Lecture 20: Topological sort, reductions

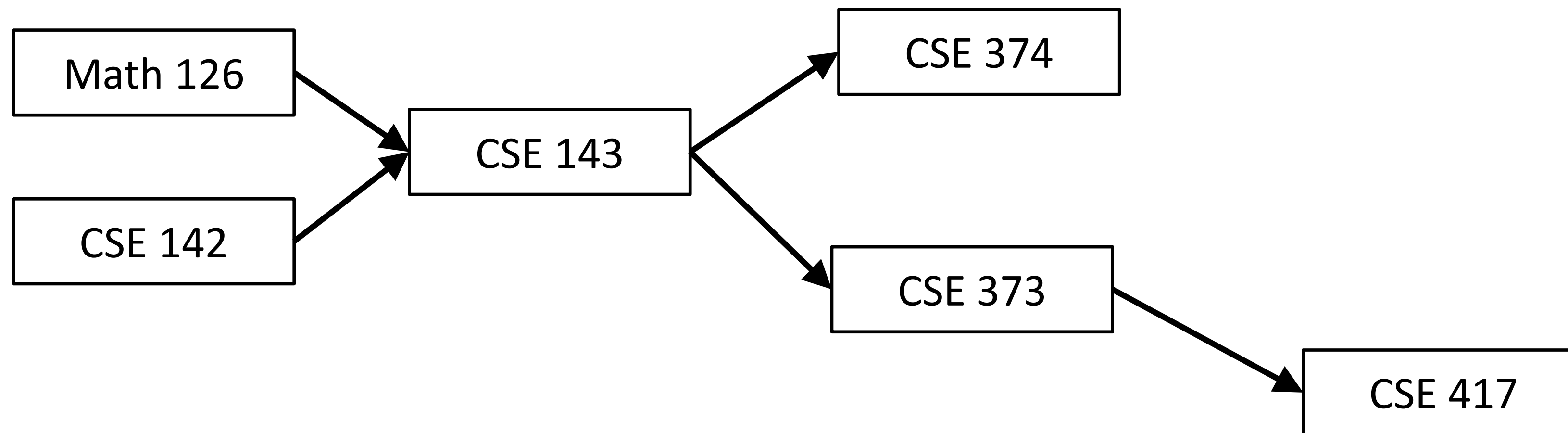
Data Structures and Algorithms

Roadmap

- topological sort
- CSE373 20su creation, 2-Sat
- reductions, 2 color
- seam carving

Problem 1: Ordering Dependencies

Today's (first) problem: Given a bunch of courses with prerequisites, find an order to take the courses in.



Problem 1: Ordering Dependencies

Given a directed graph G, where we have an edge from u to v if u must happen before v.

We can only do things one at a time, can we find an order that **respects dependencies**?

Topological Sort (aka Topological Ordering)

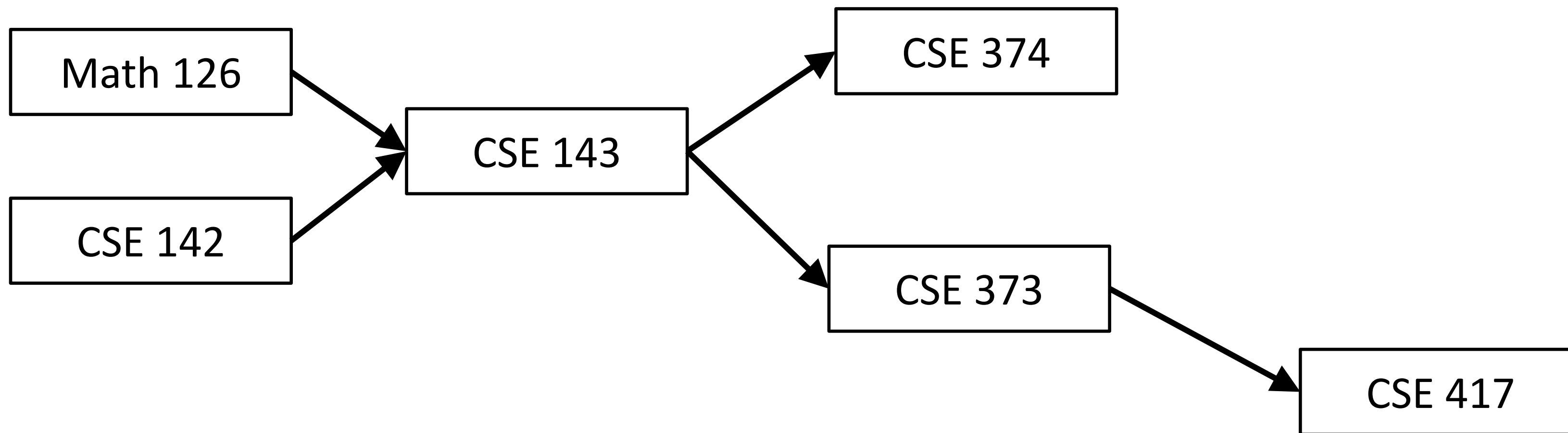
Given: a directed graph G

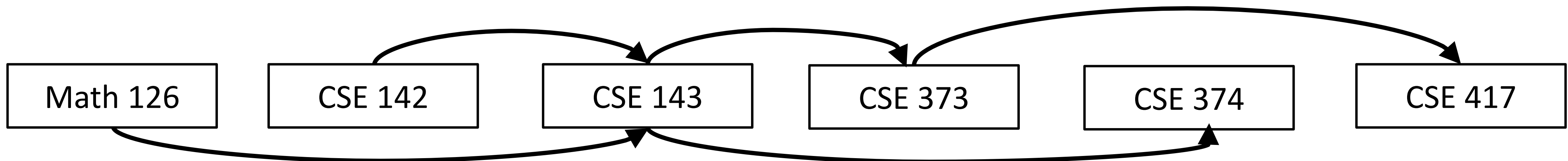
Find: an ordering of the vertices so all edges go from left to right (all the dependency arrows are satisfied and the vertices can be processed left to right with no problems) .

Topological Sort (aka Topological Ordering)

Given: a directed graph G

Find: an ordering of the vertices so all edges go from left to right (all the dependency arrows are satisfied and the vertices can be processed left to right with no problems) .





Problem 1: Ordering Dependencies

Given a directed graph G, where we have an edge from u to v if u must happen before v.

We can only do things one at a time, can we find an order that **respects dependencies**?

Topological Sort (aka Topological Ordering)

Given: a directed graph G

Find: an ordering of the vertices so all edges go from left to right (all the dependency arrows are satisfied and the vertices can be processed left to right with no problems) .

Uses:

Graduating

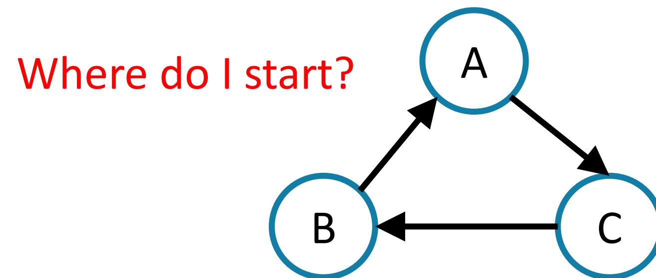
Cooking

Organizing TODO-lists

Compiling multiple files

Can we always order a graph?

Can you topologically order this graph? No



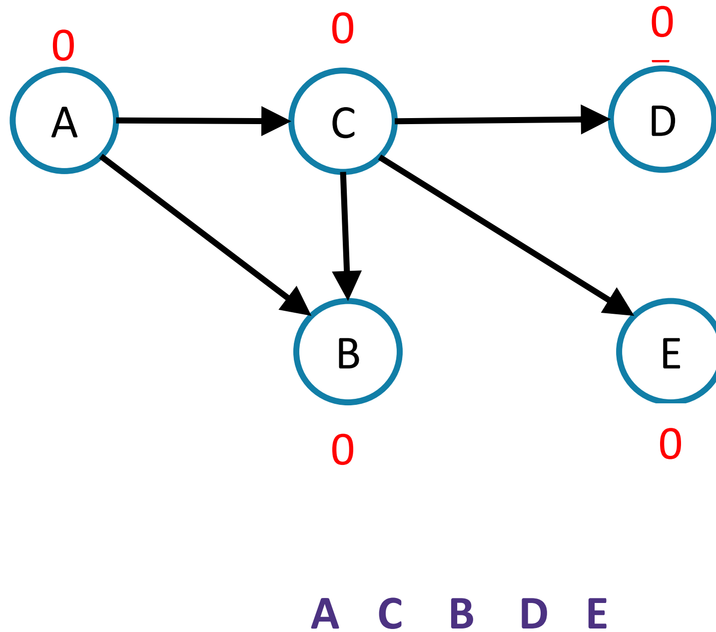
Directed Acyclic Graph (DAG)

A directed graph without any cycles.

