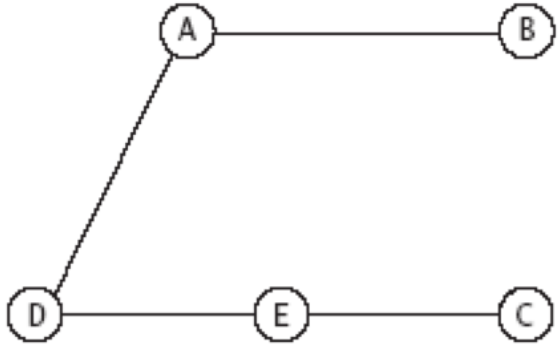
Lecture 6
Data Structures (DAT037)

Ramona Enache
(with slides from Nick Smallbone)

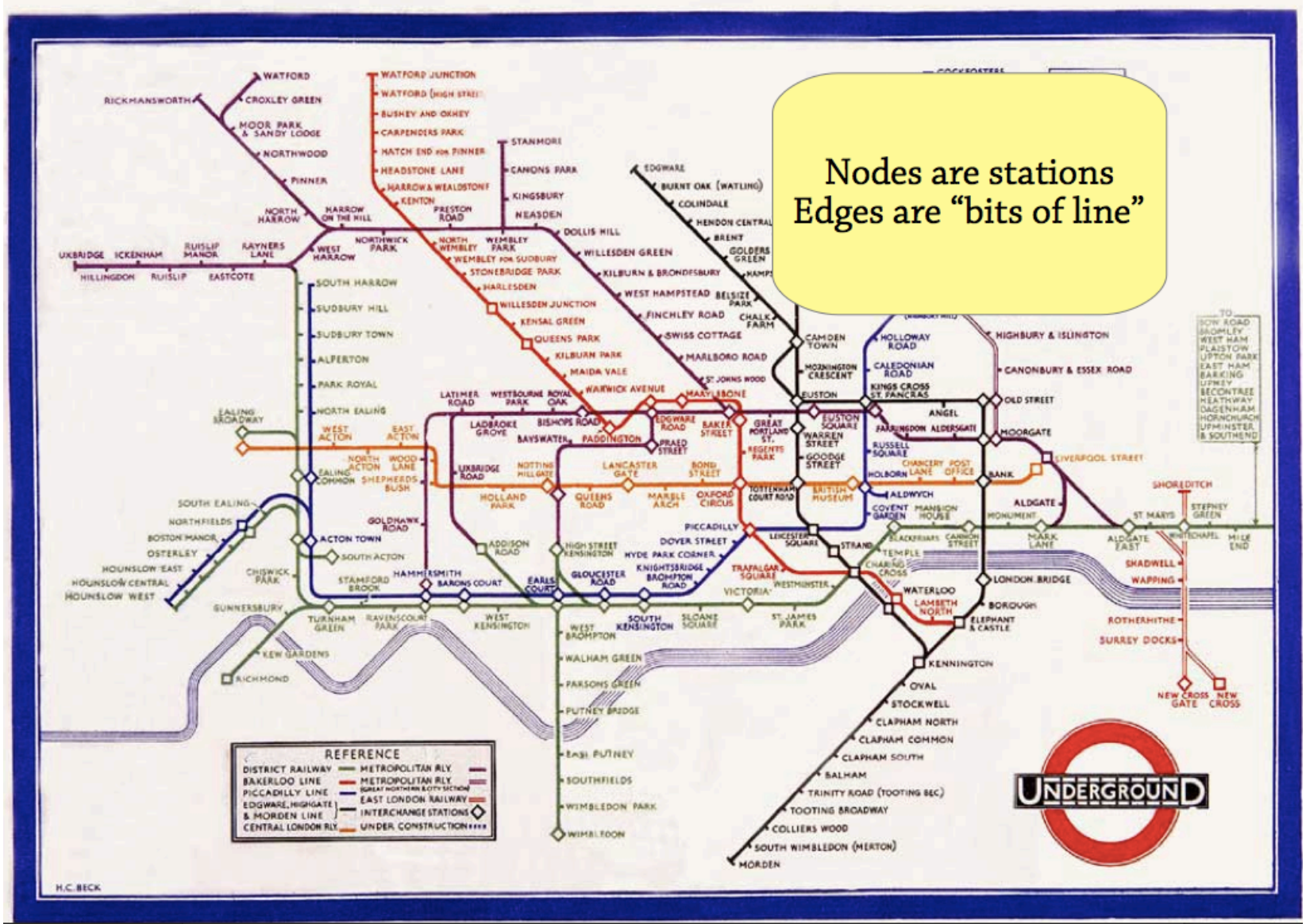
Graphs

A **graph** is a data structure consisting of *nodes* (or vertices) and *edges*
An *edge* is a connection between two nodes

