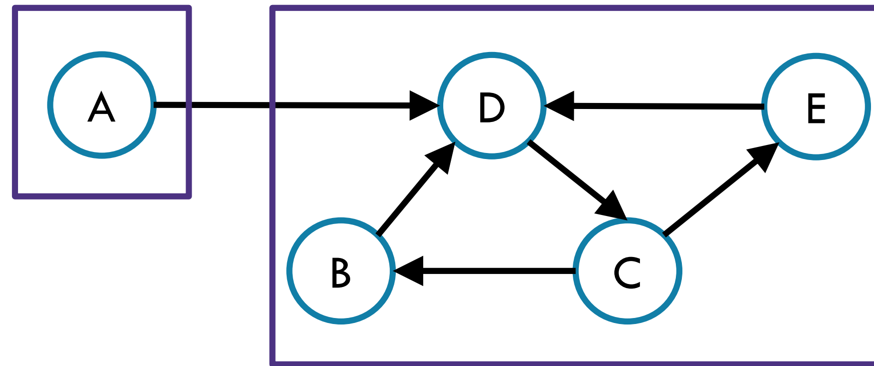
Lecture 22: SCCs and Reductions

CSE 373 – Data Structures and Algorithms

Strongly Connected Components

Strongly Connected Component

A subgraph C such that every pair of vertices in C is connected via some path **in both directions,** and there is no other vertex which is connected to every vertex of C in both directions.



Note: the direction of the edges matters!

Finding SCC Algorithm

Ok. How do we make a computer do this?

You could:

- run a BFS from every vertex
- For each vertex record what other vertices it can get to

But you can do better!

We're recomputing a bunch of information, going from back to front skips recomputation.

- Run a DFS first to do initial processing
- While running DFS, run a second DFS to find the components based on the ordering you pull from the stack
- Just two DFSs!
- (see appendix for more details)

Know two things about the algorithm:

- It is an application of depth first search
- It runs in linear time

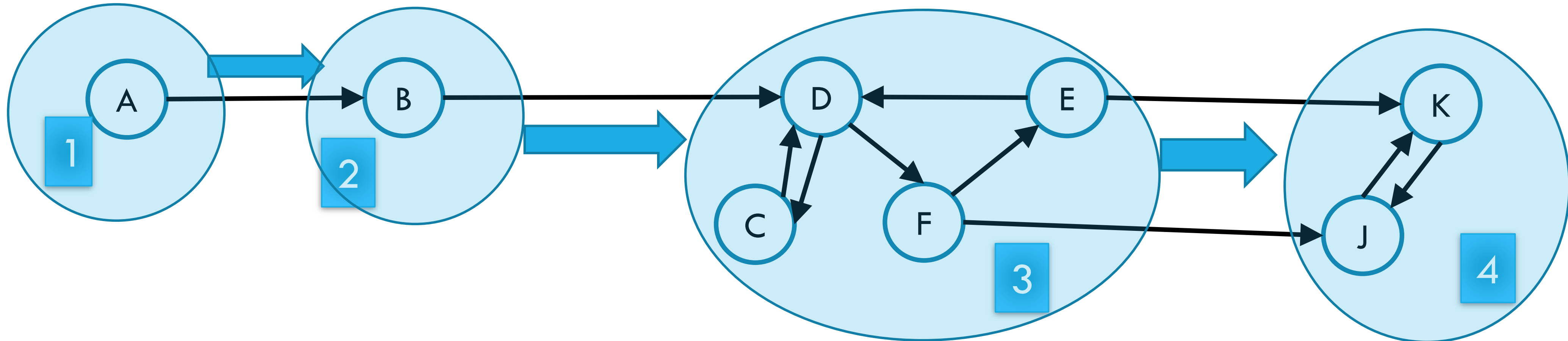
Why Find SCCs?

Graphs are useful because they encode relationships between arbitrary objects.

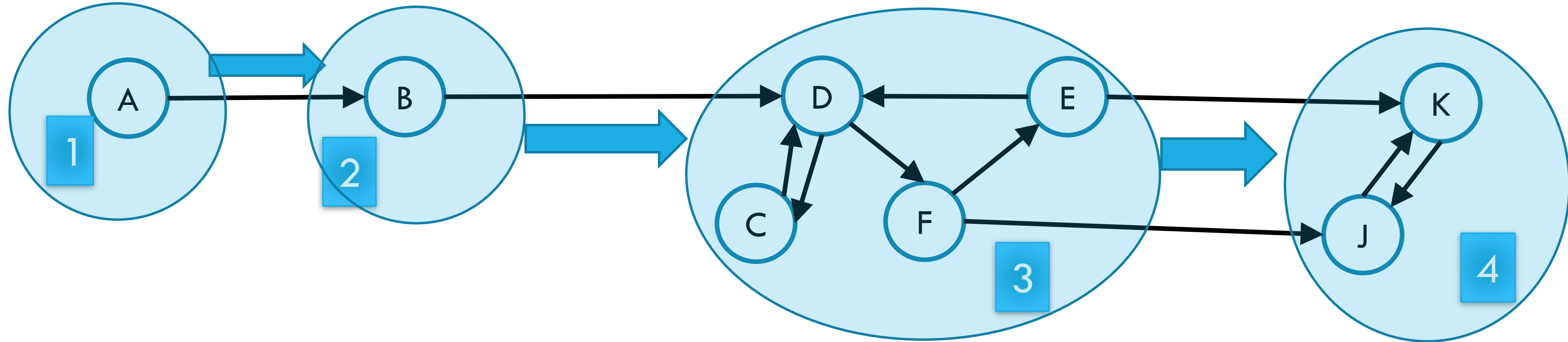
We've found the strongly connected components of G.

Let's build a new graph out of them! Call it H

- Have a vertex for each of the strongly connected components
- Add an edge from component 1 to component 2 if there is an edge from a vertex inside 1 to one inside 2.



Why Find SCCs?



That's awful meta. Why?

This new graph summarizes reachability information of the original graph.

- I can get from A (of G) in 1 to F (of G) in 3 if and only if I can get from 1 to 3 in H.

Why Must H Be a DAG?

H is always a DAG (do you see why?).

Takeaways

Finding SCCs lets you **collapse** your graph to the meta-structure.
If (and only if) your graph is a DAG, you can find a topological sort of your graph.

Both of these algorithms run in linear time.

Just about everything you could want to do with your graph will take at least as long.

You should think of these as **"almost free" preprocessing** of your graph.

- Your other graph algorithms only need to work on
  - topologically sorted graphs and
  - strongly connected graphs.

A Longer Example

The best way to really see why this is useful is to do a bunch of examples.

Take CSE 417 for that. The second best way is to see one example right now...

This problem doesn't *look like* it has anything to do with graphs

- no maps
- no roads
- no social media friendships

Nonetheless, a graph representation is the best one.

I don't expect you to remember the details of this algorithm.

I just want you to see

- graphs can show up anywhere.
- SCCs and Topological Sort are useful algorithms.

Example Problem: Final Review

We have a long list of types of problems we might want to review for the final.

- Heap insertion problem, big-O problems, finding closed forms of recurrences, graph modeling…
- What should the TAs cover in the final review – what if we asked you?

To try to make you all happy, we might ask for your preferences. Each of you gives us two preferences of the form "I [do/don't] want a [] problem on the review" *

We'll assume you'll be happy if you get at least one of your two preferences.

Review Creation Problem

Given: A list of 2 preferences per student.
Find: A set of questions so every student gets at least one of their preferences (or accurately report no such question set exists).

