CSE 417 Autumn 2025

Lecture 10: Directed graphs

Glenn Sun

Plan for today

Today, we'll take a deeper dive into directed graphs.

We'll talk about two important algorithms: **topological sort** and **strongly connected components**

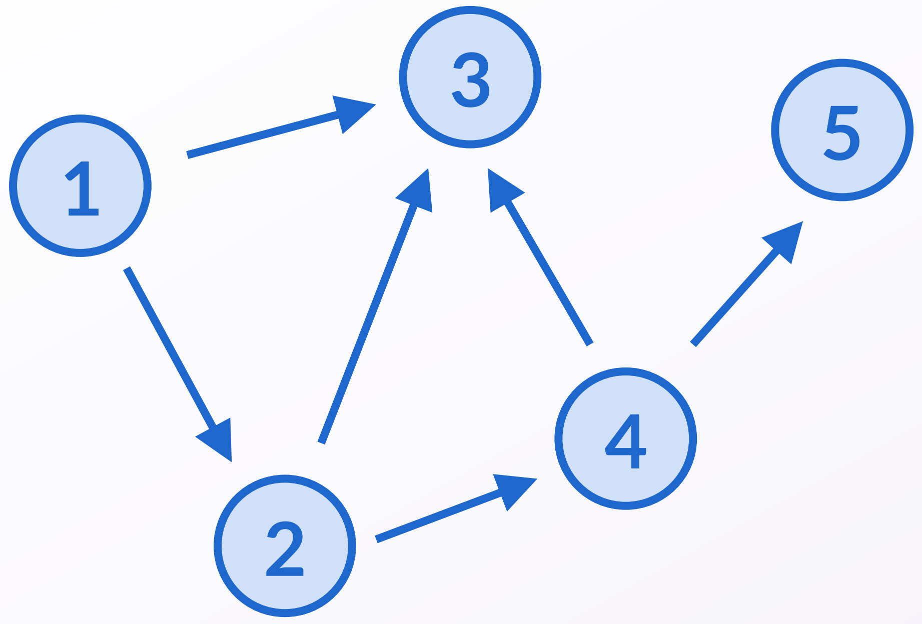
We'll apply them to decide a common problem: whether a collection of constraints can all be satisfied.

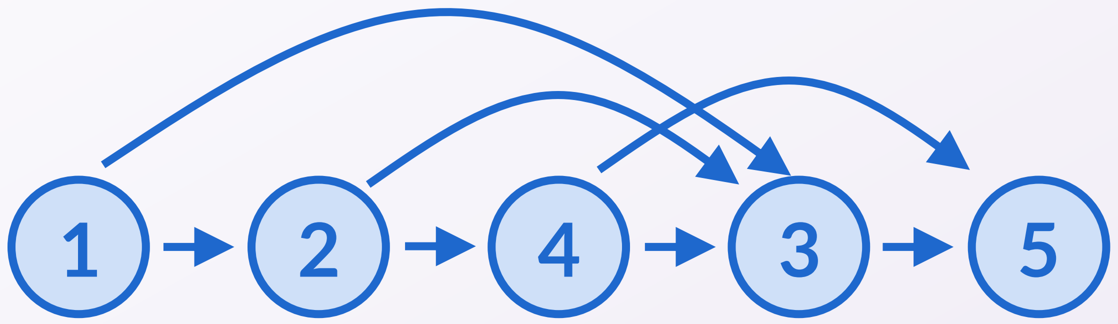
Topological sort

Prerequisites planning, again

Input: A list of courses and prerequisites

Goal: Determine if it is possible to take all courses, and **if so, a possible order to take them in.**





Topological sort

Given a directed graph with no cycles, a **topological sort** is a ordering of the vertices so that all edges point forwards.

Super easy way to find topological sort:

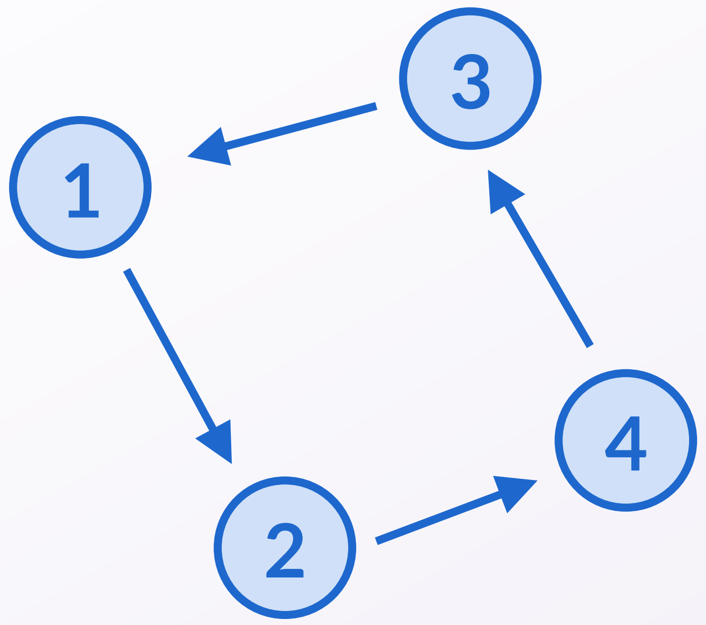
```
1. while the graph is not empty do
2.     Find a vertex with no in-neighbors, remove it, and append it
       to the output list.
```

Works well, can be optimized to take $\boldsymbol{O(n+m)}$ time.

