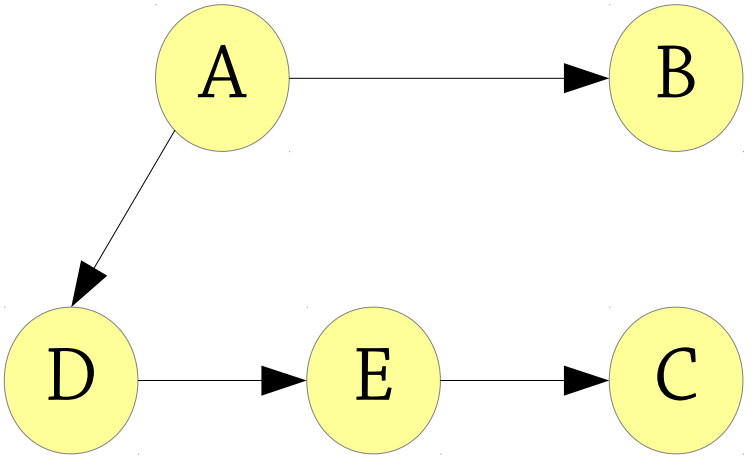
Graphs

Graphs

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

- An edge is a connection between two nodes

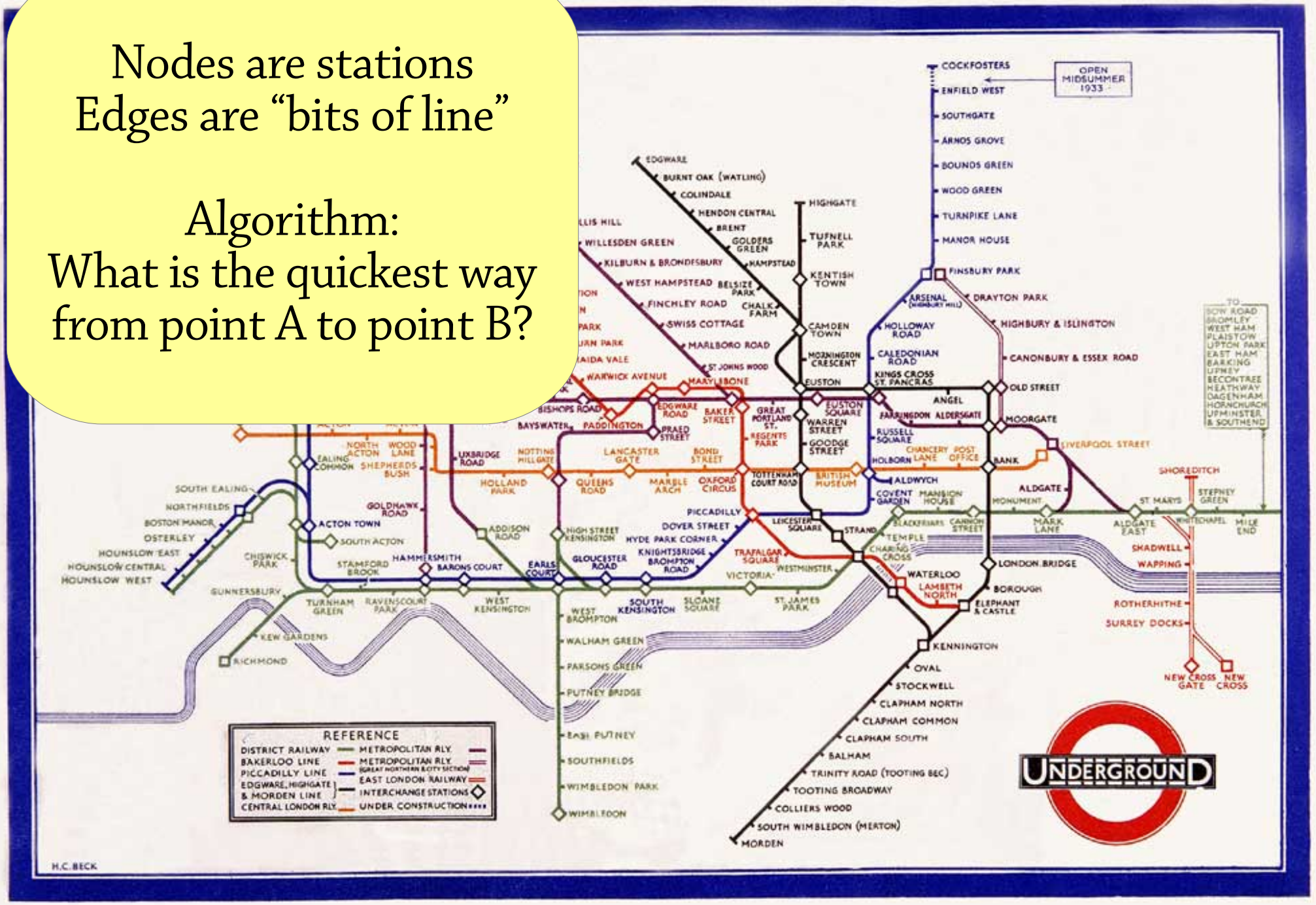


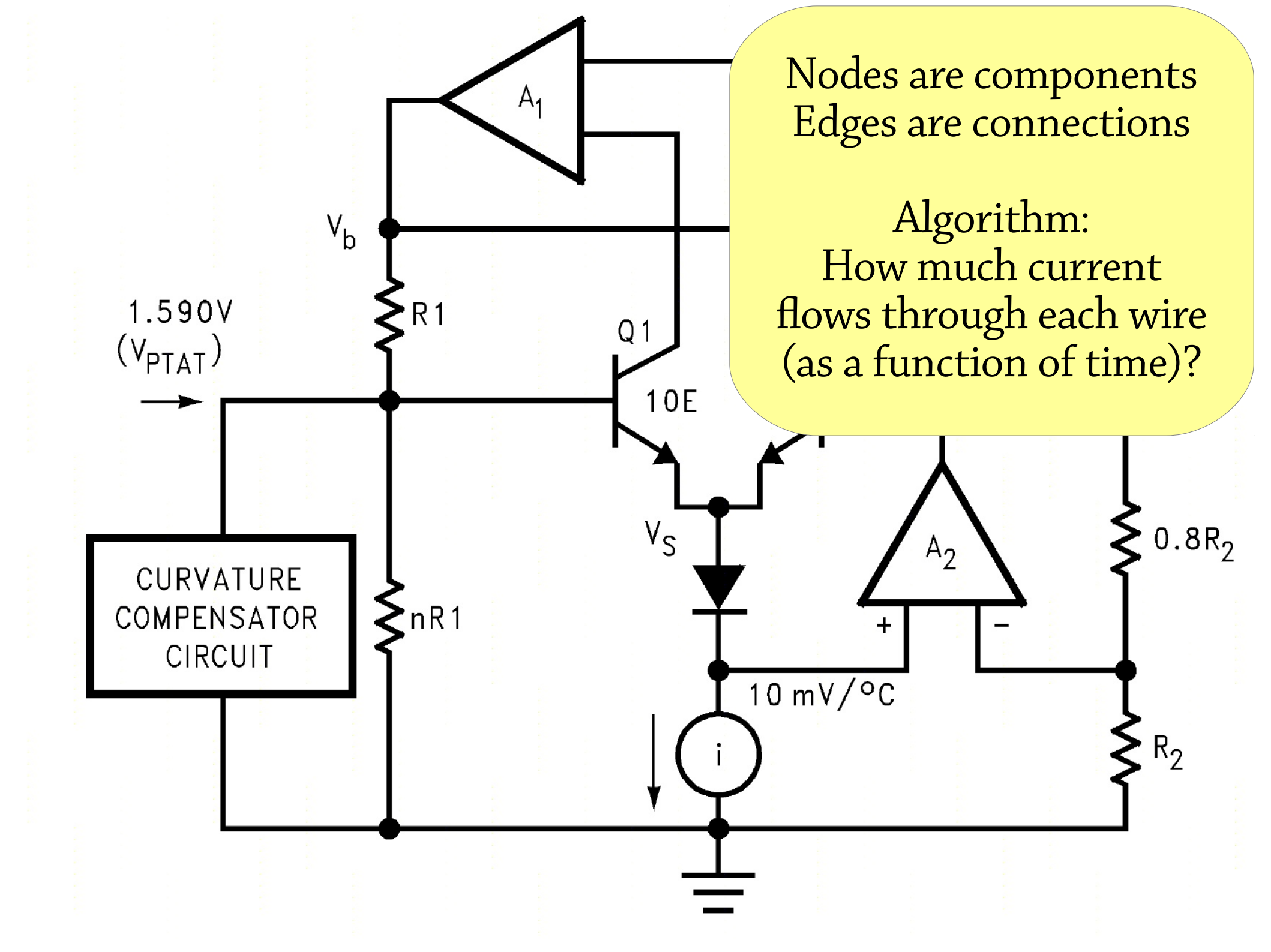
Nodes: A, B, C, D, E

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

Nodes are stations
Edges are "bits of line"

Algorithm:
What is the quickest way from point A to point B?





Nodes are components
Edges are connections

Algorithm:
How much current
flows through each wire
(as a function of time)?

Graphs

Graphs are used all over the place:

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

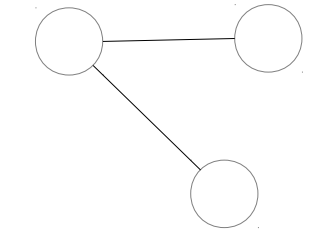
Anywhere where you have connections or relationships!

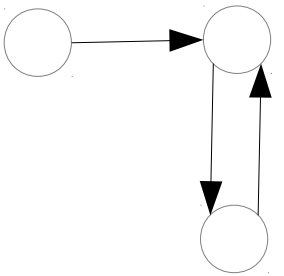
Normally the vertices and edges are *labelled* with relevant information!

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 (a *digraph*), the edge goes from the *source node* to the *target node* (we draw an arrow from the source to the target)



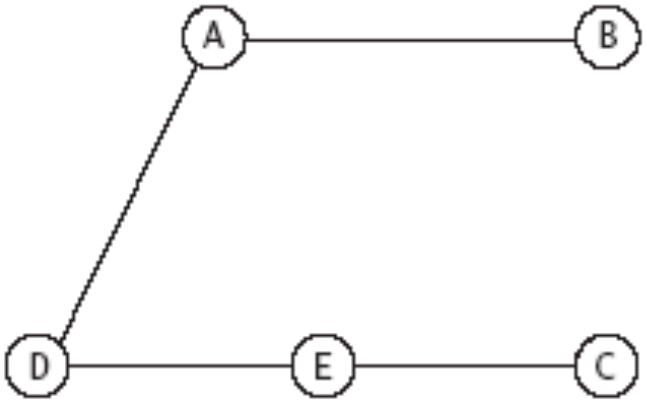


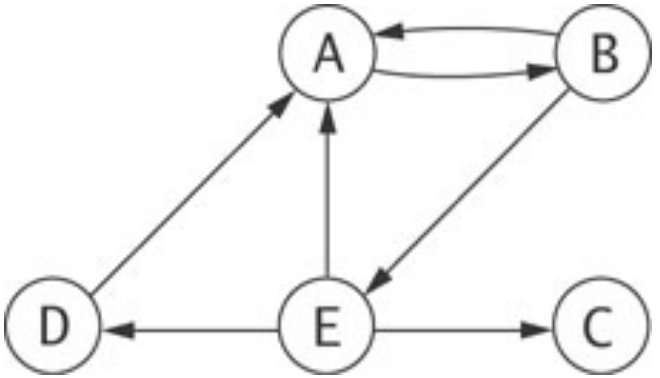
A tree is a special case of a directed graph

- Edge from parent to child
- A certain node is identified as root

Drawing graphs

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



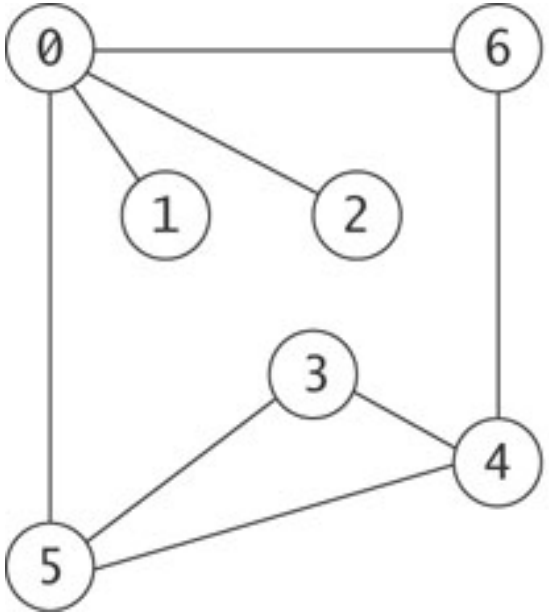


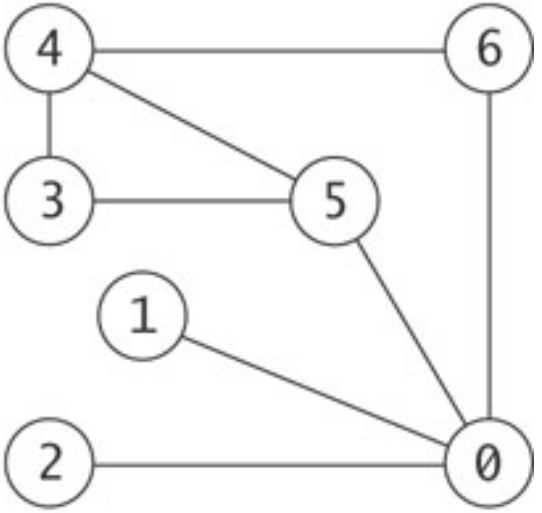
$V = \{A, B, C, D, E\}$
$E = \{(A, B), (A, D), (C, E), (D, E)\}$

