Phyll-ed Notes, Lecture 4: Graphs, DFS, SCCs

CS124 - Spring 2023

Andrew Holmes — 10 January 2023 — version 1.0

Contents

1 Graphs

1.1 Graph introduction

A graph is a data structure consisting of a set of vertices and a set of edges between vertices.

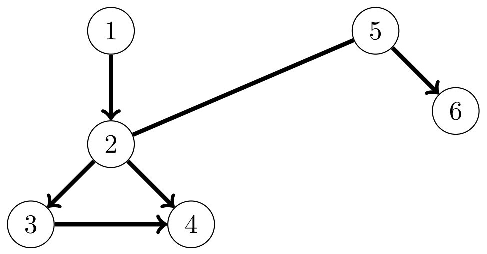


Figure 1: A basic graph

While this visual description of a graph is most intuitive to humans, computers may not appreciate my struggle with *tikz* to make pretty graphs, and thus we will generally define graphs in mathematical notation as:

$$G = (V(G), E(G)) \tag{1}$$

Thus our graph is simply a tuple of the vertices and edges that make up the graph. Breaking this down further:

- $V(G)$ is a set/list of vertices in the graph. We generally number them $\{0, 1, ..., n-1\}$.
- $E(G)$ is a set/list of edges in the graph, such as (for example) $\{(0,1),(3,5)\}$.

The edges in $E(G)$ are **directed**, such that each tuple (u, v) corresponds to an edge **from** u **to** v. For an undirected edge (an edge where we can travel from u to v and from v to u), both (u, v) and (v, u) must be in $E(G)$. A graph is *undirected* if and only if $\forall(u, v) \in E(G), (v, u) \in E(G)$. Else the graph is *directed*.

A graph's edges may also have **weights** (or lengths) associated with edges. In this case we consider the graph to have a corresponding function

$$f : E(G) \mapsto \mathbb{R} \tag{2}$$

This is simply a function that takes in an edge and outputs a real number, which is the weight associated with the input edge.

Quick check

Let's consider the graph in 1. Fill in $V(G)$ and $E(G)$ for this graph:

$$V(G) = \{1, 2, 3, 4, 5, 6\}$$
$$E(G) = \{(1,2),(2,3),(3,4),(2,4),(5,6),(2,5),(5,2)\}$$

Problem 1: Modelling problems with graphs

What are some things we might model with graphs?

Some natural real-world problems we might model with graphs include transportation problems. We could model cities as vertices and roads connecting them as edges, or airports as vertices and flight paths as edges. For both of these cases, we might also associate some extra information with each edge - for example, the expected time to complete the journey, or the cost. Another important set of problems is modelling networks (such as telecom networks). For each of these, we might be interested in finding the shortest distance between two points, the maximum traffic possible along a route or other information.

1.2 Graph representation

Problems 2 & 3: Graph representation

How can we represent a graph? How should we represent a graph?

There are two primary ways to represent a graph: the *adjacency matrix* representation and the *adjacency list* representation.

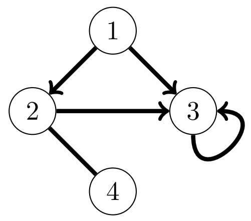
The **adjacency matrix** for a graph G with n vertices is a $n \times n$ matrix A, where $A_{i,j} = 1 \iff (i,j) \in E(G)$ and $A_{i,j} = 0$ otherwise.

The advantage of the adjacency matrix representation is that it takes constant time (just one memory access) to determine whether or not there is an edge between any two given vertices. The disadvantage of the adjacency matrix representation is that it requires $\Omega(n^2)$ storage, even if the graph has as few as $O(n)$ edges. Moreover, just examining all the entries of the matrix would require $\Omega(n^2)$ steps, thus precluding the possibility of linear time algorithms for graphs with $o(n^2)$ edges (at least in cases where all the matrix entries must be examined). In scenarios when edges have associated lengths/weights, the adjacency matrix can be modified to contain the length/weight instead of just a 1 (although there needs to be some care taken, imagine there was a 0 weight edge from u to v - this could appear as no edge if we use this idea carelessly).