*This is NOT how the TAs are making the final review.

Review Creation: Take 1

We have Q kinds of questions and S students.

What if we try every possible combination of questions.

How long does this take? O($2^Q S$)

If we have a lot of questions, that's **really** slow.

Instead we're going to use a graph.

What should our vertices be?

Review Creation: Take 2

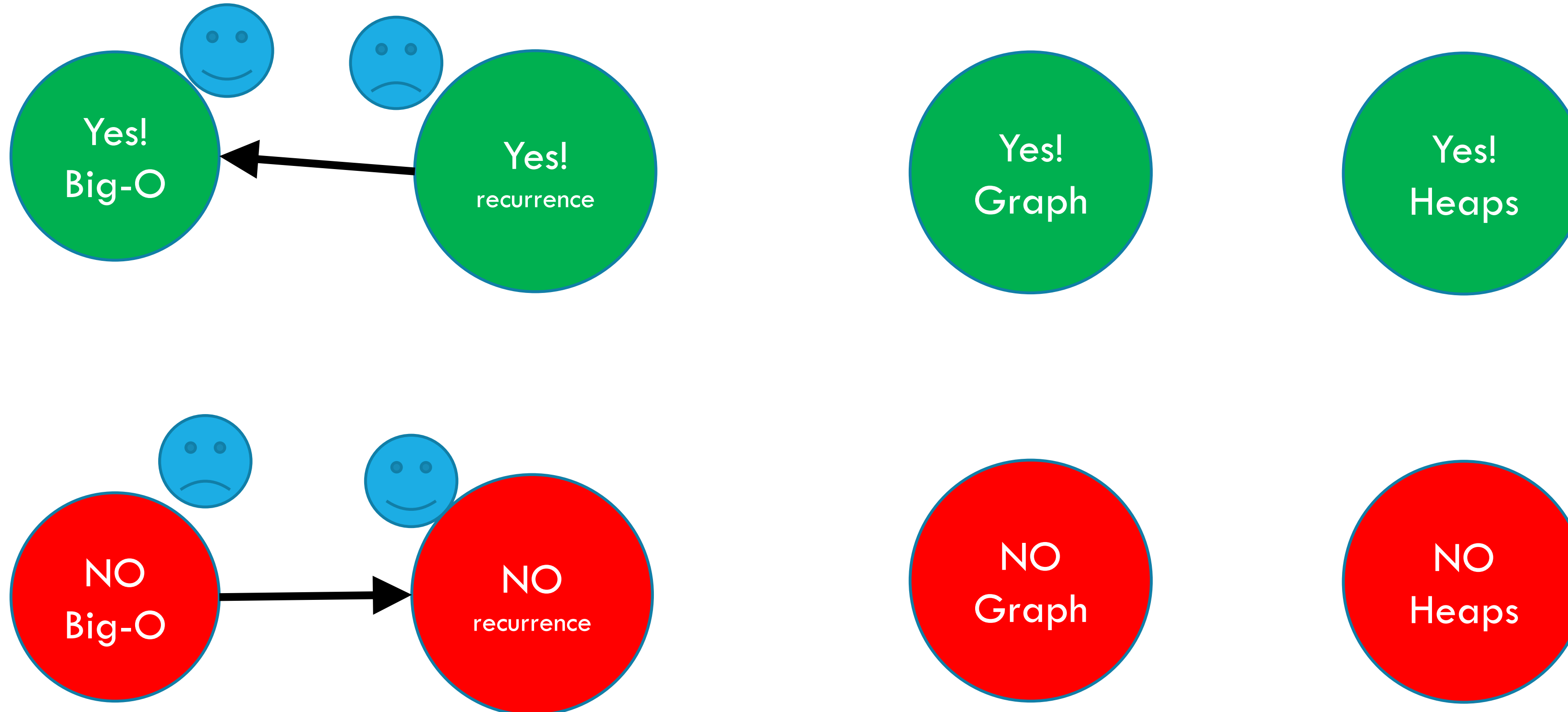
Pollev.com/cse373su19
What edges are added for the second set of preferences?

Each student introduces new relationships for data:

Let's say your preferences are represented by this table:

Problem	YES	NO
Big-O	X	
Recurrence		X
Graph		
Heaps		

Problem	YES	NO
Big-O		
Recurrence	X	
Graph	X	
Heaps		

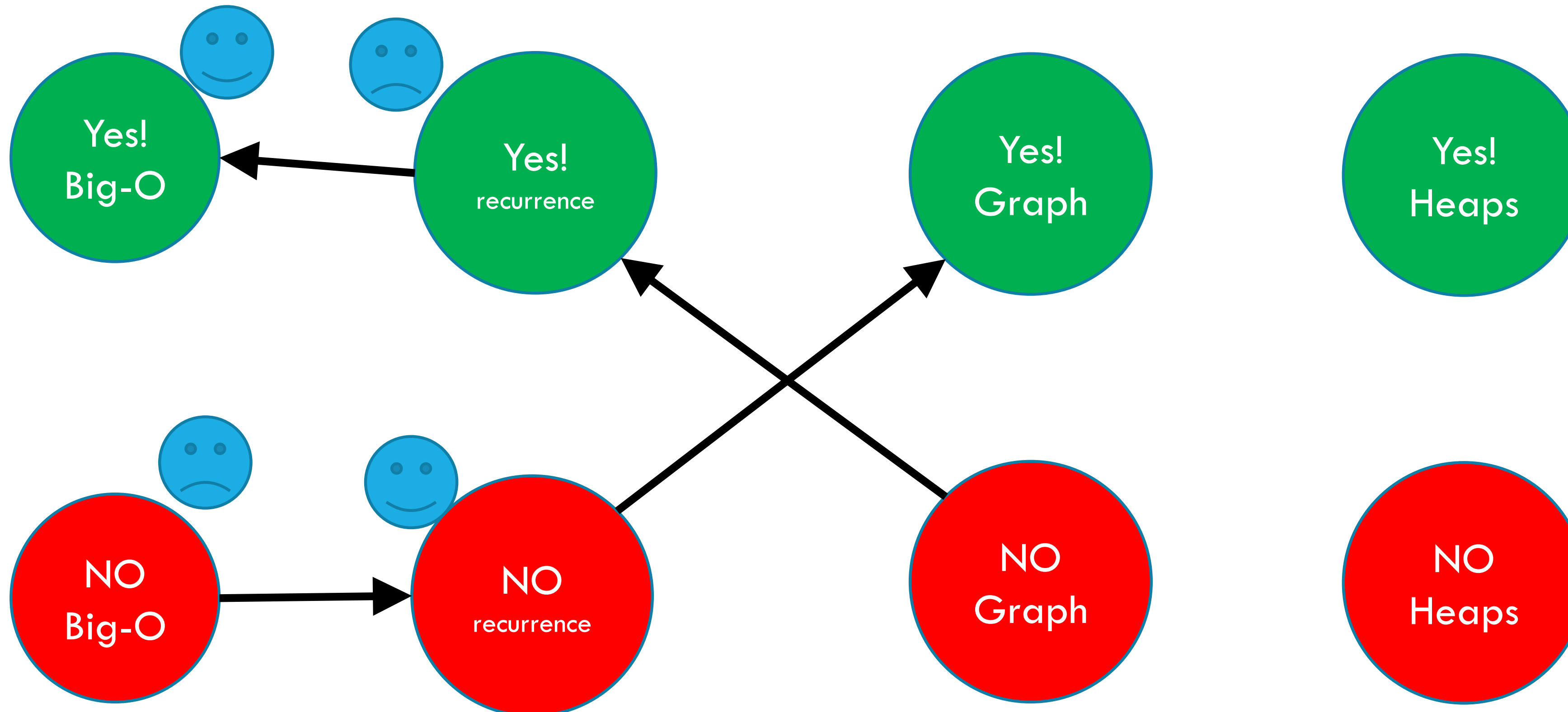


If we don't include a big-O proof, can you still be happy?
If we do include a recurrence can you still be happy?