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

Drawing graphs

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



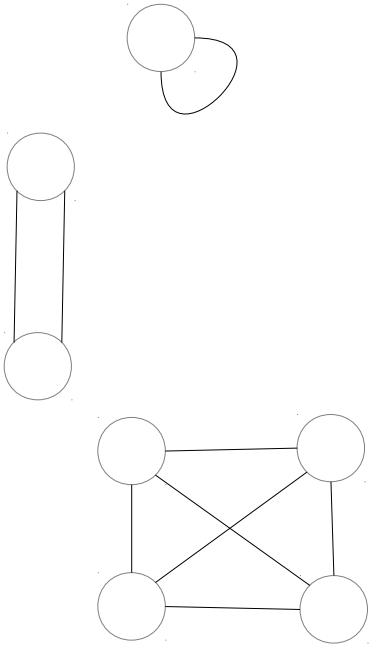


$V = \{0, 1, 2, 3, 4, 5, 6\}$

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

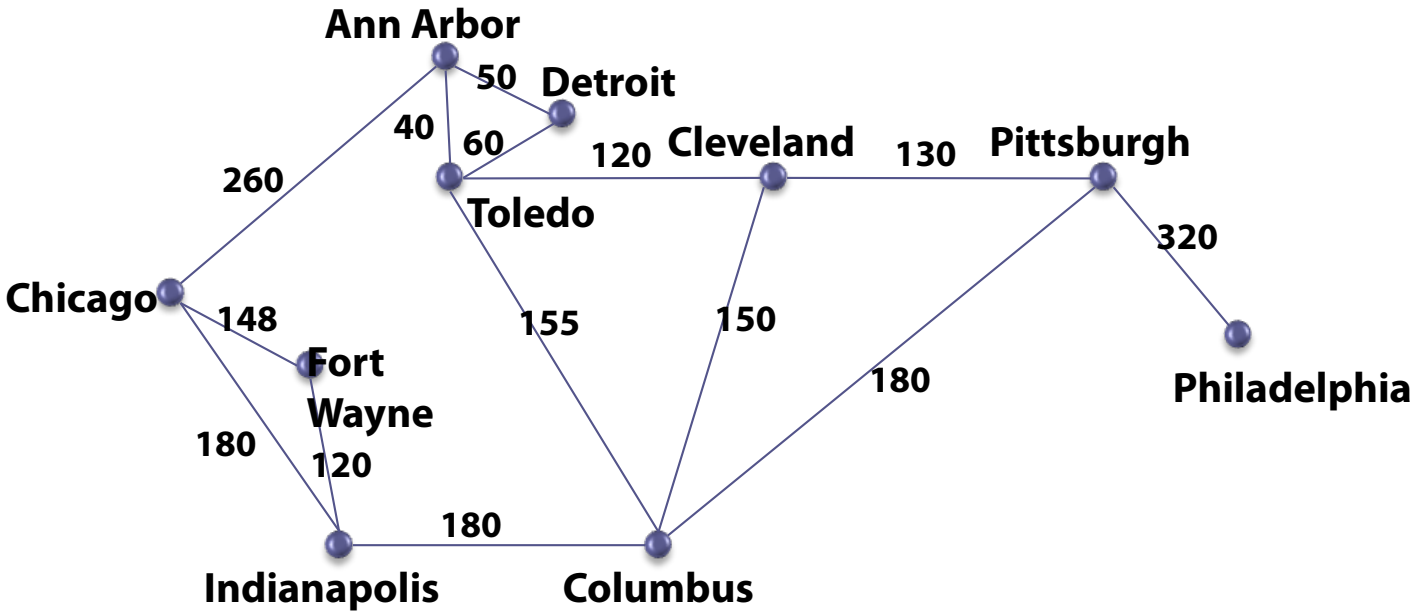
Graph terminology and properties

- A *loop* is an edge from a node to itself – often not allowed.
- A *multigraph* is a graph with multiple edge between the same pair of nodes – often not allowed.
- In a *complete* graph is every possible edge present.
- In complete graphs, the number of edges, $|E|$, is proportional to $|V|^2$.
  - Directed, with loops: $|E| = |V|^2$
  - Directed, without loops: $|E| = |V|(|V| - 1)$
  - Undirected, with loops: $|E| = |V|(|V| + 1)/2$
  - Undirected, without loops: $|E| = |V|(|V| - 1)/2$



Weighted graphs

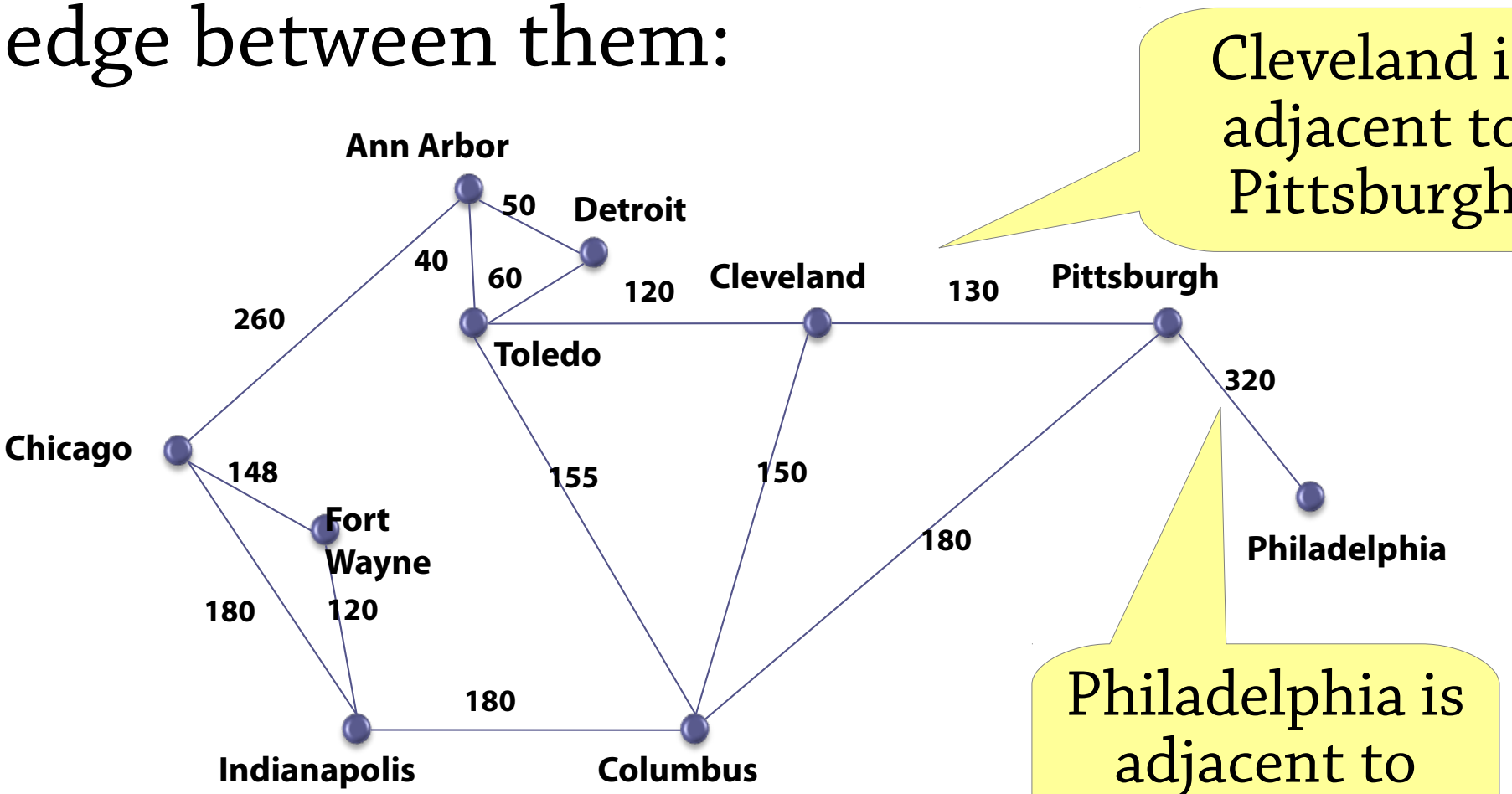
In a *weighted graph*, each edge has a *weight* associated with it:



A graph can be directed, weighted, neither or both

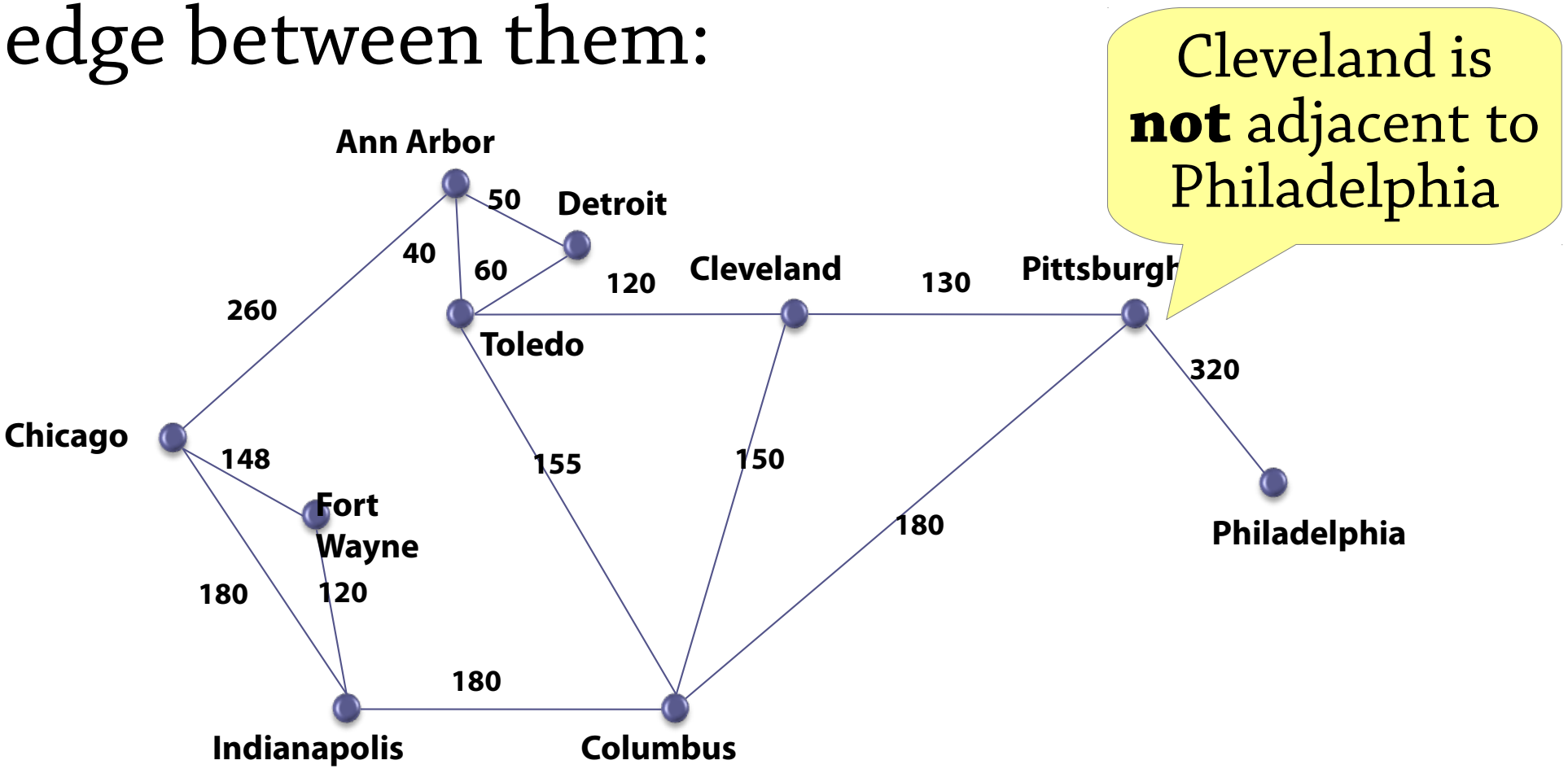
Paths and cycles

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



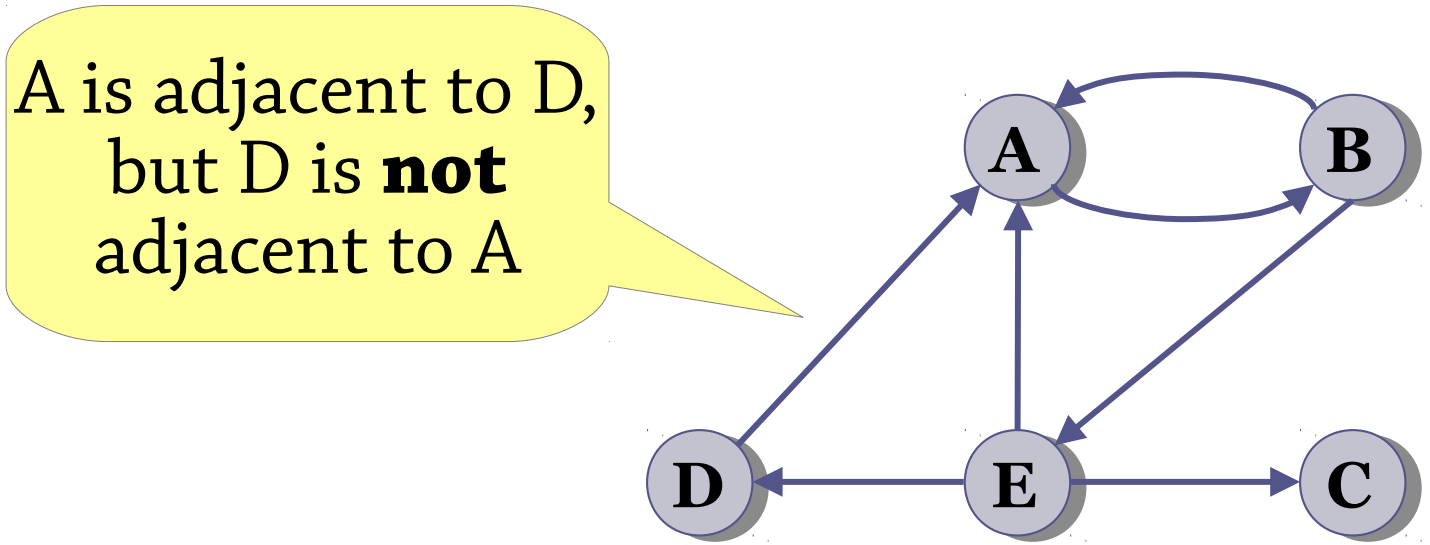
Paths and cycles

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



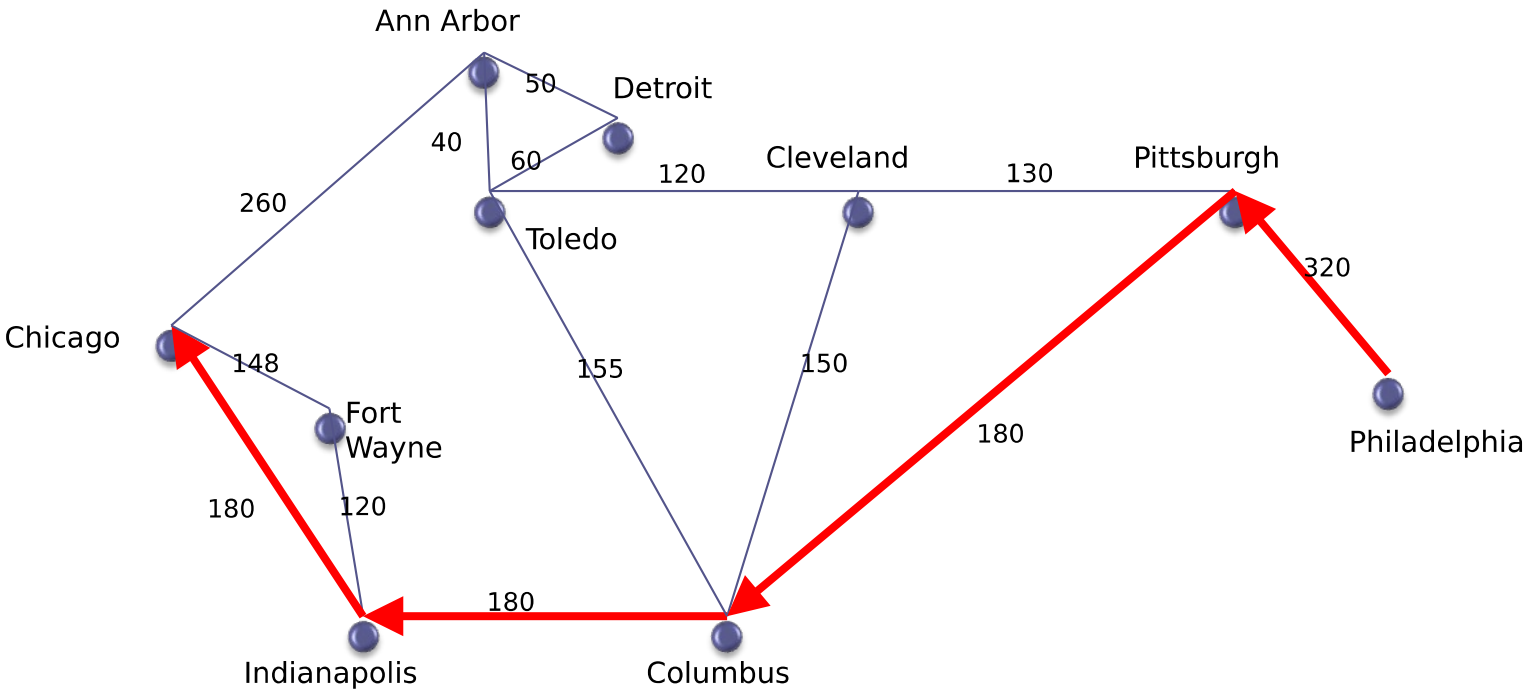
Paths and cycles

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



Paths

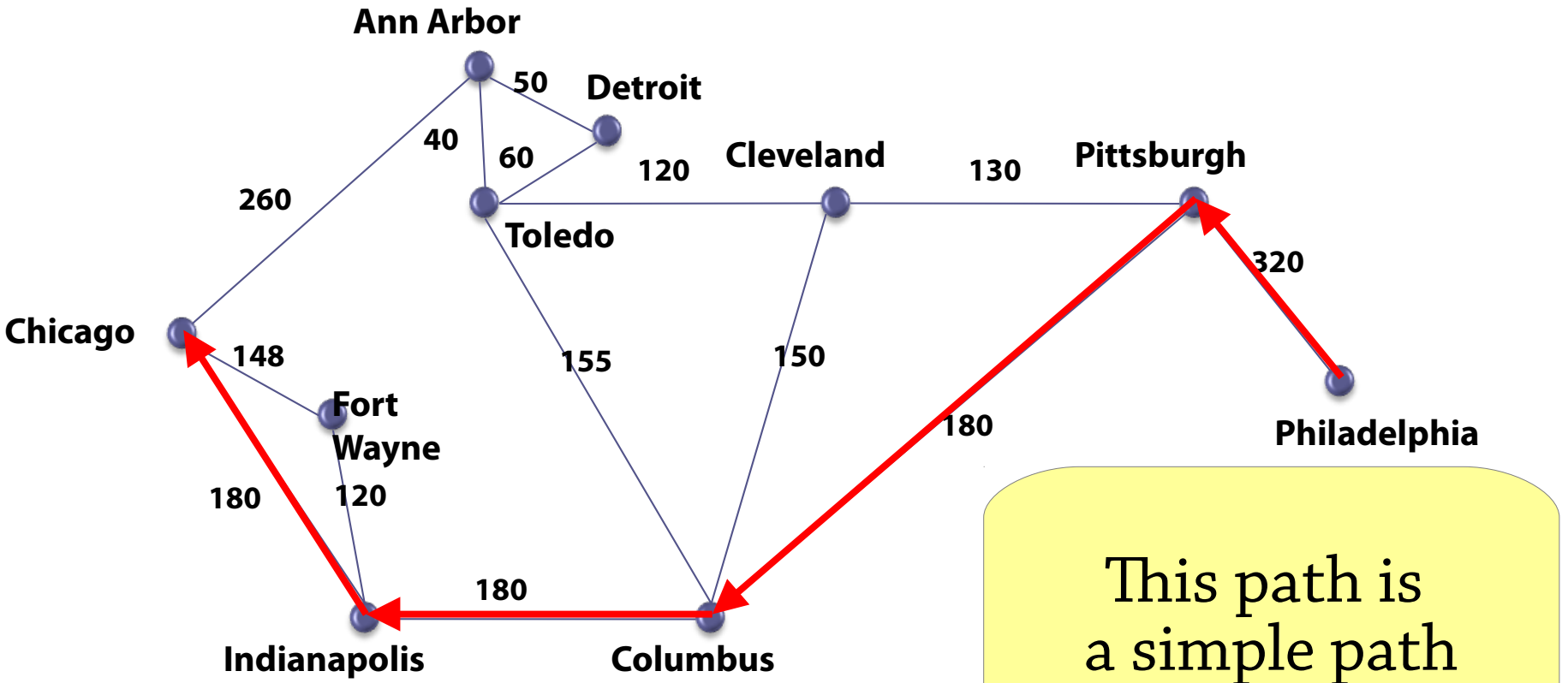
A *path* is a sequence of edges that take you from one node to another



If there is a path from node A to node B, we say that B is *reachable* from A

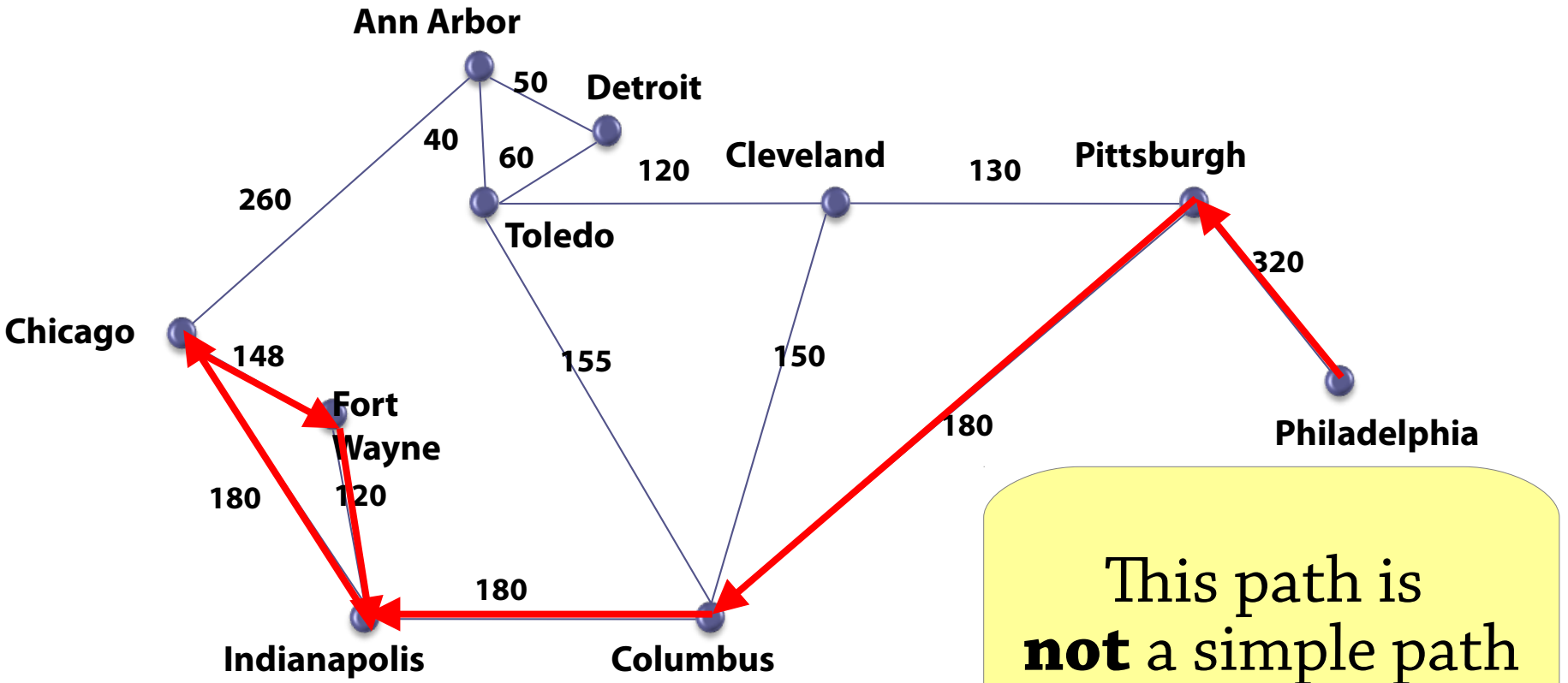
Paths and cycles

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



Paths and cycles

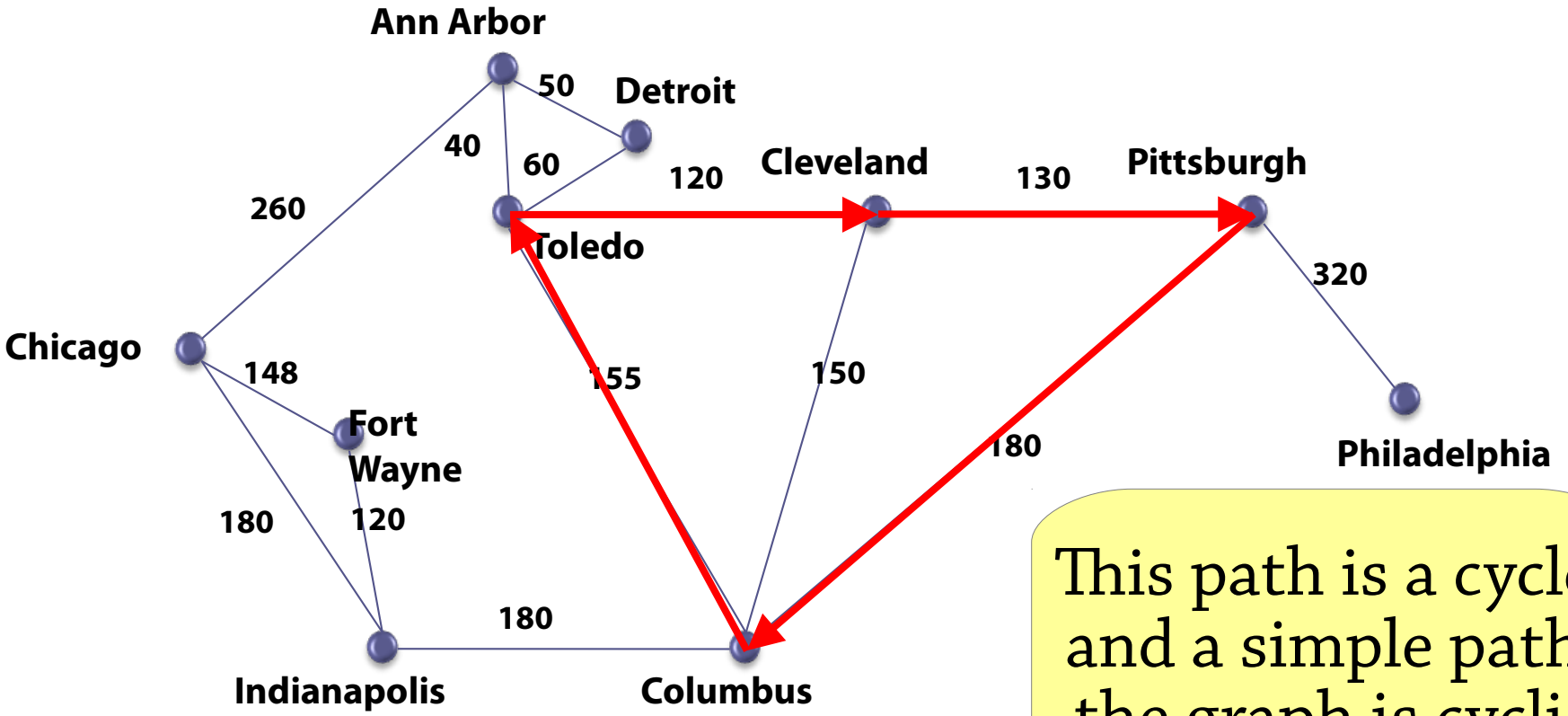
In a *simple path*, no node or edge appears twice, except that the first and last node can be the same



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

DAG

- A *DAG* is a directed, acyclig graph.

