COMP_SCI 214: Data Structures and Algorithms

Reachability in Graphs

PROF. SRUTI BHAGAVATULA

Announcements

- Homework 3 due tonight
- Homework 4 to be released tonight
- Exam 1 solution key and explanations released
  - Grades to be released after class
  - Read companion document on Canvas **before** seeing your grade
    - Context, rationale, explanations
    - Should answer most questions you have
    - Protocol for exam-related communications (mistakes, how to communicate about the exam with me, etc.)

First exam expectations

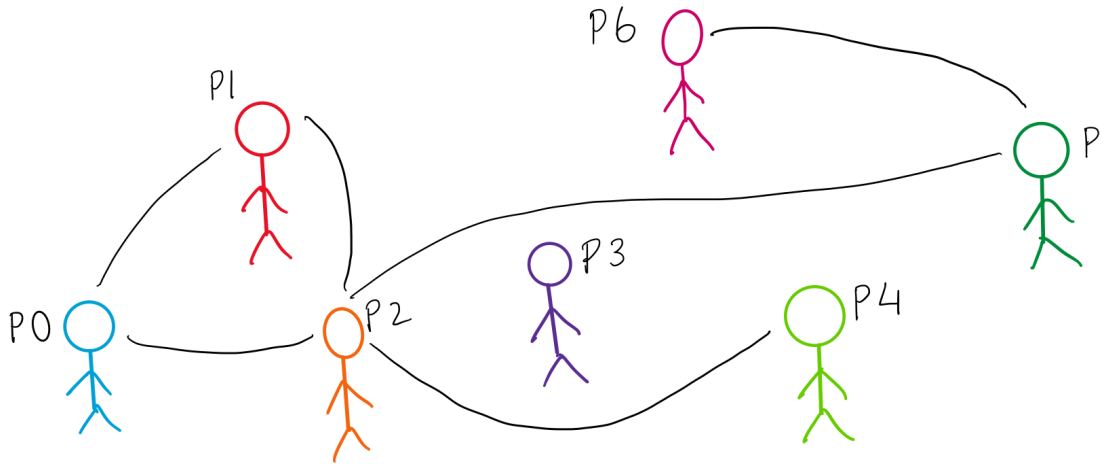
- **+0 modifier:** Good job!
  - You have met our expectations w.r.t. the theory portion of the first half of the class
  - If you get this, you're doing well
- **+1 modifier:** *Great* job!
  - You have an excellent grasp of the theory (but don't let your guard down for the rest of the quarter!)

First exam debrief

- **-X modifiers:** You may need to rethink your approach
  - You won't fail the class just because of that but it may be a warning sign!
  - I am happy to meet with you to discuss your progress or your concerns; just reach out on Piazza and I'll try to meet with you this week

Problem of the day

Below represents the interactions between people in a community during the COVID-19 pandemic. Everyone in the below community started out healthy but then they found out that P0 fell sick with COVID. Could we tell if someone else in the graph has the potential to be affected through their interactions in the network?



Connectivity in graphs

What's our goal?

1. Check if one user has interacted with P0 or interacted with people who have interacted with P0.
    1. Contact tracing!

- Let's reduce this to a general problem we can solve with graphs
    - Common approach to map real-world problems to data structures and algorithms

Goals

1. Check if a path between two vertices exists
   - Are they reachable from each other?
   - Specific to our problem: is a person is susceptible to infection?

2. Bonus: Find a path between two vertices

Goals

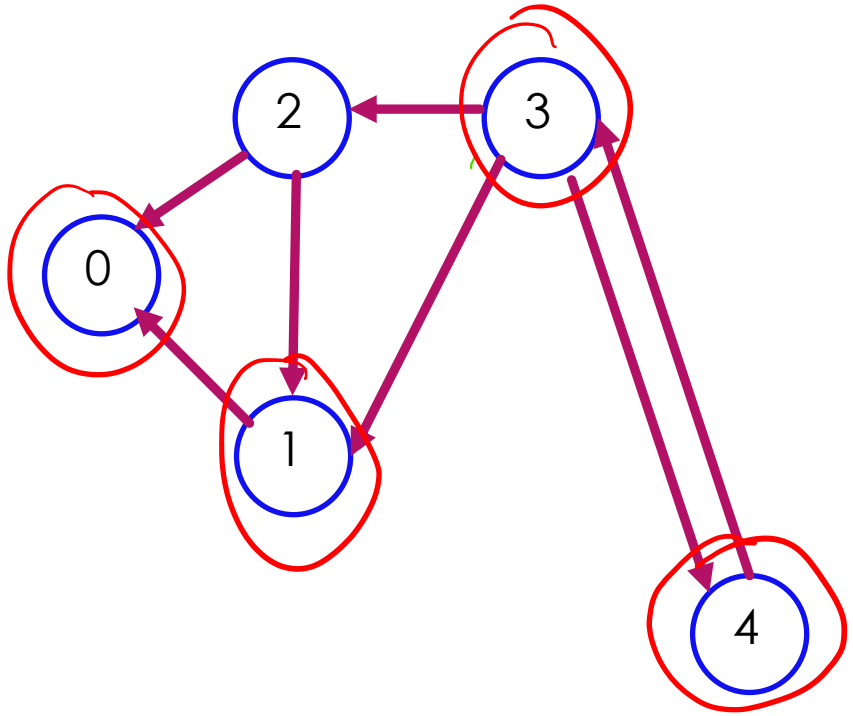
1. Check if a path between two vertices exists
2. Find a path between two vertices