Correctness of topological sort algorithm

Q: (soundness) Why is there always a vertex with no in-neighbors?

A: If every vertex had an in-neighbor, following them would eventually find you a cycle.

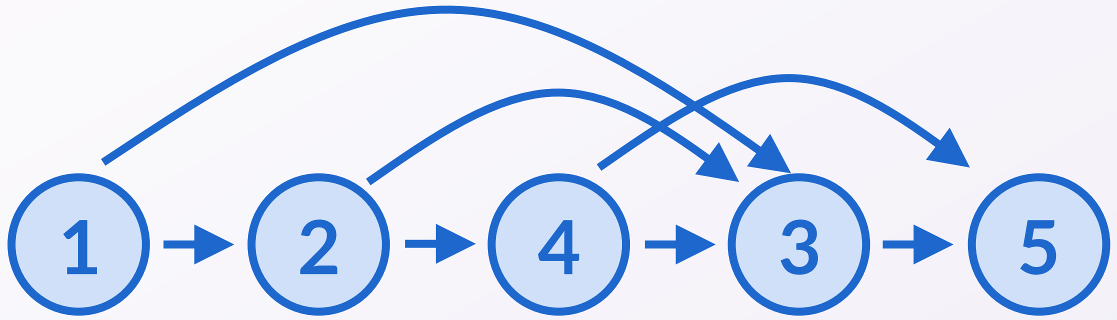


Correctness of topological sort algorithm

Q: (validity) Why do all edges end up pointing forward?

A: No out-edges from v are removed until removing vertex v, because removed vertices have no in-neighbors.

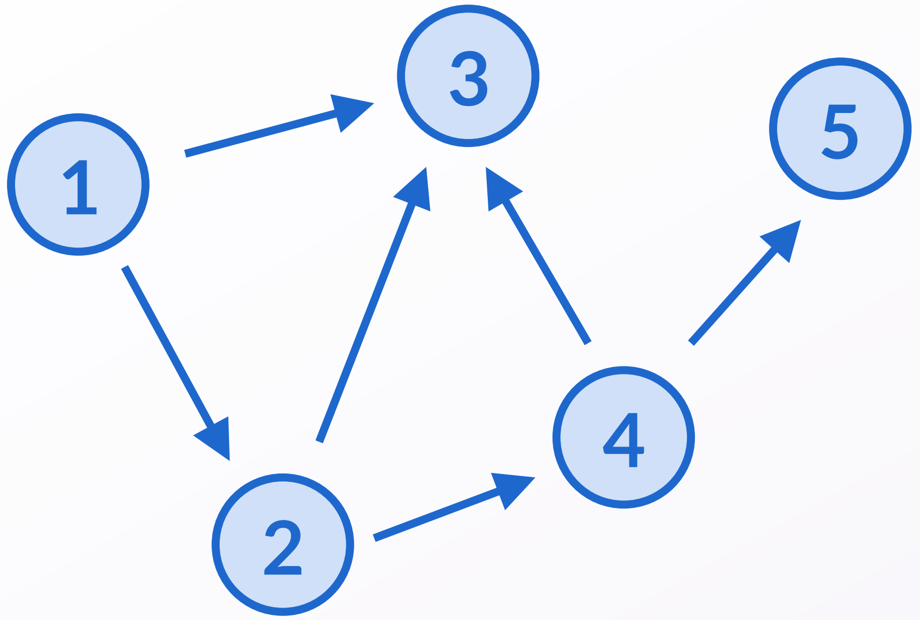
Thus, every out-edge of v is still there when it is time to remove v, and these will point to vertices that are removed later.



Topological sort via DFS

```
1.  Initialize a stack $S$ with the starting point $(s, \text{“start”})$.
2.  Mark all vertices “unstarted”.
3.  while $S$ is not empty do
4.      Get/remove the next vertex $(x, \text{action})$ from $S$.
5.      if $x$ is “in-progress” and $\text{action} = \text{“start”}$, return “cycle”
6.      else if $x$ is “unstarted” and $\text{action} = \text{“start”}$ then
7.          Add $(s, \text{“end”})$ and then all out-neighbors to $S$.
8.          Mark $x$ “in progress”.
9.      else if $\text{action} = \text{“end”}$,
10.         Mark $x$ “finished” and append $x$ to the output list
```

Topological sort via DFS



Vertices sorted by finish time are a reverse topological sort!

Topological sort via DFS

Claim. If (a, b) is an edge, then b finishes before a.

Proof. Take 3 cases for b's status when DFS looked at it from a, which must be in-progress.

Case 1: b was unstarted. Then b gets explored and $(b, \text{“end”})$ is now added to the stack after $(a, \text{“end”})$, so b finishes earlier .