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*

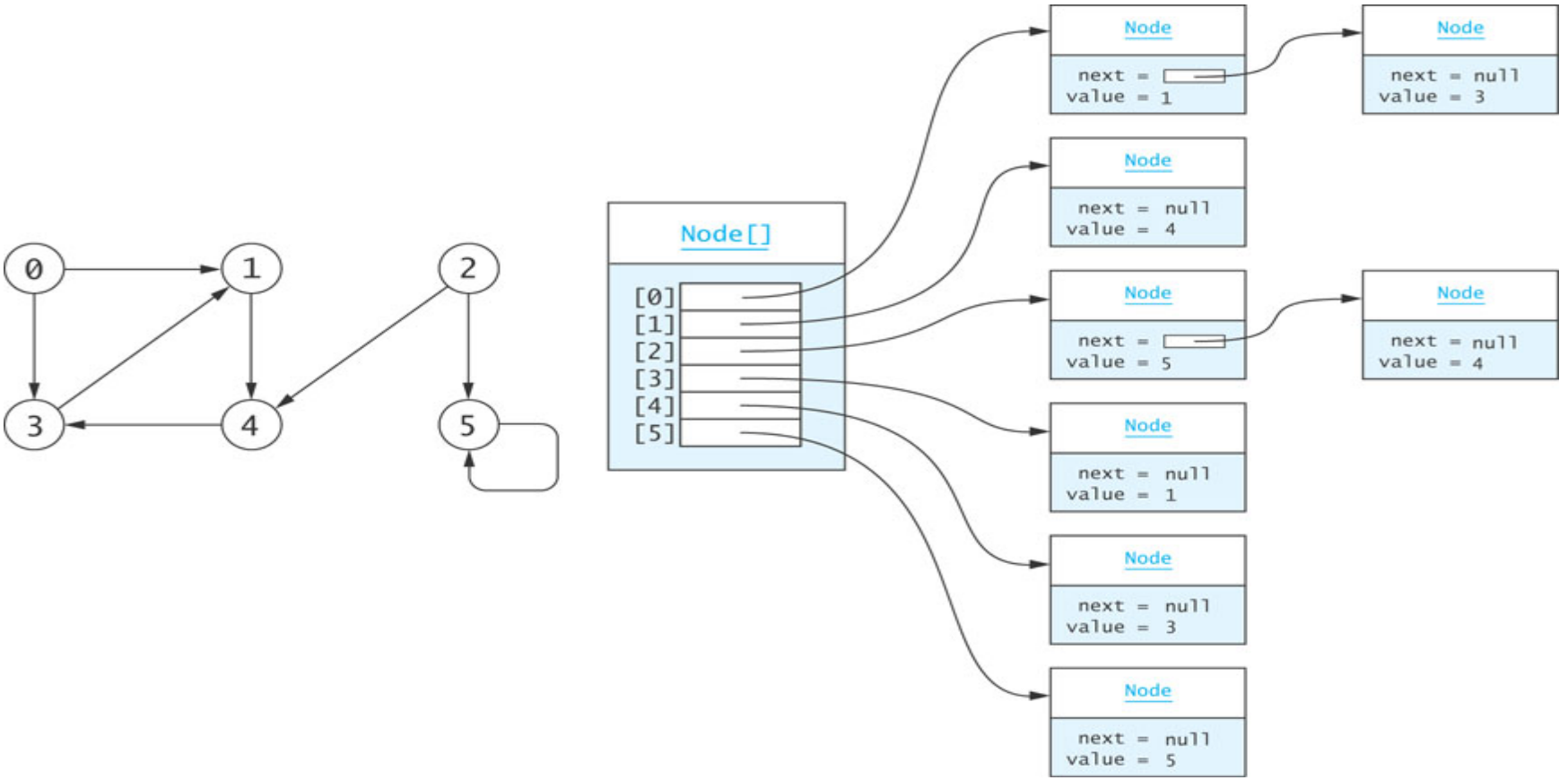
Keep a 2-dimensional array, with one entry for each pair of nodes

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

How to implement a graph

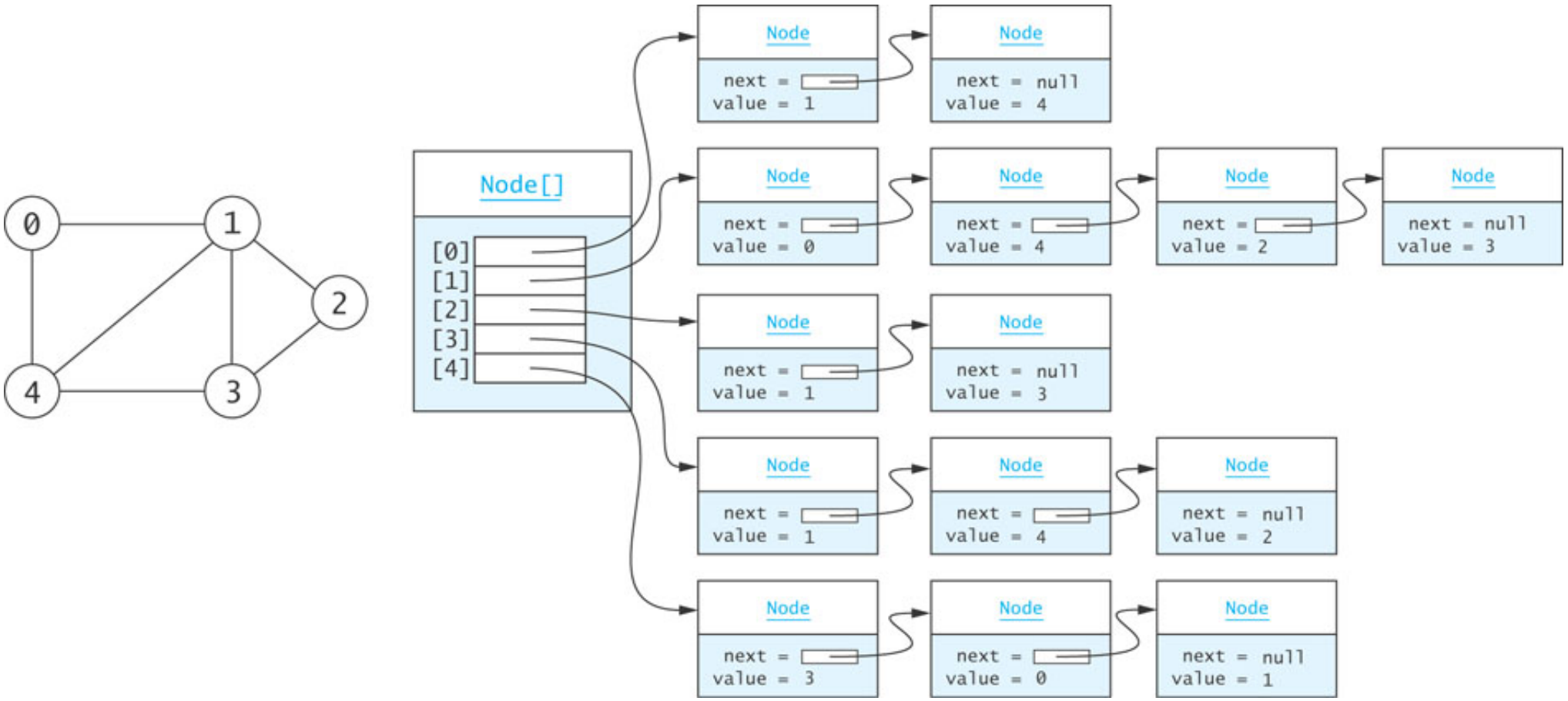
Typically: *adjacency list*

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



Adjacency list – undirected graph

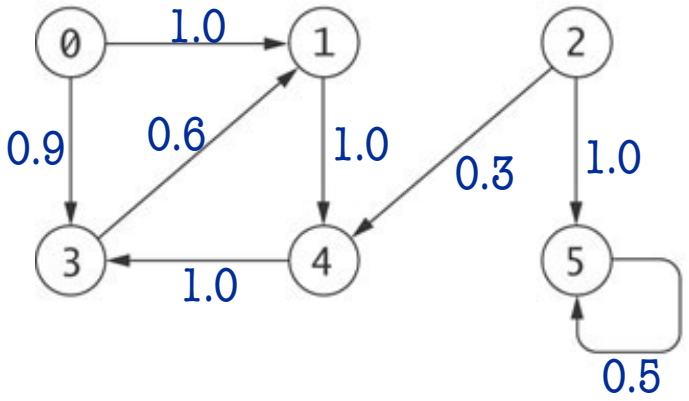
Each edge appears twice, once for the source and once for the target node



Adjecency matrix

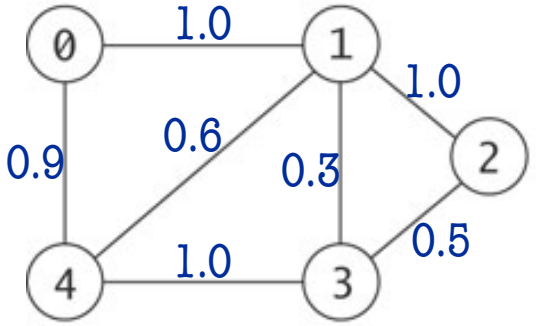
- The other main way to implement graph representation is *adjecency matrix*.
- A matrix with dimensions $|V| x |V|$. Each element (i,j) is true/contains an edge label if there is an edge from nod i to node j.
- For undirected graphs the matrix is symmetric or only one of the halves is used.
- Adjecency matrix representation can be preferable for *dense* graphs, i.e. in graphs where a large portion of the possible edges are present.
- For graphs which are not dense, the matrix representation is a waist of space.

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	

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

Object variables correspond to associations/edged in the class diagram of your program. Classes correspond to nodes.
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 algorithms:
depth-first search,
reachability,
connected components

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

Reachability

How can we tell what nodes are reachable from a given node?

We can start exploring the graph from that node, but we have to be careful not to (e.g.) get caught in cycles

Depth-first search is one way to explore the part of the graph reachable from a given node

Depth-first search

Depth-first search is a *traversal* algorithm

- This means it takes a node as input, and enumerates all nodes reachable from that node

It comes in two variants, *preorder* and *postorder* – we'll start with preorder

To do a *preorder* DFS starting from a node:

- visit the node
- for each outgoing edge from the node,
  recursively DFS the target of that edge,
  unless it has already been visited

It's called preorder because we visit each node *before* its outgoing edges

Implementing DFS

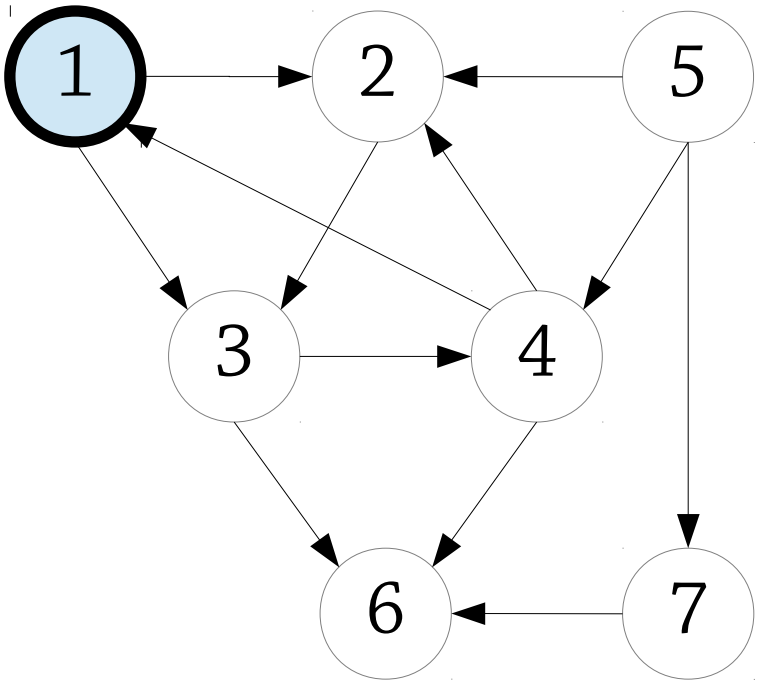
- We maintain a *stack* of nodes that we are going to visit next
  - Initially, the stack contains the start node
- We repeat the following process:
  - Remove a node from the stack
  - Visit it
  - Find all nodes adjacent to the visited node and add them to the stack, *unless* they have been visited or added to the stack already

Example of a depth-first search

Visit order: 1

DFS node 1

(By the way, is 5 reachable from 1?)