Review Creation: Take 2

Each student introduces new relationships for data:

Let's say your preferences are represented by this table:



Problem	YES	NO
Big-O	X	
Recurrence		X
Graph		
Heaps		

Problem	YES	NO
Big-O		
Recurrence	X	
Graph	X	
Heaps		

If we don't include a big-O proof, can you still be happy?

If we do include a recurrence can you still be happy?

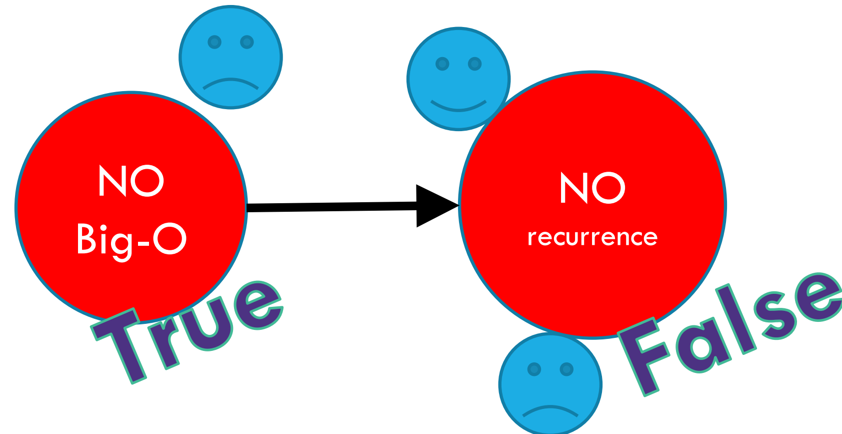
Review Creation: Take 2

Hey we made a graph!

What do the edges mean?

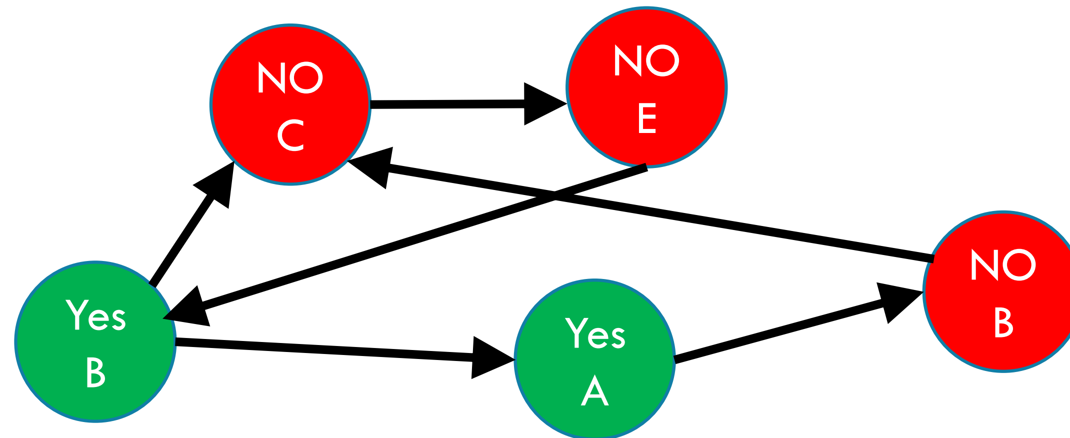
Each edge goes from something making someone unhappy, to the only thing that could make them happy.

- We need to avoid an edge that goes TRUE THING → FALSE THING



We need to avoid an edge that goes TRUE THING → FALSE THING

Let's think about a single SCC of the graph.



Can we have a true and false statement in the same SCC?

What happens now that Yes B and NO B are in the same SCC?

Final Creation: SCCs

The vertices of a SCC must either be all true or all false.

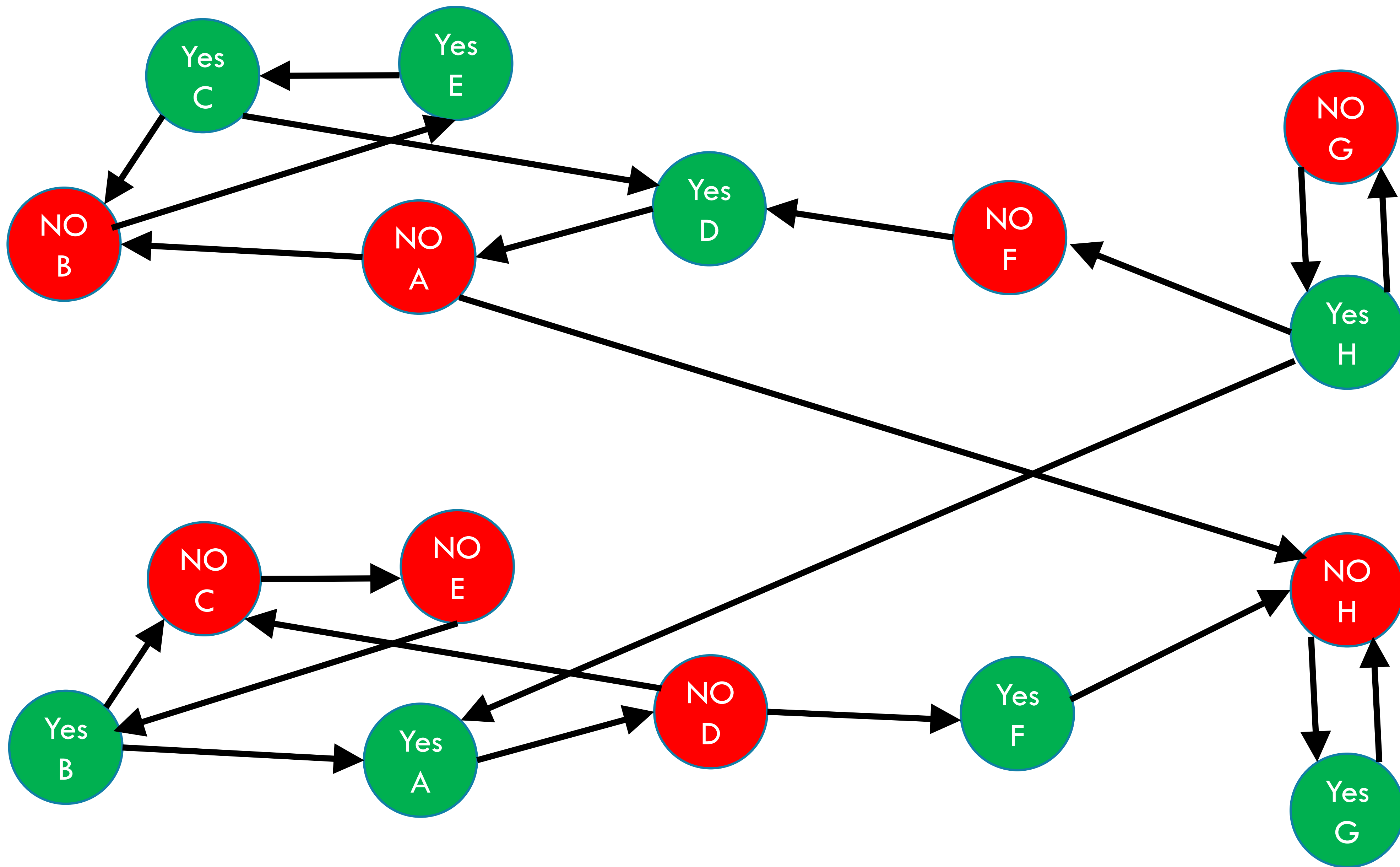
Algorithm Step 1: Run SCC on the graph. Check that each question-type-pair are in different SCC.