- How do we check if "there is a path"?
  - Till now, by looking at the graphs
  - What about for much larger graphs?
  - Two main approaches: Depth-first search, Breadth-first search

"There is a path"

- Given a graph $g = (V, E)$ and $a, b, \epsilon\, V$
- There is a path between a and b if either:
  - $a == b$
  - There is an edge from a to $w\, \epsilon\, V$ and there is a path from w to b
- This is an inductive definition
  - Could be implemented via a recursive algorithm

Visualization of recursive definition



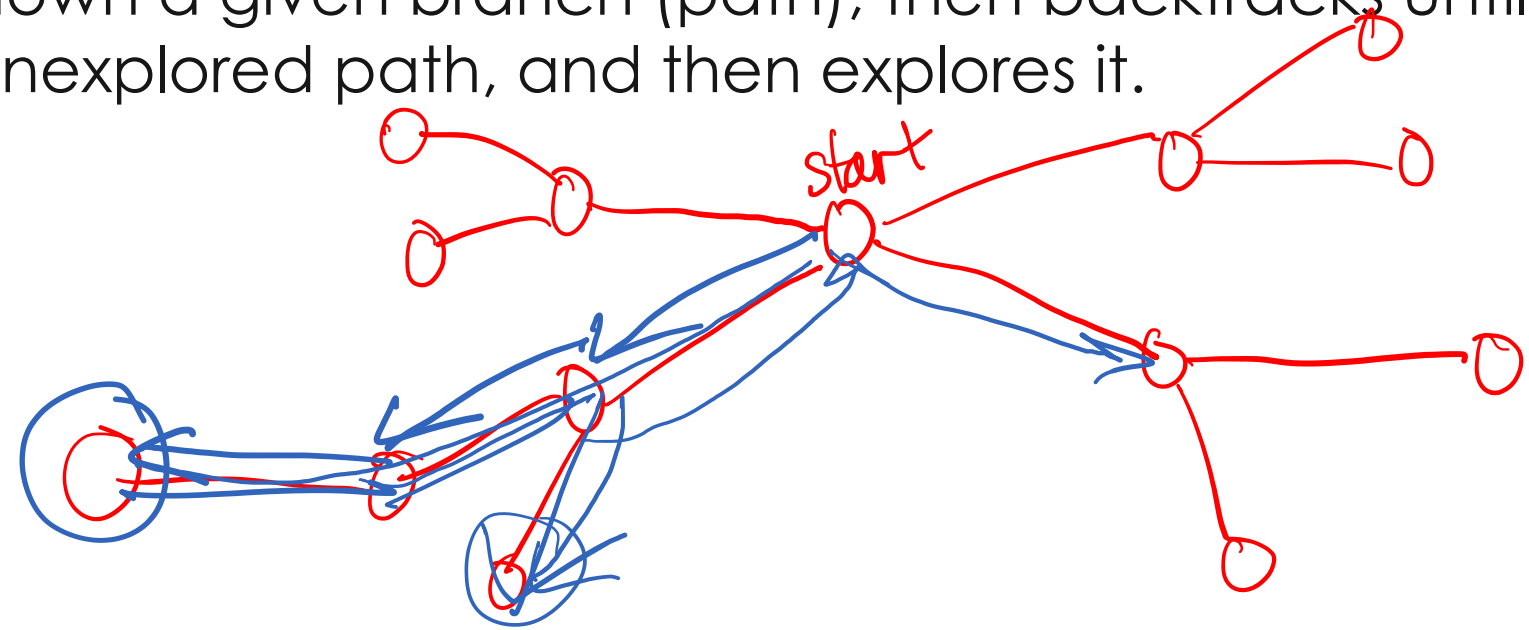
"Visiting" a node means we've arrived at a node and do something with it

Is there a path between 4 and 0?

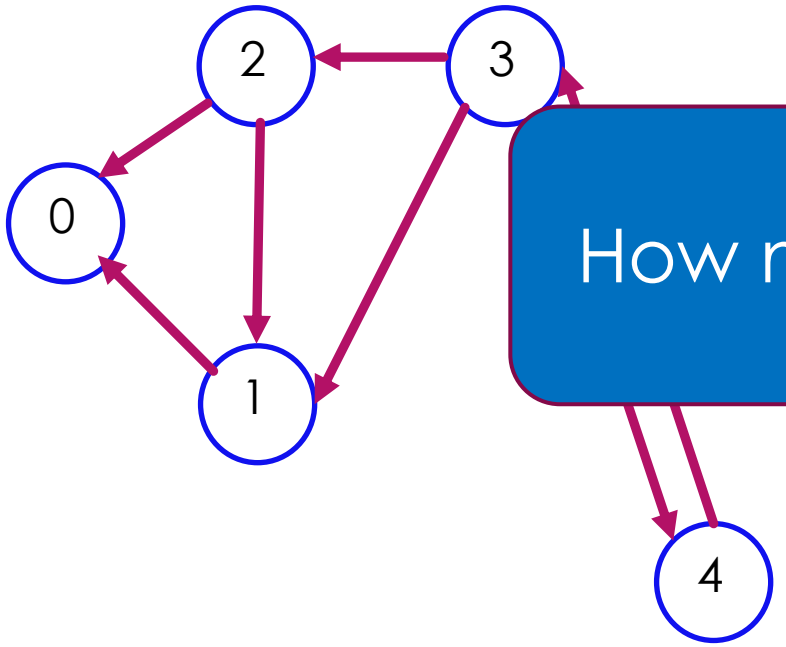
Visits to a node	Result
Visit vertex 4	4 != 0 Now find an edge v(4, ?)
Visit 4's successor: vertex 3	3 != 0 Now find an edge v(3, ?)
Visit 3's successor: vertex 1	1 != 0 Now find an edge v(1, ?)
Visit 1's successor: vertex 0	0 == 0 There is a path!

