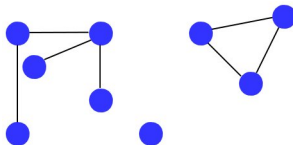


# Graphs

---

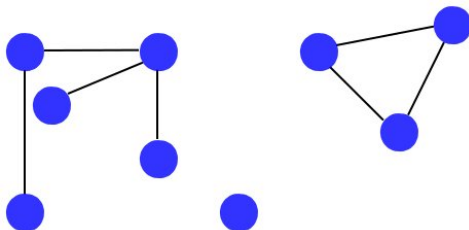
Dr Timothy Kimber

February 2018



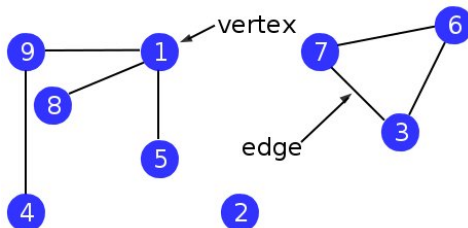
# Introduction

Graphs are fundamental to much of computer science



- We have already seen how trees are used as data structures
- All sorts of problems can be modelled using graphs
- Networks, images, programs, anything involving **related objects**

# Graph Terminology

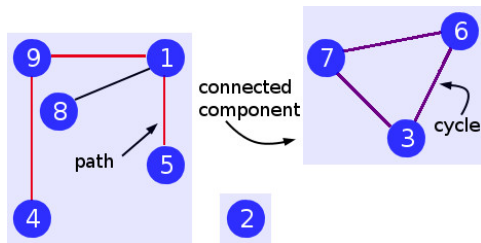


## Definition

A **graph**  $G$  is a pair  $(V, E)$  where  $V$  is a finite set (of objects) and  $E$  is a binary relation on  $V$ . Elements of  $V$  are called **vertices** and elements of  $E$  are called **edges**.

- $E$  is a set of pairs of vertices:  $\{u, v\}$  such that there is an edge between  $u$  and  $v$
- Vertices  $u$  and  $v$  are **adjacent** if there is an edge  $\{u, v\}$

# Graph Terminology



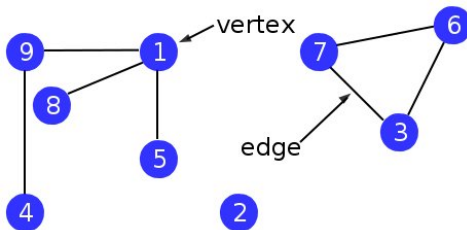
- A **path** from  $v_1$  to  $v_n$ , written  $v_1 \rightsquigarrow v_n$ , is a sequence  $\langle v_1, v_2, \dots, v_n \rangle$  such that there is an edge  $\{v_i, v_{i+1}\}$  for all  $i$ ,  $1 \leq i < n$
- A **cycle** exists if there is a path from  $v$  to  $v$ , containing at least 4 vertices, for some vertex  $v$
- Vertex  $v$  is **reachable** from vertex  $u$  if  $u = v$ , or if there is a path  $u \rightsquigarrow v$
- A **connected component** (also just called a component) is a set of vertices all reachable from each other

# Graph Representation

## Question

How should a graph be represented as a data structure?

- A graph vertex is connected to 0-to-many other vertices
- Going to assume that  $|V|$  is fixed

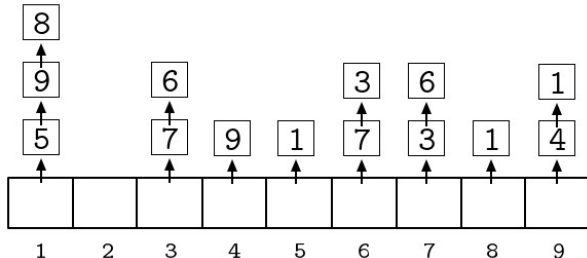


# Graph Representation

Two common ways:

- Adjacency List(s):  $adj[u]$  contains  $v$  if there is an edge  $\{u, v\}$
- Adjacency Matrix:  $adj_{uv} = 1$  if there is an edge  $\{u, v\}$ , else 0

Adjacency lists good for **sparse** graphs

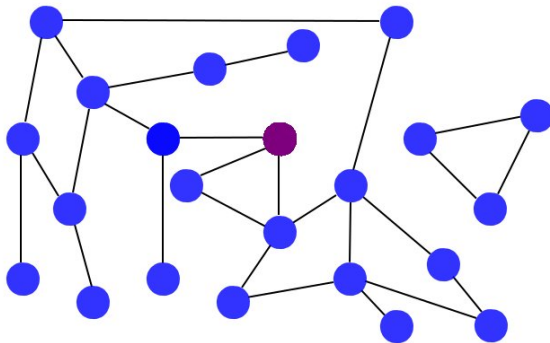


# Graph Search

## Question

Why **search** a graph?

