Lab deadlines

Some groups have had trouble making the lab 2 final deadline, so I've moved the deadlines a bit:

- For lab 2, the final deadline is *this Friday*
- For lab 3, the deadline is next Friday, the 23rd (there's no separate first and final deadline)

If you miss the deadline, there will be a chance after the end of the course to pass the lab by showing me it in person

Note on copying

It hardly needs to be said, but…

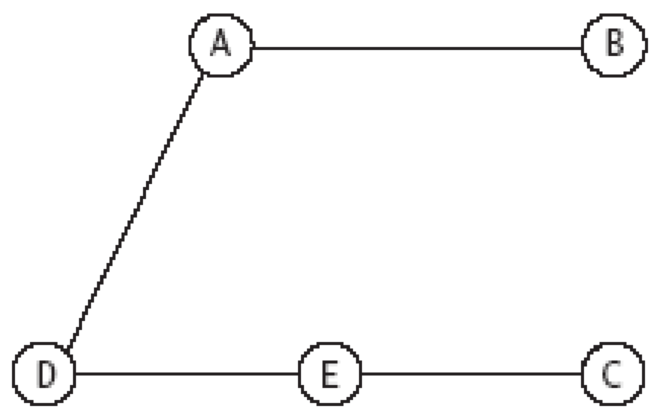
- The labs are part of the examination of the course, and as such the work your group submits must be the work of your group alone
- Although I don't mind you discussing ideas between groups, you **must not** copy from another group!
- GU considers this cheating, and both the person who copies a solution, *and the person who lets their solution be copied*, can get in serious trouble

Graphs *(chapter 13)*

Terminology

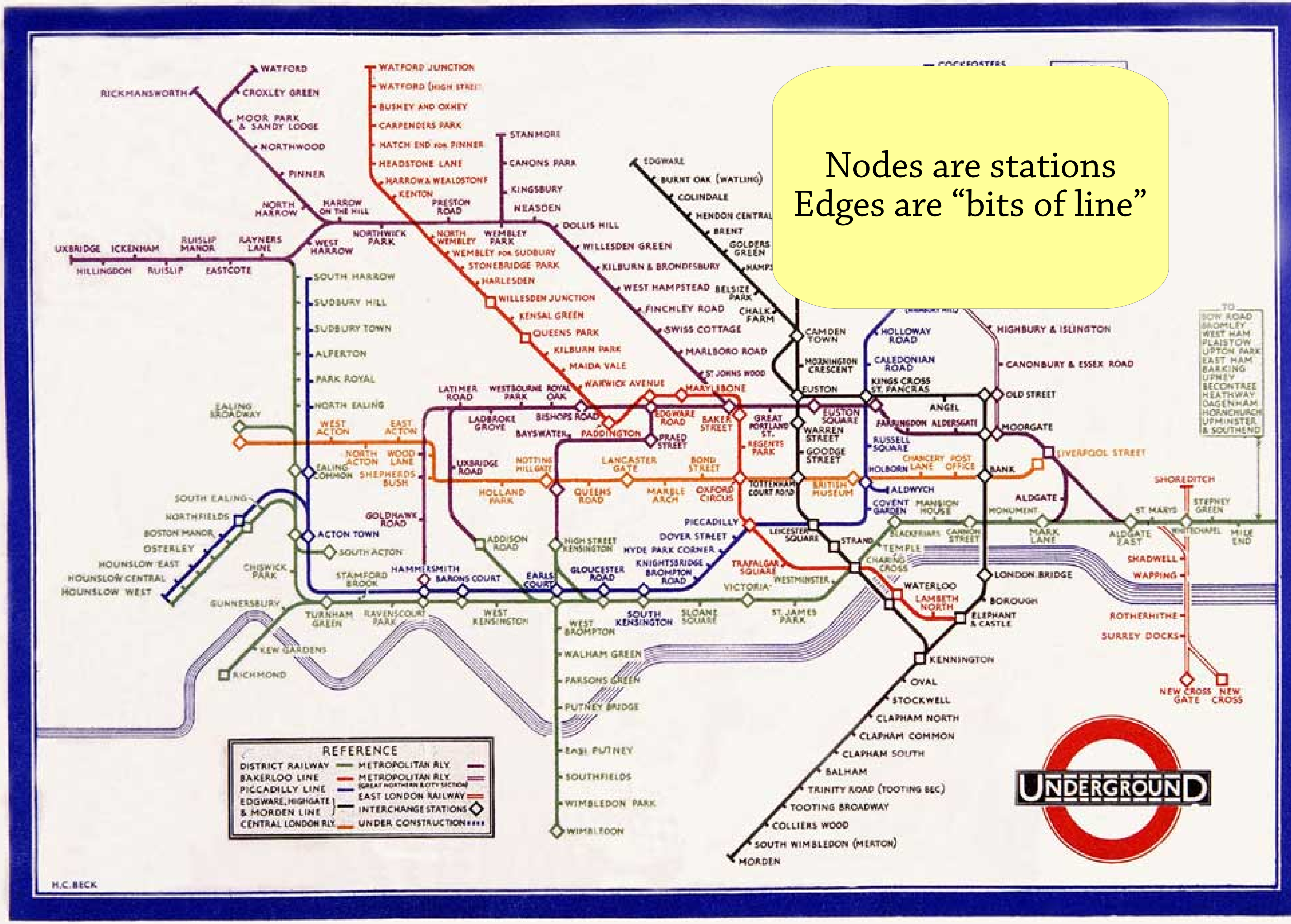
A graph is a data structure consisting of *nodes* (or *vertices*) and *edges*

- An edge is a connection between two nodes

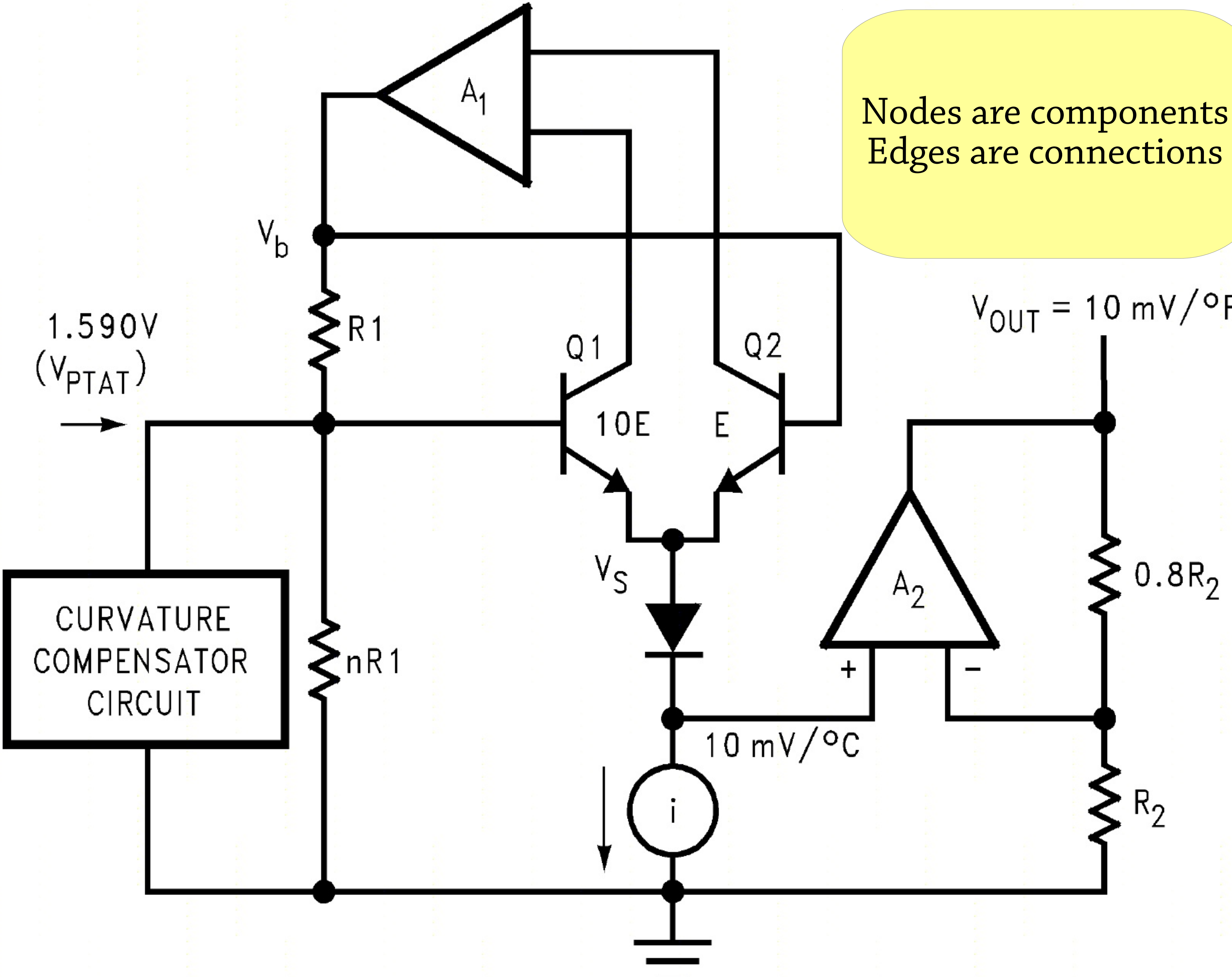


Nodes: A, B, C, D, E

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



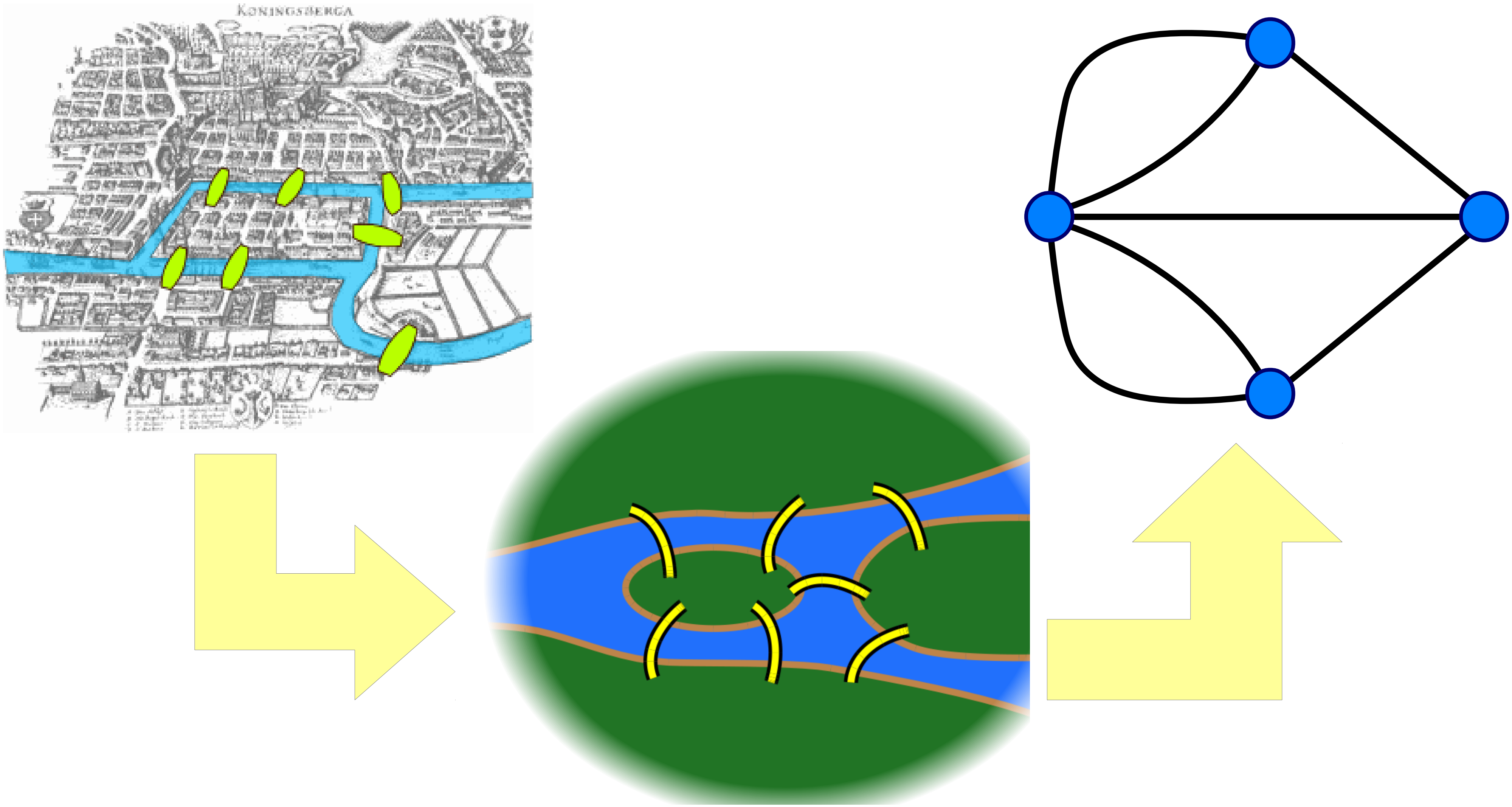
Nodes are stations
Edges are "bits of line"



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
- finding good ways to lay out components in an integrated circuit
- etc.

Anywhere where you have things, and relationships between things!

More graphs

Graphs can be either *directed* or *undirected*

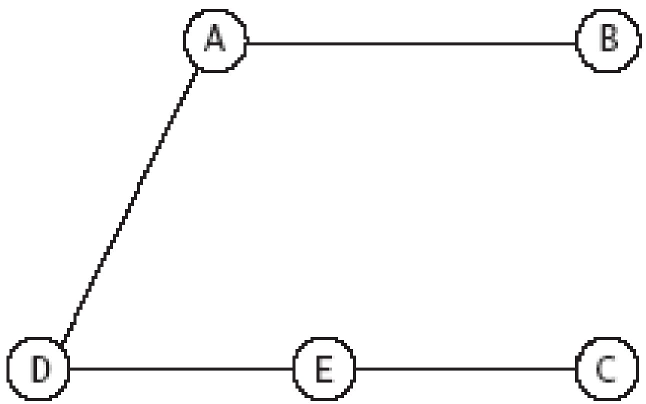
- In an undirected graph, an edge simply connects two nodes
- In a directed graph, one node of each edge is the source and the other is the target (we draw an arrow from the source to the target)

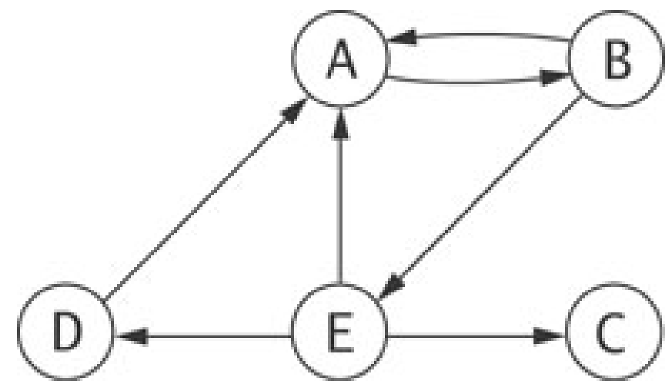
A tree is a special case of a directed graph

- Edge connecting parent to children
- But in a tree, each node can only have one parent – in a directed graph, it could have several

Drawing graphs

We represent nodes as points, and edges as lines – in a directed graph, edges are arrows:



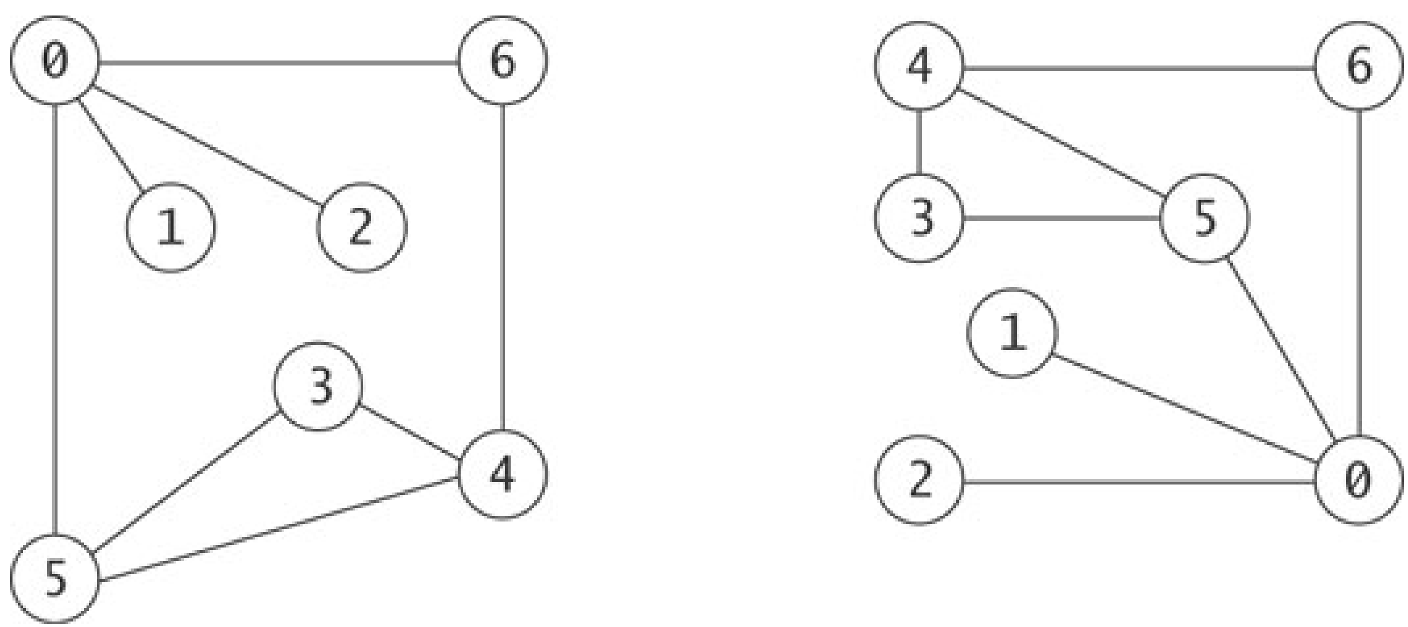


V = {A, B, C, D, E}
E = {(A, B), (A, D),
(C, E), (D, E)}

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

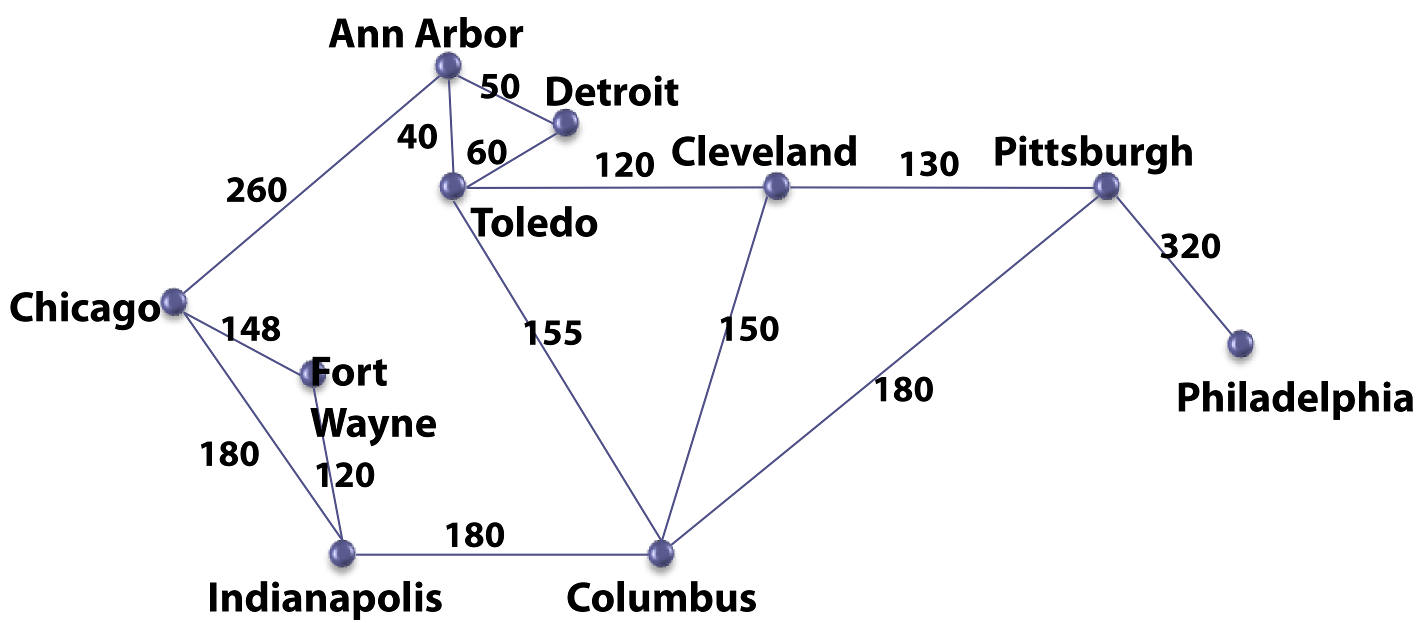


$V = \{0, 1, 2, 3, 4, 5, 6\}$

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

Weighted graphs

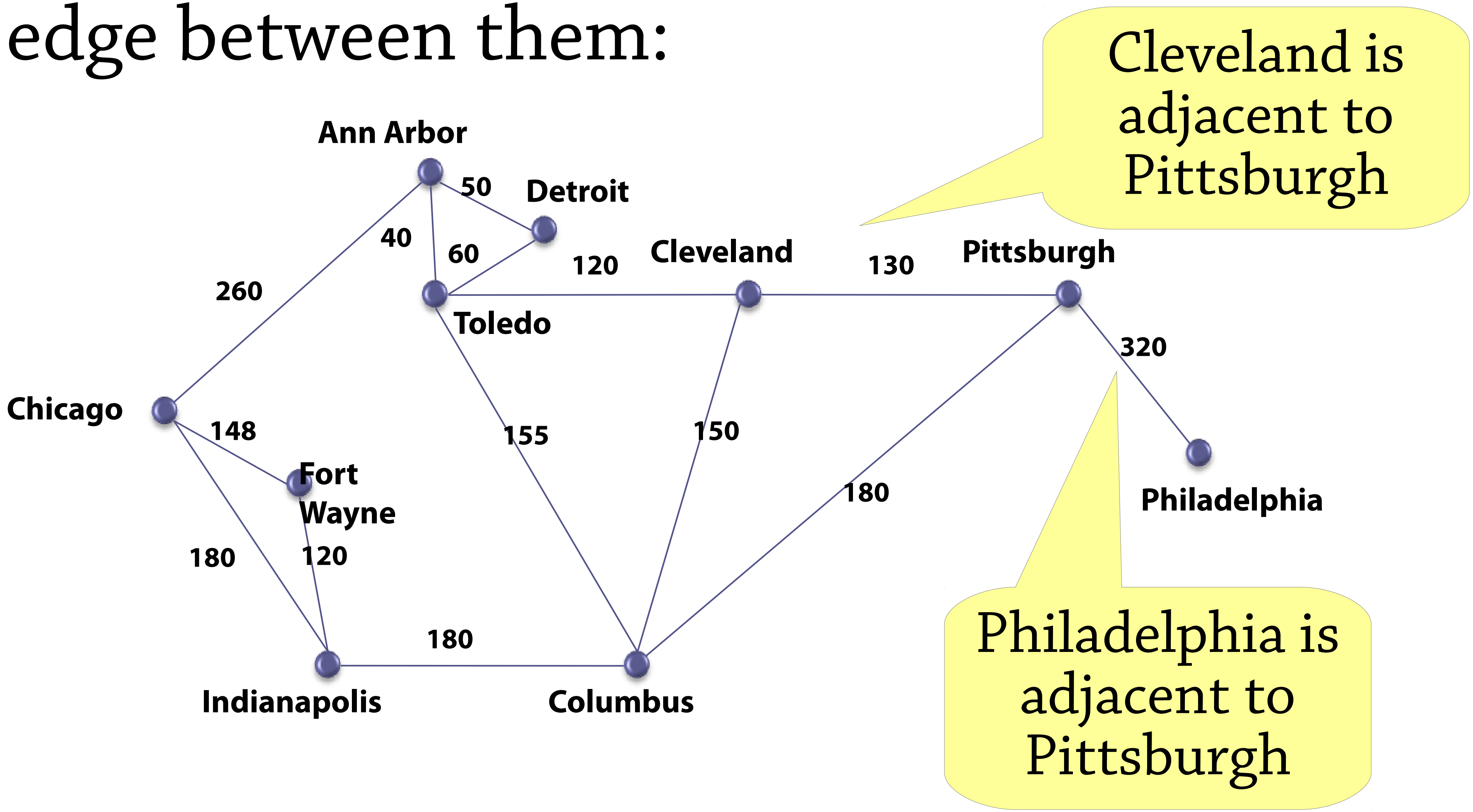
In a *weighted graph*, each edge has a *weight* associated with it:



A graph can be directed, weighted, neither or both

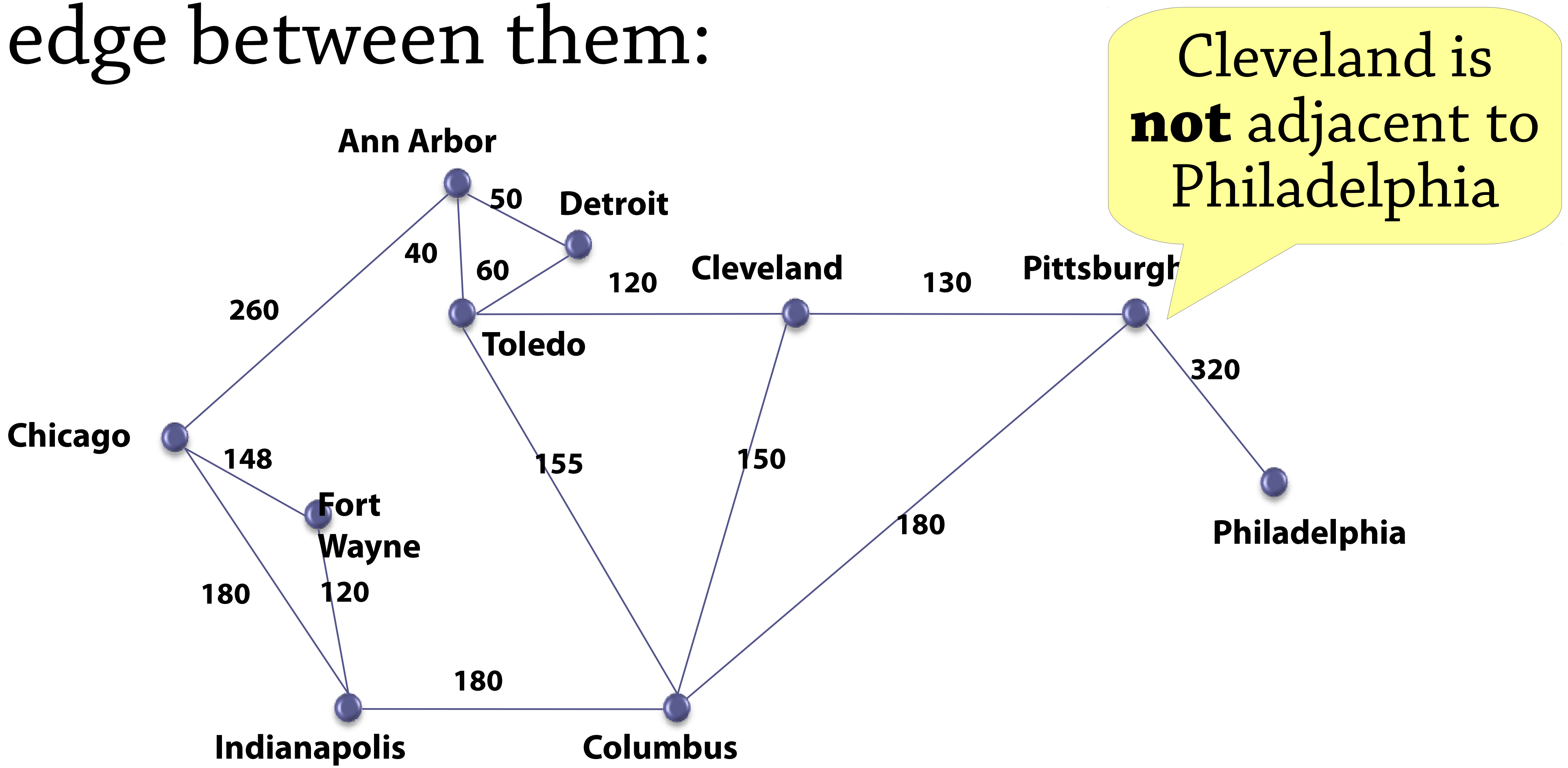
Paths and cycles

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



Paths and cycles

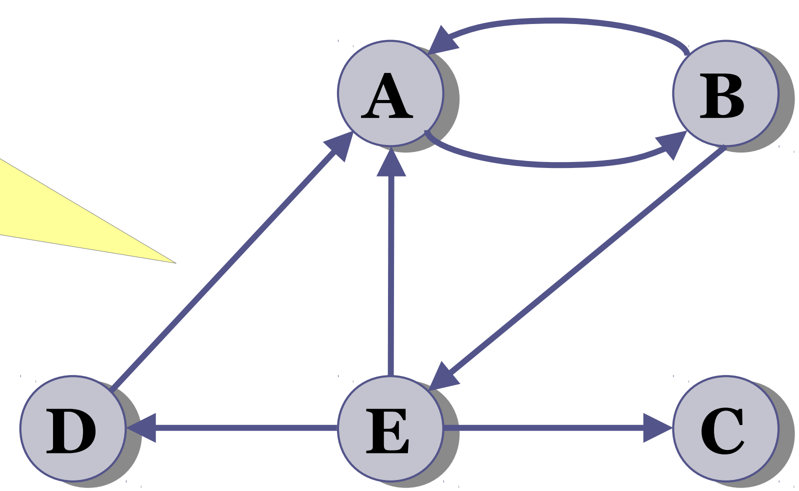
Two vertices are *adjacent* if there is an edge between them:



Paths and cycles

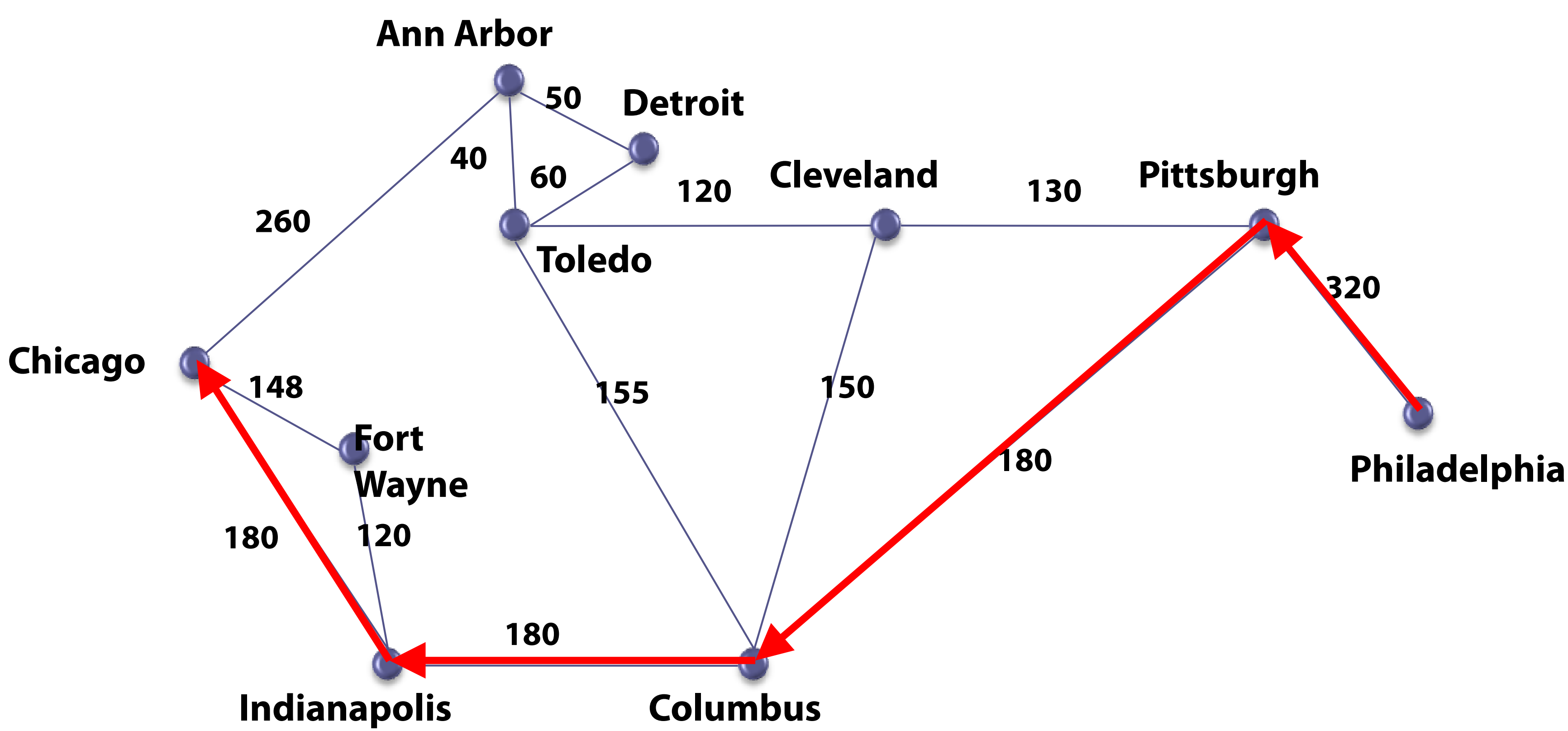
In a directed graph, the *target* of an edge is adjacent to the *source*, not the other way around:

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



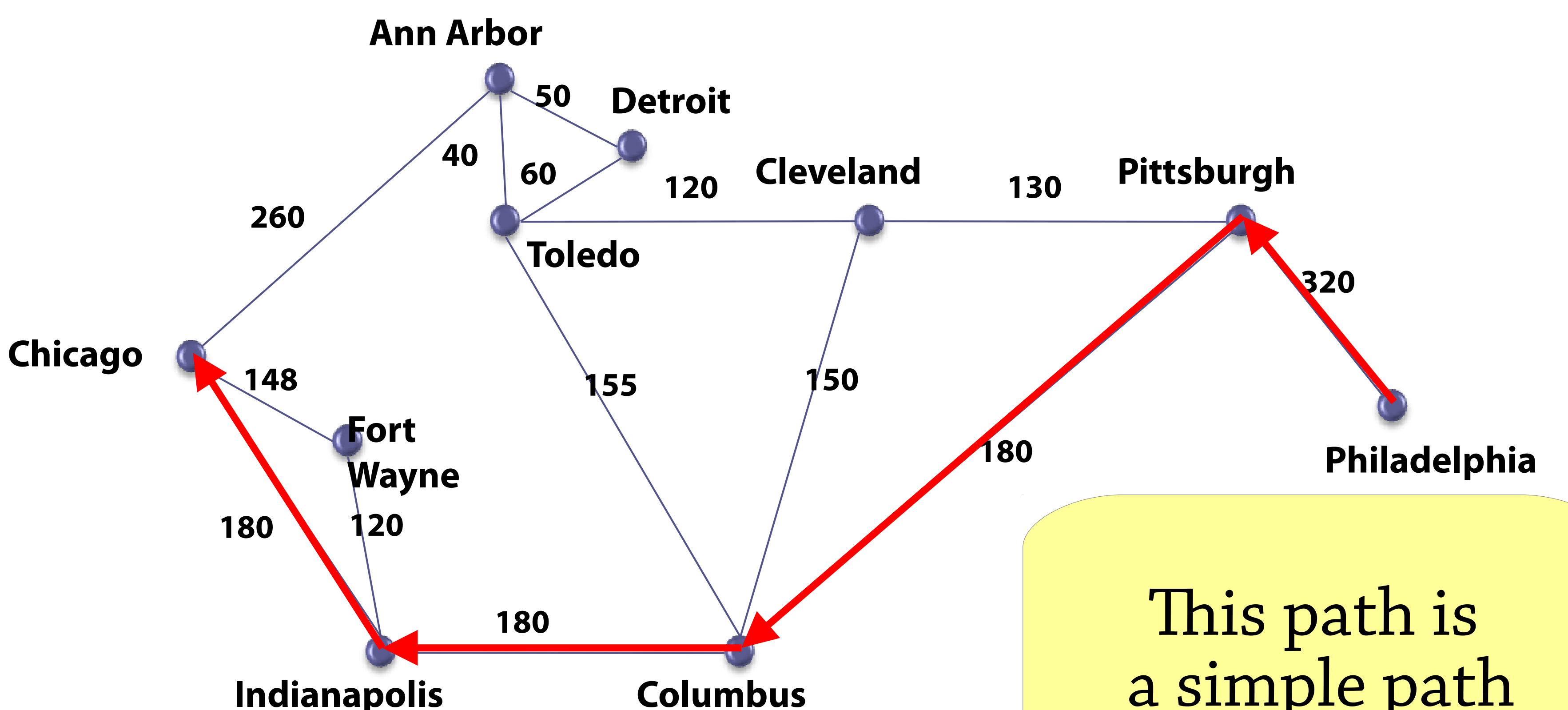
Paths and cycles

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



Paths and cycles

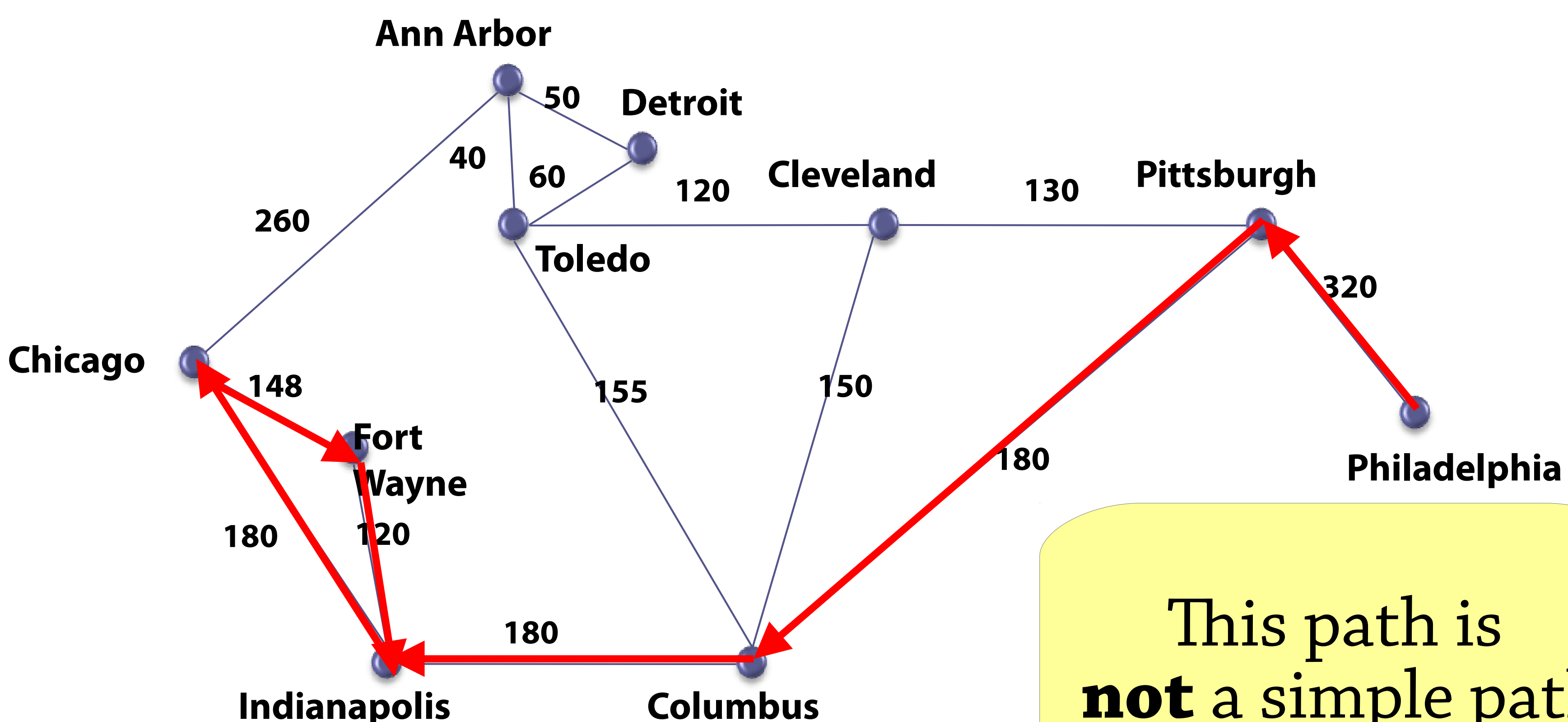
In a *simple path*, no node or edge appears twice, except that the first and last node can be the same



This path is
a simple path

Paths and cycles

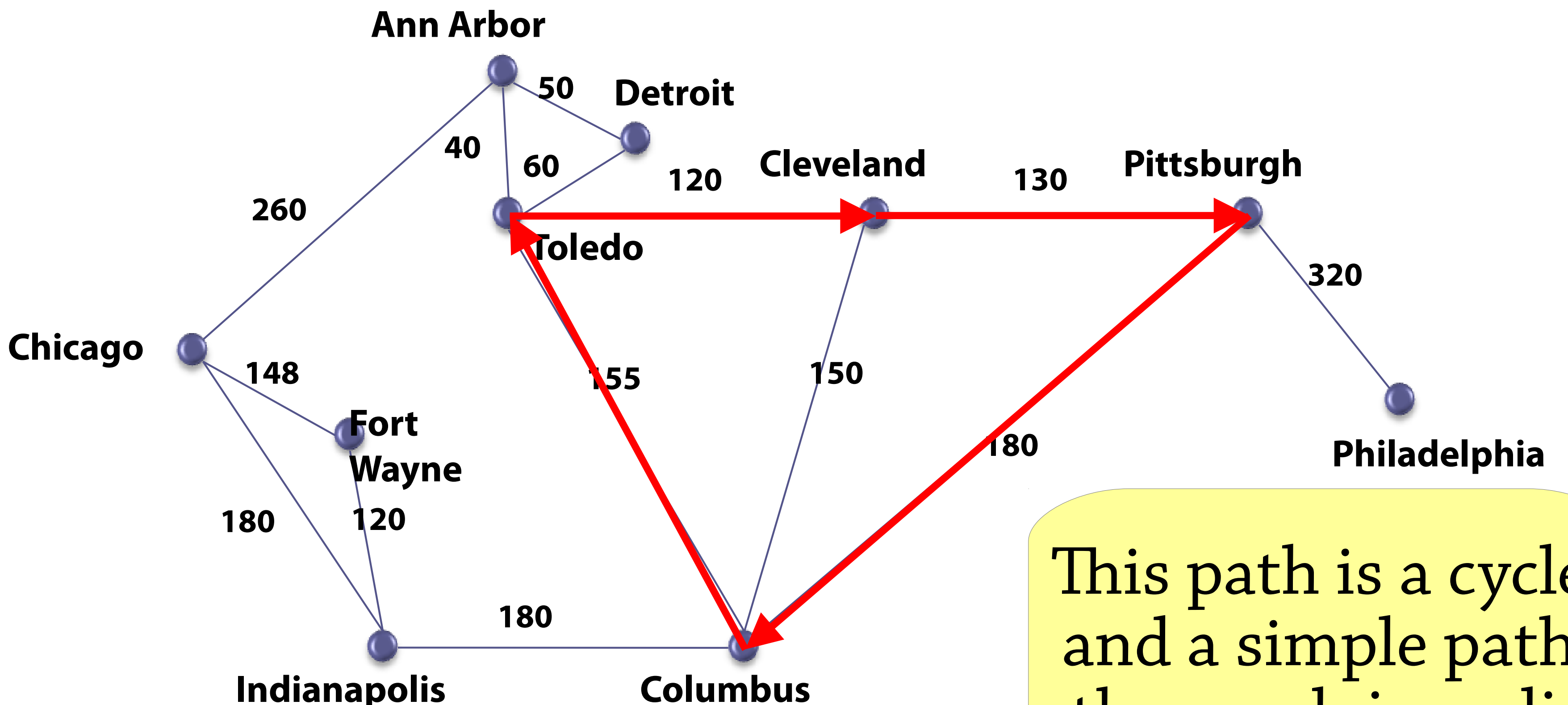
In a *simple path*, no node or edge appears twice, except that the first and last node can be the same



This path is **not** a simple path

Paths and cycles

A *cycle* is a simple path where the first and last nodes are the same – a graph that contains a cycle is called *cyclic*, otherwise it is called *acyclic*

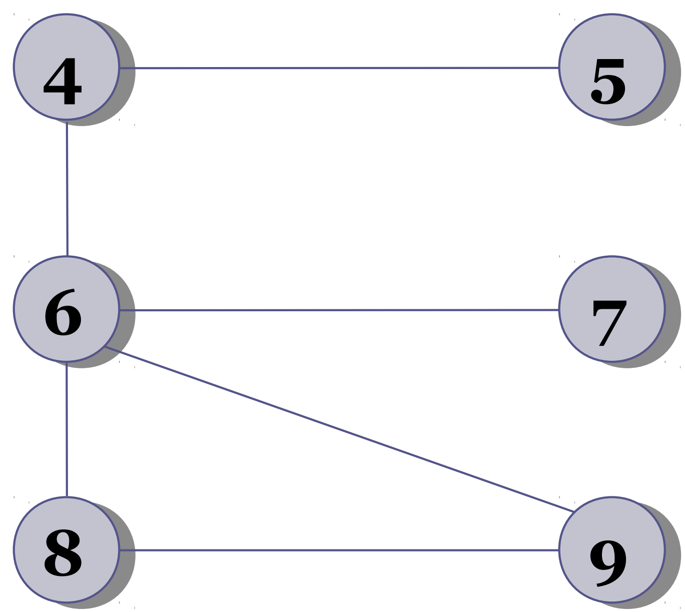


This path is a cycle, and a simple path; the graph is cyclic

Connectedness

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

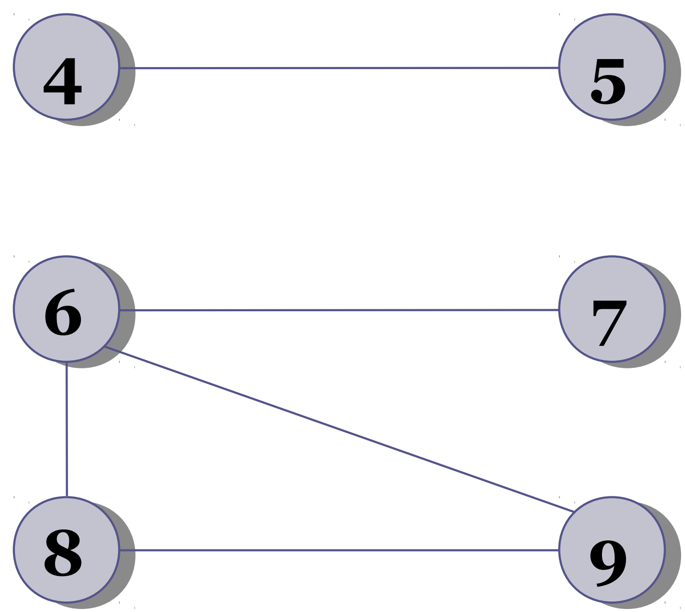
This graph is connected



Connectedness

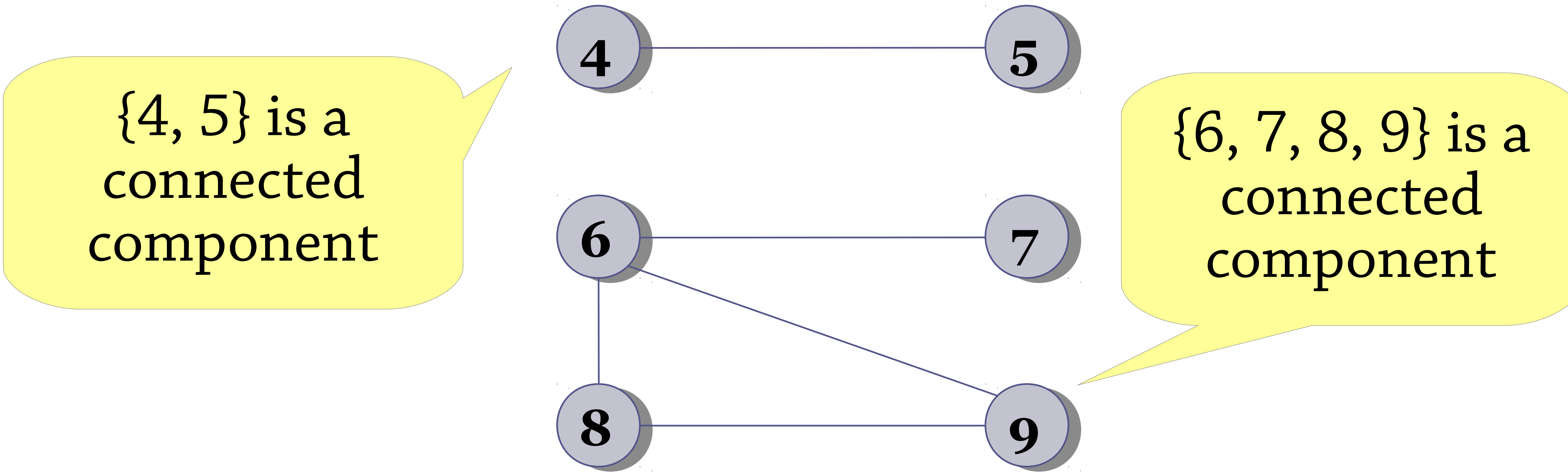
A graph is called *connected* if there is a path from every node to every other node