A graph has a topological ordering **if and only if** it is a DAG.

Ordering a DAG

Does this graph have a topological ordering? If so find one.



A C B D E

If a vertex doesn't have any edges going into it, we can add it to the ordering.
More generally, if the only incoming edges are from vertices already in the ordering, it's safe to add.

How Do We Find a Topological Ordering?

```
TopologicalSort(Graph G, Vertex source)
   count how many incoming edges each vertex has
   Collection toProcess = new Collection()
   foreach(Vertex v in G){
      if(v.edgesRemaining == 0)
         toProcess.insert(v)
   }
   topOrder = new List()
   while(toProcess is not empty){
      u = toProcess.remove()
      topOrder.insert(u)
      foreach(edge (u,v) leaving u){
         v.edgesRemaining--
         if(v.edgesRemaining == 0)
            toProcess.insert(v)
      }
   }
```

[B/D]FS
Graph linear
+ V + E

Pick something with $O(1)$ insert / removal

$+V$

Runs as most once per edge
$+E$

$\boldsymbol{O(V + E)}$

Roadmap

- topological sort
- CSE373 20su creation, 2-Sat,
- reductions, 2 color
- seam carving

Example Problem: CSE 373 20su creation problem

Every quarter we send out an additional anonymous survey about reflecting on the course and improving the course for future quarters (rather than evaluating the instructors)

- We list out each of the topics we've covered: Heap insertions, big-O problems, tree method, graph modeling… and ask you: what should we keep or throw away from the course?
- To try to make y'all and future students happy and satisfied, we ask for your preferences. In this problem, we're going to have each of you list two preferences of the form "I [do/don't] want [] topic to be in CSE 373 20su version"

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

CSE373 20su Creation Problem
Given: A list of 2 preferences per student. **Find**: A set of topics so that every student gets at least one of their preferences (or accurately report no such topic set exists).

Example Problem: CSE 373 20su creation problem

CSE373 20su Creation Problem

Given: A list of 2 yes/no preferences per student.
Find: A set of topics so that every student gets at least one of their preferences (or accurately report no such topic set exists).

Student A:

- no tree method OR

- yes graph modeling

Student B:

- yes tree method OR

- yes hash maps

Some things to note:

- example solution/thought: if there is no tree method, Student A is happy, but Student B would be happy just yet. To make Student B happy as well (make sure at least one preference is satisfied), we would make sure to include hash maps as a topic.
- just with 2 student preferences, there's already some constraints on our possible solutions! Because there's disagreement on including the tree method in future versions, it means that whoever doesn't get what they want for that question MUST have their other preference satisfied to complete this problem.

Example Problem: CSE 373 20su creation problem

CSE373 20su Creation Problem
Given: A list of 2 yes/no preferences per student. **Find**: A set of topics so that every student gets at least one of their preferences (or accurately report no such topic set exists).

Student A:
- no tree method OR
- yes graph modeling

Student B:
- yes tree method OR
- yes hash maps

Student C:
- no asymptotic analysis OR
- no heaps

Some example solutions:

- A)
  - no tree method
  - yes hash maps
  - no asymptotic analysis
- B)
  - yes tree method
  - yes graph modeling
  - no heaps

CSE373 20su Creation Problem

Given: A list of 2 yes/no preferences per student.
Find: A set of topics so that every student gets at least one of their preferences (or accurately report no such topic set exists).

Student A:
- no tree method
- yes graph modeling

Student B:
- yes tree method
- yes hash maps

Student C:
- no asymptotic analysis
- no heaps

This problem (with a more general context 373 topics) is called the 2-satisfiability problem (2-SAT).

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"

- You can thing of these student preferences as parameters to our problem where each student's preference is a boolean statement of a specific form. The student preferences above would be:
  - !treeMethod || graphModeling
  - treeMethod || hashMaps
  - !asymptoticAnalysis || !heaps

Rephrased problem statement: go through these list of boolean conditions (preferences) where each is of the form (someVar || someOtherVar), and make sure they all preference entries evaluate to true for your proposed solution

20su Creation (2-SAT) algorithm

We have T kinds of topics and S students.

What if we try/loop through every possible combination of questions and then loop over the solution to make sure it works? If it does work then we can stop there, but in the worst case we'll have to try every single combination.

```
generate all the possible solution lists of T/F for each topic

for each proposed solution of topics:

    loop over the preferences input list and check that every
     entry has at least one of their preferences satisfied
```

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

- there are 2^T possible proposed solutions, since there are T topics that could be included/not included.

- the input list is always of size S since there are S student preferences

O($2^T S$) is pretty slow but at least it's something. You never know when these types of things are going to come in handy.

If you want to see a more efficient way to solve this problem (S + T time) check out the end of this slide deck for an algorithm that will not be tested in this class. It uses a graph and topological sort (if you think about the last problem you could sort of see the fact that if some topic is chosen, it implies that something else has to be chose/not chosen, and we can represent those dependencies with a graph and topological sort to help us figure out what order to assign the problems in)

Roadmap

- topological sort
- CSE373 20su creation, 2-Sat
- reductions, 2-color
- seam carving

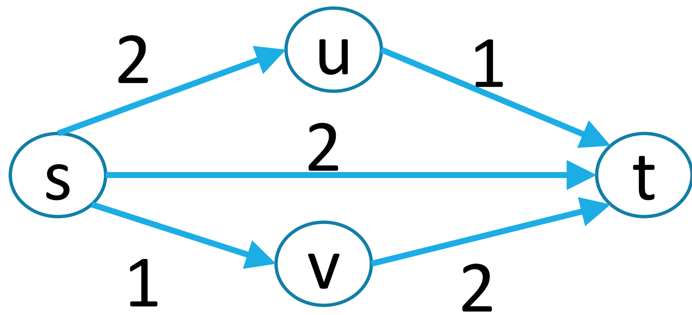
Reductions: Take 2

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.

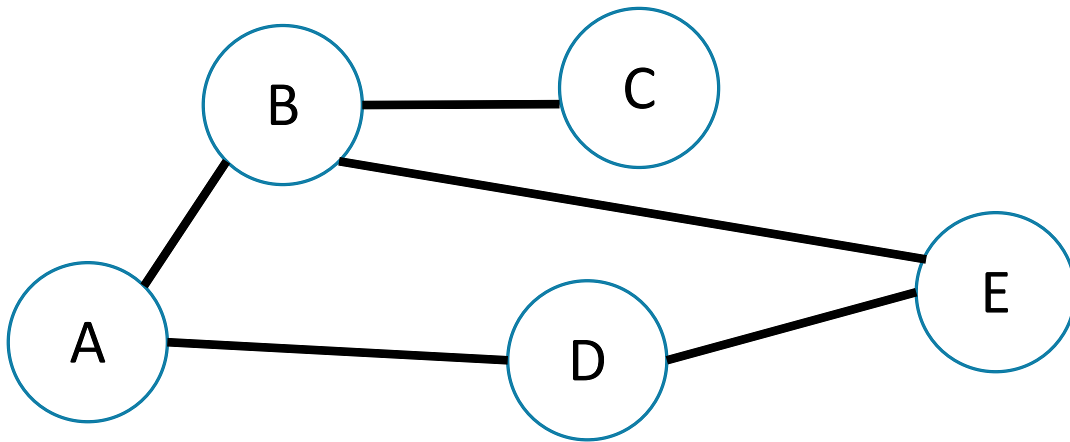
We're going to reduce a graph problem to 2-SAT.

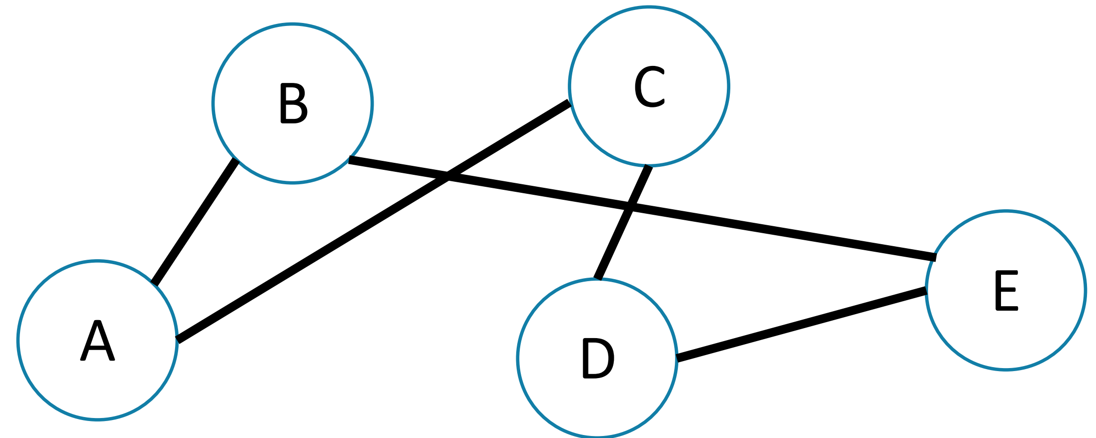
2-Coloring

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.



