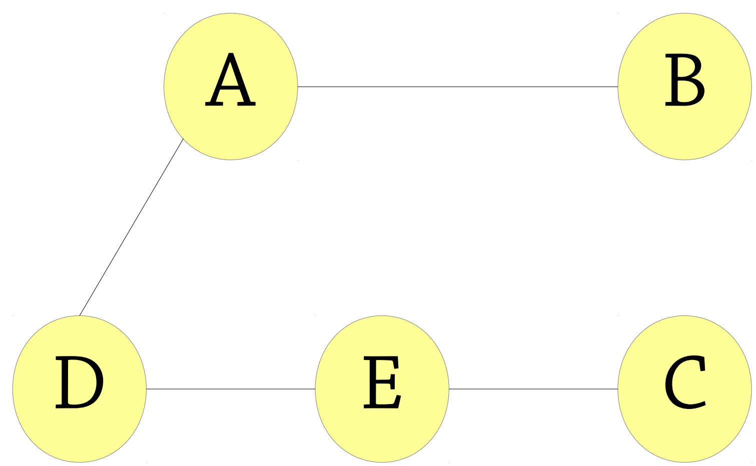
Graphs *(chapter 13)*

Terminology

A graph is a data structure consisting of *nodes* (or vertices) and *edges*

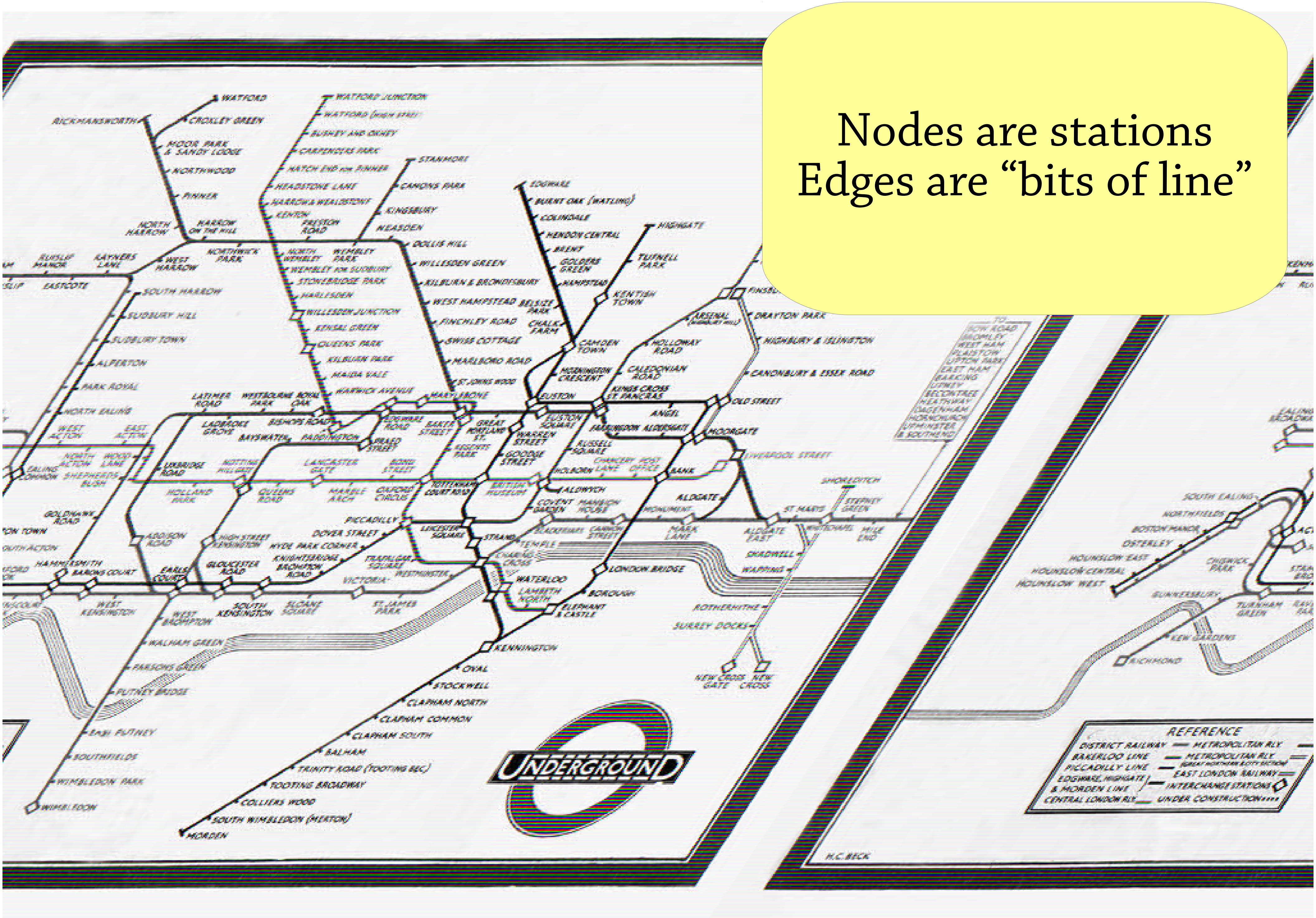
- An edge is a connection between two nodes



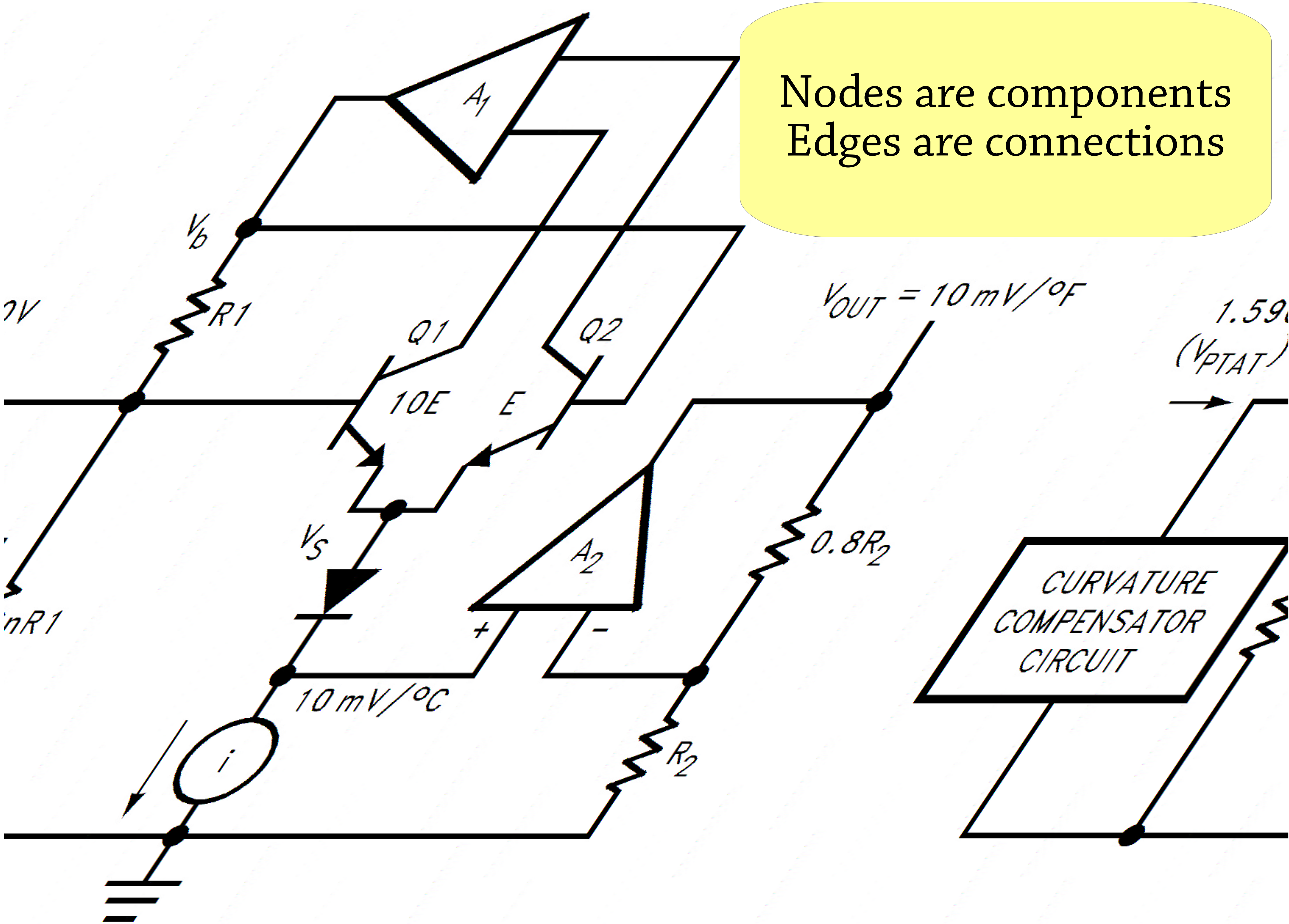
Nodes: A, B, C, D, E

Edges: (A, B), (A, D), (D, E), (E, C)

Nodes are stations
Edges are “bits of line”

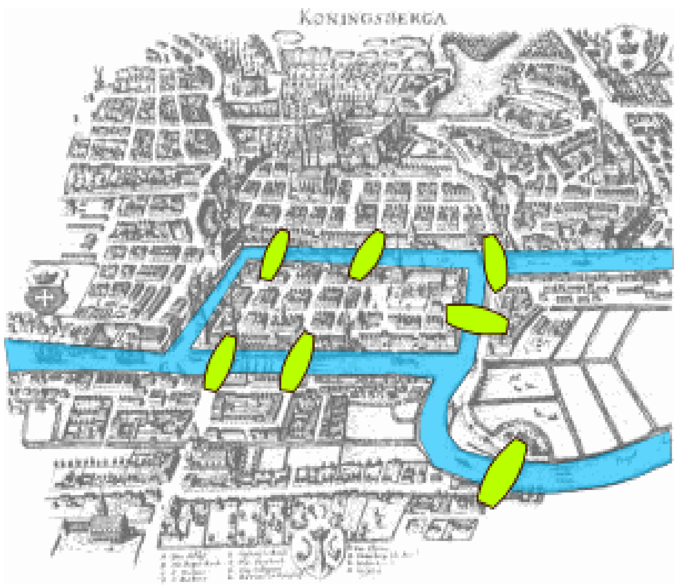


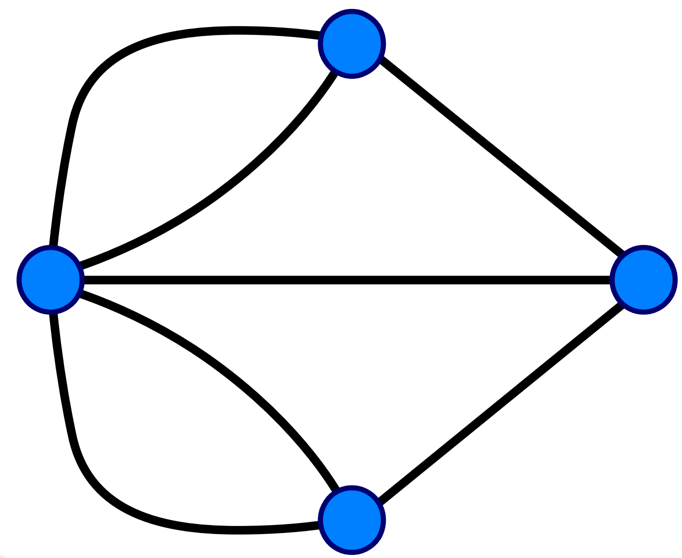
Nodes are components
Edges are connections

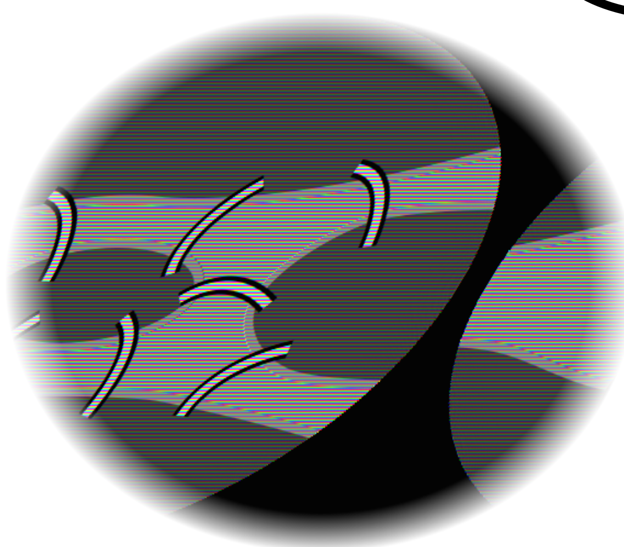


Seven bridges of Königsberg

http://en.wikipedia.org/wiki/Seven_Bridges_of_Königsberg







Graphs

Graphs are used all over the place:

- communications networks
- many of the algorithms behind the internet
- maps, transport networks, route finding
- etc.

Anywhere where you have connections or relationships!

More graphs

Graphs can be *directed* or *undirected*

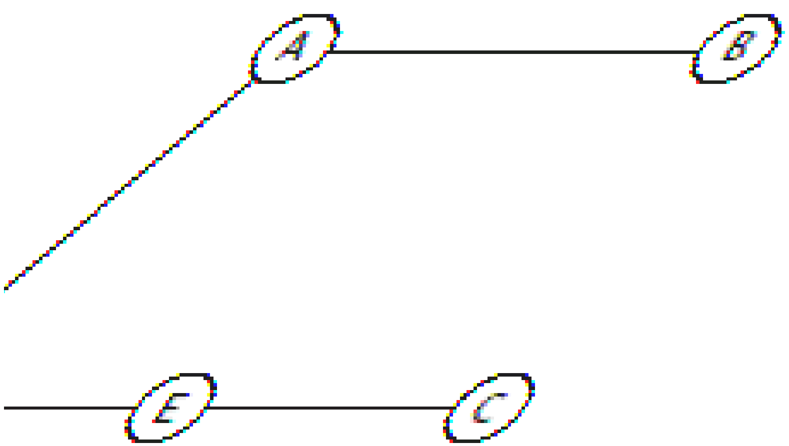
- In an undirected graph, an edge connects two nodes symmetrically (we draw a line between the two nodes)
- In a directed graph, the edge goes from the *source node* to the *target node* (we draw an arrow from the source to the target)

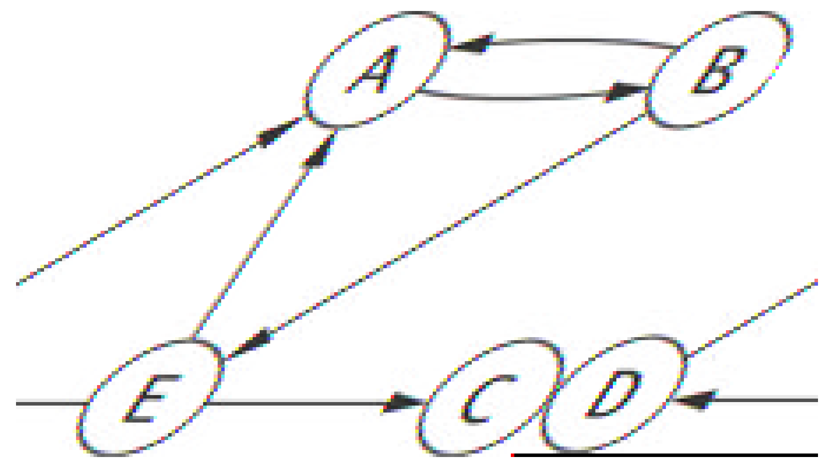
A tree is a special case of a directed graph

- Edge from parent to child

Drawing graphs

We draw nodes as points, and edges as lines (undirected) or arrows (directed):



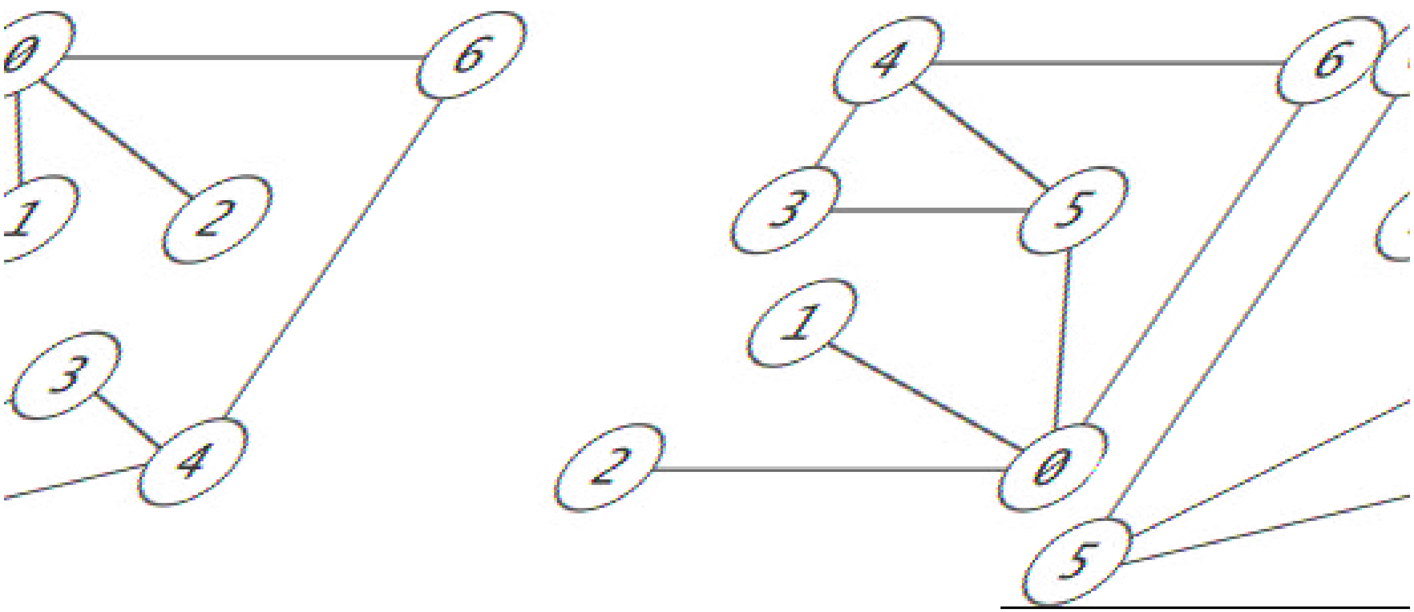


V = {A, B, C, D, E}
E = {(A, B), (A, D),
(C, E), (D, E)}

V = {A, B, C, D, E}
E = {(A, B), (B, A), (B, E),
(D, A), (E, A), (E, C), (E, D)}

Drawing graphs

The layout of the graph is **completely irrelevant**: only the nodes and edges matter