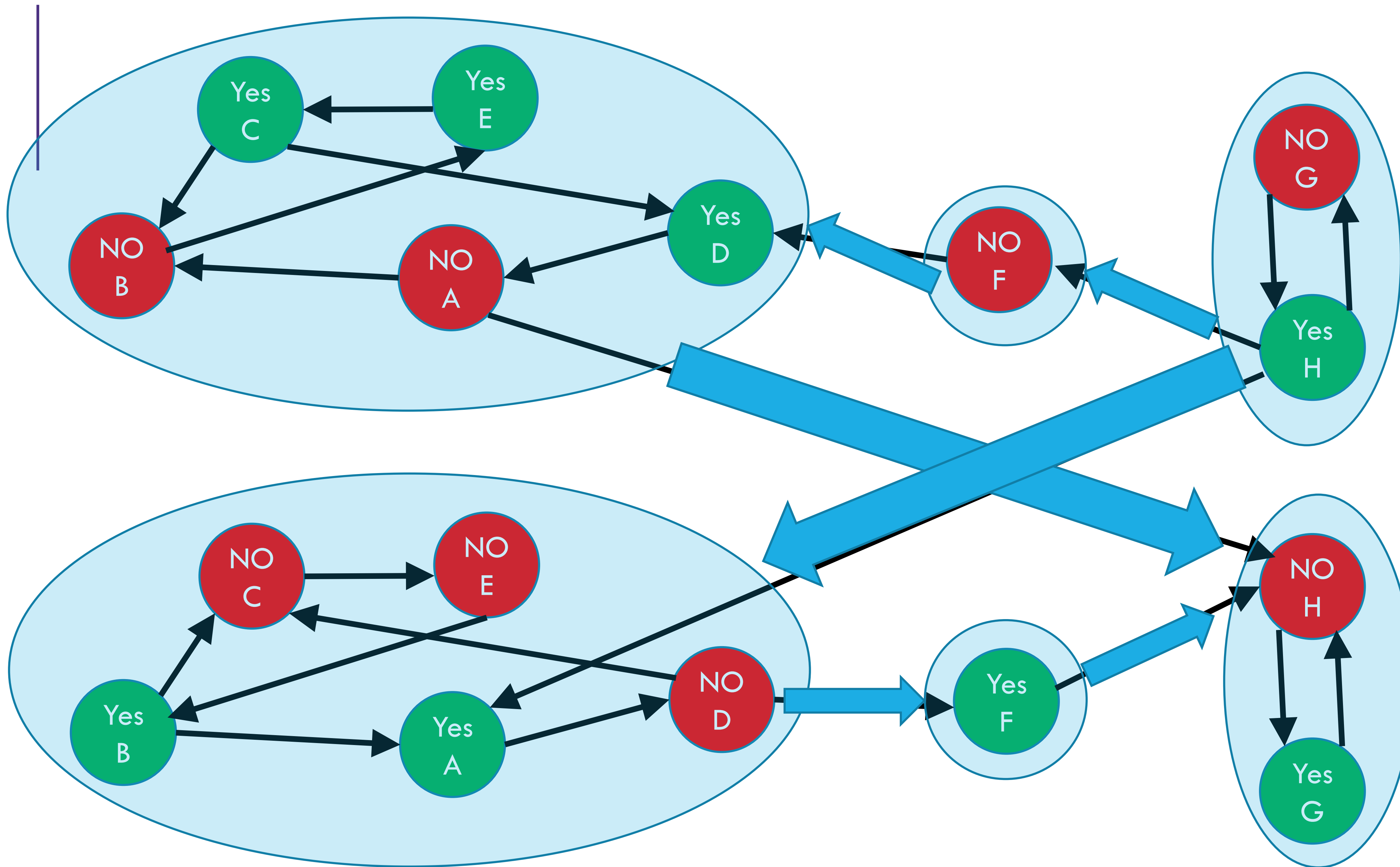
Now what? Every SCC gets the same value.
- Treat it as a single object!

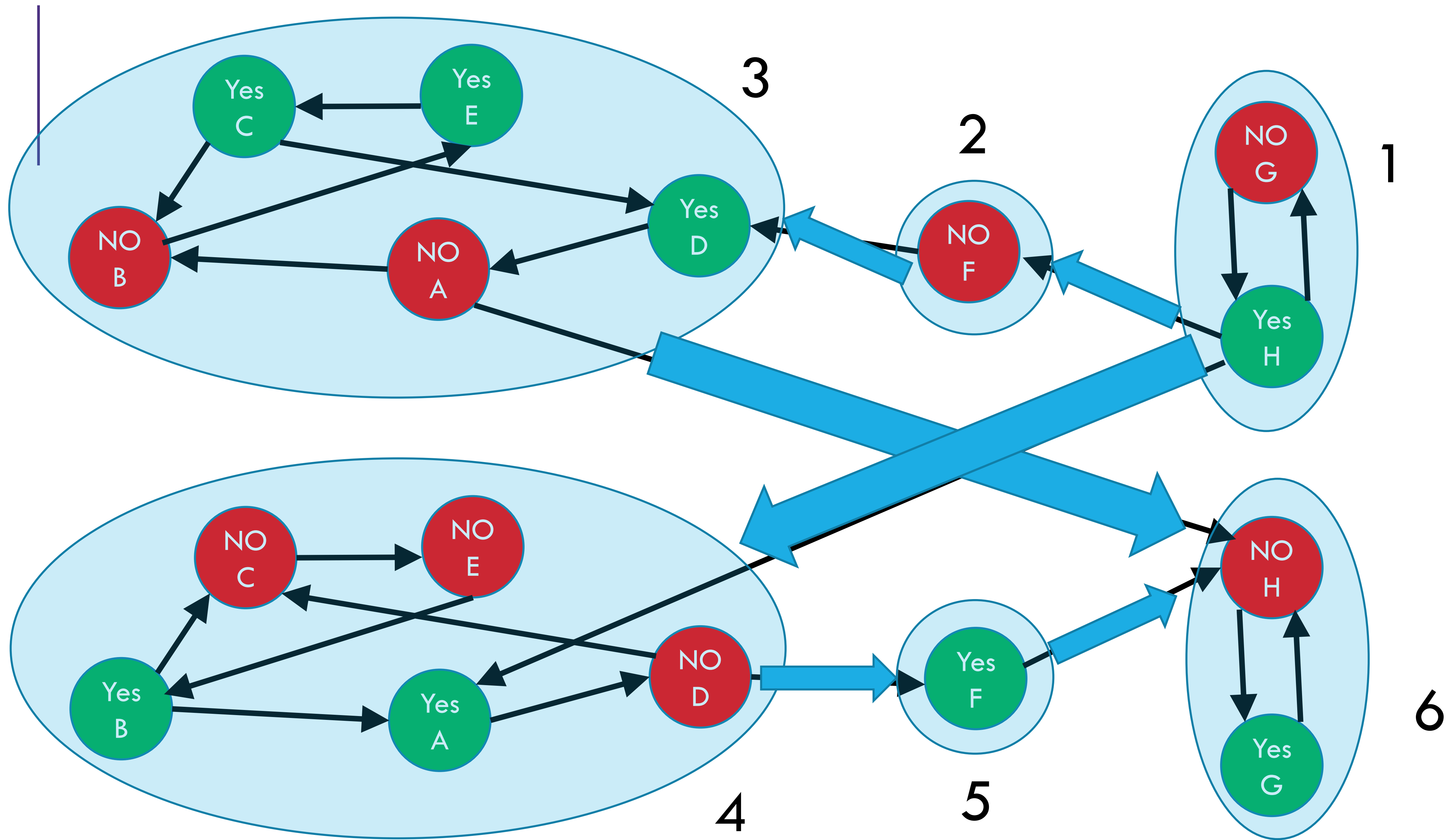
We want to avoid edges from true things to false things.
- “Trues” seem more useful for us at the end.