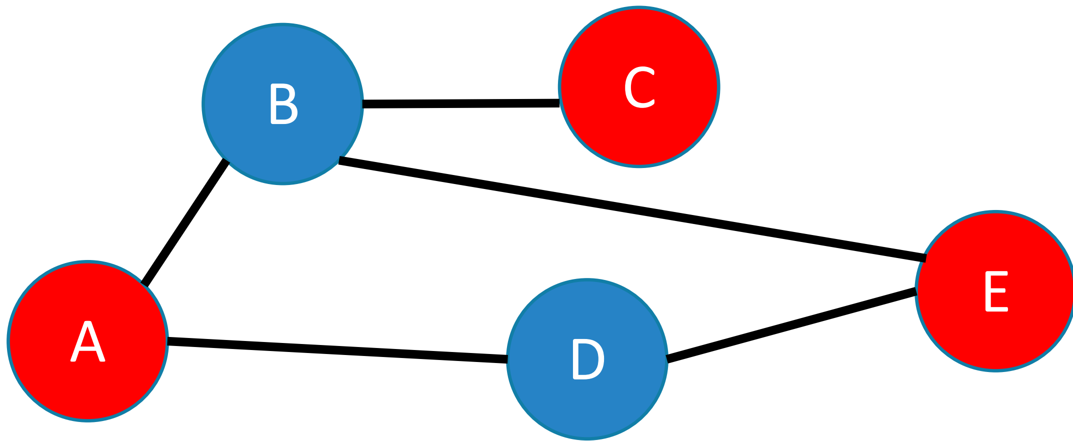


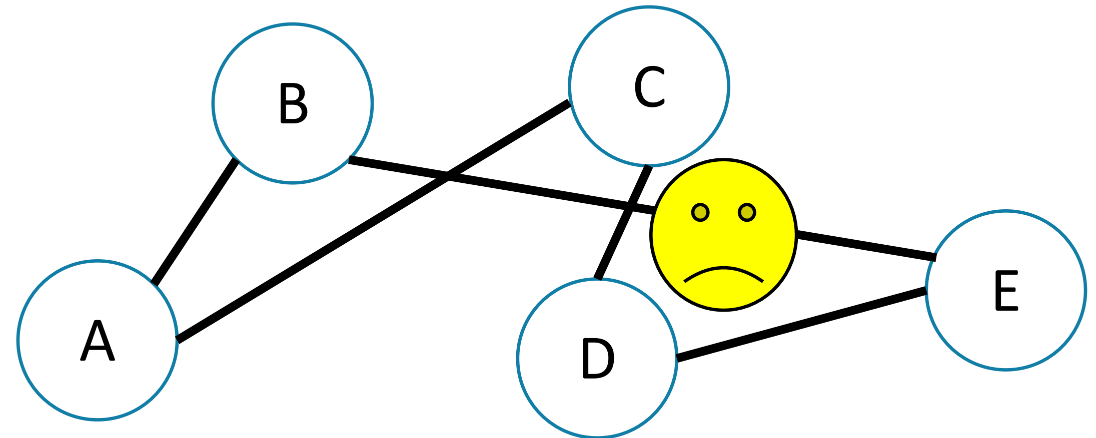
2-Coloring

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

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

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 a 2-SAT algorithm.

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

Use a 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

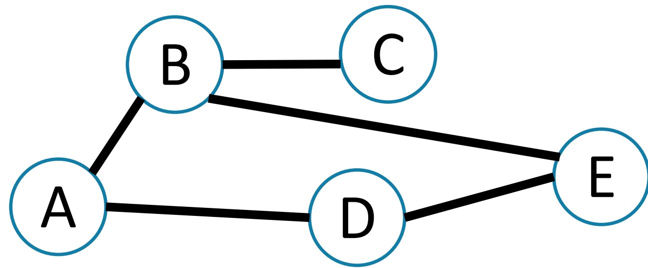
2. How to turn the ANSWER for that 2-SAT input into the answer for the original 2-coloring problem.

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

Other examples of reductions

solving list.contains(item) can be reduced to solving list.indexOf(item) and then checking what the index was (if -1 false, true otherwise)

finding the minimum value in a list can be reduced to sorting your array/ turning it into a min-heap / and then just looking up the min value afterwards.

reductions are all about realizing that a specific problem you have is actually a specific version of a general problem you already know about / have solved. Turn your specific version (with whatever formatting / processing is required) into a form your general problem can solve!

More thoughts on reductions

we can't teach you the solution to every single problem out there… it's also unrealistic to come up with new ideas and solutions to each problem.. a lot of the time reusing existing tools / solutions is a common go to! In computer science / programming: try reducing your problem to an already solved one if you're stuck. Ask yourself: how can I frame this in a different / simpler way /format that I'm more familiar with?

you already do this all the time in a less formal way with frameworks/libraries/looking up already implemented solutions — all we're doing here is explicitly calling out this reduction process as a tool for designing algorithms.

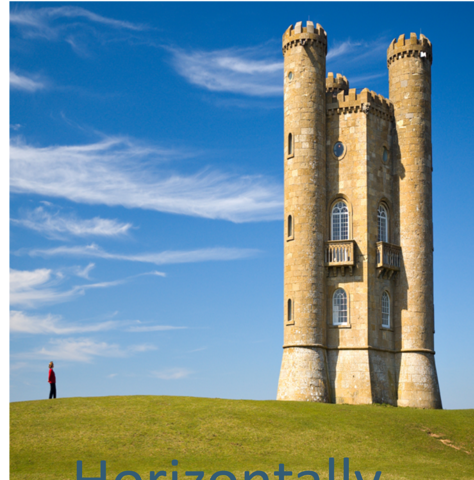
Roadmap

- topological sort
- CSE373 20su creation
- reductions, 2-color
- seam carving

Content-Aware Image Resizing

Seam carving: A distortion-free technique for resizing an image by removing "unimportant seams"

Original Photo



Horizontally-Scaled

(castle and person are distorted)

Seam-Carved

(castle and person are undistorted; "unimportant" sky removed instead)

Seam carving for content-aware image resizing (Avidan, Shamir/ACM); Broadway Tower (Newton2, Yummifruitbat/Wikimedia)

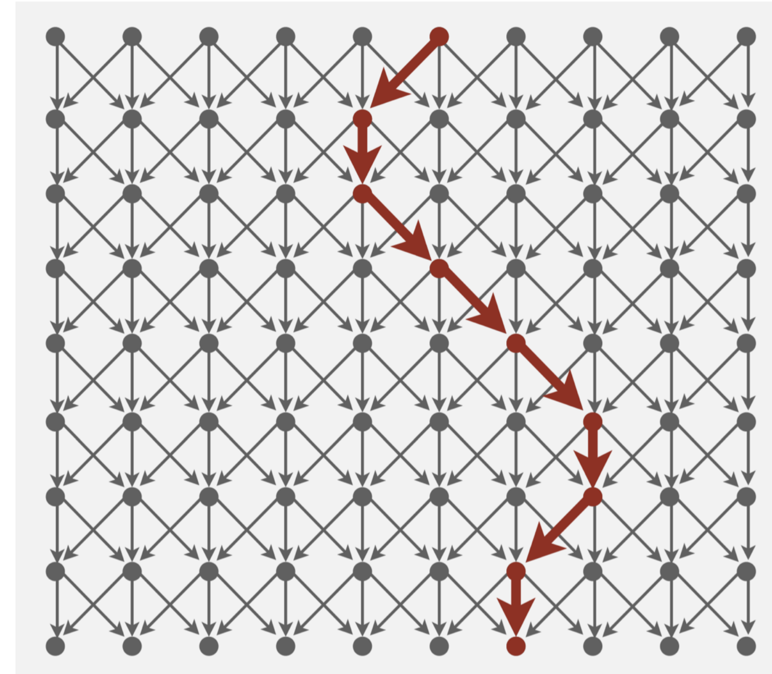
Demo: https://www.youtube.com/watch?v=vIFCV2spKtg

Seam Carving Reduces to Dijkstra's algorithm

1. *Transform the input so that it can be solved by the standard algorithm*
   - Formulate the image as a graph
     - **Vertices**: pixel in the image
     - **Edges**: connects a pixel to its 3 downward neighbors
     - **Edge Weights**: the "energy" (visual difference) between adjacent pixels

2. *Run the standard algorithm as-is on the transformed input*
   - Run Dijkstra's algorithm to find the shortest path (sum of weights) from top row to bottom row