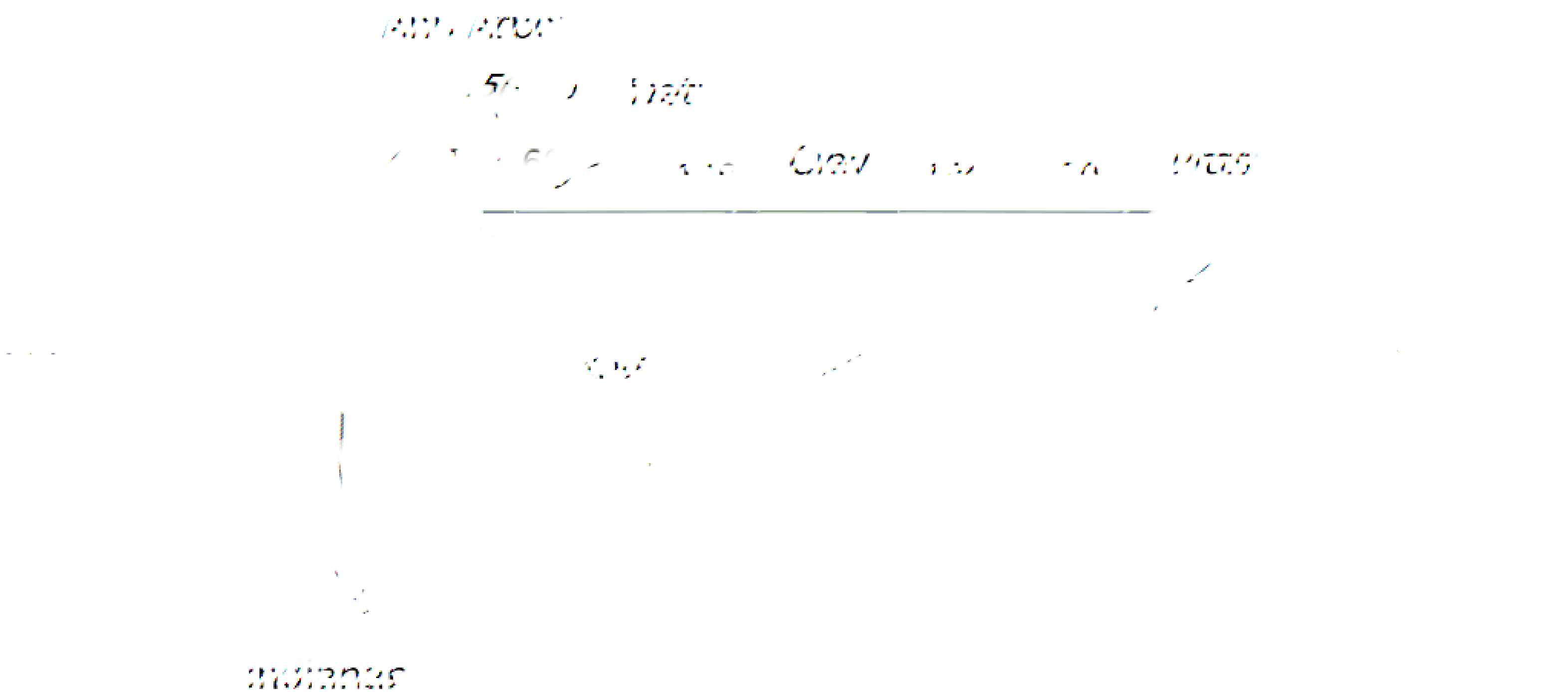


V = {0, 1, 2, 3, 4, 5, 6}

E = {(0, 1), (0, 2), (0, 5), (0, 6), (3, 5), (3, 4), (4, 5), (4, 6)}

Weighted graphs

In a *weighted graph*, each edge has a number, its *weight*:

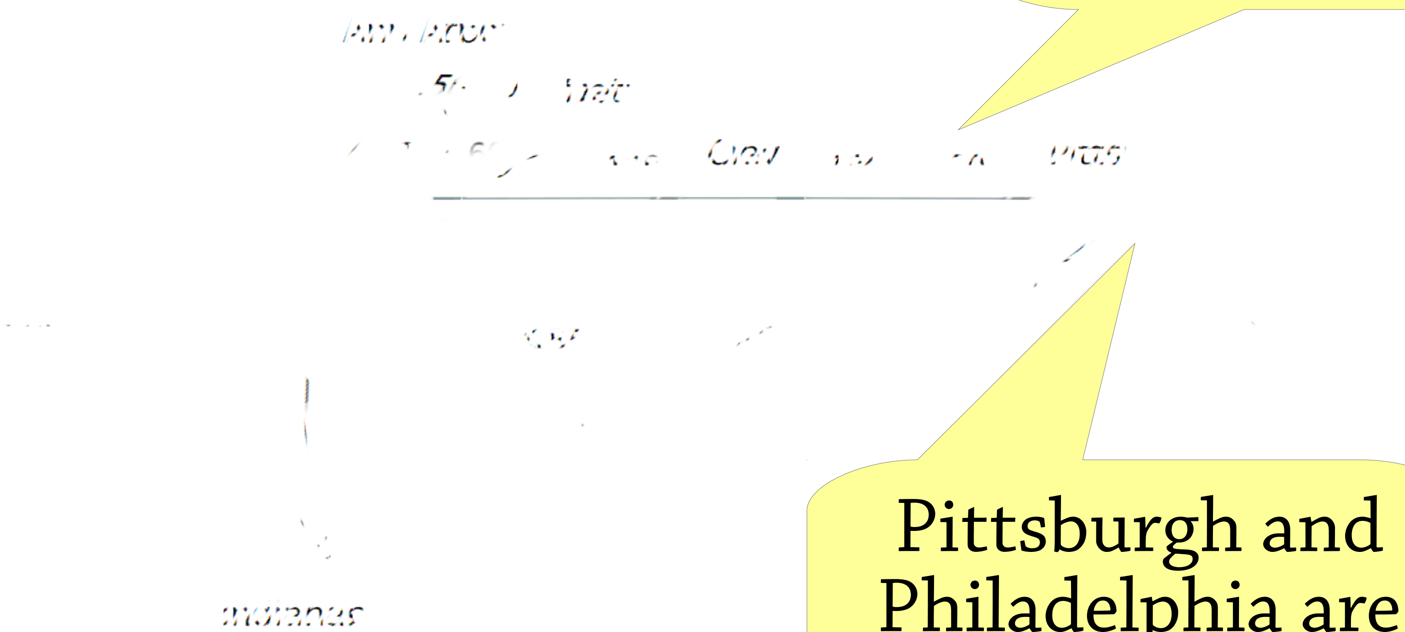


Often, graphs have extra data attached to the edges – weights are one case of this

Paths and cycles

Two vertices are *adjacent* if there is an edge between them:

Cleveland and Pittsburgh are adjacent

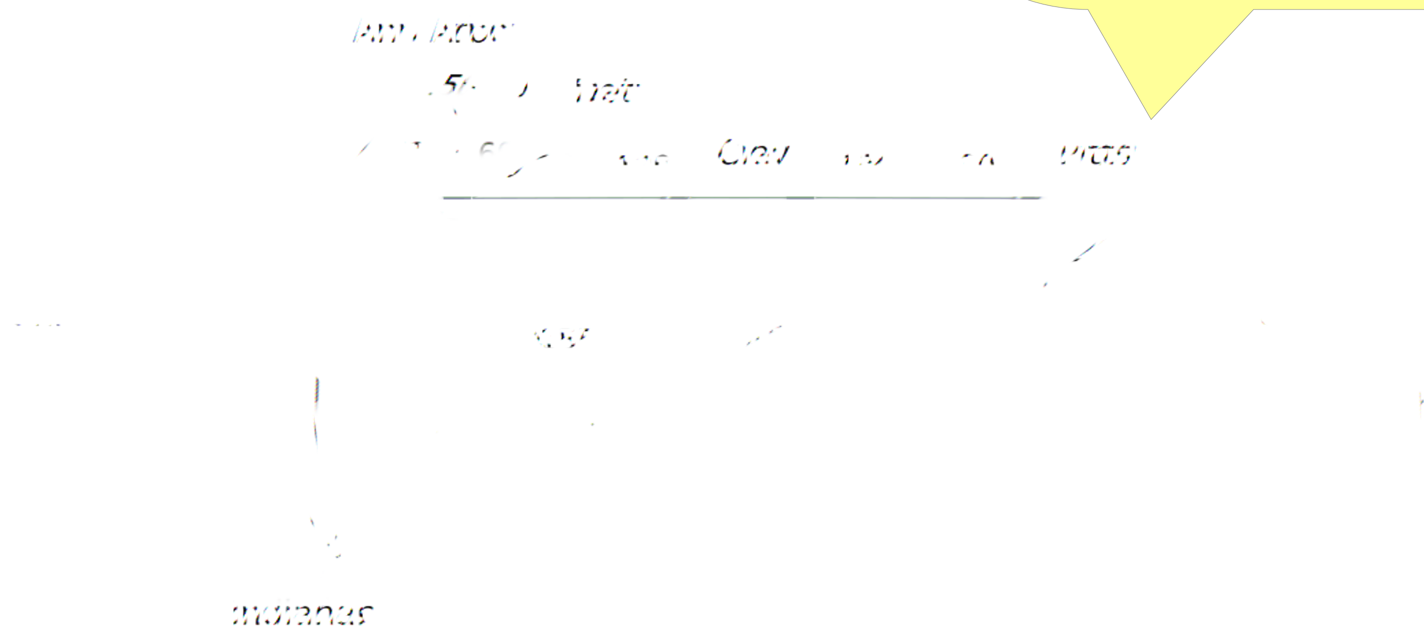


Pittsburgh and Philadelphia are adjacent

Paths and cycles

Two vertices are *adjacent* if there is an edge between them:

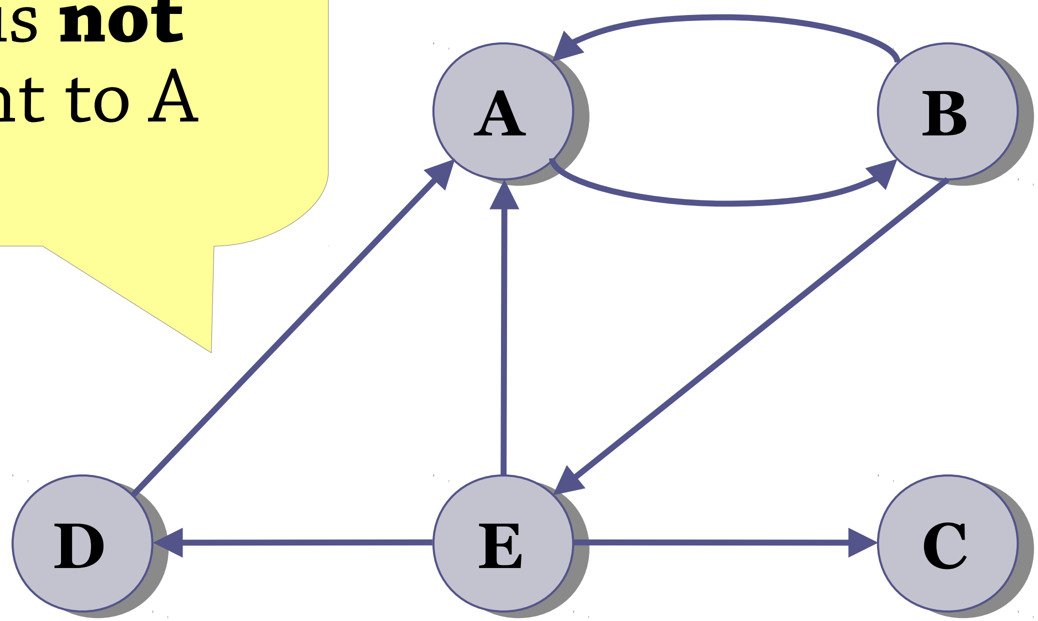
Cleveland and Philadelphia are **not** adjacent



Paths and cycles

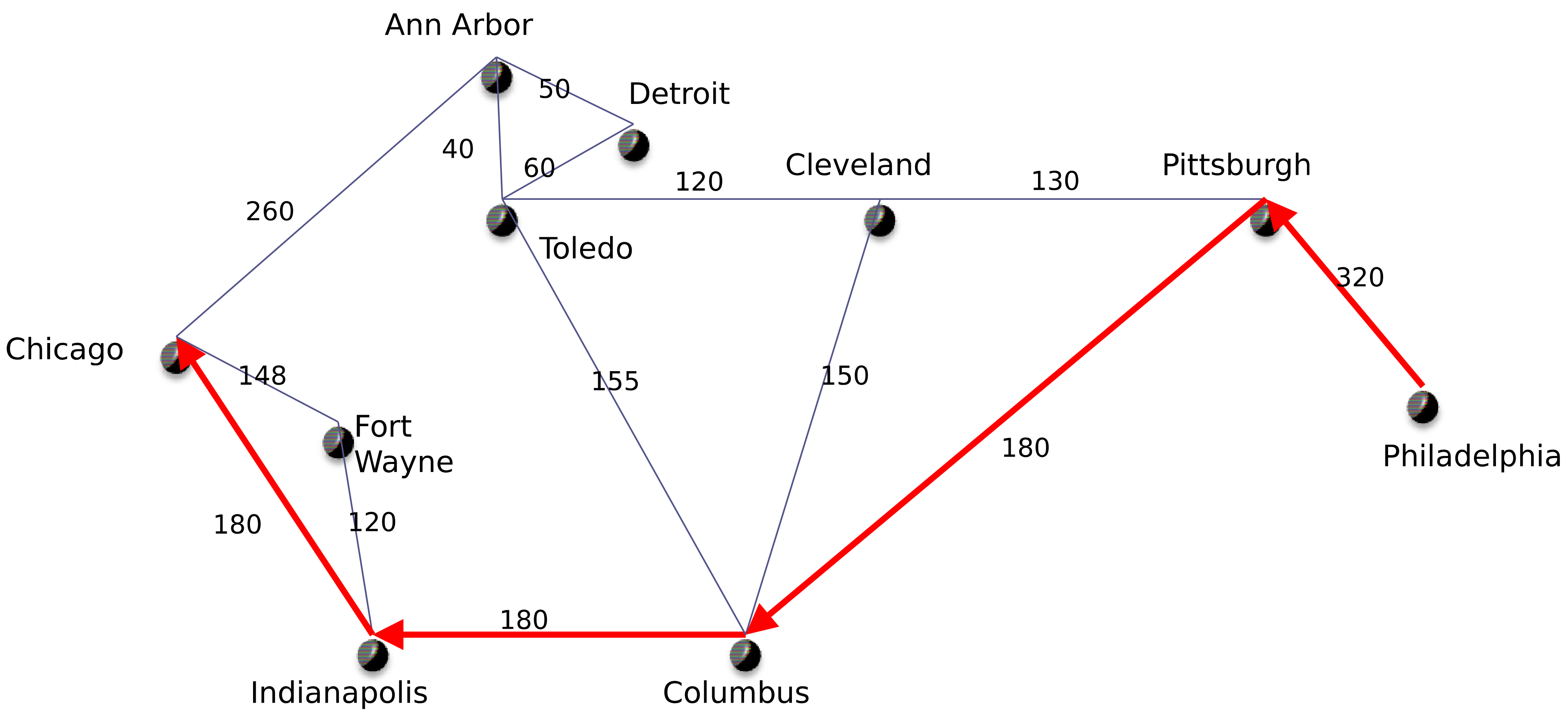
In a directed graph, the *target* of an edge is adjacent to the *source*:

A is adjacent to D, but D is **not** adjacent to A



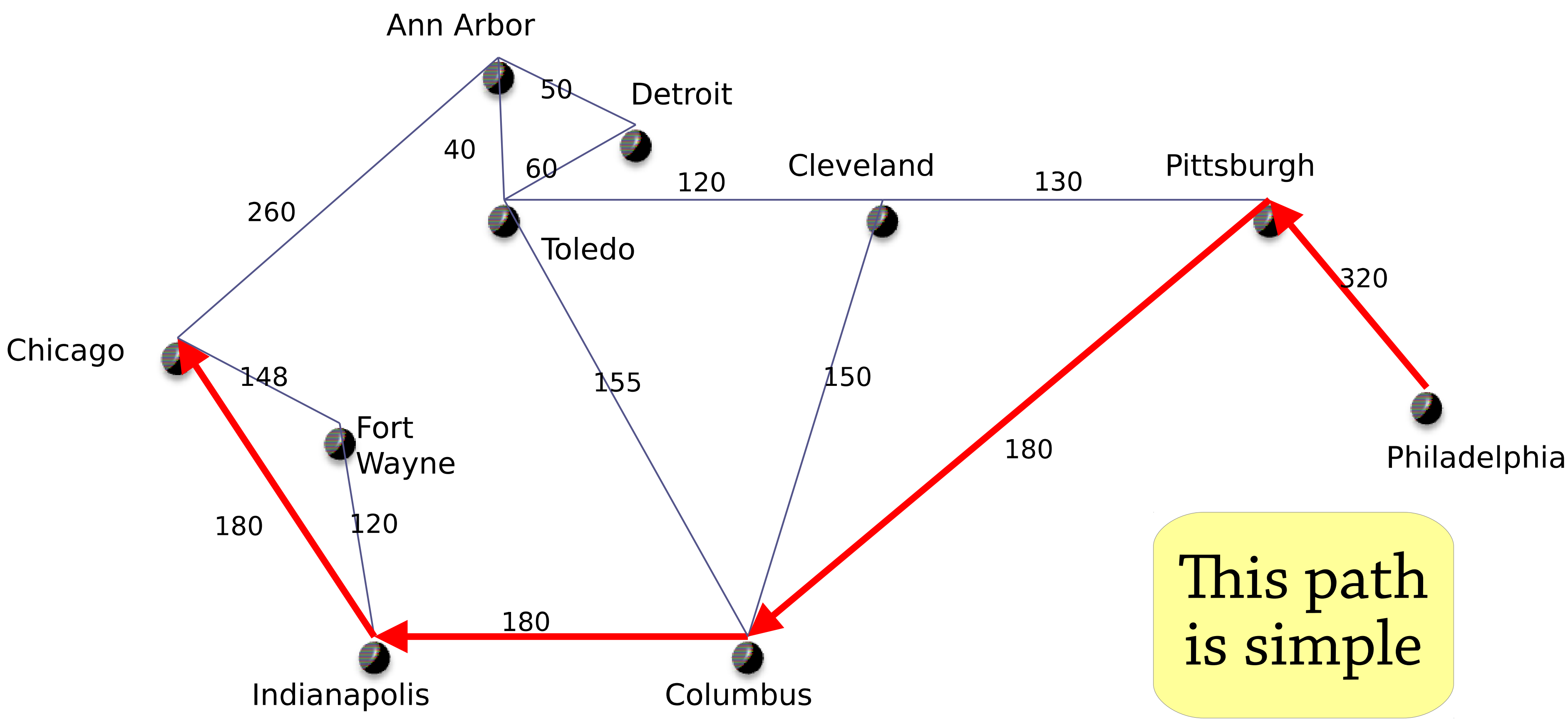
Paths and cycles

A *path* is a sequence of vertices where each vertex is adjacent to its predecessor:



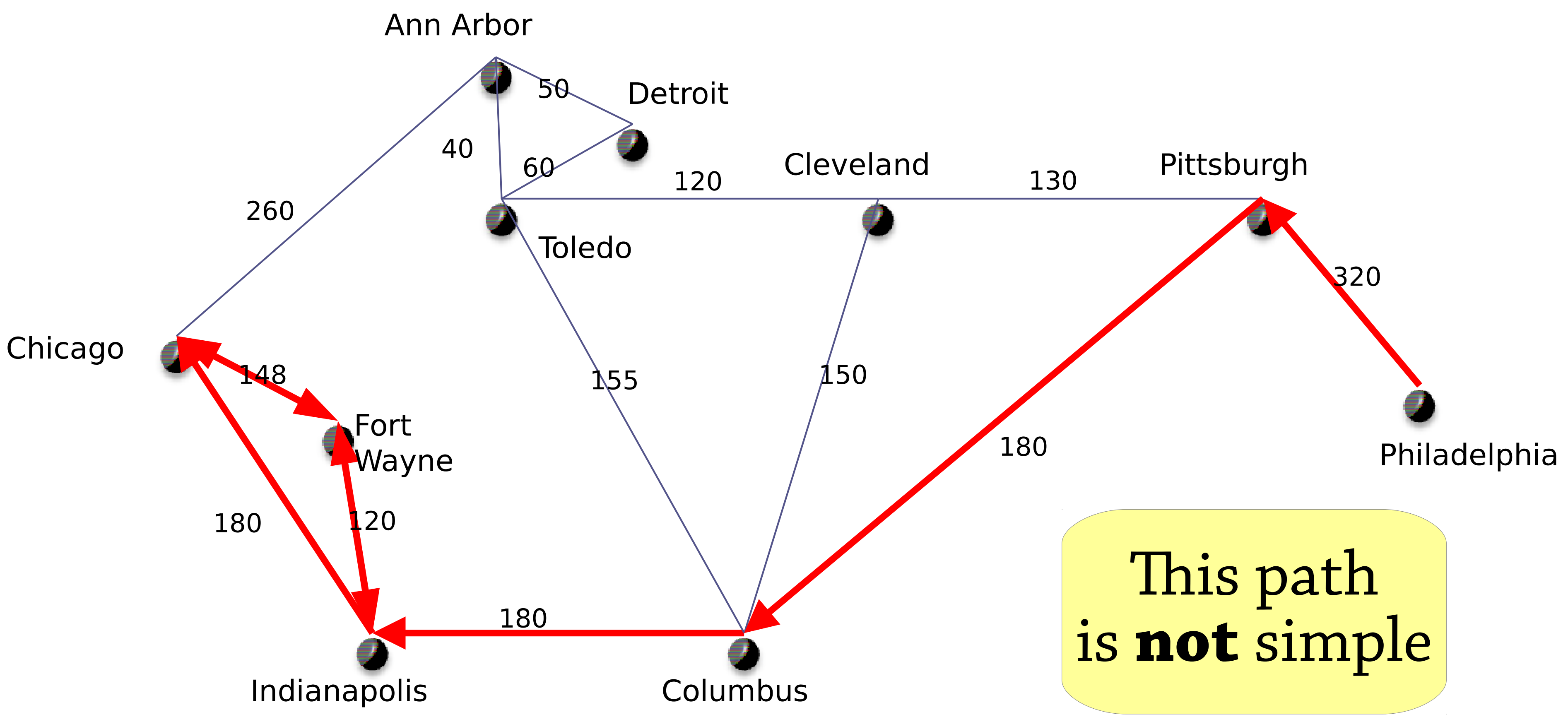
Paths and cycles

In a *simple path*, no node or edge appears twice, except that the path can start and end on the same node:



Paths and cycles

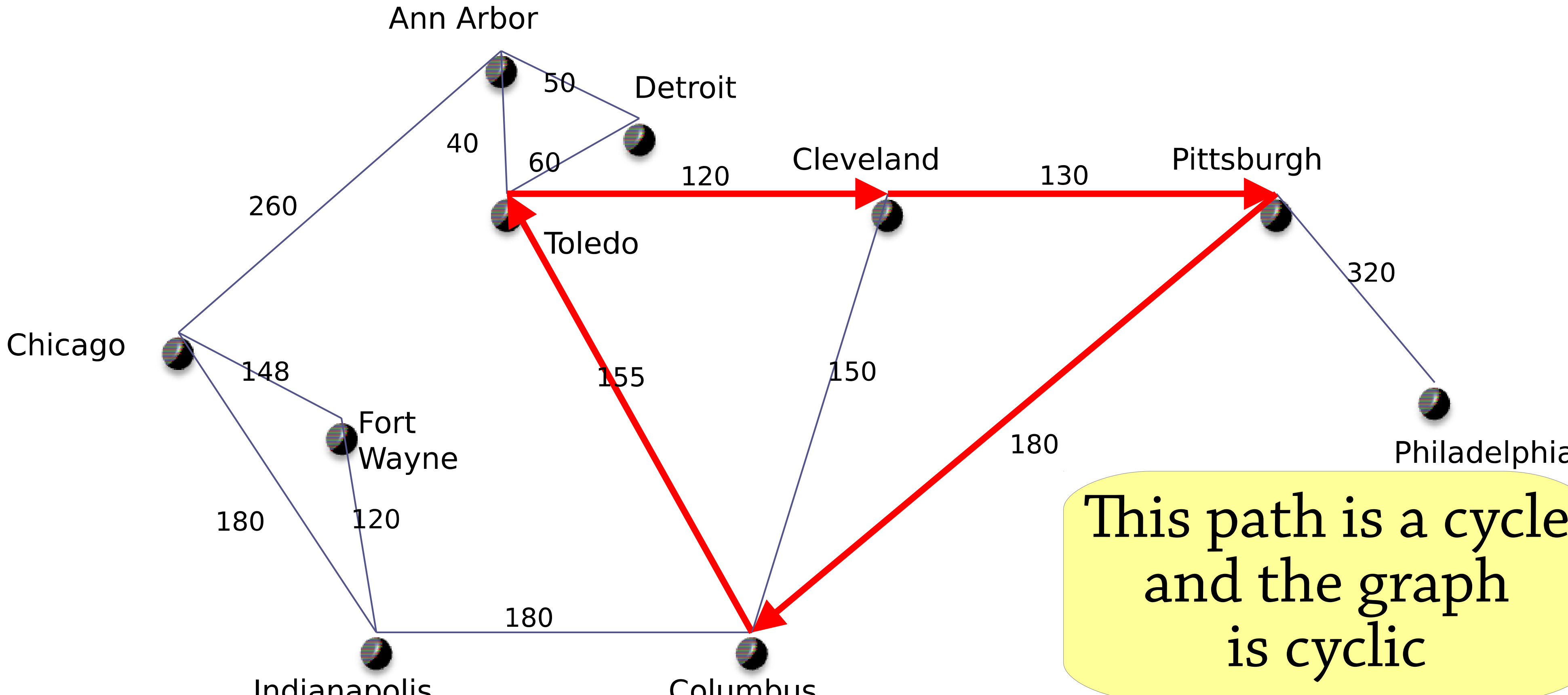
In a *simple path*, no node or edge appears twice, except that the path can start and end on the same node:



This path is **not** simple

Paths and cycles

A *cycle* is a simple path where the first and last node are the same – a graph is *cyclic* if it has a cycle, *acyclic* otherwise

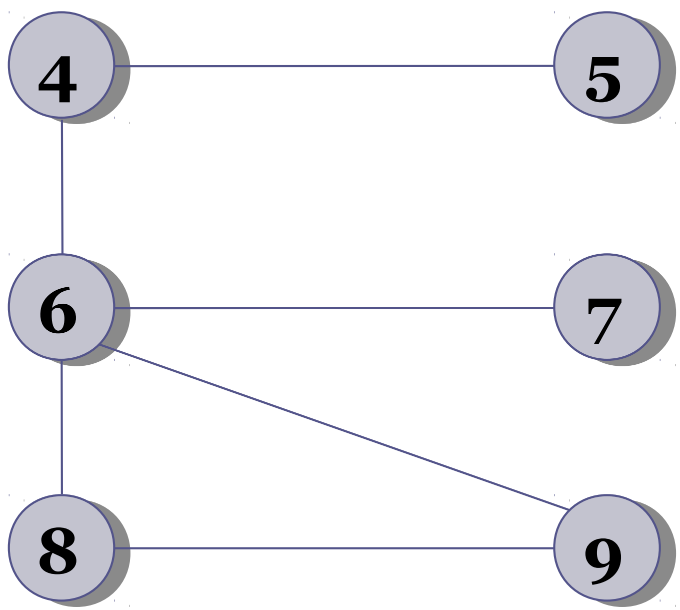


This path is a cycle and the graph is cyclic

Connectedness

A graph is called *connected* if there is a path from every node to every other node

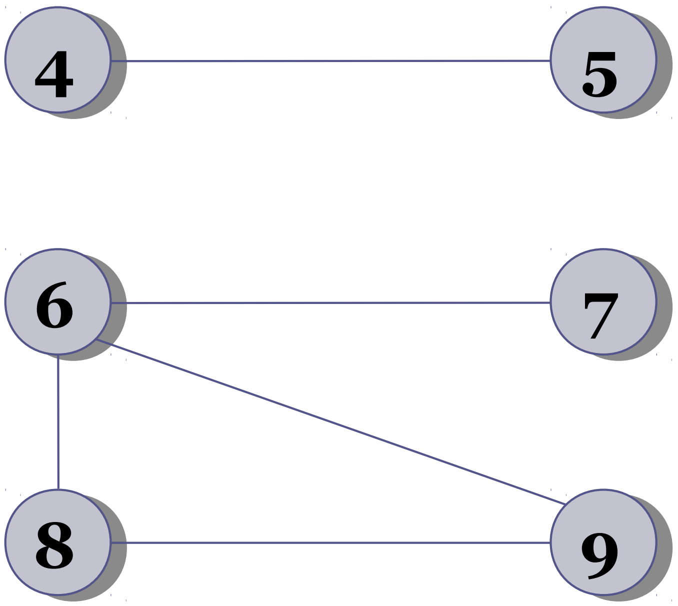
This graph is connected



Connectedness

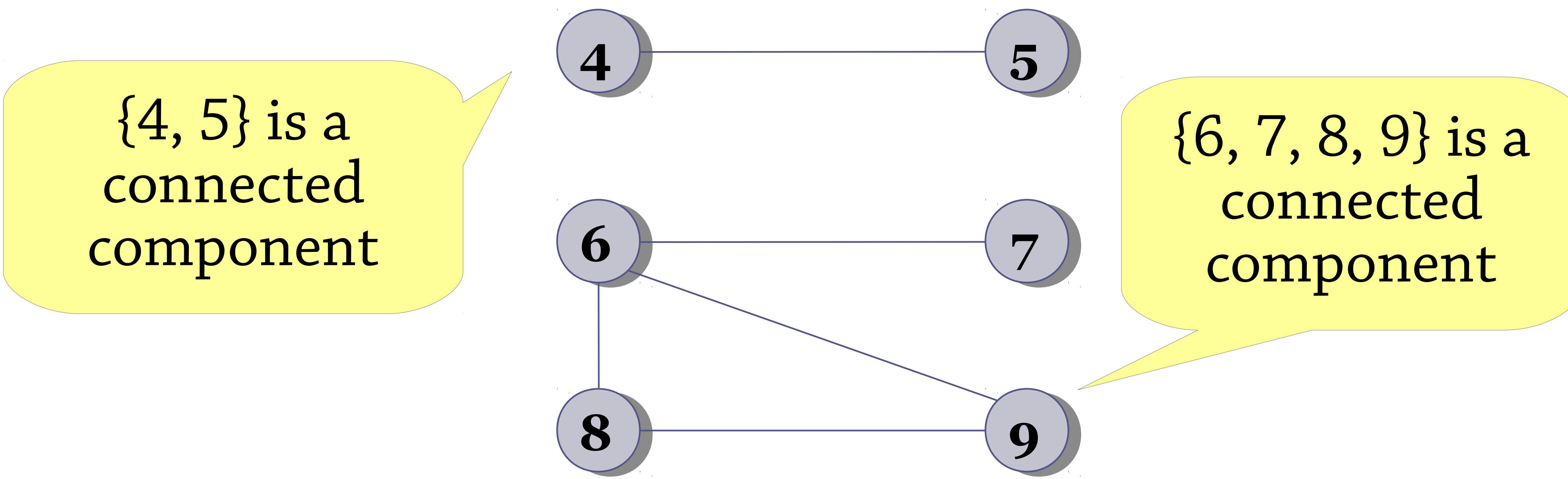
A graph is called *connected* if there is a path from every node to every other node

This graph is **not** connected



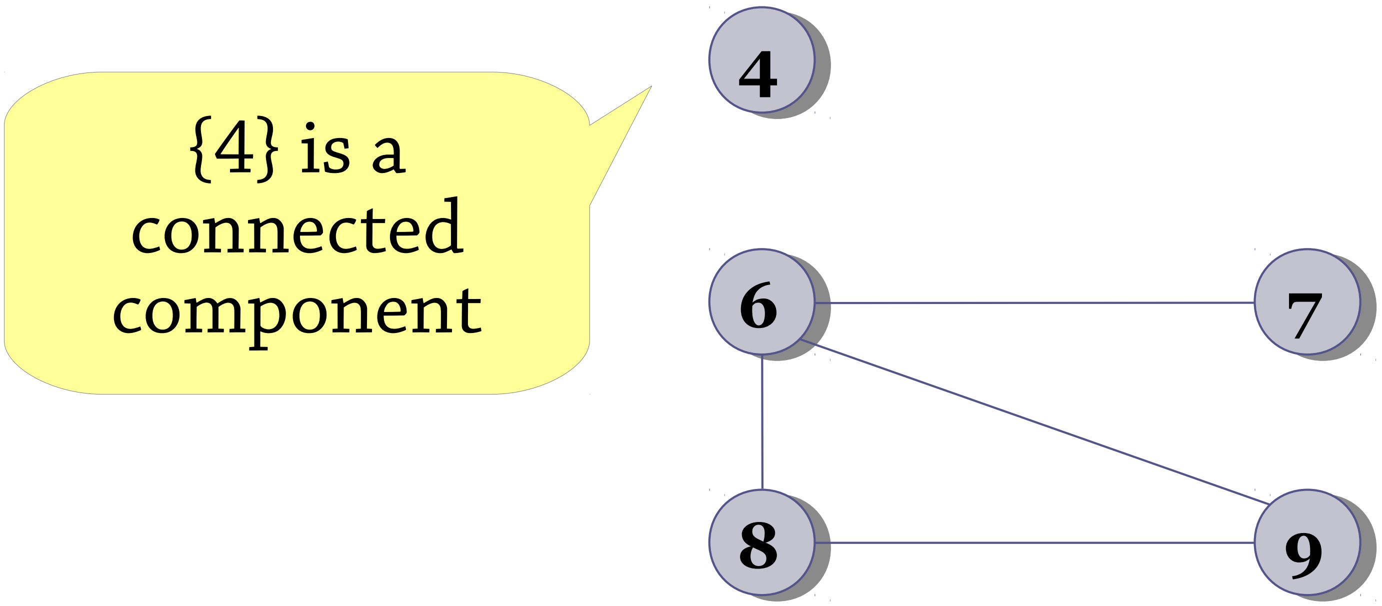
Connectedness

If a graph is unconnected, it still consists of *connected components*



Connectedness

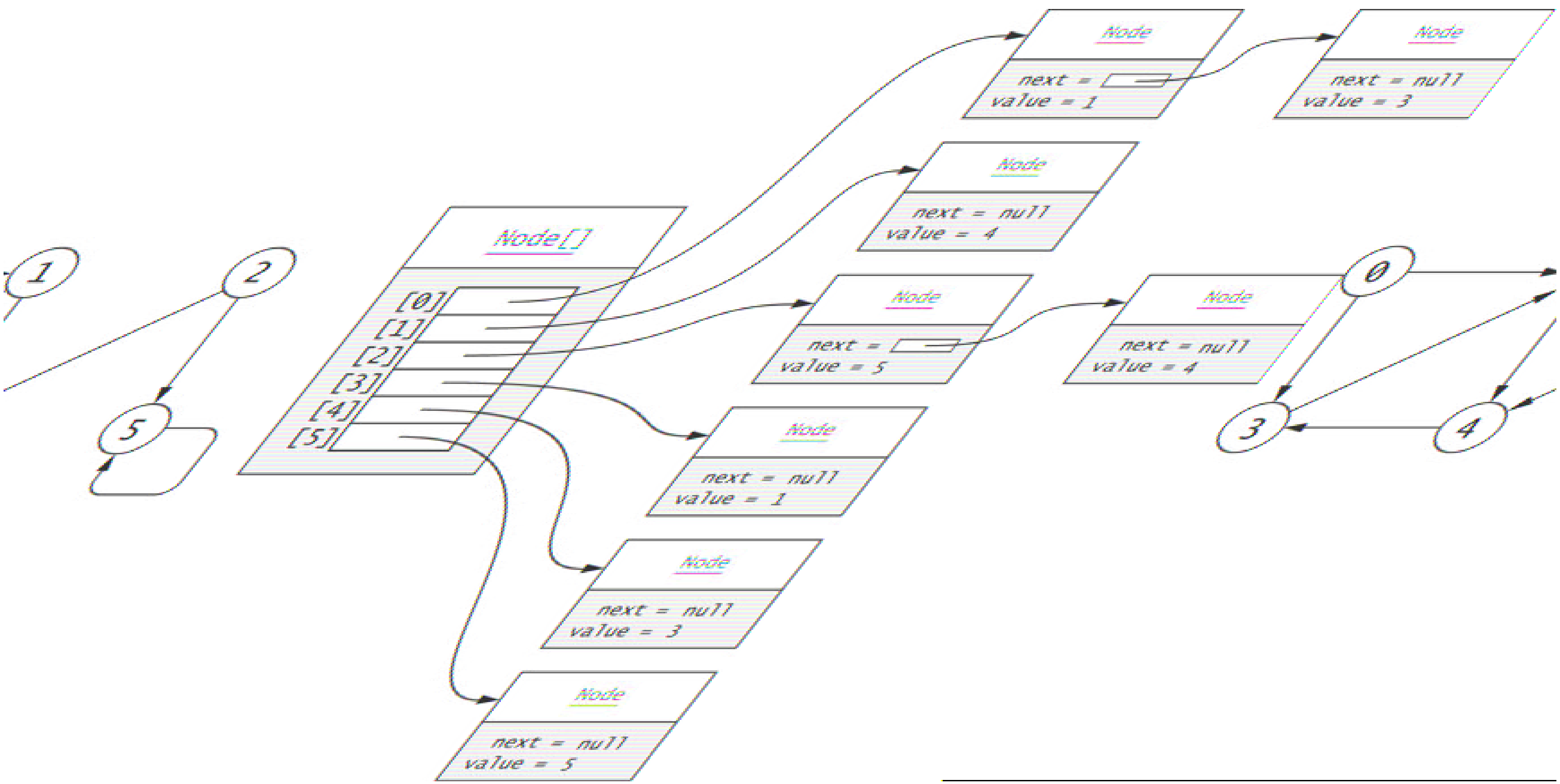
A single unconnected node is a connected component in itself