The **adjacency list** for a graph G with n vertices is a length n array, where the ith element points to the adjacency list of vertex i. The adjacency list of a vertex i is a list that contains all vertices such that $(i,v) \in E(G)$ (in any order).

The total storage used by an adjacency list representation of a graph with n vertices and m edges is $O(n+m)$, since the array has length n (size $O(n)$), and the combined space of storing all of the elements within the adjacency lists must be $O(m)$, as for each edge we store one value. The adjacency list representation hence avoids the disadvantage of using more space than necessary. We will use this representation for all our graph algorithms that take linear or near linear time. A disadvantage of adjacency lists, however, is that determining whether there is an edge from vertex i to vertex j may take as many as n steps, since there is no systematic shortcut to scanning the adjacency list of vertex i. For applications where determining if there is an edge between two vertices is the bottleneck, the adjacency matrix is thus preferable.

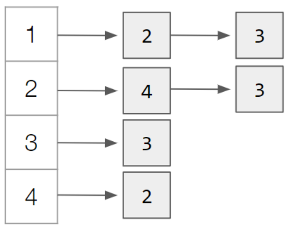
Graph representation example



Example graph

Vertex	1	2	3	4
1	0	1	1	0
2	0	0	1	1
3	0	0	1	0
4	0	1	0	0

Adjacency matrix corresponding to graph



Adjacency list corresponding to graph

Note

We will often classify graphs into certain types:

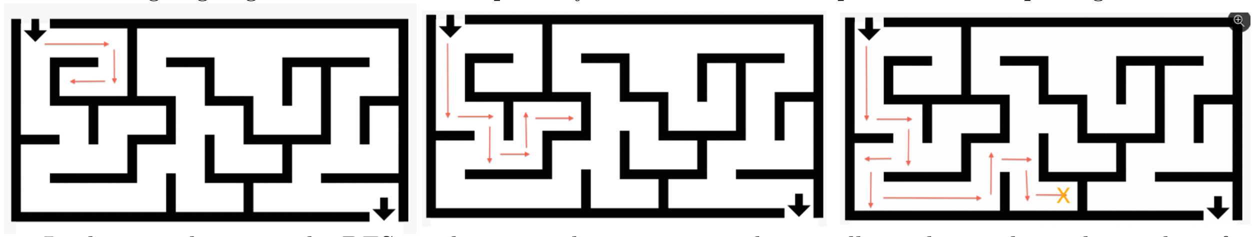
- **Directed or undirected** (see 1.1: Graph introduction)
- **Cyclic or acyclic** - does the graph contain one or more cycles?
- **Connected** - exists a **path** between every pair of vertices (not necessarily a direct edge)
- **Complete** - exists an **edge** between every pair of vertices
- **Tree** - an undirected graph where any two vertices are connected by **exactly** one path
- **DAG** - a Directed Acyclic Graph.

2 Depth-First Search (DFS)

2.1 DFS introduction

There are two fundamental algorithms for searching a graph: **depth-first search** and **breadth-first search**. We will explore depth-first search this lecture, and breadth-first search next lecture.

Depth-First Search follows edges from vertex to vertex until it reaches a vertex it has already visited or a vertex with no outgoing edges, at which point it backtracks (reverses its steps) until it reaches a vertex that has outgoing edges that it has not explored yet. One famous example of DFS is exploring mazes:



Looking at this example, DFS on this maze chooses to go right initially, and it explores this path as far as it can go, then backtracks back to the start square. Here there is another available path, so it explores down this path in the second image. At the next junction, it chooses to go right, and again explores this path until it can go no further before backtracking, and the same again in the third image.