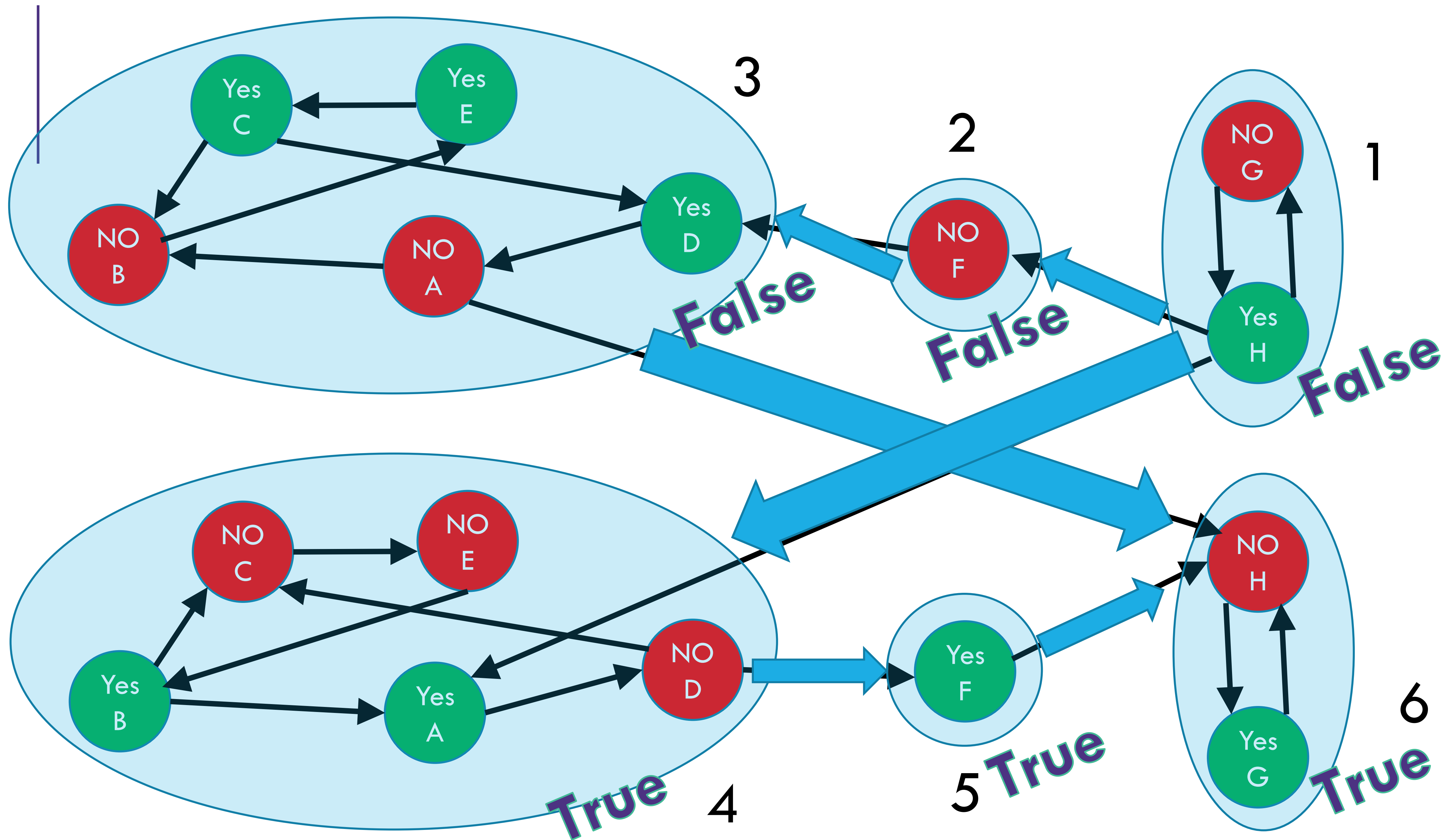
Is there some way to start from the end?

YES! Topological Sort


Yes
C
Yes
E
NO
B
NO
A
Yes
D
NO
F
NO
G
Yes
H
NO
C
NO
E
Yes
B
Yes
A
NO
D
Yes
F
NO
H
Yes
G


Yes
C
Yes
E
NO
B
NO
A
Yes
D
NO
F
NO
G
Yes
H
NO
C
NO
E
Yes
B
Yes
A
NO
D
Yes
F
NO
H
Yes
G


Yes
C
Yes
E
3
NO
B
NO
A
Yes
D
2
NO
F
NO
G
1
Yes
H
NO
C
NO
E
Yes
B
Yes
A
NO
D
Yes
F
NO
H
Yes
G
6
4
5


3
2
1
Yes
C
Yes
E
NO
B
NO
A
Yes
D
NO
F
NO
G
Yes
H
False
False
False
NO
C
NO
E
Yes
B
Yes
A
NO
D
Yes
F
NO
H
Yes
G
True
True
True
4
5
6

Making the Final

Algorithm:
Make the requirements graph.

Find the SCCs.

If any SCC has including and not including a problem, we can't make the final.

Run topological sort on the graph of SCC.

Starting from the end:

- if everything in a component is unassigned, set them to true, and set their opposites to false.

This works!!

How fast is it?

O(Q + S). That's a HUGE improvement.

Some More Context

The Final Making Problem was a type of "Satisfiability" problem.

We had a bunch of variables (include/exclude this question), and needed to satisfy everything in a list of requirements.

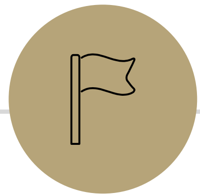
2-Satisfiability ("2-SAT")

Given: A set of Boolean variables, and a list of requirements, each of the form:

```
variable1==[True/False] || variable2==[True/False]
```

Find: A setting of variables to "true" and "false" so that **all** of the requirements evaluate to "true"

The algorithm we just made for Final Creation works for any 2-SAT problem.



Reductions, P vs. NP

What are we doing?

To wrap up the course we want to take a big step back.

This whole quarter we've been taking problems and solving them faster.

We want to spend the last few lectures going over more ideas on how to solve problems faster, and why we don't expect to solve everything extremely quickly.