How to implement a graph

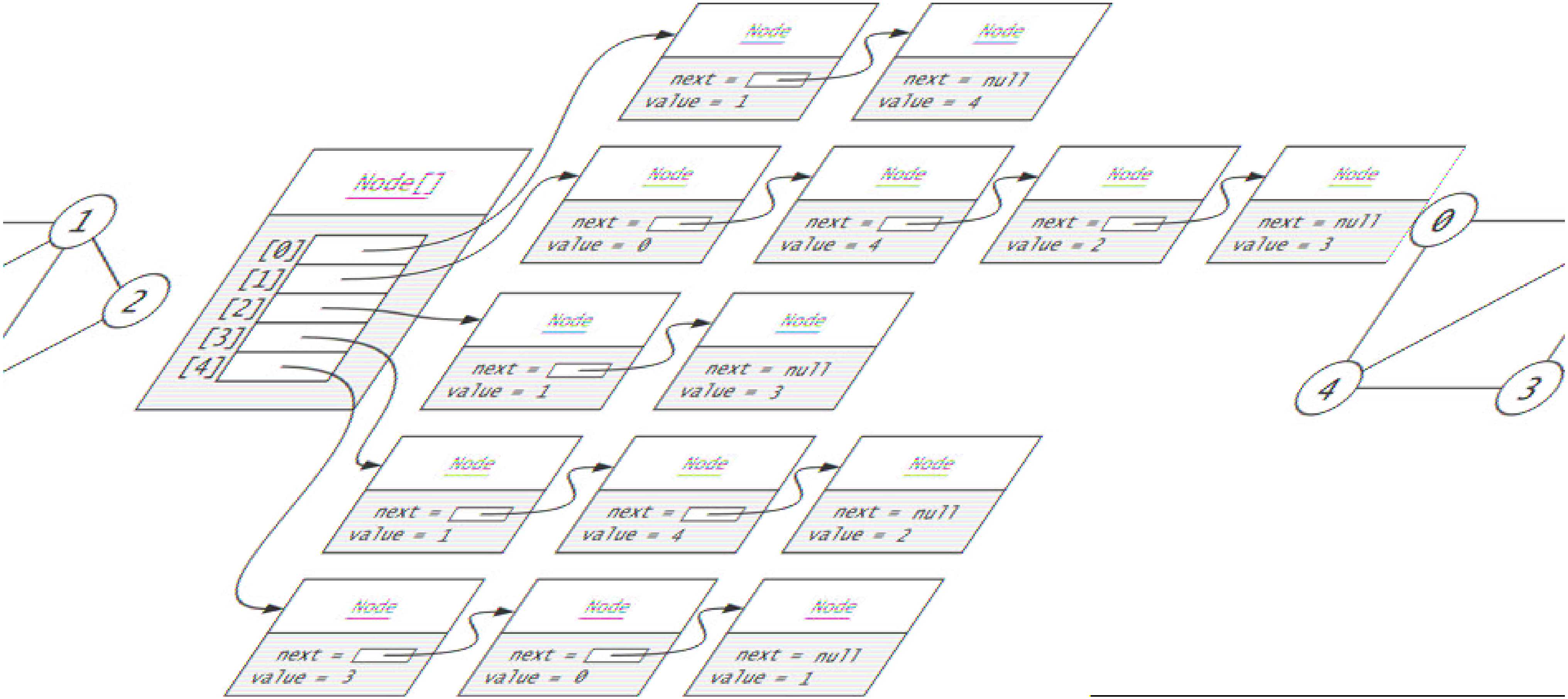
Typically: *adjacency list*

- List of all nodes in the graph, and with each node store all the edges having that node as source



Adjacency list – undirected graph

Each edge appears twice, once for the source and once for the target node



How to implement a graph

Alternative – *adjacency matrix*

- 2-dimensional array

For an unweighted graph, 2-dimensional array of booleans

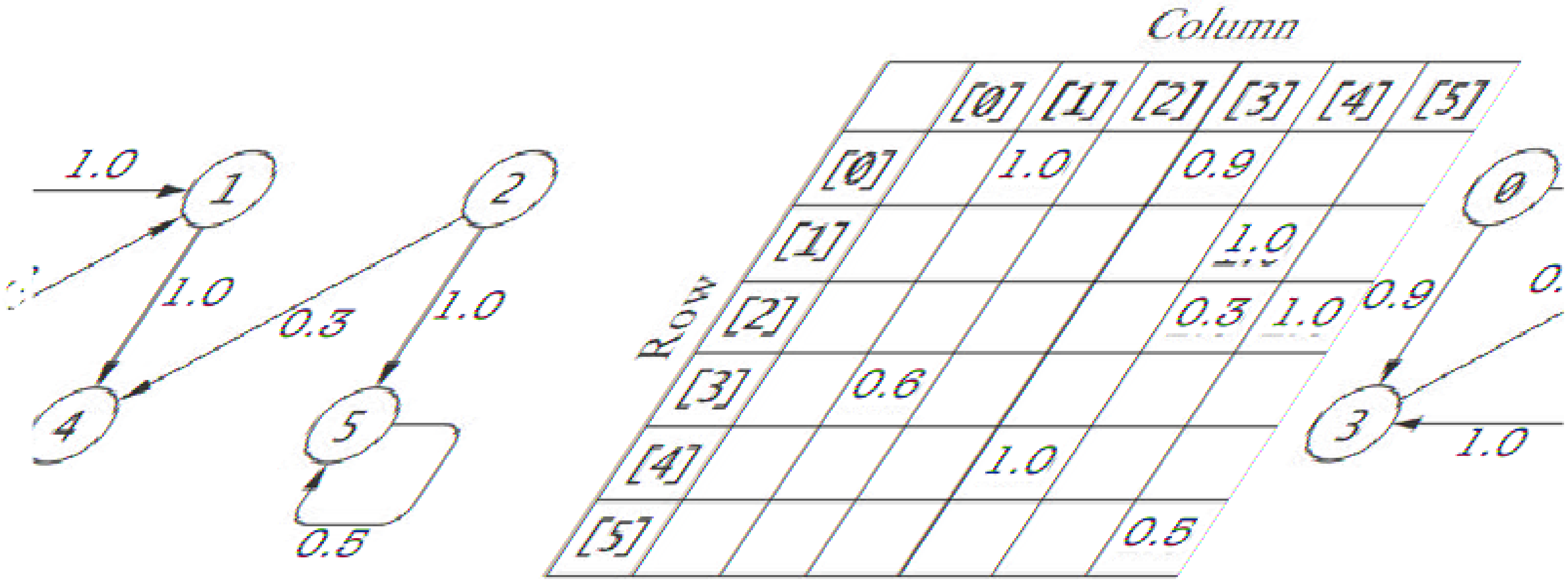
- a[i][j] = true if there is an edge between nodes i and j

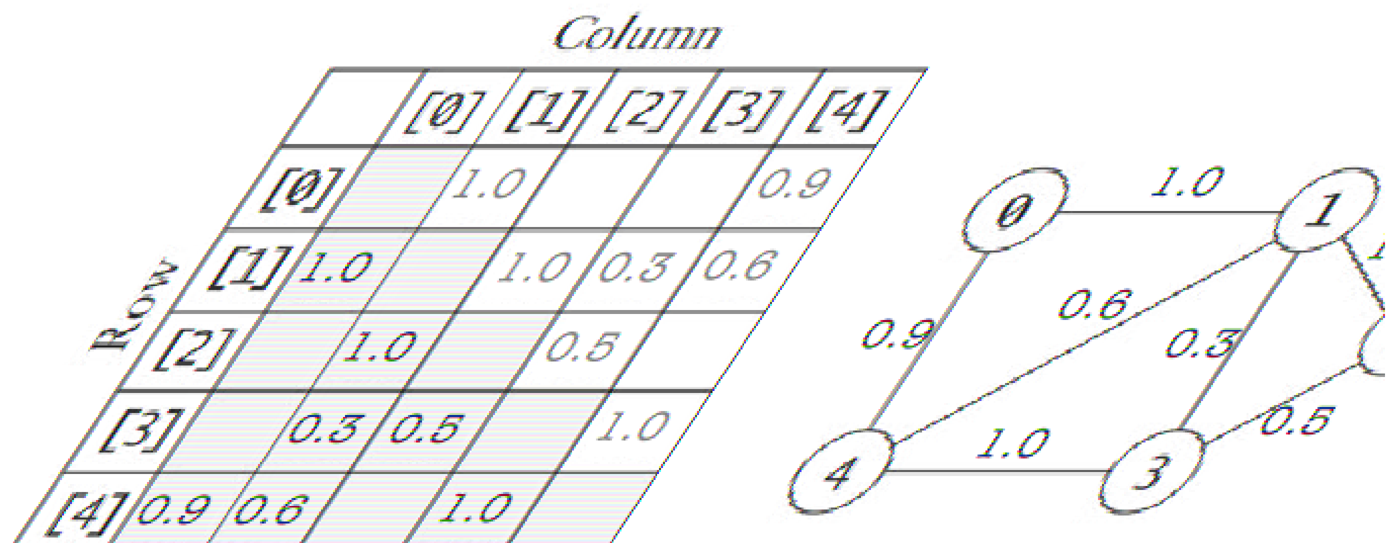
For a weighted graph, the array contains weights instead of booleans

- a[i][j] = the weight, or a special value (e.g. infinity) if there is no edge

For an undirected graph, a[i][j] = a[j][i]

Adjacency matrices





Adjacency matrices – disadvantage

Adjacency matrices need a lot of memory for big graphs

- One bit for each *pair* of nodes
- So $O(|V|^2)$ memory, where $|V|$ is the number of nodes

Adjacency lists only use memory for the nodes and edges that are actually present

- $O(|V| + |E|)$, where $|E|$ is the number of edges
- More like 64 bits for each node and edge

Adjacency lists normally better, but matrices good for:

- Small graphs (only one bit needed per pair of nodes)
- Dense graphs (1% or more (say) of pairs of nodes have edges between them) – most graphs are not dense!

Graphs implicitly

Very often, the data in your program *implicitly* makes a graph

- Nodes are objects
- Edges are references – if obj1.x = obj2 then there is an edge from obj1 to obj2

Sometimes, you can solve your problem by viewing your data as a graph and using graph algorithms on it

This is probably more common than using an explicit graph data structure!

Graph traversals

Many graph algorithms involve visiting each node in the graph in some systematic order

The two commonest methods are:

- depth-first search (DFS)
- breadth-first search (BFS)

Breadth-first search

A breadth-first search visits the nodes in the following order:

- First it visits some node (the *start node*)
- Then all the start node's neighbours (all nodes adjacent to it)
- Then *their* neighbours
- and so on

So it visits the nodes in order of how far away they are from the start node

Implementing breadth-first search

We maintain a *queue* of nodes that we are going to visit soon

- Initially, the queue contains the start node

We also remember which nodes we've already added to the queue

Then repeat the following process:

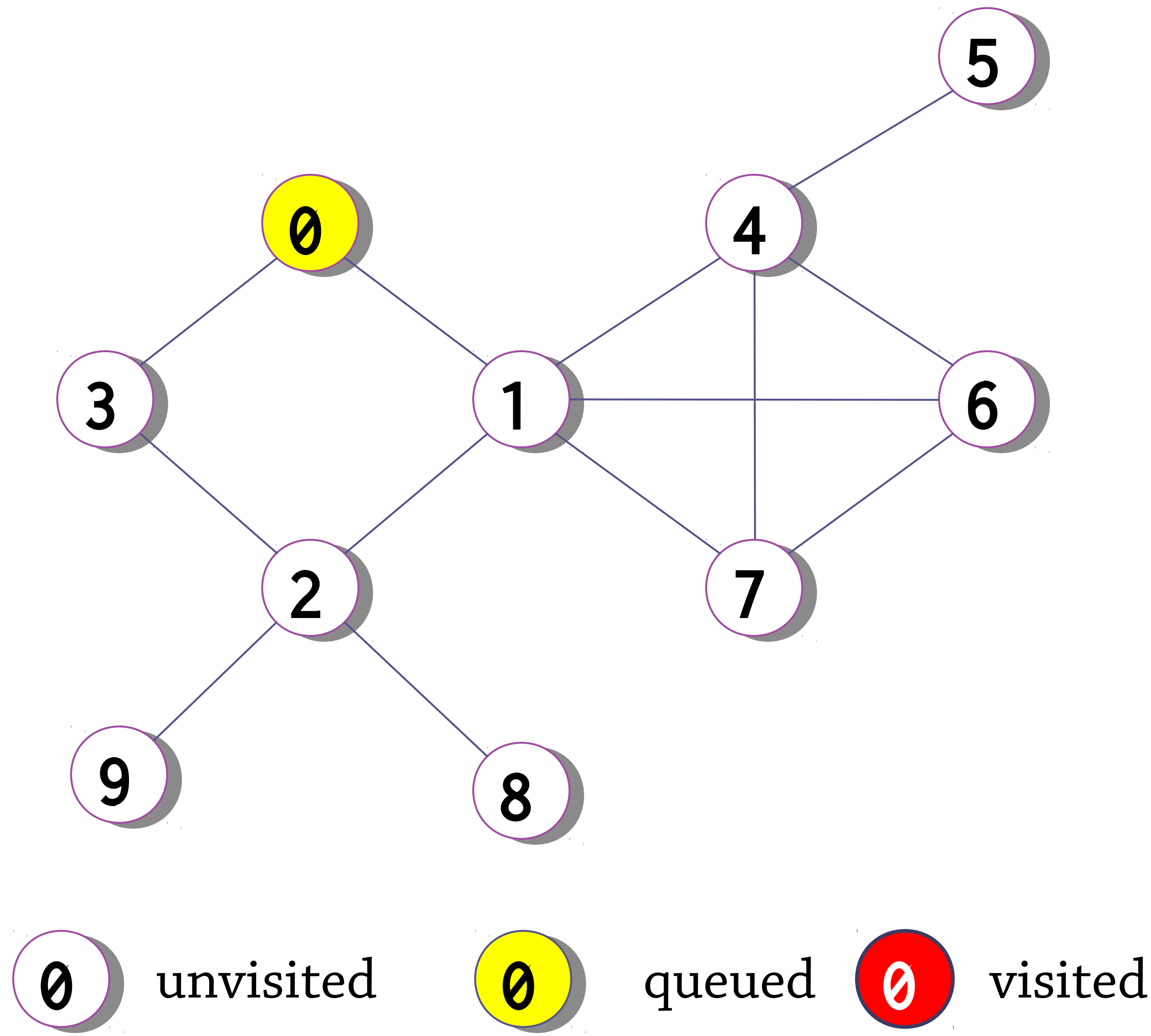
- Remove a node from the queue
- Visit it
- Find all adjacent nodes and add them to the queue, *unless* they've previously been added to the queue

Example of a breadth-first search

Queue:

0

Visit order:



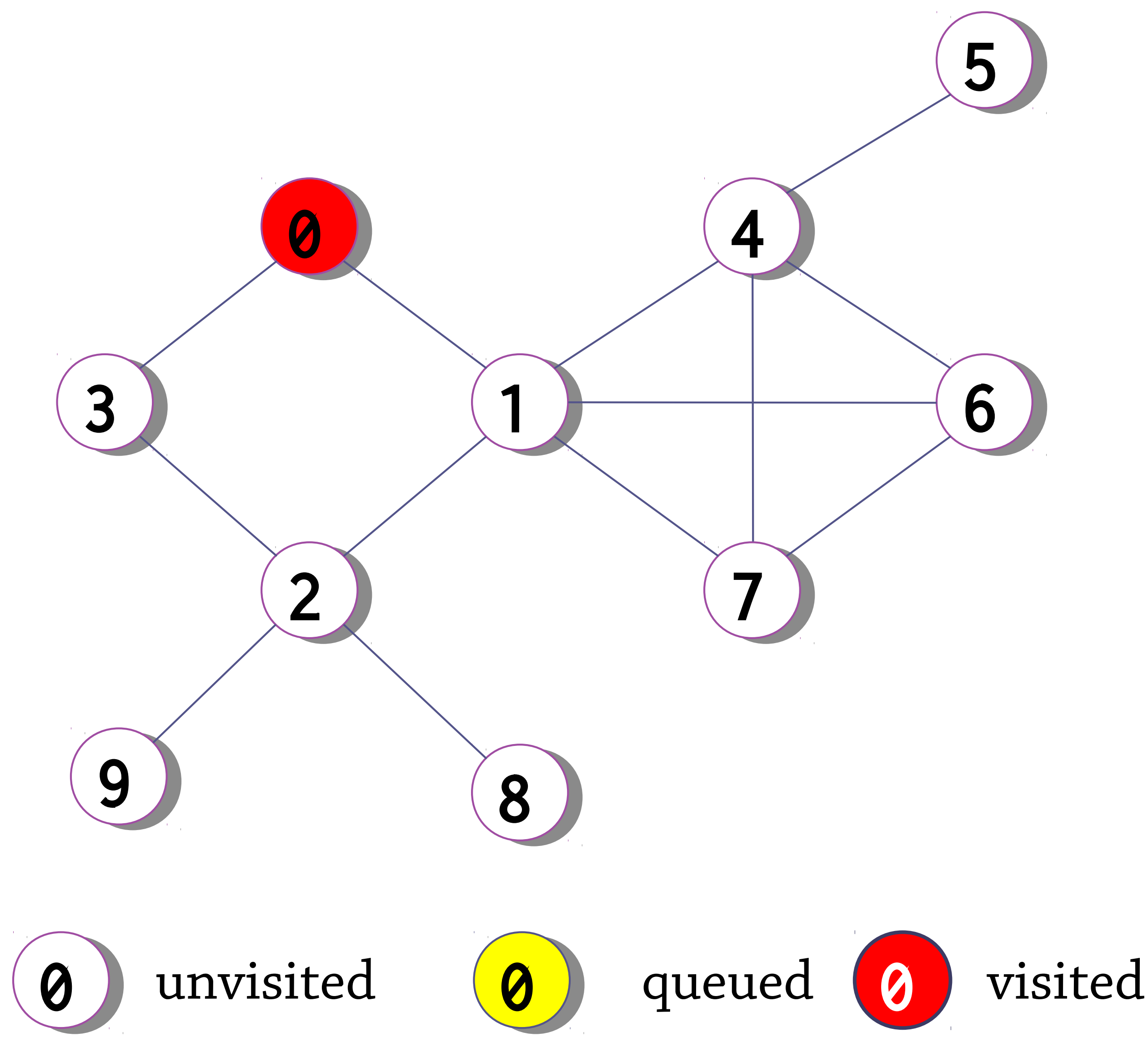
Initially,
queue contains
start node

Example of a breadth-first search

Queue:

Visit order:

0



Step 1:
remove node
from queue
and visit it

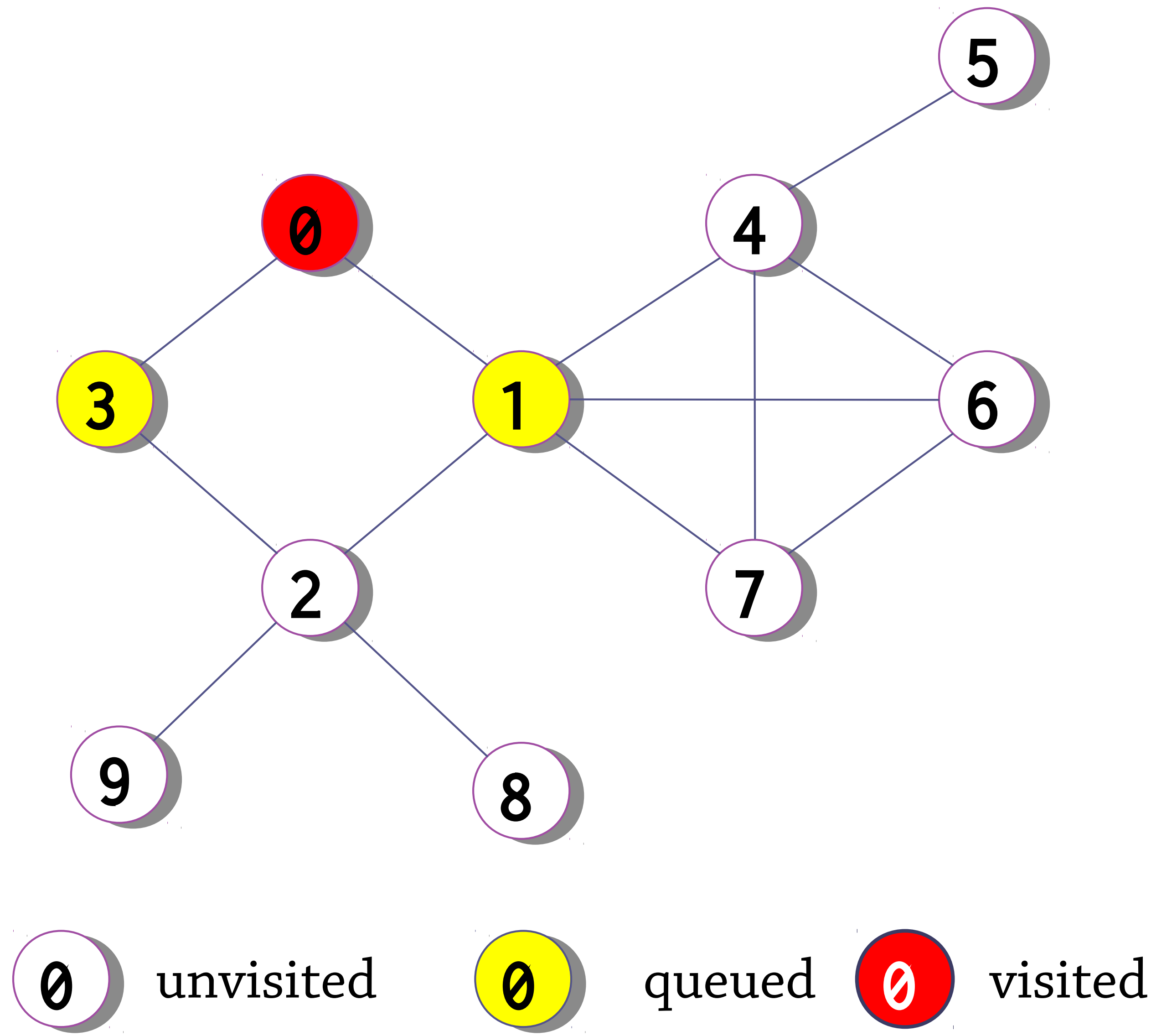
Example of a breadth-first search

Queue:

3 1

Visit order:

0



Step 2:
add adjacent nodes
to queue
(only unvisited ones)

Example of a breadth-first search

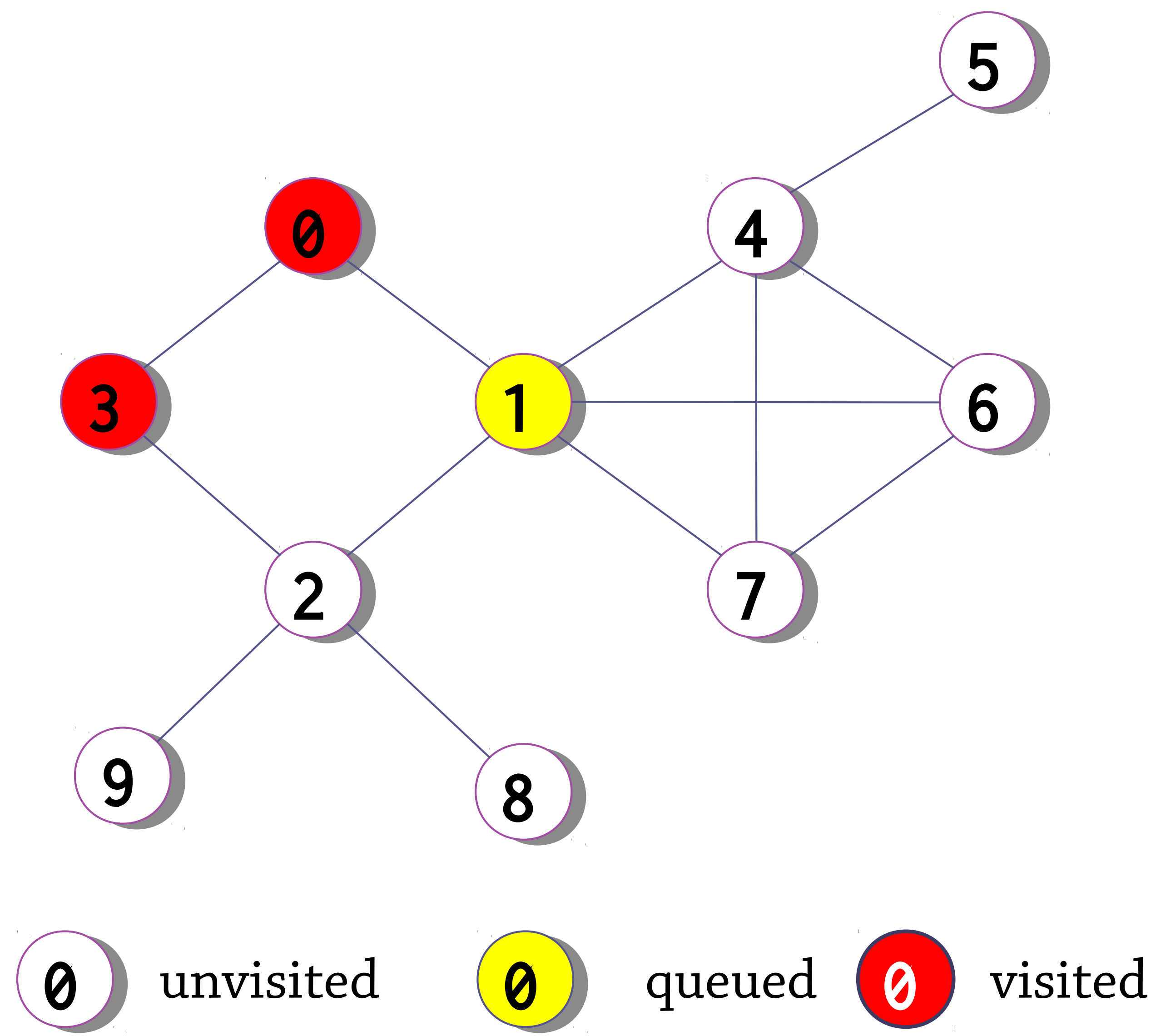
Queue:

1

Visit order:

0 3

Step 1:
remove node
from queue
and visit it



Example of a bre arch

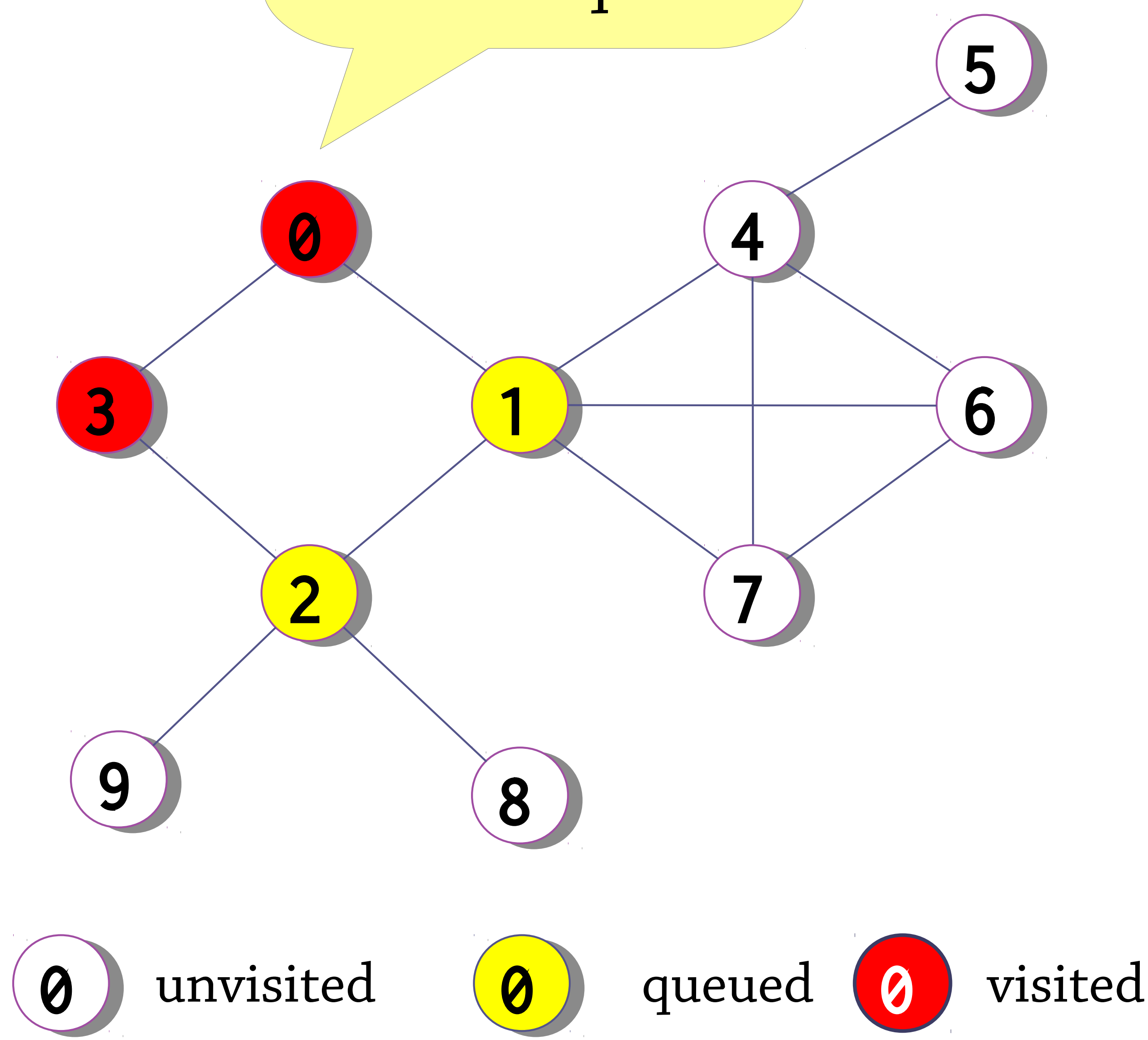
0 is already visited, so we don't add it to the queue

Queue:

1 2

Visit order:

0 3



Step 2:
add adjacent nodes to queue
(only unvisited ones)

Example of a breadth-first search

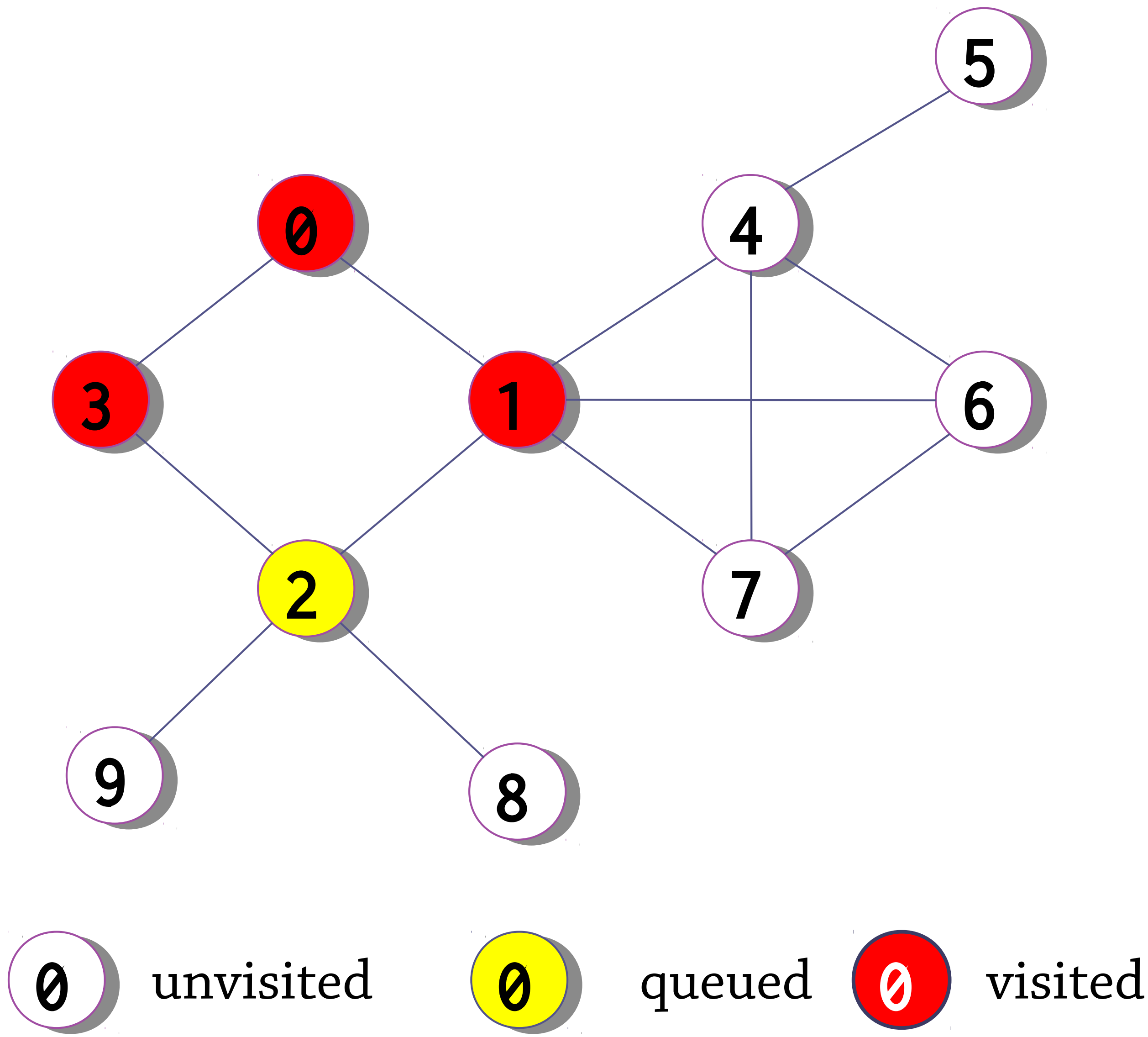
Queue:

2

Visit order:

0 3 1

Step 1:
remove node
from queue
and visit it



Example of a bre … arch

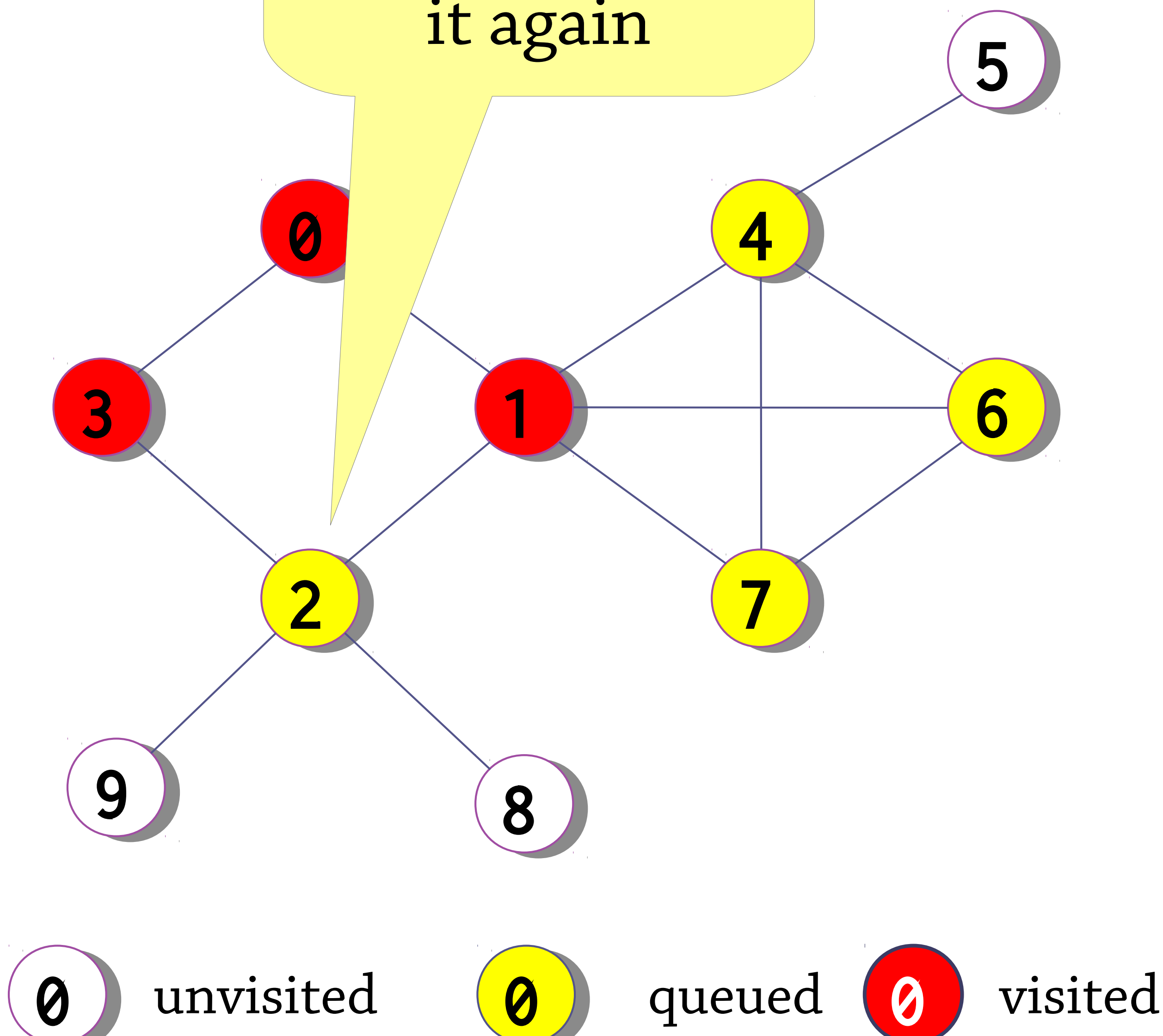
2 is already
in the queue, so
we don't add
it again

Queue:

2 4 6 7

Visit order:

0 3 1



Step 2:
add adjacent nodes
to queue
(only unvisited ones)

Example of a breadth-first search

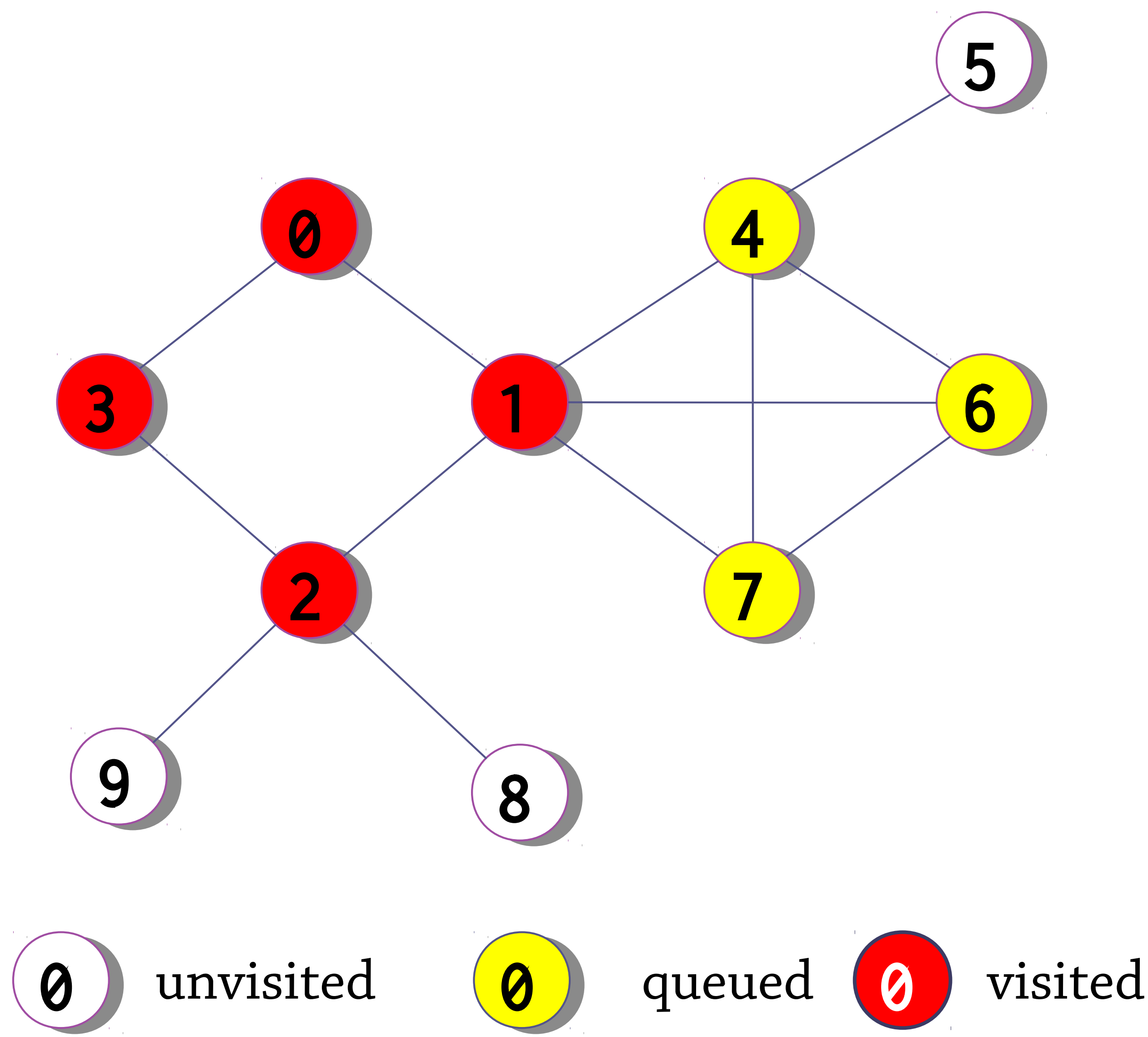
Queue:

4 6 7

Visit order:

0 3 1 2

Step 1:
remove node
from queue
and visit it



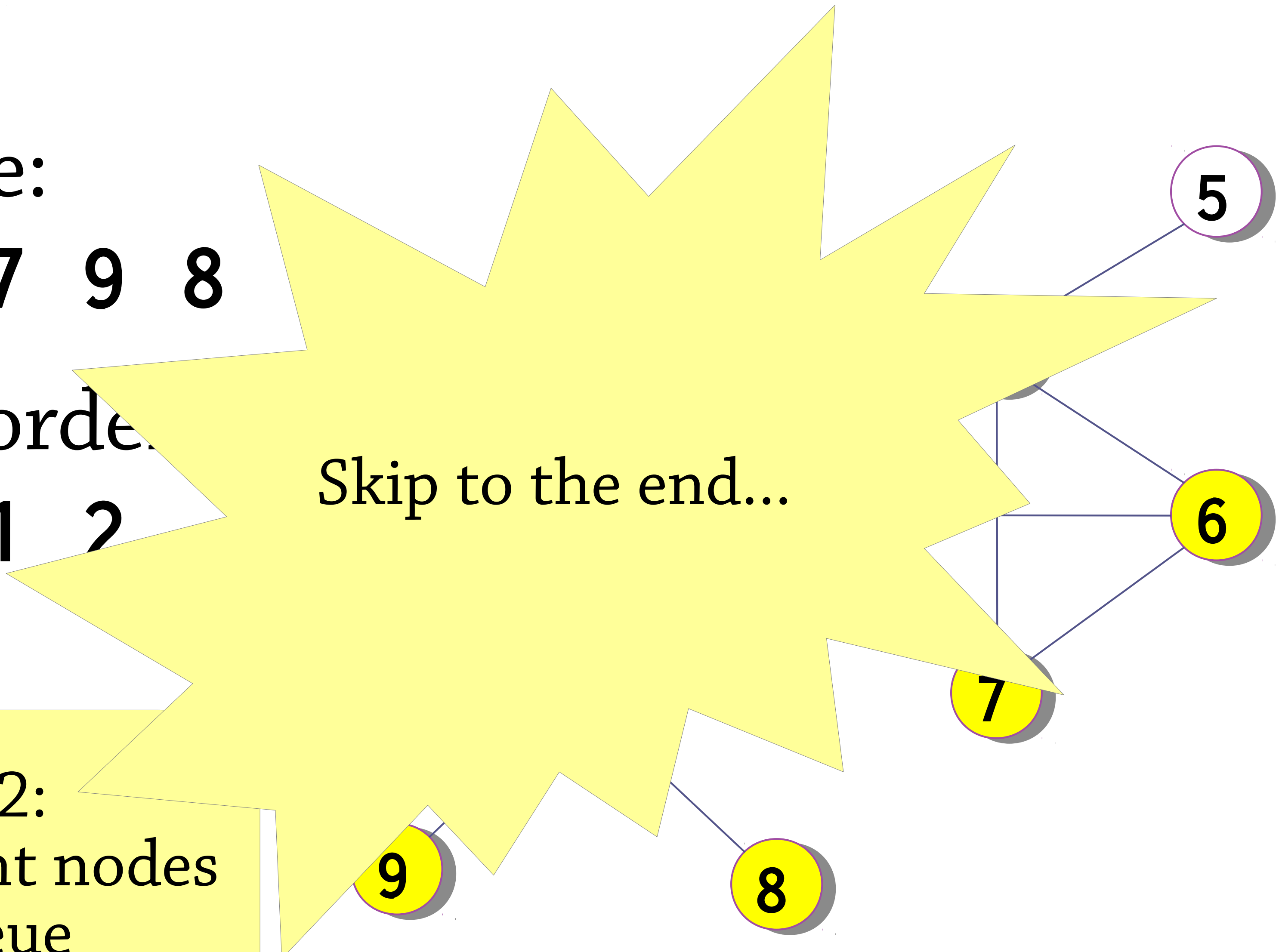
Example of a breadth-first search

Queue:

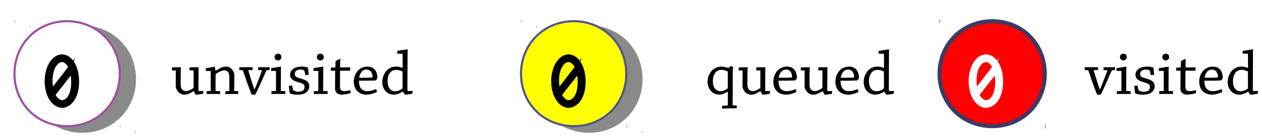
4 6 7 9 8

Visit orde

0 3 1 2



Step 2:
add adjacent nodes
to queue
(only unvisited ones)



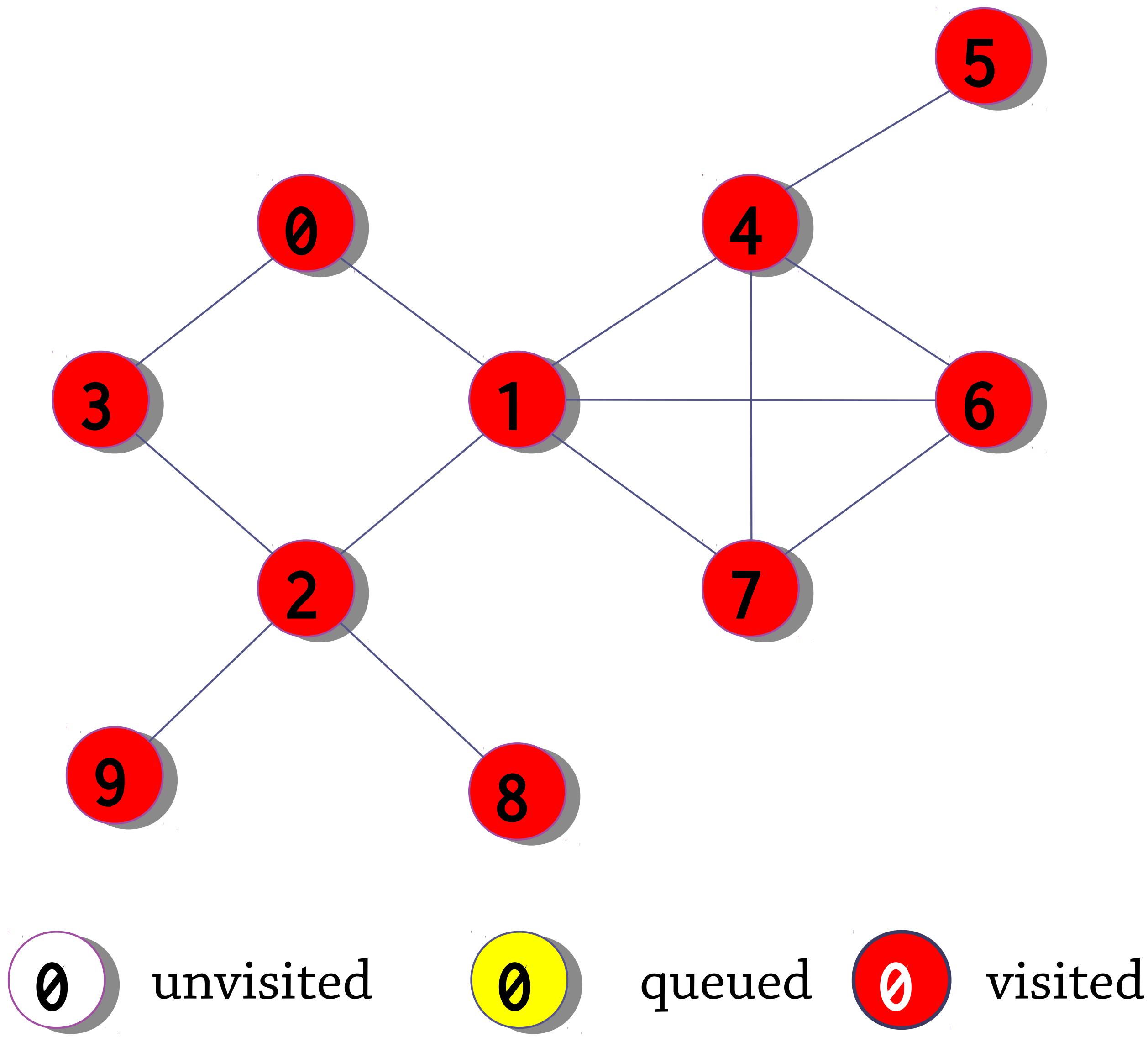
Example of a breadth-first search

Queue:

Visit order:

0 3 1 2 4
6 7 9 8 5

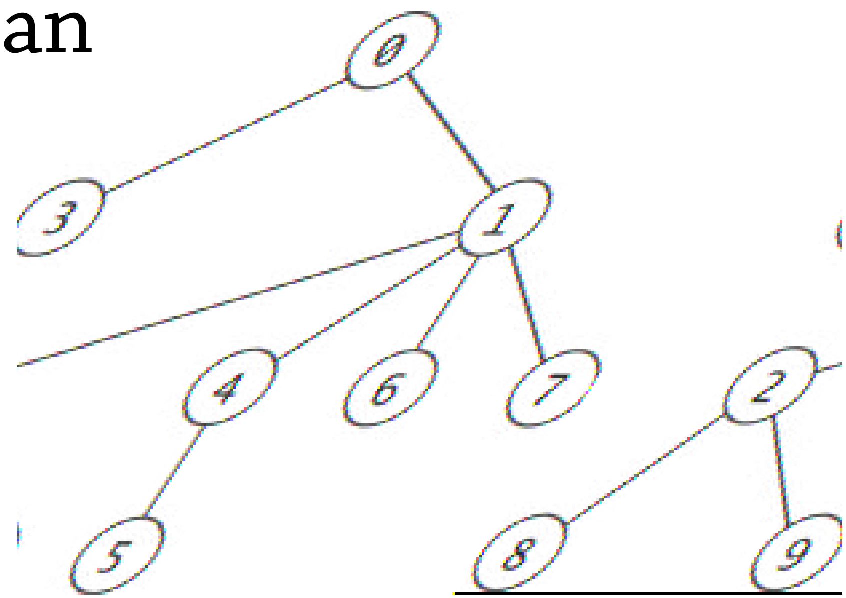
We reach step 1, but the queue is empty, and **we're finished!**



Breadth-first search tree

While doing the BFS, we can record *which node we came from* when visiting each node in the graph

(we do this when adding a node to the queue)



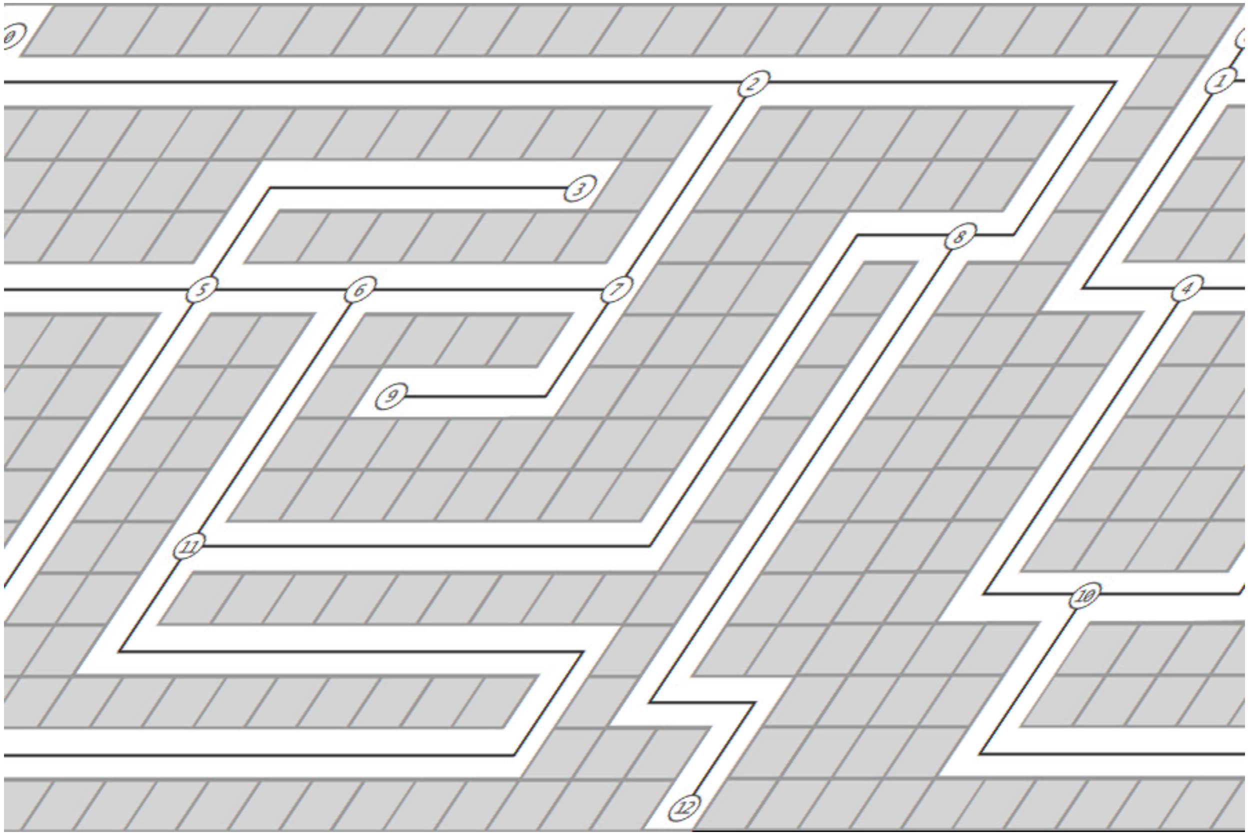
By doing this we can build a tree with the start node at the top (the *breadth-first search tree*)

Starting at a node in the tree, and following it up to the root, gives us the *shortest path* from each node to the start node

Example: unweighted shortest path

We can represent a maze as a graph – nodes are junctions, edges are paths.

