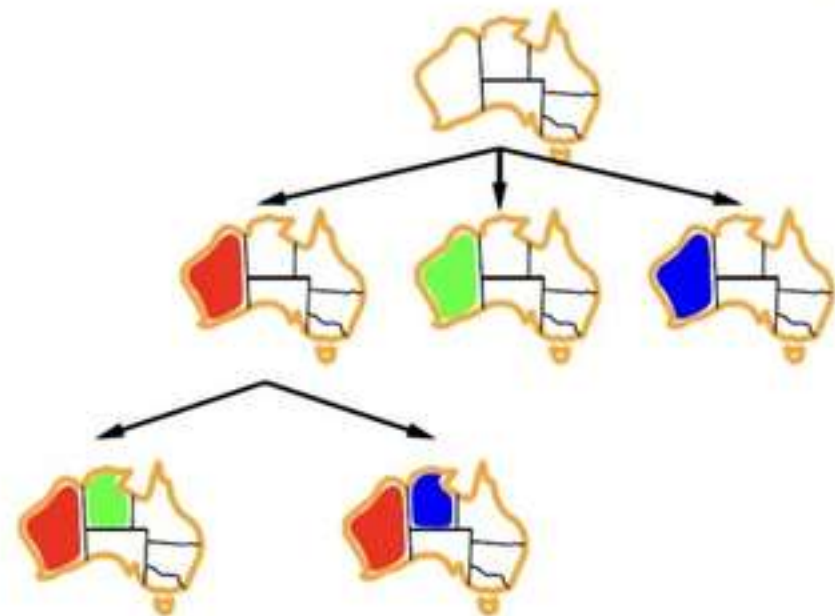


But backtracking, Search realizes, oh, wait a minute, this state is already doomed, so even though there is a gigantic sub tree here, we do not have to consider it at all, and we only have to, in queue, two children, but there's still a fringe.

Backtracking Example



Backtracking Example



So yes, I knocked out a big sub tree, but there's still a fringe. I still need to know what's underneath here or here, or here or here. [afraid] So the fringe is still there. Okay, need to pick one of these up depth first search, so I need to pick this one or this one, right? These are the deepest ones on the fringe so far, so I pick one of them up, and then I would expand it and so on, and so forth.

Backtracking Example

The diagram illustrates a backtracking search for a 3-coloring of a map of Australia. The search tree shows states where regions are colored red, green, or blue. A blue robot character is shown rejecting a state where a region is colored orange (indicated by a red X), demonstrating the backtracking process.

You can pick. Which variable to assign next? And you can also pick which value to assign next, a little bit of preview for you, those are things we can choose later to optimize our algorithm even further. But for now don't stare at it.