We're going to

- Recall reductions – Robbie's favorite idea in algorithm design.
- Classify problems into those we can solve in a reasonable amount of time, and those we can't.
- Explain the biggest open problem in Computer Science

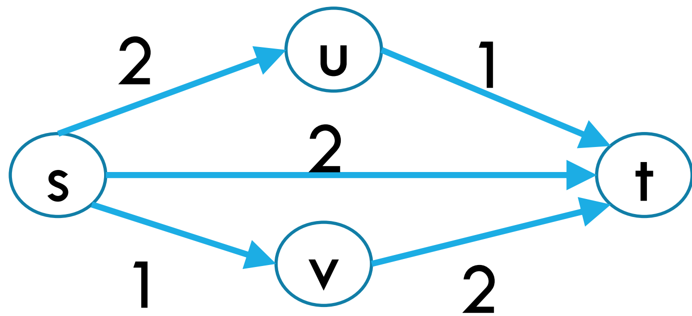
Reductions: Take 2

Reduction (informally)

Using an algorithm for Problem B to solve Problem A.

We reduced weighted shortest paths to unweighted shortest paths

Weighted Graphs: A Reduction



Transform Input

Unweighted Shortest Paths

Transform Output

Reductions

It might not be too surprising that we can solve one shortest path problem with the algorithm for another shortest path problem.

The real power of reductions is that you can sometimes reduce a problem to another one that looks very very different.

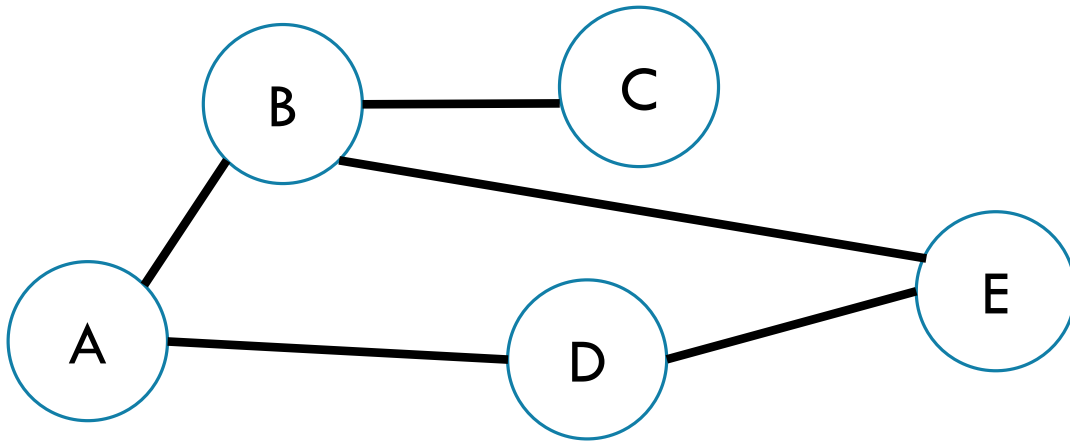
We're going to reduce a graph problem to 2-SAT.

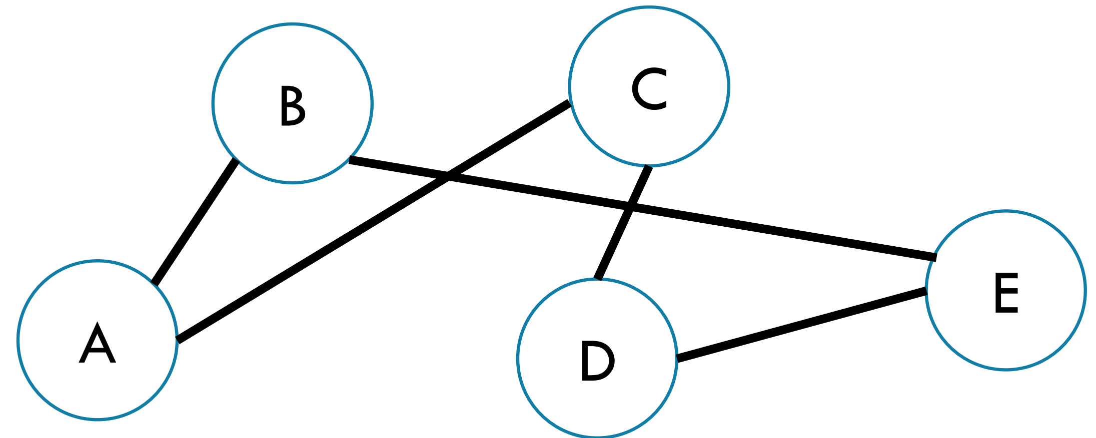
2-Coloring

Given an undirected, unweighted graph G, color each vertex "red" or "blue" such that the endpoints of every edge are different colors (or report no such coloring exists).

2-Coloring

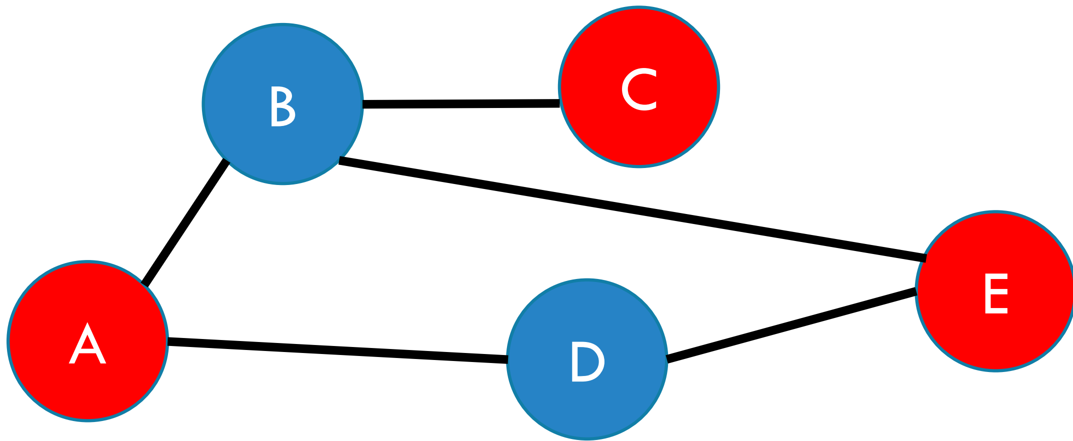
Can these graphs be 2-colored? If so find a 2-coloring. If not try to explain why one doesn't exist.

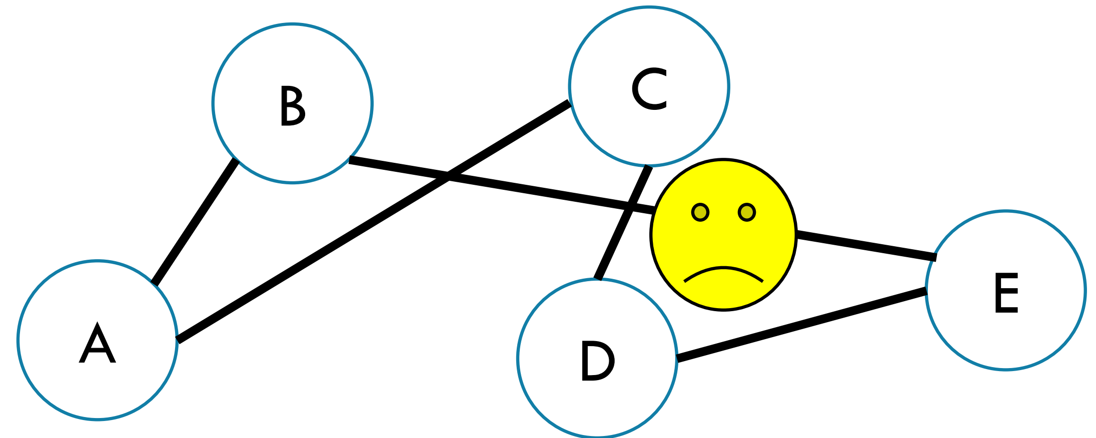




2-Coloring

Can these graphs be 2-colored? If so find a 2-coloring. If not try to explain why one doesn't exist.