Depth-first search

- Graph traversal algorithm
- The algorithm starts at vertex of a graph and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it.



Naïve depth-first search



How might this algorithm look?

Is there a path between 4 and 0?

Visits to a node	Result
	0 find an edge v(4, ?)
	0 find an edge v(3, ?)
	0 Now find an edge v(1, ?)
Visit 1's successor: vertex 0	0 == 0 There is a path! 🎉

"Visiting" a node abstractly means through traversal we arrived at a node

Implementing algorithms today

- We'll see how everything fits into code at a high level
- But for algorithms, we'll largely think in pseudocode
  - We've seen this in previous lectures; we'll see more today
  - Pseudocode is important for outlining language-agnostic algorithms
    - E.g., everything we do in this class can easily be done in another language
  - Writing pseudocode is a great first step before writing code: outline steps in English or informal language first -> focus here is on the logic
  - Code comes after and is secondary -> focus here is just on syntax and expressions

Unweighted Directed (UD) graph

```
interface UDGRAPH:
  def add_edge(self, u: nat?, v: nat?) -> NoneC
  def has_edge?(self, u: nat?, v: nat?) -> bool?
  def get_vertices(self) -> SetC[nat?]
  def get_succs(self, v: nat?) -> SetC[nat?]
  def get_preds(self, v: nat?) -> SetC[nat?]
```

- We'll consider the UDGRAPH today
  - Everything we do today can be adapted to the other 3 types of graphs
  - You'll do this for a WU graph in your homework

Implementation setup

```
class GraphImpl(UDGRAPH):
   # Here goes data and functions about the AM or AL

let g = GraphImpl()

def reachable?(g:!UDGRAPH, u, v) -> bool?:# u and v are vertices
   # All our pseudocode is for this function
```

Naïve recursive attempt (pseudocode)

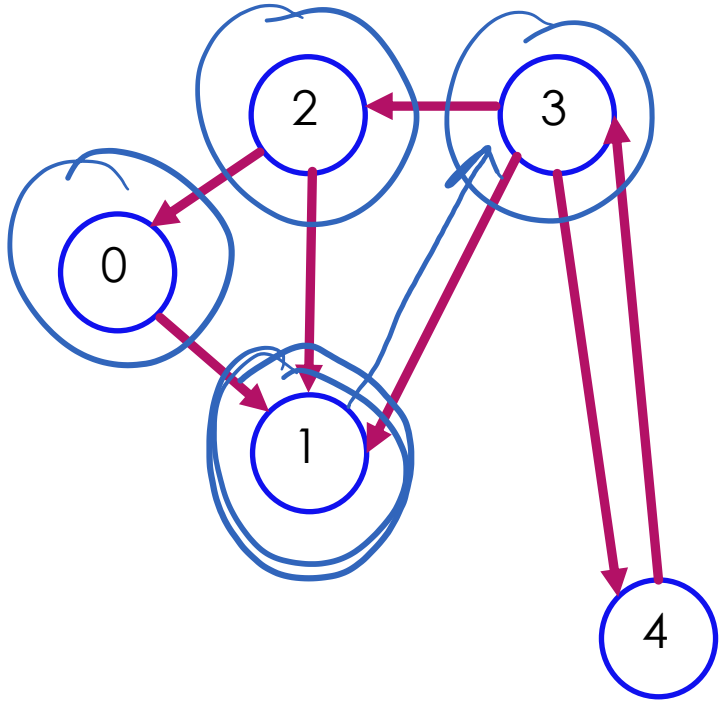
```
Procedure DFS(g, start, target):
    Procedure Traverse(w, target):
        visit(w);
        if w == target then
            return true;
        for u in g.get_succs(w) do
            if Traverse(u, target):
                return true;
        end
        return false;
    end

    return Traverse(start, target)
end
```

What if following one successor leads us nowhere? Maybe another successor will.

Instead of just picking a successor, we make sure we try to visit all a vertex's successors

Let's try this out on a graph



Is there a path between 3 and 0?

Visits to a node	Result
Visit vertex 3	3 != 0; 3's successors: 1, 2, 4
Visit 3's first successor: vertex 1	1 != 0; dead end, backtrack to 3
Visit 3's next successor: vertex 2	2 != 0; 2's successors: 1, 0
Visit 2's first successor: vertex 1	1 != 0; dead end, backtrack to 2
Visit 2's next successor: vertex 0	0 == 0; there is a path! 🎉

How would we know when to "backtrack"?