Note

We can represent this maze as a graph, hence allowing us to run DFS! For example, the top left corner can be represented as the following graph:

$$\begin{tikzcd}
1 \arrow[rr] \arrow[dd] & & 2 \arrow[d] \\
& 4 & 3 \arrow[l] \\
5 \arrow[r] & 6 &
\end{tikzcd}$$

Our DFS started at vertex 1, then went $1 \to 2 \to 3 \to 4$. 4 is a leaf vertex/has no outgoing edges, so it backtracked to 1. 1 has another edge it can follow, so it searched that branch, following $1 \to 5 \to 6$ and so on.

2.2 DFS and stacks

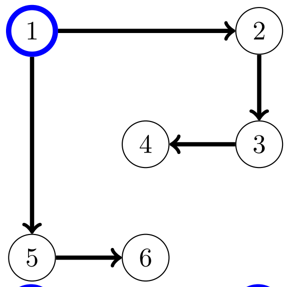
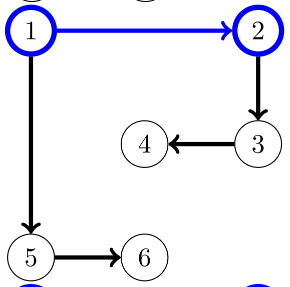
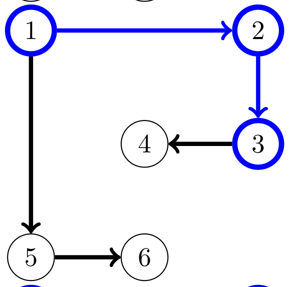
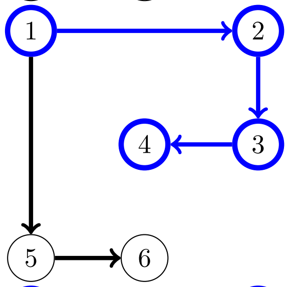
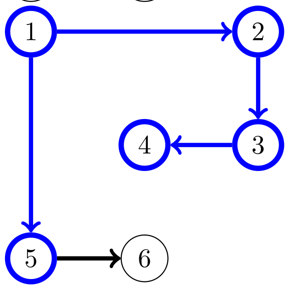
Depth-First Search uses a **stack** to keep track of which vertices to explore next. A stack is a **last in, first out (LIFO)** data structure. This means that the last item we add to the stack is always the first to be removed. We generally refer to adding to a stack as **pushing/appending** to a stack, and removing an element as **popping** an element.

For example, in the dining hall the staff create a stack of trays. The last clean tray added is at the top, and when you arrive in the dining hall, the tray you collect is the top one (the last added). The person afer you collects a tray that was added to the stack before your one. If the stack was getting low, staff might come and add several more trays, such that the tray at the very bottom might have been added first, but never gets used. In this example, it would be possible, albeit a massive pain trying to remove the first clean tray from the bottom without removing the others. People here could break the normal stack order - when programming a stack, it's common to use an abstraction barrier to prevent actions like these.

In DFS, when we find a vertex and explore its edges, we add each unvisited vertex accessible from one of these edges to the stack. We then pop (remove) one of them from the stack (the last one added), and repeat the process until our stack is empty. Let's explore an example in detail.

DFS example

Let's break down how DFS ran on the maze graph:

Diagram	Explanation	Stack at end of step
	We start with the source vertex, and set it as visited. We push the two vertices we can reach from 1 to the stack in some order, in this case 5 and then 2.	$\{5, 2\}$
	We now pop 2 from the stack and set it as visited. We push 3 to the stack since it is the only unvisited vertex we can reach from 2.	$\{5, 3\}$
	We now pop 3 from the stack and set it as visited. We push 4 to the stack since it is the only unvisited vertex we can reach from 3.	$\{5, 4\}$
	We now pop 4 from the stack and set it as visited. There is nowhere to go from 4, so we push nothing to the stack. Since only 5 remains on the stack, we will explore 5 next - this is where we encounter the idea of backtracking: we just explored 4, but now we go back to one of the outgoing edges from 1 since we have run out of paths on this branch.	$\{5\}$
	We now pop 5 from the stack and set it as visited. We push 6 to the stack since it is the only unvisited vertex we can reach from 5.	$\{6\}$

2.3 DFS pseudocode

DFS pseudocode (attempt 1)

Here's an attempt at writing some pseudocode for DFS:

```
def search(v):
    explored(v) = 1
    previsit(v)
    for (v,w) in E:
            if explored(w) == 0:
                    search(w)
    postvisit(v)
```

Problem 4: DFS pseudocode

Is this guaranteed to reach every node in a graph? When will it search all of G?