Backtracking Search

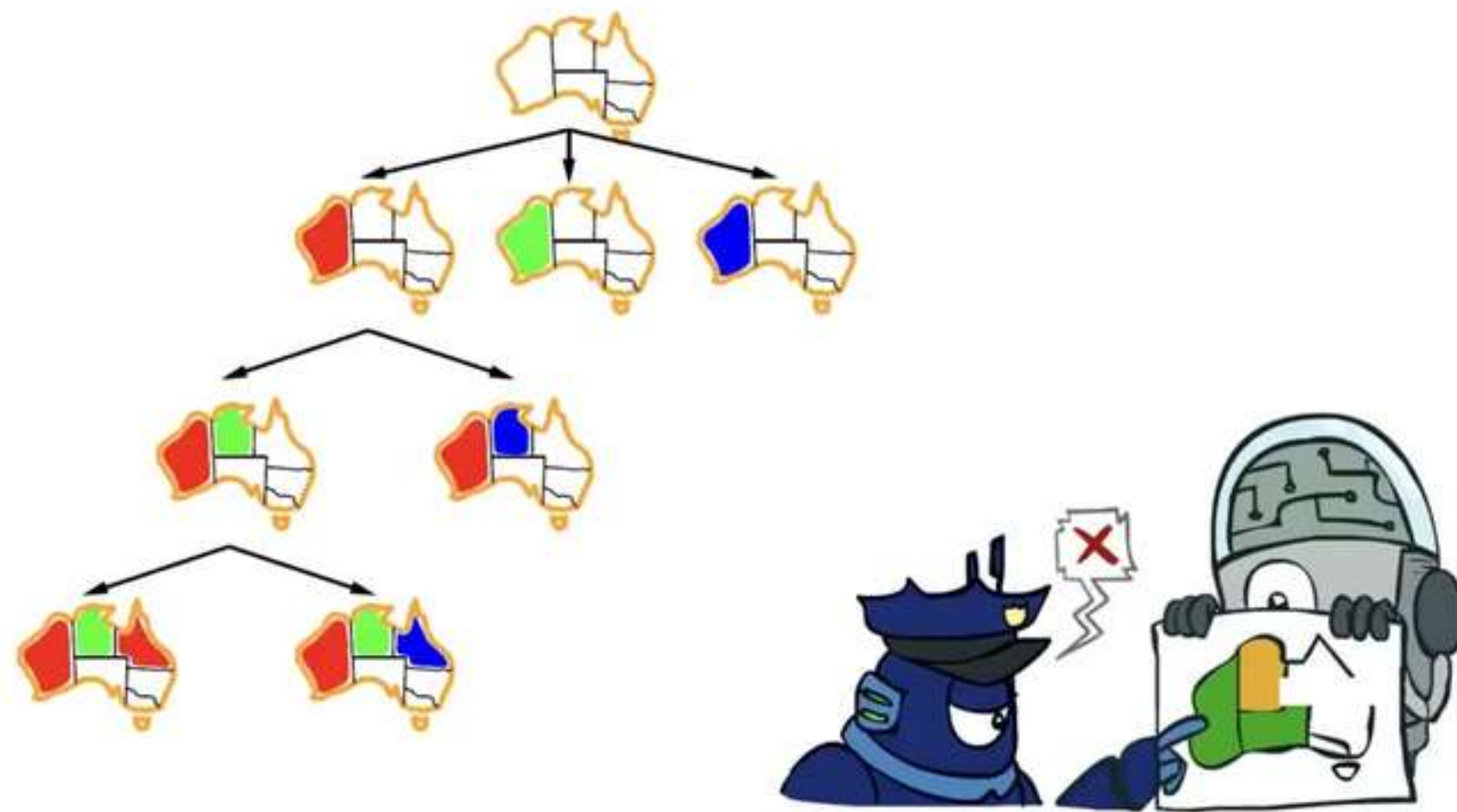
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given CONSTRAINTS[csp] then
      add { var = value } to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove { var = value } from assignment
  return failure
```

- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?

[Demo: coloring -- backtracking]

Backtracking Example



speaker 1

Yeah question. Previous slide. Yeah, that's a good question, like, who in the code recognizes that one of these sub trees is doomed and that you can totally skip it. I think it kind of depends on your implementation. Here what they've done is they've actually implemented it recursively. So what they did is just if there's a constraint violation. Just don't do the recursive call. But you can do it some other way. You can have a fringe and say I'll delete things off the fringe if they violate constraints. Your pick.

Improving Backtracking

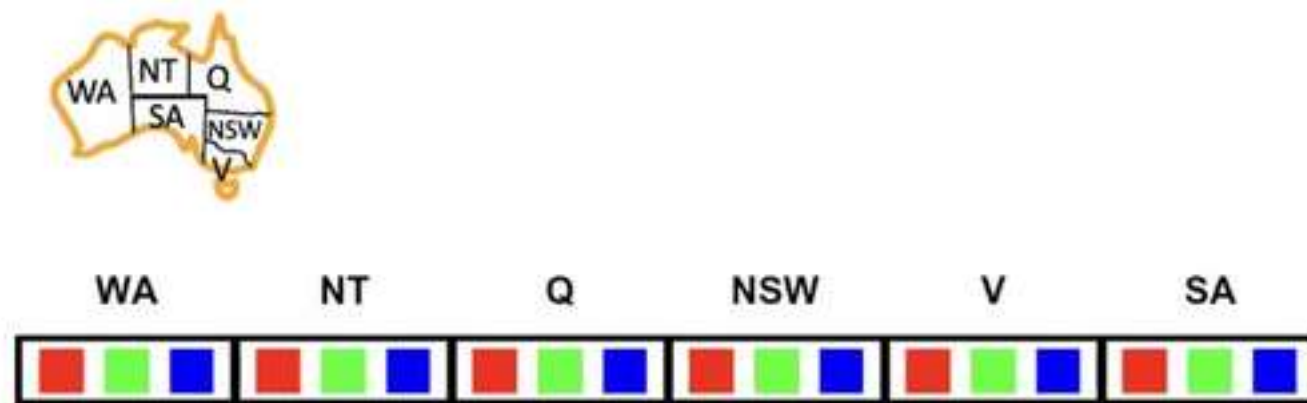
- General-purpose ideas give huge gains in speed
- Ordering:
 - Which variable should be assigned next?
 - In what order should its values be tried?
- Filtering: Can we detect inevitable failure early?
- Structure: Can we exploit the problem structure?



Or everyone implemented. So we're basically good. We have an algorithm that works. It solved the little problem. But what we'll do now and for the rest of today and also next time, is we will come up with three separate improvements to make backtracking even better. So one of them is ordering. What order do you assign the variables and the values? Like, do you pick red first or do you pick green first? And does it make a difference? Or blue? And then there's filtering which is the one we'll see today. Is there any way to detect failure a little bit earlier? And then the last one that we'll see next time is structure. Can you somehow exploit the problem structure? And do something clever. All of these are based on backtracking search. The only difference is we're doing some improvement on top of it to hopefully make the algorithms a little bit faster. But you already have the core algorithm. Now we will make it faster in three different ways. Let's start with filtering.

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

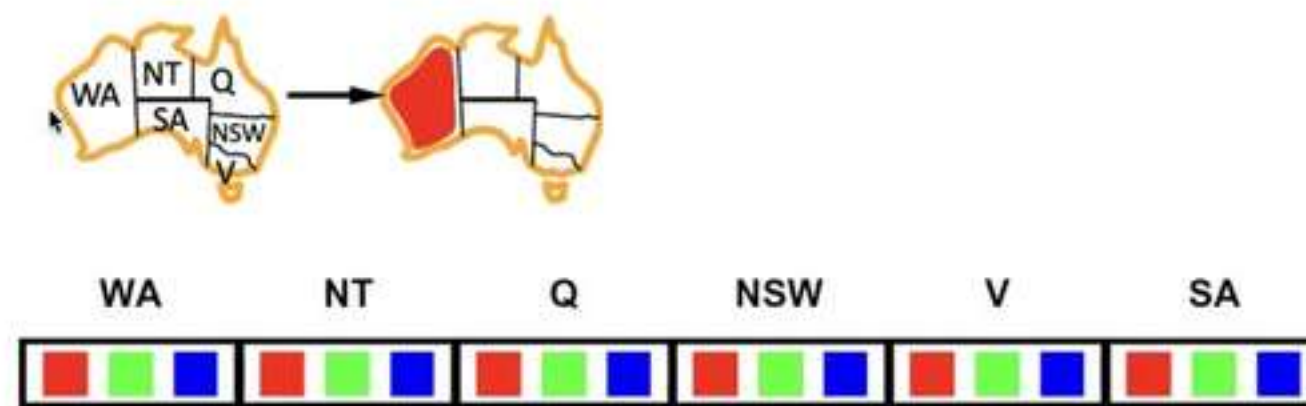


[Demo: coloring -- forward checking]

Okay. So there's different types of filtering, but all of them have to do with keeping track of what values are still valid at a variable. So already if you look at this picture, you notice that things have changed a little bit. So now, instead of each of these boxes either being unassigned or assigned, right? This is either blank or red or green or blue. What we're now going to try and do is try and detect. Failure early, is there some way we can know that something is doomed without having to explore too much of the tree. So we are. Can we somehow look a little bit into the future and say, you know what this assignment is already do? There are no valid successors that will give me an answer, so let me just stop right now, instead of trying a bunch of different things. And the way you're going to deduce that things are doomed to failure. A little bit earlier is you're going to keep track of what domains are still valid for each variable. So now, instead of each of these boxes being assigned or unassigned, like a yes, no. Now this little cloud, there's a little cloud that says, Well, right now, before anything has been assigned. Western Australia could be red, or It can be green, or it can be blue, and NT could be red or green or blue. And in fact, right now, since nothing is assigned, each of these can be red or green or blue. But what you'll do as you start assigning is you actually check the constraints as you go. And you'll cross out values that do not work anymore based on the current assignment, and as you cross out values. If you ever notice that one of these domains has all of one of these variables, has all of its values crossed out. You know that there's no possible successor, and you can quit early and go explore some other part of the tree. So this is still backtracking, search, still a tree, still DFS, but you're hopefully going to detect failure early, so you know, oh oh, this is doomed, let me go explore some other part of the tree. A little bit better. Okay, so let's do forward checking together. So here we go.

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



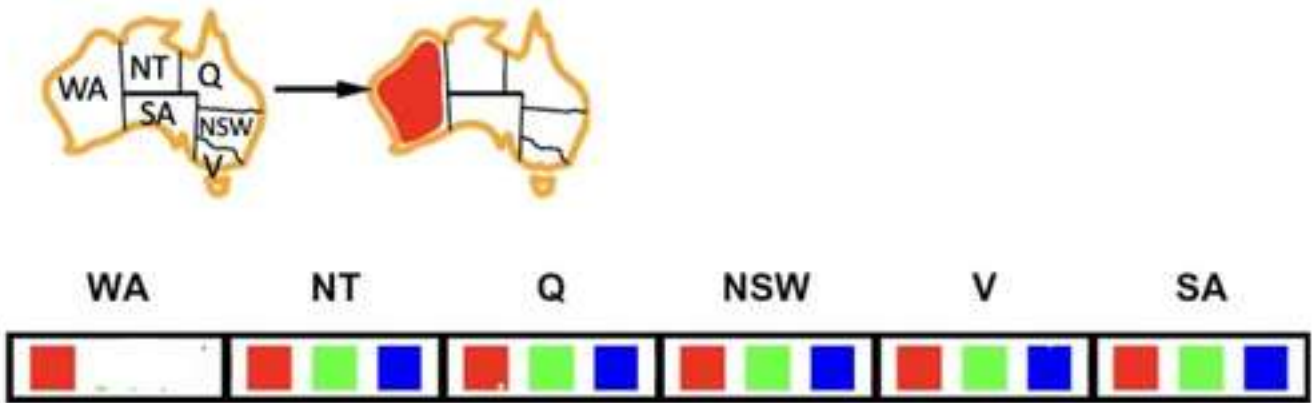
[Demo: coloring -- forward checking]

It's still backtracking search. I pick up this starting state. Are you the goal? No. So I will assign one of your successors. So you see here, I've assigned one of the successors. I've assigned. I think this is Western Australia to red. So now I can actually go here and see, once I've assigned Western Australia red, based on the constraints, that I can see my list of rules. What's changed? What values are no longer valid? Let's do some crossing out. Okay. I have a hack. Hopefully I can find it. Color is white. I have a magic eraser. Here we go. Okay. So Western Australia. Well, it's red. So green. Is no longer valid. Okay. It's gone. And blue is no longer valid. Okay. Also gone. So I'm crossing things out that are not consistent, ah, with the current assignment. Okay good. That one wasn't too bad, because I just assigned w a, so red is the only thing left. Now let's do some more interesting ones. Take a look at NT. So NT is this one right here. Originally, I said, red and green and blue are all valid colors. What about after this assignment? Are red, green and blue all still valid colors? No. Which one is no longer valid? Red right, because I can no longer color NT red, because if I color NT red, then well. It's gonna conflict. There's gonna be a constraint violation. Okay. The downside of making your magic pen or your magic eraser white, is that you can't find it anymore. Okay. Did the slides just let me advance? What do I do? Okay what do I where are you pen? Sorry. Okay this was this was very much me being. I got too bold. I was like I wanna use white to be clever. I lost my pen. Okay. I don't know what to do anymore.

What if I just. Nope that doesn't help. I'm sorry, okay, but the point being what I wanted to do? Ah, okay, what I wanted to do was erase White from NT, which I'm utterly failing to do. Okay my mouse is back thank you. Okay, good, the slide does it for me, so I don't have to be clever with it.

Filtering: Forward Checking

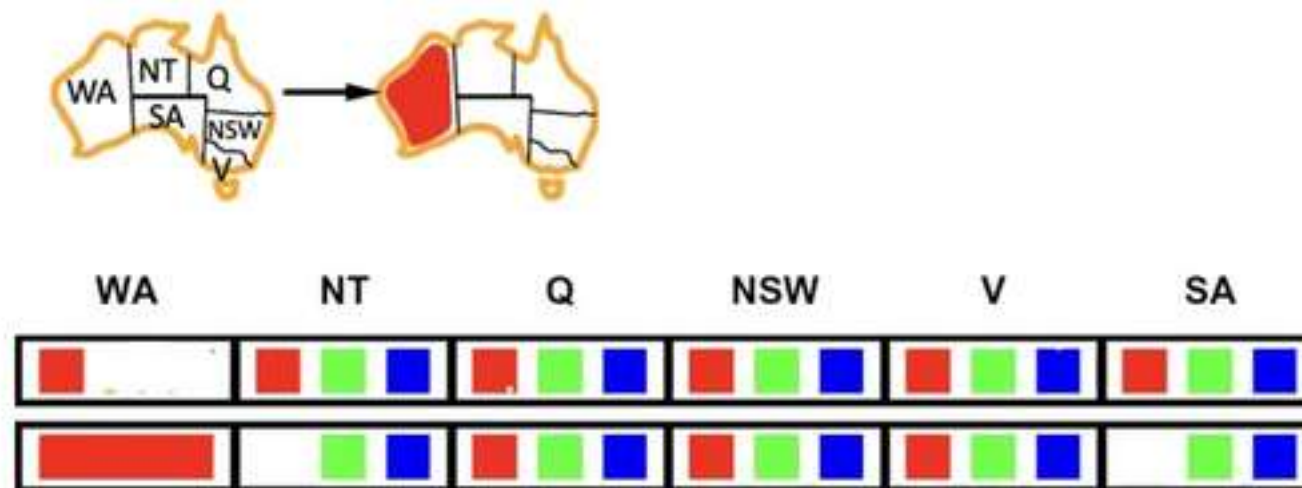
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



[Demo: coloring -- forward checking]

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

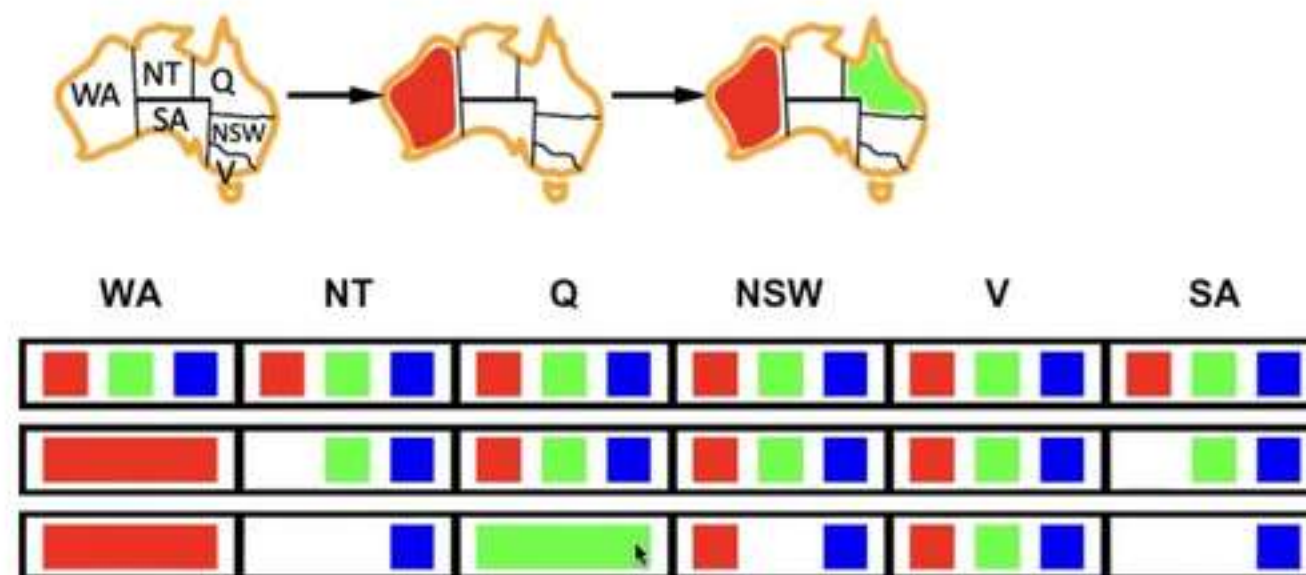


[Demo: coloring -- forward checking]

Okay, so ah, go away, okay, so what you do is, ah, Western Australia is definitely red, but notice that NT loses its red because red is no longer a valid assignment here. And notice that the s a South Australia also loses its red, because red is no longer. So now as I'm assigning for this state, I'm also keeping track of a little cloud that says, You know what? These are the possible valid successor states. These are the possible valid assignments to the values. Why is this useful? Let's assign some more to see why it's useful. Let's say so I picked this up. Are you the goal? You're not the goal. So let's go ahead and assign Queensland to green. So what happens now?

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

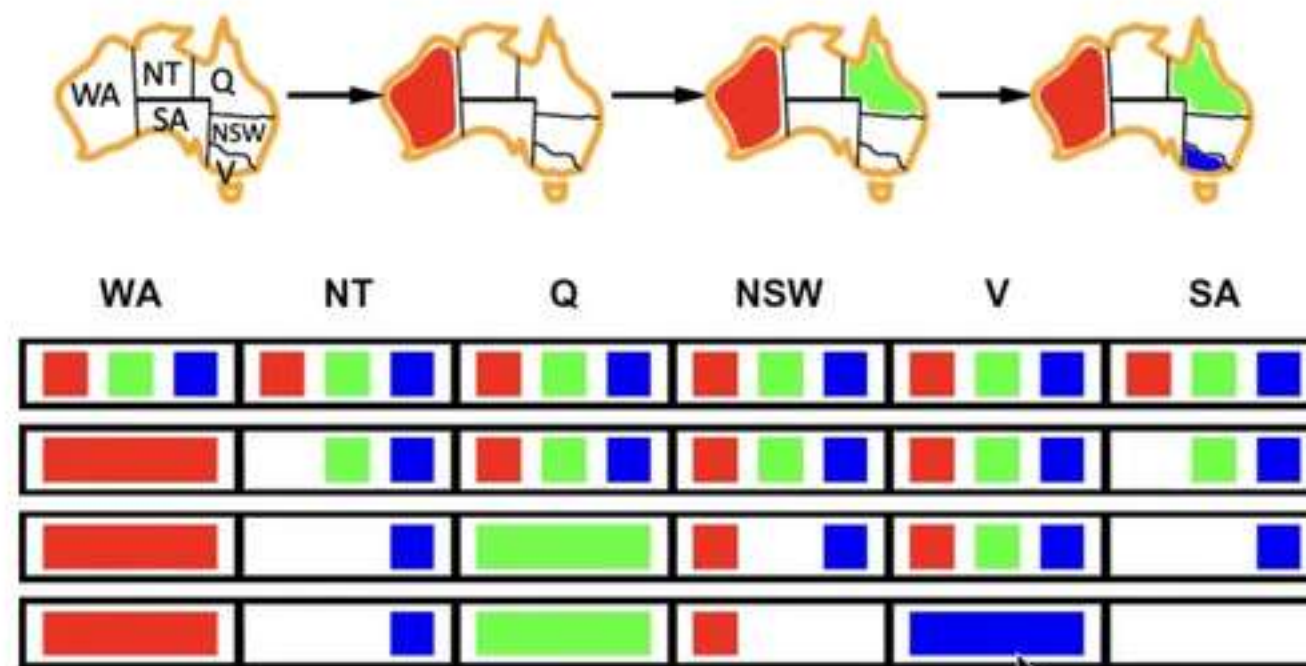


[Demo: coloring -- forward checking]

Well, I assign Queensland green very good. But now I can cross things off. So for example, NT loses its green because coloring NT green would not be consistent with my new assignment, which was Queensland equals Green. And also, SA can no longer be green, because if SA is green. What conflict? And New South Wales? You can also not be green anymore because that conflicts, so as I'm going, I'm keeping track of what's still possible. Let's do one more assignment, so let's take, let's pick this up. Are you the goal? You're not the goal?

Filtering: Forward Checking

- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



[Demo: coloring -- forward checking]

So I'm gonna color Victoria equals blue, but as soon as I color, Victoria equals blue, look, what happens? I have to cross off blue from s a, and I also have to cross off blue from NSW, the two neighbors. But now I can stop right here. If this was naive backtracking search, what would naive backtracking search do? Naive backtracking search would continue? It would try assigning all the colors before realizing that something has gone wrong. What about forward checking? Forward checking is smarter. Why do you know in Immediately that, like this state is doomed and none of its successors will work. You know, this is doomed because South Australia has no valid values anymore. So by crossing things off as you go, you can fail early. You can say, you know what, even though this kinda looks promising, like, maybe there's a solution. I know, for a fact, there isn't one because I did forward checking. I filtered as I went and I detected failure earlier. If I did not do this backtracking search, would have gone all the way to the end. Assign all the colors to realize Oops nothing works. But here I can detect failure early and say, nope, this state is already in trouble. You should go back and pick something else off the fringe. And something else might be going back here and saying, Well, this could have been blue. This could have been blue or red. So you have to go back and pick something else. That's forward checking. As you go cross off values that violate constraints. Pretty cool. Now you can detect failure early.

Filtering: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



WA	NT	Q	NSW	V	SA
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

Okay. Great. So this is pretty good. Turns out you can do even better. So take a look at this intermediate state right here. This is what we saw earlier right. We assigned w a, equals red, and then we assigned Queensland equals green. Now stop right here. Does anybody have a bad feeling about the state or like, something's gonna go wrong? I have a bad feeling about the state. Why do I have a bad feeling? Look at like NT and s a. N t has to be blue. In this configuration and Sa also has to be blue. I have a bad feeling about this. You know who doesn't have a bad feeling about this? Forward checking Forward checking does not care, Forward checking does not look this far ahead. It only thinks about information from an assigned variable to an unassigned variable. So, for example, forward checking would say, oh, you assign Q equals green. Go ahead and knock green off NT SA and NS W. It only did that because you assign Q equals green. It does not look at the unassigned variables. So it just says, you assign Q, go cross some things off. You assign w a go cross things off that have to that are inconsistent with w a equals red. But it does not. Care that there are looming constraint failures that are further ahead in the future. So forward checking is pretty smart, helps us detect failure early, but it's not totally smart. And in this particular case, it cannot find a failure between two unassigned variables. It can only find constraint failures between an assigned variable, like Q equals green and its neighbors or its other variables. Doesn't have to be neighbors, just neighbors here because it's a map.