- When vertex has no successors

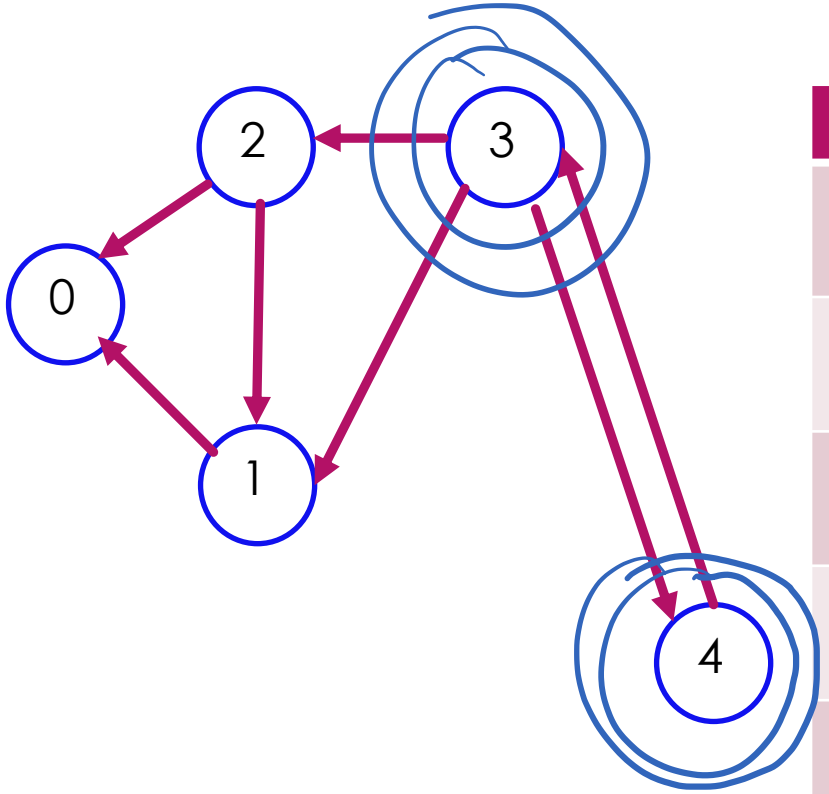
Naïve recursive attempt (pseudocode)

```
Procedure DFS(g, start, target):
    Procedure Traverse(w, target):
        visit(w);
        if w == target then
            return true;
        for u in g.get_succs(w) do
            if Traverse(u, target):
                return true;
        end
        return false;
    end

    return Traverse(start, target)
end
```

✅

But let's try that first example again



Is there a path between 4 and 0?

Visits to a node	Result
Visit vertex 4	4 != 0; 4's successors: 3
Visit 4's first successor: vertex 3	3 != 0 3's successors: 4, 2, 1
Visit 3's first successor: vertex 4	4 != 0 4's successors: 3
Visit 4's first successor: vertex 3	3 != 0 3's successors: 4, 2, 1
The above steps could keep repeating forever!	∞

Why is our approach naïve?

- We may keep visiting nodes we've already visited before!
  - Wastes time or results in infinite DFS
- Want to only visit unseen nodes

In-class exercise (5 minutes)

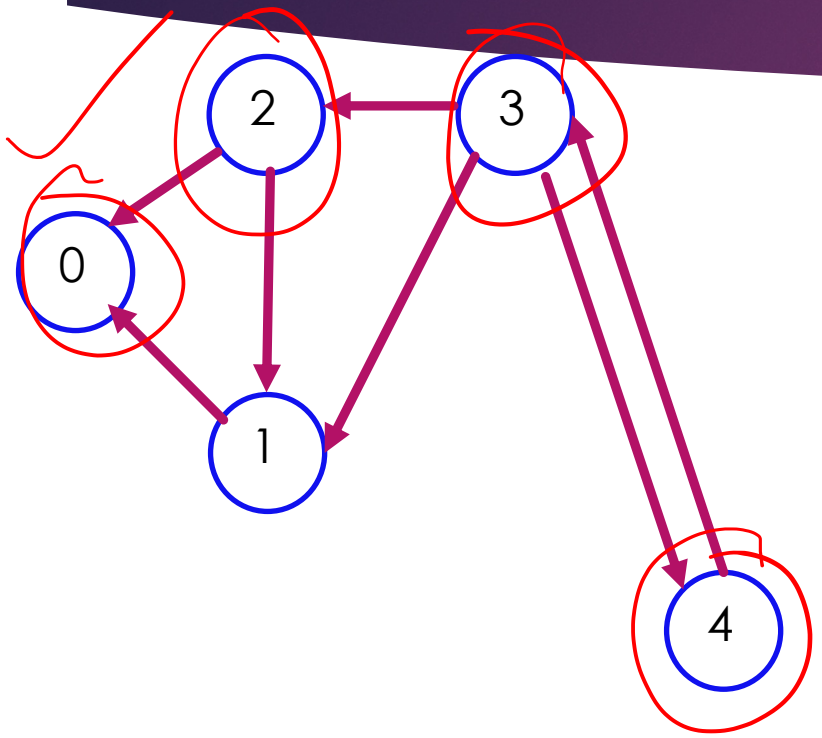
- We want to fix the problem in our naïve DFS such that we know when a node has already been visited and so we know not to do It again.

