

# Phyll-in Notes, Lecture 4: Graphs, DFS, SCCs

CS124 - Spring 2023

Andrew Holmes — 10 January 2023 — version 1.0

## Contents

<b>1</b>	<b>Graphs</b>	<b>2</b>
1.1	Graph introduction . . . . .	2
1.2	Graph representation . . . . .	3
<b>2</b>	<b>Depth-First Search (DFS)</b>	<b>4</b>
2.1	DFS introduction . . . . .	4
2.2	DFS and stacks . . . . .	4
2.3	DFS pseudocode . . . . .	6
2.4	DFS time complexity . . . . .	7
2.5	Postvisit/previsit . . . . .	8
2.6	Types of edges . . . . .	8
2.7	DFS properties . . . . .	9
2.8	Topological sort . . . . .	10
<b>3</b>	<b>Strongly Connected Components (SCCs)</b>	<b>11</b>
3.1	Strongly Connected Components introduction . . . . .	11
3.2	Finding SCCs . . . . .	12
<b>4</b>	<b>Summary</b>	<b>13</b>
<b>5</b>	<b>Practice problems</b>	<b>14</b>
5.1	Leetcode practice problems . . . . .	14

# 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.

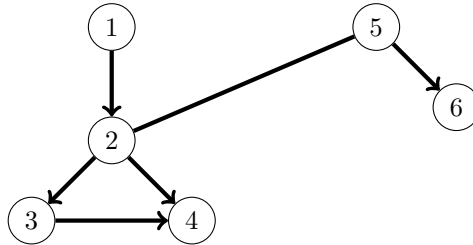


Figure 1: A basic graph

### Quick check

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

$$V(G) = \{ \quad \quad \quad \}$$
$$E(G) = \{ \quad \quad \quad \}$$

### Problem 1: Modelling problems with graphs

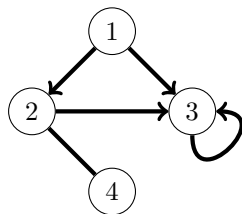
What are some things we might model with graphs?

## 1.2 Graph representation

### Problems 2 & 3: Graph representation

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

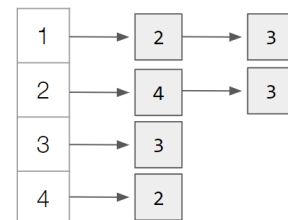
#### Graph representation example



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:

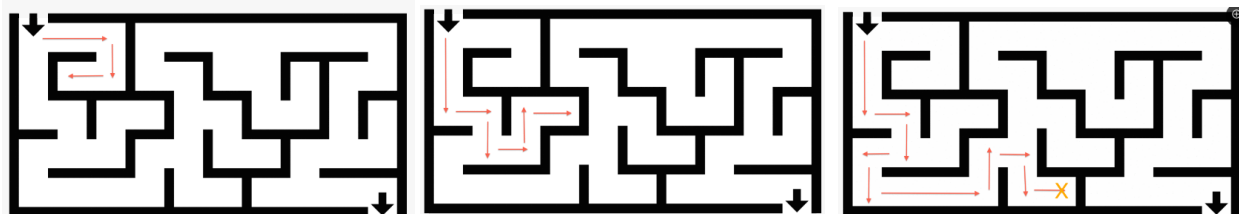
- **Directed or undirected** (see 1.1: Graph introduction)
- **Cyclic or acyclic** - does the graph contain one or more cycles?
- **Connected** - exists a **path** between every pair of vertices (not necessarily a direct edge)
- **Complete** - exists an **edge** between every pair of vertices
- **Tree**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

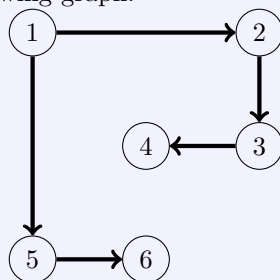
Depth-First Search follows edges from vertex to vertex until it reaches a vertex it has already visited or a vertex with no outgoing edges, at which point it backtracks (reverses its steps) until it reaches a vertex that has outgoing edges that it has not explored yet. One famous example of DFS is exploring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



Our DFS started at vertex 1, then went  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ . 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  $1 \rightarrow 5 \rightarrow 6$  and so on.