2-Coloring

Why would we want to 2-color a graph?

- We need to divide the vertices into two sets, and edges represent vertices that **can't** be together.

You can modify [B/D]FS to come up with a 2-coloring (or determine none exists)

- This is a good exercise!

But coming up with a whole new idea sounds like **work.**

And we already came up with that cool 2-SAT algorithm.

- Maybe we can be lazy and just use that!
- Let's **reduce** 2-Coloring to 2-SAT!

Use our 2-SAT algorithm
to solve 2-Coloring

A Reduction

We need to describe 2 steps

1. How to turn a graph for a 2-color problem into an input to 2-SAT

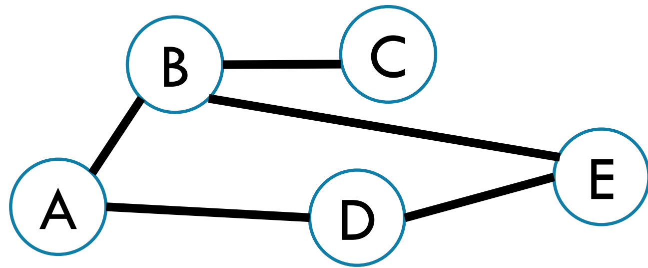
2. How to turn the ANSWER for that 2-SAT input into the answer for the original 2-coloring problem.

How can I describe a two coloring of my graph?

- Have a variable for each vertex – is it red?

How do I make sure every edge has different colors? I need one red endpoint and one blue one, so this better be true to have an edge from v1 to v2:

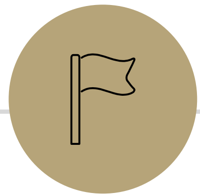
```
(v1IsRed || v2isRed) && (!v1IsRed || !v2IsRed)
```



Transform Input

2-SAT Algorithm

Transform Output



Graph Modeling Practice

Graph Modeling Process

1. What are your fundamental objects?
   - Those will probably become your vertices.
2. How are those objects related?
   - Represent those relationships with edges.
3. How is what I'm looking for encoded in the graph?
   - Do I need a path from s to t? The shortest path from s to t? A minimum spanning tree? Something else?
4. Do I know how to find what I'm looking for?
   - Then run that algorithm/combination of algorithms
   - Otherwise go back to step 1 and try again.

Scenario #1

You are a Disneyland employee and you need to rope off as many miles of walkways as you can for the fireworks while still allowing guests to access to all the rides.

What are the vertices?

What are the edges?

What are we looking for?

What do we run?

Scenario #1

You are a Disneyland employee and you need to rope off as many miles of walkways as you can for the fireworks while still allowing guests to access to all the rides.

What are the vertices?

Rides

What are the edges?

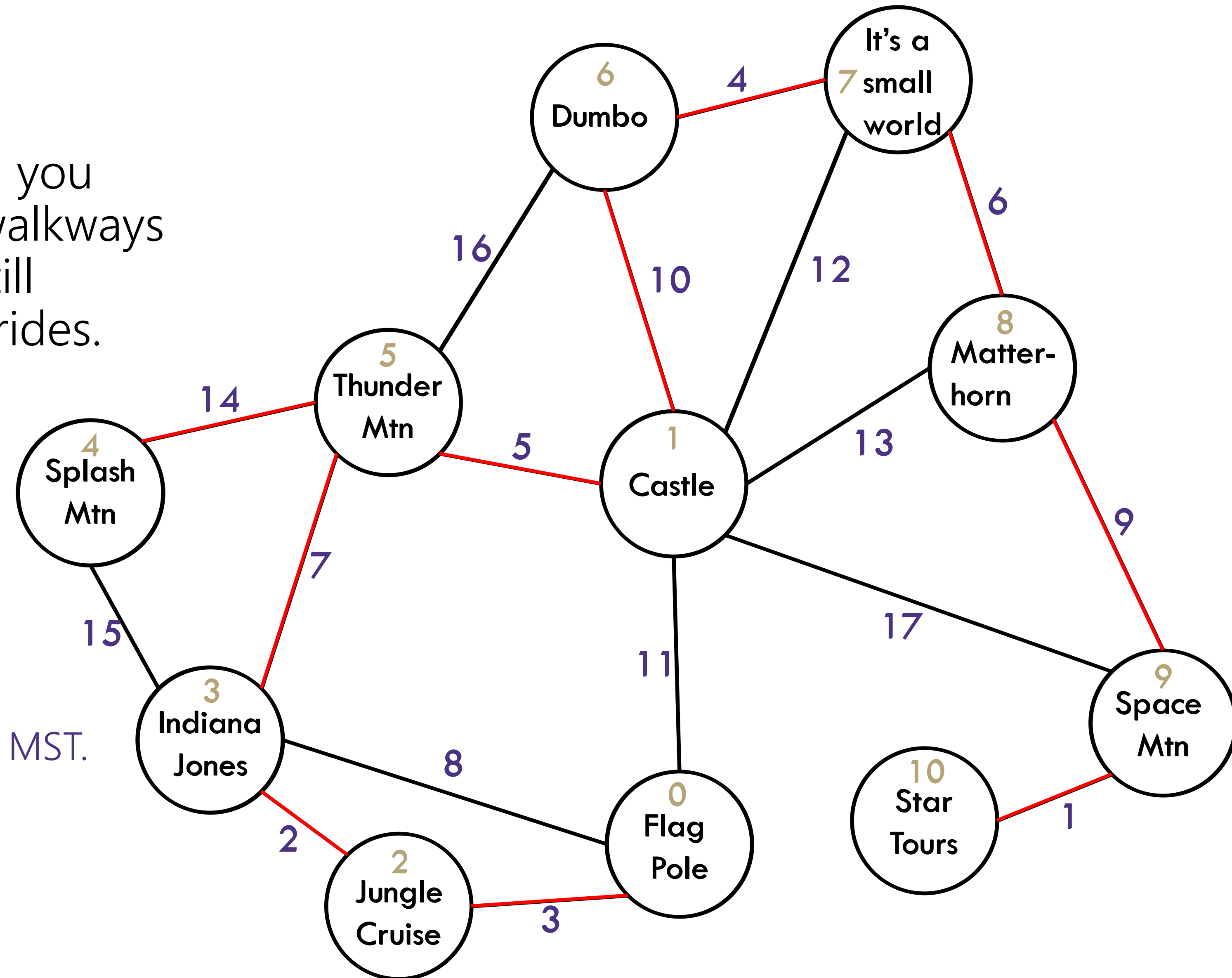
Walkways with distances

What are we looking for?

- We want to rope off everything except an MST.

What do we run?

- Kruskal's or Prim's



Scenario #2

You are at Splash Mountain. Your best friend is at Space Mountain. You have to tell each other about your experiences in person as soon as possible. Where do you meet and how quickly can you get there?

What are the vertices?

What are the edges?