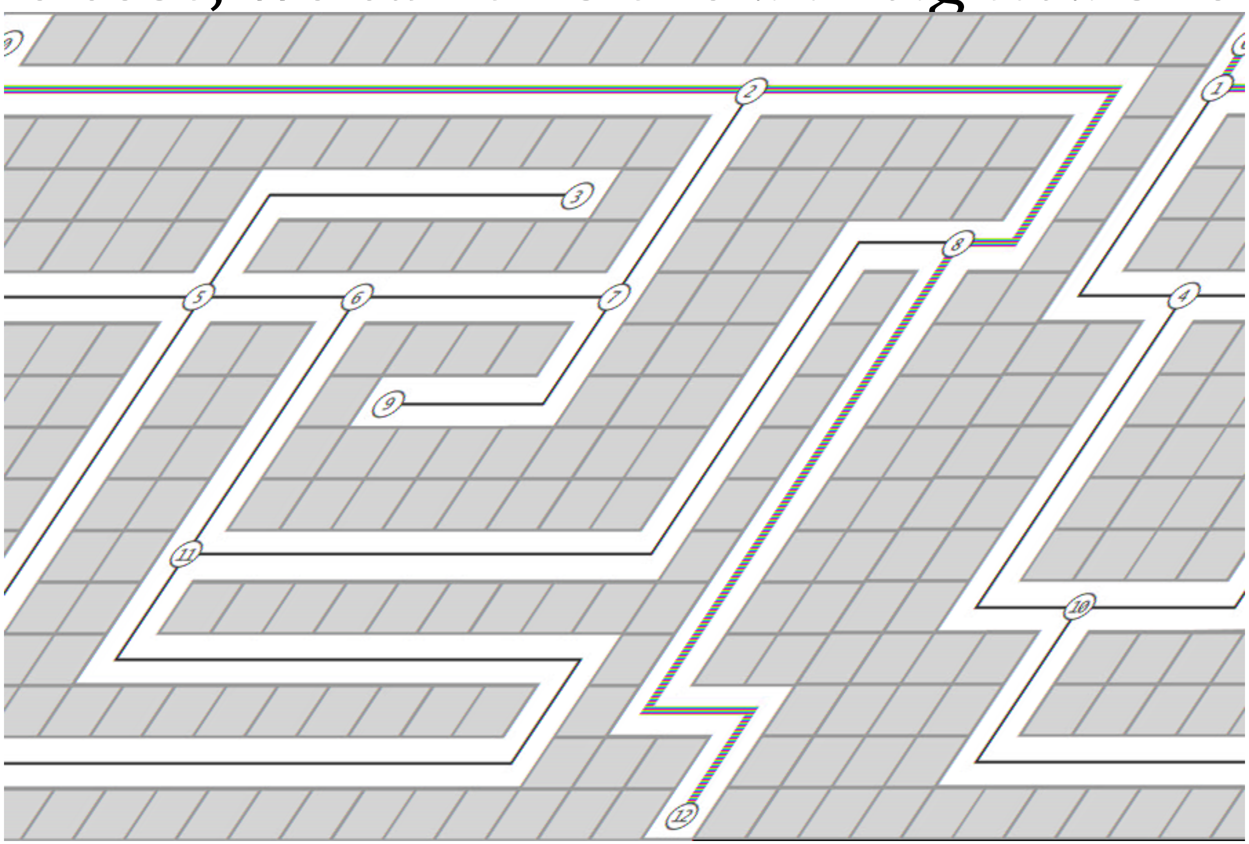
How can we find a path from the entrance to the exit?



Example: unweighted shortest path

A breadth-first search tree starting from the entrance gives us a path to any node (including the exit)

This path minimises *number of junctions* – each edge has the same cost, we call this the *unweighted* shortest path



Depth-first search

Depth-first search is an alternative search order that's easier to implement

To do a DFS starting from a node:

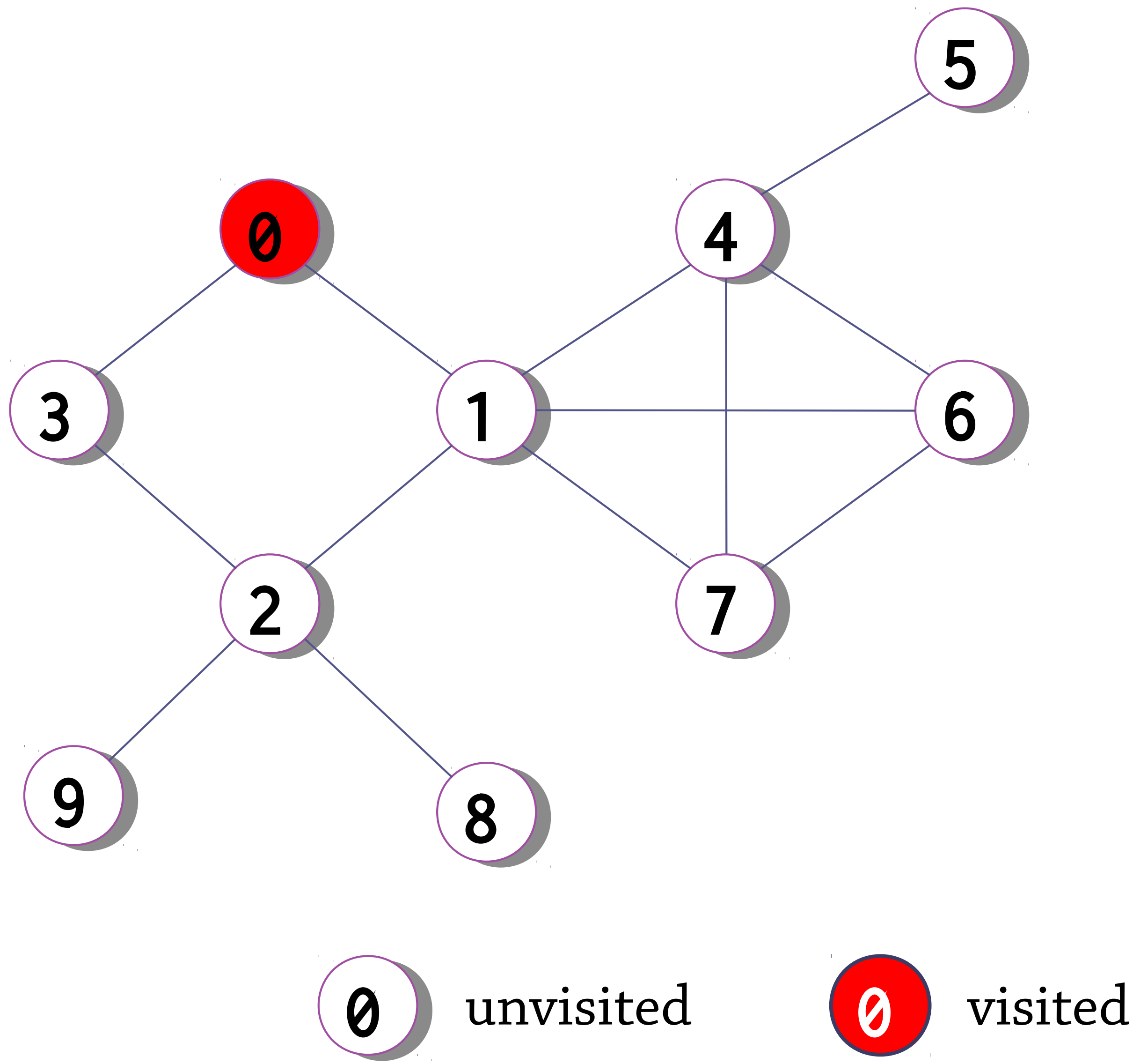
- visit the node
- recursively DFS all adjacent nodes (skipping any already-visited nodes)

Much simpler!

Example of a depth-first search

Visit order:

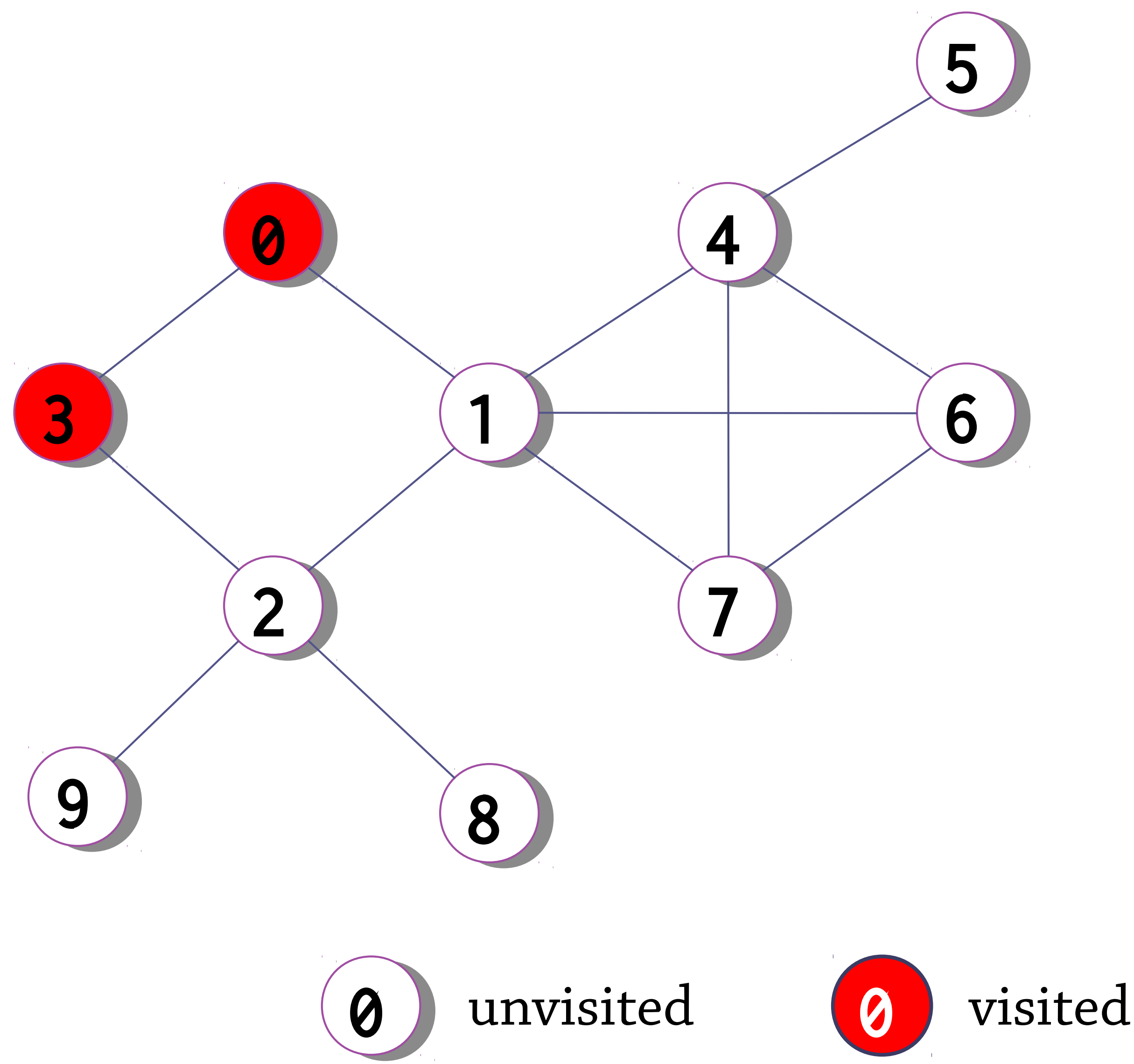
0



Example of a depth-first search

Visit order:

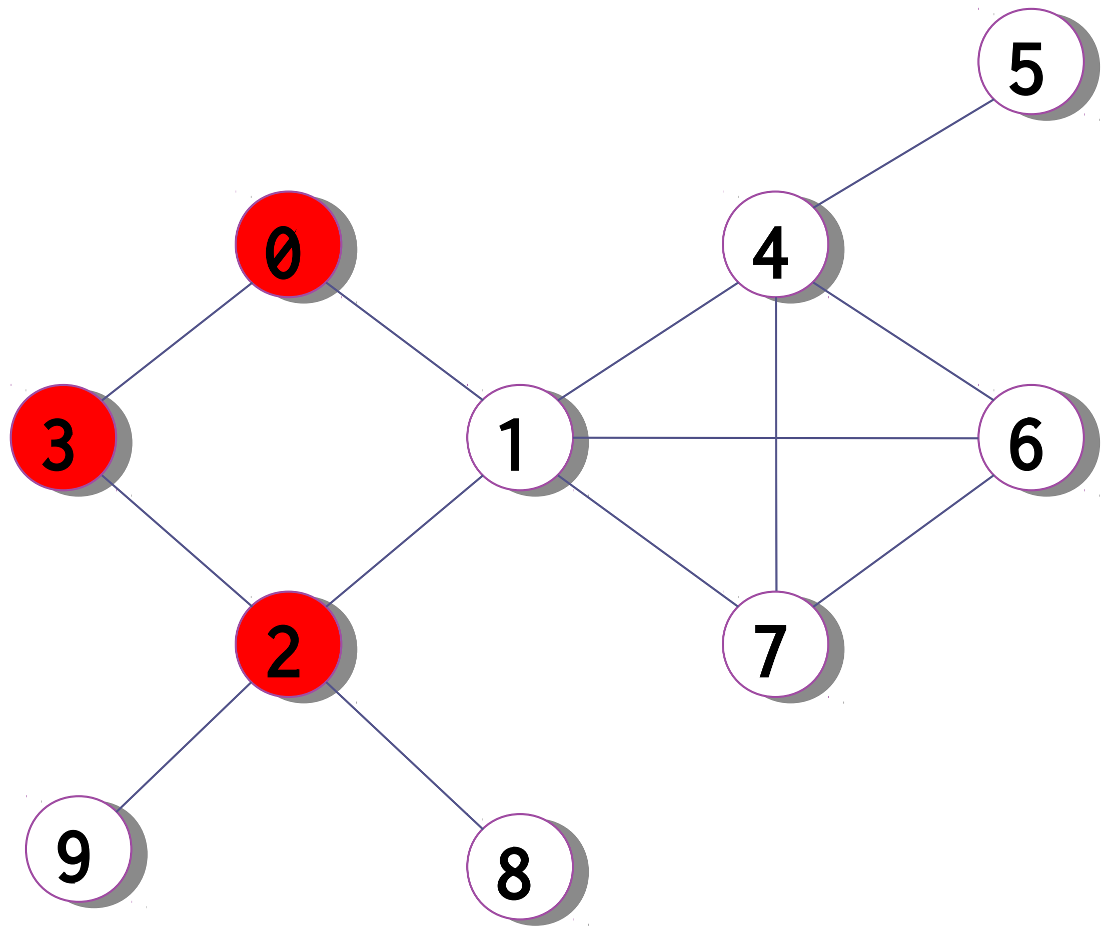
0 3



Example of a depth-first search

Visit order:

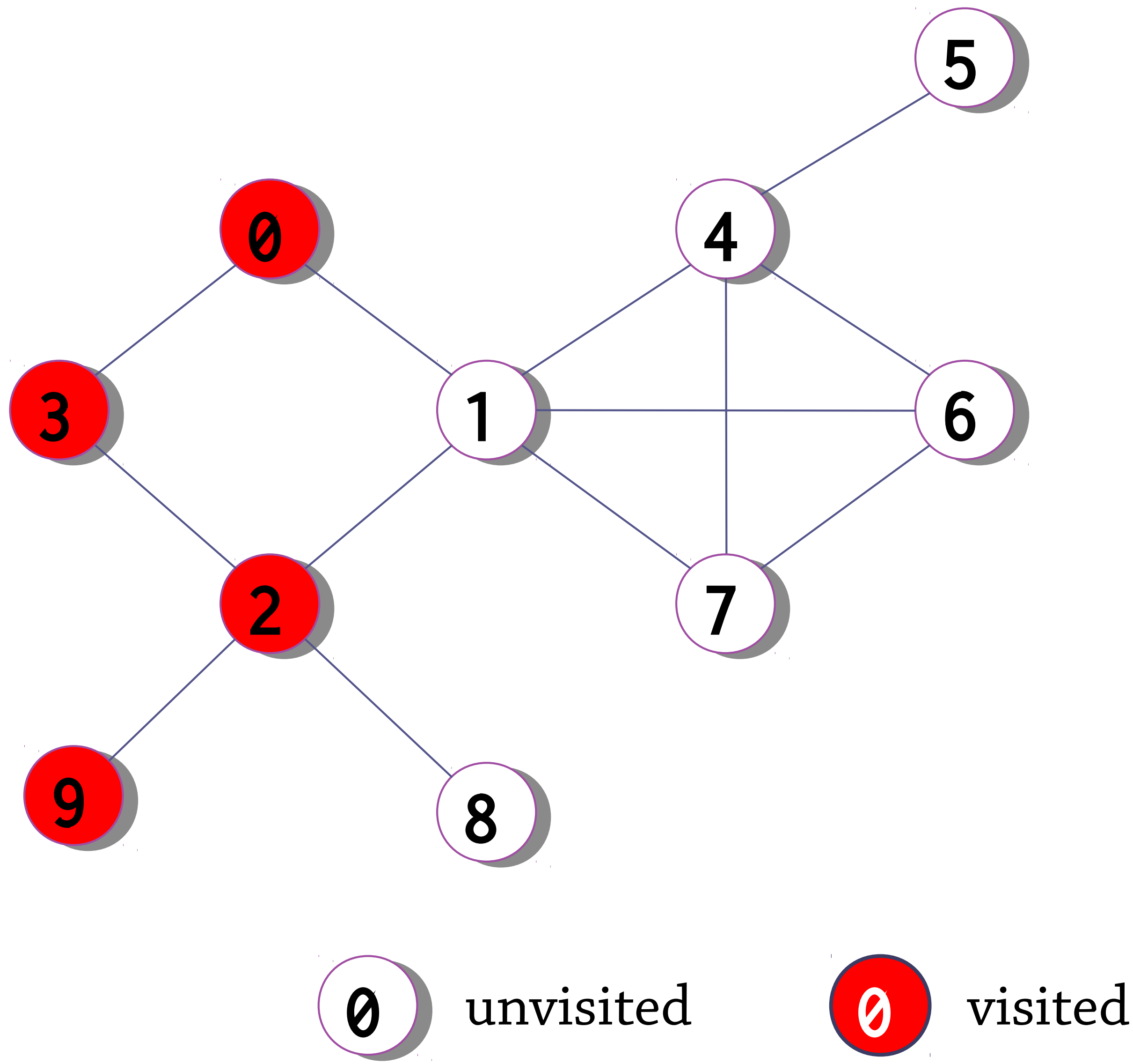
0 3 2



Example of a depth-first search

Visit order:

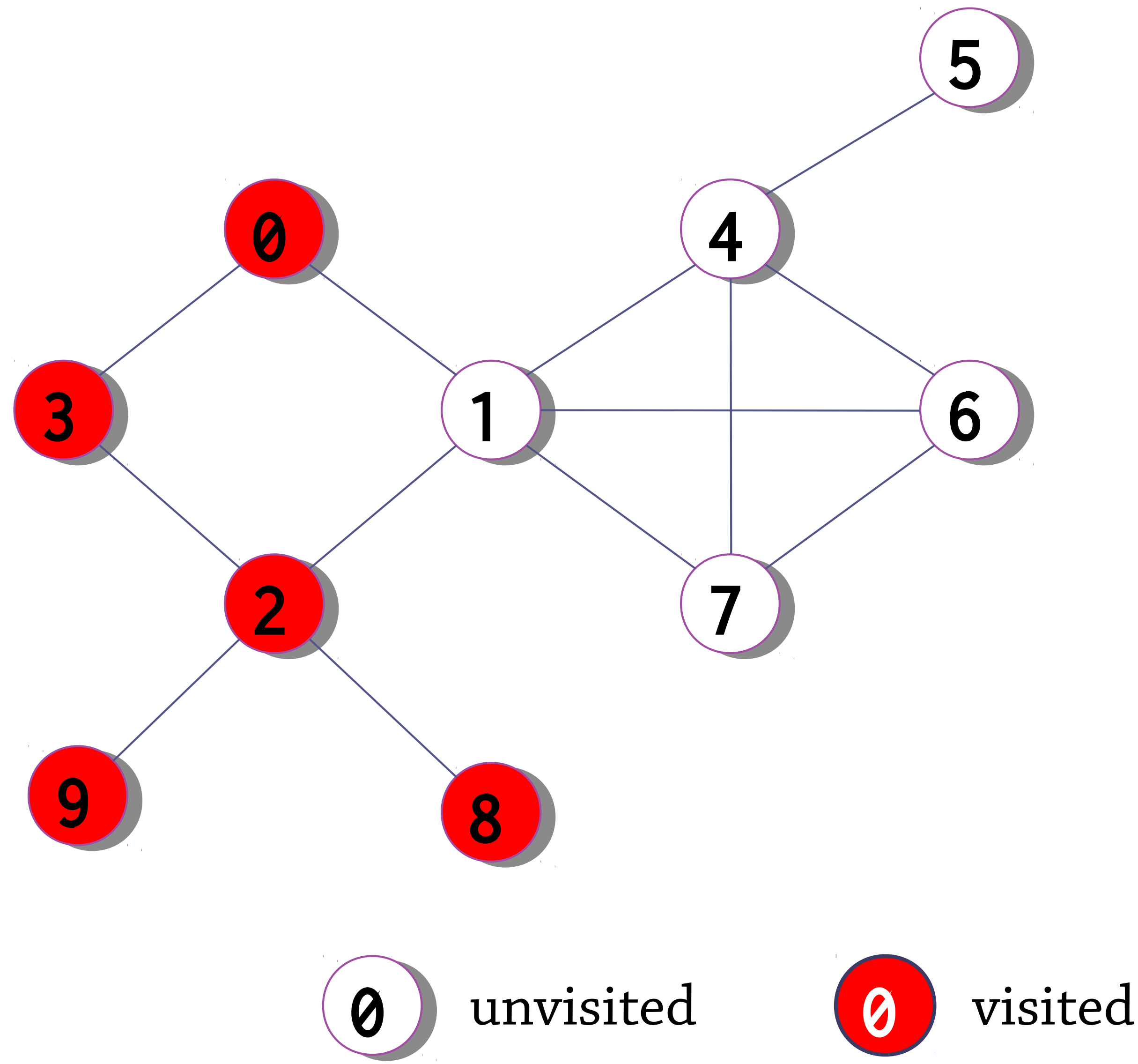
0 3 2 9



Example of a depth-first search

Visit order:

0 3 2 9 8



Depth-first search, alternative order

A variation of DFS, where we visit each node *after* visiting the adjacent nodes.

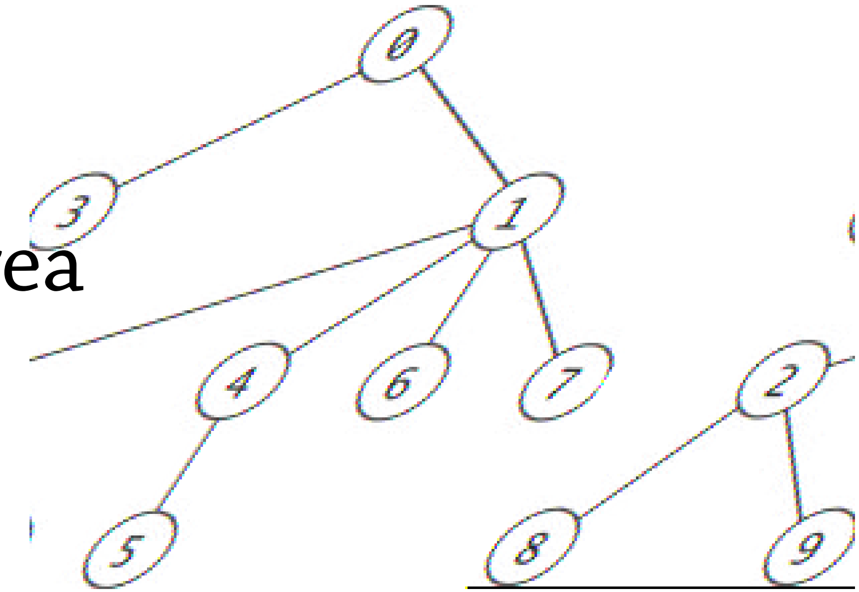
To do a DFS starting from a node:

- mark the node as visited
- recursively DFS all adjacent nodes (skipping any already-visited nodes)
- visit the node itself

(Wikipedia calls the order of nodes a *postordering*, compared to a *preordering* for the normal DFS)

What order would we visit the nodes in on the previous example?

BFS vs DFS



BFS visits the nodes in a "fair" order: the search area widens gradually

E.g. on a tree: first visit the root, then the root's children, then grandchildren, and so on.

DFS will explore a whole branch of the tree before backtracking and trying a different branch – the order is much more unpredictable which makes it unsuitable for some algorithms (e.g. on the tree to the right, you may explore 3 directly after 0, or you may explore it last)

Implementing **depth**-first search

We maintain a ***stac***
going to visit next

- Initially, the **stack** co

We repeat the follo

- Remove a node from
- Visit it
- Find all nodes adjacent to the visited node and add them to the **stack**, *unless* they have been visited or added to the **stack** already

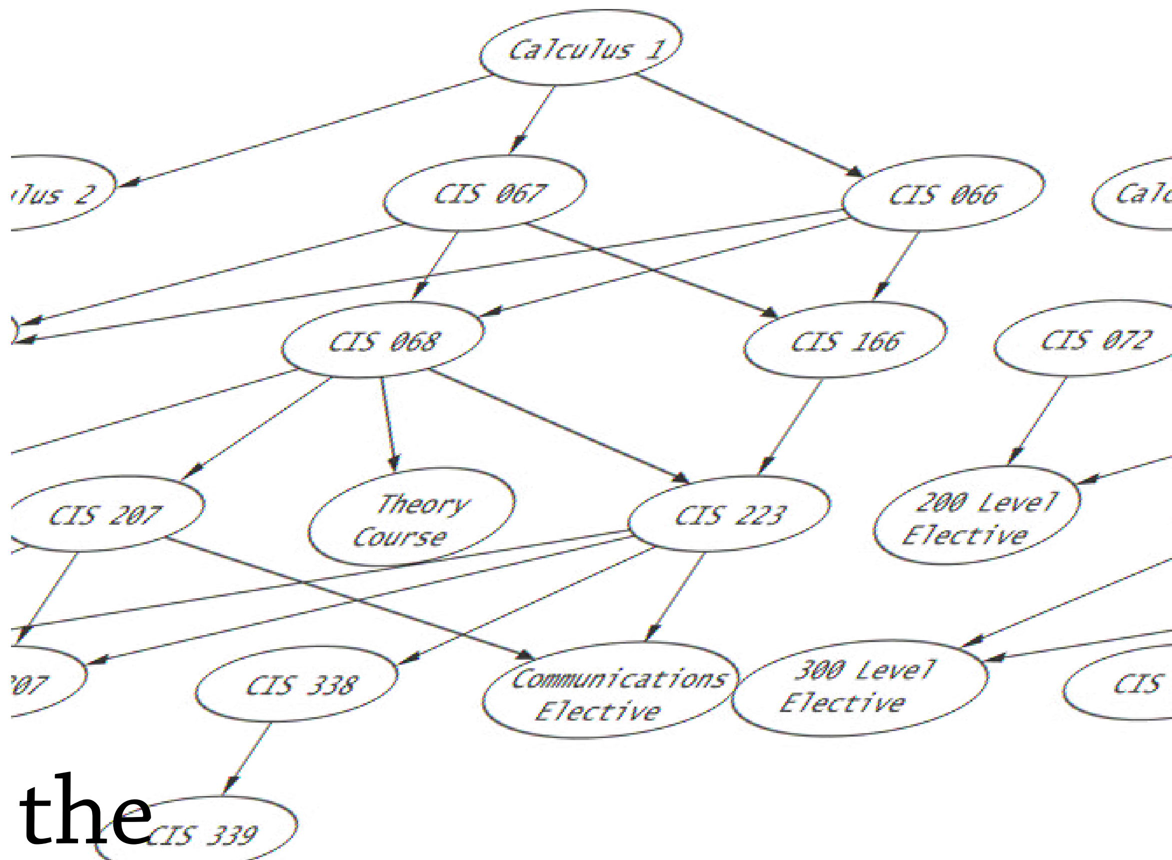
We can also implement DFS by taking the BFS algorithm and using a stack instead of a queue!

The recursive implementation uses the *call stack* to do this implicitly

Directed acyclic graphs

Here is a directed acyclic graph (DAG)

A DAG is a directed graph without cycles



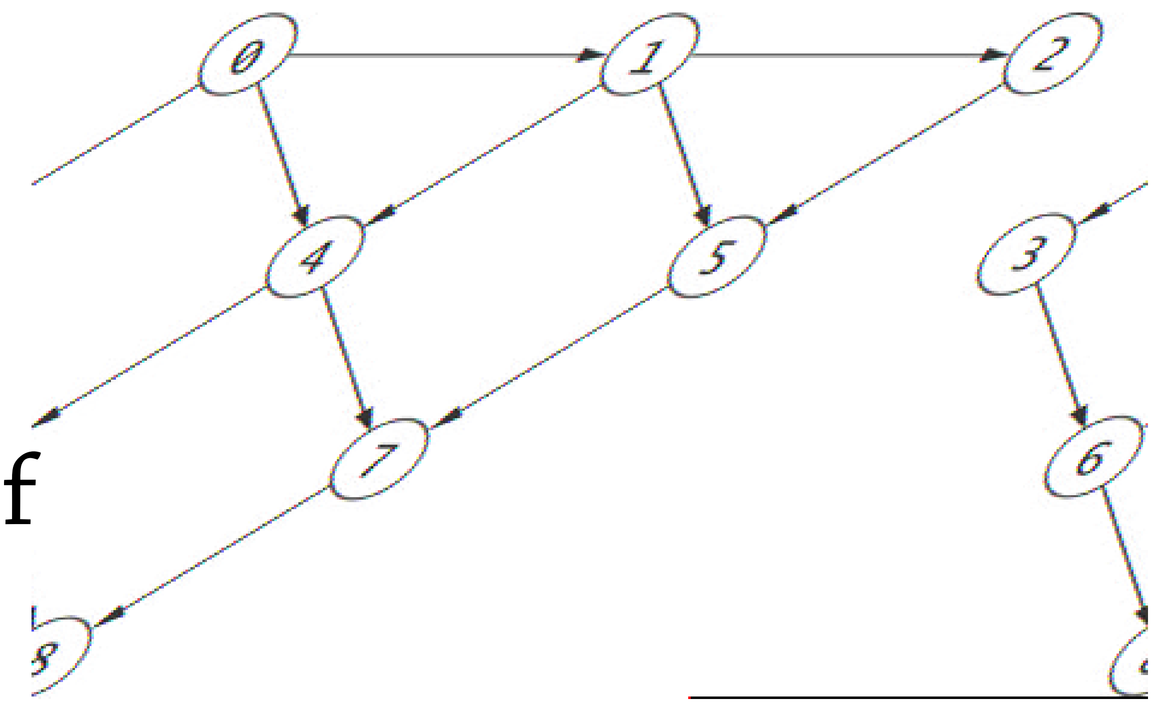
That means: once you follow an edge there is no way back to the source node – we can say that one node is *after* another in the graph

Example: topological sort

A *topological sort* of the nodes in a DAG is a list of all the nodes, such that *if (u, v) is an edge, then u comes before v in the list*

Every DAG has a topological sort, often several

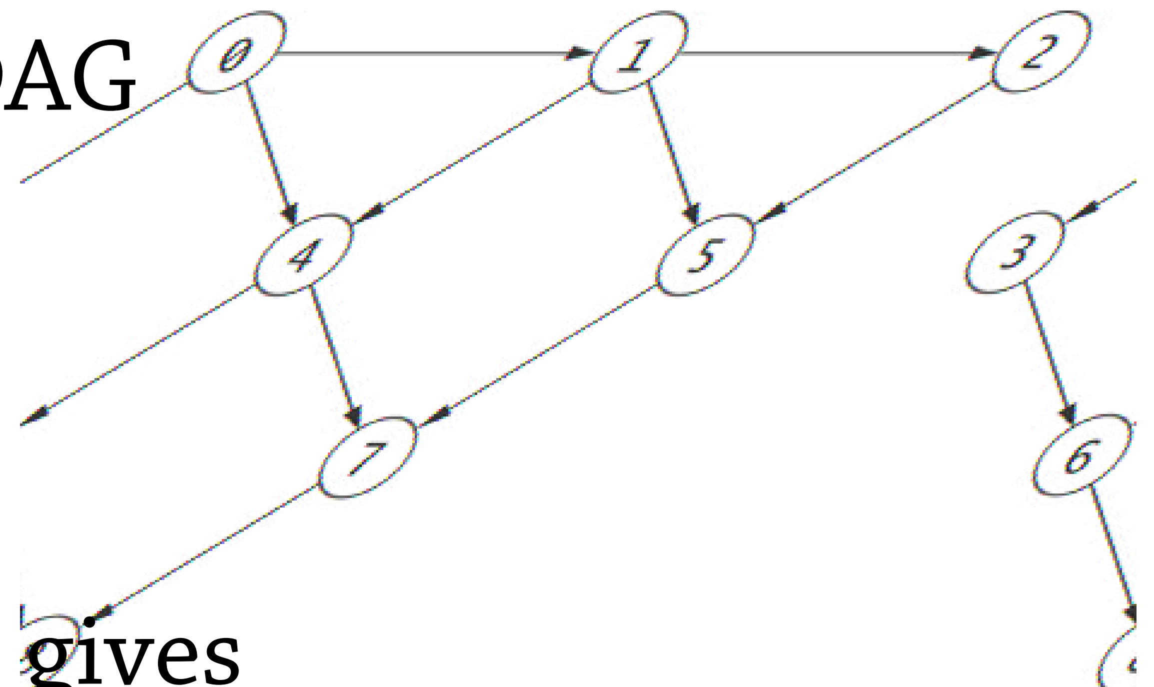
012345678 is a topological sort of this DAG, but 015342678 isn't.



Example: topological sort

An example: if nodes are tasks, and an edge (u, v) means "task u must be done before task v", then:

If the graph is a DAG it means there are no impossible dependencies between tasks



A topological sort gives a valid order to do the tasks in

Topological sort

We can use a depth-first search to topologically sort the graph:

- Suppose that we do a DFS but using the alternative version where we visit each node only after visiting the adjacent nodes
- If (u, v) is an edge, we will then visit u *after* we visit v – we will only visit a node once we've visited all nodes that come after it
- This is the exact *opposite* order to what we want for a topological sort!
- So, to topologically sort a graph, do a DFS, then return the nodes in the reverse order you visited them

Summary

Graphs:

- many varieties – directed, undirected, weighted, unweighted
- all are variations on the same basic theme
- graphs can be cyclic or acyclic (*directed acyclic graphs* very common)
- paths, cycles, connected components

Implementing them:

- adjacency lists – good for sparse graphs
- adjacency matrix – good for dense graphs
- very often you don't use either, you just treat your set of objects as a graph!

Some basic algorithms:

- breadth-first and depth-first search
- unweighted shortest path using BFS
- topological sort using DFS