1. What is an ADT or data structure we have seen till now that we can use to keep track of and check nodes already visited? What specific data would it hold?
2. How can we use this in our DFS?

Fixing naïve DFS

- What can we do to keep track of vertices we've already visited?
  - Keep a Boolean array of vertices of size v (mark array)
  - Once visited, mark corresponding element as true
  - On recursive calls, only visit a new node if it's spot in array is false

Let's try again



0	1	2	3	4
F	F	F	F	F

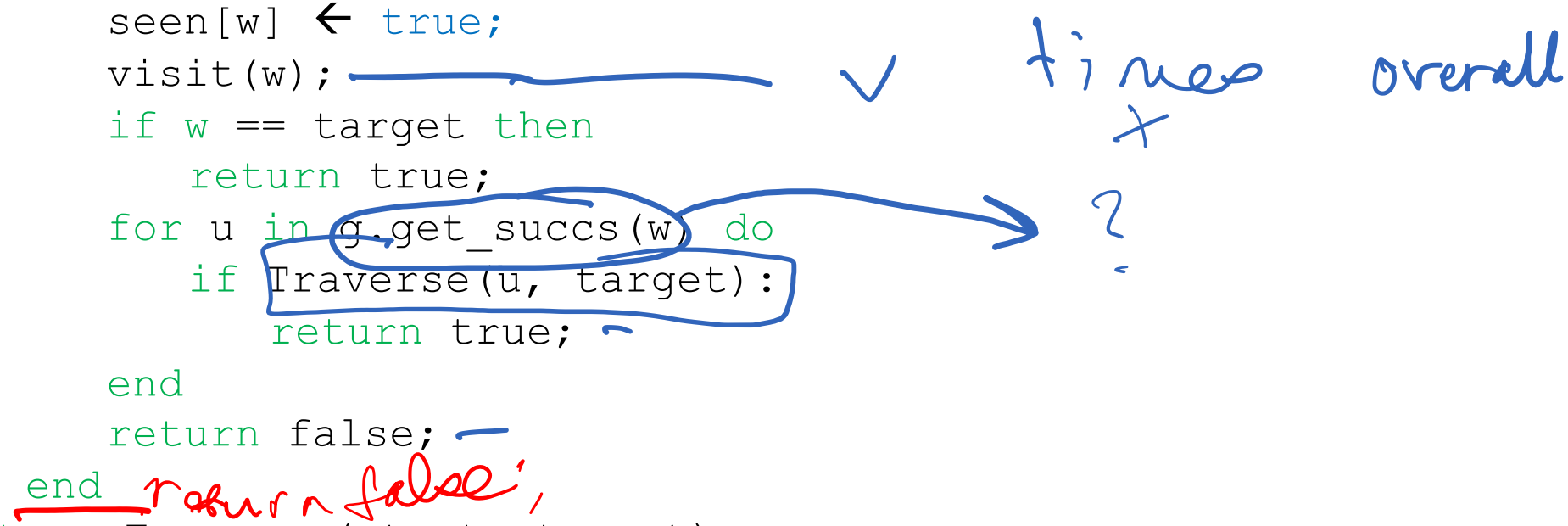
seen

Is there a path between 4 and 0?

Visits to a node	Result
Visit vertex 4	4 != 0 Set seen[4] = T 4's successors: 3
Visit 4's first unseen successor: vertex 3	3 != 0; Set seen[3] = T 3's successors: 4, 2, 1
Visit 3's first unseen successor: vertex 2	2 != 0; Set seen[2] = T 2's successors: 0, 1
Visit 2's first unseen successor: vertex 0	0 == 0 there is a path! 🎉

Better DFS algorithm

```
Procedure DFS(g, start, target):
    seen ← new array of size |V|, filled with false;
    Procedure Traverse(w, target):
        if not seen[w] then
            seen[w] ← true;
            visit(w);
            if w == target then
                return true;
            for u in g.get_succs(w) do
                if Traverse(u, target):
                    return true;
            end
            return false;
        end
    return Traverse(start, target);
end
```



Pause

- Any questions?
- Anything unclear?

Goals

1. Checking if a path between two vertices exists
   - Are they reachable from each other?
2. What is a path between two vertices?