Filtering: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



WA	NT	Q	NSW	V	SA
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

- NT and SA cannot both be blue!
- Why didn't we detect this yet?

So in this case, even though we, as humans, looked a little bit into the future and said, Nah, this is doomed, forward checking does not look that far into the future. It only checks constraint violations with assigned variables. WA is red. How does that affect my domains? Q is green. How does that affect my domains? I don't care that these two are blue or that blue is the only thing left. I haven't assigned them yet. They're not colored in yet. So I don't care. I only care about constraint violations that occur with assigned variables. So forward checking is nice.

It's simple but it's limiting. It doesn't look that far ahead into the future. So what if you want to look further ahead into the future? We just have to make it more powerful. That's what we'll do next. But that's it for forward checking. Thoughts on that before I show you arc consistency, which is the even smarter version of forward checking. I have a demo. Let me show you a demo. Okay. Let's go back here. Let's go back to our. Our favorite graph. I'm gonna reset you.

Filtering: Constraint Propagation

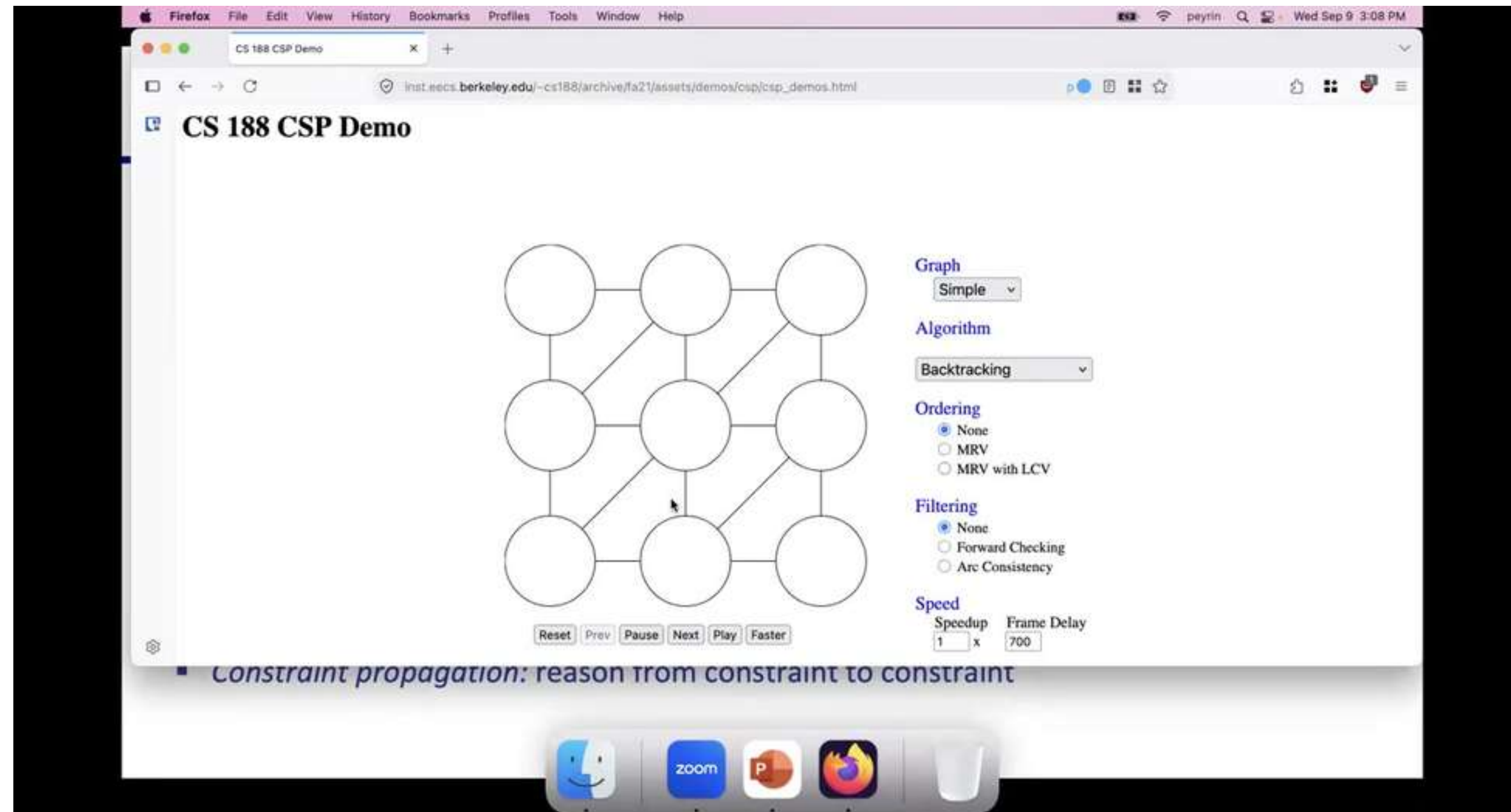
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



WA	NT	Q	NSW	V	SA
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

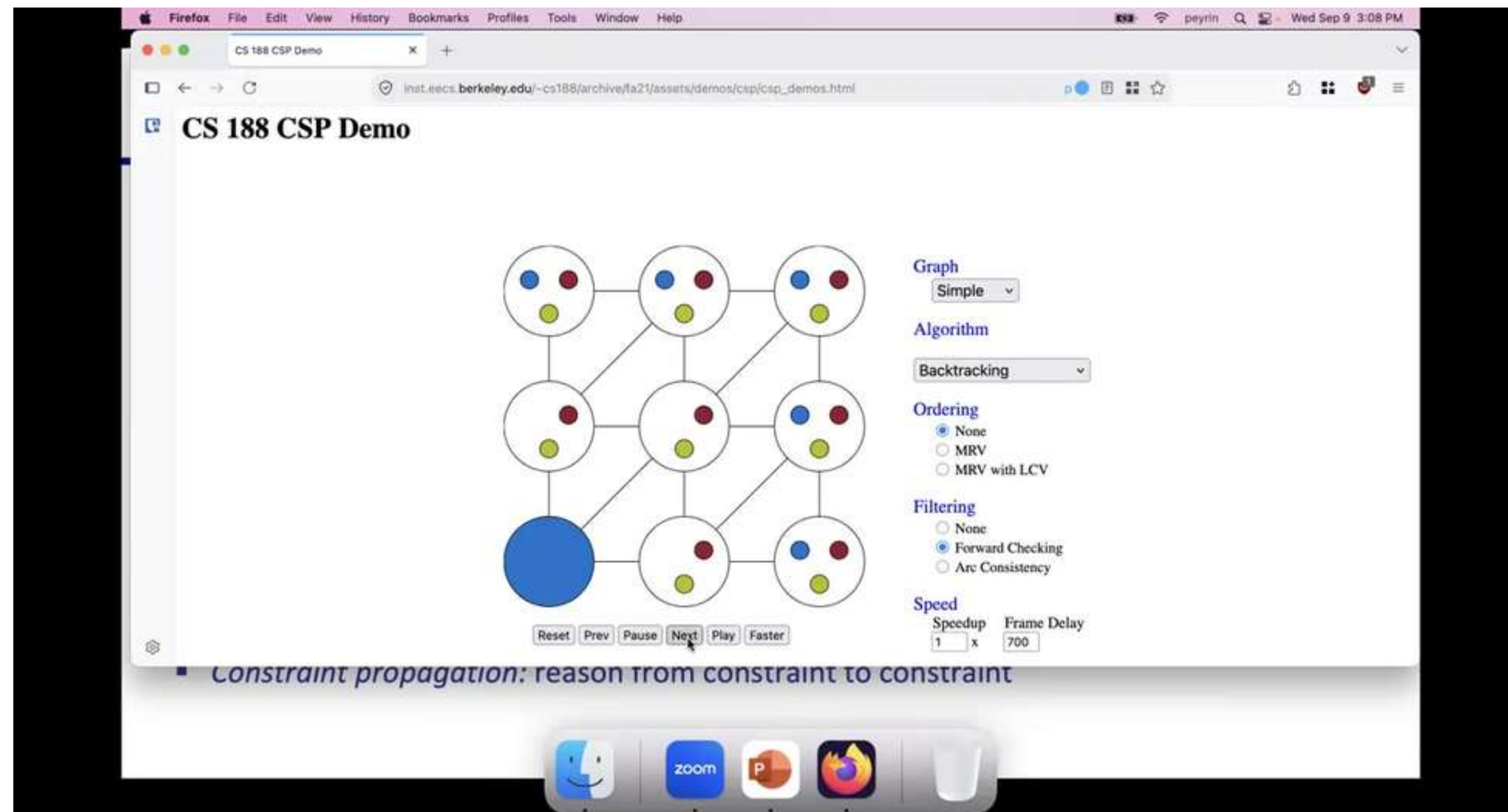
- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- Constraint propagation*: reason from constraint to constraint

But now, instead of just doing naive backtracking search where everything is unassigned or assigned, I'm gonna do forward checking. There's a little option here for forward checking. Click Okay. Now we have Forward checking. So now you see instead of values being unassigned or assigned, everyone now has a little cloud that says I could be blue or red or green. So here we go. We're still gonna do DFS. We're still gonna assign. So I pick you up and I assign.

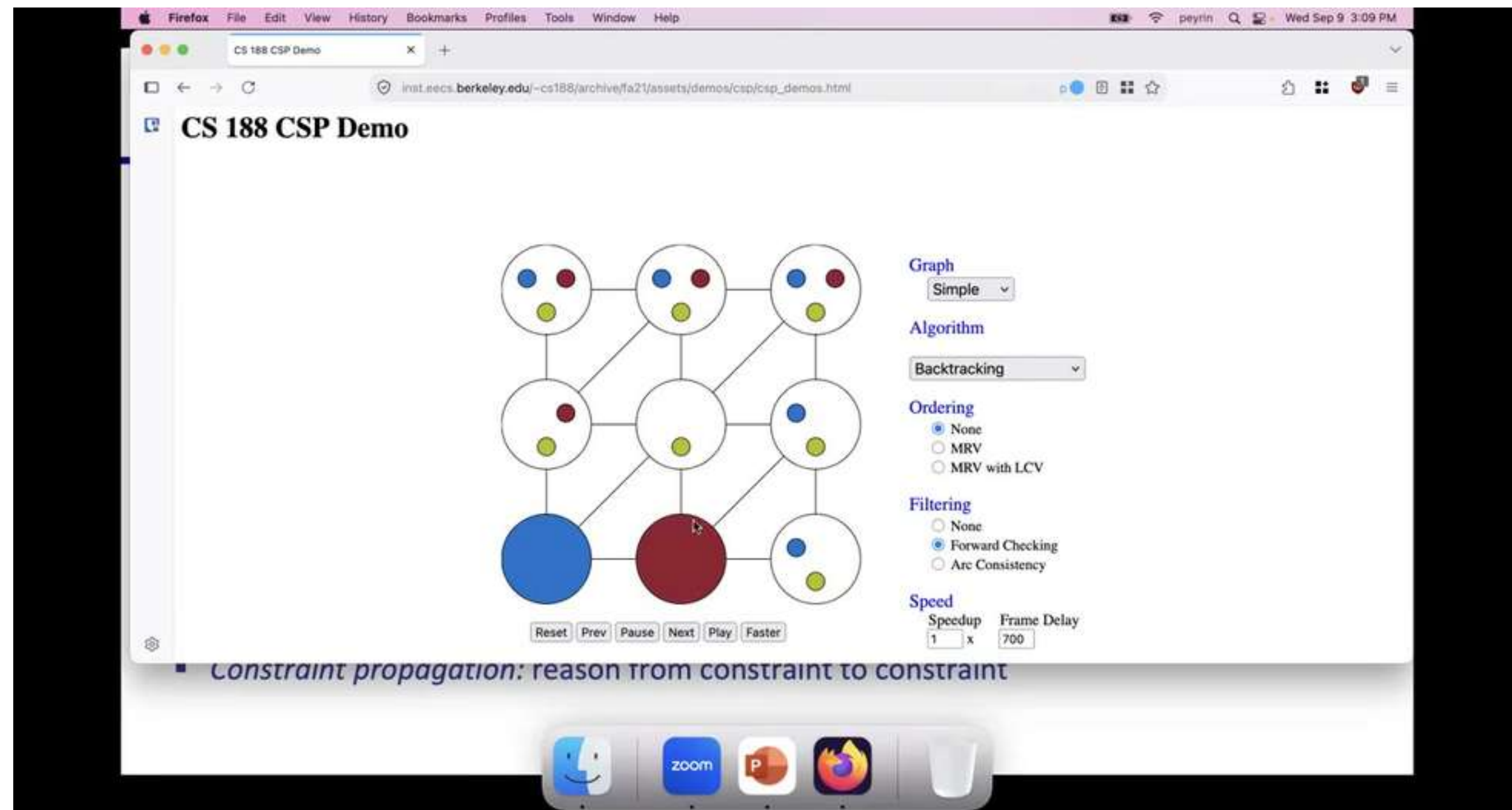


Slide 76 · 58:42

oh. Reset. Okay. So I wanna assign you to blue. But in addition to assigning this node to blue, look what happened. This one, this one and this one, they lost the blue because assigning any of these blue would cause a cause, a constraint. Violation with my newly assigned variable. So I cross them off. They cannot be blue anymore. Did I solve the problem? Nope. I need to keep going. So why can't I keep going? The button just like grayed out for some reason. Okay. Let's see backtracking forward checking good. Okay so we assigned this blue.



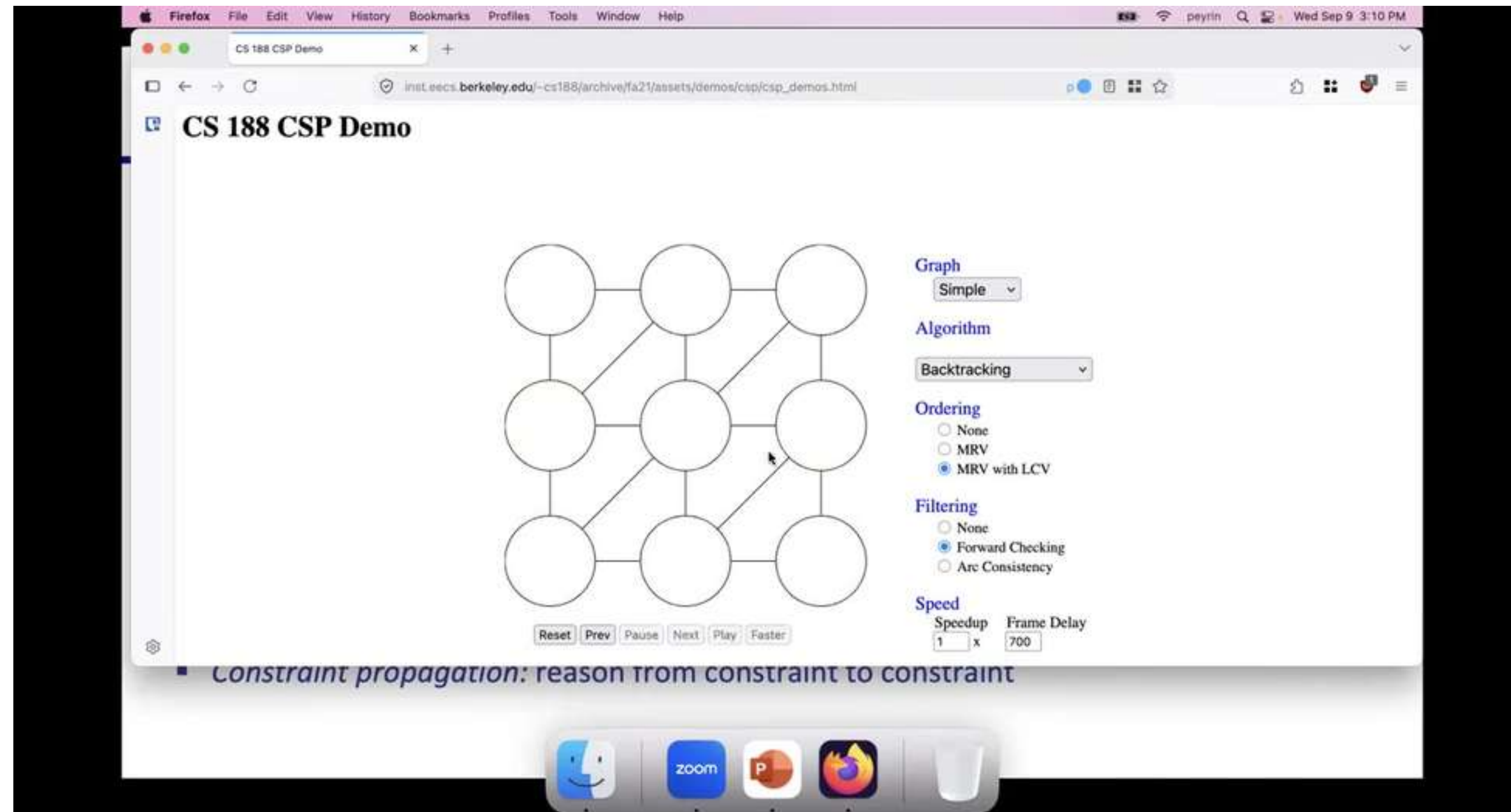
Slide 77 · 59:06



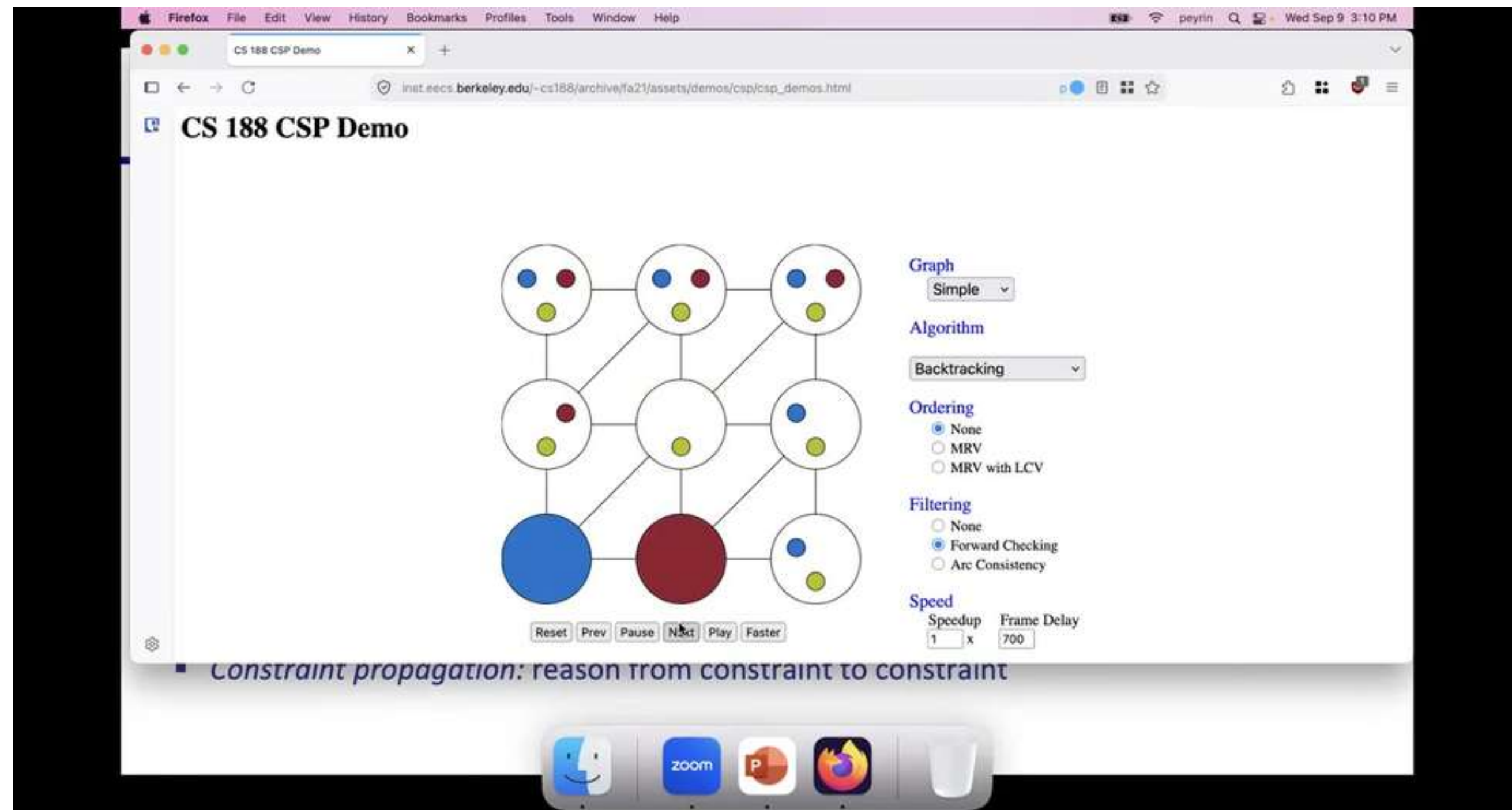
Slide 78 · 59:42

They lost their blueness. So I click again. Okay I assigned this red. Now what happens? This one, this one, this one, they all lose their redness, right? The red is gone. They're no longer, red is no longer a valid value here because they would cause a constraint violation with the red that I have assigned here. Next, assign this one. What about, will I assign it? You're blue. Actually, no, I guess I'll assign. Why did I assign that? Whatever. I'll assign this one green. That's fine. And when I assign this one green, this one and this one, they lose green because, and I guess this one and this one as well. All of its neighbors lose green, because green is not a consistent assignment with the variable that I have just assigned. And so we keep going. Okay. That's red. That's blue. It's green. Yeah. Why did it go in this order? That's bizarre. I thought I was supposed to go bottom up. Try again. There we go. No. I don't know what's going on.

Whatever. Okay. And so you do some backtracking but eventually you get. That's that's a forward checking you cross out values as you go, this needs to be none. Let me try again. Okay that's better. I don't know why I did, I don't know why I did that. Okay whatever sorry for bug. Okay, I'm gonna pretend I'm assigning the variables from bottom up, like there's a fixed order.



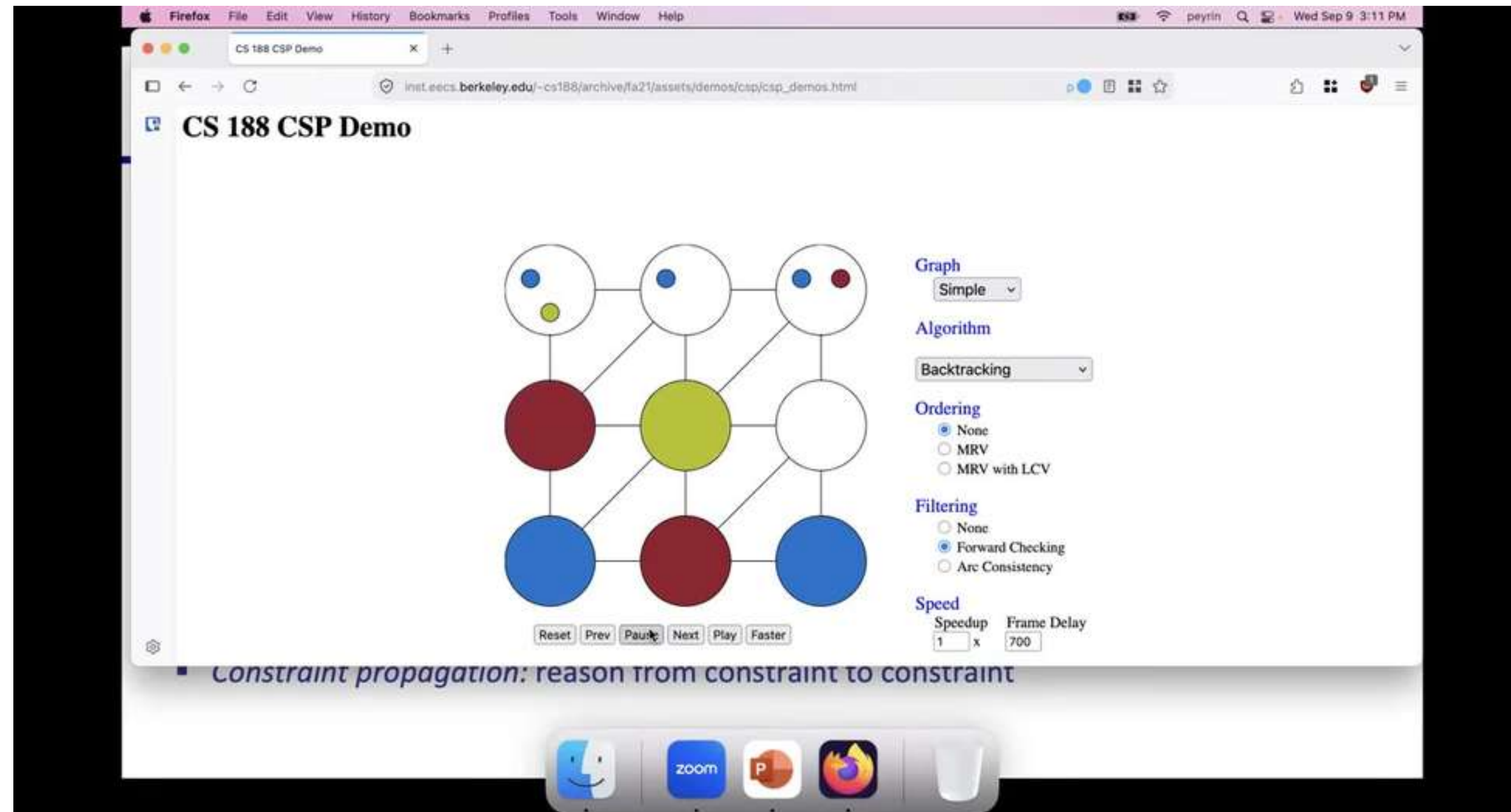
Slide 79 · 1:00:36



Slide 80 · 1:00:54

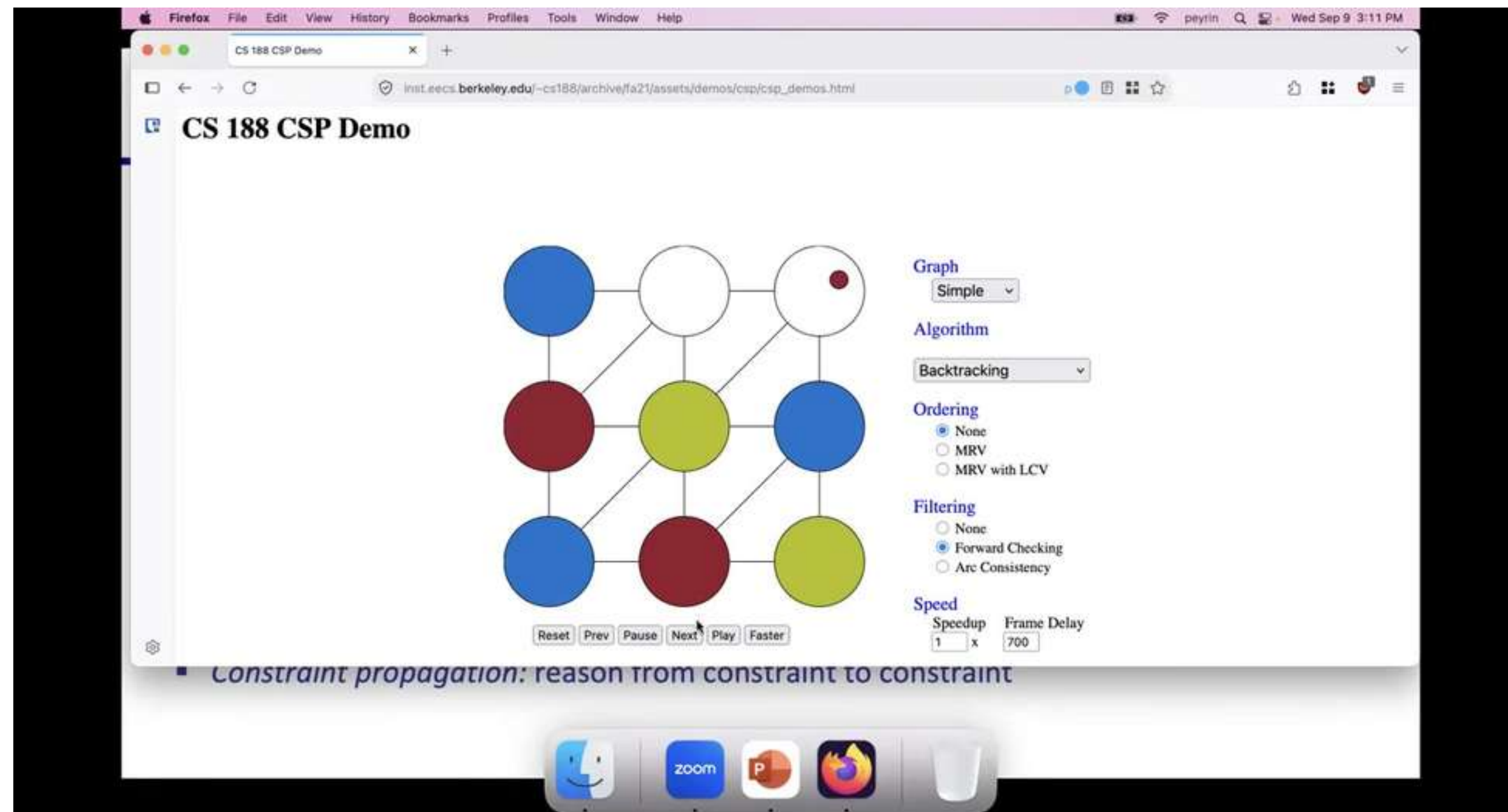
So send you Blue cross off some blues, send you red, I cross off some reds, send you Blue cross off some blues. Stop right here. Does anybody have a bad feeling about this? I have a bad feeling about this. [other] You know, who doesn't have a bad feeling about this? Forward Checking doesn't have a bad feeling, because Forward Checking only thinks about constraint violations with already assigned variables. You and I, as humans, we can see oh, problem here. You have to be green, you have to be green, you cannot both be green. But Forward checking doesn't think about that. Forward checking only thinks about relations to the immediately assigned variables like these three down here.

So if you're doing forward checking like this, you actually have to keep going until here and then as soon as you assign this one green, you're like uh-oh, that one can't be green either. I need to backtrack. I need to go back on my fringe, pick something else. And so some backtracking is required. But eventually I work through the graph. You see there's another backtrack here.



Slide 81 · 1:01:30

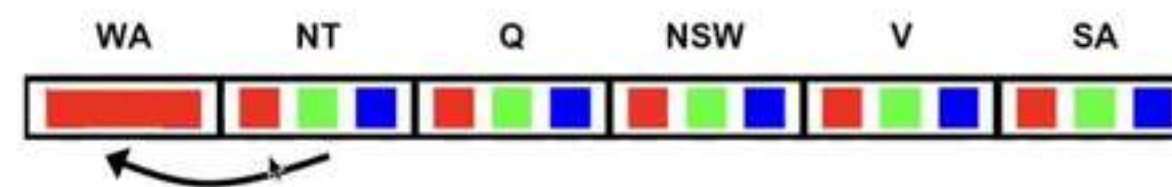
It's like Oops, that doesn't work because this would empty out this domain. So I go back, pick that instead. Okay, now I'm good. So it does solve the problem. But takes a little bit of backtracking and it doesn't detect failure. Particular? This is the critical moment where we could have done better, but we didn't because forward checking wasn't strong enough. Any thoughts before? I make it stronger using something called Arc consistency. Time to make it stronger. It's like forward checking but even stronger. So here we go.



Slide 82 · 1:01:48

Consistency of A Single Arc

- An arc $X \rightarrow Y$ is **consistent** iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

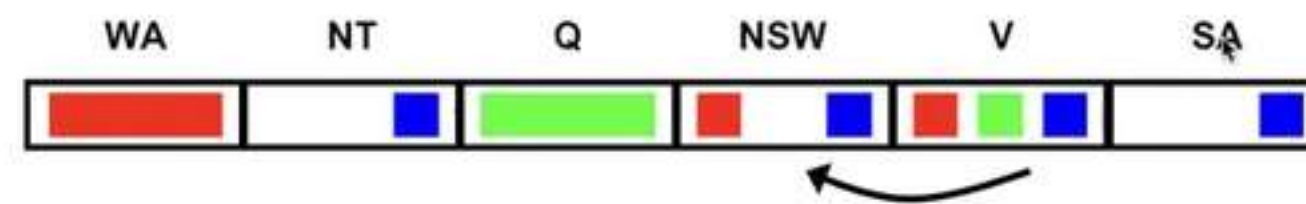


- Tail = NT, head = WA
 - If NT = blue: we could assign WA = red
 - If NT = green: we could assign WA = red
 - If NT = red: there is no remaining assignment to WA that we can use
 - Deleting NT = red from the tail makes this arc consistent
- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Okay, so before I can tell you about arc consistency in total, let me first tell you about a single arc. What is an arc? An arc is an arrow pointing from one domain to another. It is directed. See how there's an arrow. So there's actually one arc from NT to w a, and there's another arc from w a to NT. So arcs are actually bi-directional. This is something new that you haven't seen. And each arc can be consistent or inconsistent. So it has a property called consistency, totally different from anything you've seen before this. Same name, but it's either consistent or it's not consistent. So it's like a yes no answer. And the way that you check whether or not an arc is consistent is you follow this rule. You say for every possible value in the tail, which is the part without the pointy arrow, for each of these values red and yellow, sorry, green and blue, is there some at least one valid assignment in the head, which is the other side of the arrow, that can be assigned without a constraint violation. That's the rule. It'll probably make more sense if I show you some examples. So let's do some consistency checking. Remember each arc is either consistent or not consistent. So is a yes or no answer. So let's check. This arc is consistent. If I assign blue to NT, is there something in w a the head that is consistent? Yeah. I send this blue. This can be red. If I assign NT equals green, is there something consistent in the head? Yeah. W a equals red. If I assign NT equals red, is there something in w a that works? No. W a is red. And so this arc is inconsistent because I cannot assign NT equals red. This value in the tail has no corresponding. Valid value in the head. So this arc is inconsistent. The answer is no, it's not consistent. How do you make it consistent? And the answer for how you make it consistent is you can delete values from the tail. So in this case, because blue worked from the tail, and green also worked from the tail, but red didn't work. Okay, no more no more being bold with the colors. What I can do is I can delete red from the tail so that it only has two values left, and now the arc is consistent. So it used to not be consistent, but by deleting values from the tail, I can make the arc consistent. Turns out forward checking is basically this forward checking. Is checking arcs, and in particular, you check all the arcs, pointing at the thing you just assigned. So forward checking is like a special case of this, but we're gonna make it even more general because we're gonna check arcs not just between assigned variables, but also between unassigned ones. And my mouse is gone again. Okay. No, ah, if I click this at my back, ah, mouse, mouse, mouse, okay, good. So delete things from the tail, that's how you get arcs to be consistent. Okay.

Arc Consistency of an Entire CSP (1/6)

- A simple form of propagation makes sure **all** arcs are consistent:

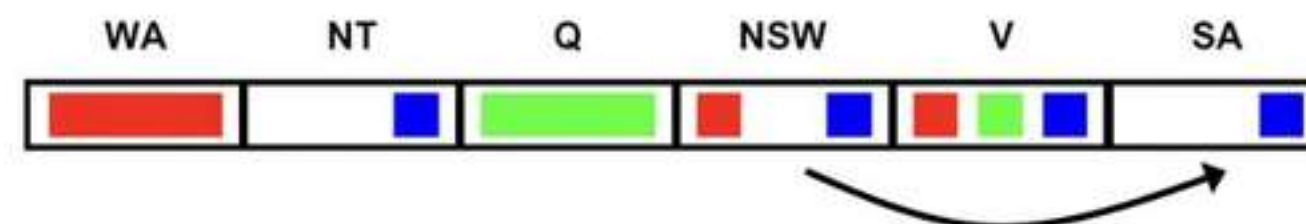


- Arc V to NSW is consistent: for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