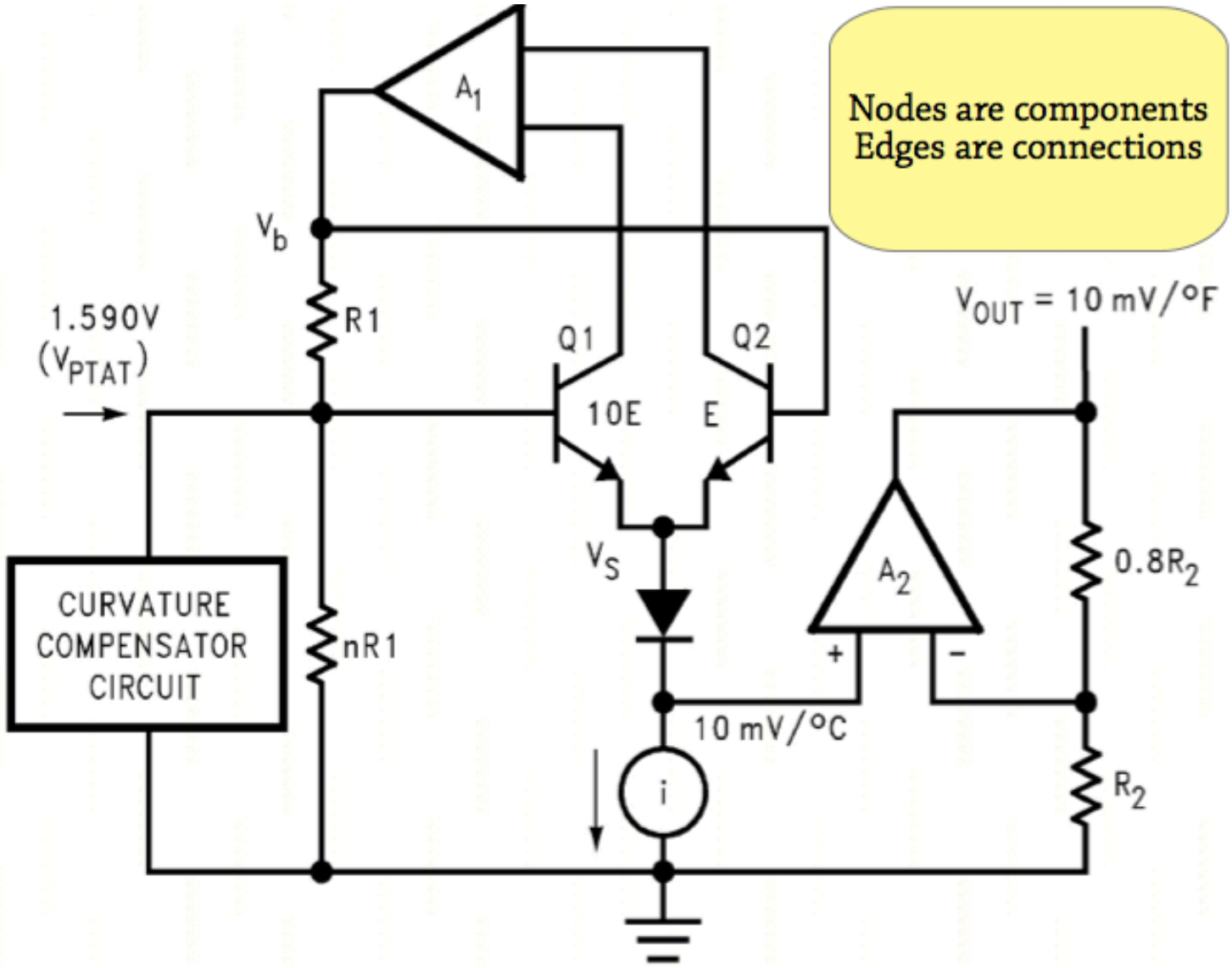


Nodes: A, B, C, D, E
Edges: (A, B), (A, D), (D, E), (E, C)