Here are a number of scenarios when this won't search the whole graph:

- The graph has multiple 'islands' where there are no edges between 'islands'. For example, consider the following graph:

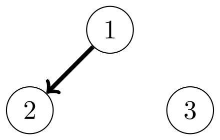


Figure 2: Problematic graph 1

No matter which vertex we start our search on, our current algorithm won't search the whole graph - there is no edge between vertex 3 and vertices $1, 2$.

- We start searching at a vertex that cannot reach all vertices (although there may be only 1 'island'):

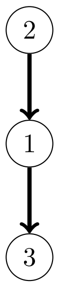


Figure 3: Problematic graph 2

Here if we ran DFS starting with vertex 1 we would never reach vertex 2.

DFS pseudocode fix

Let's fix our pseudocode for DFS. We will use the previous version as a helper function to the main DFS function to fix our problems:

```
def search(v):
    explored(v) = 1
    previsit(v)
    for (v,w) in E:
        if explored(w) == 0:
            search(w)
    postvisit(v)

def DFS(G):
    for v in V(G):
        explored(v) = 0
    for v in V(G):
        if explored(v) == 0:
            search(v)
```

Note

Looking at our pseudocode above, there is no clear implementation or usage of a stack. This is because, by using recursion, we are using the **calling stack** implicitly to handle our stack for us. search(v) is on the calling stack before search(w), and this is where the stack is hidden in this implementation. Here's an iterative implementation with an explicit stack, although without previsit/postvisit as this is trickier to do iteratively.

```
def DFS(self,s):
    for v in V(G):
        explored(v) = 0
    stack = []
    stack.append(s)
    while (stack not empty):
        v = stack.pop()
        explored(v) = 1
        for (v, w) in E(G):
            if (explored(w) == 0:
                stack.append(w)
```

2.4 DFS time complexity

Problem 5: DFS time complexity

How long does DFS take to run?

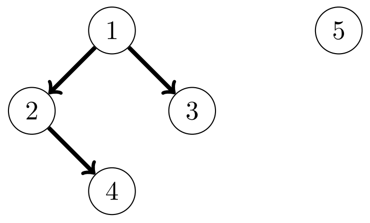
Running our DFS algorithm, we initally spend $O(n)$ time initializing the visited/explored set to zero. We then explore each vertex once, and for each vertex we explore all outgoing edges once within our search helper function. Thus the running time must be $O(n + m)$ as we use each edge and vertex only once.

2.5 Postvisit/previsit

So far we haven't discussed much what the *previsit* and *postvisit* functions do, except that we assumed they ran in $O(1)$ time for our time complexity analysis. For DFS, they simply increment a shared counter. To modify DFS' behaviour, it can be useful to change the *previsit* and *postvisit* functions, as we will see later.

Previsit/postvisit example

Let's run DFS on the following graph, assuming it picks 1 as the initial vertex and it will break ties in ascending vertex order (lowest vertex will always be picked first).



Vertex	Previsit	Postvisit
1	1	8
2	2	5
3	6	7
4	3	4
5	9	10

2.6 Types of edges

When we run DFS on a graph, we can categorize the edges of the graph into 4 categories:

- **Tree (actually forest) edges**: edges that reach a new vertex.
- **Forward edges**: edge from a vertex to non-child descendant.
- **Back edges**: edge from a vertex to ancestor.
- **Cross edges**: everything else.