So let's do some arc consistency together. Here we go. So we're gonna check arcs. In particular, we're gonna think about whether each arc is consistent or inconsistent, and if an arc is inconsistent, you will delete from the tail in order to make it consistent. And arc consistency is a property of a specific arc, and you can say that an entire CSP is arc consistent if all of its arcs are consistent. So each arc itself has a yes no answer, and then the whole CSP has a yes no answer, and the whole CSP is consistent if all of its arcs are consistent. Okay, that'll make more sense once I do a demo. So here we go. Let's check. So here is an arc, and this is the first arc you've seen, that's a little bit different because it points between two unassigned variables. So unlike this arc where w a was already assigned. Here NSW is actually not assigned yet. So unlike forward checking, we are now doing checking between two unassigned variables, but we can still do it, so let's do it. For every value in the tail, there must be something in the head that works. If I assign red, is there something in the head that works? Yeah blue works. You might say, oh, wait a minute, red doesn't work, that's fine. You just need at least one thing that works. So if I assign red in the tail, blue is a valid extension in the head. So red is still okay, because there's at least one thing, doesn't have to be everything, just at least one thing that works, and here it's blue. If I assign green for V, is there something in the head that works? Yeah either red or blue they both work. That's fine. If I say blue to V, is there something in the head that works? Yeah red works. They may say, oh, but blue doesn't work, that's fine, we don't care. We just want at least one thing that works. So this arc is consistent because everything in the tail has at least one valid extension in the head. I pick red there's something that works. I pick green there's something that works. I pick blue there's something that works. So this arc is consistent. The answer is yes. Let's check one more. What about this arc? Okay, well, this arc points from s a to NSW. Remember, check the tail and see if this a valid extension at the other end. Okay, in the tail, there's only one possible value here, it's blue. If I pick blue. SA Is there at least one valid assignment to NSW Yep it's red. At least one is all you need. So this arc is also consistent. For every value in the tail, there is at least one thing in the head that works. Let's keep going.

Arc Consistency of an Entire CSP (3/6)

- A simple form of propagation makes sure **all** arcs are consistent:

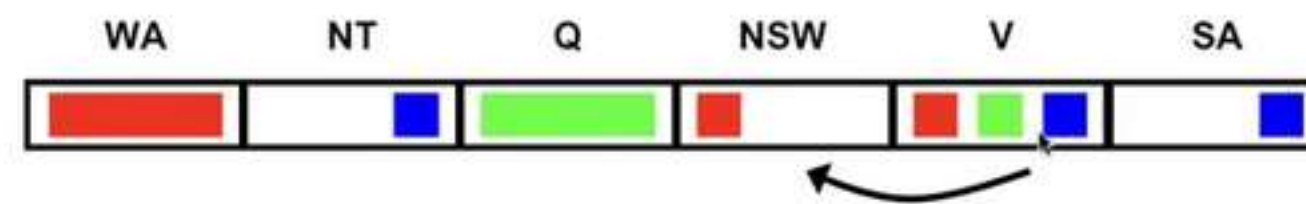


- Arc NSW to SA is not consistent: if we assign NSW = blue, there is no valid assignment left for SA
- To make this arc consistent, we delete NSW = blue from the tail

Okay let's check in the other direction. So this is another point, right? Arcs can be, arcs are directional, so there's an arc from SA To NSW There's also an arc from NSW To SA And there's just two different things. Let's check this one. Okay if I assign red to NSW In the tail is there something in the head that works? Yeah I can I can just blue all good. If I choose blue for NSW Is there something in the head that. No. If I pick blue for NSW, I'm toast, because the only thing that worked, or the only thing that's left in s a is blue, and I cannot assign them both blue. So this arc is not consistent. The thing that is making it not consistent is this blue right here, because if I assign NSW blue, there's no valid extension. So what do I do? How do I make this consistent? I delete blue. You're gone. See how it used to be there and now it's gone. So the way that you make arcs consistent is you delete from the tail. You go to the tail you say nope you're gone. You cannot be a valid, you're no longer a valid candidate, because if I pick blue here, I know for a fact that there's no valid successor at s a. Let's keep going. Okay let's check this one. Does this arc look familiar? This arc looks familiar. Why is this arc familiar? Because it was the very first one that we checked. Remember the first time we checked this arc it was consistent. But now let's recheck this arc.

Arc Consistency of an Entire CSP (4/6)

- A simple form of propagation makes sure **all** arcs are consistent:

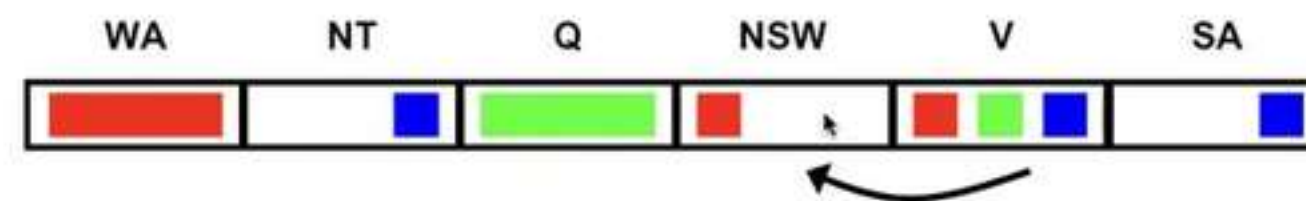


- Remember that arc V to NSW was consistent, when NSW had red and blue in its domain
- After removing blue from NSW, this arc might not be consistent anymore! We need to recheck this arc.
- Important: If X loses a value, neighbors of X need to be rechecked!

Is it still consistent? Let's check. If I pick blue at V, is there something valid at NSW? There sure is it's red. If I pick green at V, is there something valid at NSW? Sure is it's red. If I pick red at V, is there something valid at NSW? No. So this arc is no longer consistent. It's kind of interesting. Arcs that were once consistent can actually become inconsistent. Why did this arc. Become inconsistent because when we first looked at it, NSW, had both red and blue, but in the process of checking some other arc, we deleted blue from NSW.

Arc Consistency of an Entire CSP (4/6)

- A simple form of propagation makes sure **all** arcs are consistent:

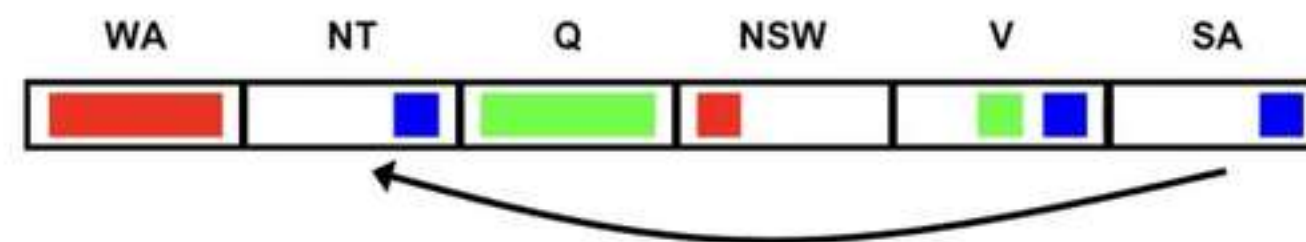


- Remember that arc V to NSW was consistent, when NSW had red and blue in its domain
- After removing blue from NSW, this arc might not be consistent anymore! We need to recheck this arc.
- Important: If X loses a value, neighbors of X need to be rechecked!

So now that blue is gone from NSW, this arc is no longer consistent, because this blue was like supporting Red's okayness. The reason why red was okay here in the tail was because there was a blue here to save it, but now there's no more blue to save the red in V. So we actually have to make this arc consistent, and the way to make it consistent is by deleting this red. So it turns out arcs can become inconsistent and as you go and check the consistency of an entire CSP. By checking this arc and then this arc and this arc, you actually have to recheck arcs as you go. So whenever you change NSW, you need to check all the arcs pointing at NSW. You need to check that one and that one and that one, because something might have been consistent because of NSW's blue, but now the NSW's blue is gone, those arcs might become inconsistent, and you need to go and make them consistent again. So how do you make this inconsistent? Well, this red has no valid extension, you're gone, deleted.

Arc Consistency of an Entire CSP (5/6)

- A simple form of propagation makes sure **all** arcs are consistent:



- Arc SA to NT is inconsistent. We make it consistent by deleting from the tail (SA = blue).

And you keep this going on and on and on. I haven't presented an algorithm yet, but I'm just telling you that you check all these arcs over and over again. Let's check one more. So let's check the SA to NT arc is this arc consistent for everything in the tail? Is there an extension in the head? If I pick blue here, is there a valid assignment here? No, the only one left is blue, and these two cannot be the same color. They're neighboring territories, so this arc is not consistent. How do I make it consistent? I delete from the tail, I delete blue because you have no valid extension.

Arc Consistency of an Entire CSP (6/6)

- A simple form of propagation makes sure **all** arcs are consistent:



- SA has an empty domain, so we detect failure. There is no way to solve this CSP with WA = red and Q = green, so we backtrack.
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Goodbye, gone. And look, what happened? Just by checking a bunch of arcs over and over again, we were able to detect failure extremely early. Like, look at this graph, you and I. As humans, we can barely tell that this is doomed to failure. You're like, this is fine so far, you're red and you're green, but Arc Consistency. Looks really far into the future because it checks all pairs of variables, even unassigned ones. So actually, our consistency already detected that this assignment is doomed to failure, even though you and I, as humans, can barely see that, that's true. That's pretty cool. So Forward Checking had to run a few more steps before realizing that something was doomed. Our consistency is more powerful because it checks arcs between unassigned variables. You can actually detect, for this particular example, just based on these two assignments, that things are totally screwed and there's no more solutions. You should backtrack and go pick something else off the fringe, because the entire sub tree with w a, equals R. And Q equals green, guaranteed to have no solution, our consistency proved to be so. Pretty cool. Also kind of tricky. So something you'll have to play with in your homework to get more familiar with. Okay. So delete from the tail. That's the only one thing you take away today. Delete from the tail. So when an arc is inconsistent, delete things from the tail to make it consistent.

Arc Consistency of an Entire CSP (6/6)

- A simple form of propagation makes sure **all** arcs are consistent:



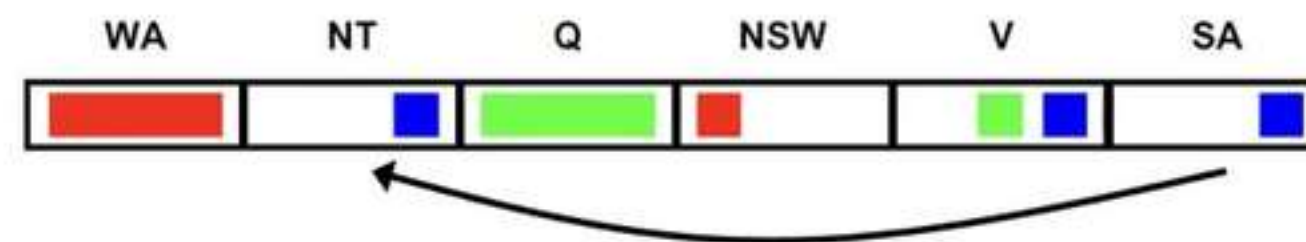
- SA has an empty domain, so we detect failure. There is no way to solve this CSP with WA = red and Q = green, so we backtrack.
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Questions? Yeah. What's up? You're good. Yeah. I guess you could. So if you're a human, you could look far ahead using your brain and be like, well, you have three neighbors and you're screwed. Yeah. I guess you're right. But arc is.

Consistency can actually see that where forward checking could not. Yeah, I guess you're right, so as humans, you can maybe be smart and see it. Okay, cool is our good, different way of saying fringe? That's a great question. The answer is no, because our consistency is a property of a c s p.

Arc Consistency of an Entire CSP (5/6)

- A simple form of propagation makes sure **all** arcs are consistent:



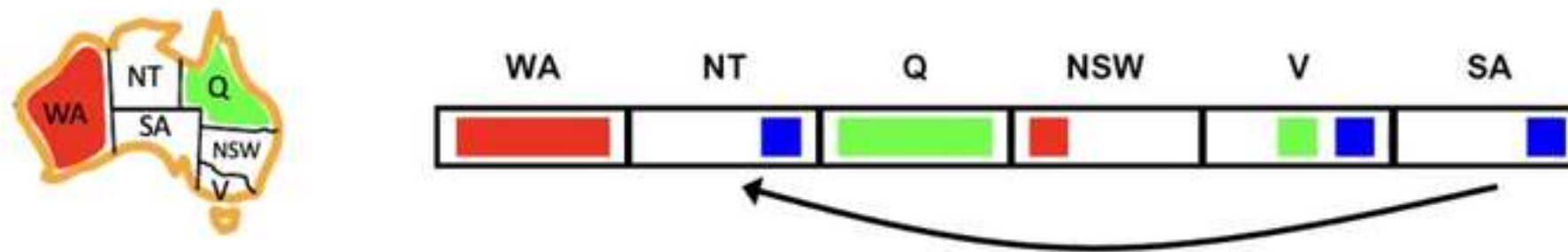
- Arc SA to NT is inconsistent. We make it consistent by deleting from the tail (SA = blue).

You are either are consistent or you're not, and to make it consistent, you delete things. That's totally separate from the backtracking search that you still have to do, so our consistency and forward checking are things you do as you backtrack search to detect failure early. That's a great question, though. Totally different from the fringe, it's something you run within. The search to try and make your search faster by detecting failure earlier, like in this case. Okay, I guess in this case, just seeing three neighbors doesn't show that you're totally screwed. You have to think a little bit further ahead and be like, Well, there's three neighbors, so one of them has to be this. It's a lot it's a lot of thinking. Okay and our consistency does the thinking for you. Okay, I haven't actually shown you how to make this an algorithm yet, so here's your actual Our Consistency algorithm. So far it's a property right here.

I'm just telling you, it's either consistent or inconsistent, and if it's inconsistent, here's how you fix it you delete from the tail.

Arc Consistency of an Entire CSP (5/6)

- A simple form of propagation makes sure **all** arcs are consistent:



- Arc SA to NT is inconsistent. We make it consistent by deleting from the tail (SA = blue).

Enforcing Arc Consistency in a CSP