Graphs


Nodes are stations
Edges are "bits of line"
REFERENCE
DISTRICT RAILWAY
BAKERLOO LINE
PICCADILLY LINE
EDGWARE, HIGHGATE & MORDEN LINE
CENTRAL LONDON RLY
METROPOLITAN RLY
METROPOLITAN RLY (GREAT NORTHERN & CITY SECTION)
EAST LONDON RAILWAY
INTERCHANGE STATIONS
UNDER CONSTRUCTION
UNDERGROUND
H.C. BECK

Graphs


Nodes are components
Edges are connections
A1
Vb
R1
1.590V
(VPTAT)
Q1
Q2
10E
E
VOUT = 10 mV/°F
VS
A2
0.8R2
CURVATURE
COMPENSATOR
CIRCUIT
nR1
+
–
10 mV/°C
i
R2

More Graphs

- Graphs are used all over the place:
  + communications networks
  + social networks – Facebook, LinkedIn, Google+
  + maps, transport networks, route finding
  + finding good ways to lay out components in an integrated circuit
  + etc.
- Anywhere where you have entities, and relations between them!

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)

Graphs

Undirected graphs

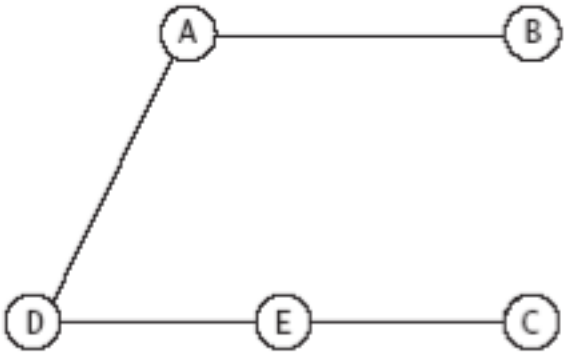
+ transportation maps
+ friendship graph in social networks

Directed graph

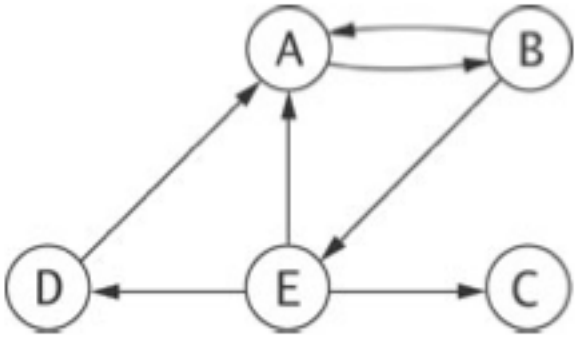
+ follower's graph in social networks
+ course prerequisite graph

Drawing Graphs

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



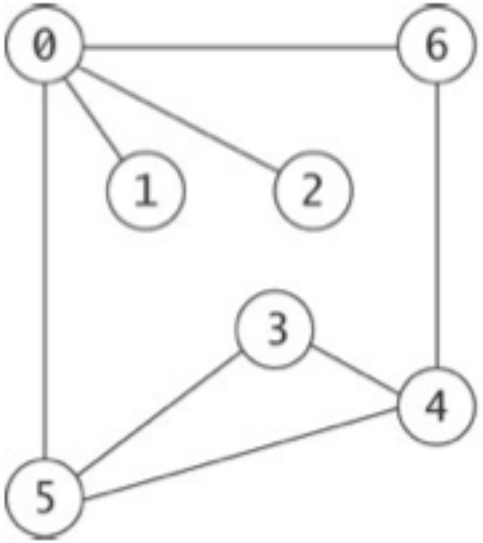
$$V = \{A, B, C, D, E\}$$
$$E = \{(A, B), (A, D), (C, E), (D, E)\}$$