### 2.2 DFS and stacks

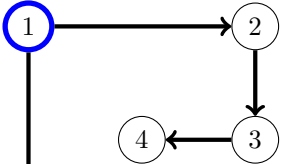
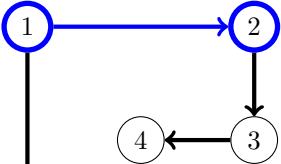
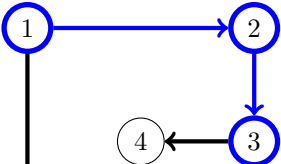
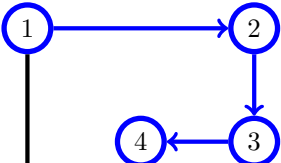
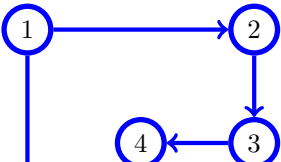
Depth-First Search uses a **stack** to keep track of which vertices to explore next. A stack is a **last in, first out (LIFO)** data structure. This means that the last item we add to the stack is always the first to be removed. We generally refer to adding to a stack as **pushing/appending** to a stack,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

### 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)
```

Complete the pseudocode for DFS to fix the problems from our first attempt!

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

### 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

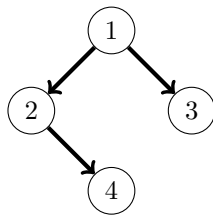
### Problem 5: DFS time complexity

How long does DFS take to run?

## 2.5 Postvisit/previsit

### 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).



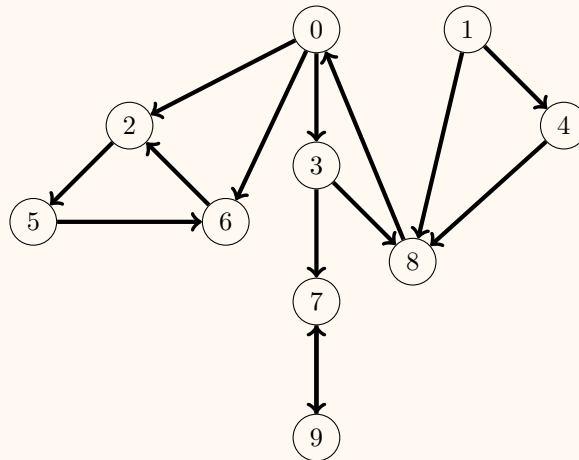
Vertex	Previsit	Postvisit
1		
2		
3		
4		
5		

## 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:
- Forward edges:
- Back edges:
- Cross edges:

Exercise: Label tree, forward, back and cross edges





## 2.7 DFS properties

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

### DFS property 1

**Proof:**

### DFS property 2

**Proof:**

### 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?

## 2.8 Topological sort

### Problem 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?

### Problem 8: Topological sort 2

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

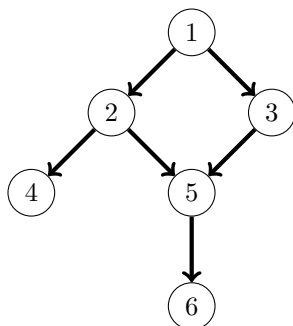
### Definition: Topological sort/ordering

### Topological sort algorithm

- 1.
- 2.

Proof that this gives a topological sort:

### Topological sort example



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

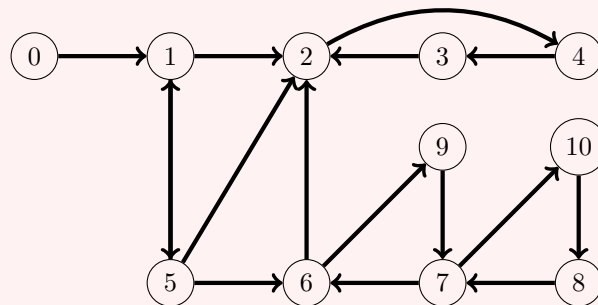
### 3 Strongly Connected Components (SCCs)

#### 3.1 Strongly Connected Components introduction

**Definition: Strongly Connected Component**

#### 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:**

## 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:**

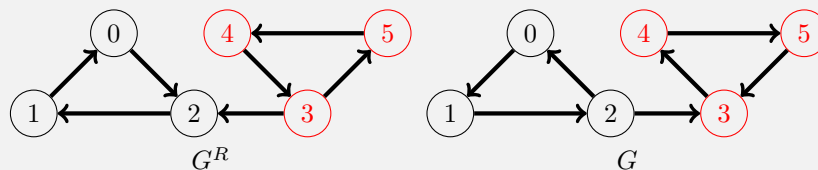
**Proof of 2:**

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

### SCC-finding algorithm

- 1.
- 2.

### 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) that 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

## 5 Practice problems

### 5.1 Leetcode practice problems

#### Programming practice problem 1

144. Binary Tree Preorder Traversal (Easy difficulty)

#### Programming practice problem 2

145. Binary Tree Postorder Traversal (Easy difficulty)

### Programming practice problem 3

100. Same Tree (Easy difficulty)

### Programming practice problem 4

695. Max Area of Island (Medium difficulty)

### Programming practice problem 5

302. Smallest Rectangle Enclosing Black Pixels (Hard difficulty)

#### Note

Note: there is a more efficient way to tackle this problem that we will cover later in the course! You can look at the Leetcode provided solutions if you are interested.