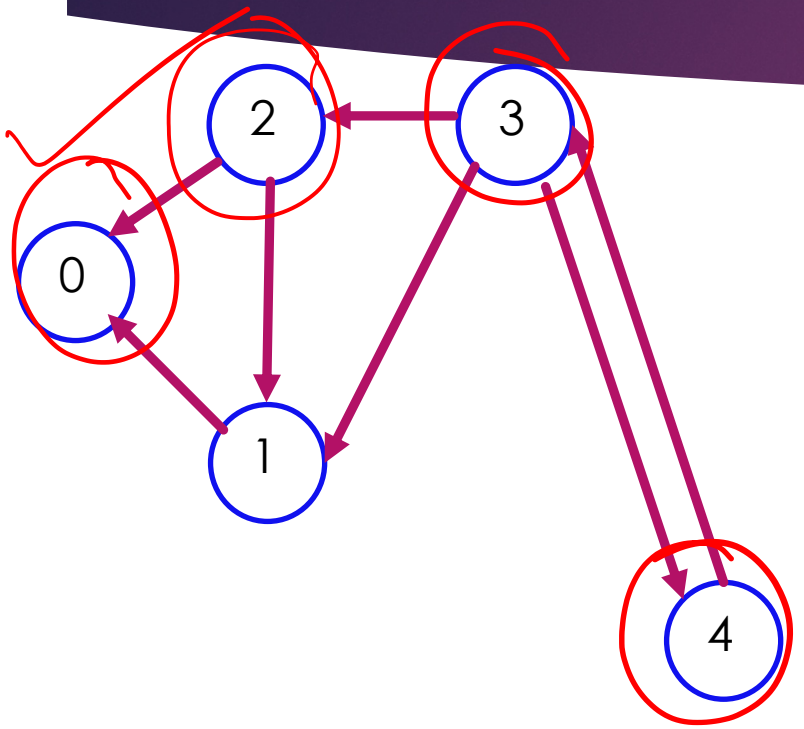
- Can we modify our DFS approach to also record a path?

Modifications to DFS to record path

- Instead of using an array of booleans (`seen`)
- Use an array of node IDs (`preds`)

- Every time we visit a node, record the vertex we visited before it in its array slot

DFS with path recording



0	1	2	3	4
-1	-1	-1	-1	4

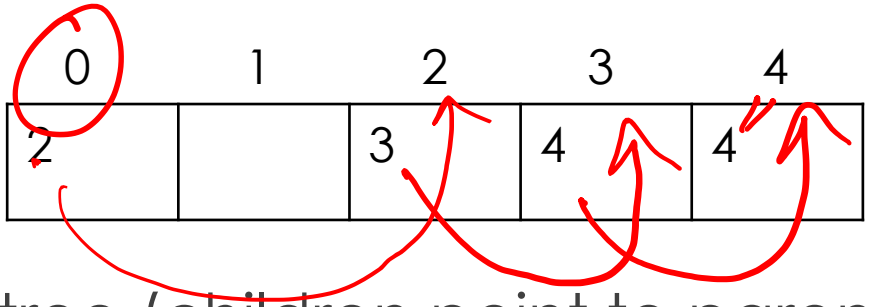
preds

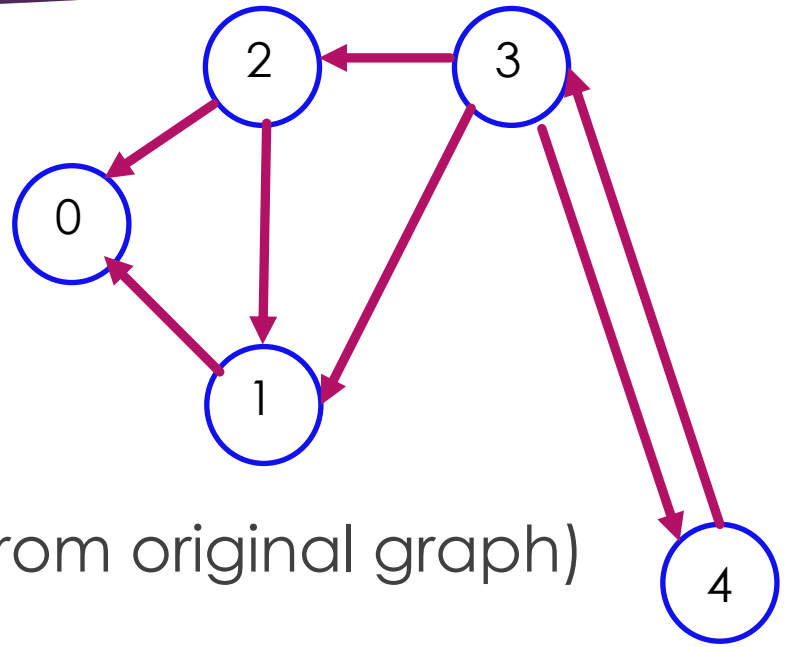
What is the path to get from 4 to 0?

Visits to a node	Result
Visit vertex 4	4 != 0 Set preds[4] = 4 4's successors: 3
Visit 4's first unseen successor: vertex 3	3 != 0; Set preds[3] = 4 3's successors: 4, 2, 1
Visit 3's first unseen successor: vertex 2	2 != 0; Set preds[2] = 3 2's successors: 0, 1
Visit 2's first unseen successor: vertex 0	0 == 0 Set preds[0] = 2 there is a path! 🎉

Getting the path from `preds`

- What is the path to get from 4 to 0?