= current

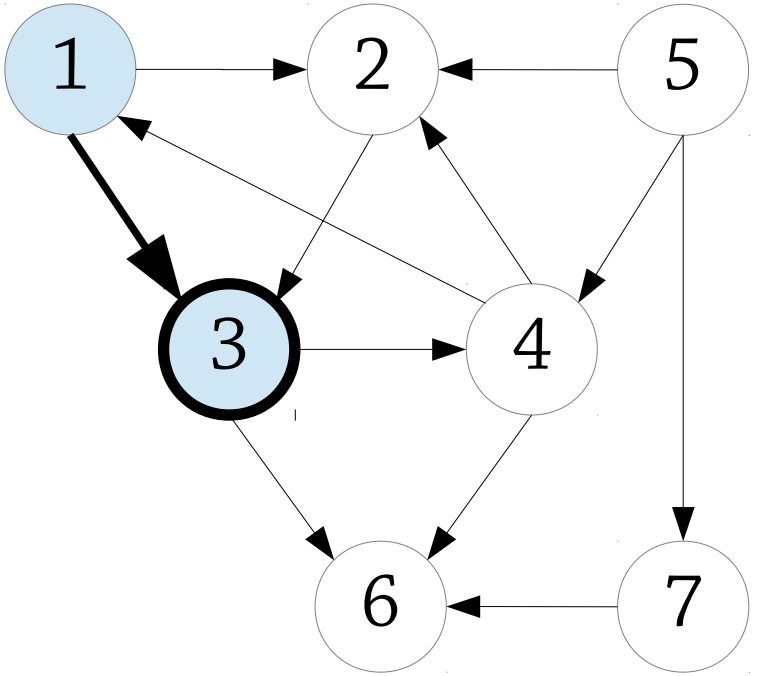
= unvisited

= visited

Example of a depth-first search

Visit order: 1 3

Follow edge 1 → 3,
recursively DFS node 3

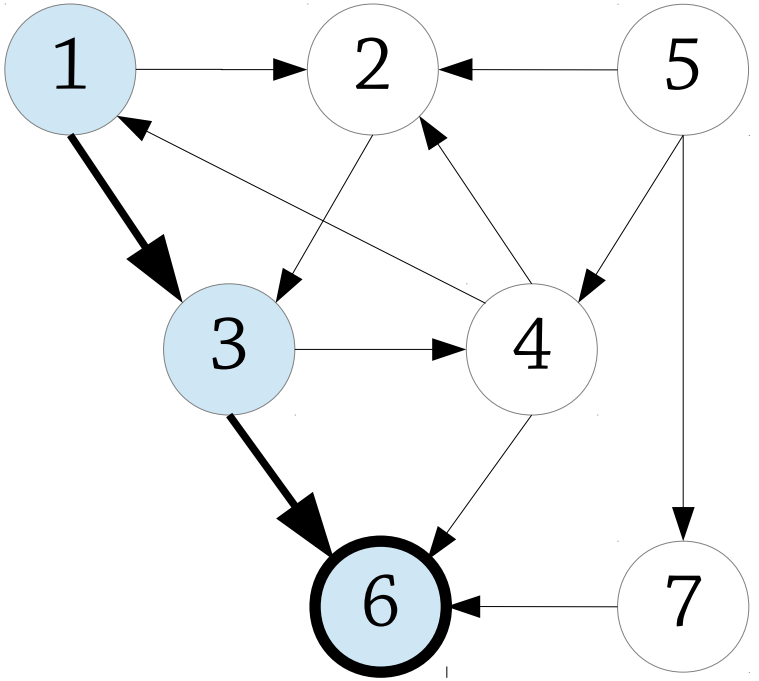


= current = unvisited = visited

Example of a depth-first search

Visit order: 1 3 6

Follow edge 3 → 6,
recursively DFS node 6

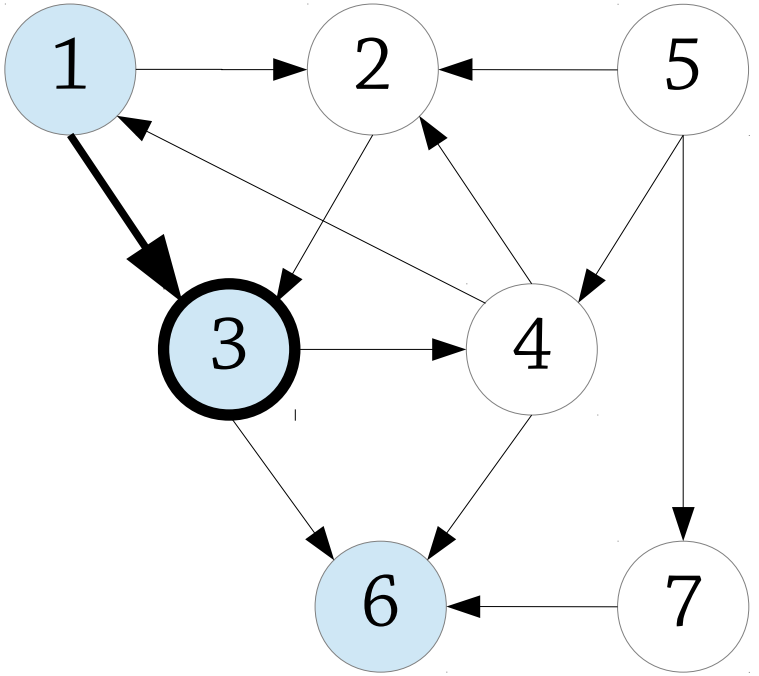


= current = unvisited = visited

Example of a depth-first search

Visit order: 1 3 6

Recursion backtracks to 3

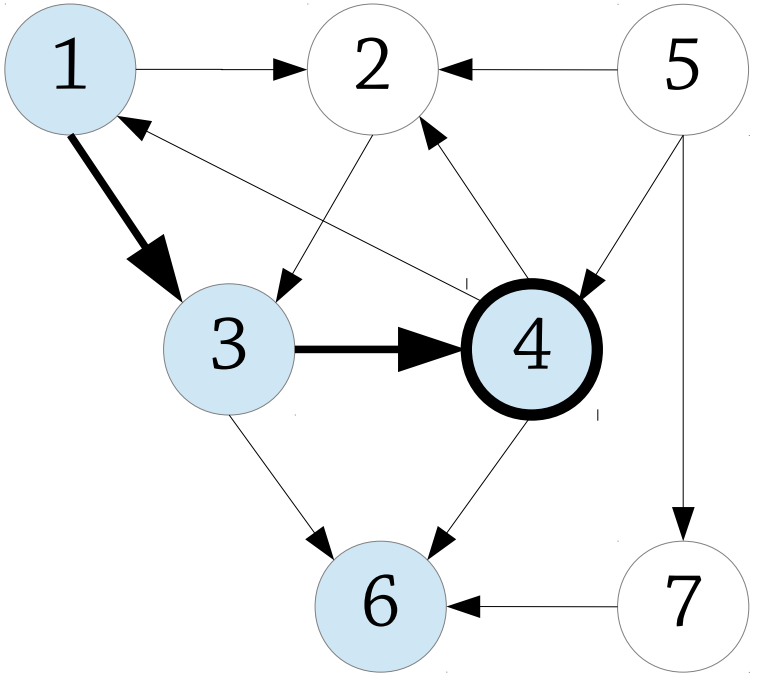


= current = unvisited = visited

Example of a depth-first search

Visit order: 1 3 6 4

Follow edge 3 → 4,
recursively DFS node 4



= current = unvisited = visited

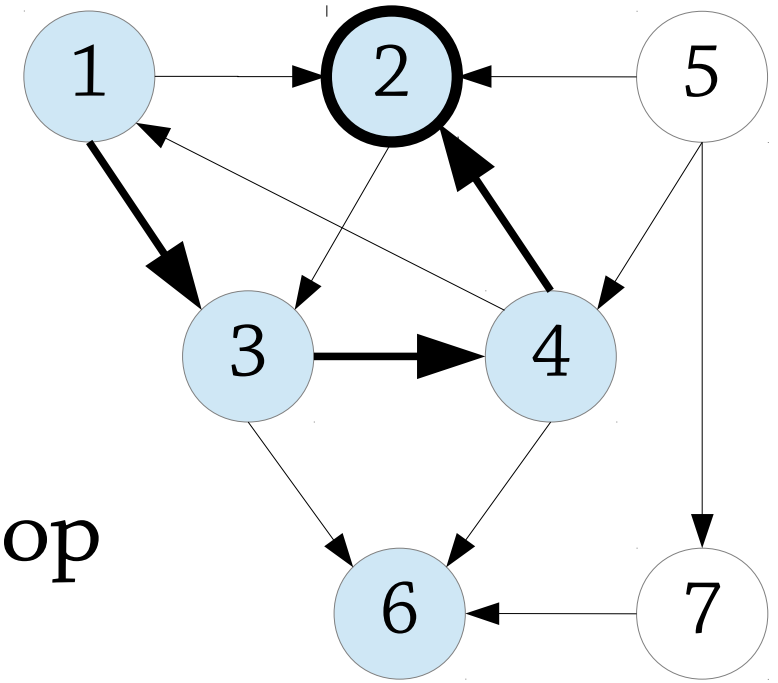
Example of a depth-first search

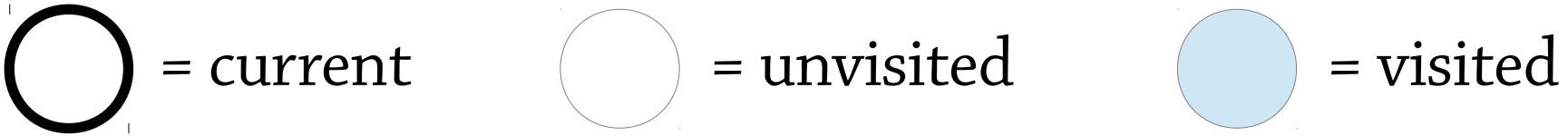
Visit order: 1 3 6 4 2

Follow edge 4 → 2, recursively DFS node 2

We don't follow 4 → 6 or 2 → 3, as those nodes have already been visited

Eventually the recursion backtracks to 1 and we stop





Reachability revisited

How can we tell what nodes are reachable from a given node?

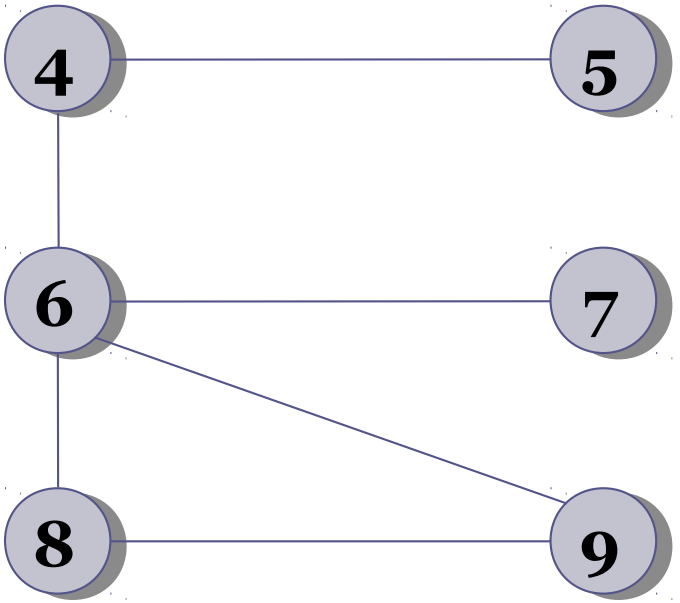
Answer:

Perform a depth-first search starting from node A, and the nodes visited by the DFS are exactly the reachable nodes

Connectedness

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

This graph is connected

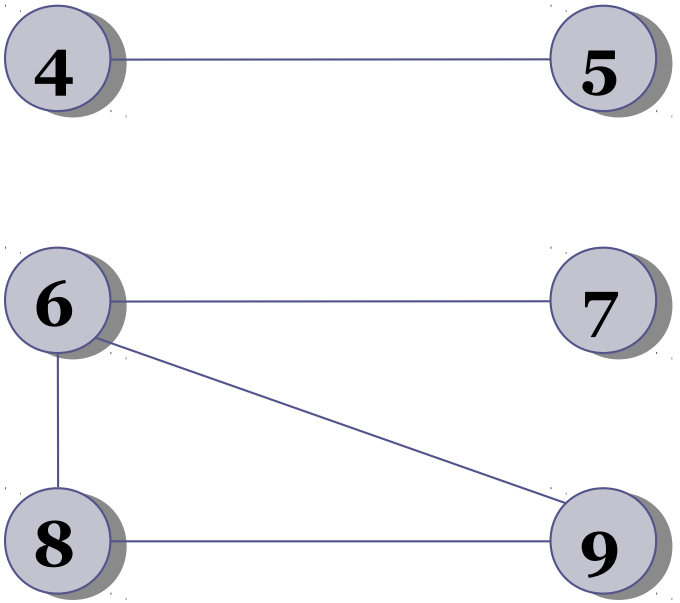


How can we tell if a graph is connected?

Connectedness

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

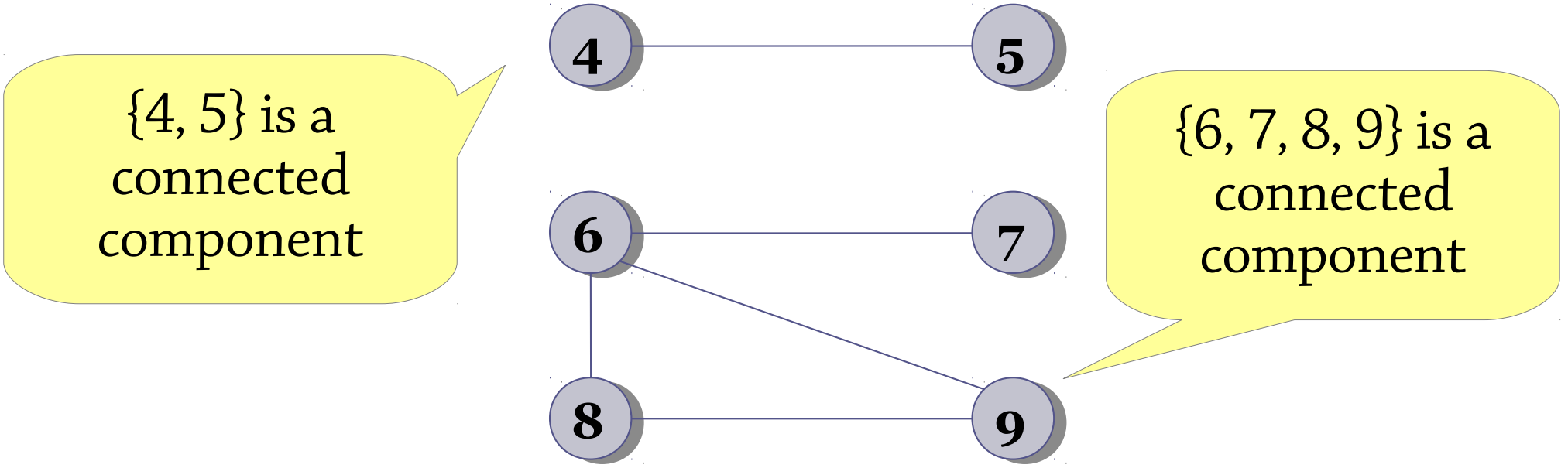
This graph is **not** connected



How can we tell if a graph is connected?

Connectedness

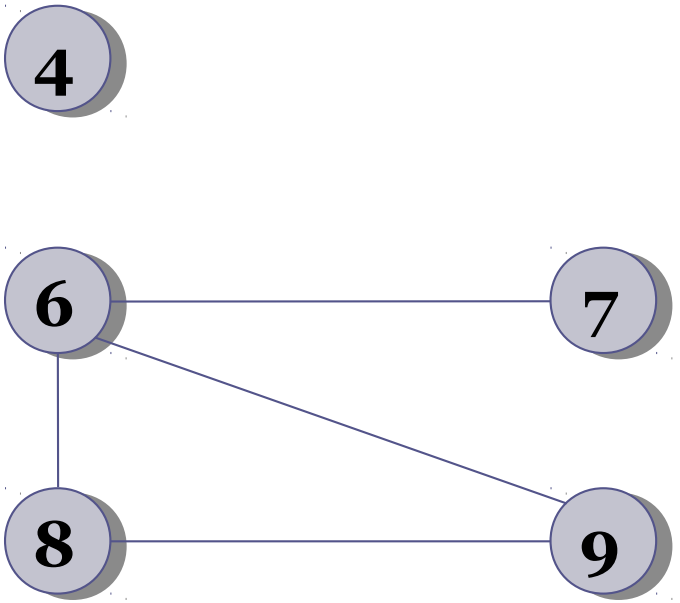
If an undirected graph is unconnected, it still consists of *connected components*



Connectedness

A single unconnected node is a connected component in itself

{4} is a connected component



Connected components

How can we find:

- the connected component containing a given node?
- all connected components in the graph?