3. *Transform the output of the algorithm to solve the original problem*
   - Interpret the path as a removable "seam" of unimportant pixels

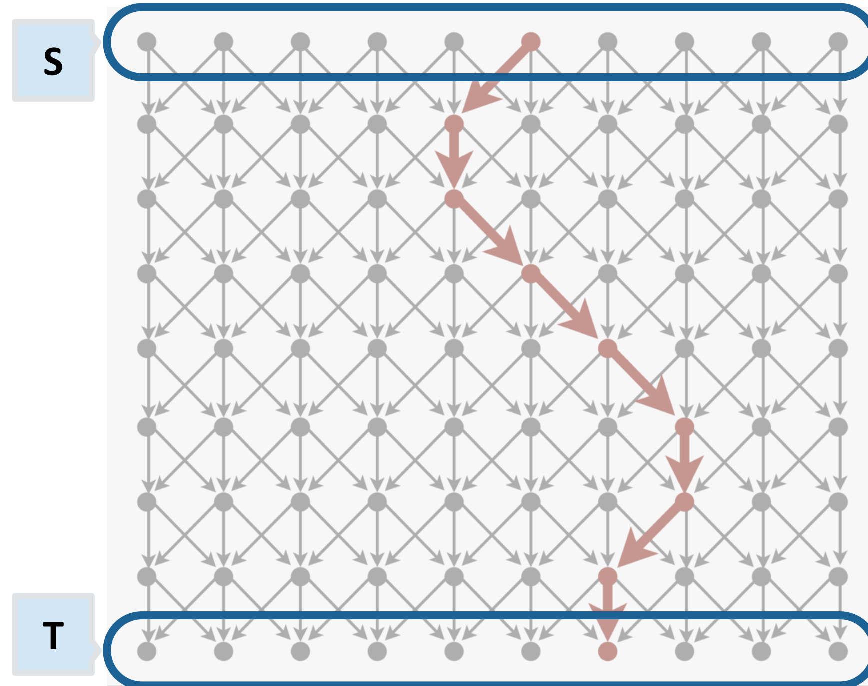


Shortest Paths (Robert Sedgewick, Kevin Wayne/Princeton)

Formal Problem Statement

Using DijkstraShortestPathFinder, find the seam from any top vertex to any bottom vertex

Given a graph with positive edge weights and two distinguished subsets of vertices S and T, find a shortest path from any vertex in S to any vertex in T



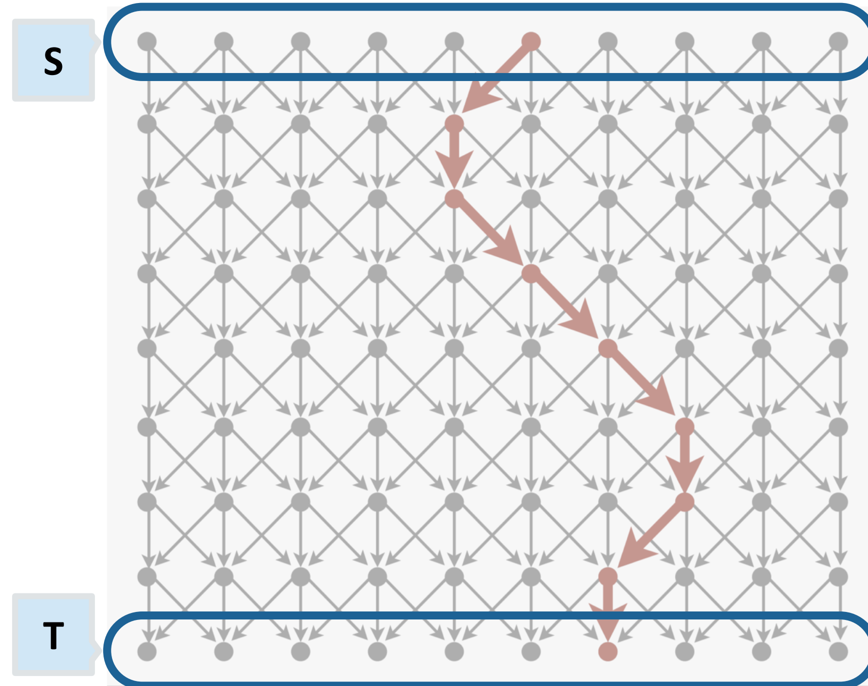
Shortest Paths (Robert Sedgewick, Kevin Wayne/Princeton)

An Incomplete Reduction

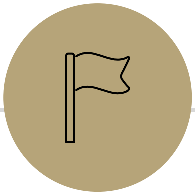
DijkstraShortestPathFinder starts with a single vertex S and ends with a single vertex T

- This problem specifies *sets of vertices* for the start and end

Your turn: brainstorm how to transform this graph into something Dijkstra's knows how to operate on



Shortest Paths (Robert Sedgewick, Kevin Wayne/Princeton)



Content beyond this point is all optional

Connected Components (undirected graphs)

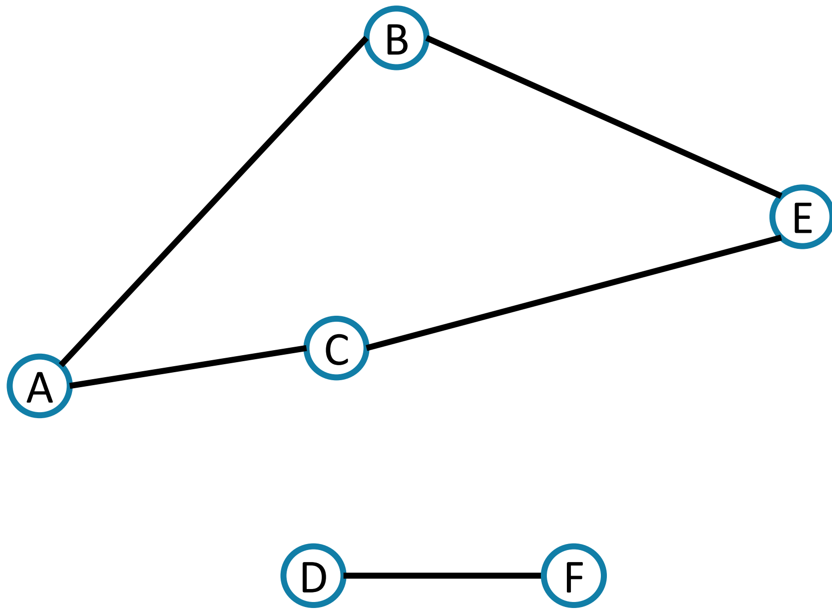
A **connected component** (or just **"component"**) is a "piece" of an undirected graph.

Connected component [undirected graphs]

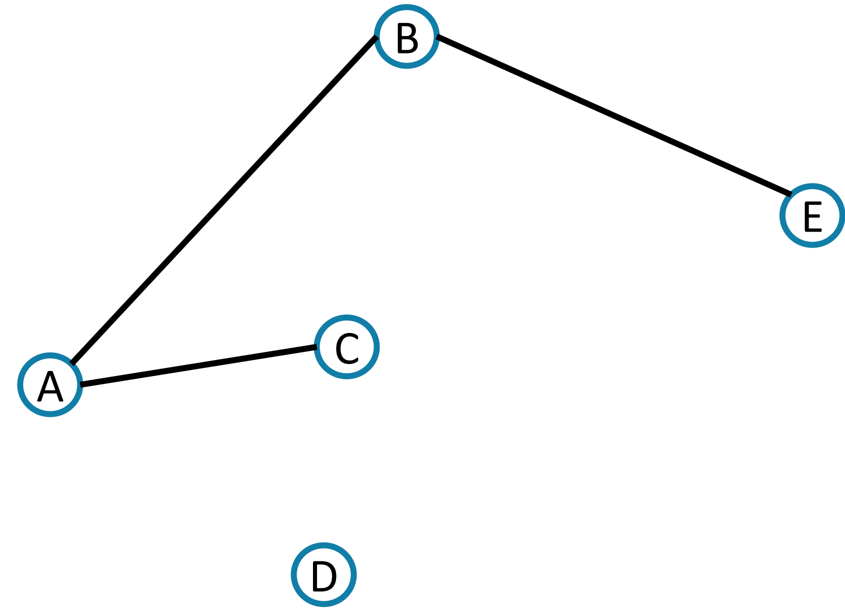
A set S of vertices is a connected component (of an undirected graph) if:

1. It is connected, i.e. for all vertices u, v in S: there is a walk from u to v
2. It is maximal:
   - Either it's the entire set of vertices, or
   - For **every** vertex u that's not in S, $S \cup \{u\}$ is not connected.