- Build a reverse tree (children point to parent from original graph) starting from target to start indices in `preds`
  - Gives **a** path → not necessarily the most direct

Try on your own

- Modify the algorithm to do path recording

DFS time-complexity

- Visit each vertex at most once
  - $O(v)$ to visit all vertices (assuming $O(1)$ visit)
- For each vertex, loop over successors
- Total number of successors, across all vertices: $O(e)$

- Not done yet: getting and iterating over successors adds more complexity

DFS time-complexity: AL

- Visit each vertex at most once
  - $O(v)$ to visit all vertices (assuming $O(1)$ visit)
- For each vertex, loop over successors
- Total number of successors, across all vertices: $O(e)$

- Adjacency list:
  - Iterating over successors for one vertex: $O(d)$ (max degree of the graph)
  - Iterating over successors for all vertices: $O(e)$
  - **Total:** $O(v) * O(1) + O(e) = O(v + e)$

DFS time-complexity: AM

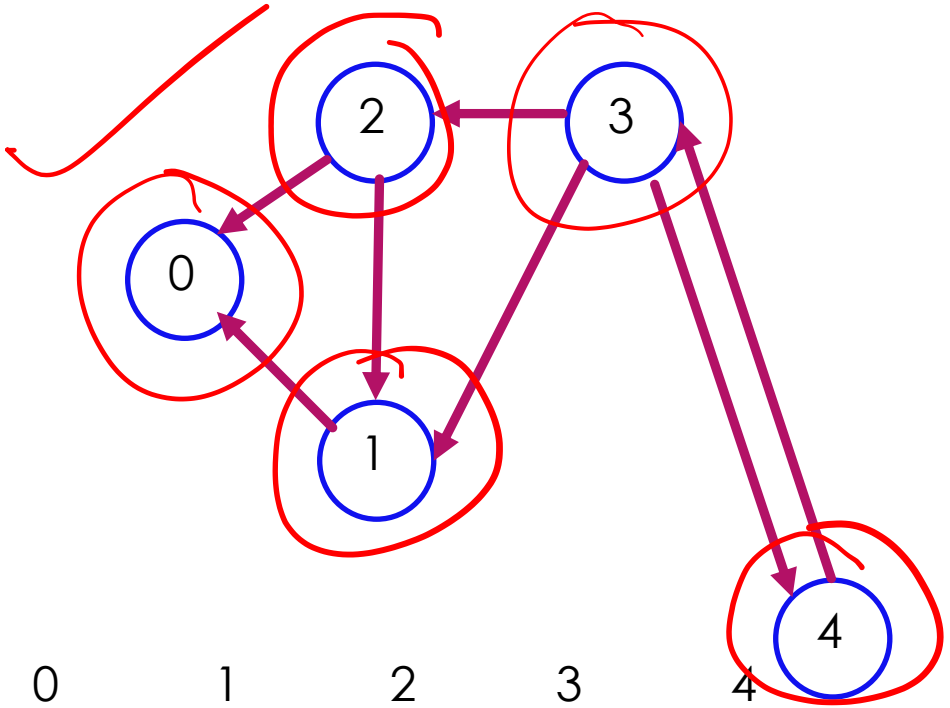
- Visit each vertex at most once
  - $O(v)$ to visit all vertices (assuming $O(1)$ visit)
- For each vertex, loop over successors
- Total number of successors, across all vertices: $O(e)$

- Adjacency matrix:
  - Iterating over successors for one vertex: $O(v)$
  - Iterating over successors for all vertices: $O(v^2)$
  - **Total:** $O(v) * O(1) + O(v^2) = O(v^2)$

DFS uses LIFO

- Depth-first search is recursive
  - It uses a stack to keep track of which node to look at next
    - What stack? The program call stack! Implicit in recursive calls
  - When we reach a dead end, we go back to the last node we saw
- Which means DFS can also be implemented non-recursively with an explicit stack!

DFS with a stack



0	1	2	3	4
F	F	F	F	F

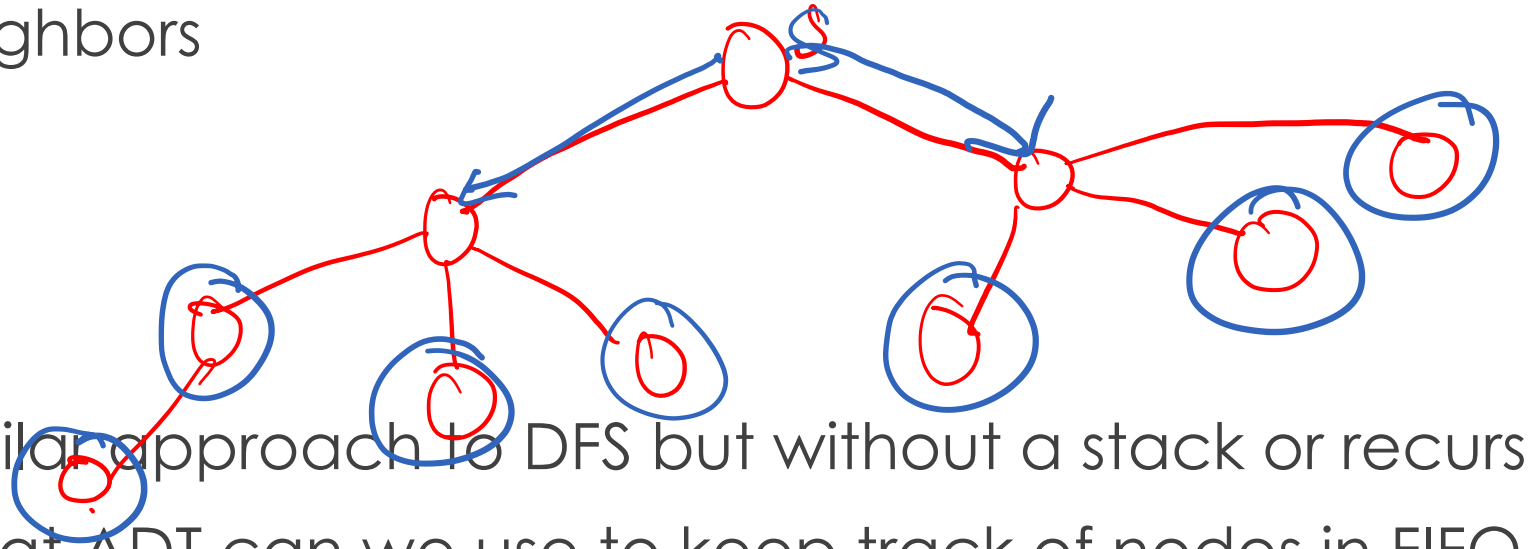
Is there a path from 4 to 0?