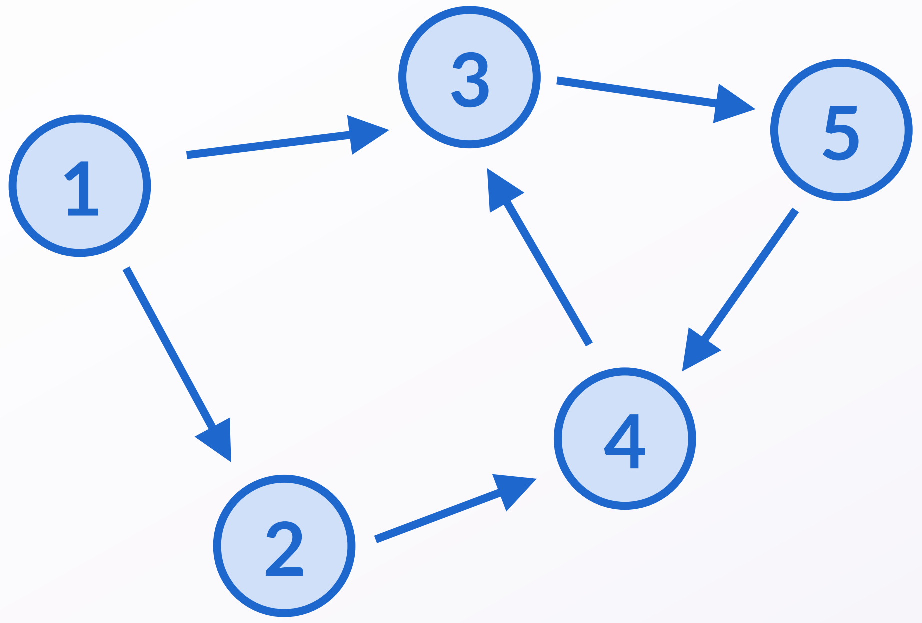
Case 2: b was in-progress. Impossible because the graph is acyclic.

Case 3: b was finished. Then obviously b finishes before a.

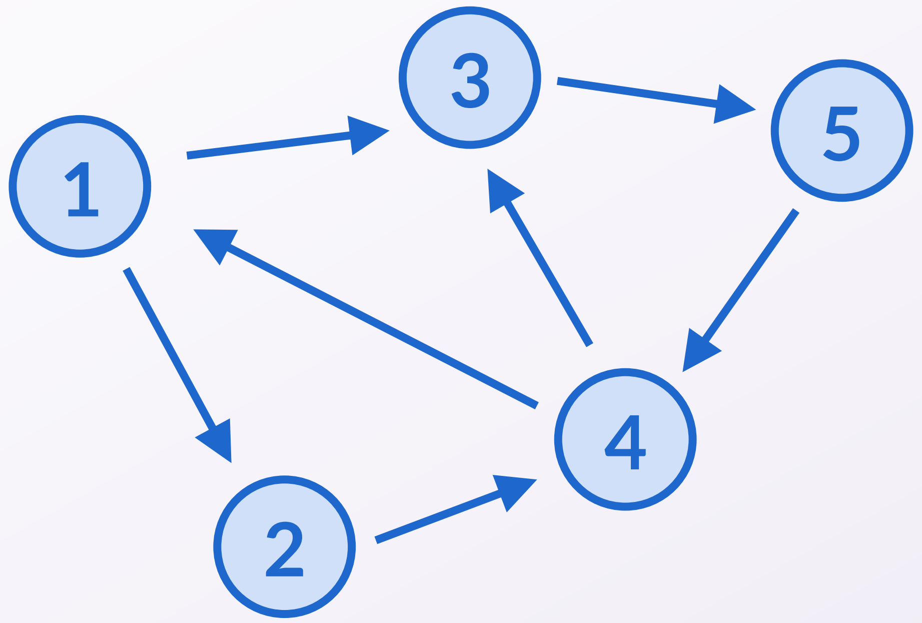
Strongly connected components

Strongly connected graphs

An undirected graph is "connected" if you can go from any vertex to any other vertex. What does this look like in directed graphs?