Connected components

To find the connected component containing a given node:

- Perform a DFS starting from that node
- The set of visited nodes is the connected component

To find all connected components:

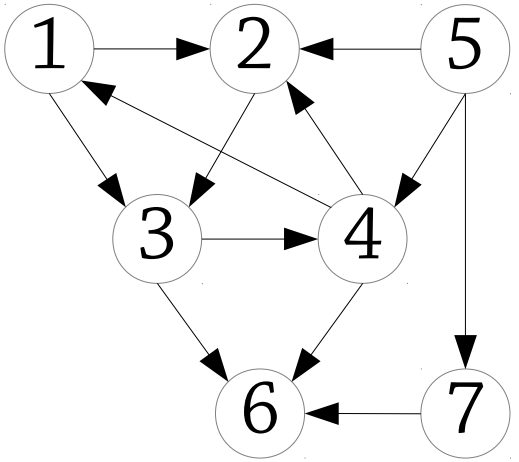
- Pick a node that doesn't have a connected component yet
- Use the algorithm above to find its connected component
- Repeat until all nodes are in a connected component

Strongly-connected components

In a directed graph, there are two notions of connectedness:

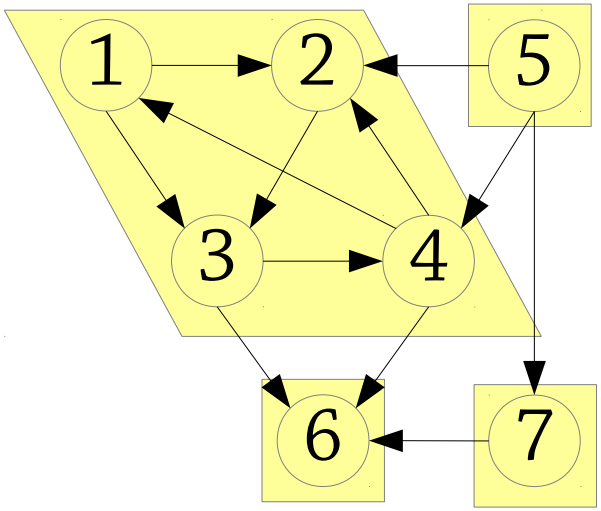
- *strongly connected* means there is a path from every node to every other node
- *weakly connected* means the graph is connected if you ignore the direction of the edges
  (the equivalent undirected graph is connected)

This graph is weakly connected, but not strongly connected (why?)



Strongly-connected components

You can always divide a directed graph into its *strongly-connected components (SCCs)*:



In each strongly-connected component, every node is reachable from every other node

- The relation "nodes A and B are both reachable from each other" is an *equivalence relation* on nodes
- The SCCs are the equivalence classes of this relation

Strongly-connected components

To find the SCC of a node A, we take the intersection of:

- the set of nodes reachable from A
- the set of nodes which A can be reached from (the set of nodes "backwards-reachable" from A)

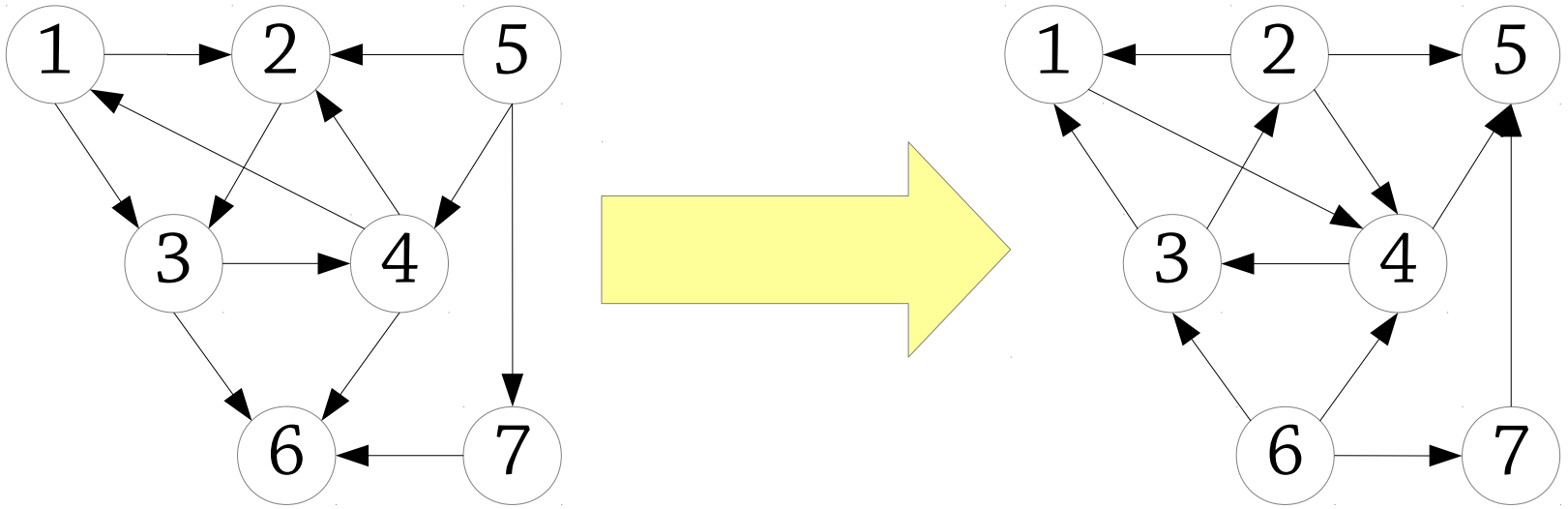
This gives us all the nodes B such that:

- there is a path from A to B, and
- there is a path from B to A

To find the set of nodes backwards-reachable from A, we will use the idea of the *transpose* of a graph

Transpose of a graph

To find the transpose of a directed graph, flip the direction of all the graph's edges:



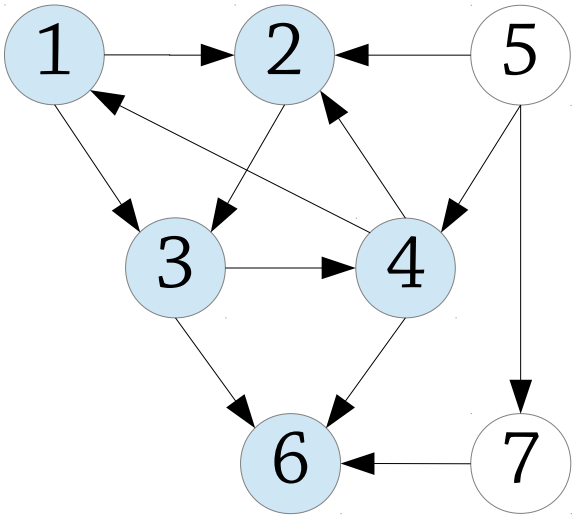
Graph

Transpose

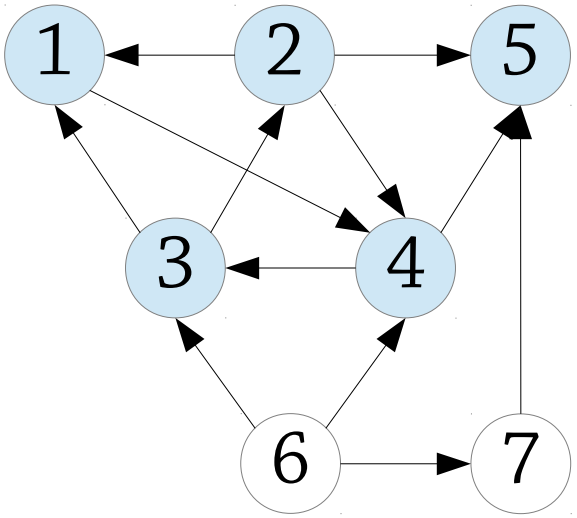
Note that: there is a path from A to B in the original graph iff there is a path from B to A in the transpose graph!

Strongly-connected components

To find the SCC of a node (such as 2), perform a DFS in the graph and the transpose graph:



Graph



Transpose

The nodes visited in both DFSs are the SCC – in this case {1, 2, 3, 4}

Strongly-connected components

To find the SCC of a node A:

- Find the set of nodes reachable from A, using DFS
- Find the set of nodes which have a path to A, by doing a DFS in the *transpose* graph
- Take the intersection of these two sets

Implementation in practice:

- When doing the DFS in the transpose graph, we restrict the search to the nodes that were reachable from A in the original graph

What do SCCs mean?

The SCCs in a graph tell you about the *cycles* in that graph!

- If a graph has a cycle, all the nodes in the cycle will be in the same SCC
- If an SCC contains two nodes A and B, there is a path from A to B and back again, so there is a cycle

A directed graph is acyclic iff:

- All the SCCs have size 1, and
- no node has an edge to itself (SCCs do not take any notice of self-loops)

If the SCCs are collapsed to single nodes, the resulting graph is a DAG.

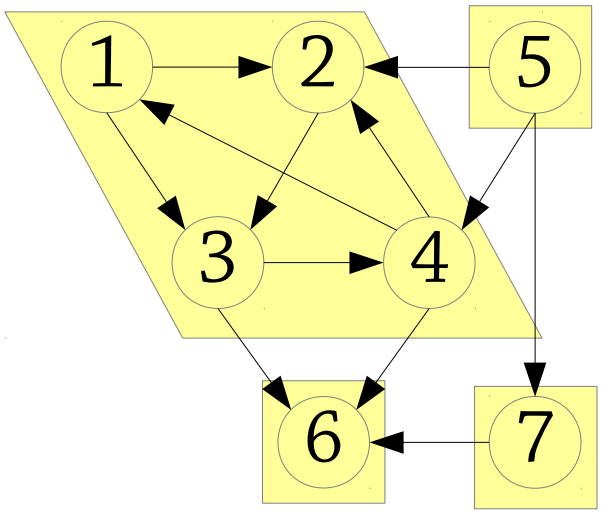
Cycles and SCCs

Here is the directed graph from before.

Notice that:

- The big SCC is where all the cycles are
- The acyclic "parts" of the graph have SCCs of size 1

The SCCs characterise the cycles in the graph!

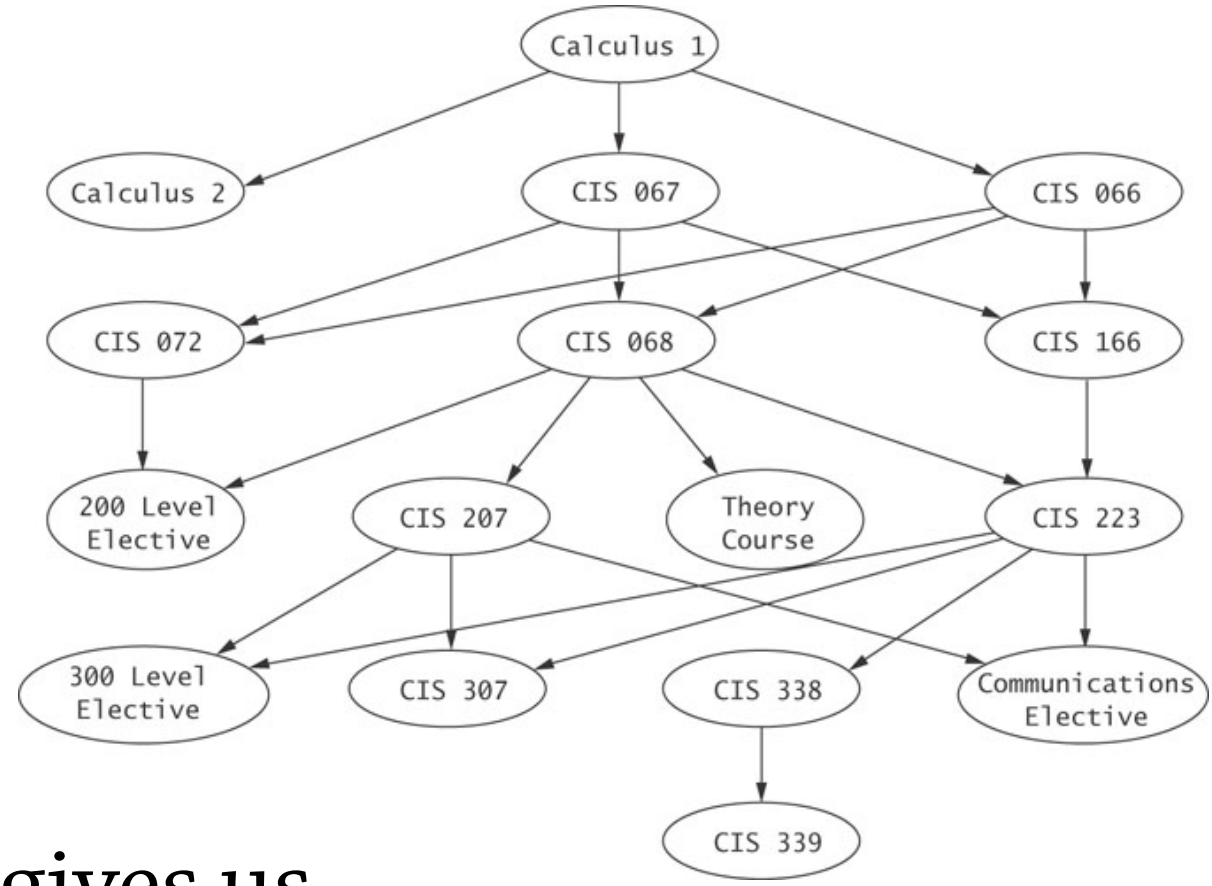


Graph algorithms:
postorder DFS,
detecting cycles,
topological sorting

Topological sorting

Here is a DAG with courses and prerequisites:

We might want to find out: what is a possible order to take these courses in?



This is what *topological sorting* gives us.