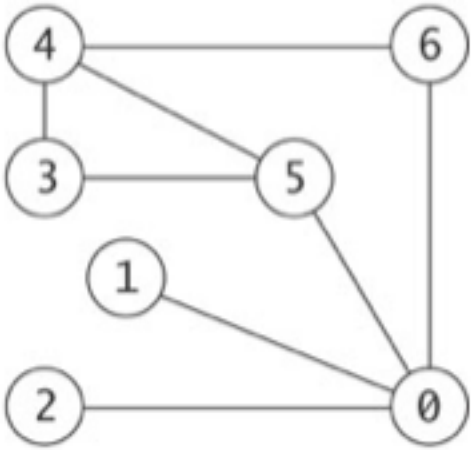


$$V = \{A, B, C, D, E\}$$
$$E = \{(A, B), (B, A), (B, E), (D, A), (E, A), (E, C), (E, D)\}$$

Drawing Graphs

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



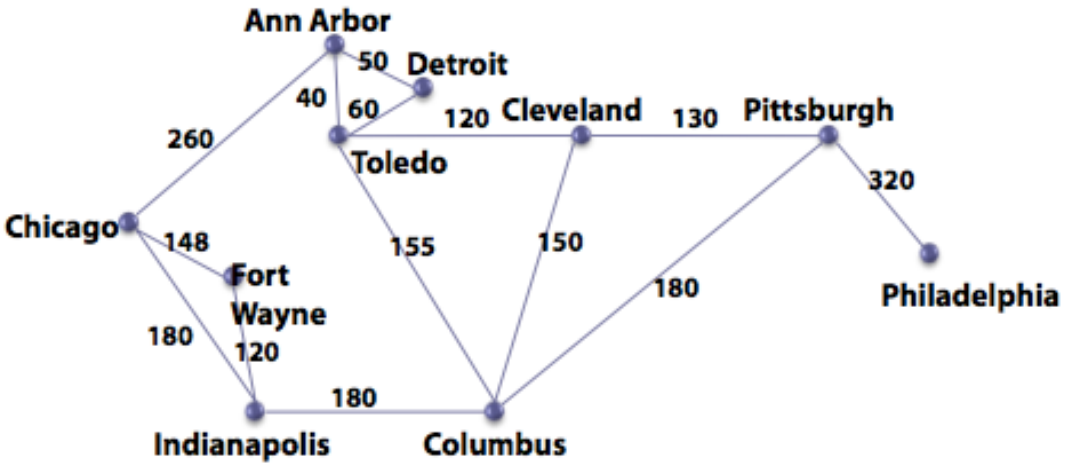


$V = \{0, 1, 2, 3, 4, 5, 6\}$

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

Weighted graphs

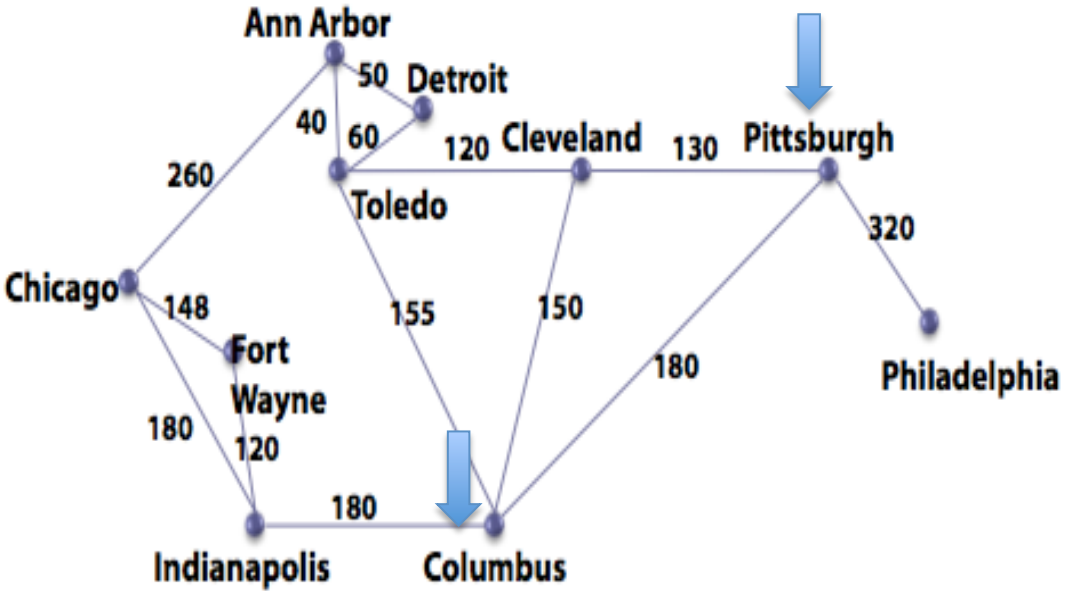
- In a weighted graph, each edge has a weight associated with it