❌



✅

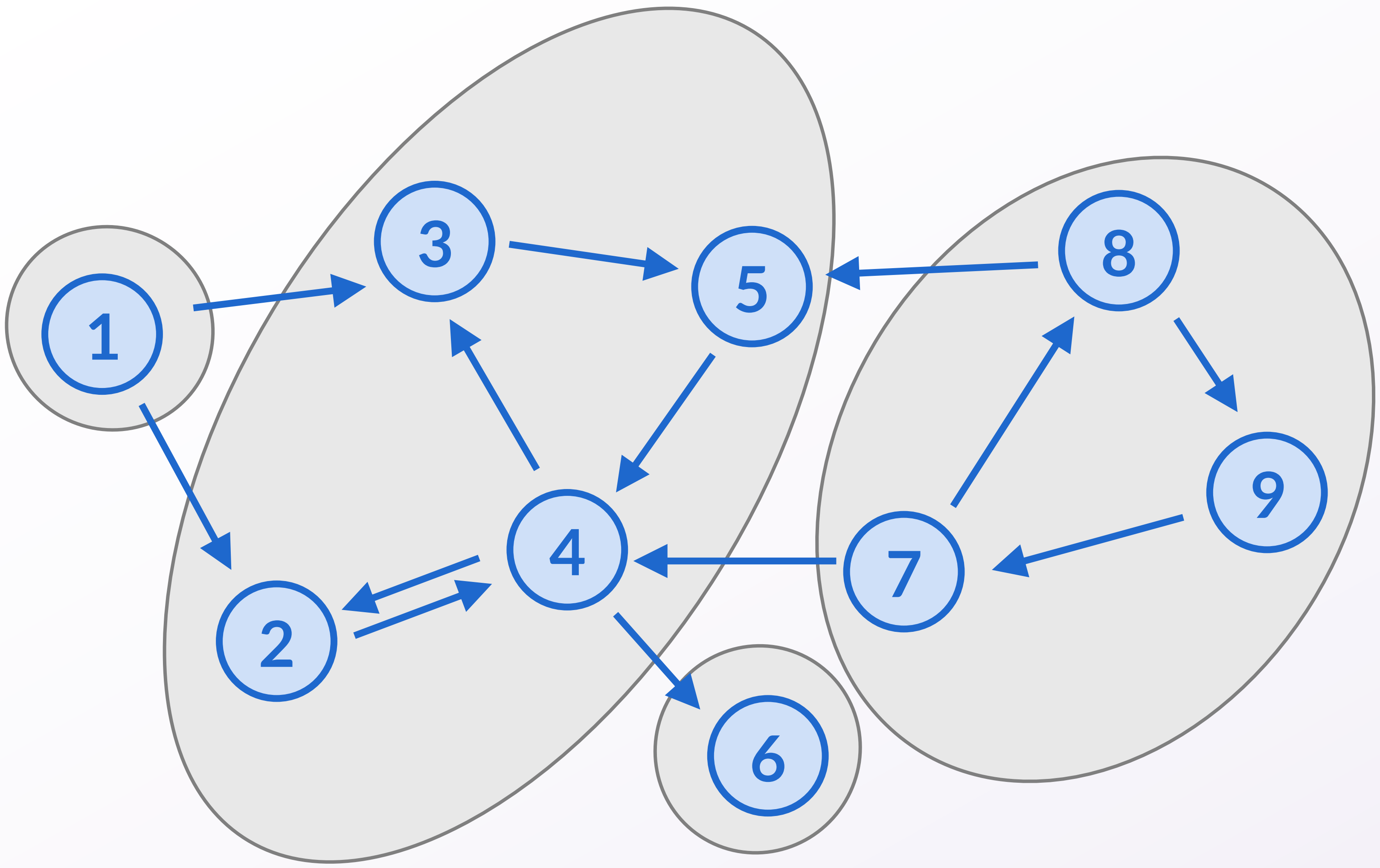
Strongly connected graphs

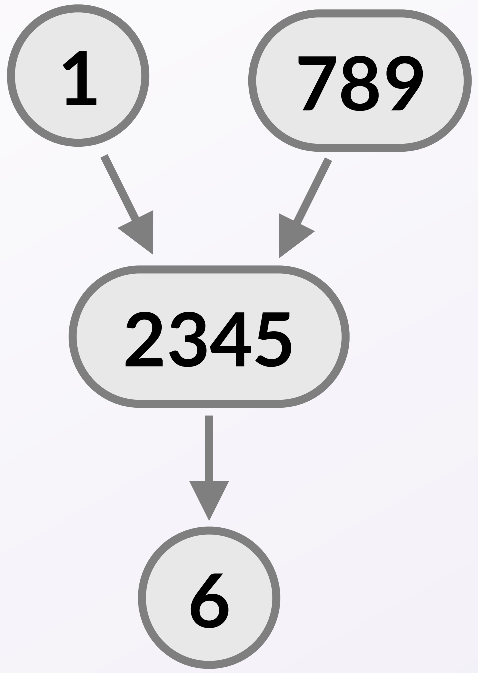
Because directed graphs can still look “connected” in the English sense without satisfying the definition, we use the phrase “strongly connected” to avoid ambiguity.

To summarize:

- A directed graph is **strongly connected** if you can go from any vertex to any other vertex.
- A **strongly connected component** is a maximal strongly connected subgraph.

Strongly connected components





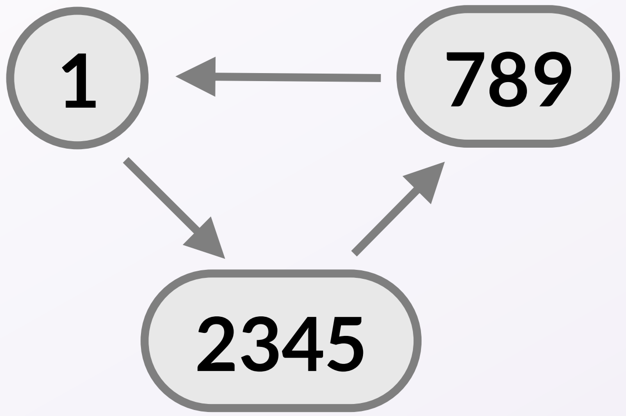
"condensation graph"

Condensation graph

The **condensation graph** has a vertex for every SCC, and an edge between SCCs if at least one analogous edge exists in the original.

Theorem. Condensation graphs are acyclic.

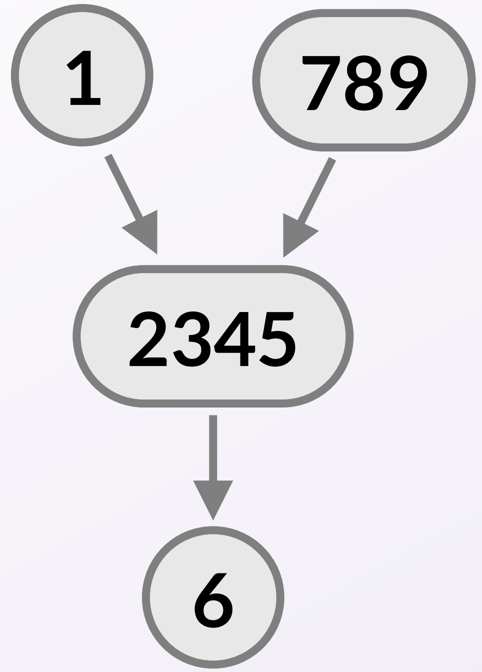
Proof. If there were a cycle, you could go from any vertex to any other vertex in the cycle, so the SCCs were not maximal.