```
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
   $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
  if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
    for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
      add  $(X_k, X_i)$  to queue

function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
  removed  $\leftarrow$  false
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy the constraint  $X_i \leftrightarrow X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true
  return removed
```

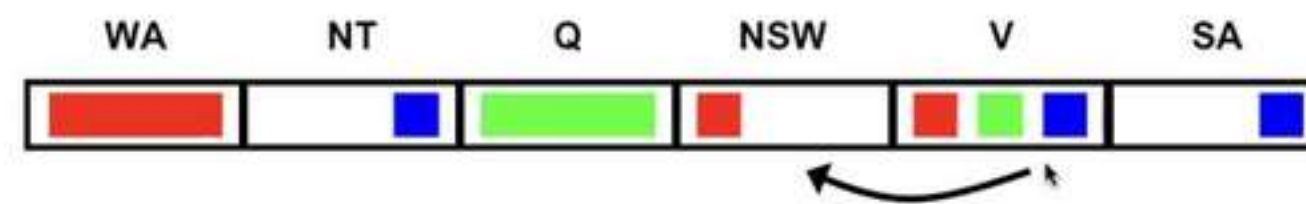
- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- ... but detecting all possible future problems is NP-hard – why?

[Demo: CSP applet (made available by ayspace.org) -- n-queens]

Here's the actual algorithm. There's a lot of algorithms out there, this one's called a c three. Here's how it works. So a c three has a queue. This is a queue inside of a queue, because you're doing backtracking search still, but while you're doing backtracking search, you enforce arc consistency with a queue. So it's like a queue inside of a queue. It's like an exception. So what do you do? You keep a queue, and the queue is the list of arcs you have not checked yet. And you check the arcs one by one. And each time you check the arcs, you delete from the tail. So if the arc is inconsistent, you make it consistent by deleting from the tail. And if it's ever the case that one of them becomes totally empty, like one of the domains becomes empty, you can detect failure early. And one more thing you have to do, the reason why this code is kind of ugly and long is because if. You delete something from the tail.

Arc Consistency of an Entire CSP (4/6)

- A simple form of propagation makes sure **all** arcs are consistent:



- Remember that arc V to NSW was consistent, when NSW had red and blue in its domain
- After removing blue from NSW, this arc might not be consistent anymore! We need to recheck this arc.
- Important: If X loses a value, neighbors of X need to be rechecked!

Remember this case right here. You have to recheck a bunch of arcs pointing at that variable, so if you change NSW, you need to recheck everyone pointing at NSW. Because something in NSW like the blue might have been supporting the consistency of other arcs pointing at NSW. And once you delete the blue from NSW, you must recheck all those other arcs. Because they might have something that is no longer consistent, because the consistency was relying on the blue that is now gone.

Enforcing Arc Consistency in a CSP

```
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
  ( $X_i, X_j$ )  $\leftarrow$  REMOVE-FIRST(queue)
  if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
    for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
      add ( $X_k, X_i$ ) to queue

function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
  removed  $\leftarrow$  false
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy the constraint  $X_i \leftrightarrow X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true
  return removed
```

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- ... but detecting all possible future problems is NP-hard – why?

[Demo: CSP applet (made available by aispace.org) -- n-queens]

So this is an actual algorithm as a queue that processes all the arcs one by one by one and checks them to see if they're consistent, and, if anything. Is inconsistent, you remove values from the tail and then put some more arcs on the queue, and you do this over and over, and over again until the queue is empty, and once you do that, you know for sure that the CSP is arc consistent. All of this is running inside backtracking search. This is just a way to help you detect failure early as you go. So you're like, I sign this and this and this arc consistency, please look in the future and tell me if I'm in trouble, and that's what it does. Yeah. Yeah, that's a good point, so this algorithm a c three, it starts with all the arcs on the queue. Yeah. So you generate all the arcs and then you check them one by one by one, as you check, you might have to put some arcs back. On the queue for rechecking, and then once everything is checked and rechecked, you're done. And you know for sure, the CSP is are consistent and it's allowed you to look in the future. Because it's possible that by enforcing this, you end up in a state like this. That tells you that failure is inevitable and it's time to go look somewhere else. Yeah more questions.

Oh yeah sure runtime. Oh oh what's the runtime of forward checking? I didn't present it as an algorithm, so I don't really have a good answer.

Enforcing Arc Consistency in a CSP

```
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
  ( $X_i, X_j$ )  $\leftarrow$  REMOVE-FIRST(queue)
  if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
    for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
      add ( $X_k, X_i$ ) to queue

function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
  removed  $\leftarrow$  false
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy the constraint  $X_i \leftrightarrow X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true
  return removed
```

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- ... but detecting all possible future problems is NP-hard – why?

[Demo: CSP applet (made available by aisspace.org) -- n-queens]

Filtering: Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

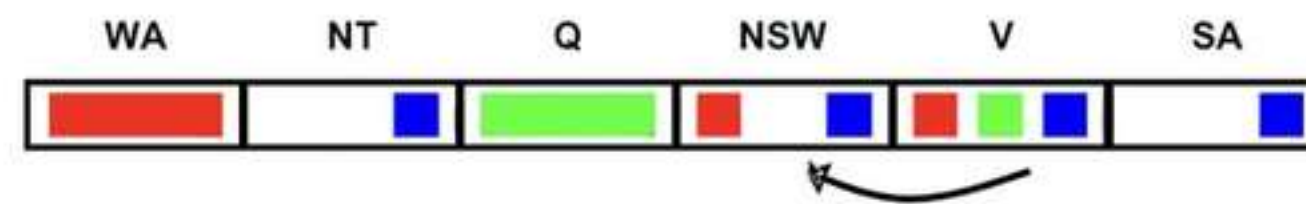


WA	NT	Q	NSW	V	SA
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

I didn't present it as an algorithm, so I don't really have a good answer. If I had to guess, I'd say it's D^2 because you have to check. [afraid] Is it D^2 ? It might be $D \times N$. And maybe it really catching me off guard. But I think it might be something like $D \times N$. Because for each of these n variables, you have to check D values. Check all these values against W equals red. Yeah, you got me. As on the slides, Okay, yeah, good call though. Yeah. This one has a more complicated runtime. I'm not gonna go through in crazy detail. But there's a d^2 term. Why is there d^2 square? Because you're doing this thing right, like, where are you? You're doing this thing right?

Arc Consistency of an Entire CSP (1/6)

- A simple form of propagation makes sure **all** arcs are consistent:



- Arc V to NSW is consistent: for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

There's D values here and D values here, so you have to do this like D square thing. And then how many pairs are there? This n squared pairs, because you have a pair, these two and these two, You have an N squared term and then the final term.

Enforcing Arc Consistency in a CSP