- A graph can be directed, weighted, neither or both

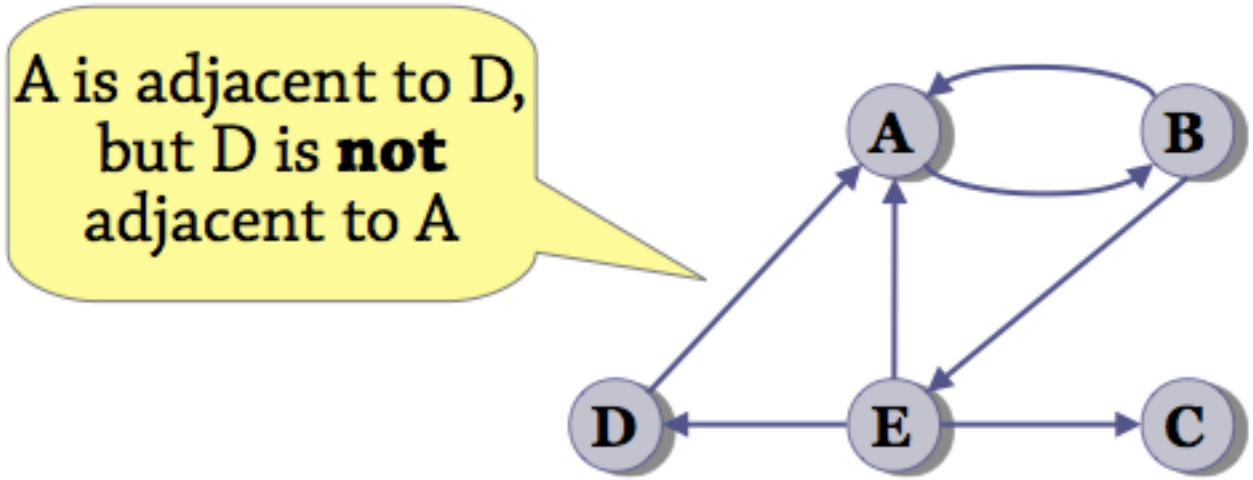
Paths and Cycles

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



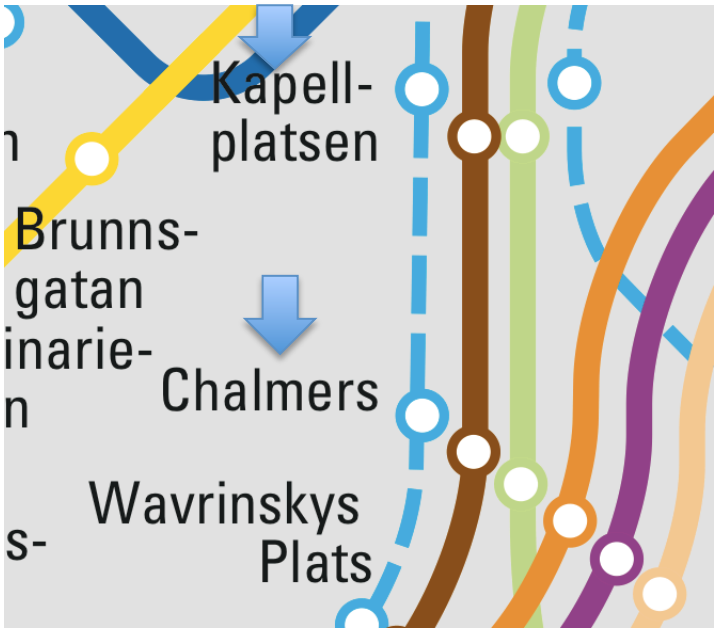
Paths and Cycles