Kosaraju's algorithm

Main idea:

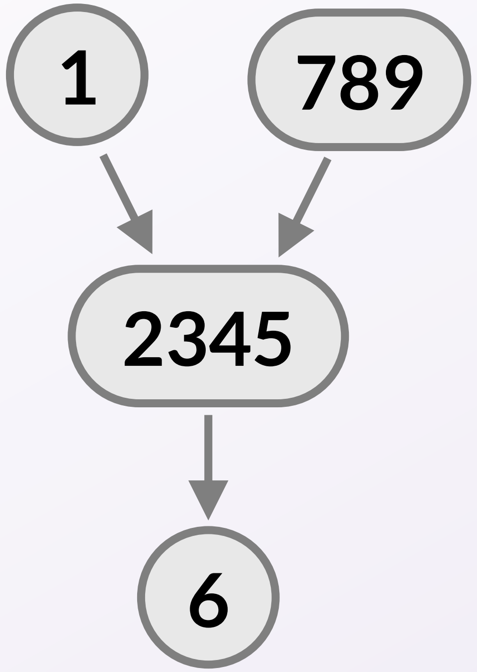
- Traverse the condensation graph with BFS/DFS in **reverse** topological order.
- We will discover all of the last SCC and then get stuck.
- Thus, every time we get stuck, output all newly discovered vertices as an SCC!



Kosaraju's algorithm

Second key idea: When running DFS with finish times, the **list of nodes sorted by finish time is *exactly* a reverse topological order of the condensation graph!**

The previous analysis for DAGs shows that the only edges that can point backwards are those that make cycles. Only edges in the same SCC can make cycles, so we are good!



Kosaraju's algorithm

```
1. Run DFS with finish times, record vertices in order of finishing.
2. Initialize an empty list of SCCs.
3. Mark all vertices as unseen again.
4. for all vertices in order of finishing do
5.       Run BFS/DFS, mark discovered vertices seen, and add this
         group to the list of SCCs.
6. return all SCCs
```

2SAT

In many applications, edges $(\boldsymbol{a}, \boldsymbol{b})$ mean "if $\boldsymbol{a}$, then $\boldsymbol{b}$".

Example with package installer:

- If $\boldsymbol{a}$ is installed, then dependency $\boldsymbol{b}$ must be installed
- If $\boldsymbol{c}$ is installed, then conflicting package $\boldsymbol{d}$ cannot be installed

Vertices: statements *and their negations*

Edges: implications $(\boldsymbol{a}, \boldsymbol{b})$ *and their contrapositives* $(\neg\boldsymbol{b}, \neg\boldsymbol{a})$

Goal: Determine if all the constraints can be simultaneously satisfied by picking each statement to be true or false.