```
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
  ( $X_i, X_j$ )  $\leftarrow$  REMOVE-FIRST(queue)
  if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
    for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
      add ( $X_k, X_i$ ) to queue

function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
  removed  $\leftarrow$  false
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows ( $x, y$ ) to satisfy the constraint  $X_i \leftrightarrow X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true
  return removed
```

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- ... but detecting all possible future problems is NP-hard – why?

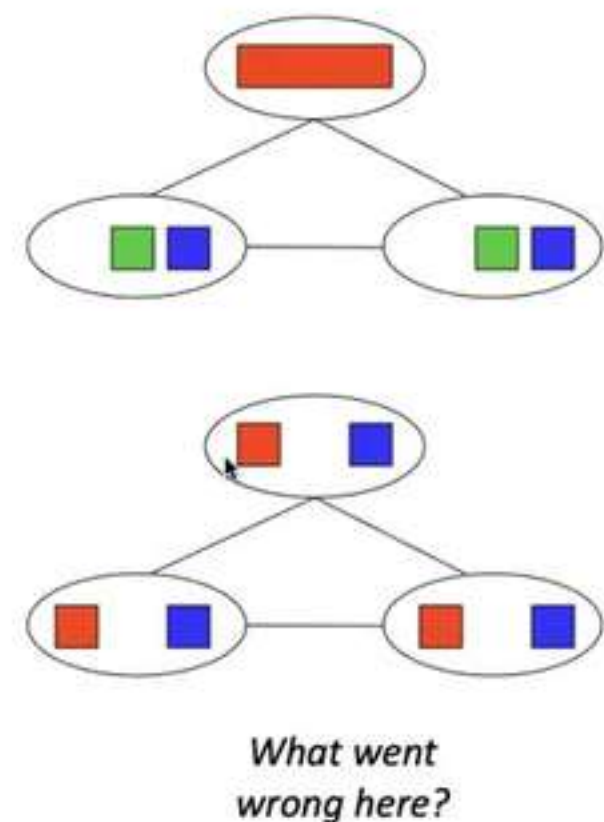
[Demo: CSP applet (made available by aisspace.org) -- n-queens]

The reason why it's D^3 is because you have to recheck arcs as you go. So you might have to rerun this entire process D times once for every value that you lose. That was really really rough. Ask me later if you want to see it in more detail. Point being it's polynomial. That's sort of the catch here. At least it's not exponential. So you can do this somewhat efficiently. It's polynomial. It's not something like two to the end that's ridiculous. So you can do this in a reasonable amount of time, and by doing it, it lets you see a little bit into the future, to see if failure is inevitable or if I should keep going. Cool. Okay. I'm almost done. Don't go yet.

Limitations of Arc Consistency

- After enforcing arc consistency:

- Can have one solution left
- Can have multiple solutions left
- Can have no solutions left (and not know it)

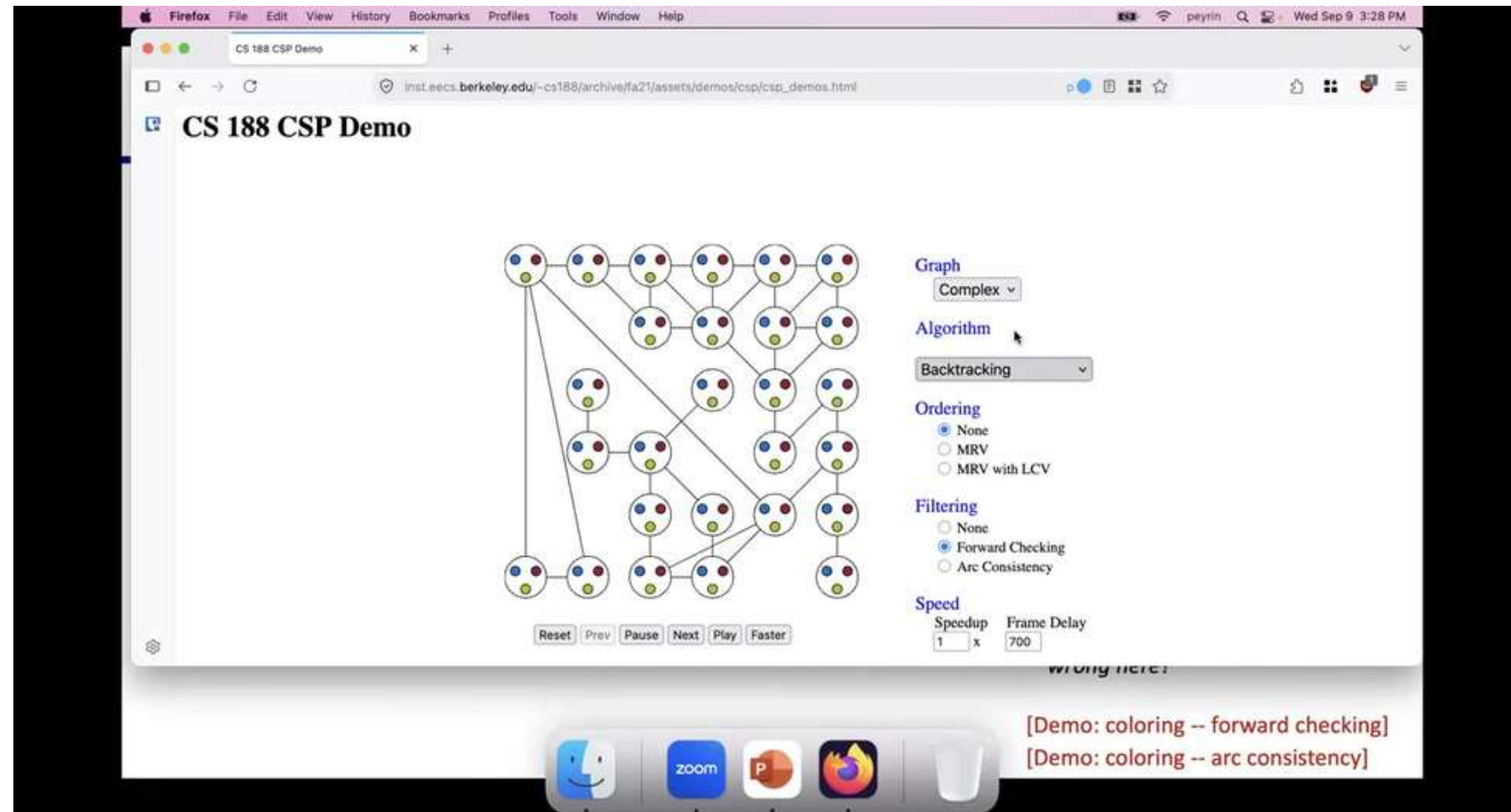


[Demo: coloring -- forward checking]

[Demo: coloring -- arc consistency]

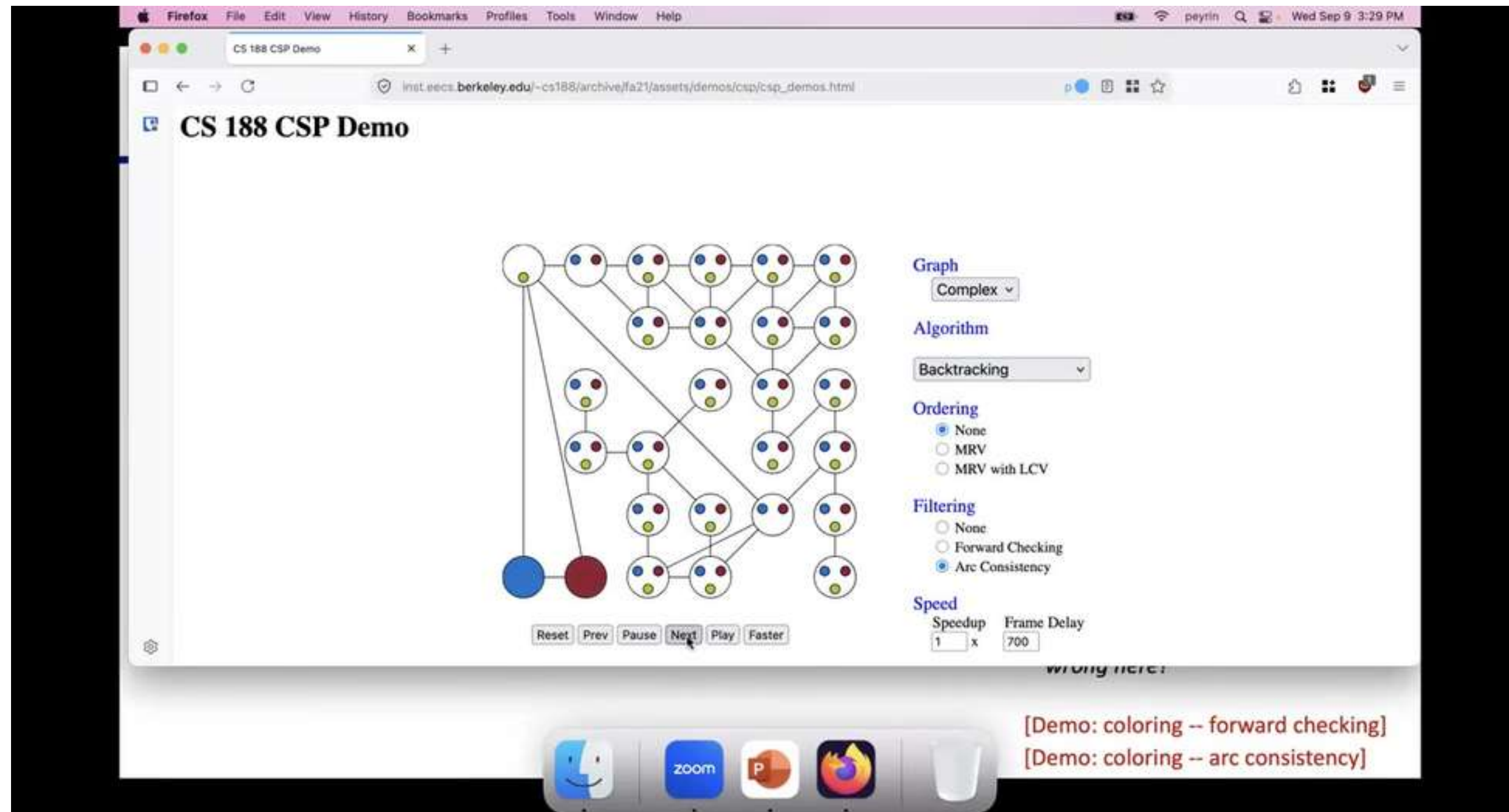
So first, I'm gonna tell you that our consistency is good, but it is not perfect. You can't be like, oh, I have our consistency, I'm gonna solve all my problems using it. You still have to do backtracking search. So let me give you an example where our consistency doesn't solve all your problems. So look at the one down here. You can run our consistency on this. In fact let's do it. Okay, let the arc from the bottom node to the top node, are you consistent? If I sign red here is there a valid extension? Yes. If I sign blue here is there a valid extension? Yes it's red. [afraid] So this arc is consistent. What about the other direction? Also consistent. In fact every arc here is consistent. For every value in the tail, like red, there's an extension at the head, it's blue. For every value at the tail blue, there's a valid extension red. So. Every arc here is consistent. Are we doomed? Yeah, we're doomed because there's no way to finish this assignment right. If I put red here, I can't put both Blues here and so on and so forth. So, arc consistency is good, but it will not solve all of your problems. It just allows you to peek a little bit into the future to see if you're definitely doomed, or if you're maybe doomed, maybe not, and we should keep going. But if one of these domains is empty, then you know for sure that you're screwed. Okay, let's do our consistency on this demo and then I'll let you go.

So the big graph, the big graph, right. So arc consistency is very powerful. It can do things on this larger graph. So let's do backtracking and let's do our consistency. Okay here we go. Start assigning at the bottom. Blue, Good, and what did this do? It just crossed off blue from two of his neighbors.



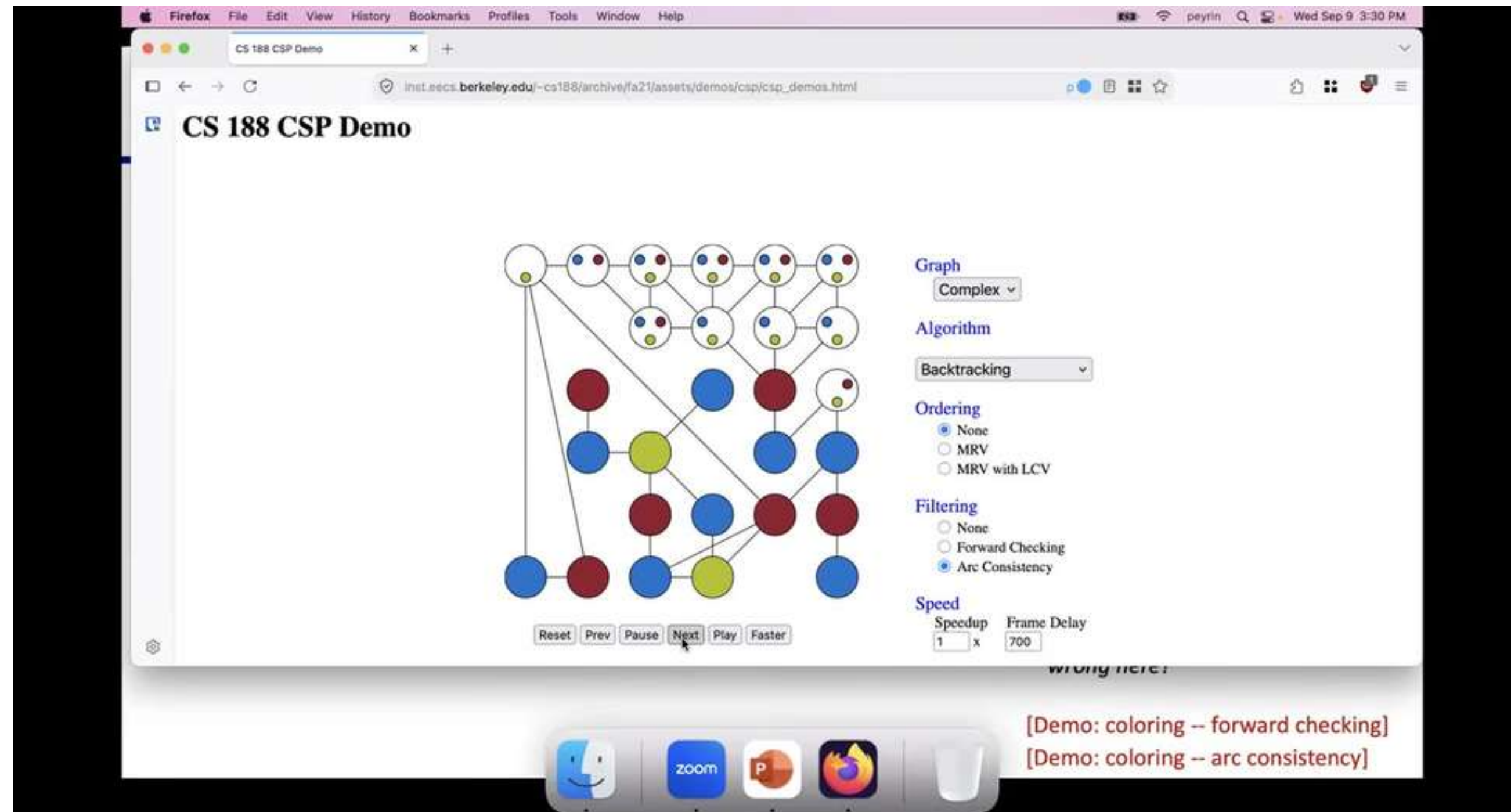
Slide 101 · 1:19:12

Nothing too special yet, but here comes the interesting one. Green or, sorry, red? Okay, now check this out. Not only once I assign red. Not only did this variable lose some of its values, but this one also lost some of its values, and so did this one. So our consistency looks further into the future. It propagates the consequences of your assignment just a little bit further. Let's assign one more. Okay, this one got assigned blue and check this out. This variable, it's so far away. And yet our consistency was able to figure out. Oh, because you assigned this blue, red is no longer valid here. It checked all the arcs, the thing that we just did. And so as you assign, you'll notice that it's looking really far into the future. And it's allowing you to lose values from domains that are really far away from your current variable.



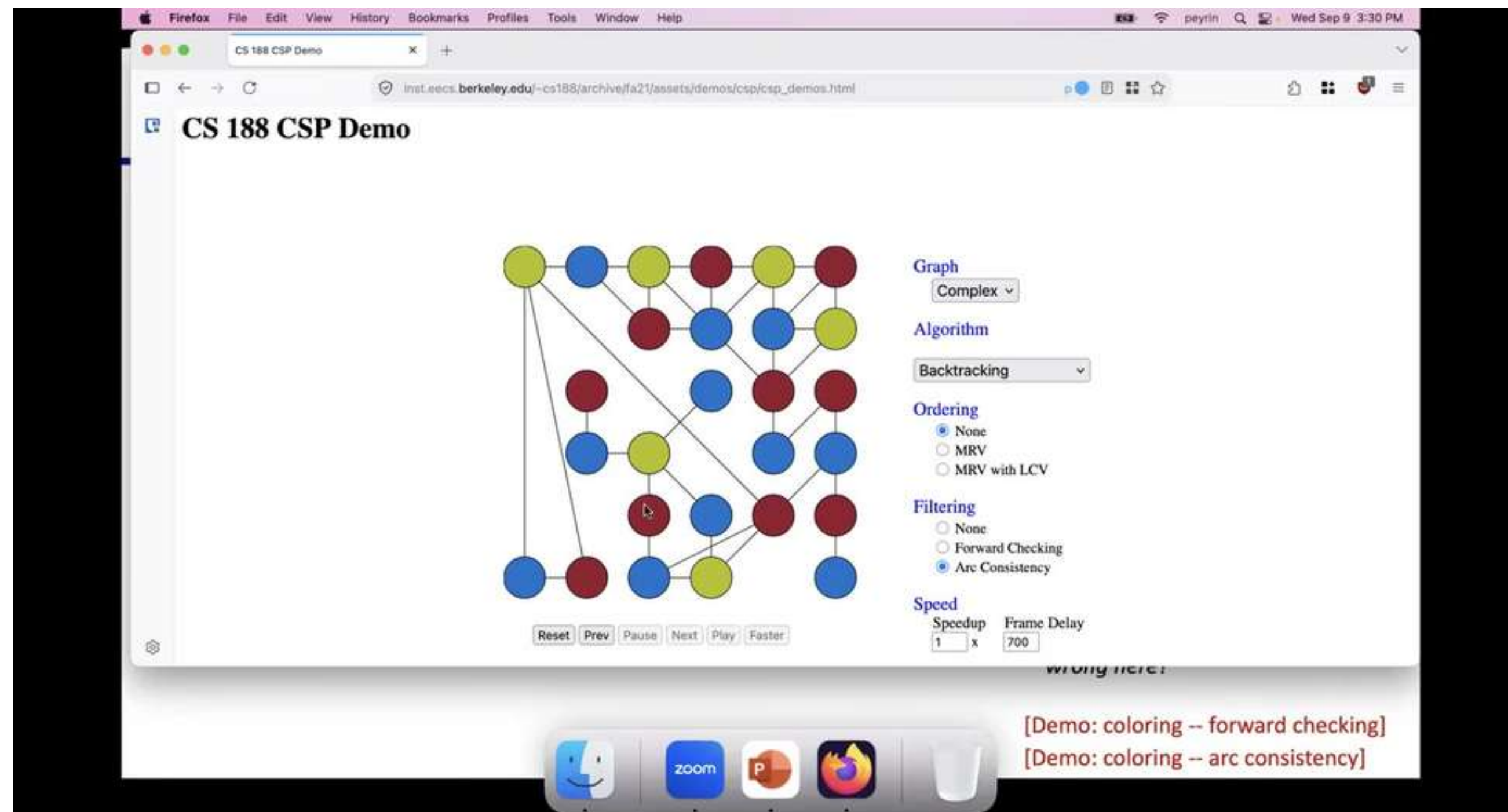
Slide 102 · 1:19:30

And so as a result, as you do this, you can start detecting failure earlier. Now it's not perfect. So like, look right here, right, you start here, you send this. You run our consistency, all the domains empty out, and you're like, oh man, there's no answer. So I need to backtrack. So backtracking is still inevitable, but with some luck, you can get through the big graph.



Slide 103 · 1:20:18

Pretty cool. Okay, I'm out of time, so come back next time if you want to see more our consistency and other ways to brew CSP's. Yeah but we did it we solved the big one. Very cool. I have some time up here if you have questions.



Slide 104 · 1:20:36

Full transcript

Okay let's talk about CSP's. Okay Search you saw Search last time. Search made a lot of assumptions about the world in which we were in. For example, Search assumed that there was just one agent that was you. There was no one else in the world doing adversarial or other actions that could affect you.

Search also assumed that there was a deterministic world. In other words, once you come up with your master plan that says, I will take this action and this action, and then this action, it's expected that if you execute those actions, they are guaranteed to work and your plan will execute exactly in the way that you planned. This might not be true. Maybe you take an action and it doesn't do what you expect. Search also assumed that you could fully observe the state.

In other words, you know exactly where you are, and once you take an action, you know exactly where you end up. So Search made all of these assumptions, and what we'll do in the rest of the first half of the semester anyway, is we'll start relaxing these assumptions. What if this isn't true? This isn't true. What if this isn't true?

By relaxing them, we'll be able to solve different types of problems. That's sort of the sketch of what's gonna happen in the rest of this first half of this semester. And so this section, this lecture, and the next one, will be about a specific type of search problem. And so to motivate that, first, let me remind you that search problems that we've seen so far, you can think of them like planning problems. So for example, here we have this like thief robot, and the robot wants to get the gem and the robot's thinking, well, if I want to plan ahead, these are the sequences of actions I can take so that I can go and successfully steal this gem from its holding position.

And so what we would think about is the path to the goal. In other words, what sequence of actions allows this robot to get the gem? And this path would have things like costs. For example, each action could have a different cost, and we want to find the least cost. We think about the depths of different paths and to help you solve these problems, what you would do is take your problem, like the robot and the Gem.

You would formulate it as a specific instance of a search problem, and then you would give the problem to one of your many general purpose algorithms like Breadth First Search, Uniform Cost Search, a Star Search and along the way, you can also give this problem a special hint, like a heuristic that says, I'm getting closer to the solution. I'm getting further from the solution, takes it a state and outputs a real number telling. You, whether the state is good close to the goal or bad far from the goal, this is all stuff from last time. Okay, and the reason why I'm telling you this is because today is more about a specific class of search problem called the identification problem. So here we're actually not too concerned about the path to the goal, and we're instead more concerned about a specific set of assignments to variables.

So, as an example, let's say that here we have this detective robot. This robot comes in after the gem was stolen, and the detective robot wants to figure out what was what happened to the gem? And like, what happened here? Why is the gem gone? Why is it, ah, lying underneath this blanket?

And so here we don't really care how, the detective. Gets to the answer We just care about the assignments to the variables. In other words, what's the explanation for why this thing happened? We don't really care about how the detective gets there, we just care that there is an answer. And so this is a specialized class of search problems that we'll look at, and because it's a specialized class, we'll have specialized tools to solve it.

It's all kind of abstract I will make it very concrete. Okay, but interrupt me if there are problems, or tech problems, because it's a new building, I still don't understand how all the buttons work. Okay cool. It's all like about constraint satisfaction problems, or CSPs, for short. So what are they?

So first let's think back to. One more time, because CSP's are a special class of search problems. If you think back to the general purpose algorithms we developed, like uniform cost Search, a Star Search, Depth First Search, those search problems didn't care about what the specific underlying search problem was. In other words, you would just give it a state and you would say, here's the state, and what can you do with the state? You can't open it and look inside.

Remember, if you're working on depth first search or uniform cost search, you don't get to look inside the state and say, oh, the state is Romania, or the state is pancakes. All you get is a black box. It's a state and you can only ask two questions. Question number one, are you a goal? And question number two, what are your successors?

And that's it. You can't look inside the state. And this allowed for our algorithms to be very general purpose. Depth First Search can solve any search problem as long as you formulated using this formulation. But it was limited because you can't look inside the state and see what's going on.

Okay. And so you can almost think of the search states as these black boxes, and the only thing you can do is present it to this judge, and the judge would say, yes, it's a goal, no it's not a goal, yes it's a goal, no it's not a goal. It would just give you a yes no answer, and that was it. So it's general purpose, but that also means we can't use any specialized techniques on it. Now CSP's are more specialized, so here I'm actually going to let you open the search state for the first time.

So in general search problems like pancakes, Romania can't do that. But for CS P's? I'm going to let you open the state and look inside, and I will tell you that for this

special subset of problems. When you open the state and you look inside. What you will see is a set of variables, and each variable has a value from a domain.

And again, examples will hopefully make this more clear. But the idea is there's a bunch of variables, and each variable has like a list of possible values. So this variable can be one of these five values, this variable can be one of these five values, this variable can be one of these ten values, so on and so forth. And your goal test, which I'm also going to let you open up for the first time, so instead of a judge telling you, yes, no, yes no. I'm going to let you open up the goal Test, down inside the goal Test, you will see a set of rules, also called constraints, that tell you what values can and cannot be assigned.

So it's still a search problem. There's still a state. There's still a goal test. Still a successor function, but the difference is now, for the first time, I'm letting you open up the state and open up the goal Test, and by doing so, you will get special purpose algorithms that solve this specific type of problem really well. Okay.

So now the difference is, instead of a judge telling you, yes, yes, no, yes, we have like a little book, and it has a set of laws, and the laws are the constraints, telling you what's allowed and what's not. Okay. And so what we'll do in both search problems and CSP's is we will take general purpose problems, pancakes, pac-man eating all the dots. Today's gonna be about maps and colors, and you'll take these problems, and you'll always formulate them as a specific instance of search problem or the CSP problem, and taking your problem and formulating it as an instance of one of these problems. It's like your first job, and then your second job is once you have the problem formulated in this very specific way, what are the methods to solve it.

So just like last time we'll see both of these things. How do you formulate the problems, and then how do you solve them? And as mentioned earlier, because CSP's are a more specific class of problems, they have more specific algorithms that work. Very, very, very well for these problems only, and not the other ones. Yeah question.

Why isn't a heuristic? Yeah good question. Why is a heuristic part of a state? It doesn't have to be. So if you go back here heuristics don't have to be part of a state.

In fact the heuristic is just a function. It says Here's a state. Give me a number that tells me something about the state. Sorry can you say that one more time? Yeah, so you're using the state.

You're calculating a state. Yeah, so the question was about how states and heuristics are related, and the answer is a heuristic is a function. It's a piece of code and it

takes in one. Argument which is the state? And it outputs one value, which is a number, telling you something about the state.

Yeah good question. Anything else? Last one. Okay cool so this is the thing we're gonna formulate. It's kind of abstract now.

Let me show you some examples. Okay, so the running example for this lecture, and also in your book, is map coloring. So I hope you're all really familiar with the geography of Australia. So we're about to become really familiar. This is basically what you need.

It's a set of states. So Australia has 1234567 states, and our goal is to assign colors to each of the states. And when you color maps, you ever look at a map with colors on the states in the country. It'd be really nice if two adjacent territories or states don't have the same color. Like this will be really ugly.

If Western Australia and Northern Territory were both red, that's really gross. We want all the neighboring states to have different colors now. This is not the only CSP example, so don't go out into the real world and start thinking every CSP is maps. In Australia not the case, so there's lots of other CSP's, and I'll show you some examples later. But the running example for us, just to see how to solve them and how to formulate them, will be Australia Map.

And the rules are we need to assign colors to all the states so that no two adjacent states have the same color. Same color. Simple enough. Okay. So we can formulate this.

We can take this problem that I just described to you in English and make it into a c s p, which has very specific mathematical definitions. So the CSP asks What are the variables. Well, in this case, I could say the variables are the states. So look, there's a variable for w a, that's Western Australia, one for the northern territories, one for Queensland, New South Wales, what is this? Victoria, South Australia and Tasmania all by itself in a little island.

So there are seven variables. Then I have to specify the domains. What can I assign to each of these variables. So there are seven variables each one gets a value. And what's the pool of possible values I can pick from?

Those are the domains, and so I can say, well, each of the variables can be assigned to either red or green or blue. So I have now taken my English language problem and formulated it as a specific instance of a c s p, because I told you what the variables are, I told you what the domains are. Now I'm not quite done,

because if I stop right here, you can just say I'll just assign everything red and I win, but can't do that, so I'm going to introduce constraints, rules that say there are certain things you can't do. So I could say the constraints in English, like adjacent regions, must have different colors, that's what I just said to you in English. But you can also express these constraints mathematically.

So here are two possible ways to do it. One is, you can write an implicit constraint, and there's like a little piece of code that you can run to check whether or not the constraint has been satisfied. So this little piece of code says, the value in w a, which might be red or green or blue, cannot be equal to the value in NT, which might be red or green or blue. You can run this little piece of code to check whether or not they're the same. And if they are the same, then, well, your constraints have not been satisfied.

But if they're different then the constraint is satisfied. So you can write a little piece of code like this, or you can write it out explicitly, and you can say, well, if I'm assigning w a and n t i can assign w a equals red and NT equals green, or I can assign. W a equals red and NT equals blue, or I can assign w a equals green and NT equals blue, I can write out the entire set of possible assignments. So you can write a little piece of code, or you can list out all the possible assignments either, one's fine, just depends on how you want to specify your constraints, but here you can see, this is like a mathematical version of the thing I wrote in English here. Cool.

So what's the answer? How do you solve one of these problems? And the solution you're looking for? Here is an assignment, so you want every variable to have a value, in other words, every state must be colored red or green or blue, and you need to satisfy all the constraints, so, for example, the one I'm showing. Here, this is a valid solution.

It's not the only valid solution. In fact, CSP's often have many solutions, but here's just one example of a solution. You can check for yourself. No neighboring states have the same color. So this is a good solution.

Other ones exist too. But this is our goal. We want to take our problems, specify it in this specific way, and then out pops a solution that is an assignment of values to variables that satisfies all the constraints. Haven't told you how to come up with these solutions yet. So far, we're just taking English language problems and turning them into math.

And once we have the math, we can go solve the math later, which we will do. Okay, thoughts on map coloring and Australia. Okay we have Romania we have

Australia. Okay we'll wonder what countries will come up next. Okay let's talk about N Queens.

Have people seen the N Queens problem? Is anyone a frequent N Queens player? Okay maybe not. Okay this is the N Queens problem. So this is a Queen, it's on a little chess board, and if you've played or played chess, you know that queens can attack in the up down direction.

They can also attack in the left right direction. They can also attack along the diagonal. So what our goal here is is to place n Queens on the board, where n is some number you pick such that none of the Queens are attacking, and you can check this. So, for example, I'm not gonna drop, but like this Queen over here. Well, there's no one else in the same row, so it's not attacking any other Queen.

There's no one in the same column, so it's not attacking any other Queen, and there's no one on the same diagonals, so it's not attacking any other queens. So you have all these queens and your goal is to place them on the board so that no two Queens are attacking, and you have a peaceful kingdom, and you don't have these nasty fights. Okay. So how do you take this problem that I just described in English and pictures and turn it into a mathematical CSP object? There's actually multiple ways to do it.

So let me show you two possible ways. So, one possible way you can take this N Queens problem and make it into a c s p, with variables and values and constraints is, I can say that the variables touch panel is offline. I don't know what that means but if. Anything is broken you should tell me. Sorry, this screen just popped up, had some weird error message popped up.

Okay it's back. I don't know what happened. Okay let's talk about ah n Queens again. So I can say the variables are the squares. So, for example, this square is a variable, this square is a variable, this square is a variable.

So I'd have 16 variables here. So what values can I assign to the variables, like the square right here? What values can I assign? Well, this case, I could say this square only has two possible values. So either got a Queen or it doesn't have a Queen.

So the domain might be something like zero and one, no Queen, or put a Queen on that square. So I have the variables I have the domains. I'm not quite done yet, because if I stop here, you might say, well, a valid solution is let me put a Queen on every square. Okay, but that doesn't work, so I need constraints to tell me what is

and isn't allowed. And once again, you can write the constraints a lot, lots of different ways, here's an explicit way to write it.

And so it would say, Okay, well, and again, don't read this too carefully, right? It's like math, you can go share it on your own time later, but it basically says something like, Okay, well, if the two queens are, if you have two squares in the same column, they could either both not have queens. Or one of them can have a Queen or the other one can have a Queen, but they can't both have a Queen. So I can't have like one comma one, because if two queens are in the same row, that would be a problem. And the same thing for all the columns.

So it's a lot of writing, and these are for the diagonals. Oh, if I have this diagonal, I can have a Queen on one of the squares or a Queen on another one of the squares, but I can't have two queens on two squares, otherwise, they would be fighting, and I don't want fighting. So this might be like an explicit way to write out the rules that I just described in English for you, which is that the Queens can't attack. So this is one possible formulation, you can take this formulation, plug it into your favorite CSP solver and hopefully out pops, solution where all the Queens don't attack, and this board is an example where Queens are all nice and peaceful, Yeah, what's up. Yep.

Yeah, you're absolutely right. So the last thing I was gonna mention, you you beat me to it, is if you stop right here, you've actually under defined the problem because there's a cheating solution as the one you brought up. The cheating solution is what? No queens anywhere. Just set every square to zero.

That wouldn't solve my problem. So I need one more constraint that says something like, if you add up all the squares, there are actually N queens on the board. This is a good example of your formulations have to be very carefully specified. If you accidentally miss a rule like Oops, I forgot this. Well, then the CSP solver will just spit out and say, Look, I solved the problem.

I just don't put any queens anywhere on the board. That doesn't solve your problem. You want n queens. So you have to add one more constraint that says, maybe like, add up all the. Squares and queens in total.

Okay, this is a pretty good formulation, but it's not the only one. So turns out, it's often the case in these problems and in the problems on your projects, that you can actually take the problems and rewrite them in different ways. So here's another formulation of the exact same problem. So here I'm actually building a little bit of the solution into the formulation. So me, as a human, I'm thinking, well, as a human, I know that, every row is going to have exactly one queen, because if it's a four by

four board and I need four queens on the board, every queen has to be on its own row.

I can't have two queens on the same row, they would be fighting. So I know for sure that every row has only one Queen. So maybe another. Way to formulate the same problem is to say my variables are actually the rows. So now, instead of 16 variables, one per square, now I have four variables and the first variable is row one, second variable is row two, row three, row four.

So what are my domains? The domains might be like, Where's the Queen on row one? Is it in square number 123 or four? And Where's the Queen on row number two? Is it in square number 123 or four.

So, for each of these variables, Q one through Q four, I would assign a number that tells me is the Queen? In column one, column two, column three, all the way up to column four. This is the same problem as before, I just specified it slightly differently, and here I'm sort of baking a bit of the solution into the formulation because I has a huge. Know that each Queen has to be on its own row. But I would still have to write constraints, and here the constraints are actually a little bit trickier to write.

So I just got lazy and wrote a function that says somewhere in my code, there's a function called non threatening that lets me check are they on the same row, the same diagonal or the same column? And for all queens, they need to satisfy this non threatening function. This would be an example of an implicit constraint. You can also be explicit and write out, Okay, between one and two, they can both, they can be on columns one and two one and three one and four oh, but they can't be on one and one, that would be an explicit version of the same constraint. It's up to you, you can write whichever one you like, whichever one your solver demands.

Okay, so far, so good, okay now. One powerful tool. Because once you start looking at these constraints, they get kind of hard to look at, like, oh, what is this, what is that? Or like, oh, what is this? So one way to look at these constraints and really think through them is to draw a graph.

And here the same warning from last time comes up. It's a graph. This class is just graphs on graphs on graphs. So every time you see a graph, you should stop and ask yourself, What does this one mean? This is a totally different graph from the ones you've seen before, so in this graph, what are the nodes represent?

Like, when I draw a circle, this node represents a variable like this is the Northern Territories variable, this is the Victoria variable, this is the New South Wales variable. So each These circles represents a variable in your CSP that you want to assign values to. What are the edges represent? The edges represent the fact that

there is a constraint between these two variables. In other words, the value of V depends on the value of s_a in some way.

So for example, here, it would be the fact that South Australia and Victoria are neighbors, so they can't be the same color. So I draw this edge to remind myself, oh yeah, these two things can't be the same color. And so by doing this, you sort of get a structural picture of where the constraints are, which I think is nicer than looking at all the math. You can look at the math if you want, but I like looking at pictures. So, the nodes represent variables, and the edges represent constraints that live between the variables.

Okay. Now there's all sorts of different constraints. So for example here, I'm showing a CSP where all the constraints are binary, which means they only relate to variables. So, for example, w_a and NT you cannot be the same color, and that's it. I just thought there.

I didn't have to say anything about this one or that one, or that one, or so on. Okay. But there are other constraints out there. There are unary constraints that say, okay, w_a has to be blue for some reason, or it can only be blue or green. That would be a constraint on one variable.

That'd be a unary constraint. You can also have higher order constraints. So for some reason, maybe you're like, oh, w_a and South Australia and NSW all have to be different colors. That would be a higher order constraint but in this case, all the constraints are binary, they just relate two variables, and because all the constraints relate two variables, I can draw this nice picture where the nodes are variables and the arcs show constraints.

Cool, and then once I have this, I can shove it into an algorithm, a general purpose algorithm that can solve any problem, as long as I formulate it in this special CSP way, where I say variables, values and constraints. Cool, let me show you some other ones, so you're like, oh, everything is map colors, let me prove to you that not everything is map colors, we saw N Queens earlier, here's another example of a CSP that's not maps or colors, or Australia, so. Every one of these little puzzles is like, oh, you need to find a number, a digit that goes in T, and a digit for W and a digit for O, so that this equation makes sense. So how would you do that? Well, once again, I can formulate this ACSP A-C-S-P, so every one of these letters represents a digit.

I wanna know what digit they represent, so I can say the variables are each of the letters, like F is a variable and T is a variable, and W is a variable, and what are the domains? Well, the domains are like zero through nine, because each of these is assigned to exactly one digit. What the heck are X_1 , X_2 and X_3 ? I don't see them anywhere in my problem. Now you have to think back to your arithmetic class.

Why are there three more variables that I didn't ah write down here? Well, like, what if O was equal to? Let's say O was equal to nine? Okay, there's a nine really ugly nine, Let's say that was a nine, was nine plus nine, let's say 18. Ah, I need a carry bit, right?

So the X one and X two and X three. Those are the carry bits that, like, live up here. Because when you do math the sum is greater than ten, you need some carry bits. So X one, X two, x three, they're the carry bits that live up here. Okay, again, not the most important thing.

The sort of take away here is that you can take problems that have nothing to do with maps and formulate them as CSP's. And once again, you need constraints, and the constraints might look something. Like, well, if I add O and O, I'd better get R plus, maybe some carry bit, and if I add W and W, I'd better get U plus some carry bit. And if I add the carry bit and T and t i guess some letter the number O again, and so on, and so forth. And maybe one more constraint to make this problem interesting is, you want, you might want to say, they should all be different, because what if I don't have this constraint?

That all the digits are different, you would just say 000 plus, 000 equals 0000. It's too boring. So when people write these little puzzles or whatever, they might add a constraint that says all the digits have to be different. Cool once again. You can draw it as a graph so you can say, what are the circles?