Find the connected components



{A,B,C,E}, {D,F} are the two components



{A,B,C,E}, {D} are the two components

Directed Graphs

In directed graphs we have two different notions of "connected"

One is "I can get there from here OR here from there"
The other is "I can get there from here AND here from there"

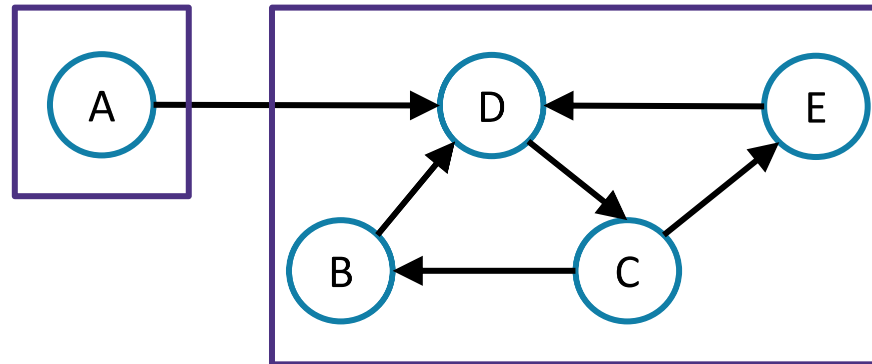
Weakly Connected/Weakly Connected Components:
- Pretend the graph is undirected (ignore the direction of the arrows)
- Find the components of the undirected graph.

Strongly connected components
- Want to get both directions

Strongly Connected Components

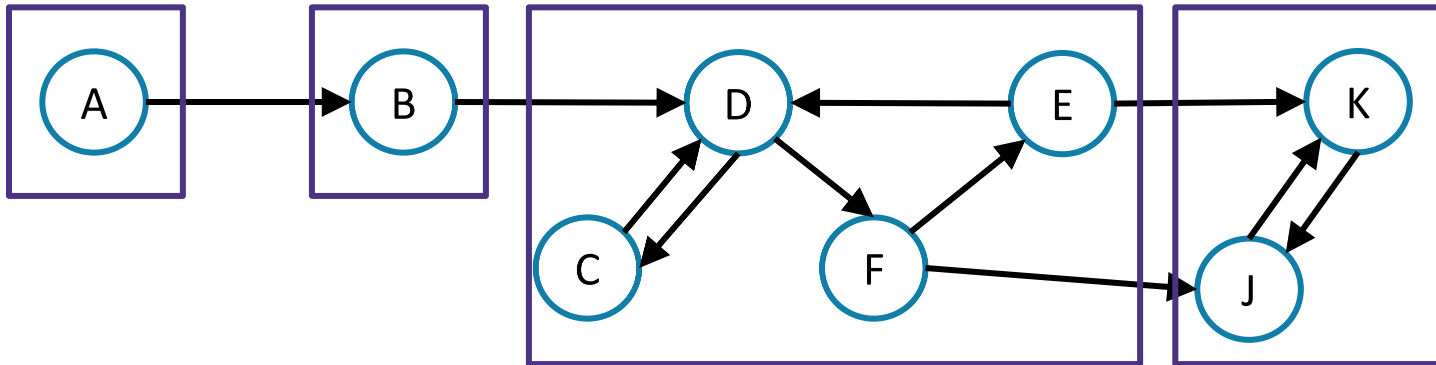
Strongly Connected Component

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!

Your turn: Find Strongly Connected Components



{A}, {B}, {C,D,E,F}, {J,K}

Strongly Connected Component

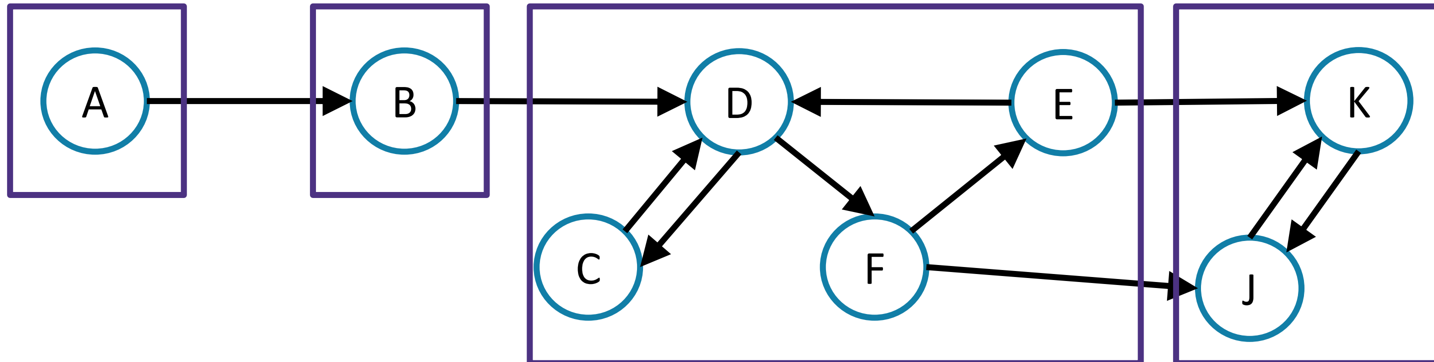
A subgraph C such that every pair of vertices in C is connected via some path **in both directions,** and there is no other vertex which is connected to every vertex of C in both directions.

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



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

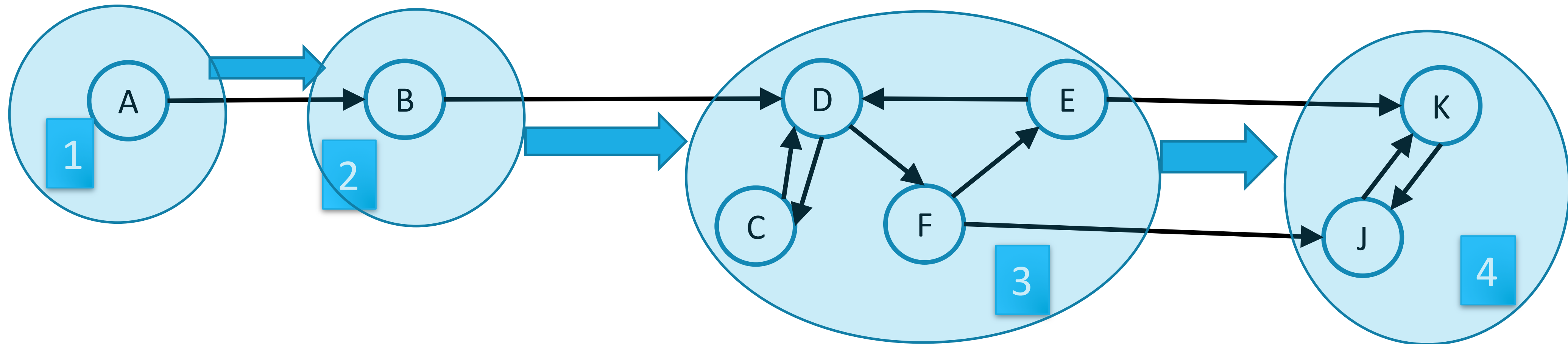
Why Find SCCs?

Graphs are useful because they encode relationships between arbitrary objects.

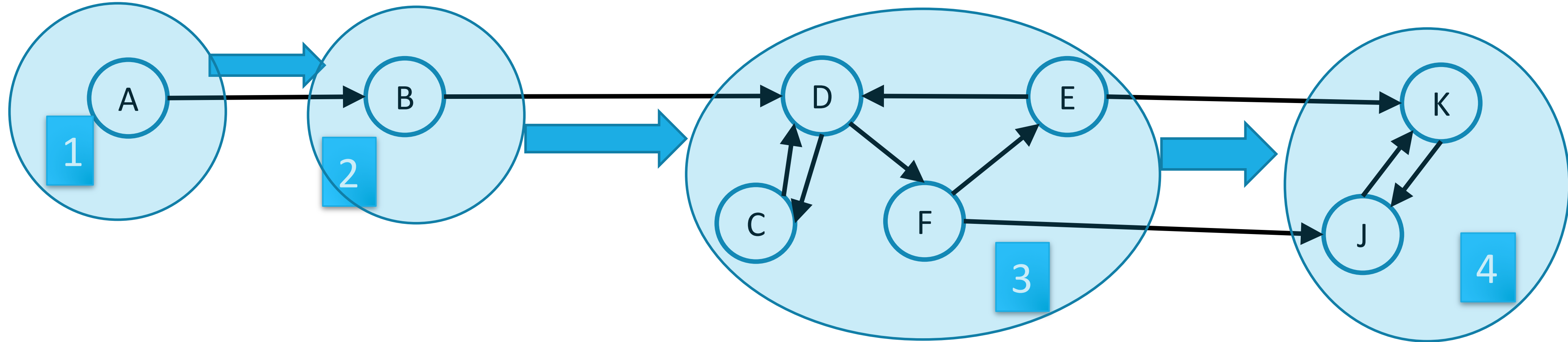
We've found the strongly connected components of G.