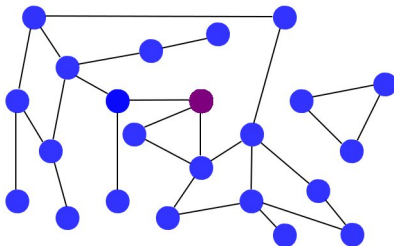
- Searching a graph is like iterating through an ordered structure
- Want to use data in the graph for some computation



# Graph Search Actions

Searching a graph has two actions:

- **Find** adjacent vertices
- **Visit** vertices not found before



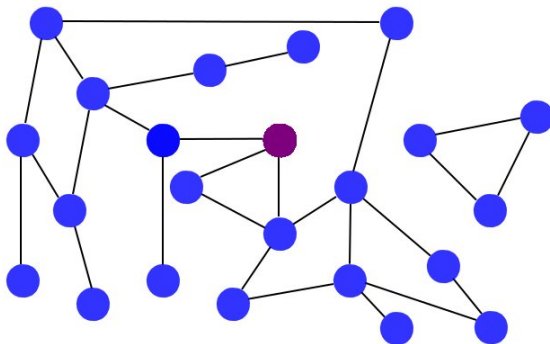
- Visiting means using the vertex: includes finding further vertices
- Vertices are visited in the order they are **first** found
- Vertices are **coloured** when they are first found/visited



# Breadth-First Search

## Question

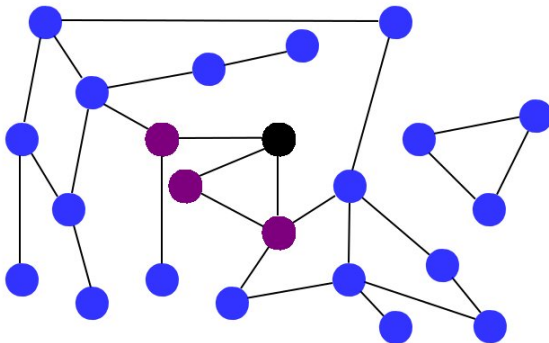
What is a **breadth-first search** of a graph?



# Breadth-First Search

In **breadth-first** search

- Visit a vertex  $v$  (starting with **source vertex**  $s$ )
- Find **all vertices** adjacent to  $v$  before visiting another
- Result: search proceeds gradually down every path at the same rate

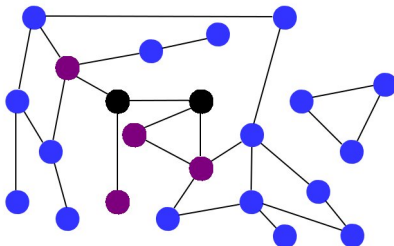


# BFS Procedure

## Question

How would you implement BFS?

- Inputs are graph  $g$  and vertex  $s$
- $g.adj[u]$  returns list of vertices
- $g.vertices$  is number of vertices
- Objective: find all **reachable** vertices (will add actions later)



# Breadth-First Search

BFS (Input: graph  $g$ , vertex  $s$ )

```
found = new boolean[g.vertices]
q = new Queue(s)           // FIFO queue
while q is not empty
    u = q.remove()
    for v in g.adj[u]
        if not found[v]    // avoid loops
            found[v] = true
            q.add(v)
```

- The use of a (FIFO) queue is characteristic of BFS
- By convention only search from given  $s$

# Shortest Paths

BFS searches all paths at the same rate, so ...

## Question

How would you modify the BFS procedure to find the length (number of edges) of the **shortest path** from  $s$  to every other vertex?

## BFS (Input: graph $g$ , vertex $s$ )

```
found = new boolean[g.vertices]
q = new Queue(s)           // FIFO queue
while q is not empty
    u = q.remove()
    for v in g.adj[u]
        if not found[v]    // avoid loops
            found[v] = true
            q.add(v)
```

# Shortest Paths

BFS (Input: graph  $g$ , vertex  $s$ )

```
q      = new Queue(s)
dist = new int[g.vertices]
dist.fill(-1)
dist[s] = 0
while q is not empty
    u = q.remove()
    for v in g.adj[u]
        if dist[v] == -1           // not found
            dist[v] = dist[u] + 1
            q.add(v)
```

- The distance is recorded when a vertex is (first) found
- Arrays of size  $|V|$  like *dist* are also common in graph search
- Unreachable vertices have  $\text{dist}[v] = -1$

# Time

## Question

For a **connected** graph with  $V$  vertices and  $E$  edges, how long does BFS take?