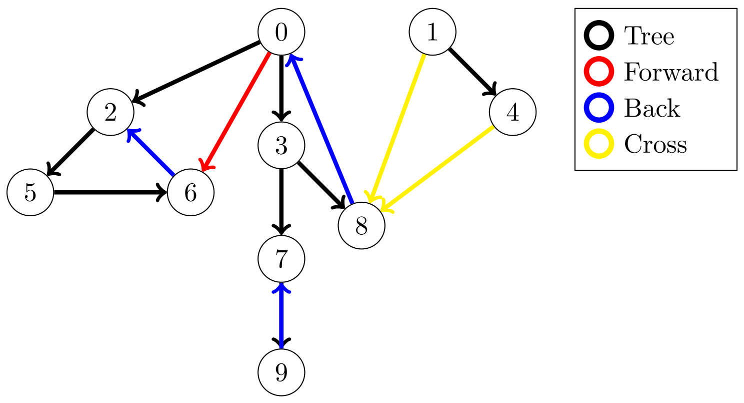


Figure 4: DFS edge type examples

2.7 DFS properties

DFS has a number of useful properties that we will use in the future:

DFS property 1

If $(u, v) \in E(G)$ then (u, v) is a back edge $\iff$ $\mathit{postorder}(u) < \mathit{postorder}(v)$.

Proof:

($\Rightarrow$) if (u, v) is a back edge, then v is an ancestor of u. In Depth-First Search, we explore all descendants of a node before backtracking, thus since u is a descendant of v we must explore and postvisit u before v, hence $\mathit{postorder}(u) < \mathit{postorder}(v)$.

($\Leftarrow$) if $\mathit{postorder}(u) < \mathit{postorder}(v)$ and $(u, v) \in E(G)$ then u must be a descendant of v. Let's assume for contradiction that u is not a descendant of v. Then once we reached u, DFS would eventually explore the edge (u, v) and reach v. Then it would explore all descendants of v before backtracking to u (or if (u, v) were a cross edge, backtrack immediately), which would ensure $\mathit{postorder}(v) < \mathit{postorder}(u)$, a contradiction. Thus since u is a descendant of v, (u, v) by definition must be a back edge.

DFS property 2

G has a cycle $\iff$ DFS yields a back edge.

Proof:

($\Rightarrow$) Let's assume for contradiction that G has a cycle but DFS does not yield a back edge. Since there exists no back edges in the graph, there can not exist an edge from any vertex to an ancestor of that vertex. But we know there exists a cycle - for there to be a cycle, there must exist some path from a vertex u that returns to u. Thus there must exist some descendant of u, v, such that there is an edge from v to u. But since u is an ancestor of v, by definition (v, u) is a back edge and we have reached a contradiction. (*Note: here we are not allowing edges from a vertex back to itself*)

($\Leftarrow$) if DFS yields a back edge, then there exists some edge $(v, u) \in E(G)$ such that u is an ancestor of v. Since u is an ancestor of v, there exists some path from u to v, and we know there is a path from v to u (namely the edge (v, u), although there may be more), thus there exists a cycle in the graph.

Problem 6: DFS & back edges

Is it possible that one DFS of a graph G yields a back edge, but another DFS yields no back edges?

Using the second DFS property, if one DFS of G yields a back edge, then G has a cycle. Thus for any other DFS, we know that if G has a cycle, DFS must yield a back edge, thus this is not possible.

2.8 Topological sort

Problen 7: Topological sort

Let's suppose you have a list of tasks where some tasks must be done before others. Is it always possible to order them?

This is not always possible. Let's consider two tasks, buying your partner chocolate and finding out what chocolate they like without asking them (or, for this example, any of their friends etc). You need to know what chocolate they like to buy them chocolate, but you seem to have no other option but to buy them chocolate to find out what chocolate they like. Both tasks require the other task to be completed before being able to complete themselves - it's a cycle, and you are stuck (it's a cruel world)! Topological sorting is **not possible** on cyclic graphs.

Problem 8: Topogical sort 2

Topological Sort: you have a list of tasks; some tasks must be done before others. How to order them?