This graph is **not** connected



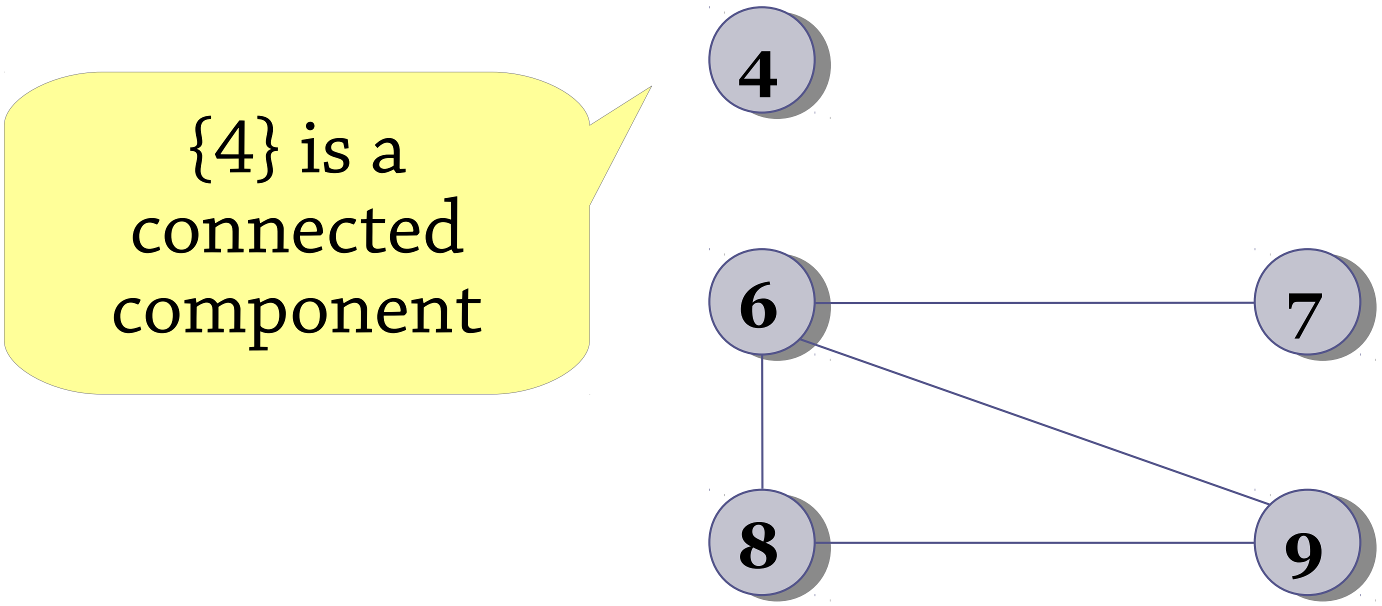
Connectedness

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



Connectedness

A single unconnected node is a connected component in itself



Implementing a graph

Alternative 1: *adjacency lists*

Keep a list of all nodes in the graph

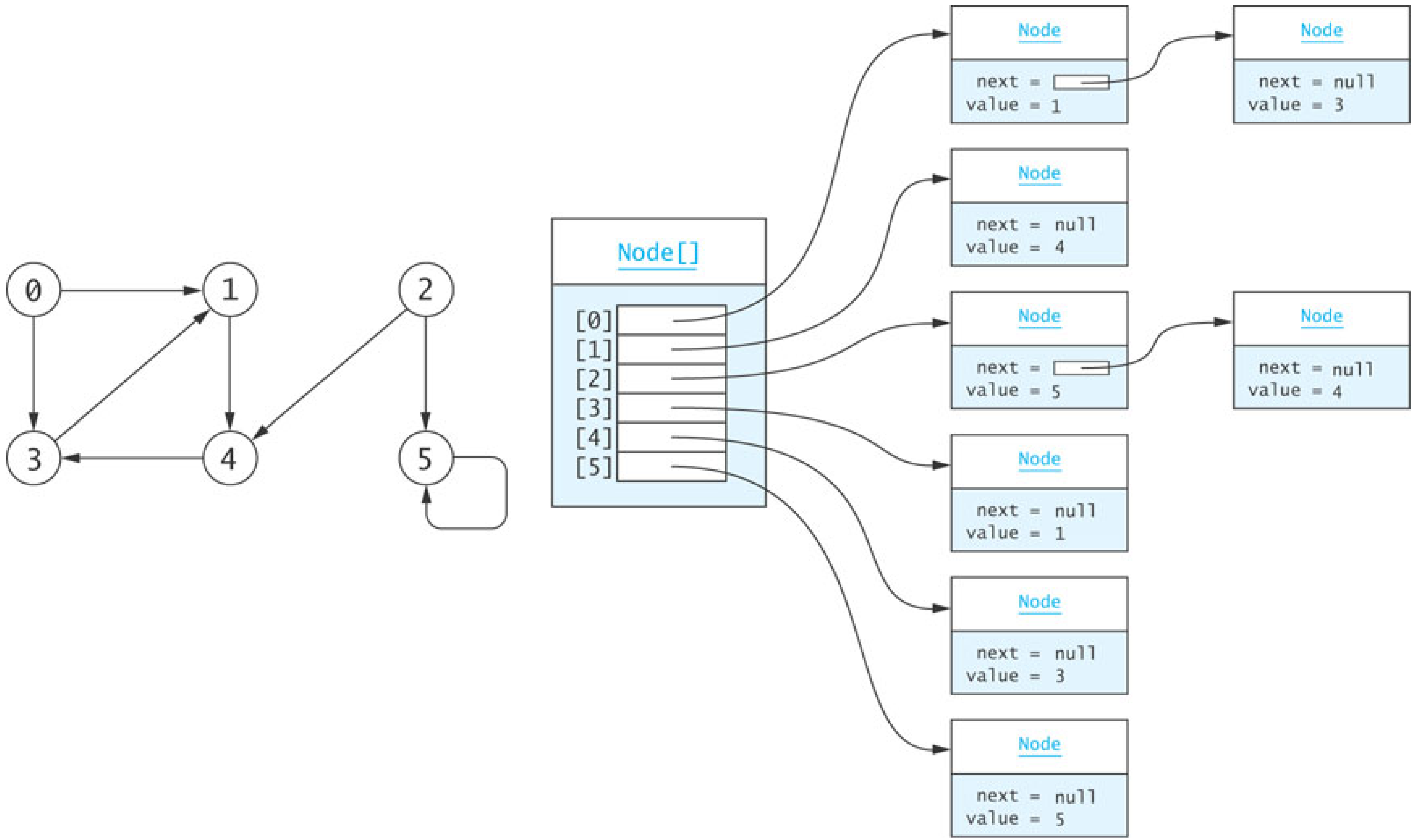
- With each node, associate a list of all the nodes adjacent to that nodes

Alternative 2: *adjacency matrix*

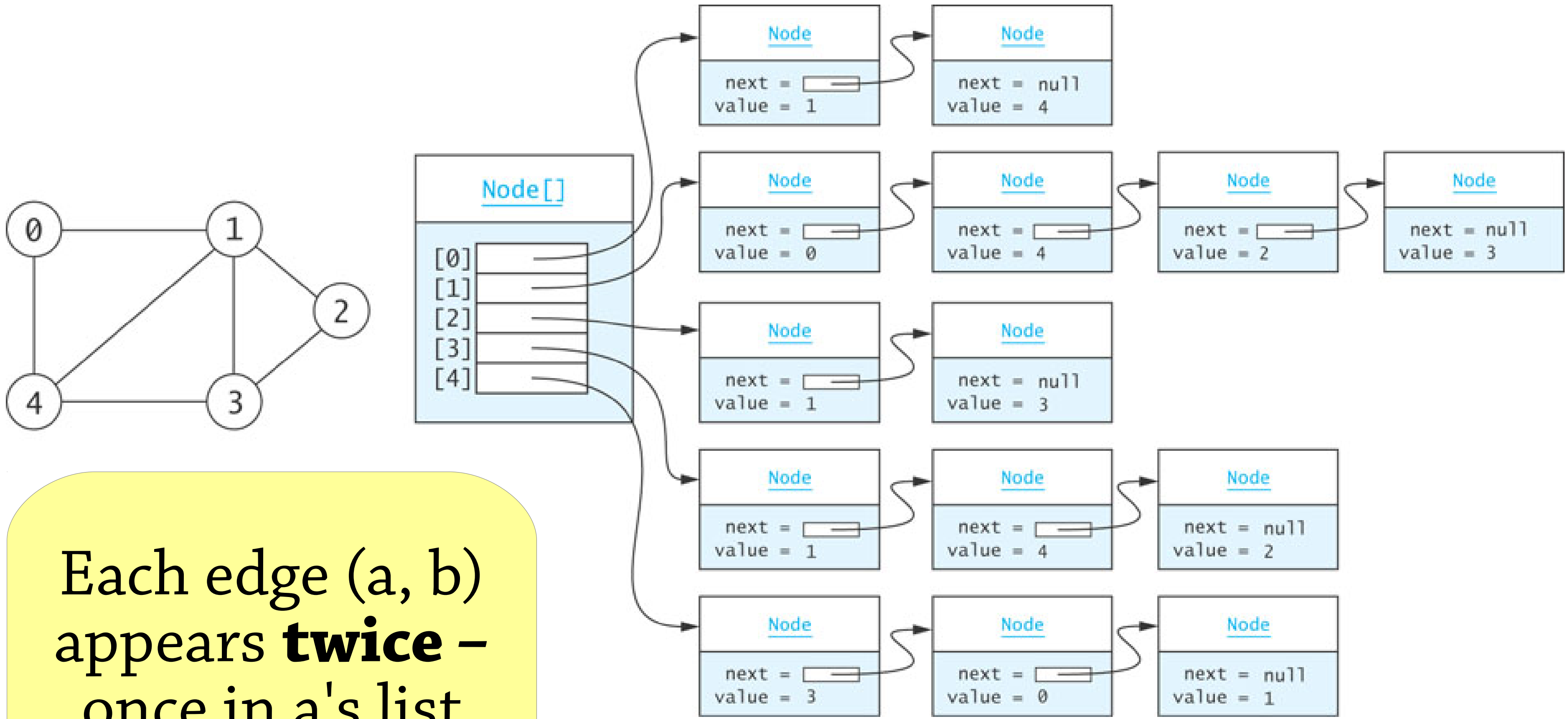
Keep a 2-dimensional array, with one entry for each pair of nodes

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

Adjacency list – directed graph



Adjacency list – undirected graph



Each edge (a, b) appears **twice –** once in a's list and once in b's list

Adjacency matrix

We use a 2-dimensional array

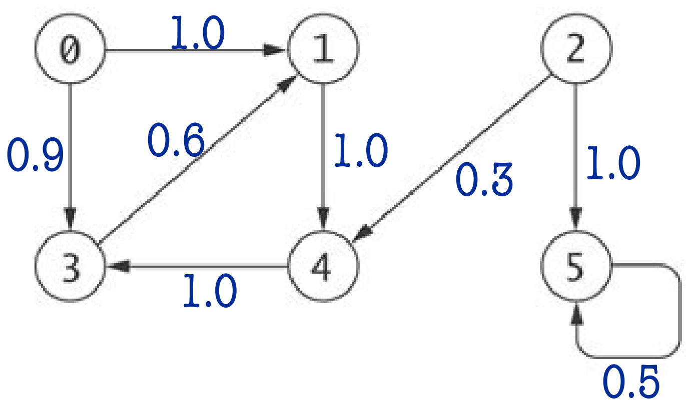
For an unweighted graph, we use an array of booleans

- a[i][j] = true if there is an edge between node i and node j
- For an undirected graph, a[i][j] = a[j][i]

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

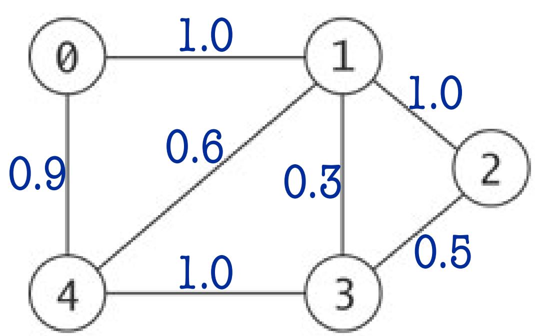
- We can e.g. use an infinite value if there is no edge between a pair of nodes

Adjacency matrix, weighted graph



Column

Row	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1.0		0.9		
[1]					1.0	
[2]					0.3	1.0
[3]		0.6				
[4]				1.0		
[5]						0.5



Column

Row	[0]	[1]	[2]	[3]	[4]
[0]		1.0			0.9
[1]	1.0		1.0	0.3	0.6
[2]		1.0		0.5	
[3]		0.3	0.5		1.0
[4]	0.9	0.6		1.0	

Which representation is best?

It depends on the graph's *density*

- The quantity $|E| / |V|^2$, where $|V|$ is the number of nodes and $|E|$ the number of edges
- In a *dense* graph, $|E|$ is close to $|V|^2$
- In a *sparse* graph, $|E|$ is much lower than $|V|^2$

Most graphs are sparse!

- If each node has a bounded number of edges, then $|E|$ will be proportional to $|V|$

Which representation is best?

Many graph algorithms have the form:

```
for each node u in the graph
  for each node v adjacent to u
    do something with edge (u, v)
```

With an adjacency list, we can just iterate through all nodes and edges in the graph

- This gives a complexity of $O(|V| + |E|)$

With an adjacency matrix, we must try each pair (u, v) of nodes to check if there is an edge

- This gives a complexity of $O(|V|^2)$

Winner: adjacency lists for sparse graphs, unclear for dense graphs

Which representation is best?

So:

- if the graph is sparse adjacency lists are better (common)
- if the graph is dense an adjacency matrix are better (rare)

What about memory consumption?

- An adjacency matrix needs space for $|V|^2$ values, so takes $O(|V|^2)$ memory – but with a low constant factor because each value is just a double
- An adjacency list needs $O(|V| + |E|)$ space – but with a higher constant factor because of the node objects
- Again depends on how sparse the graph is

Graph traversals

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

- Just like with trees, there are several orders you might want

The two commonest methods are:

- breadth-first search
- depth-first search

Breadth-first search

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

- First the start node
- Then all nodes that are adjacent to the start node
- Then all nodes that are adjacent to those
- and so on

We end up visiting all nodes that are k edges away from the start node, before visiting any nodes that are $k+1$ edges away

Implementing breadth-first search

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

- Initially, the queue contains the start node

We repeat the following process:

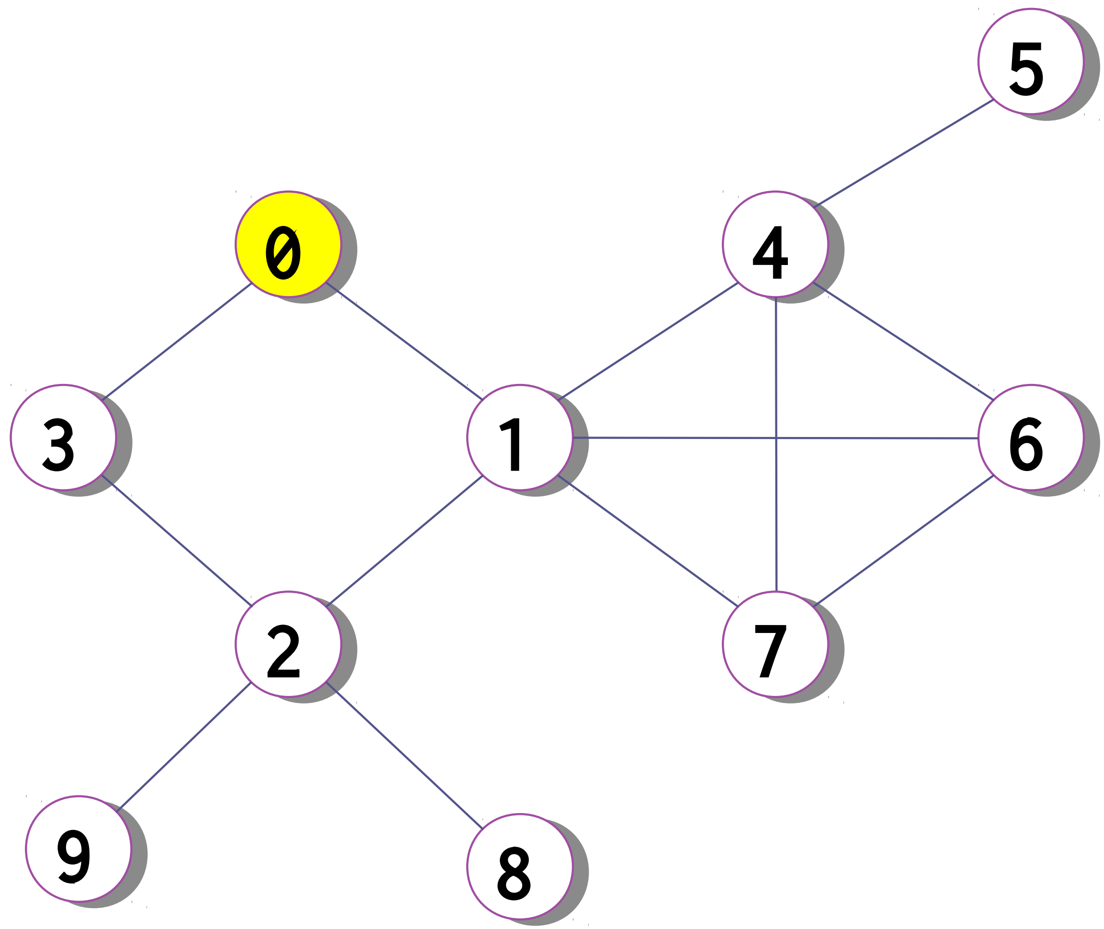
- Remove a node from the queue
- Visit it
- Find all nodes adjacent to the visited node and add them to the queue, *unless* they have been visited or added to the queue already

Example of a breadth-first search

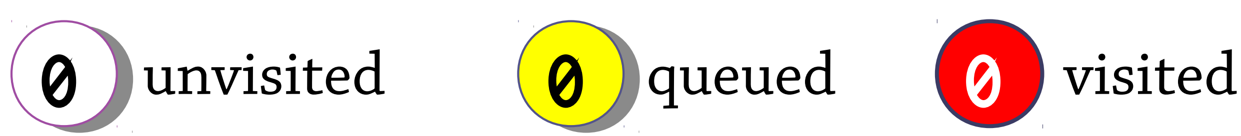
Queue:

0

Visit order:



Initially,
queue contains
start node

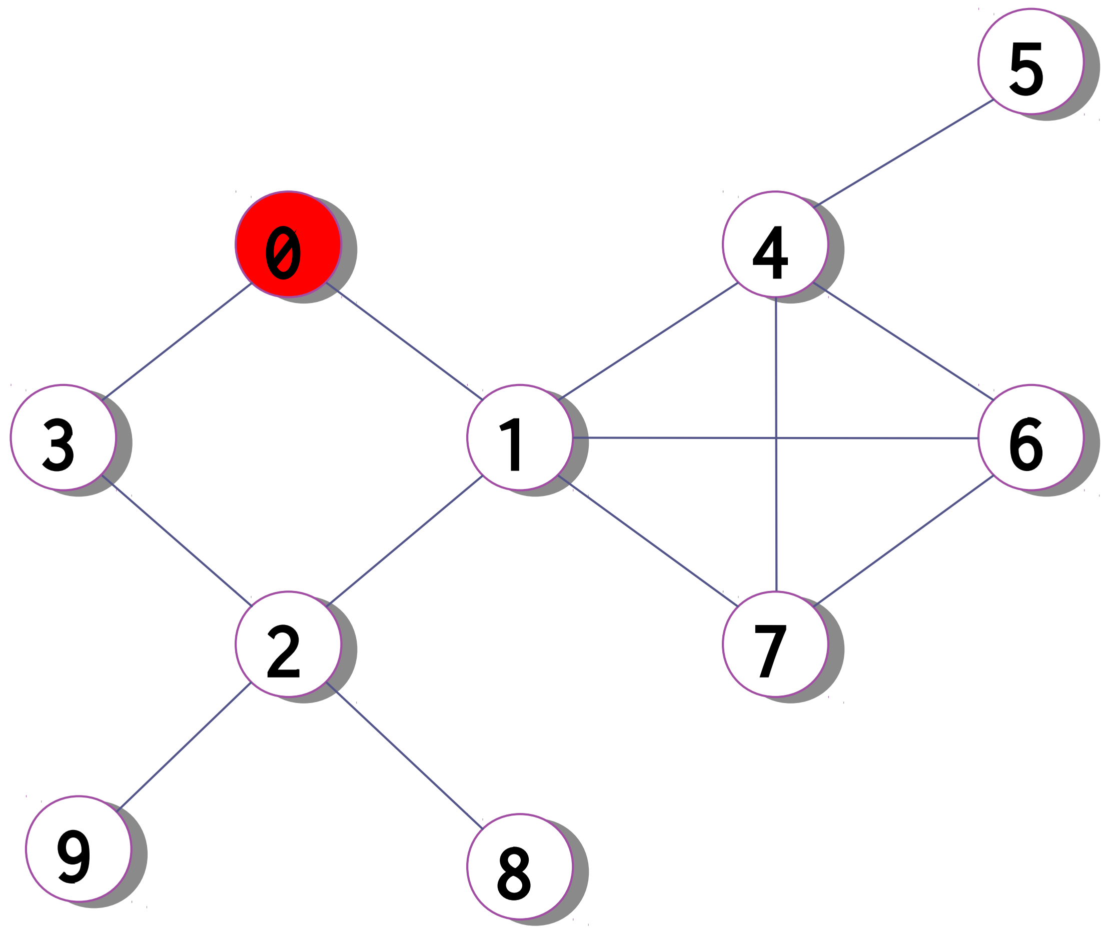


Example of a breadth-first search

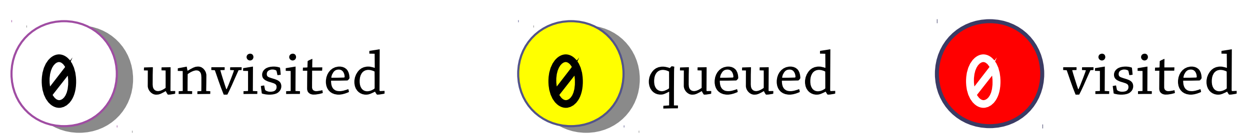
Queue:

Visit order:

0



Step 1:
remove node
from queue
and visit it



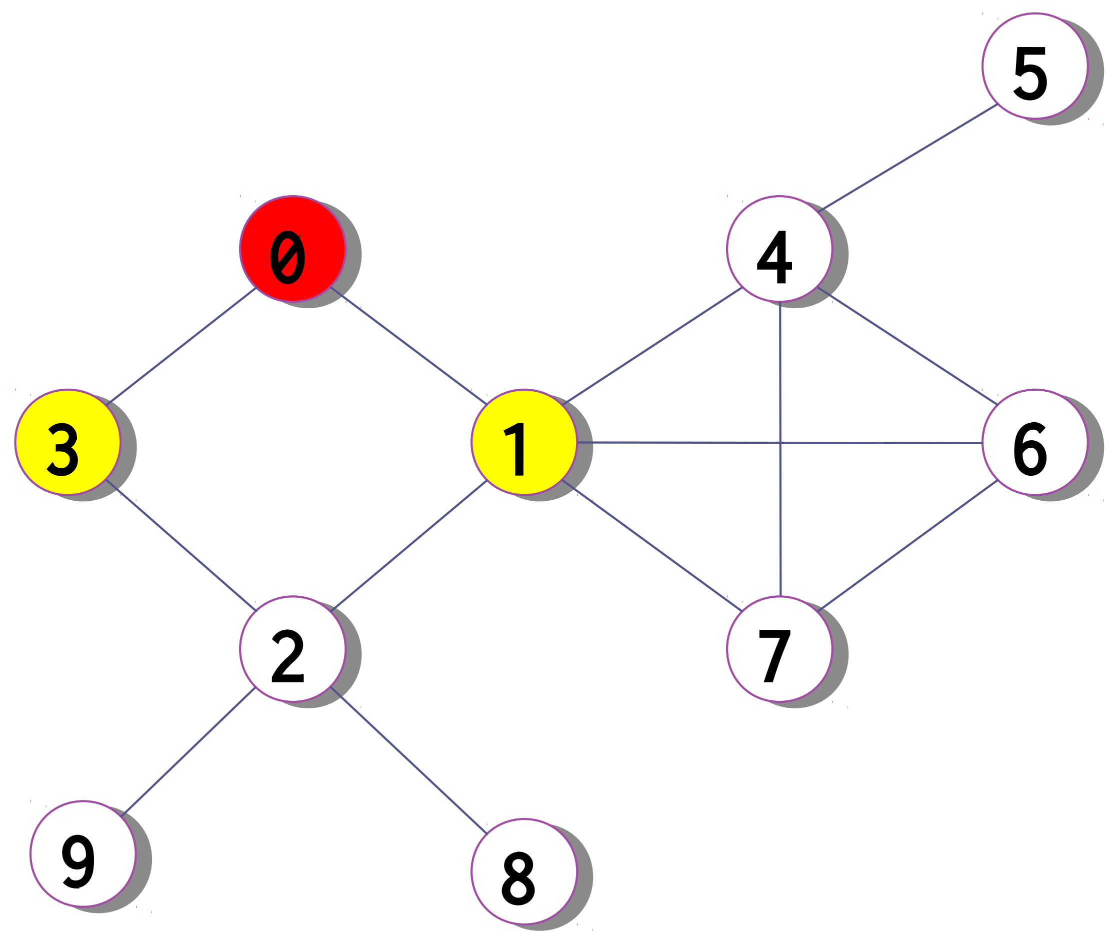
Example of a breadth-first search

Queue:

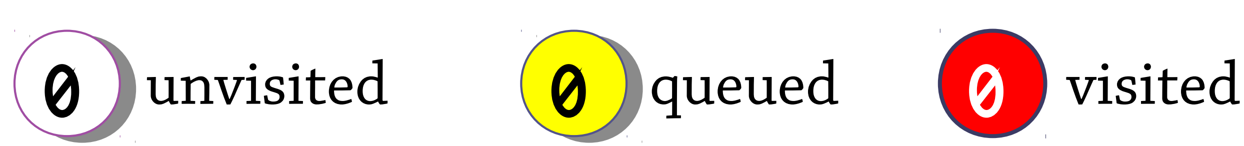
3 1

Visit order:

0



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



Example of a breadth-first search

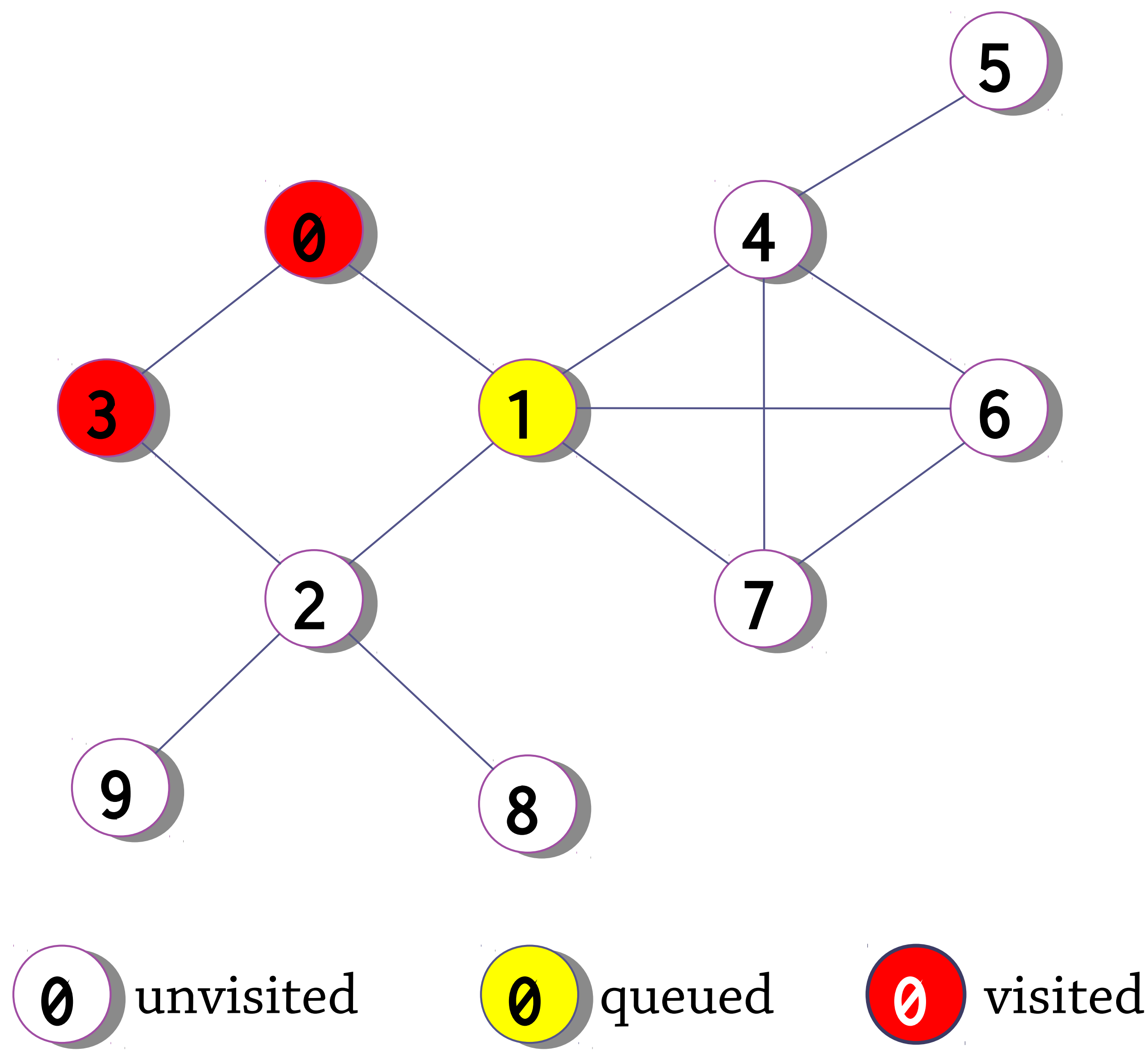
Queue:

1

Visit order:

0 3

Step 1:
remove node
from queue
and visit it



Example of a bre arch

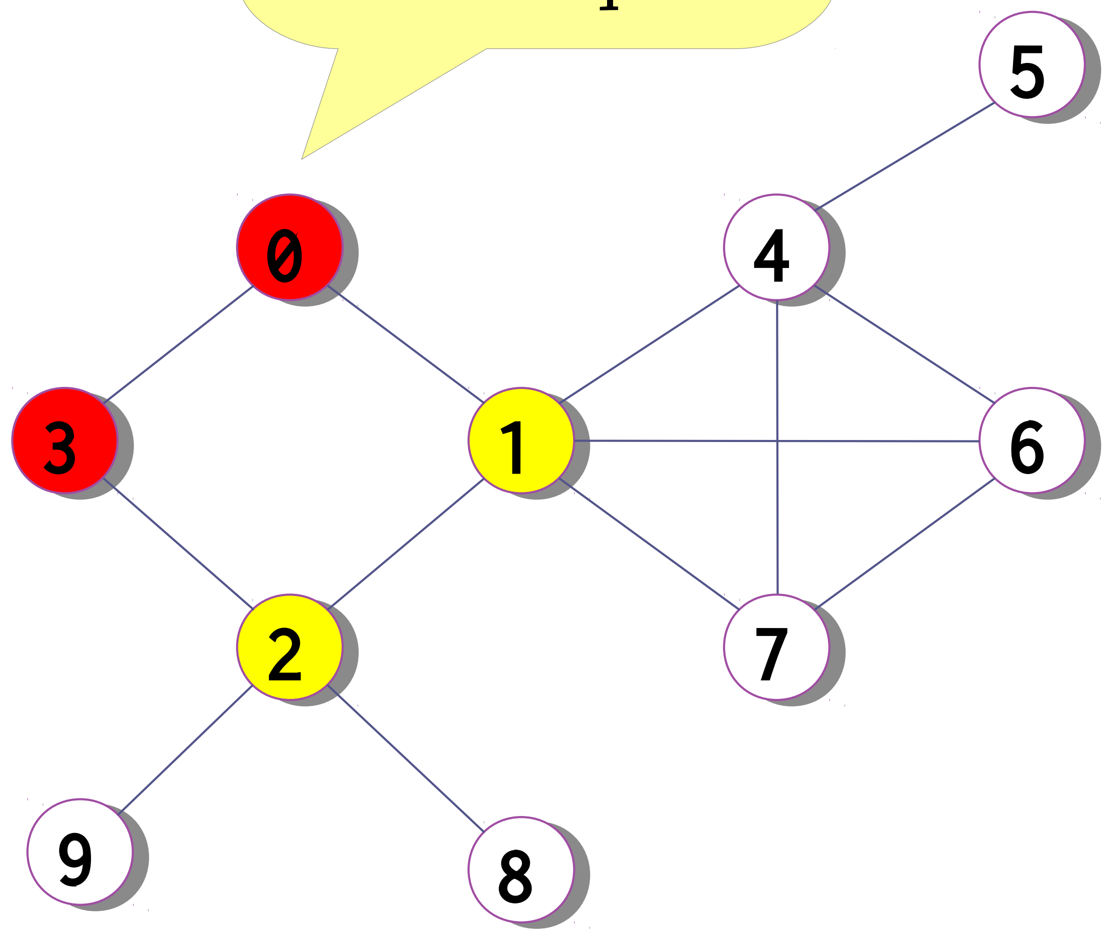
0 is already visited, so we don't add it to the queue

Queue:

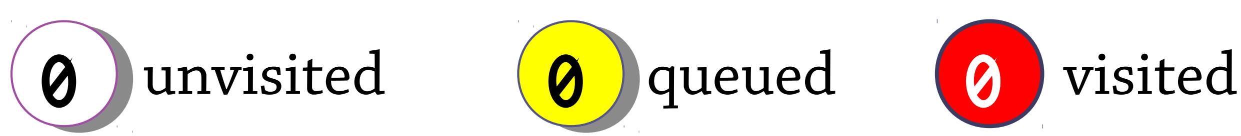
1 2

Visit order:

0 3



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



Example of a breadth-first search

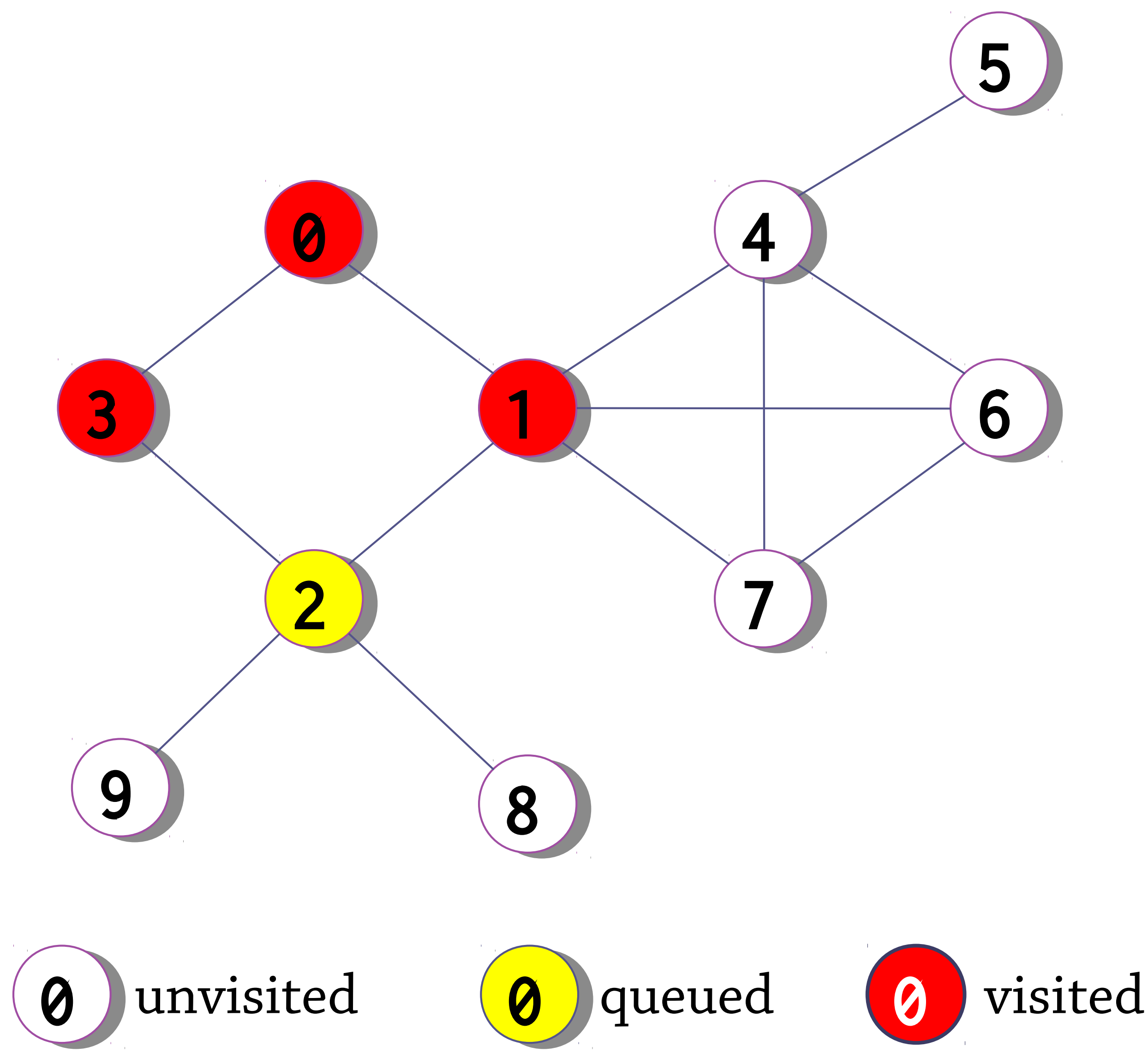
Queue:

2

Visit order:

0 3 1

Step 1:
remove node
from queue
and visit it



Example of a bre   arch

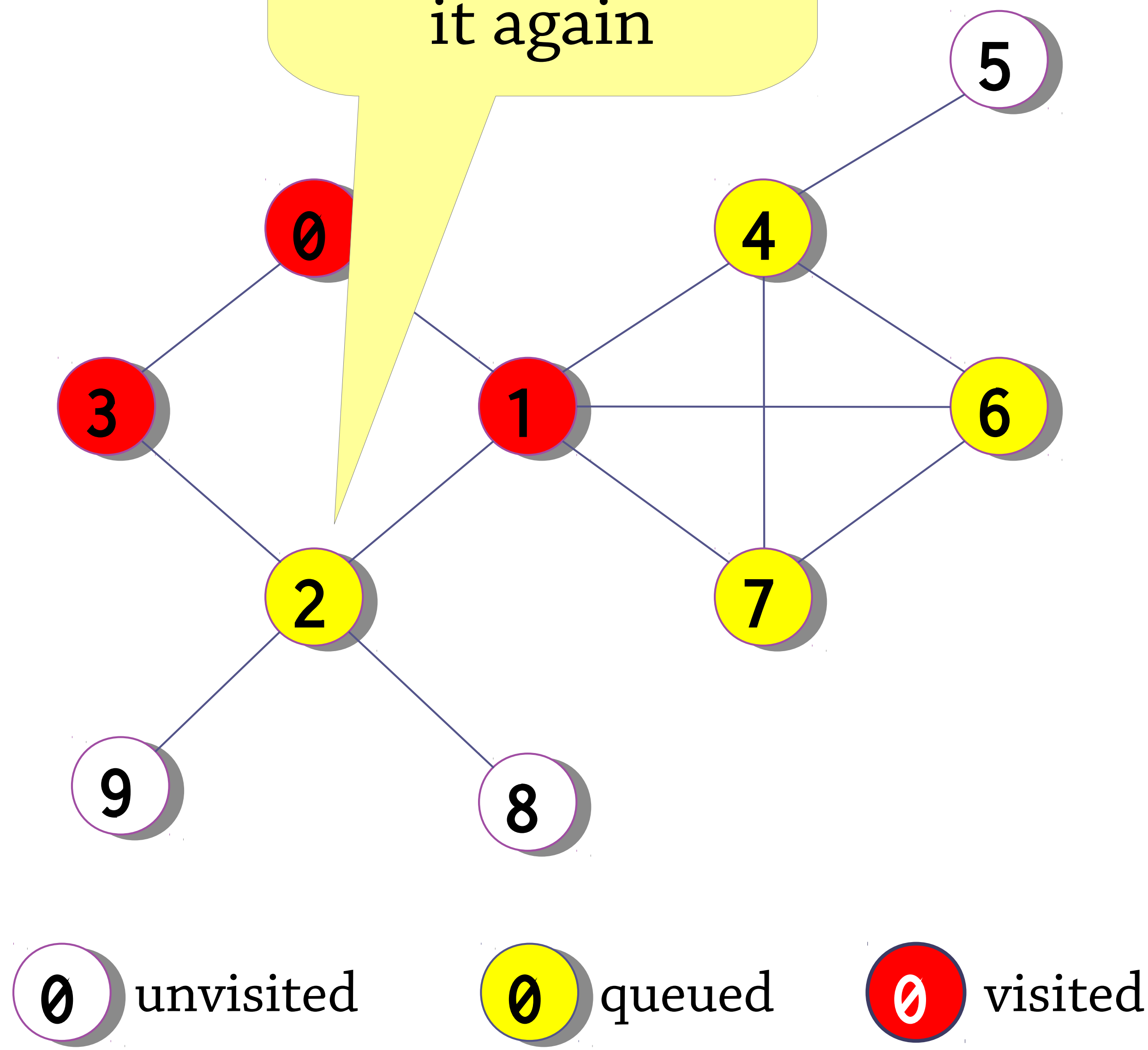
2 is already in the queue, so we don't add it again

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

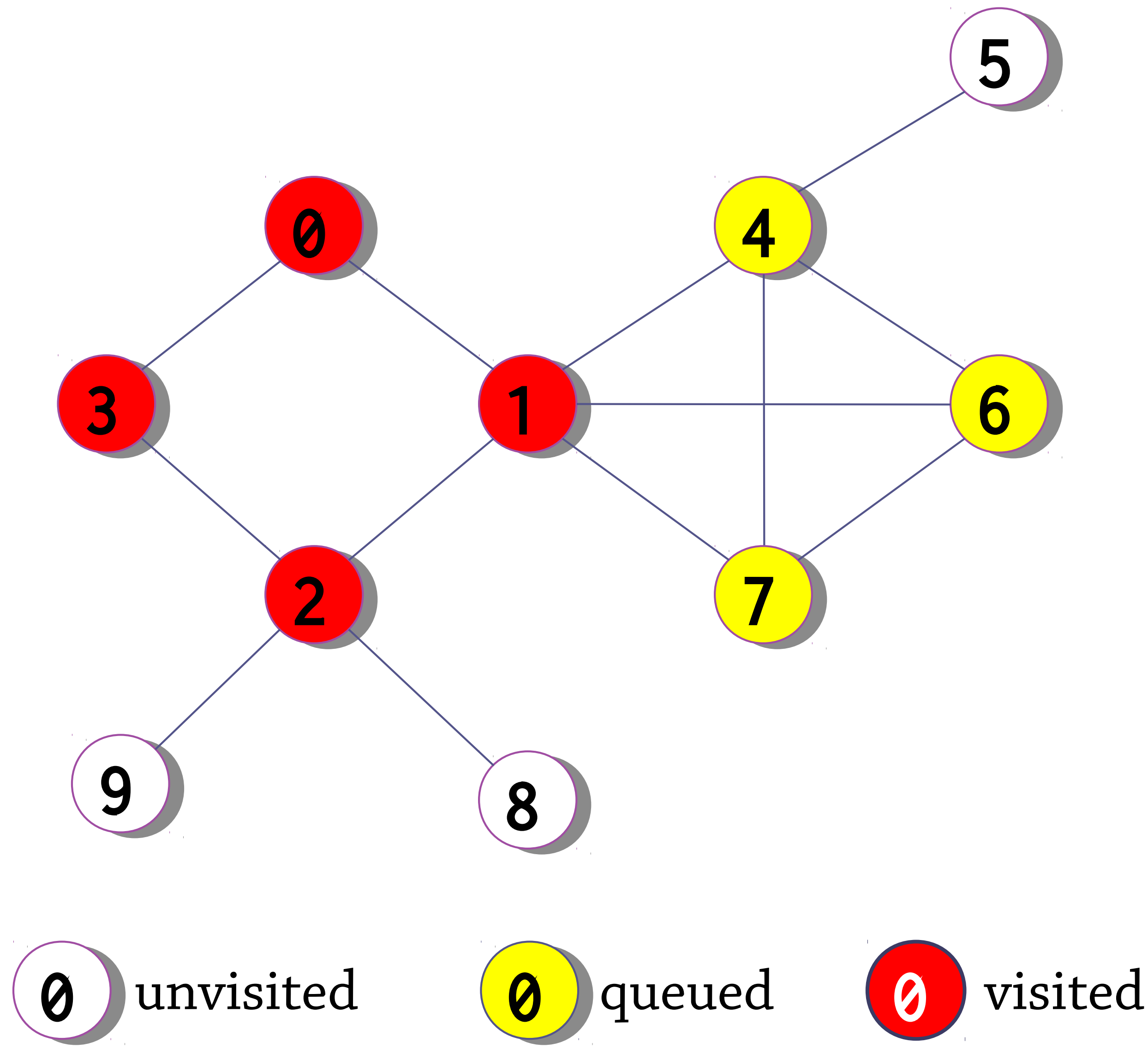
Queue:

4 6 7

Visit order:

0 3 1 2

Step 1:
remove node
from queue
and visit it



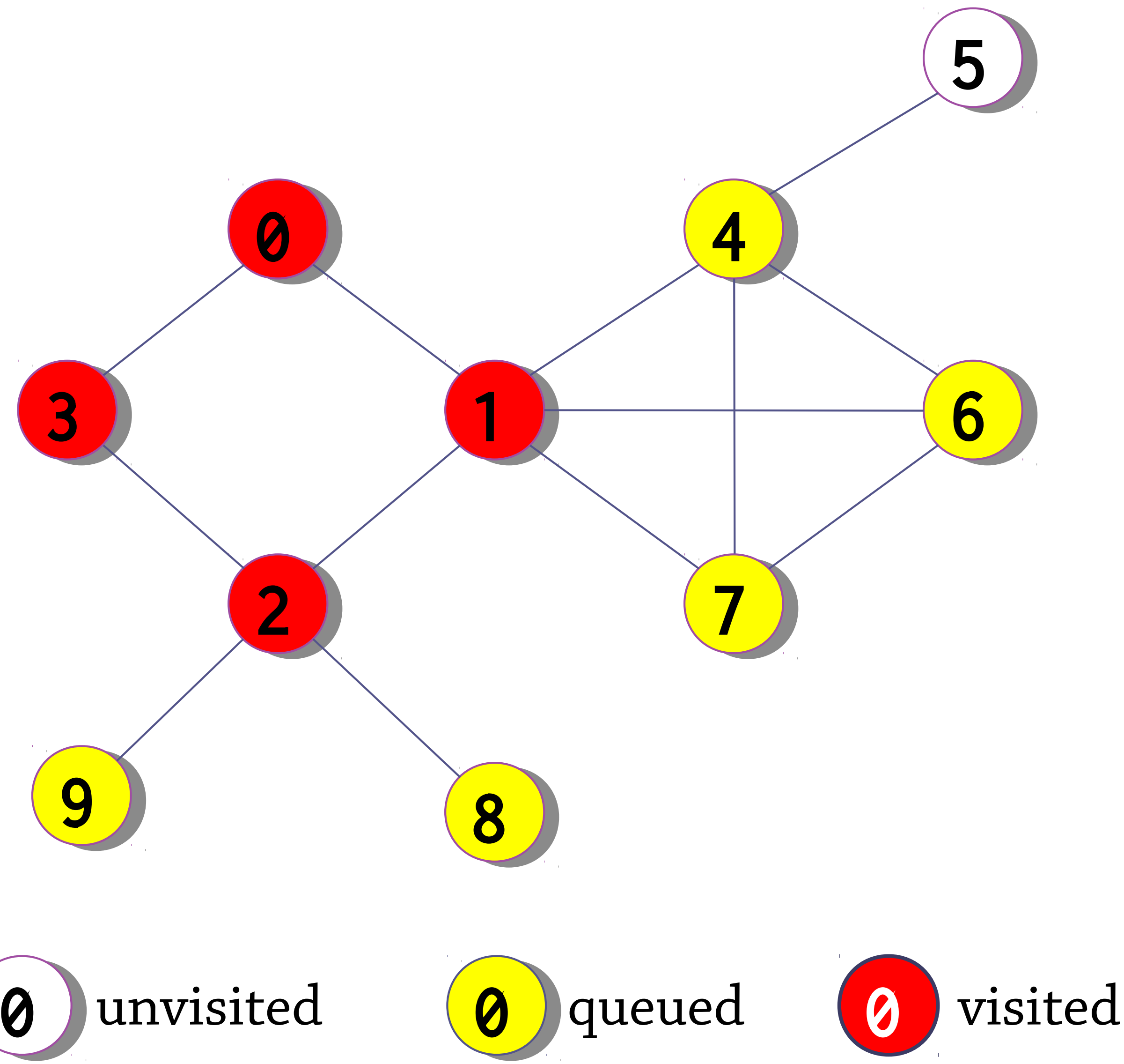
Example of a breadth-first search

Queue:

4 6 7 9 8

Visit order:

0 3 1 2



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

Example of a breadth-first search

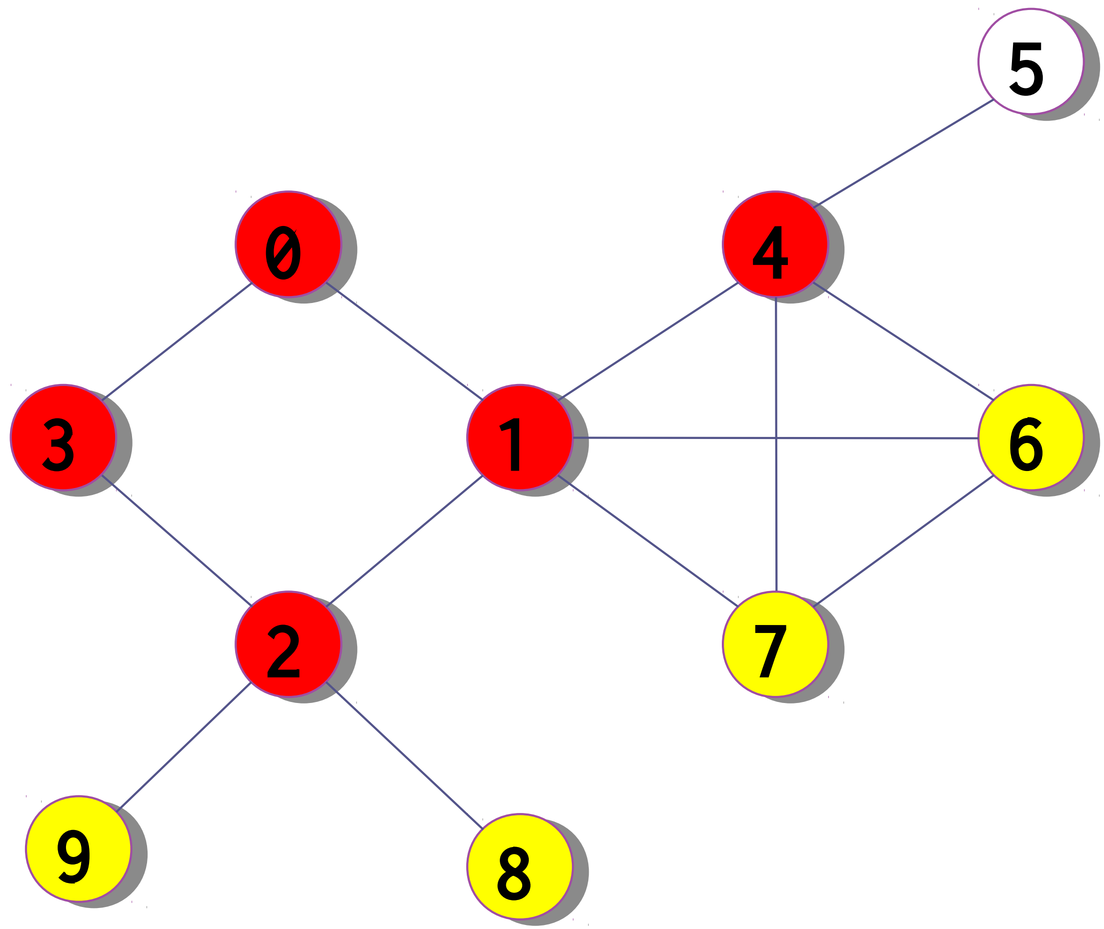
Queue:

6 7 9 8

Visit order:

0 3 1 2 4

Step 1:
remove node
from queue
and visit it



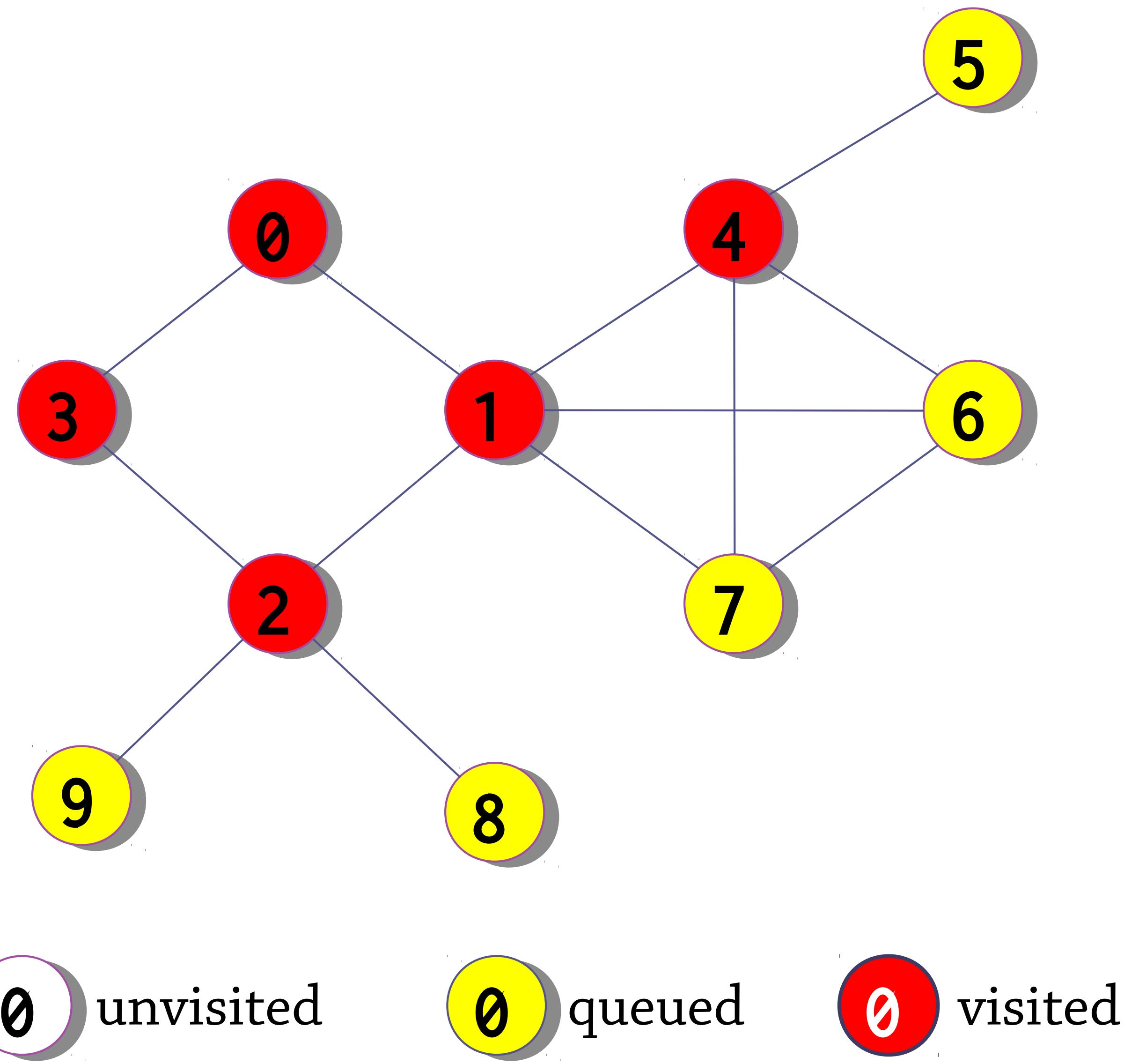
Example of a breadth-first search

Queue:

6 7 9 8 5

Visit order:

0 3 1 2 4



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

Example of a breadth-first search

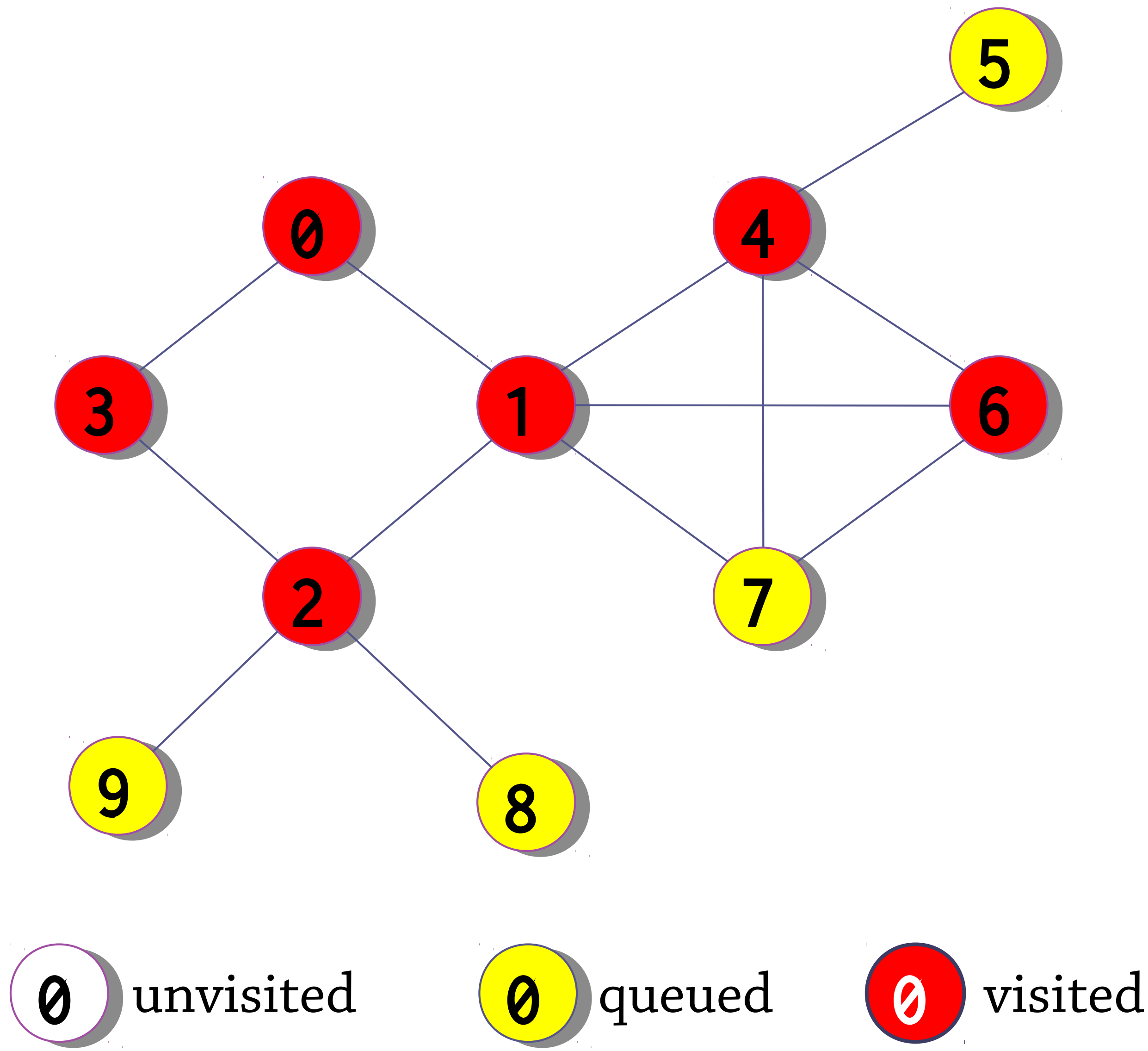
Queue:

7 9 8 5

Visit order:

0 3 1 2 4 6

Step 1: remove node from queue and visit it



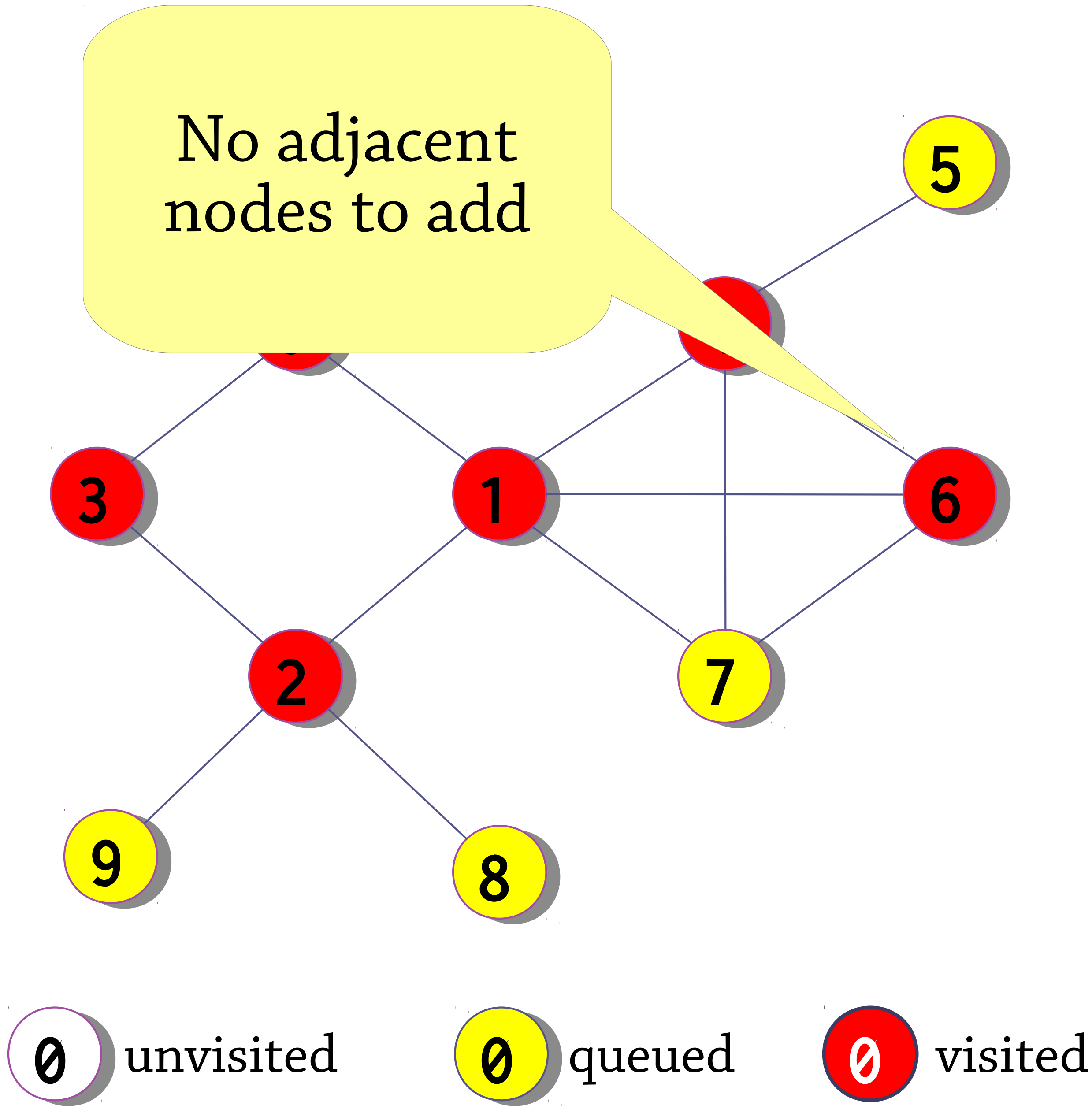
Example of a breadth-first search

Queue:

7 9 8 5

Visit order:

0 3 1 2 4 6



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

Example of a breadth-first search

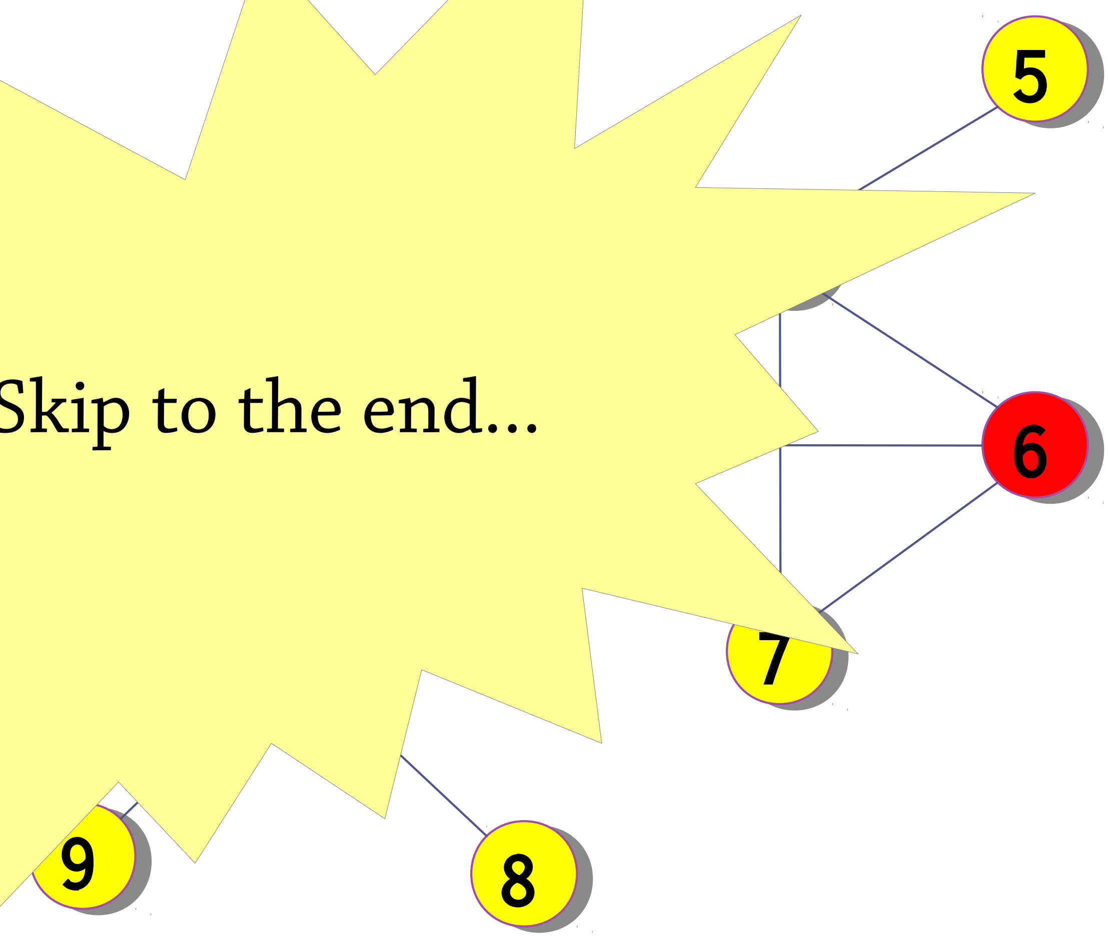
Queue:

7 9 8 5

Visit order:

0 3 1 2 4
6

Skip to the end...



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

0 unvisited

0 queued

0 visited

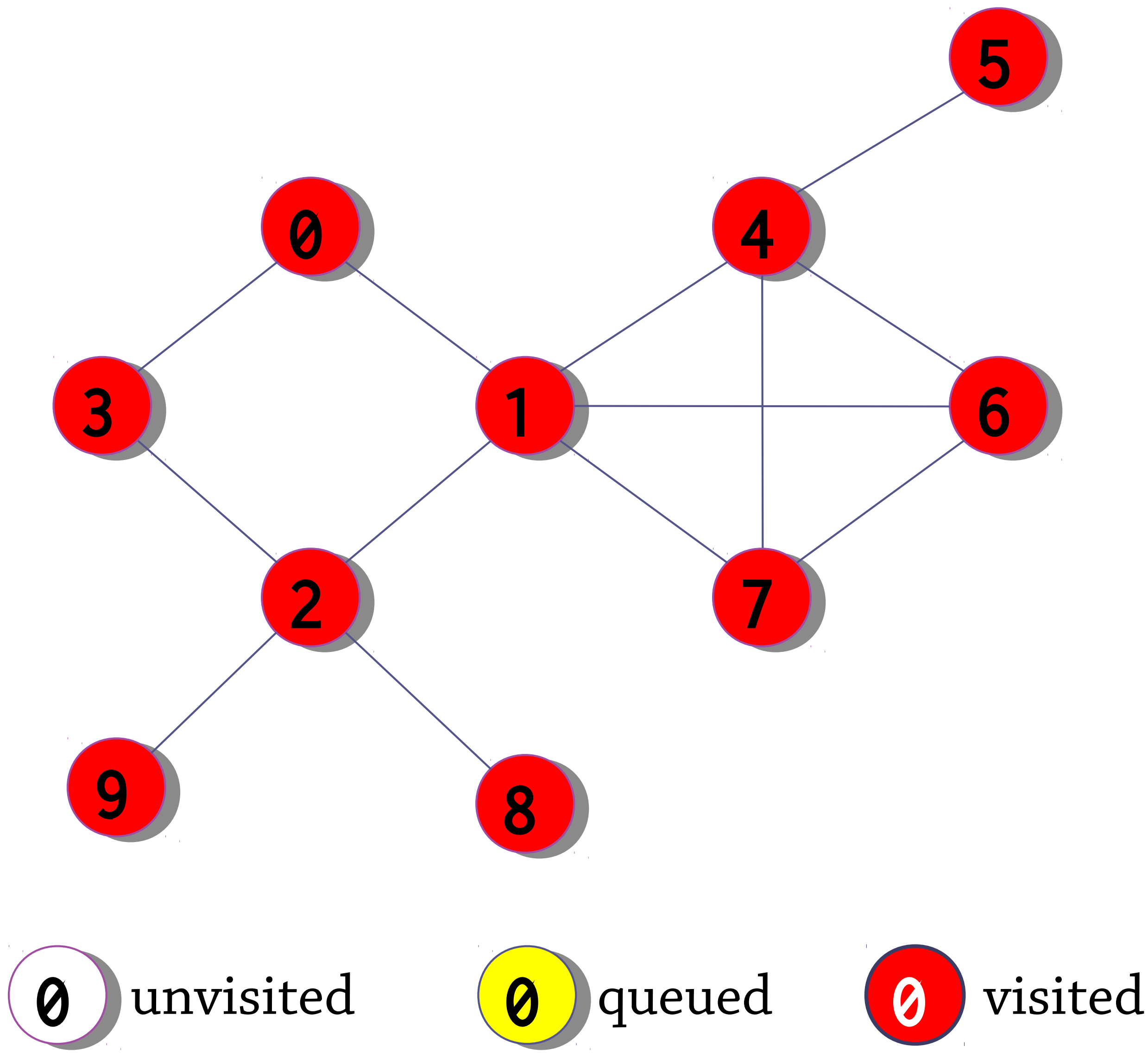
Example of a breadth-first search

Queue:

Visit order:

0 3 1 2 4
6 7 9 8 5

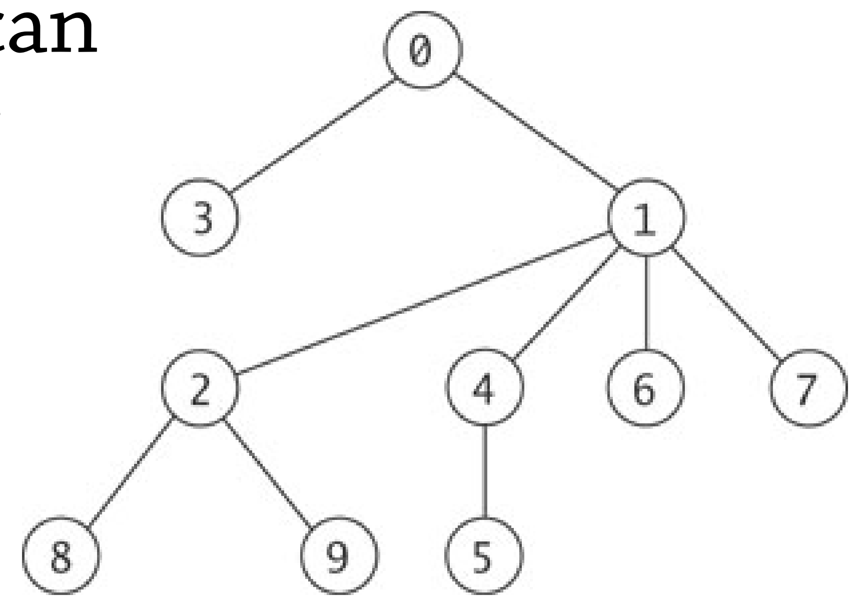
We reach step 1, but the queue is empty, and **we're finished!**



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)



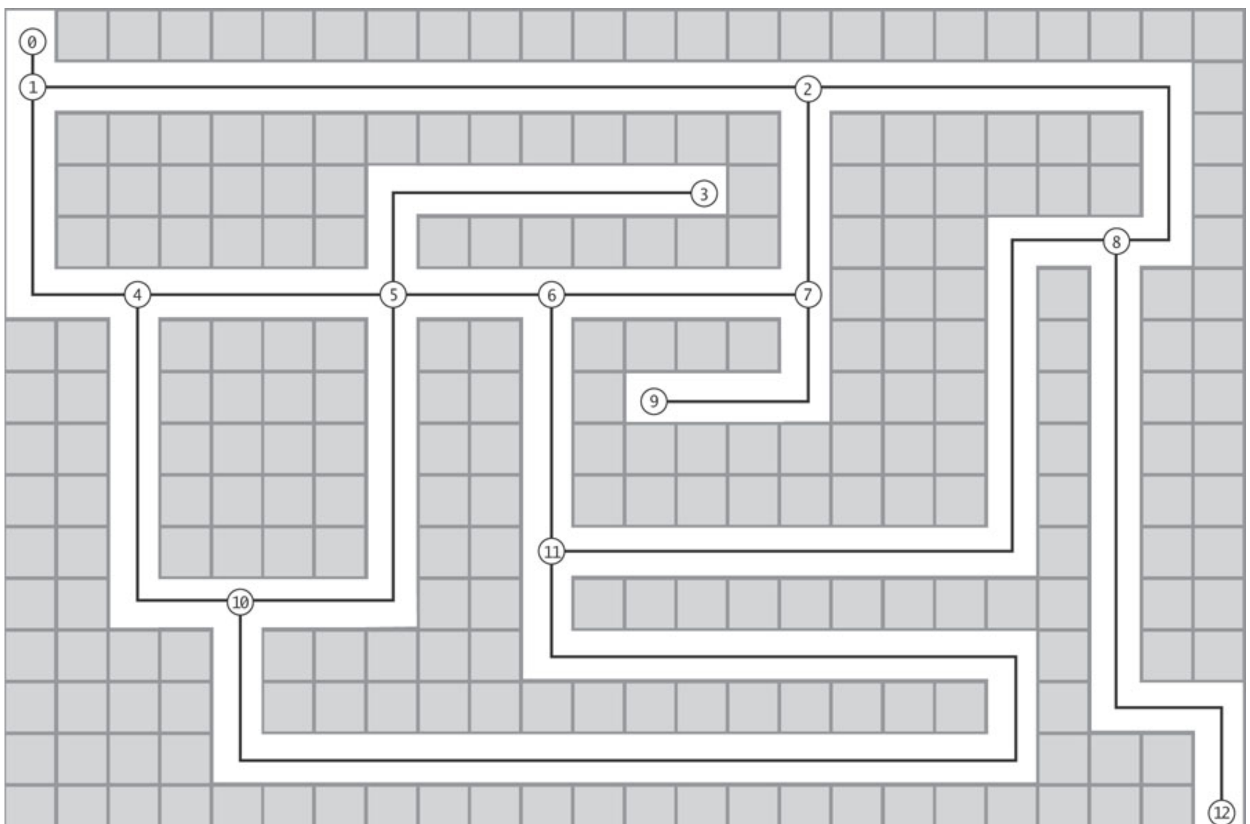
By doing this we can build a tree with the start node at the top (the *breadth-first search tree*)

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.