## BFS (Input: graph $g$ , vertex $s$ )

```
found = new boolean[g.vertices]
q = new Queue(s)
while q is not empty
    u = q.remove()
    for v in g.adj[u]
        if not found[v]
            found[v] = true
            q.add(v)
```

# BFS Time Complexity

- Each vertex is added and removed from the queue exactly once
- Each adjacency list is used exactly once
- Each edge contributes exactly two vertices to the adjacency lists
- Time depends on **both** variables:  $O(V + E)$

## BFS (Input: graph $g$ , vertex $s$ )

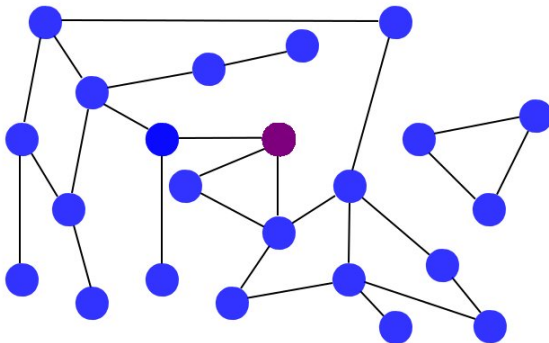
```
found = new boolean[g.vertices]
q = new Queue(s)
while q is not empty
    u = q.remove()           runs once per vertex
    for v in g.adj[u]
        if not found[v]     runs twice per edge
            found[v] = true
            q.add(v)
```



# Depth-First Search

## Question

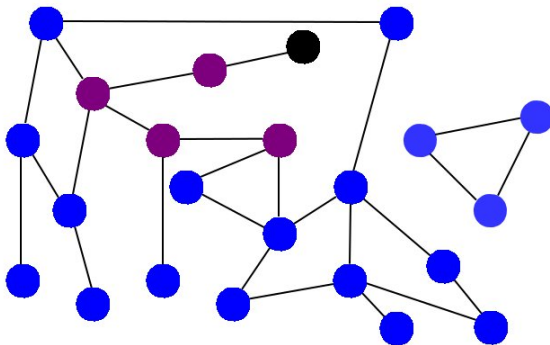
What is a **depth-first search** of a graph?



# Depth-First Search

In **depth-first** search

- **Visit** every vertex as soon as it is found
- i.e. start the next visit before completing current visit
- Result: search follows a single path as far as possible and then **backtracks** to the last alternative path

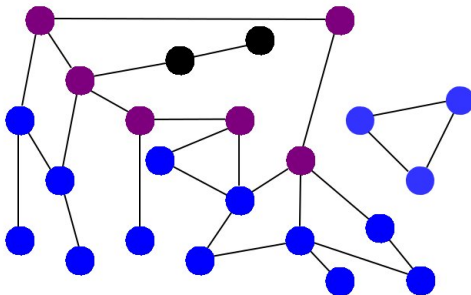


# DFS Procedure

## Question

How would you implement DFS?

- Input is graph  $g$
- Assume  $g.adj[u]$  returns list of vertices
- Objective: find **all** vertices



# Depth-First Search

DepthFirstSearch (Input: graph  $g$ )

```
found = new boolean[g.vertices]
for v in g
    if not found[v]
        DFS(g, v, found)
```

DFS (Input: graph  $g$ , vertex  $s$ , array found)

```
found[s] = true
for v in g.adj[s]
    if not found[v]
        DFS(g, v, found)
```

- DFS can use call stack instead of explicit queue
- Restart until whole graph searched (or not)

# An Application

- Program checks if (whole) graph is acyclic
- Returns true or false

## DepthFirstAcyclic (Input: graph $g$ )

```
parent = new int[g.vertices]
parent.fill(-1)           // nothing found
for v in g
    if parent[v] == -1    // not found
        parent[v] = -2    // found, no parent
        if not DFSAcyclic(g, v, parent)
            return false
return true
```

# Depth-First Search

DFSacyclic (Input: graph  $g$ , vertex  $u$ , array  $\text{parent}$ )

```
for  $v$  in  $g.\text{adj}[u]$ 
    if  $\text{parent}[v] == -1$            // not found
         $\text{parent}[v] = u$ 
        if not DFSacyclic( $g, v, \text{parent}$ )
            return false
    else if  $\text{parent}[u] != v$        // cycle detected
        return false
return true
```

- A cycle exists if  $v$  was already found, unless it is  $u$ 's parent
- Since  $u$  was just found, and not from  $v$ , the edge  $\{u, v\}$  completes an alternative path to  $u$  from the source

# Time

## Question

For a **connected** graph with  $V$  vertices and  $E$  edges, how long does DFS take?

DFS (Input: graph  $g$ , vertex  $s$ , array found)

```
found[s] = true
for v in g.adj[s]
  if not found[v]
    DFS(g, v, found)
```

# DFS Time Complexity

- DFS is called exactly once per vertex
- Each adjacency list is used exactly once
- Each edge contributes exactly two vertices to the adjacency lists
- Time depends on **both** variables:  $O(V + E)$

DFS (Input: graph  $g$ , vertex  $s$ , array  $found$ )

```
found[s] = true           runs V times
for v in g.adj[s]
    if not found[v]       runs 2E times
        DFS(g, v, found)
```