Now assuming we have a DAG, is it possible to order/schedule them such that for any task, all prerequisite tasks are earlier in the oredering than the task (all prerequisite tasks are scheduled/completed before the task)? The answer is yes, using a topological ordering:

Definition: Topological sort/ordering

A **topological sort** or **ordering** is a linear ordering of vertices such that for all edges $(u, v) \in E(G)$, u comes before v in the ordering.

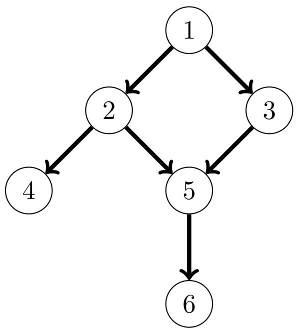
Finding a topological ordering is remarkably easy using our DFS knowledge:

Topological sort algorithm

1. Run DFS on the graph (which must be a DAG).
2. Sort vertices by decreasing postorder number.

Proof that this gives a topological sort: to prove this, we will use our two DFS properties. Firstly, by property 2, since we have a DAG, we know there are no back edges in the graph. Then from property two, we know for all edges (u, v), $postorder(v) < postorder(u)$. Thus sorting by decreasing postorder number will give vertices in an order such that for any vertex v, any vertices with incoming edges to v will be earlier in the ordering than v. In terms of task scheduling, this ensures that all tasks required by v will be completed before v.

Topological sort example



One possible topological ordering for this graph is: $[1, 2, 3, 4, 5, 6]$. However, there are many more! For example, $[1, 3, 2, 5, 6, 4]$.

3 Strongly Connected Components (SCCs)

3.1 Strongly Connected Components introduction

In an undirected graph, determining whether a graph is connected is easy: we can run DFS from a start vertex, and all vertices we reach are connected by virtue of all the edges being undirected - if we can reach two vertices a and b from u, then we can certainly reach b from a by re-tracing our steps from a to u, and then following our path from u to b. A set of vertices where there exists a path from any vertex to any other vertex in the set in an undirected graph is known as a *connected component*.

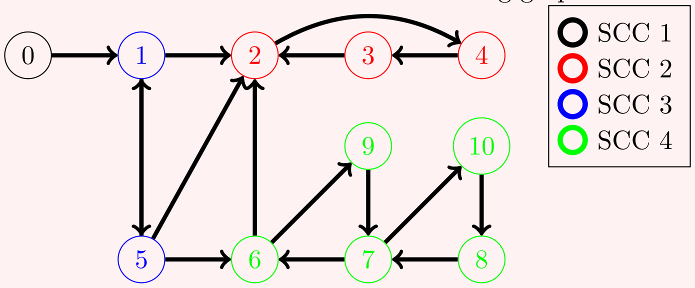
For a directed graph, things are a little more complicated. We say two vertices u, v are *connected* if there exists a path from u to v and from v to u.

Definition: Strongly Connected Component

A **Strongly Connected Component** (SCC) is a maximal vertex set such that every vertex in the set has a path to every other.

Problem 9: Identify SCCs

What are the SCCs of the following graph?



Problem 10: SCC graph as a DAG

Suppose you make a new graph G' whose vertices are the SCCs of G. Prove that G' is a DAG.

Proof: Let's assume for contradiction that G' is not a DAG - i.e there exists a cycle (if it's undirected, any edges would imply a cycle, so just we just assume a cycle to cover both). If there exists a cycle between vertices in this graph, then we can traverse between two SCCs in either direction. But if we could reach any vertex in an SCC from any other vertex in the SCC, and we can there is an edge between the two SCCs in both directions, then we can every vertex in SCC #1 must also be able to reach every vertex in SCC #2 and vice versa. We have then reached a contradiction - the two vertices here cannot be strongly connected components, as they are not **maximal** sets such that every vertex in the set has a path to every other: we could make a bigger set by combining the two.