They represent? My variables FTU. What are these squares? Well now suddenly you have constraints that are higher order. They link up more than two variables.

For example look at this constraint right here. This says O, plus o equals R, and then there's a possibly a carry bit up here. Like nine plus nine equals 18 you gotta carry the one. So this is a constraint that relates three variables, and the way you draw, that is, you say, well, O and R and X one, they're all related by a constraint. So you draw a little square.

This square is my constraint, and the constraint relates R and O and X one. So the circles are variables and the squares are constraints, and the lines tell me which constraints tied to which variables. What about this one up here? Why is there. A constraint that relates all the letters oh, because that's the all diff constraint that says f t u w r, and O, they all must be different, digits cannot reuse the same digit twice.

So I took my problem and I formulated it in a very specific way so that my CSP solvers can give me answers. Cool. Sudoku Anyone play sudoku? Sudoku champion?

Okay, so how does sudoku work?

The idea is we have all these squares and you need to fill in numbers and there's a bunch of rules, right? It's like, well, within the column I can't have the same number twice, and within the rows I can't have the same number twice. And then within each of these little squares, I can have the same number twice. How would you formulate this Acsp A-C-S-P? You're starting to get the hang of it.

Okay, I could say, well, each square is a variable, and each domain might be one through nine, because those are the only legal numbers I can write in Sudoku. And the constraints might be something like, well, within each column, I cannot have the same digit twice and within each row, I can have the same digit twice and then within each of these little squares, this like a three by three regions, I cannot have the same number twice. So if I formulate sudoku in this way, I can plug it into a c s p and solve it. By the way, when you play sudoku, have you ever noticed there are some problems that are really easy? You're like, oh, well, this number is nine, so this one must be one, so this one must be two and then you're done.

And then there are some sudoku problems that are really hard because you have to. Go back and try this thing and then it doesn't work and you have to go back and try another thing. Turns out, the difficulty of that sudoku problem for you as a human is really closely related to how difficult these CSP, problems are for computers. So now, you know, there's easy sudoku problems and there's hard ones. And the difference between them is very closely related to how computers solve these things.

Which is something we'll see soon. Okay, a sudoku. Okay, I'm not gonna keep going through all the examples, but I'll show you just one more. This is not how people do computer vision nowadays, but ah, one of the earlier approaches to computer vision, where you wanna, you know, see what the picture is is you would get this little line drawing, and in this line drawing, you have to stop and ask yourself, it's like, a. One of those like visual tricks that play on your brain, like, look at this corner right here.

Like, is that corner in inY? Because it's, ah, facing inwards, or is like an outy? Because it like, sticks out? And so what you could do is there's some clever way you can formulate this Acsp A-C-S-P. And you can say, well, if this one points inwards, well, then this corner has to point out and this one has to point in.

And you can somehow use this to detect which parts of this picture are like an iny or an outy, right? So this could be like something you can, ah, this could be like a

little stage you can step on, or like this would be a box that you're looking at. So which of the two is it? A c s p can help you solve that again? This is not how people do it nowadays, but the point is CSP can solve things that are totally.

Really different from maps and colors, even though we will spend most of today on maps and colors. Okay cool. Alright, so in this case, it was an INY all along, right? It was a INY because it's like a little corner that I cut out of the cube. But if you look here, maybe it was a box, you don't know, the visual illusion.

Okay cool. Let me tell you a little bit about different types of CSPS. There are some CSPS we'll talk about, there's a whole class we will not talk about because we don't have time, but let me just remind you that they exist. So it turns out there's lots of different CSPS. So we will focus on CSPS, where the domains are finite, like the colors are red, green and blue, or the assignments are true or false, or the sudoku.

Puzzle was digits one through nine, but there are also CS, p's where the domains are infinite. For example you can put any integer in the box. How many integers are there, like infinitely many, or what? You can put a real number in the box or a string. Well, then now it's like there's infinitely many values you can put in the box.

Those require different approaches from what we're about to see today. And there are also CS, p's that have continuous domains, like for example, like scheduling, like when are we all free? Right? That could be a continuous ah, domain, because time is continuous, right? And the variables might be like, oh, when does this job start?

When does this job end? When are you free? When am I free? We can schedule rooms. Like, who's teaching in which classroom, so that the whole campus works and we don't have two classes in the same room.

So there's all these different problems, different approaches to solving them, if you've ever seen a linear program, that's a special class of CSPS as well that we will not talk about here. But point being, we're just looking at a small subset of them. I'm just reminding you that other ones exist. Right, so there's also other ones, like continuous variables, again, if you have linear programs, those are ways you can solve them, maybe instead of just three colors, you can use any color that you can possibly think of, well, now, there's too many variables to think about, infinitely many even. And so we will not think about this, but just do know that they exist.

So there's infinite domains and there's also continuous domains. Okay. And there's also different types of constraints. So we already talked about the ones we will

cover, unary constraints, talk about a single variable, like South Australia cannot be green for some reason. Binary constraints are pairs like these two states cannot have the same color.

And then higher order constraints are, like the crypt arithmetic from earlier, the two plus two equals four, where you have three variables that are related by a single constraint or more. And one thing we will not talk about is preferences. So maybe you want, you'll say, I want to color the Australia map, but I like red. Please use red if possible. And how would you do that?

You'd have to add costs and that's not something we're gonna talk about, but it's something you can do. You can go check it out on your own time, but we will just. To constraints that are either satisfied or not, but do know that preferences are something, you could also bake into these problems if you want to, or if your problem demands it, because you really want red for some reason. Okay. Cool, and again, there's so many CSP's out there, right?

So as mentioned earlier, right? There's like, when can we meet? Or you know, like, where do you put the classes? Or who teaches the classes? Or you know, how do you schedule?

Airplanes so that they land and depart at the correct times? And they don't like bump into each other in the air and explode and things like that. So lots of different things can be solved using CSP's, and they all involve assigning values to variables. Okay. Just remember, it's not all maps, even though they're all maps today.

Okay, we're done with formulating, we've taken all sorts of different problems, the cryptarithmic, the sudoku, the n Queens, and we made them into CS P's, now we will build a general purpose algorithm for solving CS P's. You ready? Okay. Let's get end of first half and now let's do the second half, let's solve some CS p's. See if questions everything on Zoom okay.

Okay no explosions very good. Let's talk about solving CS p's. Okay, so it turns out CS p's, as mentioned way at the start of today, it's just a special version of Search. We have states, we have a successor function, we have a goal test. So we can.

Actually, just use basic search algorithms like d f s b f s to solve CSPS, but it turns out those will not be efficient, and so we'll come up with improvements that only exist for CSPS, where you can look inside the state and see values and variables. But let's do the dumb thing first. So the dumb thing would say What's my initial

state? The initial state is an empty map. It's a blank map of Australia.

It's a empty assignment where none of the variables have values. What color is Western Australia? Don't know yet. It's blank. What color is Tasmania?

Don't know yet. It's blank. And the successor function, which takes a state and gives you all the successor states, would say, let's assign a value to one of the variables that is currently not assigned. So your successor function. I'd say, well, given this state right here.

You can color Western Australia red, or you can color Western Australia blue, or you can color it green, or you can color Tasmania red, you can color Tasmania green. So it takes a single variable, assigns it a value. And now you had a new state, and then once you're at that new state, you can call Success Successor Function. And it will tell you what other assignments are available, and so on, and so forth. And what's the goal Test?

Who's the judge? The judge looks at your map or looks at your CSP and asks, is the assignment complete? Did you assign every variable? And also did you satisfy all constraints? And the goal Test gives you a yes or no?

Answer? Yes, it's a complete assignment that satisfies all constraints, or No, it is not complete or did not satisfy all constraints. So this is a simple yes no test that you use to check whether or not the CSP has been solved. So you can see, I've taken my CSP and I've made it into a search problem. So it's another case of taking a problem and formulating it as a specific instance of another problem.

So this works and go around d s d f s b f s, you will get the answer, but it's really inefficient. Let's see why. Okay here comes the annotation again. I will try and get this fixed in next week, but until then, let's think about BFS. So BFS, would say, let's start with the empty assignment.

The empty assignment is there's no variable. And then we can ask for all of its successors. So what are your successors? Well, your successors are something like, ah, let's do. Queensland is blue, or maybe another successor might be like, Queensland is, or my other colors, red, okay, and then another successor might be Queensland is green.

Okay, very good. And then there's also other successors using the other states. But in the interest of not boring you with my drawing skills, I'll cut it here, okay, so

these are the three successors. What does Breadth First Search do? Well, it picks one of these and expands it, so let's pick this one.

We'll expand it. Okay what are the successors of this one? Well, the successor would be something like, Queensland equals blue, and let's do another one, ah, Tasmania equals blue. That's another successor, and another successor here might be okay, Queensland equals blue and Tasmania equals green. Okay, maybe you're starting to see a problem here, because what will BFS do?

BFS will continue expanding and it'll expand all these. Okay good we'll expand these and then we'll expand these. I'm not gonna draw them because I don't have time, but each of these will be an assignment of two variables. Okay what does BFS do next? BFS will pick one of these up.

Q and then expand. So remember BFS sort of stripes through, ah, from top to bottom, going layer by layer, right? So we do the first layer, we did the second layer, we did the third layer. Okay where are the solutions to this problem? Are there any solutions at this layer?

No, because at this layer, only two of the variables are assigned. So I can guarantee you there's no answers here. A solution requires every state to be colored, and here only two states are colored, only two variables have values to use the general term. Okay so we'll go and expand all these. Do you think we'll find any solutions at the next layer?

Any solutions here? Nope because the solution or the next. Layer will only have states with three variables assigned. Where do all the solutions live in the BFS search tree? They live at the bottom because that's the place where all the variables get assigned.

So if you think about your BFS cartoon, go away. Okay if you think about your big BFS tree, you're striping through layer by layer by layer. Okay no solutions, no solutions. Where are all the solutions? All the solutions live down here.

So BFS just wasted all this time looking everywhere where the solution is, not before ever looking at a single solution. So, BFS, at first, when we saw Search last time, I felt pretty good for driving directions or something. For CSPs, oh no, BFS is like the worst possible choice, because you're gonna look everywhere where there's no solutions before you ever start looking at this bottom layer, where some of those solutions live. So it turns out for this one case, ah, where's my next slide? For this one case?

It's actually the case that DFS is a little bit better. Why is that? Because if you think about DFS. And the search tree, what Dfs D-F-S. Do?

At least it dives down here and tries to find some solutions first, because you know, the solutions live at the bottom. That's the place where all the variables get assigned. So at least DFS. Comes to the bottom and tries out solutions. Whereas BFS.

Tried everywhere where it's not before looking. For solutions. So this is actually a case where DFS is better. So last time we were laughing at DFS being like, oh, you give us all these crazy solutions, but this is actually a case where DFS, does the right thing. It at least goes down and tries to find solutions that work.

Okay great. So let's try using, let's try using DFS to solve this c s p. I have a demo. I forgot to pull it up earlier, so you're gonna have to bear with me while I pull it up. Okay.

No Firefox. I don't care about what's new. I just wanna see my CSP demo. I'll put this on the website at some point. I promise.

Okay. But point being ah I have a c s p demo. It's basically. The map coloring problem from earlier, even though, remember, not all CSPs are maps. Where are you?

CSP demo loading? No, I don't want to login, I have to login, ah, what do I do. Don't look at my password. No, I don't want to do two factor. Auth.

Okay, but I'll do it anyway. Okay, very good. CSP. Demo, Great, that wasn't as bad as I thought. Okay, so let's do the basic thing, let's do naive search.

Okay, so starting with naive search. And people on Zoom can see this, people here can see it. Okay, very good. Okay, so let's start assigning. Variables.

This is the starting state. This is the state where everything is unassigned. So let's go on the fringe. Let's pick a state and let's expand it. Let's look at one of its successors.

So the successor might be something like blue. I colored one of the nodes blue. I took a variable and I assigned it a value. So now I have this on the fringe and a

bunch of other stuff on the fringe. Remember, the thing I'm not showing here is that in a search problem, there's a big fringe of unexplored answers.

But let's pick one of them off the fringe. Let's say I pick this one. I pick it up. Are you the goal? Have I assigned all variables and met all constraints?

No. This is not a complete assignment. So I have to expand this and look at one of its successors. So let's go and get one of its successors. So one of its success.

Would be coloring one of these other nodes with a color assigning another value to one of the variables. So let's do it. You ready? Blue. Uh-oh.

Okay so now we have some other stuff on the fringe. We go back on the fringe. We pick the deepest thing, which means it has the most assignments so far, most actions taken. So I picked this up. Are you the goal?

No. So I will look for one of your successors, or I'll expand you. I'll put all your successors on the fringe, and one of your successors will be blue. Okay. Blue, blue, blue because DFS goes to the deepest thing next.

Blue, blue, blue. Okay, now I pick this up and I check, are you the goal? No you're not the goal because these. Lines, they represent the constraints, and this is clearly not a valid map coloring because a lot of these adjacent nodes connected by lines have the same color. So what do I do?

I go back on the queue. So I go back on the queue. This is another state that was on the queue. Can I check this one? Are you the goal?

No. Okay. Are you the goal? No. Is all go back check this one.

Are you the goal? No. Go over this. Okay. So we could do this.

Okay. In the interest of doing anything else today, I'm gonna stop this. But if you look at this, it's kind of ridiculous. Like why would you do this? Why would you do this when you know that you were already busted right here.

So this is the problem with Naive Search. Naive Search will work. I run this for like ten years. It will get to the solution, still not there yet, not even close, right? But this will eventually get to the solution.

But the problem with why this isn't working is because Naive Search isn't looking inside the states, it's treating the states as a black box. It doesn't know that there are values and variables and assignments and constraints, but by decomposing the problem into these smaller pieces, like, there's a constraint and there's a variable and there's a value. I can get more intelligent solutions than whatever this is doing faster. Okay, still, no good. Okay, so that didn't work, so that's naive search.

It doesn't work because it doesn't think about the problem at all, it just takes your successor function. And your state space and just runs it. Let's do better. It's a good start. We'll solve the problem, but let's do better.

Okay. So to do better, we'll introduce something called backtracking search. Now honestly, I don't really like this name. I don't know who came up with it. I think it's a little bit of a misnomer, but that's what they called it.

So that's what I have to call it. Oh well. Okay. But this is a better algorithm. So we're gonna make two improvements.

So we saw DFS earlier is terrible. Even though it technically works. Let's make two very big improvements. So the first big improvement is I will actually fix an ordering. So I will say, I'm going to assign this node first, and then this node first, and then this node first.

This reduces the fan out of my. Search Tree. Because if you think back to the original naive version, the successors might be. I'm gonna color Western Australia, or I'm gonna color Queensland, or I'm gonna color Tasmania. Here, I'm gonna say, you have to color w a first, then you have to color NT first, and so on, and so forth.

And this is fine. Because, remember, we don't care how you get to the answer. Remember the Detective Gem, the Detective Robot and the Gem from earlier. We don't care how you get to the answer, we just care that you get there. So I can fix an ordering and that's fine.

The order of how I get to the answer doesn't matter. That'll save me some time. Because I don't have to consider twelve different ways to get to the same answer, I can just consider one way to get to the answer. And the second, big improvement, and this is where we're going to start opening up the problem. In a way that search

could not is, we're going to actually check the constraints as you go.

So instead of waiting until the end, where everything is blue and say, are you a goal? Oh no, you're not a goal, we're gonna check as we go. And so now, unlike the sort of dummy algorithm you saw from earlier, you can actually check the constraints as you go and check, because if you see a constraint that's been failed, you know that beneath that search tree, all of its children are not going to be valid paths to goal. And this is useful because in a c s p, once you violate a constraint, you cannot unviolate it, there's no way to go back. Once you color two nodes blue in your map, if they're next to each other, there's no way to fix that.

You've already colored them blue. So you know that everything underneath that tree is doomed to failure, and you can fail earlier and go explore other parts of your tree. So what we've done here is it's still DFS. We didn't change that at all. Still a successor function, still a goal test.

But the difference is, instead of the goal test being this judge, the black box that either says yes or no, the Goal Test has now been decomposed into a bunch of little rules. And you can check through all like Rule one's good, Rule two's good oh, Rule three is not good and I will never fix rule three underneath this tree. So I can stop searching this tree and go search other parts of my big search tree. So it's kinda like a goal test, but you're checking the goal test little by little instead of waiting until the very end. And this will make a big improvement.

So let's go watch it. So it's not the best name, even the slides agree with me, it's not the best name. So depth first search, if you add these two ideas, become something called backtracking search. Let's see if it does any better. So no more naive.

Let's do backtracking search. Okay. I do not want any of these fancy things. So we'll start with a totally blank grid. Here we go.

Okay blue. So far so good. This is still depth first search. We're still building a little fringe. Okay what's coming next?

Red. Great. Why did we pick red here? Because if we had picked blue, that would have caused a constraint violation, and once you violate the constraint, you can't go back. So Backtracking Search will notice.

That entire search tree is doomed to failure, and unlike Naive Search, which would have checked the whole tree because it didn't know we can now fail earlier, and

say this sub tree, it sucks, let's go check this sub tree where the second node is red. Okay, what's next blue? Okay, what's next red green? Why is this green? Because we checked blue here, that doesn't work.

The entire sub tree with blue here is doomed to failure. Because of this constraint here, the entire sub tree with red here is also doomed to failure. So I'm only going to check the sub tree with green. Okay, now, keep in mind, this is still backtracking search, so we're not going to get an answer on the first try, and, in fact, right. Once I get here, there's actually no valid successor, but there's no color you can put here.

So you're still going to have to go back and pick other things on the fringe. So this is not a perfect algorithm, right? You're still gonna have to go back and try things on the fringe, just like Depth First Search, but with some struggling we get to the problem. So this is better. We solved the problem and we did it by adding those two improvements.

Pick an ordering like Go in this order and fail as soon as you see a constraint that's been violated. That's called backtracking search, and it's the core of all the algorithms we're about to see. We'll just make improvements to this for the rest of today. Yeah. Mm-hmm, yeah, that was a great question.

What is on the fringe? At any given time? The answer is exactly the same as naive search, the only difference is that some sub trees don't get explored. If I had a pencil, I would show it to you and draw it. But basically, the idea is still that you're gonna have a big search tree.

We'll try, okay, so you're still gonna have a big search tree, the only. And you're still gonna be expanding stuff. Just like Depth First Search does, and as you do, you're still gonna have a big fringe, right? Like, I don't know what's underneath here or here Changed at all, which is why I don't really like this name. The only difference is that if you see that one of these states, like this one, has a constraint violation, because it has two blues next to each other.

You know, this entire search tree is doomed to failure, so you don't have to check it. But the fringe idea is still there. Does that help a little bit with my crappy drawing? Very good. The drawings will become less crappy.

I hope. Okay. Cool. So that's backtracking search. So we're doing a little bit better.

If you think back to the N Queens problem, we can now solve, N equals 25. We can do a 25 by 25 grid just using this naive approach. But I'm gonna do even better by introducing some more improvements unless you have questions. Okay, let me show you an example. Oh, I do have this slide.

Okay, hopefully this slide helps you a little bit by showing you that. It's still DFS with a fringe, even though the demo, I think, doesn't make it super clear. So here's your starting state. None of the variables have values. So what do you do?

You pick it up and you ask, are you the goal? No, you're not the goal. So we're gonna expand you, and the way we're gonna expand you is we're gonna look at your three successor states and your successor states. Are this one's red, this one's green or this one's blue? Okay, good, what do I do now?

I pick one of these up, they're all the same depth, so I can pick any of them. Depth first Search says, pick the deepest one, but here they're all depth one. So I'll pick one of you up and I'll say, are you the goal? No, and this is where backtracking search differs from naive search. Because if this was naive search, there would be three children and the third child here would be the one where there's like a red and then there's another red.

But backtracking, Search realizes, oh, wait a minute, this state is already doomed, so even though there is a gigantic sub tree here, we do not have to consider it at all, and we only have to, in queue, two children, but there's still a fringe. So yes, I knocked out a big sub tree, but there's still a fringe. I still need to know what's underneath here or here, or here or here. So the fringe is still there. Okay, need to pick one of these up depth first search, so I need to pick this one or this one, right?

These are the deepest ones on the fringe so far, so I pick one of them up, and then I would expand it and so on, and so forth. And this one also only has two children, because if I color this one green, like this little state here, if I color it green, that sub tree is also doomed to failure, so I can skip it. So backtracking search is basically like depth first search. There's still a fringe, there's still a success or function, still a goal test. The only difference is, as soon as you see a constraint violation, you can knock out entire sub trees.

And this solves the problem of the blue, blue, blue, blue, blue. From earlier that we saw that was terrible and slow. Okay, cool, here's pseudo code, don't read it, but you can read it on your own time, but you can see that there's already some choice points, then we'll see these choice points come up a little bit later as well. You can pick. Which variable to assign next?

And you can also pick which value to assign next, a little bit of preview for you, those are things we can choose later to optimize our algorithm even further. But for now don't stare at it. Yeah question. Previous slide. Yeah, that's a good question, like, who in the code recognizes that one of these sub trees is doomed and that you can totally skip it.

I think it kind of depends on your implementation. Here what they've done is they've actually implemented it recursively. So what they did is just if there's a constraint violation. Just don't do the recursive call. But you can do it some other way.

You can have a fringe and say I'll delete things off the fringe if they violate constraints. Your pick. Or everyone implemented. So we're basically good. We have an algorithm that works.

It solved the little problem. But what we'll do now and for the rest of today and also next time, is we will come up with three separate improvements to make backtracking even better. So one of them is ordering. What order do you assign the variables and the values? Like, do you pick red first or do you pick green first?

And does it make a difference? Or blue? And then there's filtering which is the one we'll see today. Is there any way to detect failure a little bit earlier? And then the last one that we'll see next time is structure.

Can you somehow exploit the problem structure? And do something clever. All of these are based on backtracking search. The only difference is we're doing some improvement on top of it to hopefully make the algorithms a little bit faster. But you already have the core algorithm.

Now we will make it faster in three different ways. Let's start with filtering. Okay. So there's different types of filtering, but all of them have to do with keeping track of what values are still valid at a variable. So already if you look at this picture, you notice that things have changed a little bit.

So now, instead of each of these boxes either being unassigned or assigned, right? This is either blank or red or green or blue. What we're now going to try and do is try and detect. Failure early, is there some way we can know that something is doomed without having to explore too much of the tree. So we are.

Can we somehow look a little bit into the future and say, you know what this assignment is already do? There are no valid successors that will give me an answer, so

let me just stop right now, instead of trying a bunch of different things. And the way you're going to deduce that things are doomed to failure. A little bit earlier is you're going to keep track of what domains are still valid for each variable. So now, instead of each of these boxes being assigned or unassigned, like a yes, no.

Now this little cloud, there's a little cloud that says, Well, right now, before anything has been assigned. Western Australia could be red, or It can be green, or it can be blue, and NT could be red or green or blue. And in fact, right now, since nothing is assigned, each of these can be red or green or blue. But what you'll do as you start assigning is you actually check the constraints as you go. And you'll cross out values that do not work anymore based on the current assignment, and as you cross out values.