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.

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.

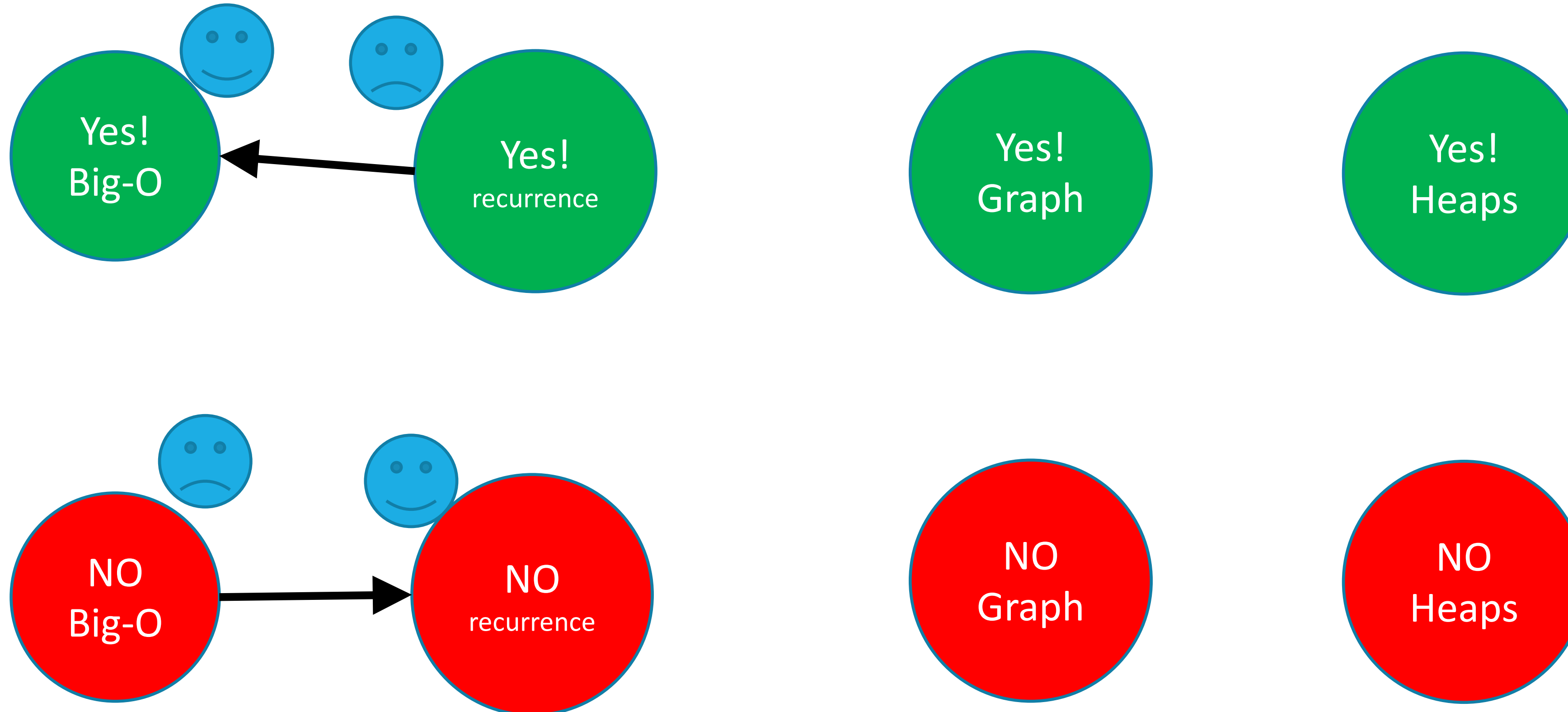
Instead we're going to use a graph.

What should our vertices be?

Review Creation: Take 2

Each student introduces new relationships for data:

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



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?

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

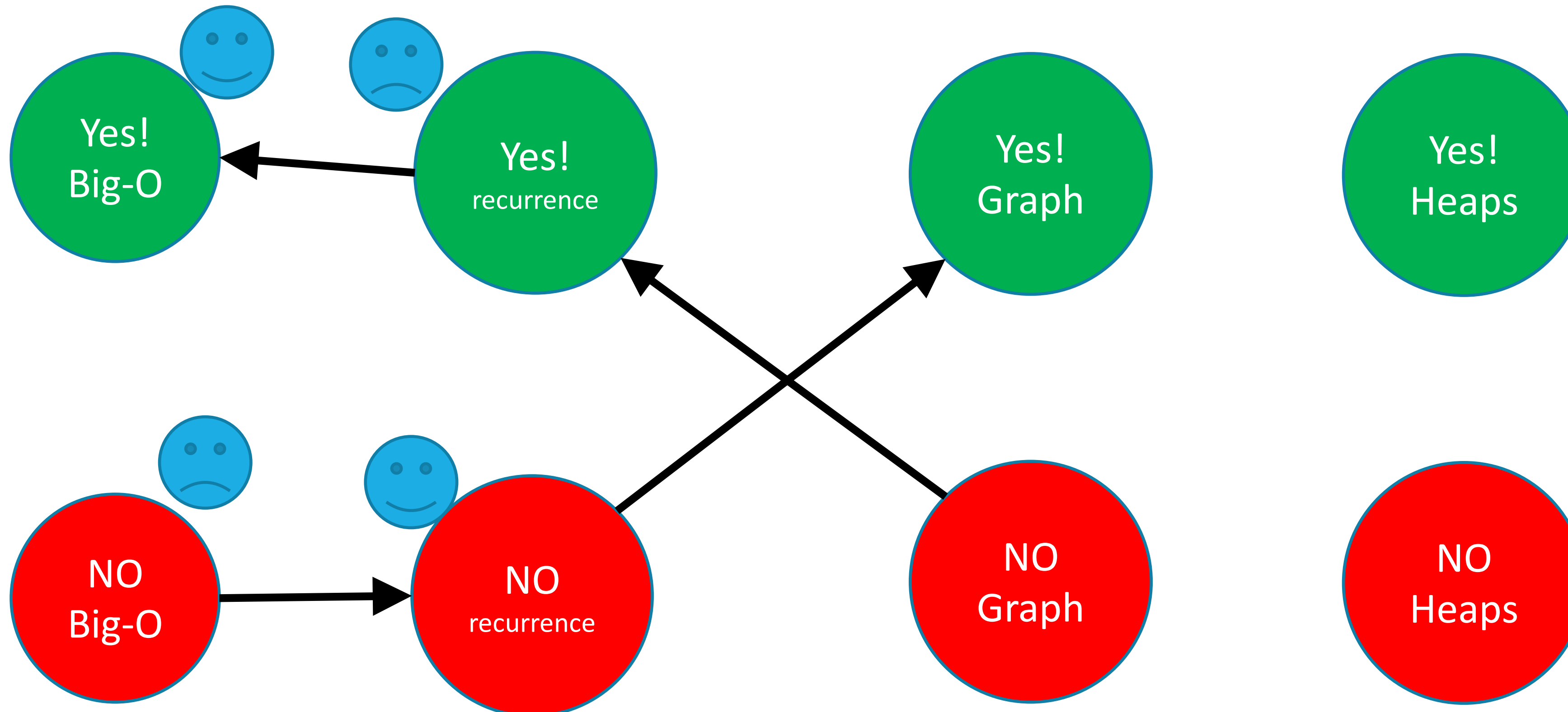
Problem	YES	NO
Big-O	X	
Recurrence		X
Graph		
Heaps		

Problem	YES	NO
Big-O		
Recurrence	X	
Graph	X	
Heaps		

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?

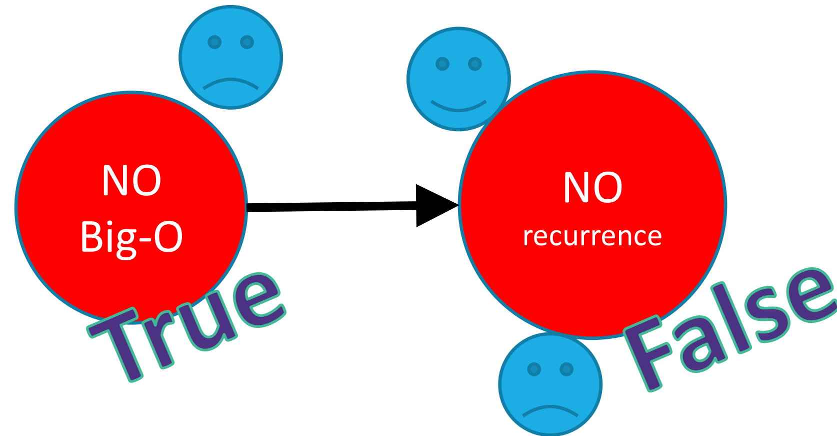
Review Creation: Take 2

Hey we made a graph!