3.2 Finding SCCs

To help us with an algorithm to find SCCs, we will quickly prove two theorems:

SCC Theorems

1. If you start DFS from a vertex in a sink SCC, you visit exactly that SCC.
2. The vertex with the highest postorder number is in a source SCC.

Proof of 1: a sink SCC is a SCC with no edges that leave the SCC. In this case, starting a DFS must reach every vertex in the SCC (since there is a path from every vertex in the SCC to every other vertex) and no other vertices, since there is no edge that leaves the SCC. Hence this only visits this SCC.

Proof of 2: a source SCC is a SCC with no edges into the SCC. Assume for contradiction that the vertex with the highest postorder number is not in a source SCC. Then there is an incoming edge to this SCC from some vertex u not in this SCC. There can't be an outgoing edge to u's SCC, else we would our SCCs wouldn't be maximal and we would have a contradiction. u must have a greater postorder number than any vertex in this SCC, since it is either explored when we restart our DFS some time after we finish a DFS in this SCC, our we ran a DFS that reached u, and then subsequently this SCC, where it must explore the whole SCC before backtracking to u. Hence in either case the vertex with the highest postorder number cannot be in this SCC, and we have reached a contradiction.

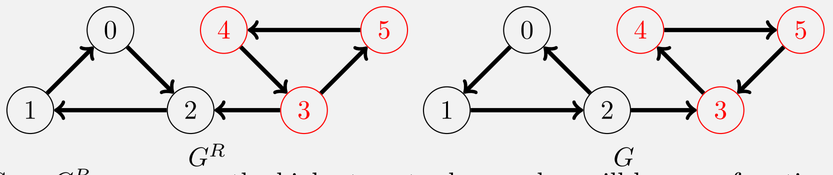
Now we combine these two into an algorithm to find SCCs:

SCC-finding algorithm

1. Run DFS on G^R
2. Run DFS on G, processing vertices in decreasing postorder number from step 1. At the beginning and after every restart, print "New SCC". Previsit(v): print v.

Why does this algorithm work? By running DFS on G^R, any sink SCCs become source SCCs and vice versa. Thus using our second theorem, the highest postorder number from step 1 is a source SCC in G^R, or a sink SCC in G. Then using the first theorem, in our second step when we run DFS on G starting at this vertex, we must visit only this SCC as desired. Then when we restart, we essentially remove all the vertices from the first SCC - it is a sink SCC, so there are edges into it, but when we run DFS in the future, any edge we traverse and reach this SCC will simply backtrack immediately as all vertices in the SCC must have already been visited. Thus essentially the highest postorder number of the vertices remaining must also now be in a sink SCC, and we can continue this process until we have explored all SCCs.

Quick check



Running DFS on G^R we can see the highest postorder number will be one of vertices $3, 4, 5$. Let's assume (without loss of generality) tht it is vertex 3. Running DFS from 3 will encounter $3, 4, 5$, one SCC. Then running DFS on the rest of the graph, we will get $0, 1, 2$. Note that we could essentially ignore the first SCC once we've found it, since any DFS that reaches it will immediately backtrack. This means we can essentially ignore the $(2, 3)$ edge, such that the second SCC is an 'effective sink' when we run DFS on it.

4 Summary

Summary

- Graphs
  - Tuple of vertices and edges, $G = (V(G), E(G))$.
  - Can have weights attached to edges using function $f : E(G) \mapsto \mathbb{R}$.
  - Can be represented by adjacency matrix ($\Omega(n^2)$ space, $O(1)$ edge check) or adjacency list ($O(n+m)$ space, $O(n)$ edge check)
  - Can be directed/undirected, cyclic/acylic, connected, complete, tree, DAG.
- DFS
  - Search a graph by repeatedly following edges and backtracking. Runs in $O(n+m)$.
  - Postvisit and previsit increment shared counter (can be modified, e.g. SCC algorithm)
  - Four types of edges: tree, forward, back and cross edges.
  - DFS properties: if (u, v) is a back edge, $postorder(u) < postorder(v)$ and G has a cycle $\iff$ DFS yields a back edge.
  - DFS can be used to find a topogical sort on a DAG by ordering in decreasing postorder number.
- SCCs
  - SCCs are maximal vertex sets such that every vertex in the set has a path to every other.
  - Graph G' with SCCs of G as vertices forms a DAG.
  - Can find SCCs by running DFS on G^R, then running DFS on G and processing vertices in decreasing postorder number from first DFS.