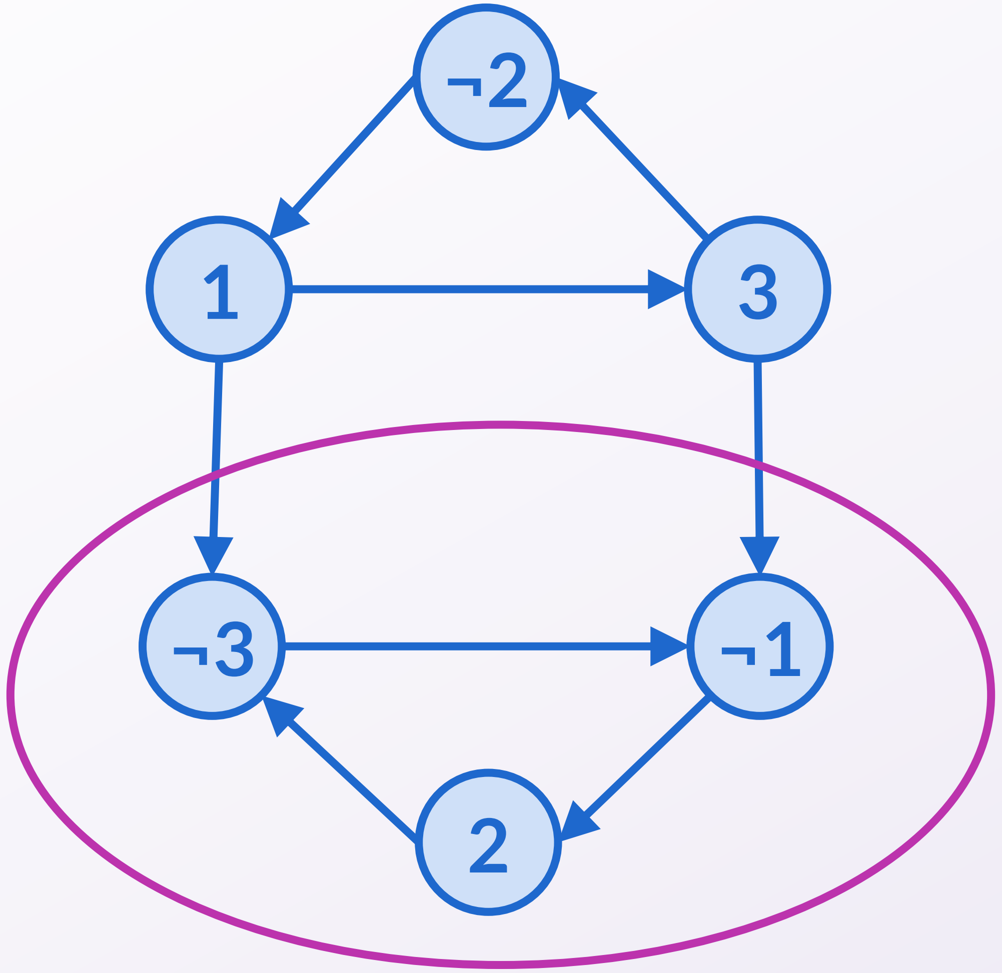
2SAT example

Four constraints (and their contrapositives):

- If 1 is true, then 3 is true
- If 1 is true, then 3 is false
- If 2 is true, then 3 is false
- if 2 is false, then 1 is true

Q: Can this be satisfied? If so, how?

A: Take 2 true, 1 and 3 false



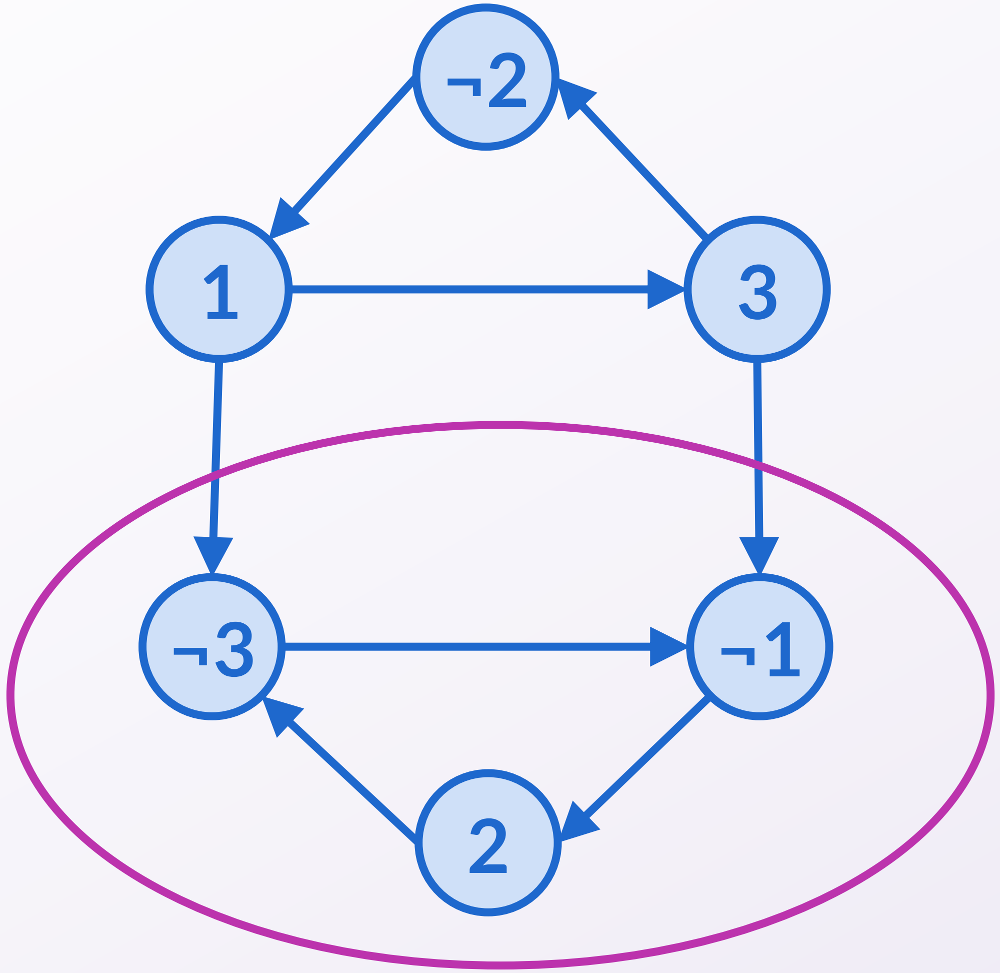
2SAT example

Four constraints (and their contrapositives):

- If 1 is true, then 3 is true
- If 1 is true, then ¬3 is true
- If 2 is true, then ¬3 is true
- if ¬2 is true, then 1 is true

Q: Can this be satisfied? If so, how?

A: Take 2, ¬1, and ¬3 all true.



Cycles in implication graphs

When edges meant prerequisites, a cycle meant "impossible to satisfy requirements".