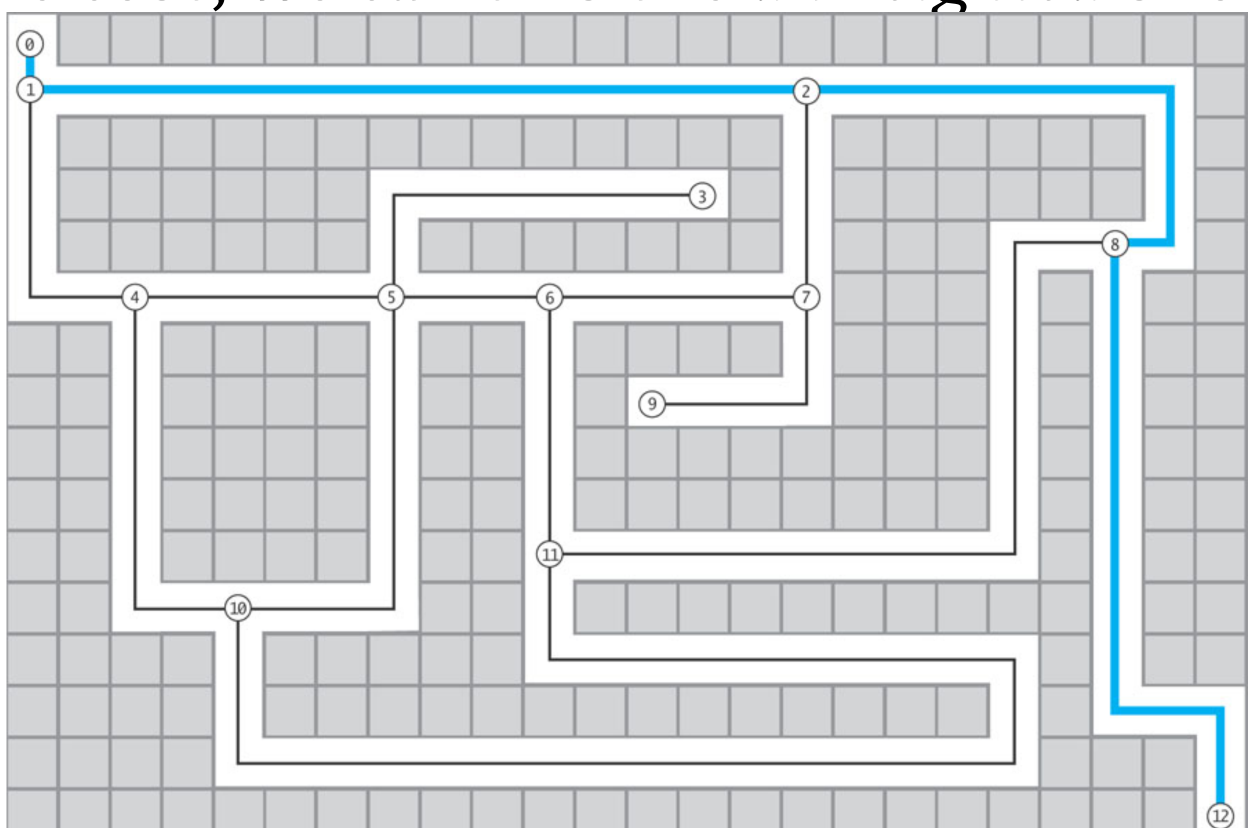
How can we find a path from the entrance to the exit?



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!

Depth-first search, alternative order

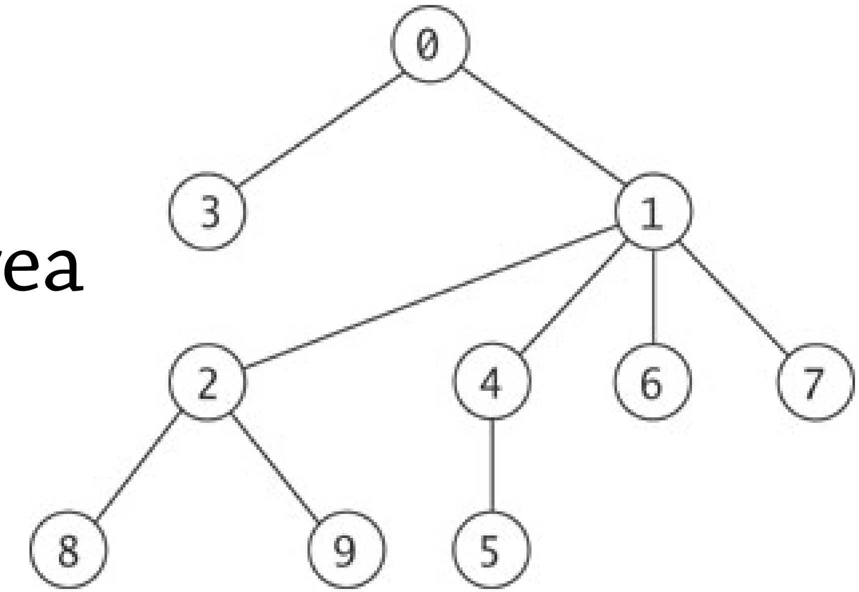
A variation of DFS, where we visit each node *after* visiting the adjacent nodes.

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)

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

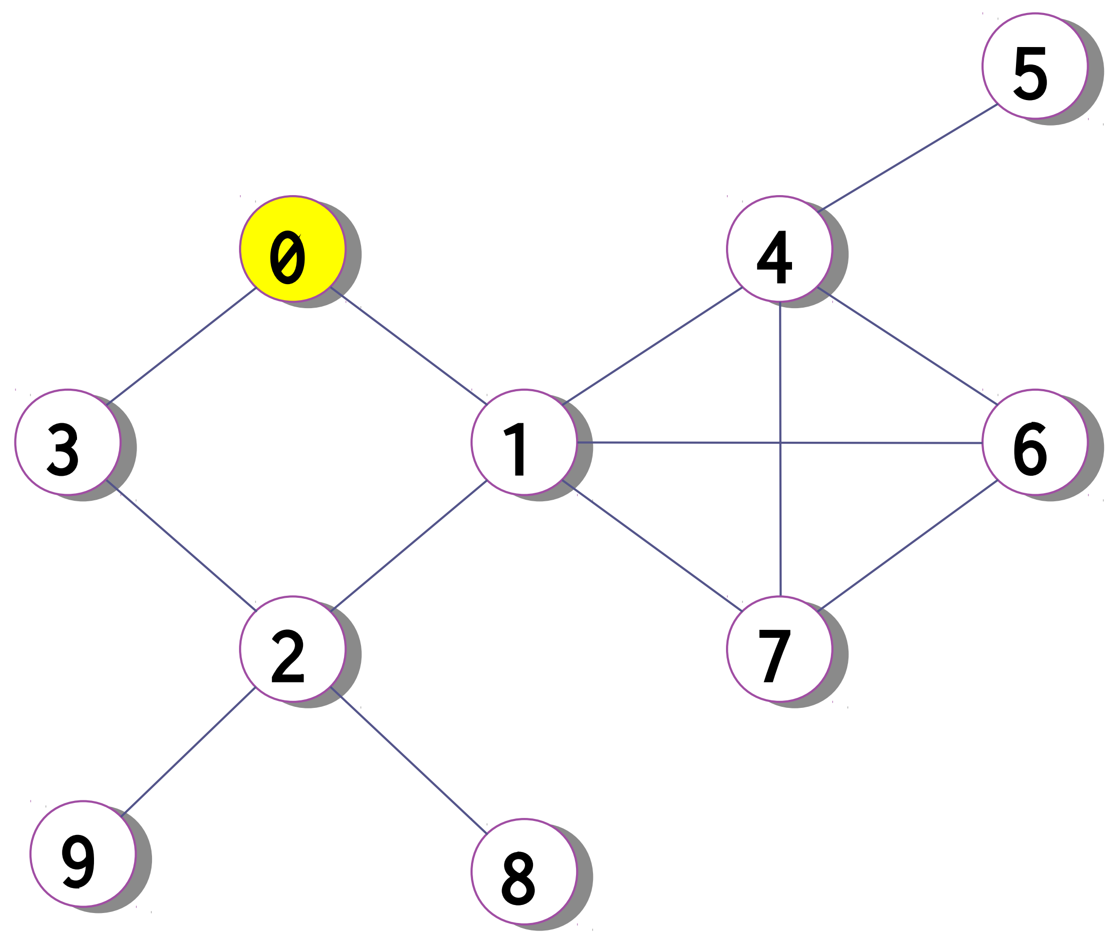
We can implement DFS just by taking the BFS algorithm and using a stack instead of a queue!

Example of a depth-first search

Stack:

0

Visit order:



Initially,
stack contains
start node

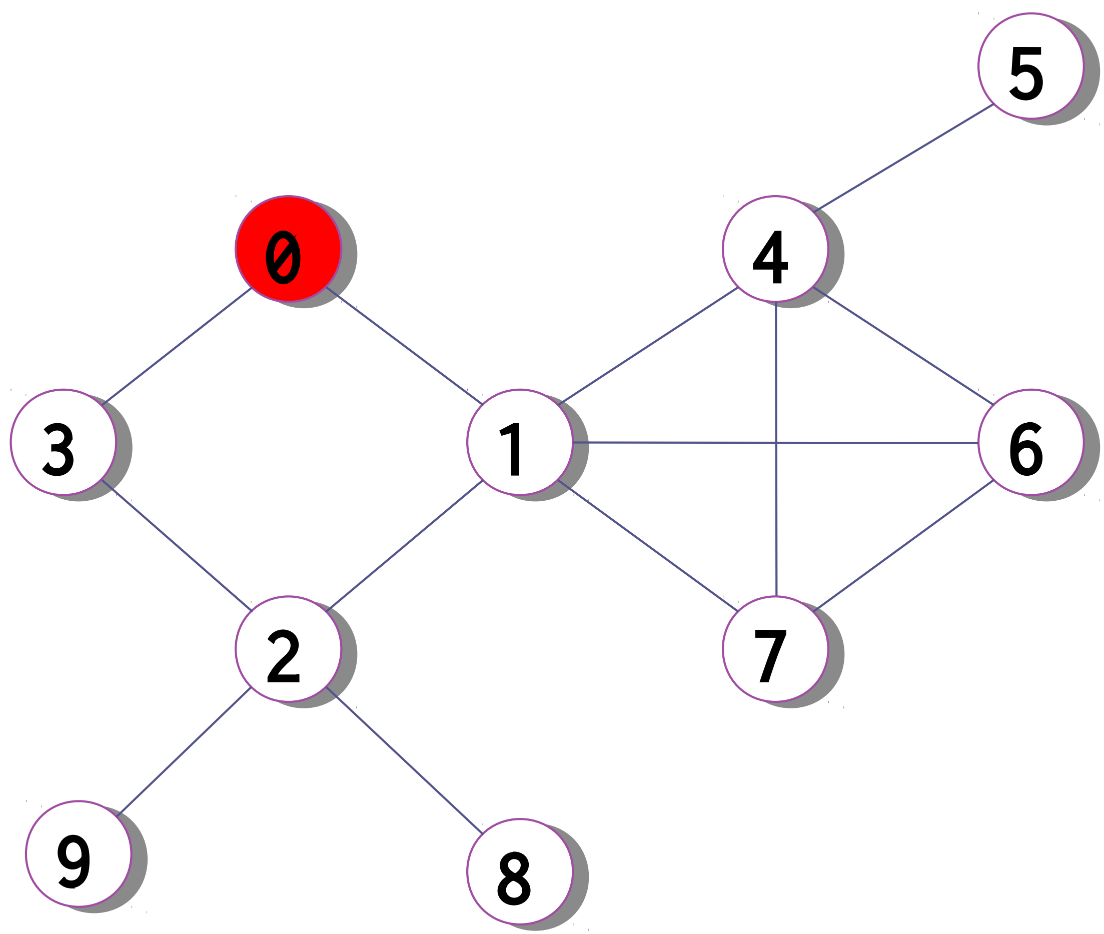
Example of a depth-first search

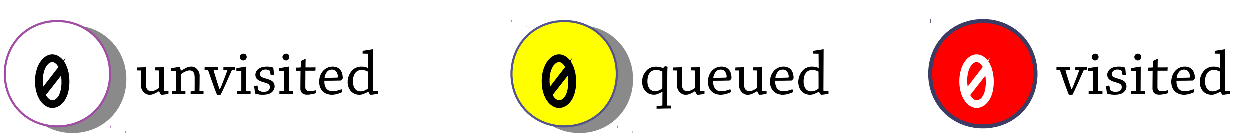
Stack:

Visit order:

0

Step 1:
remove node
from stack
and visit it





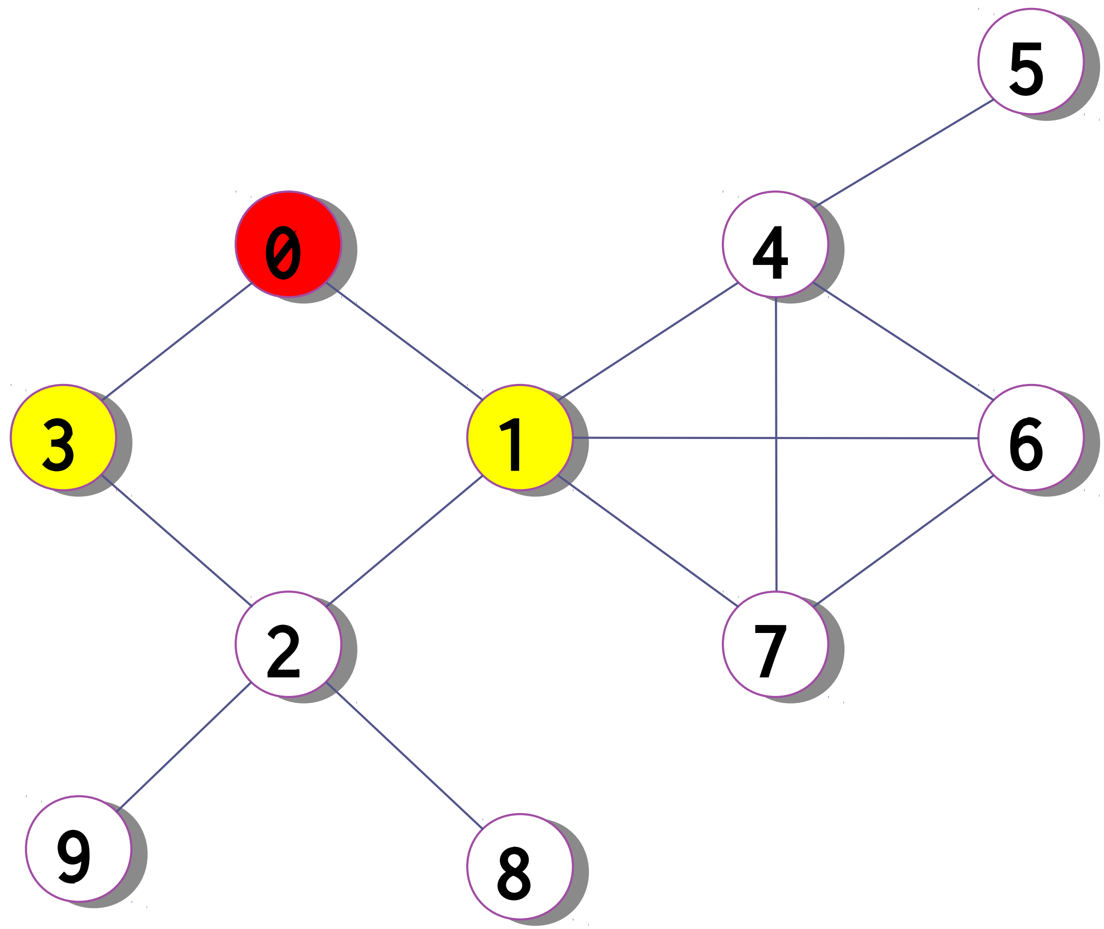
Example of a depth-first search

Stack:

3 1

Visit order:

0



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

Example of a depth-first search

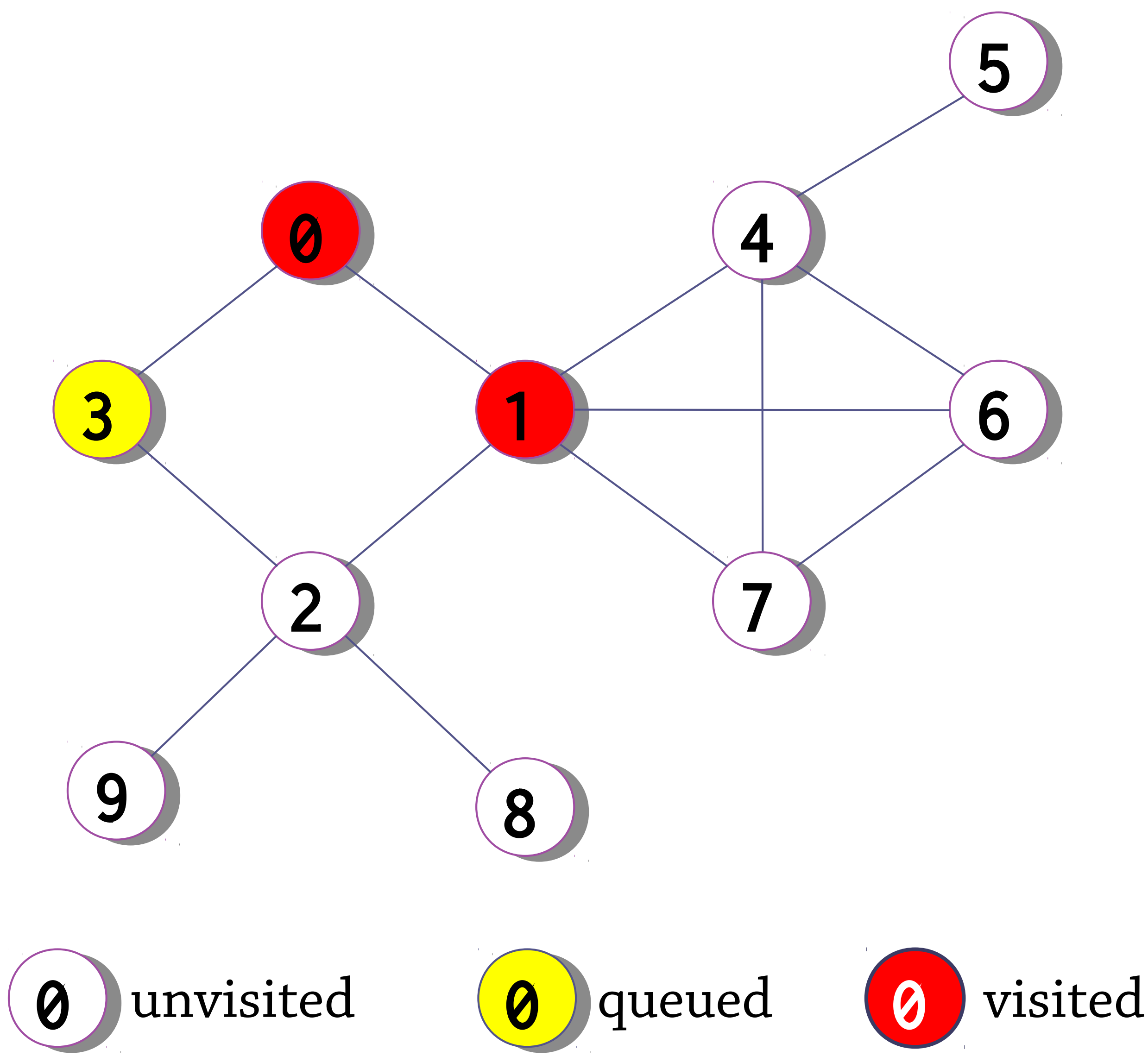
Stack:

3

Visit order:

0 1

Step 1:
remove node
from stack
and visit it



Example of a de rch

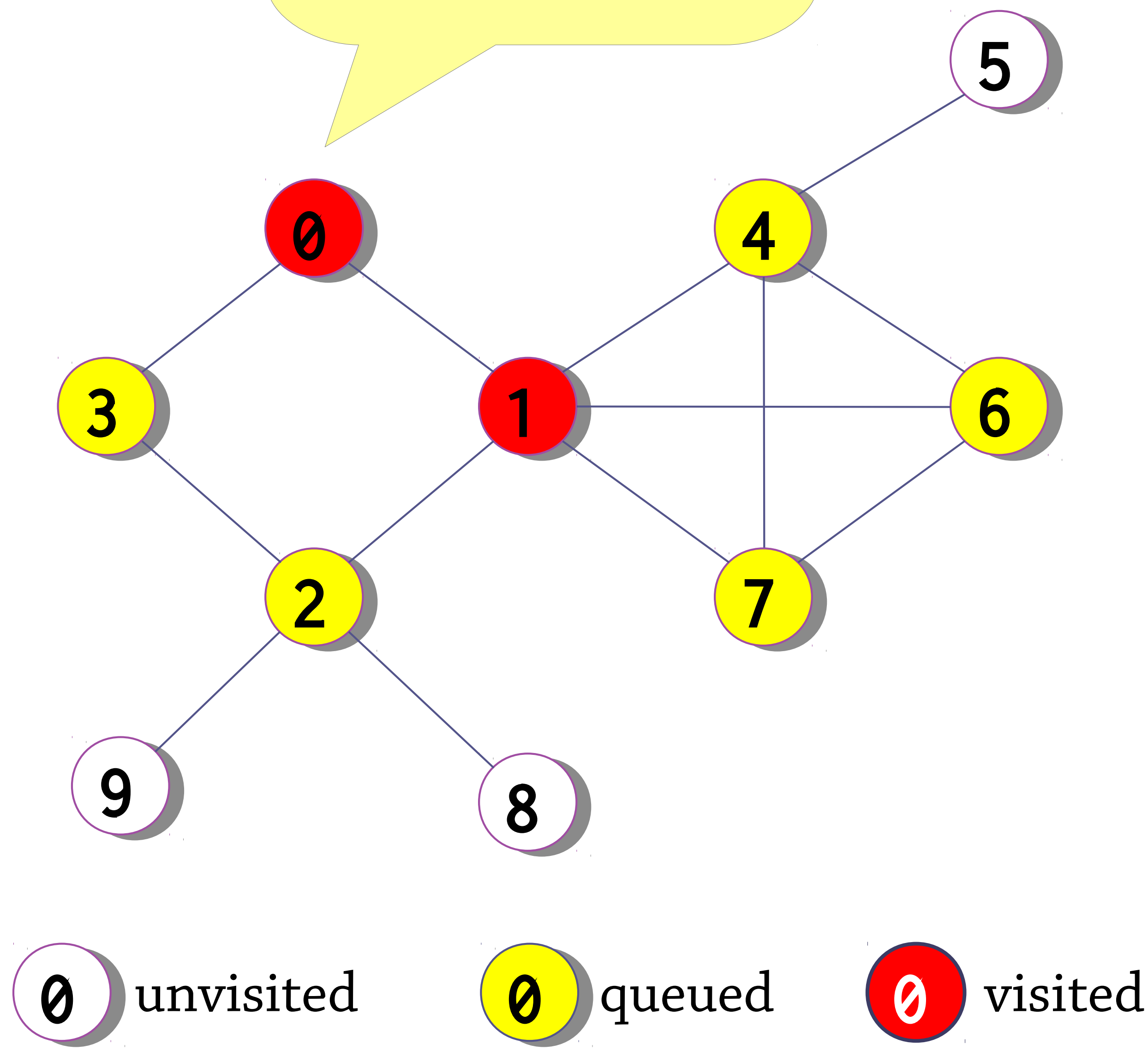
0 is already visited, so we don't add it to the stack

Stack:

3 2 7 4 6

Visit order:

0 1



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

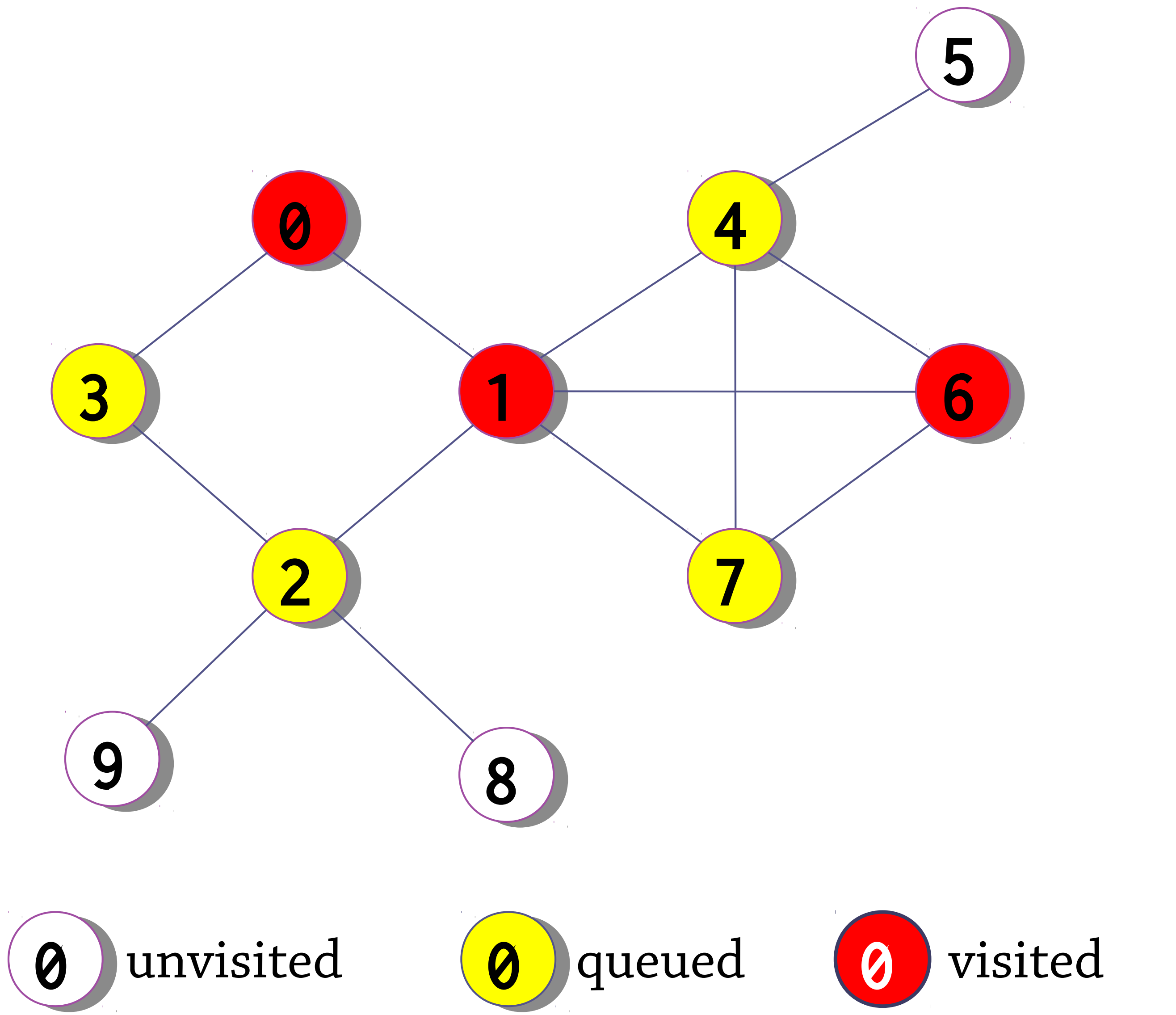
Example of a depth-first search

Stack:

3 2 7 4

Visit order:

0 1 6



Step 1:
remove node
from stack
and visit it

Example of a depth-first search

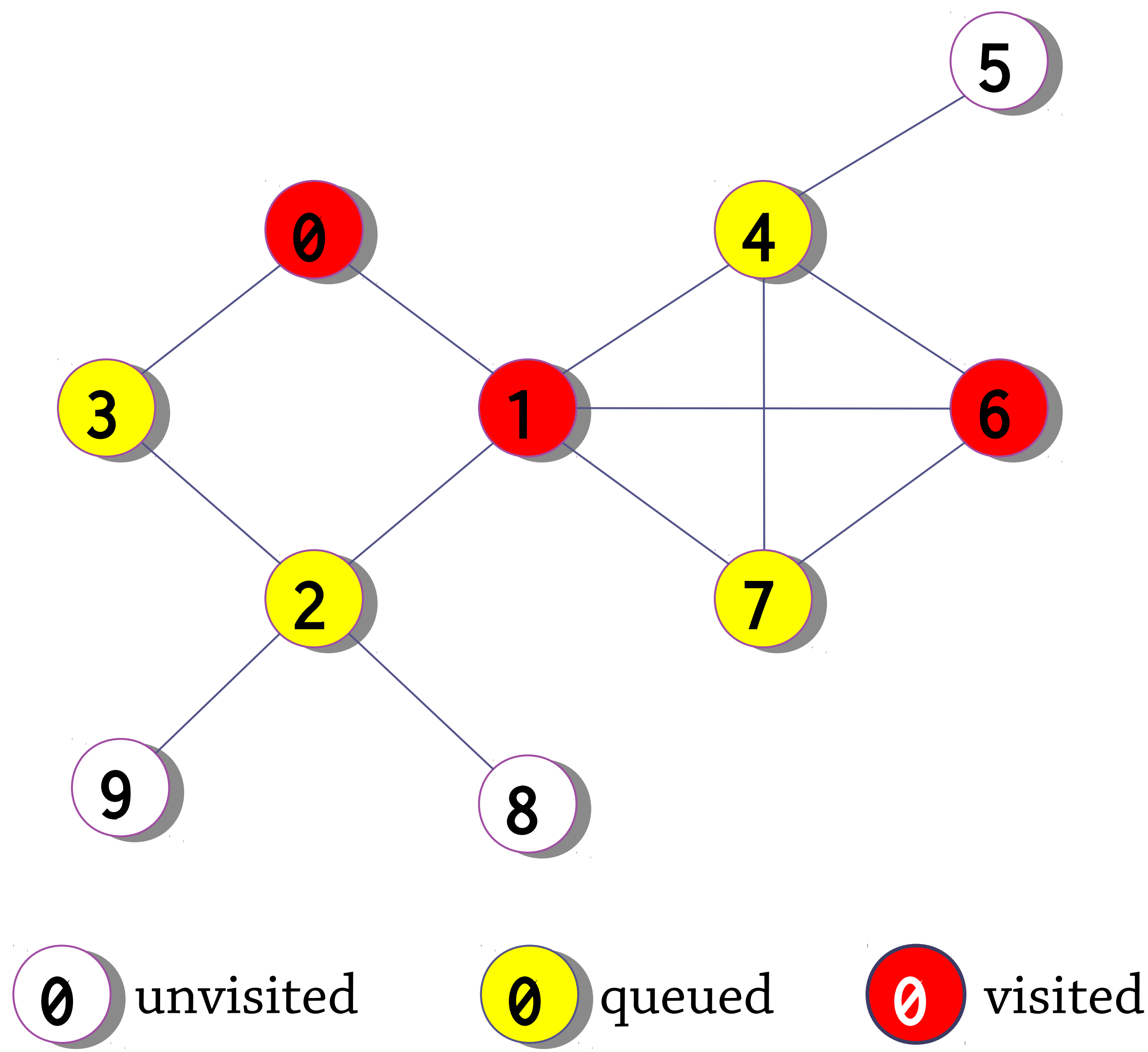
Stack:

3 2 7 4

Visit order:

0 1 6

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



Example of a depth-first search

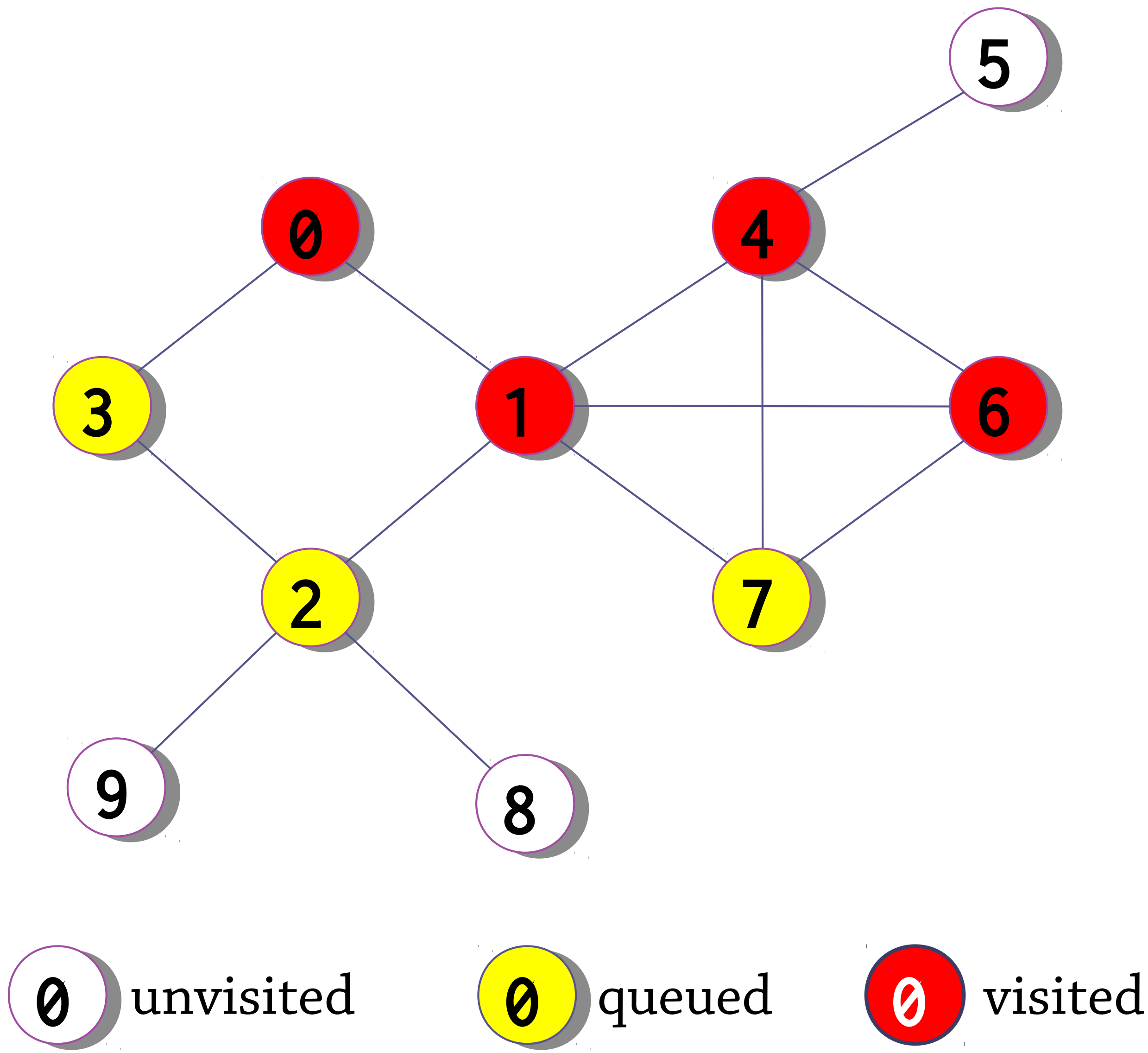
Stack:

3 2 7

Visit order:

0 1 6 4

Step 1:
remove node
from stack
and visit it



Example of a depth-first search

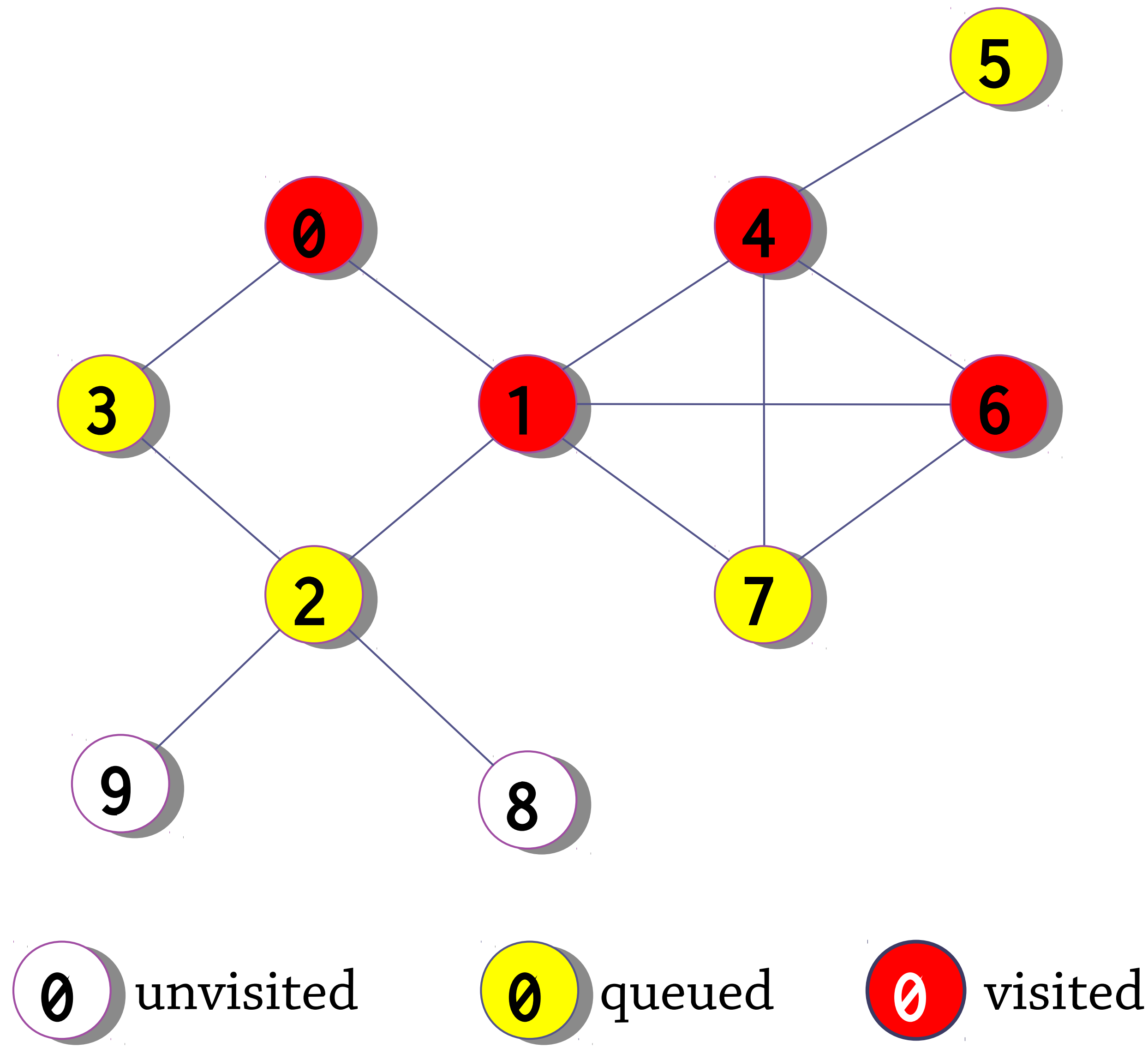
Stack:

3 2 7 5

Visit order:

0 1 6 4

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



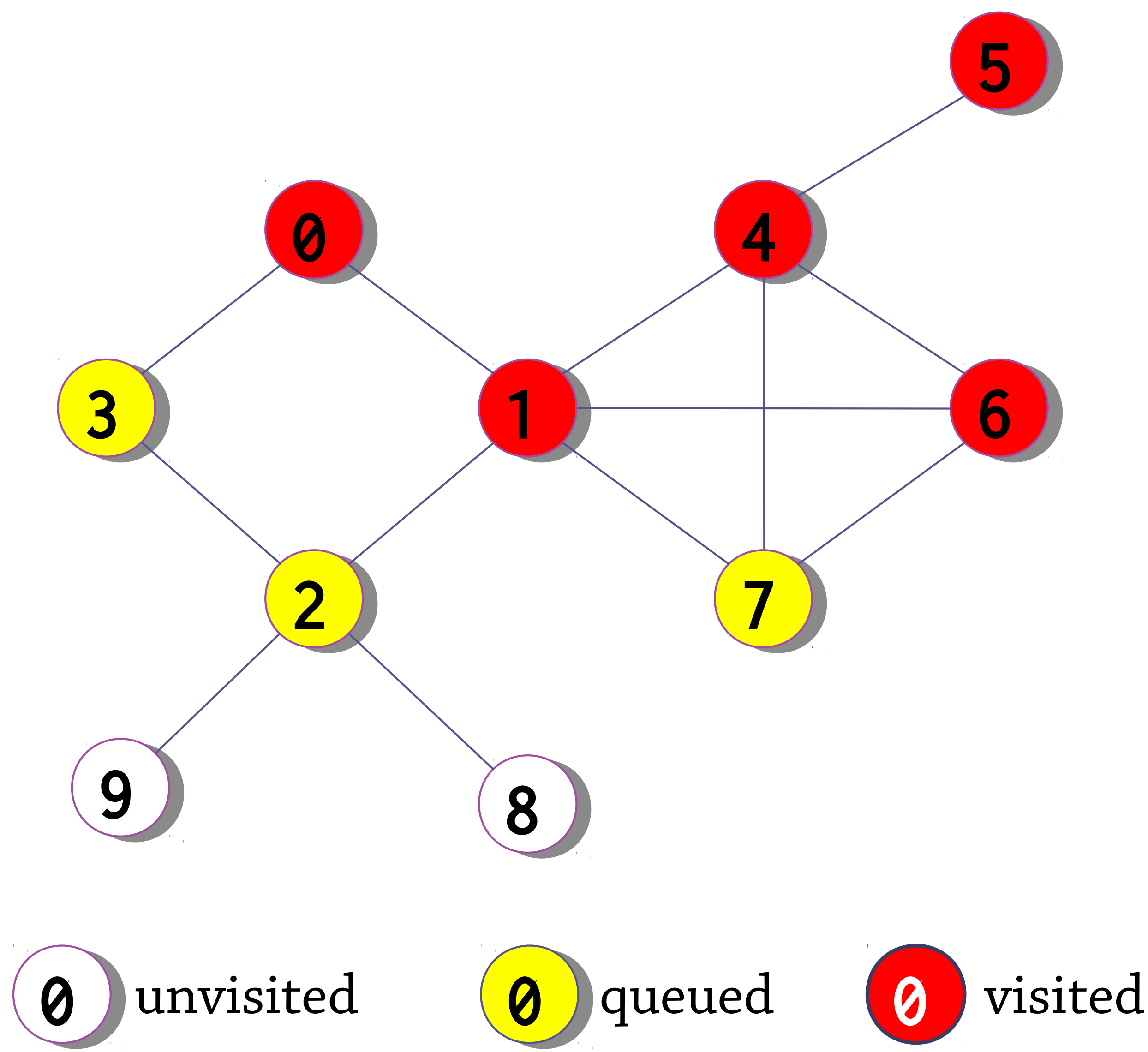
Example of a depth-first search

Stack:

3 2 7

Visit order:

0 1 6 4 5



Step 1:
remove node
from stack
and visit it

Example of a depth-first search

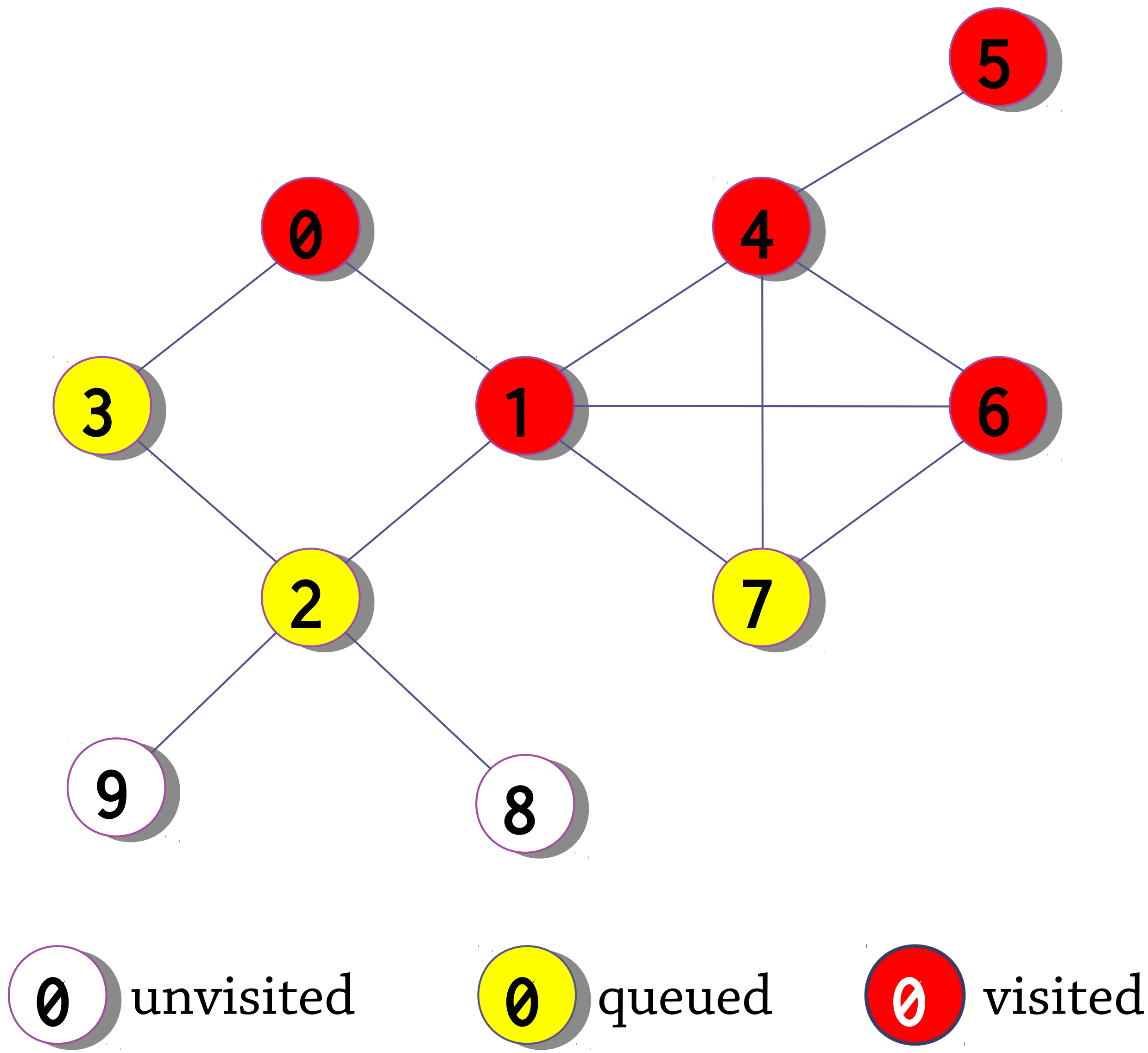
Stack:

3 2 7

Visit order:

0 1 6 4 5

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



Example of a depth-first search

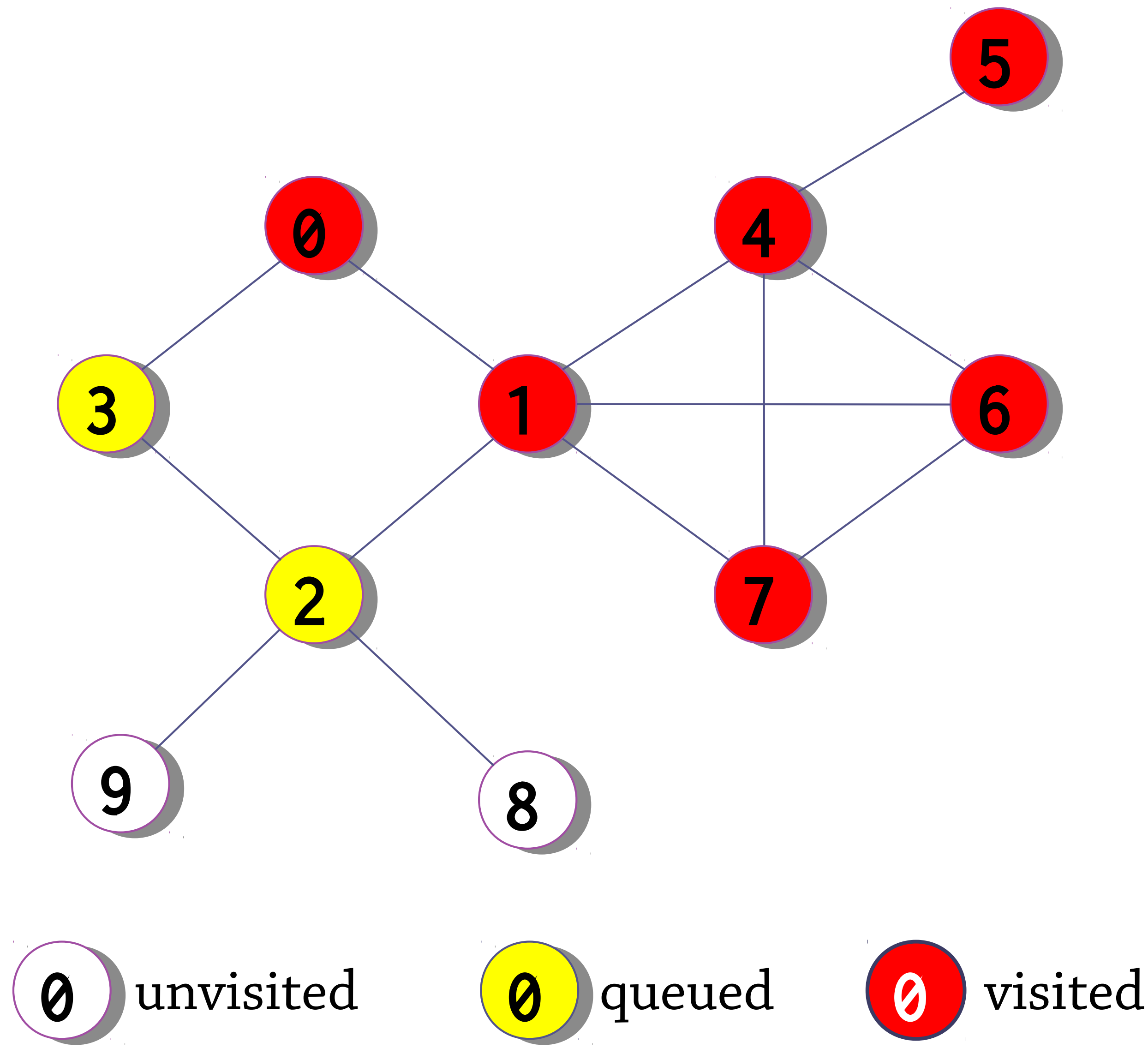
Stack:

3 2

Visit order:

0 1 6 4 5 7

Step 1:
remove node
from stack
and visit it



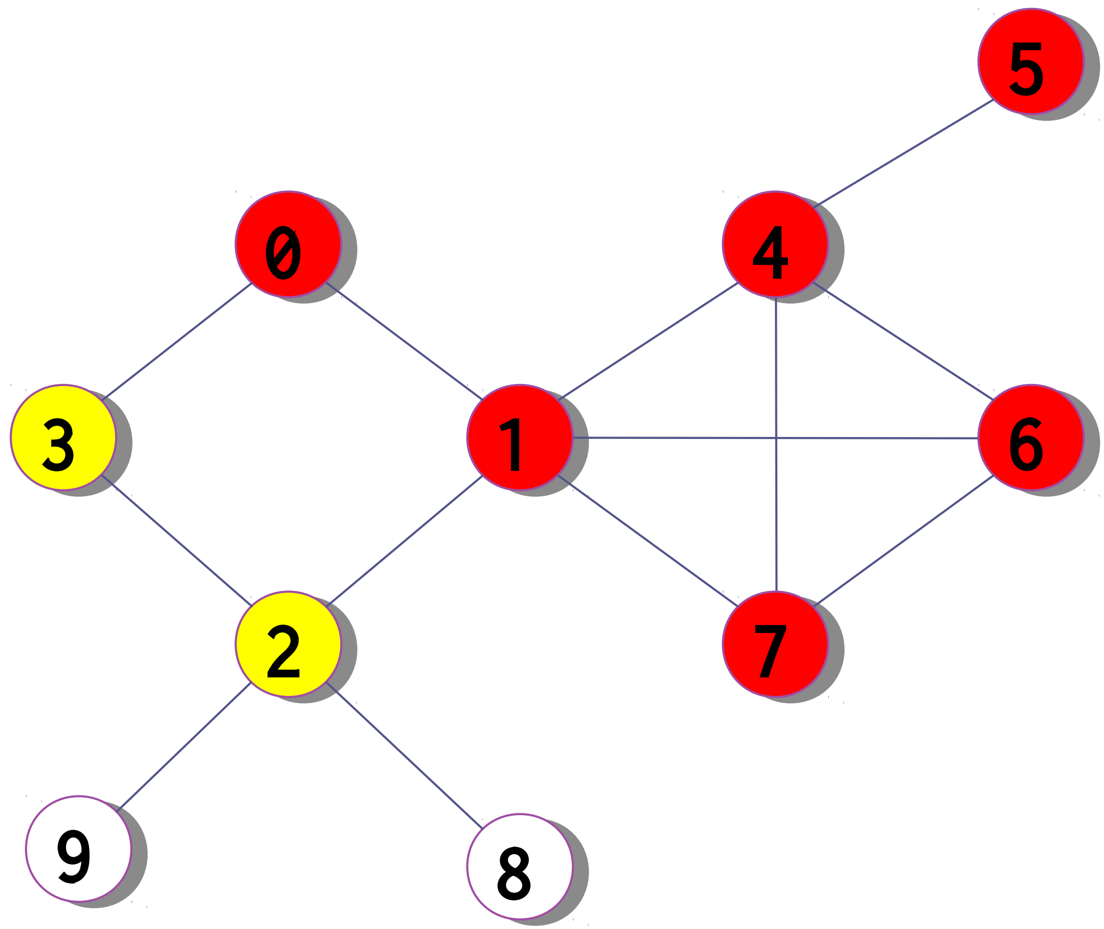
Example of a depth-first search

Stack:

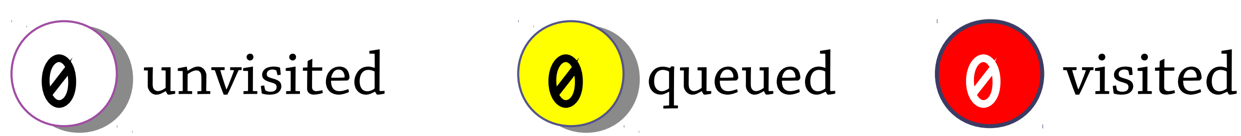
3 2

Visit order:

0 1 6 4 5 7



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



Example of a depth-first search

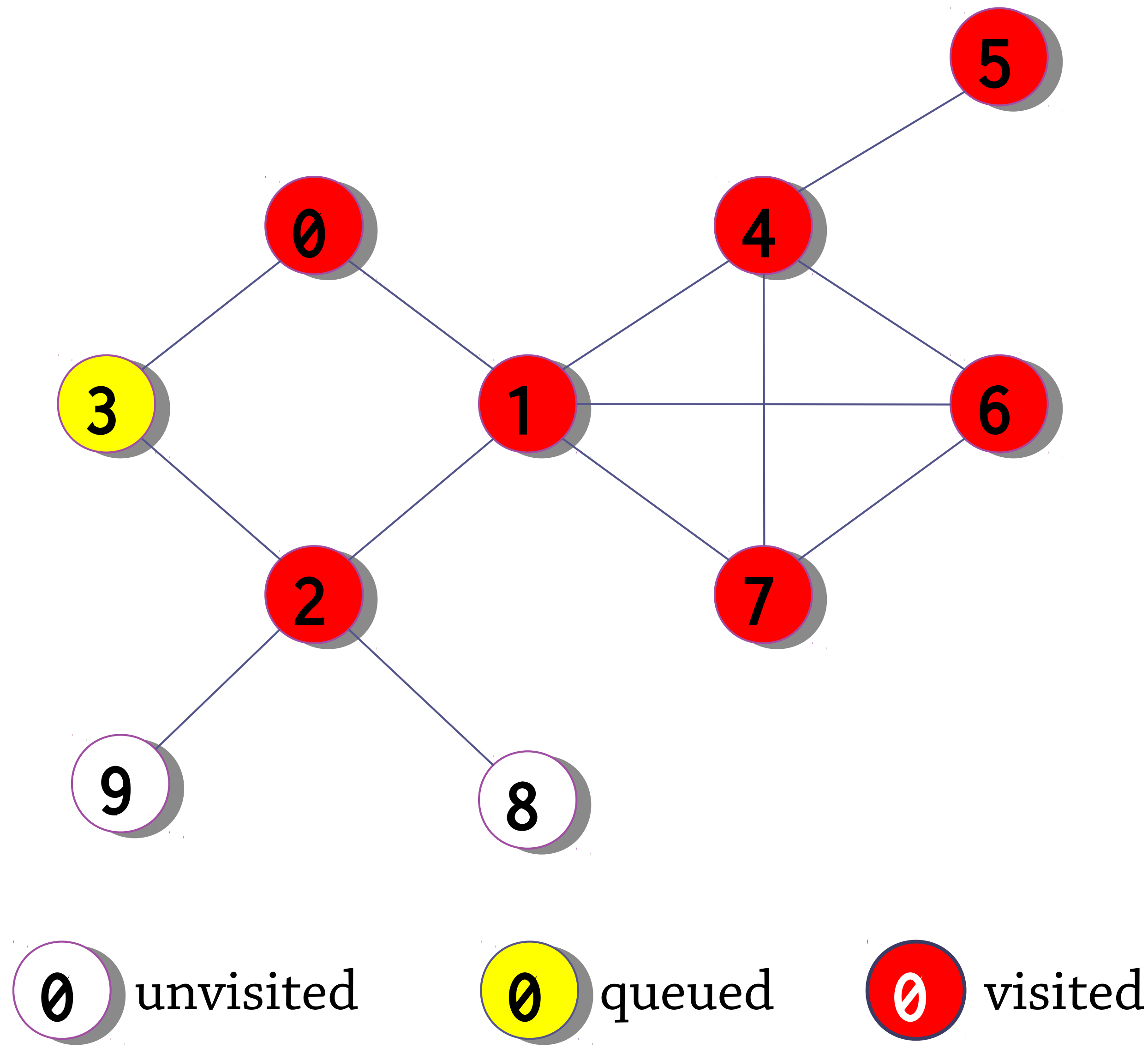
Stack:

3

Visit order:

0 1 6 4 5 7 2

Step 1:
remove node from stack and visit it



Example of a depth-first search

Stack:

3 9 8

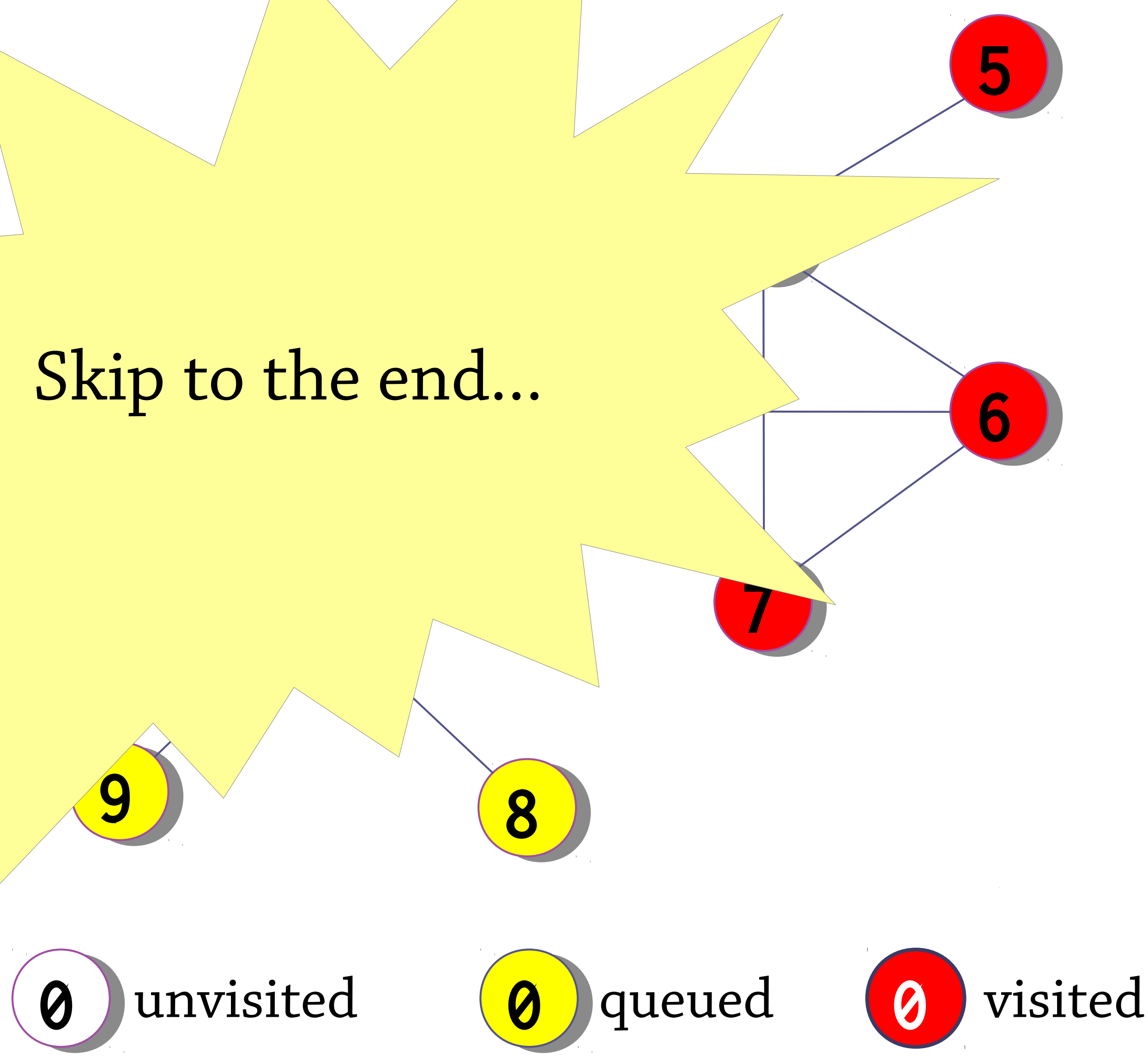
Visit orde

0 1 6 4 5

7 2

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

Skip to the end…

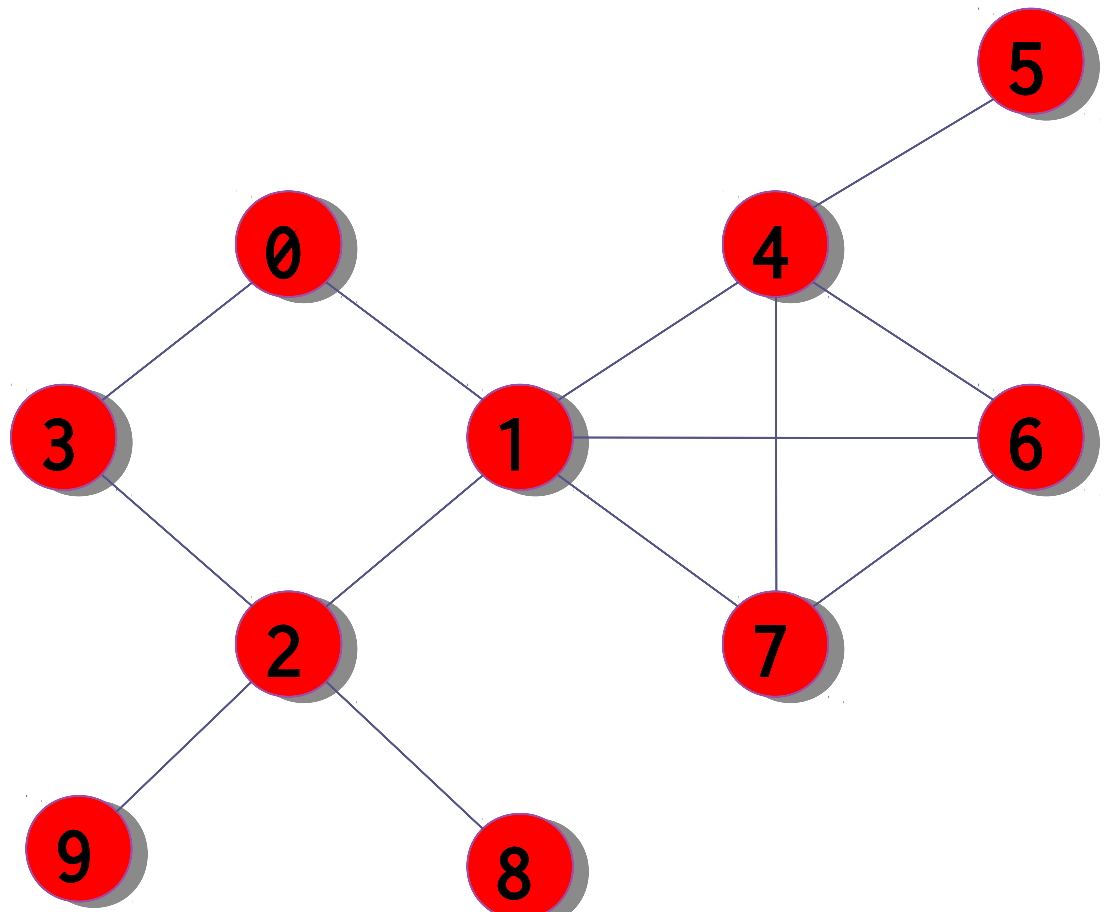


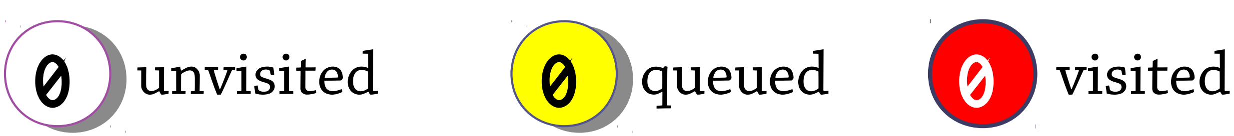
Example of a depth-first search

Stack:

Visit order:

0 1 6 4 5
7 2 8 9 3





Complexity of BFS and DFS

We only look at each edge once (twice for undirected graphs)

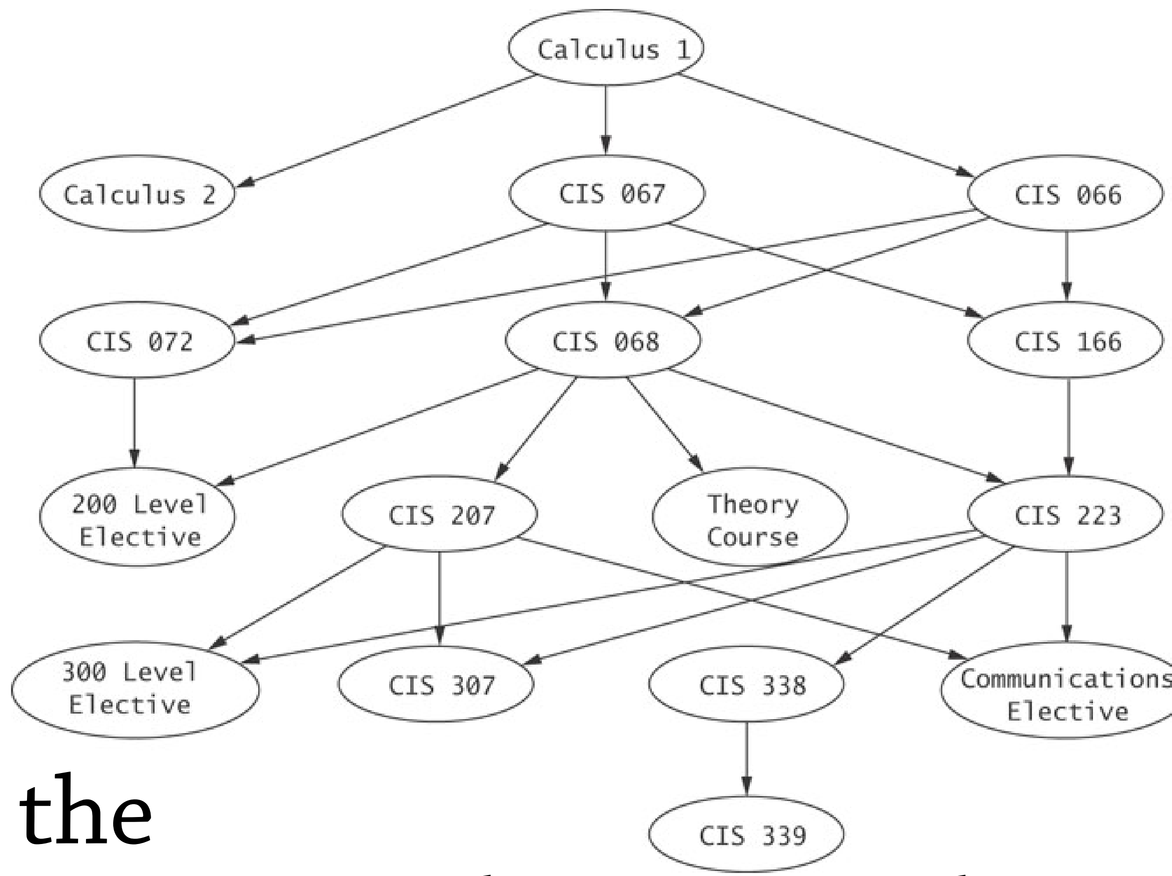
- So we look at maximum $|E|$ edges
- ($2 \times |E|$ for undirected graphs)

Complexity is therefore $O(|E|)$ - for both breadth-first and depth-first search

Directed acyclic graphs

Here is a directed acyclic graph (DAG)

A DAG is a directed graph without cycles



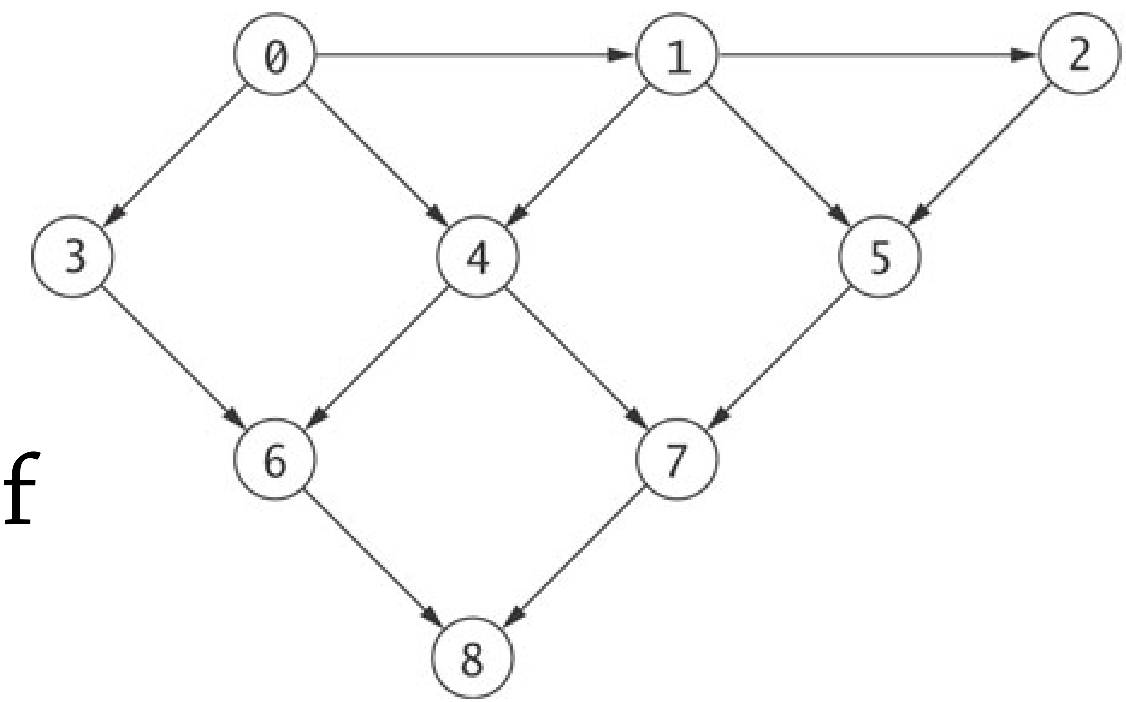
That means: once you follow an edge there is no way back to the source node – we can say that one node is *after* another in the graph

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

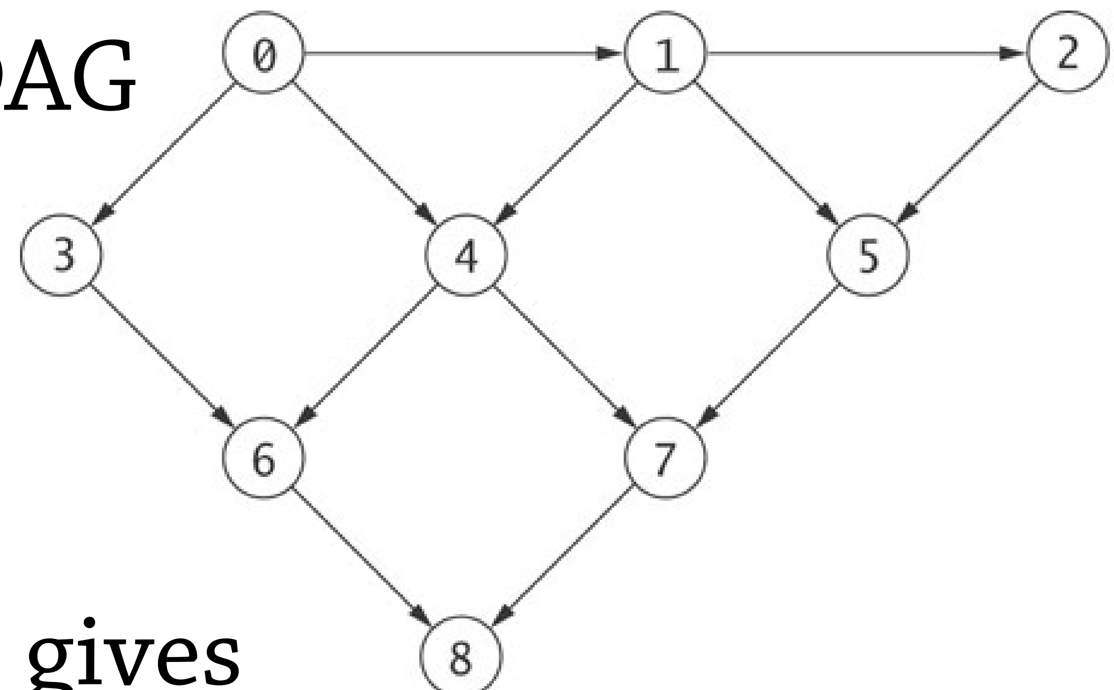
012345678 is a topological sort of this DAG, but 015342678 isn't.



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
- So if we print each node as we visit it, we will almost get a topological sort but in reverse order
- So, by printing the nodes in the reverse order we visit them, we will topologically sort the graph!

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

Some basic algorithms:

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