5 Practice problems

5.1 Leetcode practice problems

Programming practice problem 1

144. Binary Tree Preorder Traversal (Easy difficulty)

Python solution using DFS below. Note how for a binary tree we don't need a visited set, we are guaranteed acyclic and connected so a visited set is redundant.

```python
def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
    lst = []
    def dfs(node: Optional[TreeNode]):
        if not(node):
            return
        lst.append(node.val)
        dfs (node.left)
        dfs (node.right)
    dfs(root)
    return lst
```

There are even more condensed (and slightly speedier) solutions, here's one:

```python
def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
    if not root:
        return []
    return [root.val] + self.preorderTraversal(root.left) + self.preorderTraversal(root.right)
```

Programming practice problem 2

145. Binary Tree Postorder Traversal (Easy difficulty)

This is a simple modification from the previous part using the first set of code

```python
def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
    lst = []
    def dfs(node: Optional[ListNode]):
        if not(node):
            return
        dfs(node.left)
        dfs(node.right)
        lst.append(node.val)
    dfs(root)
    return lst
```

How can you modify the condensed solution to handle postorder traversal? (*hint: it's just as simple as the modification for the longer solution*)

Programming practice problem 3

100. Same Tree (Easy difficulty)

Python solution:

```
def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
    if not(p) and not(q):
        return True
    if not(p) or not(q):
        return False
    return p.val == q.val and self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

Programming practice problem 4

695. Max Area of Island (Medium difficulty)

Python solution using DFS:

```
def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
    self.height = len(grid)
    self.width = len(grid[0])
    max_area = 0

    def dfs(grid: List[List[int]], x: int, y: int):
        if (y < 0 or y >= self.height) or (x < 0 or x >= self.width):
            return 0
        if (grid[y][x] == 0):
            return 0
        # Save space on visited set by rather setting visited locations to 0!
        grid[y][x] = 0
        return 1 + dfs(grid, x + 1, y)\
                 + dfs(grid, x - 1, y)\
                 + dfs(grid, x, y + 1)\
                 + dfs(grid, x, y - 1)

    for y in range(self.height):
        for x in range(self.width):
            if(grid[y][x] == 1):
                max_area = max(dfs(grid, x, y), max_area)
    return max_area
```

Programming practice problem 5

302. Smallest Rectangle Enclosing Black Pixels (Hard difficulty)

Python solution using DFS (*note: I disagree with the presentation of the problem, to me x and y should be flipped but the example makes it clear that x actually relates to height not width*):

```python
def minArea(self, image: List[List[str]], x: int, y: int) -> int:
    # Initially given a black pixel
    maxY = minY = y
    maxX = minX = x
    height = len(image)
    width = len(image[0])

    def updateStore(x, y):
        nonlocal maxX, maxY, minX, minY
        maxX = max(x, maxX)
        maxY = max(y, maxY)
        minX = min(x, minX)
        minY = min(y, minY)

    def dfs(x: int, y: int):
        if image[x][y] != "1":
            return
        # Rather than visited set, replace all visited pixels! Save some space :)
        image[x][y] = "0"
        updateStore(x, y)
        dfs(min(x + 1, height - 1), y)
        dfs(max(x - 1, 0), y)
        dfs(x, min(width - 1, y + 1))
        dfs(x, max(0, y - 1))
    dfs(x, y)
    # Beware sneaky off-by-1 error
    return (maxX - minX + 1) * (maxY - minY + 1)
```

Note that there is a more efficient way to tackle this problem that we will cover later in the course!