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



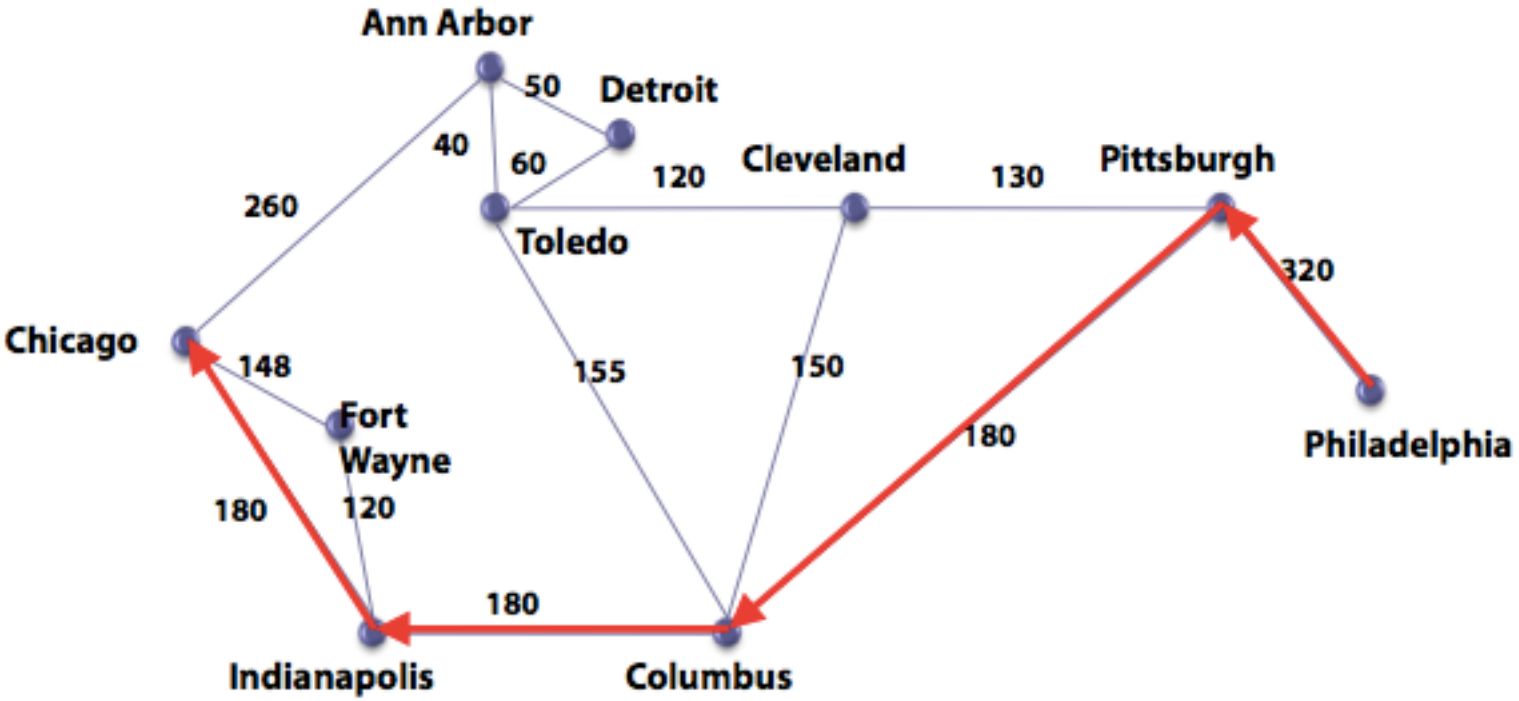
Paths and Cycles

- A **multigraph** has multiple edges between 2 nodes (directed/undirected/weighted)
- Here adjacent nodes have at least 1 edge between them