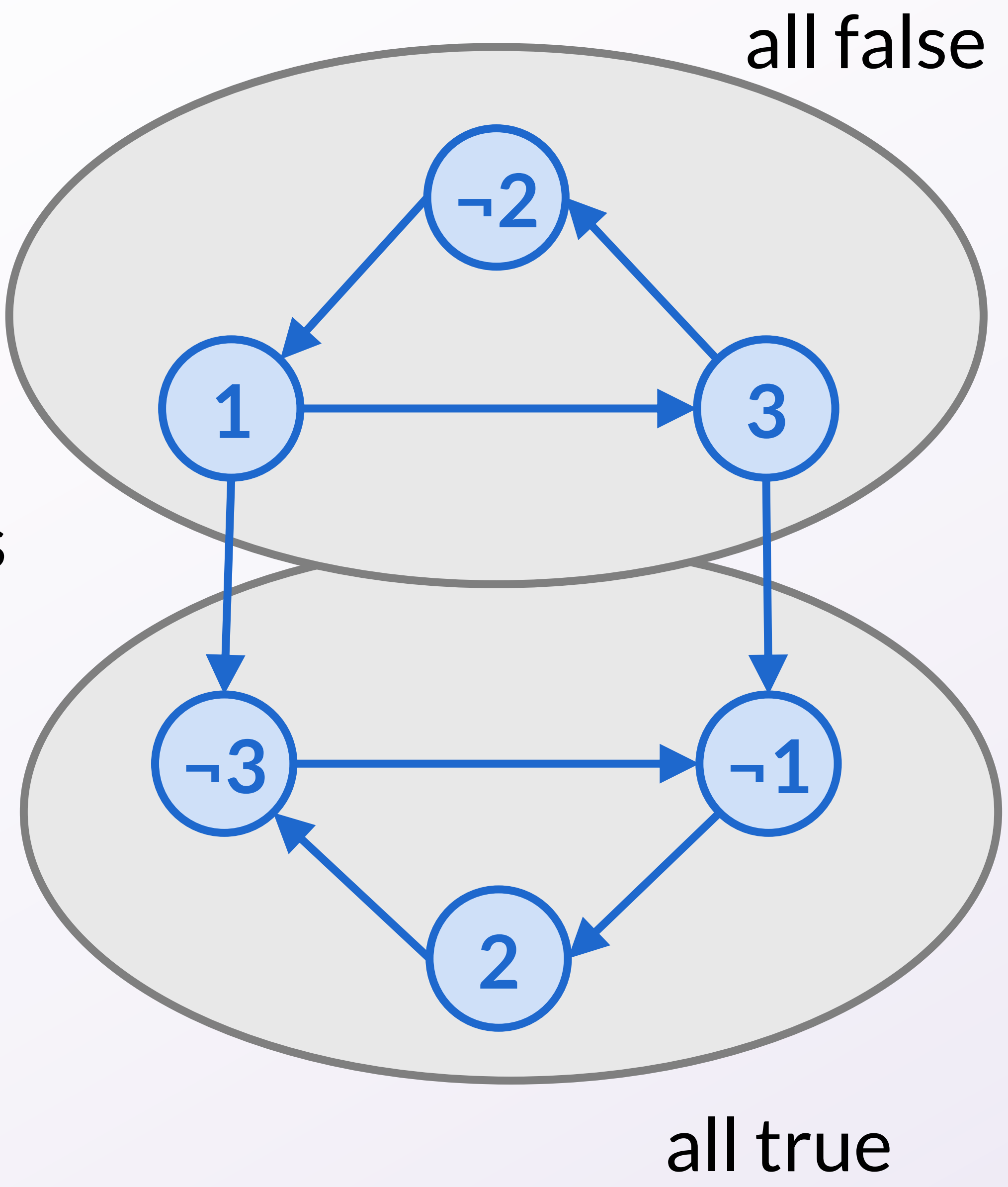
Q: When edges mean implication, in general, what does a cycle mean?

A: Either all are true, or all are false.

$$\begin{tikzcd}
\neg 3 \arrow[rr] & & \neg 1 \arrow[dl] \\
& 2 \arrow[ul] &
\end{tikzcd}$$

SCCs in implication graphs

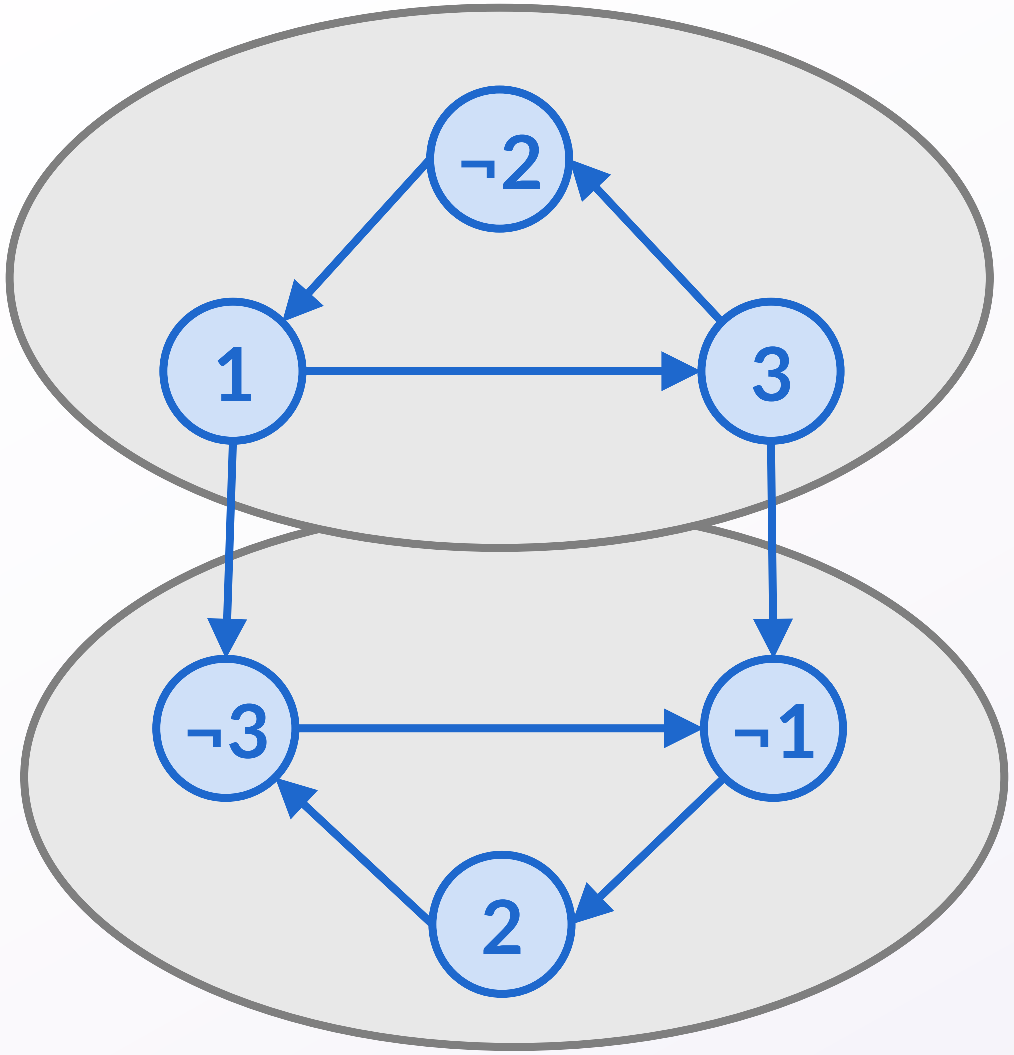
- By the last slide, every SCC must be either all true or all false too!
- Furthermore, because we included every contrapositive edge, the SCCs come in pairs.
- One SCC is all true ↔ the opposite SCC is all false