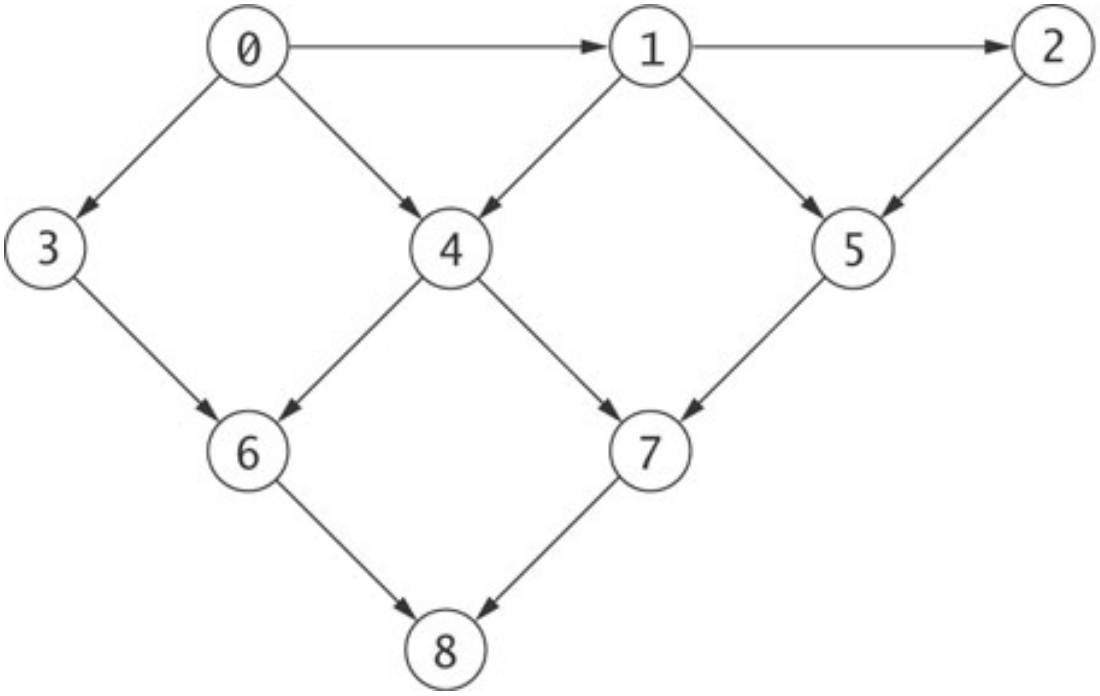
Note that the graph must be acyclic!

Example: topological sort

A topological sort of the nodes in a DAG is a list of all the nodes, so that *if there is a path from u to v, 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.



Postorder depth-first search

One way to implement topological sorting is to use a variant of DFS called *postorder* depth-first search

To do a postorder DFS starting from a node:

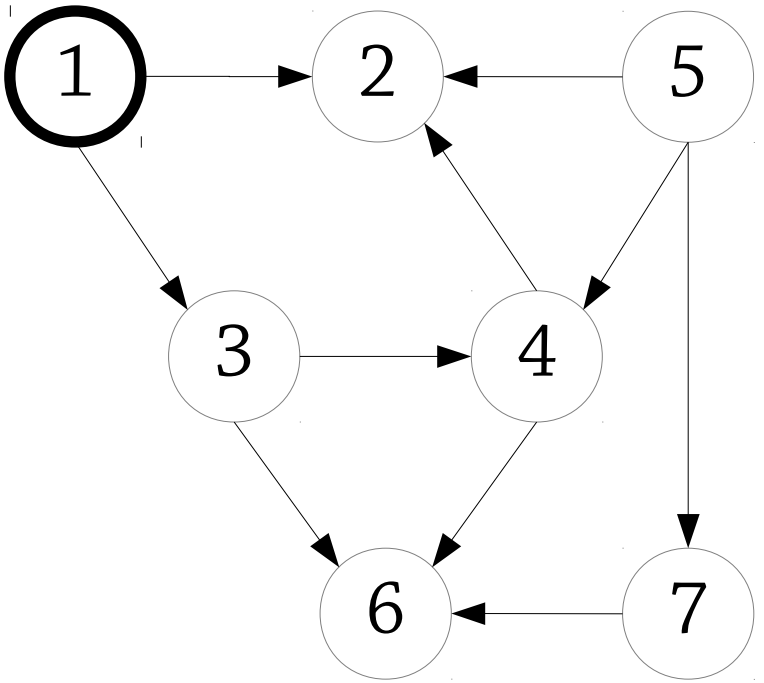
- mark the node as reached
- for each outgoing edge from the node, recursively DFS the target of that edge, unless it has already been reached
- visit the node

In postorder DFS, we visit each node *after* we visit its outgoing edges!

Postorder depth-first search

Visit order:

DFS node 1 (don't visit it yet, but remember that we
have reached it)

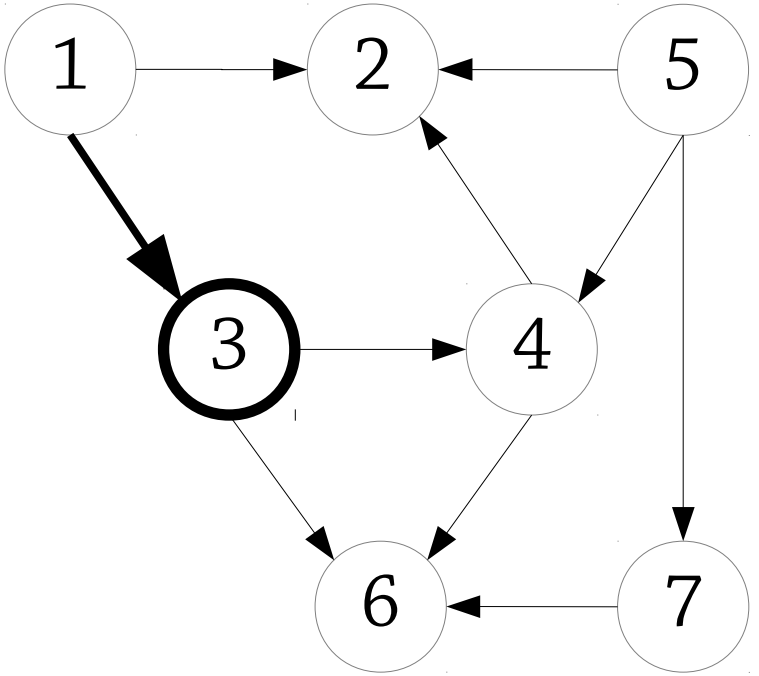


= current = unvisited = visited

Postorder depth-first search

Visit order:

Follow edge 1 → 3,
recursively DFS node 3



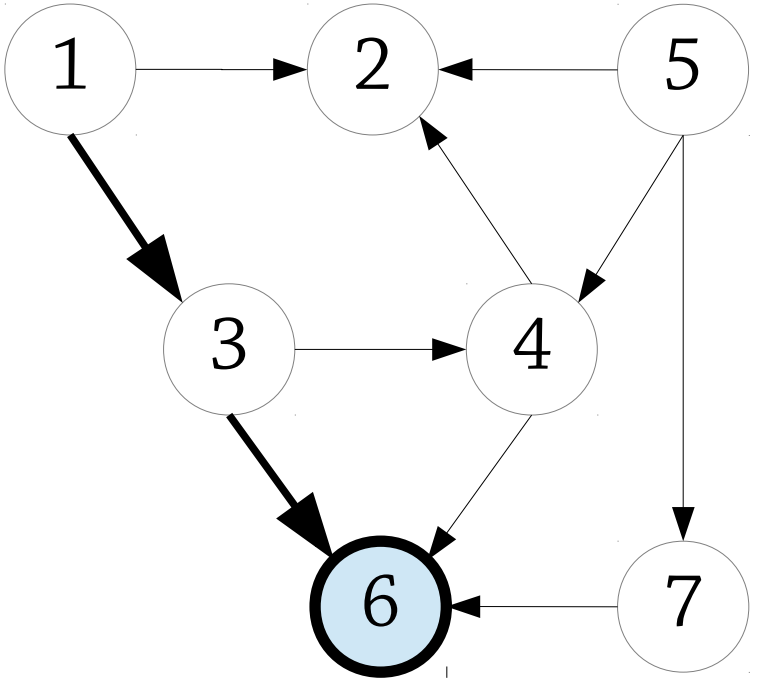
= current = unvisited = visited

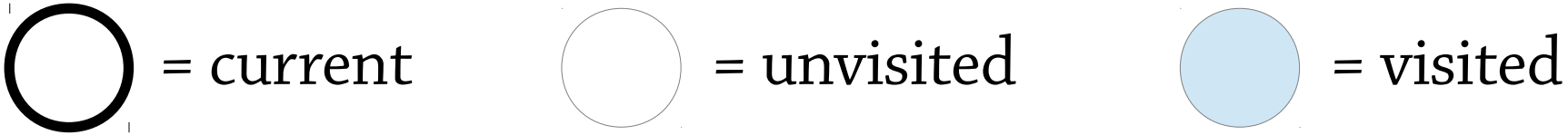
Postorder depth-first search

Visit order: 6

Follow edge 3 → 6, recursively DFS node 6

The recursion bottoms out, visit 6!