Non-recursive DFS with explicit stack

```
Procedure DFS(g, start, target):
    seen ← new array of size |V|, filled with false;
    todo ← new stack;
    todo.push(start);
    while todo is not empty do
        w ← todo.pop()
        if not seen[w] then
            seen[w] ← true;
            visit(w);
            if w == target:
                return true;
            for u in graph.get_succs(w) do
                todo.push(u)
            end
        end
    end
    return false;
end
```

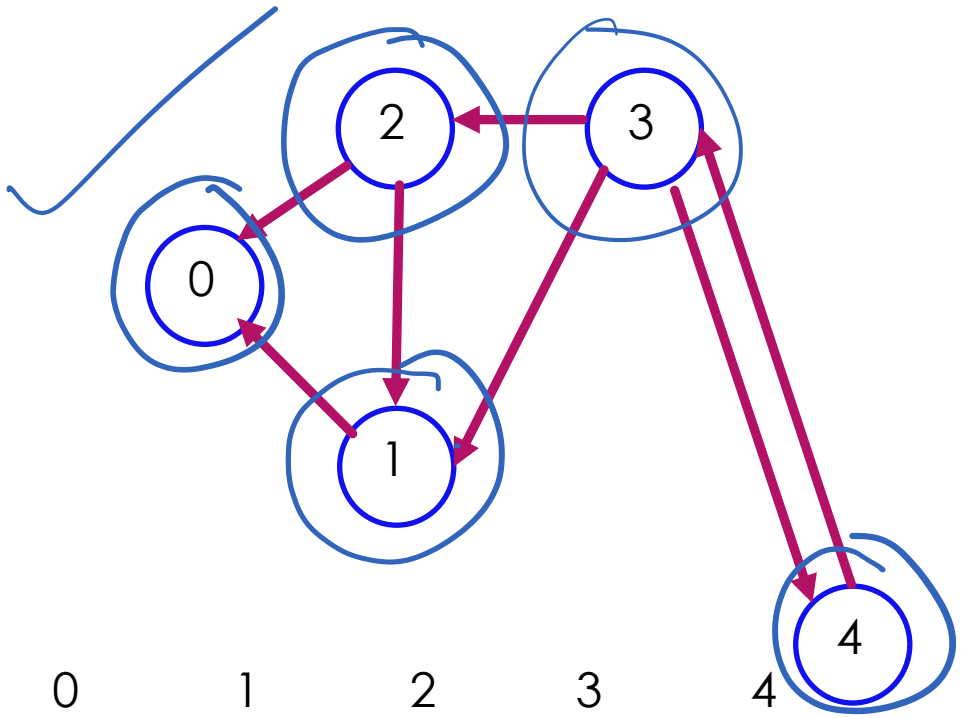
Breath-first search

- Graph traversal algorithm that uses FIFO instead
- Visits a node's immediate neighbors before moving onto their neighbors



- Similar approach to DFS but without a stack or recursion
- What ADT can we use to keep track of nodes in FIFO order?
  - A queue!

BFS using a queue



0	1	2	3	4
F	F	F	F	F

Is there a path from 4 to 0?

back 1 0 2 1 3 4 front

BFS pseudocode

```
Procedure BFS(g, start, target):
    seen ← new array of size |V|, filled with false;
    todo ← new queue;
    todo.enqueue(start);
    while todo is not empty do
        w ← todo.dequeue()
        if not seen[w] then:
            seen[w] ← true;
            visit(w);
            if w == target:
                return true;
            for u in graph.get_succs(w) do
                todo.enqueue(u)
            end
        end
    end
    return false;
end
```

BFS time-complexity

- Same as DFS in the worst case:
  - Adjacency list: $O(v + e)$
  - Adjacency matrix: $O(v^2)$
- Space complexity: $O(v)$
  - Queue can get as big as $O(v)$ in worst case

What if there is no path?

- The algorithms stop running and returns false when there is nothing left on the stack or in the queue and the target node has still not been found
- At that point they've traversed the "whole" graph

Traversing a graph

- Graph search is also for traversing a graph and enumerating all vertices
  - Take an action at every node reachable from some node v
  - Check if there are any cycles in a graph

- Just pick a starting vertex and do DFS or BFS through the whole graph until you've visited every node or until you need to stop
  - In code, remove the target check and let the stack or queue run out
  - **Unfortunately, will miss out nodes that are not reachable from starting point (by definition)**

DFS vs. BFS?

- BFS is good when you are looking for vertices close to the starting node
- DFS is good when you are looking for vertices further from the starting node