What do the edges mean?

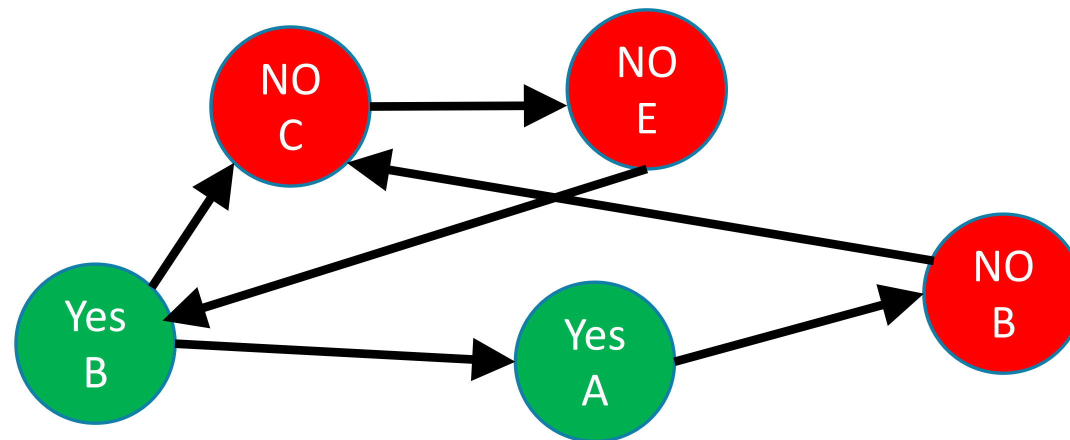
Each edge goes from something making someone unhappy, to the only thing that could make them happy.

- 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.

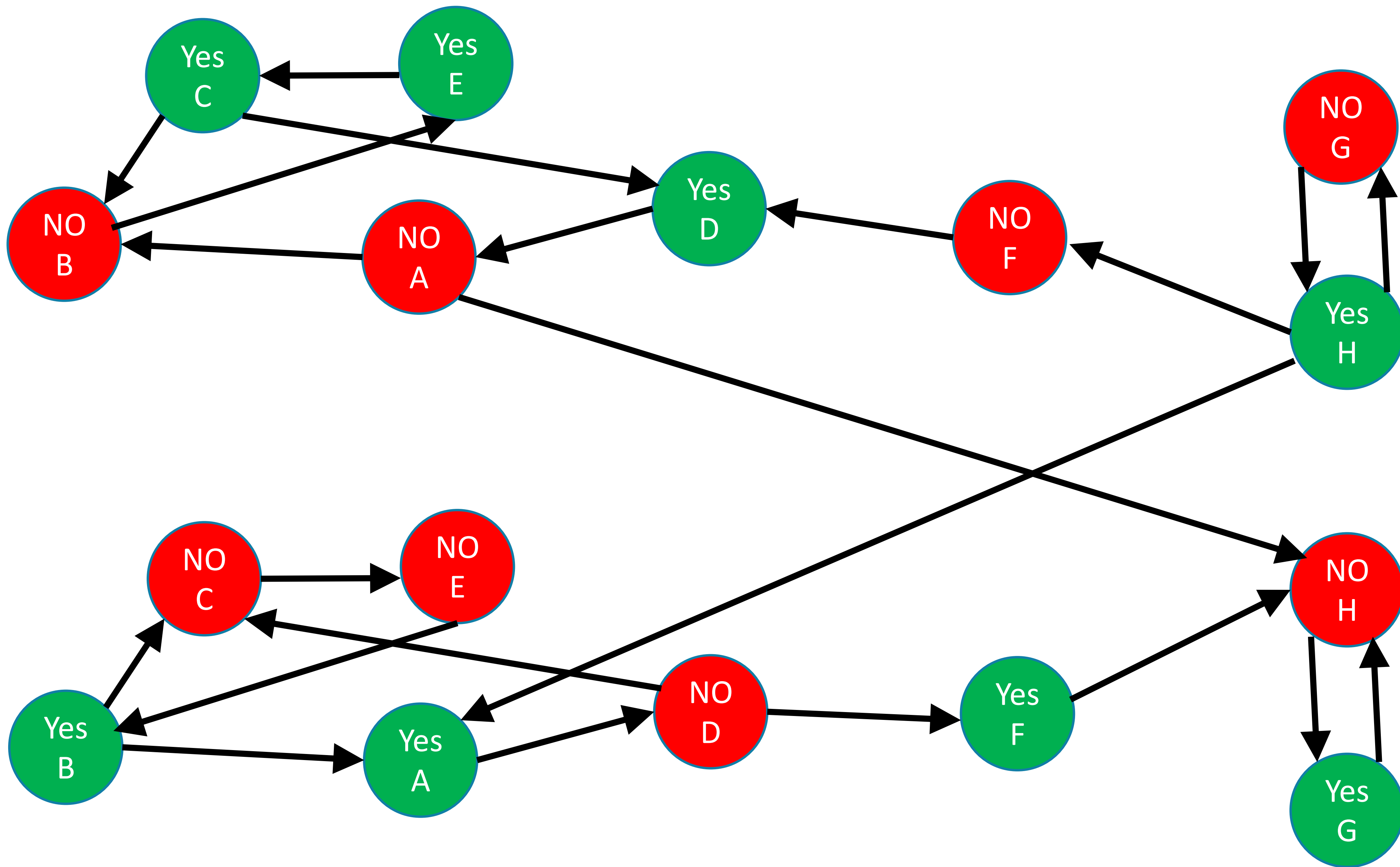
- Treat it as a single object!

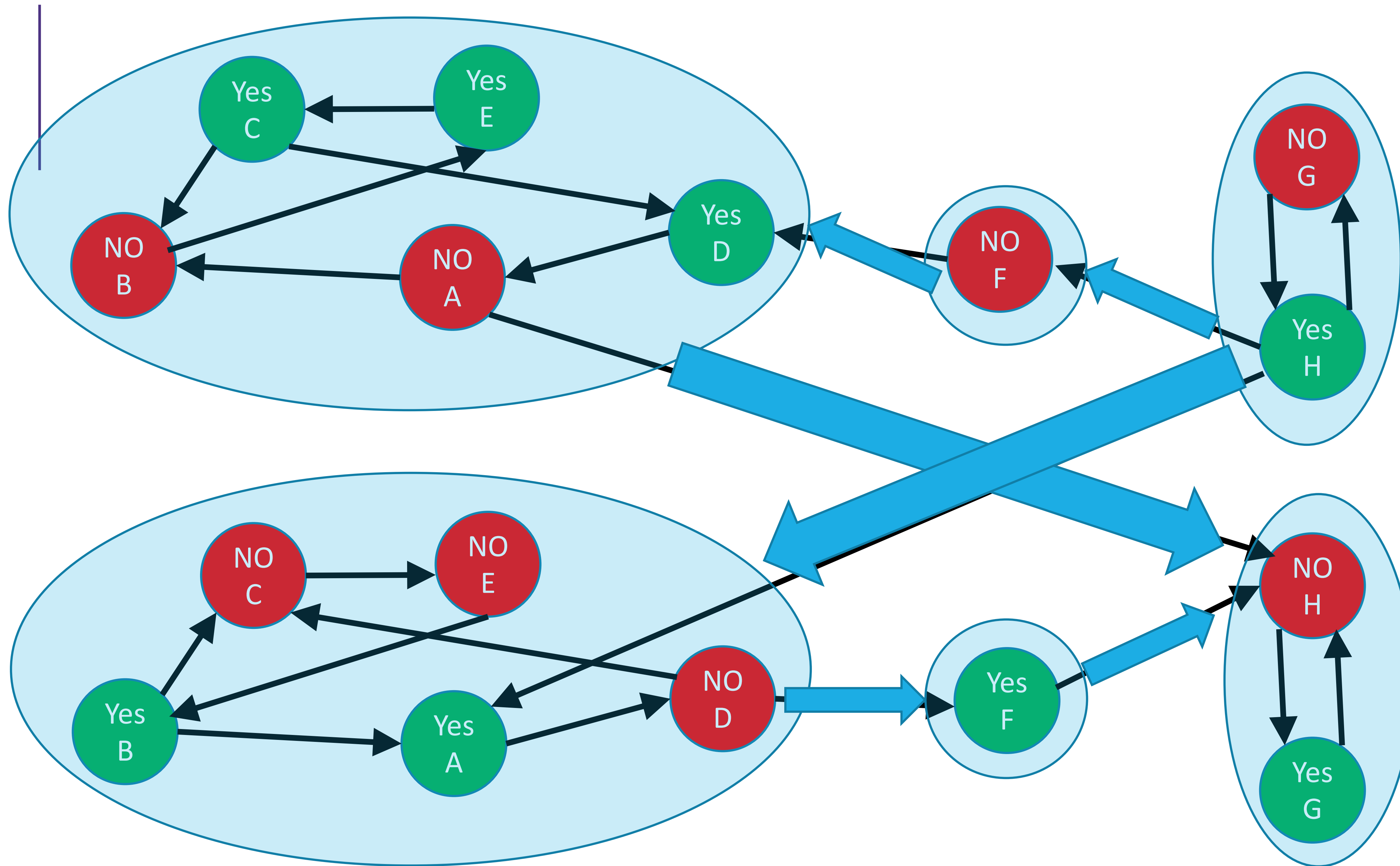
We want to avoid edges from true things to false things.