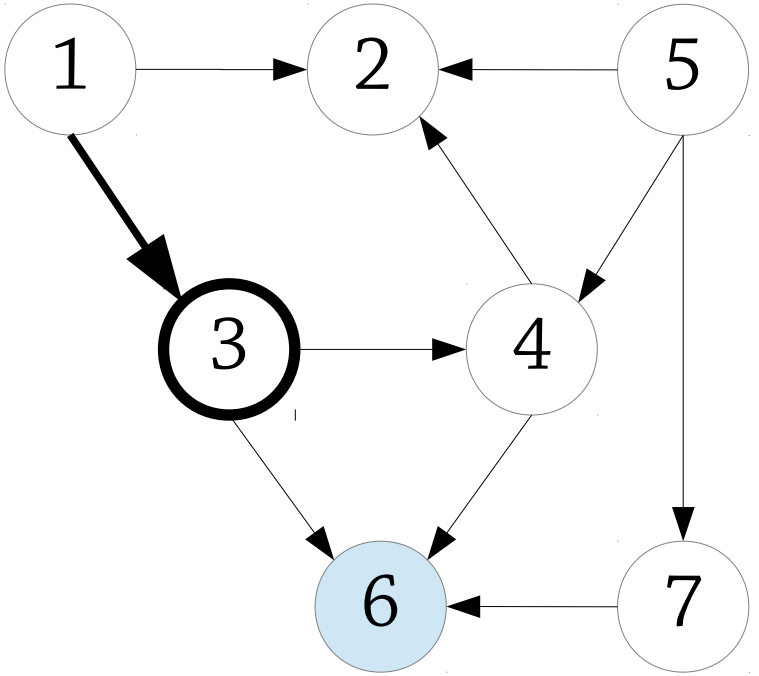




Postorder depth-first search

Visit order: 6

Recursion backtracks to 3



= current

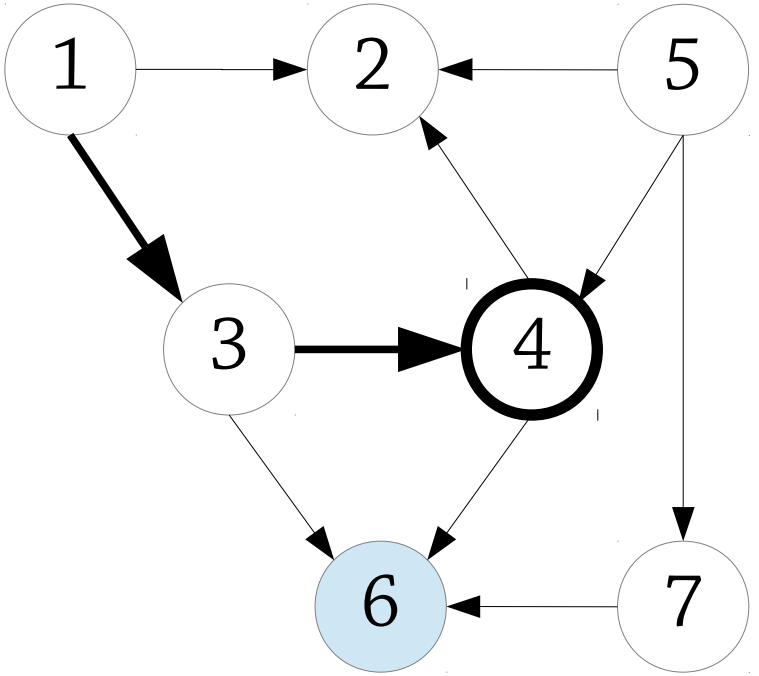
= unvisited

= visited

Postorder depth-first search

Visit order: 6

Follow edge 3 → 4,
recursively DFS node 4



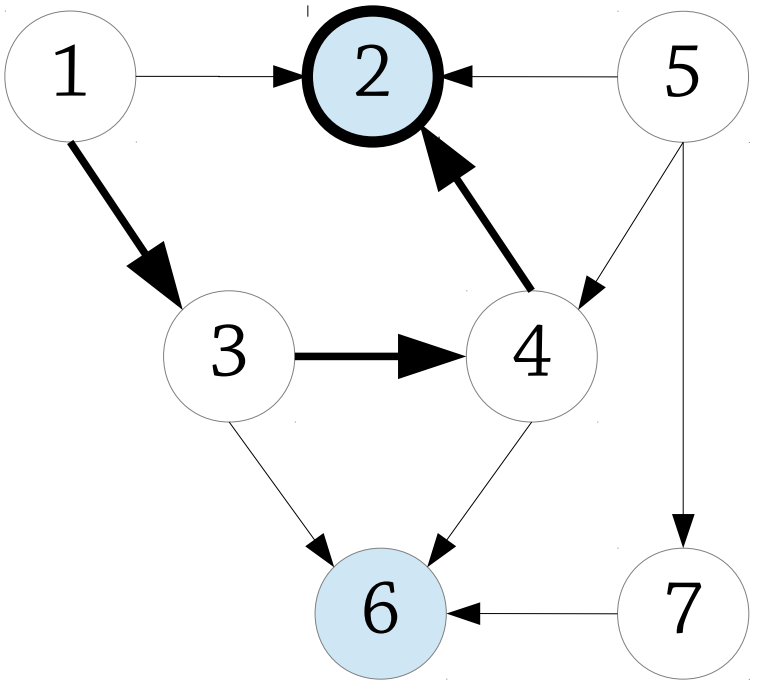
= current = unvisited = visited

Postorder depth-first search

Visit order: 6 2

Follow edge 4 → 2, recursively DFS node 2

The recursion bottoms out again and we visit 2

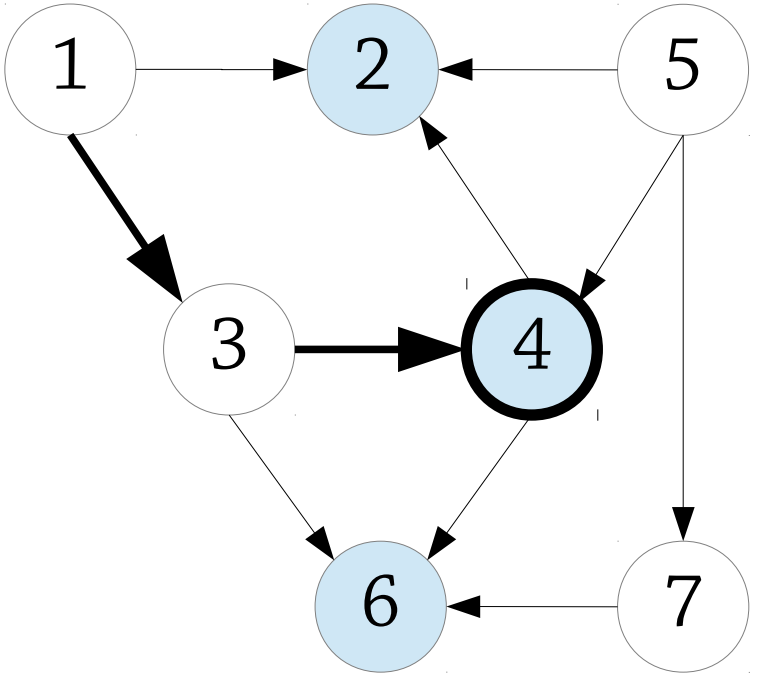


= current = unvisited = visited

Postorder depth-first search

Visit order: 6 2 4

The recursion backtracks and
now we visit 4

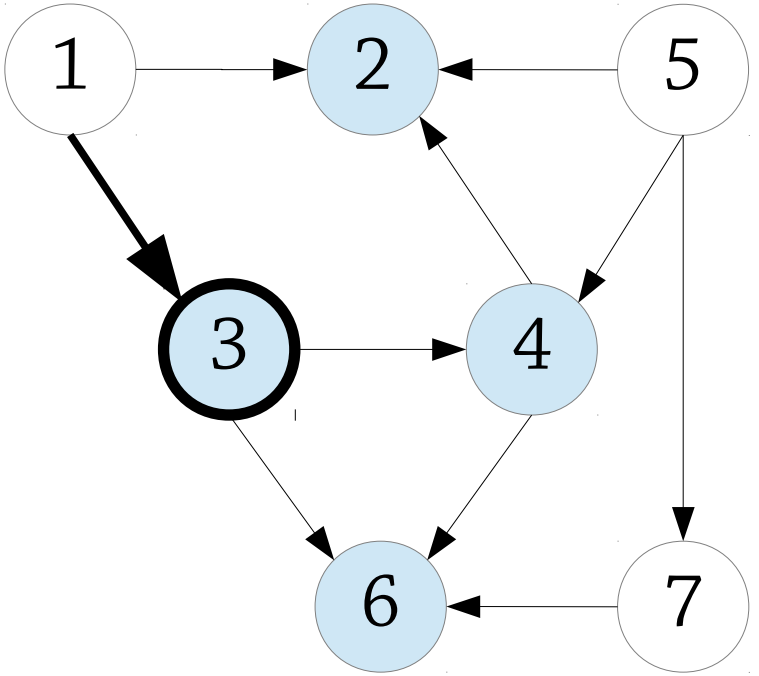


= current = unvisited = visited

Postorder depth-first search

Visit order: 6 2 4 3

The recursion backtracks and
now we visit 3



= current

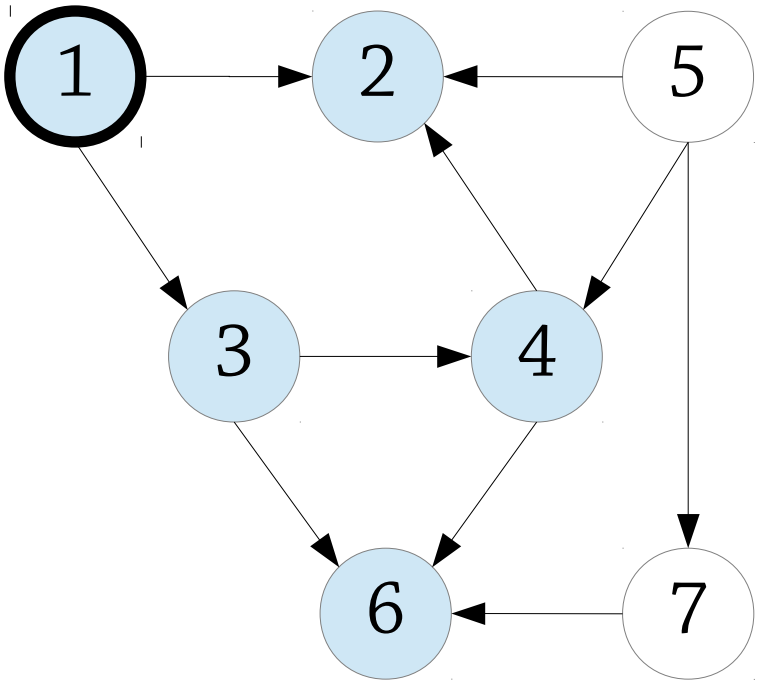
= unvisited

= visited

Postorder depth-first search

Visit order: 6 2 4 3 1

The recursion backtracks and
now we visit 1



= current = unvisited = visited

Why postorder DFS?

In postorder DFS:

- We only visit a node *after* we recursively DFS its successors (the nodes it has an edge to)

If we look at the order the nodes are visited (rather than the calls to DFS):

- If the graph is acyclic, we visit a node only after we have visited all its successors

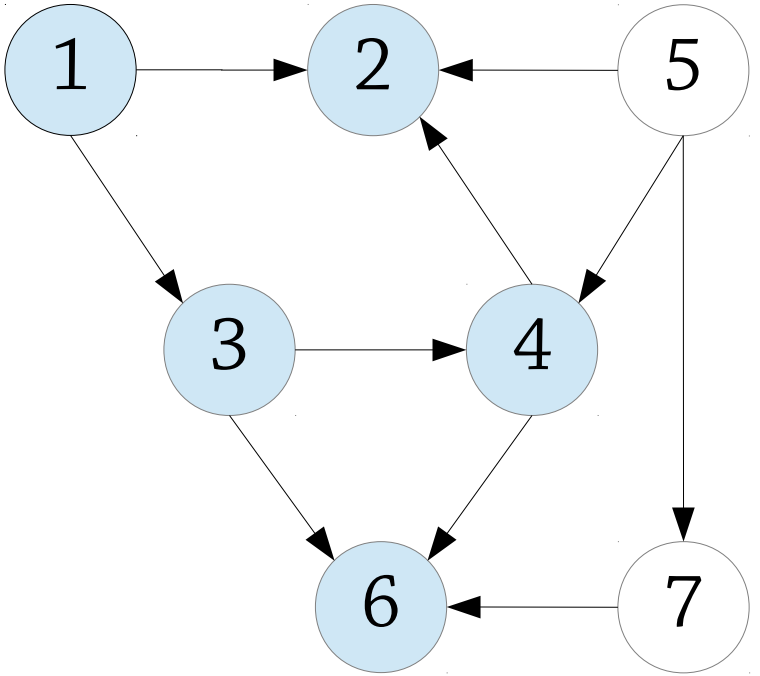
If we look at the list of nodes in the order they are visited, each node comes after all its successors (look at the previous slide)

Topological sorting

Visit order: 6 2 4 3 1

In topological sorting, we want each node to come *before* its successors...

With postorder DFS,
each node is visited
after its successors!



Idea: to topologically sort,
do a postorder DFS,
look at the order the nodes
are visited in and *reverse* it

Small problem: not all nodes are visited!
Solution: pick a node we haven't visited and DFS it

Topological sorting

To topologically sort a DAG:

- Pick a node that we haven't visited yet
- Do a postorder DFS on it
- Repeat until all nodes have been visited

Then take the list of nodes in the order they were visited, and reverse it

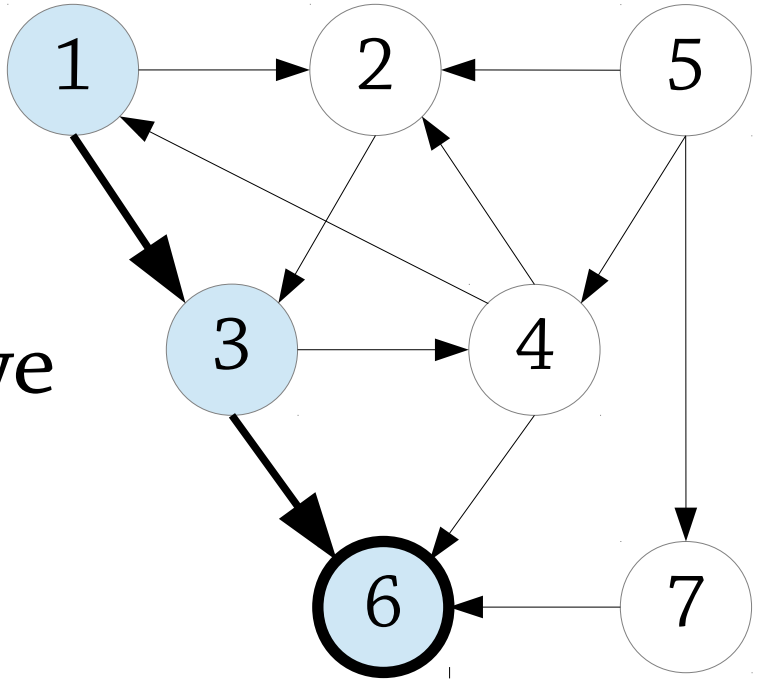
If the graph is acyclic, the list is topologically sorted:

- If there is a path from node A to B, then A comes before B in the list

Preorder vs postorder

You might think that in preorder DFS, we visit each node *before* we visit its successsors

But this is not the case, in this example from earlier we visited 6 before its predecessor 4, because we happened to go through 3



Postorder DFS is more well-behaved in this sense.

Detecting cycles in graphs

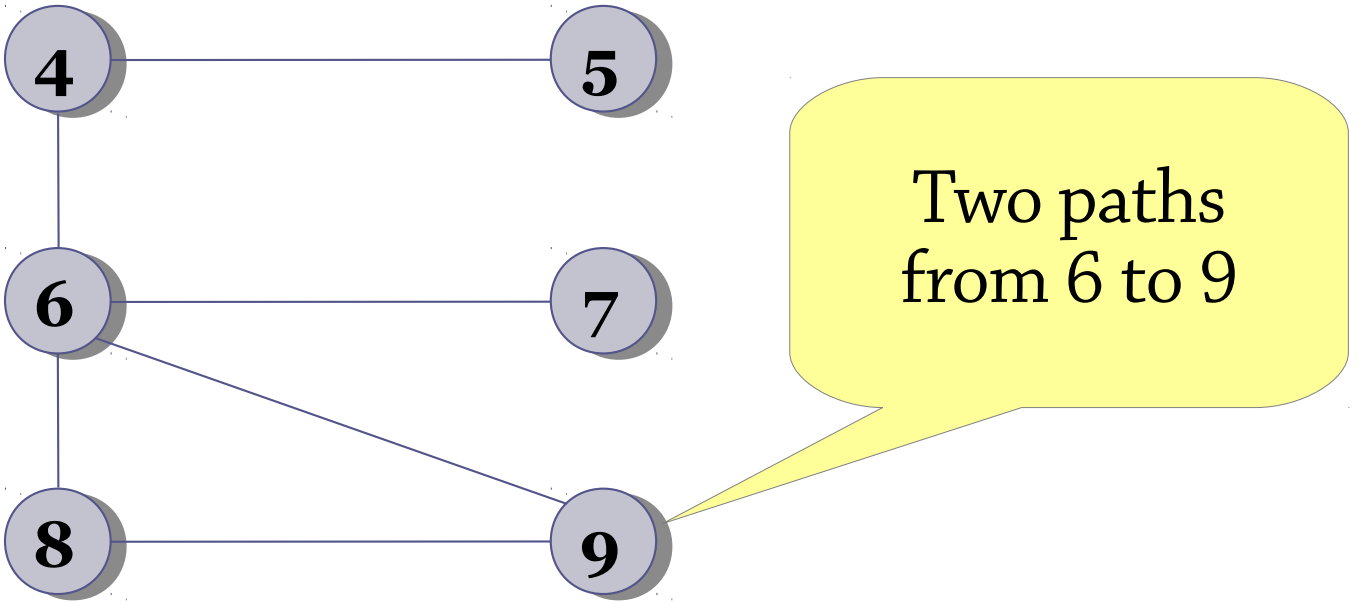
We can only topologically sort *acyclic* graphs – how can we detect if a graph is cyclic?

Easiest answer: topologically sort the graph and check if the result is actually topologically sorted

- Does any node in the result list have an edge to a node *earlier* in the list? If so, the topological sorting failed, and the graph must be cyclic
- Otherwise, the graph is acyclic

Cycles in undirected graphs

An undirected graph has a cycle if there are two different paths between two nodes:



You can join the two paths to get a cycle!

Detecting cycles in undirected graphs

To check if an undirected graph has a cycle:

- Pick a node
- Do a DFS starting from that node, but…
- …if we ever reach a node that has already been visited, stop: the graph has a cycle because there are two paths to the node
  (normal DFS would skip the node)
- Repeat for each connected component

Summary

Graphs are extremely useful!

- Common representation: adjacency lists (or just implicitly as references between the objects in your program)

Several important graph algorithms:

- Reachability – can I get from node A to B?
- Does the graph have a cycle?
- Strongly-connected components – where are the cycles in the graph?
- Topological sorting – how can I order the nodes in an acyclic graph?

All based on depth-first search!

- Enumerate the nodes reachable from a starting node
- Preorder: visit each node before its successors
- Postorder: visit each node after its successors, gives nicer order
- Common pattern in these algorithms: repeat DFS from different nodes until all nodes have been visited