If you ever notice that one of these domains has all of one of these variables, has all of its values crossed out. You know that there's no possible successor, and you can quit early and go explore some other part of the tree. So this is still backtracking, search, still a tree, still DFS, but you're hopefully going to detect failure early, so you know, oh oh, this is doomed, let me go explore some other part of the tree. A little bit better. Okay, so let's do forward checking together.

So here we go. It's still backtracking search. I pick up this starting state. Are you the goal? No.

So I will assign one of your successors. So you see here, I've assigned one of the successors. I've assigned. I think this is Western Australia to red. So now I can actually go here and see, once I've assigned Western Australia red, based on the constraints, that I can see my list of rules.

What's changed? What values are no longer valid? Let's do some crossing out. Okay. I have a hack.

Hopefully I can find it. Color is white. I have a magic eraser. Here we go. Okay.

So Western Australia. Well, it's red. So green. Is no longer valid. Okay.

It's gone. And blue is no longer valid. Okay. Also gone. So I'm crossing things out that are not consistent, ah, with the current assignment.

Okay good. That one wasn't too bad, because I just assigned w a, so red is the only thing left. Now let's do some more interesting ones. Take a look at NT. So NT is

this one right here.

Originally, I said, red and green and blue are all valid colors. What about after this assignment? Are red, green and blue all still valid colors? No. Which one is no longer valid?

Red right, because I can no longer color NT red, because if I color NT red, then well. It's gonna conflict. There's gonna be a constraint violation. Okay. The downside of making your magic pen or your magic eraser white, is that you can't find it anymore.

Okay. Did the slides just let me advance? What do I do? Okay what do I where are you pen? Sorry.

Okay this was this was very much me being. I got too bold. I was like I wanna use white to be clever. I lost my pen. Okay.

I don't know what to do anymore. What if I just. Nope that doesn't help. I'm sorry, okay, but the point being what I wanted to do? Ah, okay, what I wanted to do was erase White from NT, which I'm utterly failing to do.

Okay my mouse is back thank you. Okay, good, the slide does it for me, so I don't have to be clever with it. Okay, so ah, go away, okay, so what you do is, ah, Western Australia is definitely red, but notice that NT loses its red because red is no longer a valid assignment here. And notice that the s a South Australia also loses its red, because red is no longer. So now as I'm assigning for this state, I'm also keeping track of a little cloud that says, You know what?

These are the possible valid successor states. These are the possible valid assignments to the values. Why is this useful? Let's assign some more to see why it's useful. Let's say so I picked this up.

Are you the goal? You're not the goal. So let's go ahead and assign Queensland to green. So what happens now? Well, I assign Queensland green very good.

But now I can cross things off. So for example, NT loses its green because coloring NT green would not be consistent with my new assignment, which was Queensland equals Green. And also, s a can no longer be green, because if s a is green. What conflict? And New South Wales?

You can also not be green anymore because that conflicts, so as I'm going, I'm keeping track of what's still possible. Let's do one more assignment, so let's take, let's pick this up. Are you the goal? You're not the goal? So I'm gonna color Victoria equals blue, but as soon as I color, Victoria equals blue, look, what happens?

I have to cross off blue from s a, and I also have to cross off blue from NSW, the two neighbors. But now I can stop right here. If this was naive backtracking search, what would naive backtracking search do? Naive backtracking search would continue? It would try assigning all the colors before realizing that something has gone wrong.

What about forward checking? Forward checking is smarter. Why do you know in Immediately that, like this state is doomed and none of its successors will work. You know, this is doomed because South Australia has no valid values anymore. So by crossing things off as you go, you can fail early.

You can say, you know what, even though this kinda looks promising, like, maybe there's a solution. I know, for a fact, there isn't one because I did forward checking. I filtered as I went and I detected failure earlier. If I did not do this backtracking search, would have gone all the way to the end. Assign all the colors to realize Oops nothing works.

But here I can detect failure early and say, nope, this state is already in trouble. You should go back and pick something else off the fringe. And something else might be going back here and saying, Well, this could have been blue. This could have been blue or red. So you have to go back and pick something else.

That's forward checking. As you go cross off values that violate constraints. Pretty cool. Now you can detect failure early. Okay.

Great. So this is pretty good. Turns out you can do even better. So take a look at this intermediate state right here. This is what we saw earlier right.

We assigned w a, equals red, and then we assigned Queensland equals green. Now stop right here. Does anybody have a bad feeling about the state or like, something's gonna go wrong? I have a bad feeling about the state. Why do I have a bad feeling?

Look at like NT and s a. N t has to be blue. In this configuration and Sa also has to be blue. I have a bad feeling about this. You know who doesn't have a bad feeling

about this?

Forward checking Forward checking does not care, Forward checking does not look this far ahead. It only thinks about information from an assigned variable to an unassigned variable. So, for example, forward checking would say, oh, you assign Q equals green. Go ahead and knock green off NT SA and NS W. It only did that because you assign Q equals green.

It does not look at the unassigned variables. So it just says, you assign Q, go cross some things off. You assign w a go cross things off that have to that are inconsistent with w a equals red. But it does not. Care that there are looming constraint failures that are further ahead in the future.

So forward checking is pretty smart, helps us detect failure early, but it's not totally smart. And in this particular case, it cannot find a failure between two unassigned variables. It can only find constraint failures between an assigned variable, like Q equals green and its neighbors or its other variables. Doesn't have to be neighbors, just neighbors here because it's a map. So in this case, even though we, as humans, looked a little bit into the future and said, Nah, this is doomed, forward checking does not look that far into the future.

It only checks constraint violations with assigned variables. WA Is red. How does that affect my domains? Q is green. How does that affect my domains?

I don't care that these two are blue or that blue is the only thing left. I haven't assigned them yet. They're not colored in yet. So I don't care. I only care about constraint violations that occur with assigned variables.

So forward checking is nice. It's simple but it's limiting. It doesn't look that far ahead into the future. So what if you want to look further ahead into the future? We just have to make it more powerful.

That's what we'll do next. But that's it for forward checking. Thoughts on that before I show you arc consistency, which is the even smarter version of forward checking. I have a demo. Let me show you a demo.

Okay. Let's go back here. Let's go back to our. Our favorite graph. I'm gonna reset you.

But now, instead of just doing naive backtracking search where everything is unassigned or assigned, I'm gonna do forward checking. There's a little option here for forward checking. Click Okay. Now we have Forward checking. So now you see instead of values being unassigned or assigned, everyone now has a little cloud that says I could be blue or red or green.

So here we go. We're still gonna do DFS. We're still gonna assign. So I pick you up and I assign. Oh.

Reset. Okay. So I wanna assign you to blue. But in addition to assigning this node to blue, look what happened. This one, this one and this one, they lost the blue because assigning any of these blue would cause a cause, a constraint.

Violation with my newly assigned variable. So I cross them off. They cannot be blue anymore. Did I solve the problem? Nope.

I need to keep going. So why can't I keep going? The button just like grayed out for some reason. Okay. Let's see backtracking forward checking good.

Okay so we assigned this blue. They lost their blueness. So I click again. Okay I assigned this red. Now what happens?

This one, this one, this one, they all lose their redness, right? The red is gone. They're no long, red is no longer a valid value here because they would cause a constraint violation with the red that I have assigned here. Next, assign this one. What about, will I assign it?

You're blue. Actually, no, I guess I'll assign. Why did I assign that? Whatever. I'll assign this one green.

That's fine. And when I assign this one green, this one and this one, they lose green because, and I guess this one and this one as well. All of its neighbors lose green, because green is not a consistent assignment with the variable that I have just assigned. And so we keep going. Okay.

That's red. That's blue. It's green. Yeah. Why did it go in this order?

That's bizarre. I thought I was supposed to go bottom up. Try again. There we go. No.

I don't know what's going on. Whatever. Okay. And so you do some backtracking but eventually you get. That's that's a forward checking you cross out values as you go, this needs to be none.

Let me try again. Okay that's better. I don't know why I did, I don't know why I did that. Okay whatever sorry for bug. Okay, I'm gonna pretend I'm assigning the variables from bottom up, like there's a fixed order.

So send you Blue cross off some blues, send you red, I cross off some reds, send you Blue cross off some blues. Stop right here. Does anybody have a bad feeling about this? I have a bad feeling about this. You know, who doesn't have a bad feeling about this?

Forward Checking doesn't have a bad feeling, because Forward Checking only thinks about constraint violations with already assigned variables. You and I, as humans, we can see oh, problem here. You have to be green, you have to be green, you cannot both be green. But Forward checking doesn't think about that. Forward checking only thinks about relations to the immediately assigned variables like these three down here.

So if you're doing forward checking like this, you actually have to keep going until here and then as soon as you assign this one green, you're like uh-oh, that one can't be green either. I need to backtrack. I need to go back on my fringe, pick something else. And so some backtracking is required. But eventually I work through the graph.

You see there's another backtrack here. It's like Oops, that doesn't work because this would empty out this domain. So I go back, pick that instead. Okay, now I'm good. So it does solve the problem.

But takes a little bit of backtracking and it doesn't detect failure. Particular? This is the critical moment where we could have done better, but we didn't because forward checking wasn't strong enough. Any thoughts before? I make it stronger using something called Arc consistency.

Time to make it stronger. It's like forward checking but even stronger. So here we go. Okay, so before I can tell you about arc consistency in total, let me first tell you

about a single arc. What is an arc?

An arc is an arrow pointing from one domain to another. It is directed. See how there's an arrow. So there's actually one arc from NT to w a, and there's another arc from w a to NT. So arcs are actually bi-directional.

This is something new that you haven't seen. And each arc can be consistent or inconsistent. So it has a property called consistency, totally different from anything you've seen before this. Same name, but it's either consistent or it's not consistent. So it's like a yes no answer.

And the way that you check whether or not an arc is consistent is you follow this rule. You say for every possible value in the tail, which is the part without the pointy arrow, for each of these values red and yellow, sorry, green and blue, is there some at least one valid assignment in the head, which is the other side of the arrow, that can be assigned without a constraint violation. That's the rule. It'll probably make More sense if I show you some examples. So let's do some consistency checking.

Remember each arc is either consistent or not consistent. So is a yes or no answer. So let's check. This arc is a consistent. If I assign blue to NT, is there something in w a the head that is consistent?

Yeah. I send this blue. This can be red. If I assign NT equals green, is there something consistent in the head? Yeah.

W a equals red. If I assign NT equals red, is there something in w a that works? No. W a is red. And so this arc is inconsistent because I cannot assign NT equals red.

This value in the tail has no corresponding. Valid value in the head. So this arc is inconsistent. The answer is no, it's not consistent. How do you make it consistent?

And the answer for how you make it consistent is you can delete values from the tail. So in this case, because blue worked from the tail, and green also worked from the tail, but red didn't work. Okay, no more no more being bold with the colors. What I can do is I can delete red from the tail so that it only has two values left, and now the arc is consistent. So it used to not be consistent, but by deleting values from the tail, I can make the arc consistent.

Turns out forward checking is basically this forward checking. Is checking arcs, and in particular, you check all the arcs, pointing at the thing you just assigned. So forward checking is like a special case of this, but we're gonna make it even more general because we're gonna check arcs not just between assigned variables, but also between unassigned ones. And my mouse is gone again. Okay.

No, ah, if I click this at my back, ah, mouse, mouse, mouse, okay, good. So delete things from the tail, that's how you get arcs to be consistent. Okay. So let's do some arc consistency together. Here we go.

So we're gonna check arcs. In particular, we're gonna think about whether each arc is consistent or inconsistent, and if an arc is inconsistent, you will delete from the tail in order to make it consistent. And arc consistency is a property of a specific arc, and you can say that an entire CSP is arc consistent if all of its arcs are consistent. So each arc itself has a yes no answer, and then the whole CSP has a yes no answer, and the whole CSP is consistent if all of its arcs are consistent. Okay, that'll make more sense once I do a demo.

So here we go. Let's check. So here is an arc, and this is the first arc you've seen, that's a little bit different because it points between two unassigned variables. So unlike this arc where w a was already assigned. Here NSW is actually not assigned yet.

So unlike forward checking, we are now doing checking between two unassigned variables, but we can still do it, so let's do it. For every value in the tail, there must be something in the head that works. If I assign red, is there something in the head that works? Yeah blue works. You might say, oh, wait a minute, red doesn't work, that's fine.

You just need at least one thing that works. So if I assign red in the tail, blue is a valid extension in the head. So red is still okay, because there's at least one thing, doesn't have to be everything, just at least one thing that works, and here it's blue. If I assign green for V, is there something in the head that works? Yeah either red or blue they both work.

That's fine. If I say blue to V, is there something in the head that works? Yeah red works. They may say, oh, but blue doesn't work, that's fine, we don't care. We just want at least one thing that works.

So this arc is consistent because everything in the tail has at least one valid extension in the head. I pick red there's something that works. I pick green there's something that works. I pick blue there's something that works. So this arc is consistent.

The answer is yes. Let's check one more. What about this arc? Okay, well, this arc points from s a to NSW. Remember, check the tail and see if this a valid extension at the other end.

Okay, in the tail, there's only one possible value here, it's blue. If I pick blue. SA Is there at least one valid assignment to NSW Yep it's red. At least one is all you need. So this arc is also consistent.

For every value in the tail, there is at least one thing in the head that works. Let's keep going. Okay let's check in the other direction. So this is another point, right? Arcs can be, arcs are directional, so there's an arc from SA To NSW There's also an arc from NSW To SA And there's just two different things.

Let's check this one. Okay if I assign red to NSW In the tail is there something in the head that works? Yeah I can I can just blue all good. If I choose blue for NSW Is there something in the head that. No.

If I pick blue for NSW, I'm toast, because the only thing that worked, or the only thing that's left in s a is blue, and I cannot assign them both blue. So this arc is not consistent. The thing that is making it not consistent is this blue right here, because if I assign NSW blue, there's no valid extension. So what do I do? How do I make this consistent?

I delete blue. You're gone. See how it used to be there and now it's gone. So the way that you make arcs consistent is you delete from the tail. You go to the tail you say nope you're gone.

You cannot be a valid, you're no longer a valid candidate, because if I pick blue here, I know for a fact that there's no valid successor at s a. Let's keep going. Okay let's check this one. Does this arc look familiar? This arc looks familiar.

Why is this arc familiar? Because it was the very first one that we checked. Remember the first time we checked this arc it was consistent. But now let's recheck this arc. Is it still consistent?

Let's check. If I pick blue at V, is there something valid at NS w? There sure is it's red. If I pick green at V, is there something valid at NS w? Sure is it's red.

If I pick red at V, is there something valid at NS w? No. So this arc is no longer consistent. It's kind of interesting. Arcs that were once consistent can actually become inconsistent.

Why did this arc. Become inconsistent because when we first looked at it, NSW, had both red and blue, but in the process of checking some other arc, we deleted blue from NSW. So now that blue is gone from NSW, this arc is no longer consistent, because this blue was like supporting Red's okayness. The reason why red was okay here in the tail was because there was a blue here to save it, but now there's no more blue to save the red in V. So we actually have to make this arc consistent, and the way to make it consistent is by deleting this red.

So it turns out arcs can become inconsistent and as you go and check the consistency of an entire CSP. By checking this arc and then this arc and this arc, you actually have to recheck arcs as you go. So whenever you change NSW, you need to check all the arcs pointing at NSW. You need to check that one and that one and that one and that one, because something might have been consistent because of NS W's blue, but now the NS W's blue is gone, those arcs might become inconsistent, and you need to go and make them consistent again. So how do you make this in consistent?

Well, this red has no valid extension, you're gone, deleted. And you keep this going on and on and on. I haven't presented an algorithm yet, but I'm just telling you that you check all these arcs over and over again. Let's check one more. So let's check the s a to NT arc is this arc consistent for everything in the tail?

Is there an extension in the head? If I pick blue here, is there a valid assignment here? No, the only one left is blue, and these two cannot be the same color. They're neighboring territories, so this arc is not consistent. How do I make it consistent?

I delete from the tail, I delete blue because you have no valid extension. Goodbye, gone. And look, what happened? Just by checking a bunch of arcs over and over again, we were able to detect failure extremely early. Like, look at this graph, you and I.

As humans, we can barely tell that this is doomed to failure. You're like, this is fine so far, you're red and you're green, but Arc Consistency. Looks really far into the future because it checks all pairs of variables, even unassigned ones. So actually, our consistency already detected that this assignment is doomed to failure, even though you and I, as humans, can barely see that, that's true. That's pretty cool.

So Forward Checking had to run a few more steps before realizing that something was doomed. Our consistency is more powerful because it checks arcs between unassigned variables. You can actually detect, for this particular example, just based on these two assignments, that things are totally screwed and there's no more solutions. You should backtrack and go pick something else off the fringe, because the entire sub tree with $w = a$, $R = Q$ and $Q = \text{green}$, guaranteed to have no solution, our consistency proved to be so.

Pretty cool. Also kind of tricky. So something you'll have to play with in your homework to get more familiar with. Okay. So delete from the tail.

That's the only one thing you take away today. Delete from the tail. So when an arc is inconsistent, delete things from the tail to make it consistent. Questions? Yeah.

What's up? You're good. Yeah. I guess you could. So if you're a human, you could look far ahead using your brain and be like, well, you have three neighbors and you're screwed.

Yeah. I guess you're right. But arc is. Consistency can actually see that where forward checking could not. Yeah, I guess you're right, so as humans, you can maybe be smart and see it.

Okay, cool is our good, different way of saying fringe? That's a great question. The answer is no, because our consistency is a property of a c s p. You are either are consistent or you're not, and to make it consistent, you delete things. That's totally separate from the backtracking search that you still have to do, so our consistency and forward checking are things you do as you backtrack search to detect failure early.

That's a great question, though. Totally different from the fringe, it's something you run within. The search to try and make your search faster by detecting failure earlier, like in this case. Okay, I guess in this case, just seeing three neighbors doesn't show that you're totally screwed. You have to think a little bit further ahead and

be like, Well, there's three neighbors, so one of them has to be this.

It's a lot it's a lot of thinking. Okay and our consistency does the thinking for you. Okay, I haven't actually shown you how to make this an algorithm yet, so here's your actual Our Consistency algorithm. So far it's a property right here. I'm just telling you, it's either consistent or inconsistent, and if it's inconsistent, here's how you fix it you delete from the tail.

Here's the actual algorithm. There's a lot of algorithms out there, this one's called a c three. Here's how it works. So a c three has a queue. This is a queue inside of a queue, because you're doing backtracking search still, but while you're doing backtracking search, you enforce arc consistency with a queue.

So it's like a queue inside of a queue. It's like an exception. So what do you do? You keep a queue, and the queue is the list of arcs you have not checked yet. And you check the arcs one by one.

And each time you check the arcs, you delete from the tail. So if the arc is inconsistent, you make it consistent by deleting from the tail. And if it's ever the case that one of them becomes totally empty, like one of the domains becomes empty, you can detect failure early. And one more thing you have to do, the reason why this code is kind of ugly and long is because if. You delete something from the tail.

Remember this case right here. You have to recheck a bunch of arcs pointing at that variable, so if you change NSW, you need to recheck everyone pointing at NSW. Because something in NSW like the blue might have been supporting the consistency of other arcs pointing at NSW. And once you delete the blue from NSW, you must recheck all those other arcs. Because they might have something that is no longer consistent, because the consistency was relying on the blue that is now gone.

So this is an actual algorithm as a queue that processes all the arcs one by one by one and checks them to see if they're consistent, and, if anything. Is inconsistent, you remove values from the tail and then put some more arcs on the queue, and you do this over and over, and over again until the queue is empty, and once you do that, you know for sure that the CSP is arc consistent. All of this is running inside backtracking search. This is just a way to help you detect failure early as you go. So you're like, I sign this and this and this arc consistency, please look in the future and tell me if I'm in trouble, and that's what it does.

Yeah. Yeah, that's a good point, so this algorithm a c three, it starts with all the arcs on the queue. Yeah. So you generate all the arcs and then you check them one by one by one, as you check, you might have to put some arcs back. On the queue for rechecking, and then once everything is checked and rechecked, you're done.

And you know for sure, the CSP is are consistent and it's allowed you to look in the future. Because it's possible that by enforcing this, you end up in a state like this. That tells you that failure is inevitable and it's time to go look somewhere else. Yeah more questions. Oh yeah sure runtime.

Oh oh what's the runtime of forward checking? I didn't present it as an algorithm, so I don't really have a good answer. If I had to guess, I'd say it's D squared because you have to check. Is it D squared? It might be D Times.

And maybe it really catching me off guard. But I think it might be something like D Times N. Because for each of these ah n variables, you have to check D values. Check all these values against W equals red. Yeah, you got me.

As on the slides, Okay, yeah, good call though. Yeah. This one has a more complicated runtime. I'm not gonna go through in crazy detail. But there's a d square term.

Why is there d square? Because you're doing this thing right, like, where are you? You're doing this thing right? There's D values here and D values here, so you have to do this like D square thing. And then how many pairs are there?

This n squared pairs, because you have a pair, these two and these two, You have an N squared term and then the final term. The reason why it's D cubed is because you have to recheck arcs as you go. So you might have to rerun this entire process d times once for every value that you lose. That was really really rough. Ask me later if you want to see it in more detail.

Point being it's polynomial. That's sort of the catch here. At least it's not exponential. So you can do this somewhat efficiently. It's polynomial.

It's not something like two to the end that's ridiculous. So you can do this in a reasonable amount of time, and by doing it, it lets you see a little bit into the future, to see if failure is inevitable or if I should keep going. Cool. Okay. I'm almost done.

Don't go yet. So first, I'm gonna tell you that our consistency is good, but it is not perfect. You can't be like, oh, I have our consistency, I'm gonna solve all my problems using it. You still have to do backtracking search. So let me give you an example where our consistency doesn't solve all your problems.

So look at the one down here. You can run our consistency on this. In fact let's do it. Okay, let the arc from the bottom node to the top node, are you consistent? If I sign red here is there a valid extension?

Yes. If I sign blue here is there a valid extension? Yes it's red. So this arc is consistent. What about the other direction?

Also consistent. In fact every arc here is consistent. For every value in the tail, like red, there's an extension at the head, it's blue. For every value at the tail blue, there's a valid extension red. So.

Every arc here is consistent. Are we doomed? Yeah, we're doomed because there's no way to finish this assignment right. If I put red here, I can't put both Blues here and so on and so forth. So, arc consistency is good, but it will not solve all of your problems.

It just allows you to peek a little bit into the future to see if you're definitely doomed, or if you're maybe doomed, maybe not, and we should keep going. But if one of these domains is empty, then you know for sure that you're screwed. Okay, let's do our consistency on this demo and then I'll let you go. So the big graph, the big graph, right. So arc consistency is very powerful.

It can do things on this larger graph. So let's do backtracking and let's do our consistency. Okay here we go. Start assigning at the bottom. Blue, Good, and what did this do?

It just crossed off blue from two of his neighbors. Nothing too special yet, but here comes the interesting one Green or, sorry, red? Okay, now check this out. Not only once I assign red. Not only did this ah variable lose some of its values, but this one also lost some of its values, and so did this one.

So our consistency looks further into the future. It propagates the consequences of your assignment just a little bit further. Let's assign one more. Okay, this one got assigned blue and check this out. This variable, it's so far away.

And yet Our consistency was able to figure out. Oh, because you assigned this blue red is no longer valid here. It checked all the arcs, the thing that we just did. And so as you assign, you'll notice that it's looking really far into the future. And it's allowing you to lose values from domains that are really far away from your current variable.

And so as a result, as you do this, you can start detecting failure earlier. Now it's not perfect. So like, look right here, right, you start here, you send this. You run our consistency, all the domains empty out, and you're like, oh man, there's no answer. So I need to backtrack.

So backtracking is still inevitable, but with some luck, you can get through the big graph. Pretty cool. Okay, I'm out of time, so come back next time if you want to see more our consistency and other ways to brew CSP's. Yeah but we did it we solved the big one. Very cool.

I have some time up here if you have questions.