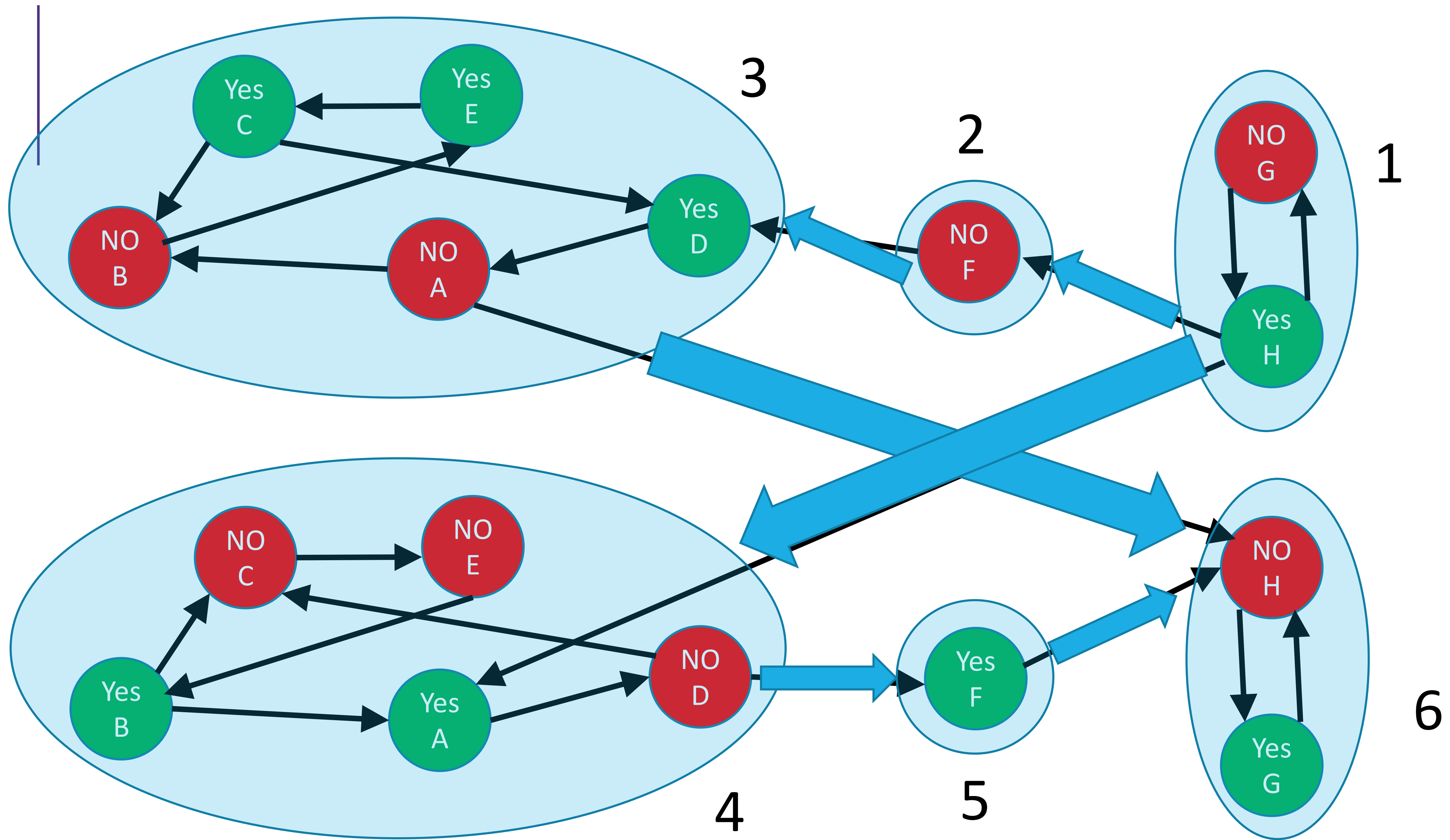
- "Trues" seem more useful for us at the end.

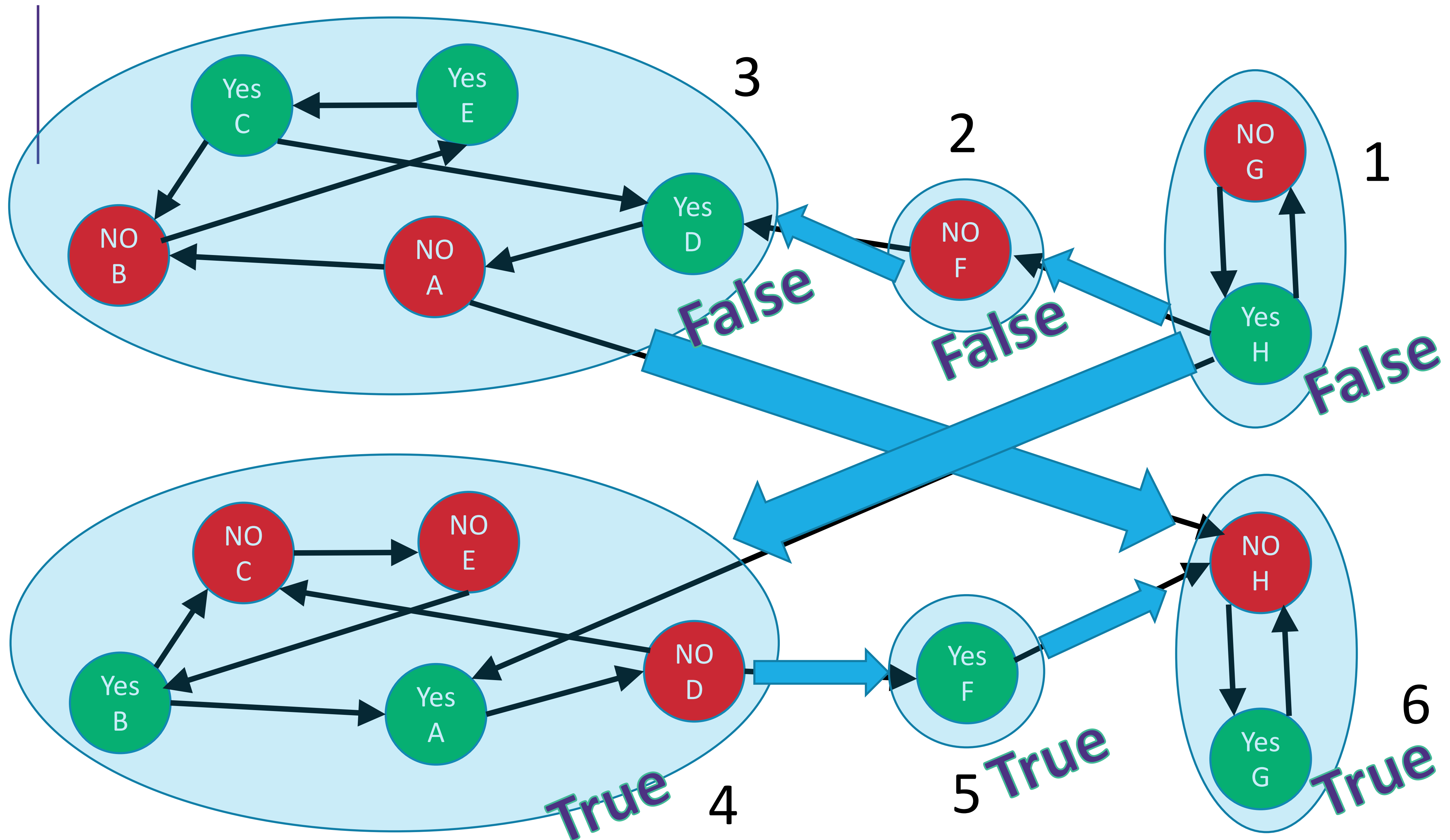
Is there some way to start from the end?

YES! Topological Sort


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


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


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



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

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.