What are we looking for?

1

What do we run?

Scenario #2

You are at Splash Mountain. Your best friend is at Space Mountain. You have to tell each other about your experiences in person as soon as possible. Where do you meet and how quickly can you get there?

What are the vertices?

Rides

What are the edges?

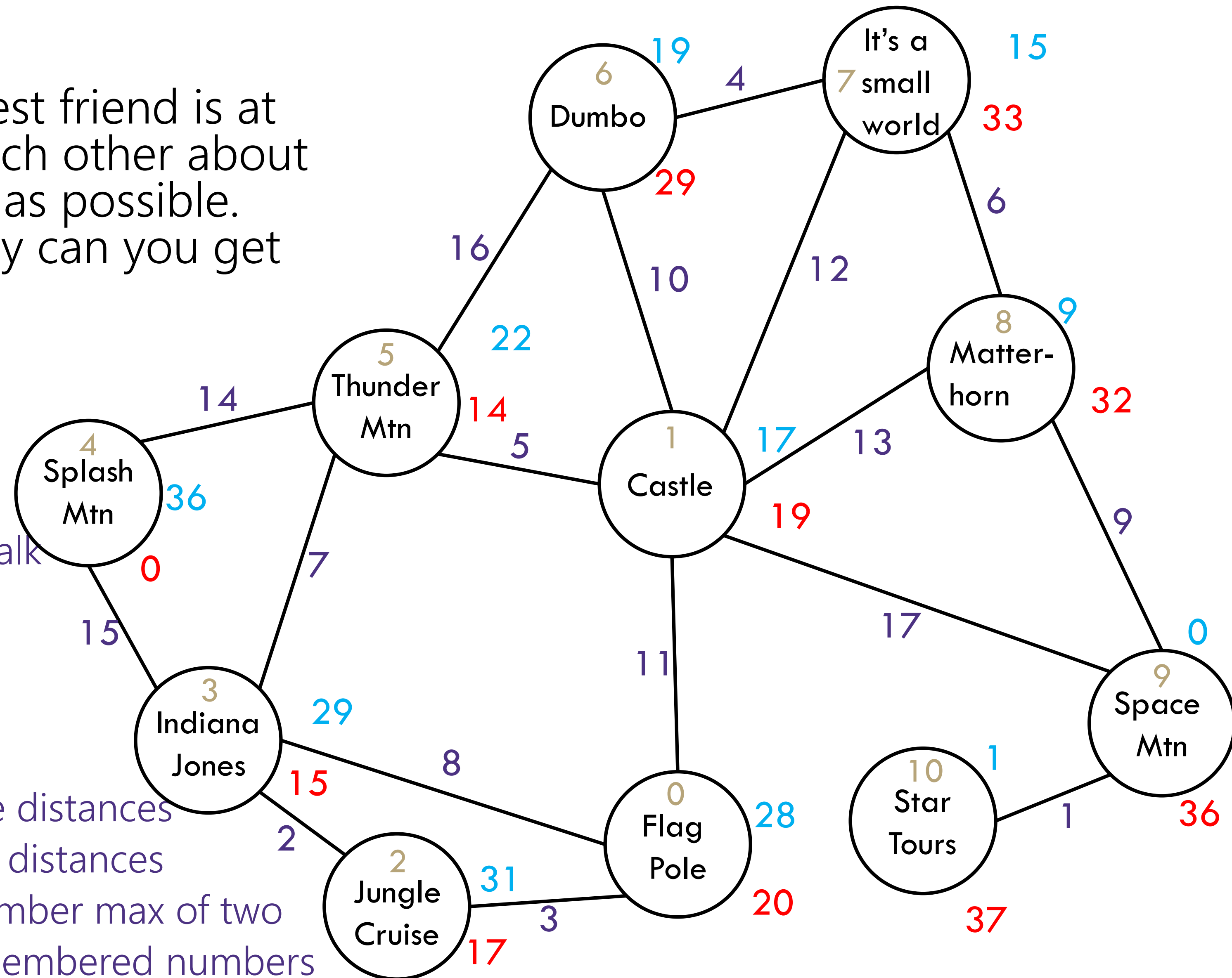
Walkways with how long it would take to walk

What are we looking for?

- The "midpoint"

What do we run?

- Run Dijkstra's from Splash Mountain, store distances
- Run Dijkstra's from Space Mountain, store distances
- Iterate over vertices, for each vertex remember max of two
- Iterate over vertices, find minimum of remembered numbers



Scenario #3

You're a Disneyland employee, working the front of the Splash Mountain line. Suddenly, the crowd-control gates fall over and the line degrades into an unordered mass of people.

Sometimes you can tell who was in line before who; for other groups you aren't quite sure. You need to restore the line, while ensuring if you **knew** A came before B before the incident, they will still be in the right order afterward.

What are the vertices?

What are the edges?

What are we looking for?

What do we run?

Scenario #3

You're a Disneyland employee, working the front of the Splash Mountain line. Suddenly, the crowd-control gates fall over and the line degrades into an unordered mass of people.

Sometimes you can tell who was in line before who; for other groups you aren't quite sure. You need to restore the line, while ensuring if you **knew** A came before B before the incident, they will still be in the right order afterward.

What are the vertices?

People

What are the edges?

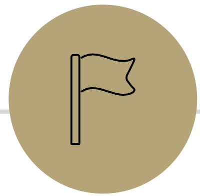
Edges are directed, have an edge from X to Y if you know X came before Y.

What are we looking for?

- A total ordering consistent with all the ordering we do know.

What do we run?

- Topological Sort!



Reductions, P vs. NP

Efficient

We'll consider a problem "efficiently solvable" if it has a polynomial time algorithm.

I.e. an algorithm that runs in time $O(n^k)$ where k is a constant.

Are these algorithms always actually efficient?

Well………no

Your n^{10000} algorithm or even your $2^{2^{2^{2^{2}}}} \cdot n^3$ algorithm probably aren't going to finish anytime soon.

But these edge cases are rare, and polynomial time is good as a low bar

- If we can't even find an n^{10000} algorithm, we should probably rethink our strategy

Decision Problems

Let's go back to dividing problems into solvable/not solvable.
For today, we're going to talk about **decision problems**.

Problems that have a "yes" or "no" answer.

Why?

Theory reasons (ask me later).

But it's not too bad

- most problems can be rephrased as very similar decision problems.

E.g. instead of "find the shortest path from s to t" ask
Is there a path from s to t of length at most k?

P

P (stands for "Polynomial")

The set of all decision problems that have an algorithm that runs in time $O(n^k)$ for some constant k.

The decision version of all problems we've solved in this class are in P.

P is an example of a "complexity class"

A set of problems that can be solved under some limitations (e.g. with some amount of memory or in some amount of time).

I'll know it when I see it.

Another class of problems we want to talk about.

"I'll know it when I see it" Problems.

Decision Problems such that:

If the answer is YES, you can prove the answer is yes by
- Being given a "proof" or a "certificate"
- Verifying that certificate in polynomial time.

What certificate would be convenient for short paths?
- The path itself. Easy to check the path is really in the graph and really short.

I’ll know it when I see it.

More formally,

NP (stands for “nondeterministic polynomial”)

The set of all decision problems such that if the answer is YES, there is a proof of that which can be verified in polynomial time.

It’s a common misconception that NP stands for “not polynomial”
Please never ever ever ever say that.

Please.

Every time you do a theoretical computer scientist sheds a single tear.

(That theoretical computer scientist is me)