Algorithm for deciding 2SAT

```
1. Find all SCCs, and return “impossible” if any SCC contains both a
   statement and its negation (such as both 1 and ¬1).
2. for all SCCs in reverse topological order of the condensation
   graph do
3.     if the opposite SCC has not been seen yet then
4.         Decide the entire SCC to be true.
```

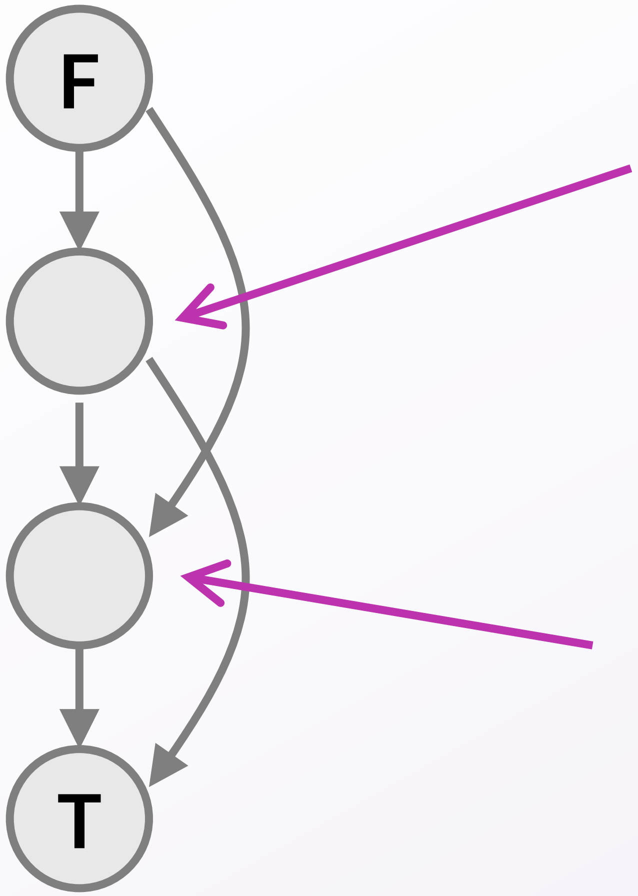
2SAT example, continued



safe to make these false, because they aren’t implied by anything

safe to make these true, because they don’t imply anything

2SAT example, continued



safe to make these false, because they're all implied by false things

safe to make these true, because they only imply things already true

2SAT algorithm correctness, more formally

Need to show:

- If the constraints are not satisfiable, we say “impossible”.
- If the constraints are satisfiable, we output a valid assignment.

2SAT algorithm correctness, more formally

- If the constraints are not satisfiable, we say “impossible”.

Contrapositive is easier: If we output an assignment, then the constraints are satisfiable. In fact, our assignment satisfies them.

- Constraints inside SCCs: These are set to “true implies true” or “false implies false”, so all good.

- Constraints between SCCs: Because we assign the start of the topological order to false and the end to true, there is never “true implies false”, so all good!

2SAT algorithm correctness, more formally

- If the constraints are satisfiable, we output a valid assignment.

We don’t output “impossible” because satisfiable constraints would never force an SCC to have both a variable and its negation.

So we output an assignment. By the same reasoning as before, the assignment must be valid.

Final reminders

HW3 due Friday @ 11:59pm.

HW1 resubmissions due tonight @ 11:59pm.

I have OH now-12:30pm:

- Meet at front of classroom, we'll walk over together
- CSE (Allen) 214 if you're coming later

Nathan has online OH 12–1pm:

- https://washington.zoom.us/my/nathanbrunelle