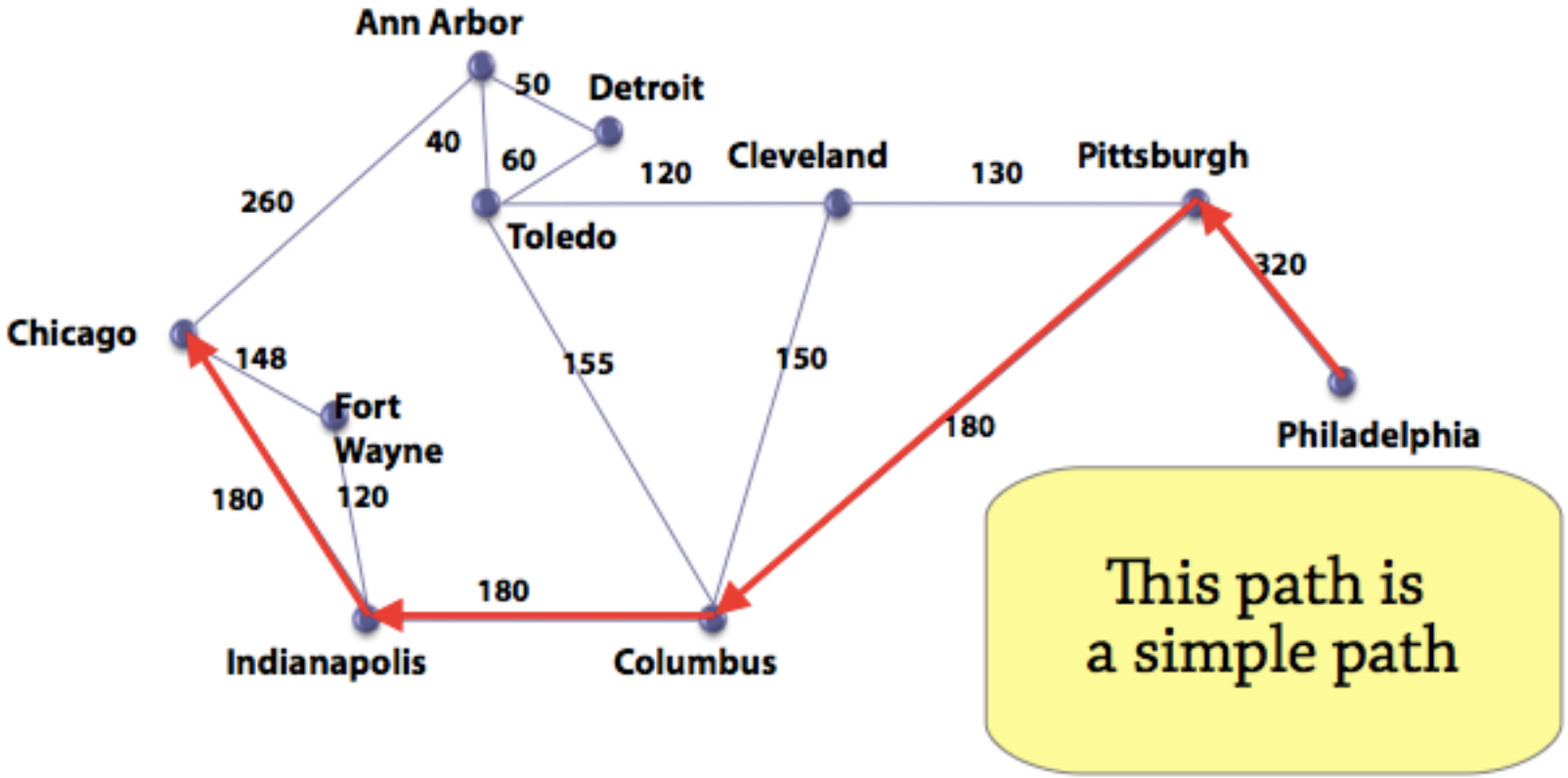
Paths and Cycles

- 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



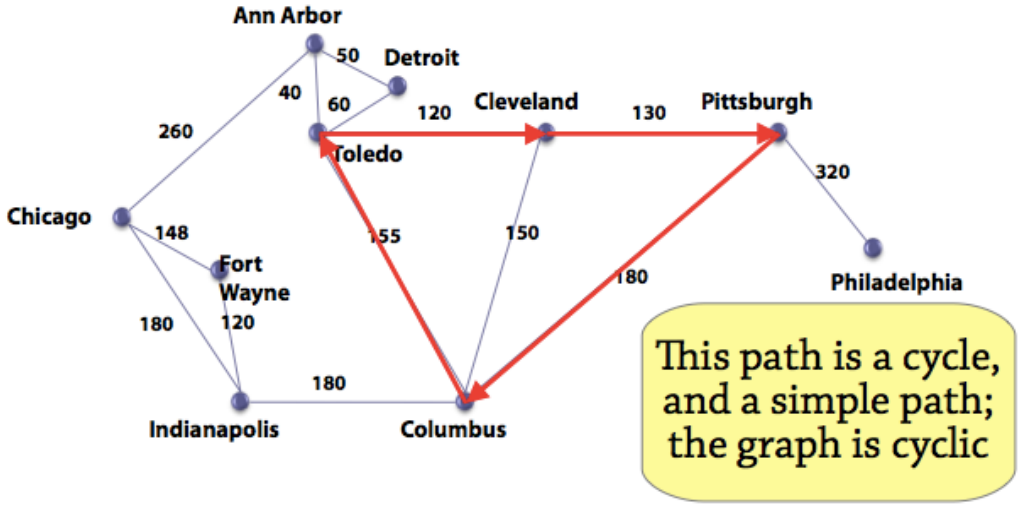
Paths and Cycles

- A **cycle** is a (non-trivial) simple path where the first and last

nodes are the same

! Two cycles are the same, if they contain the same edges **!**

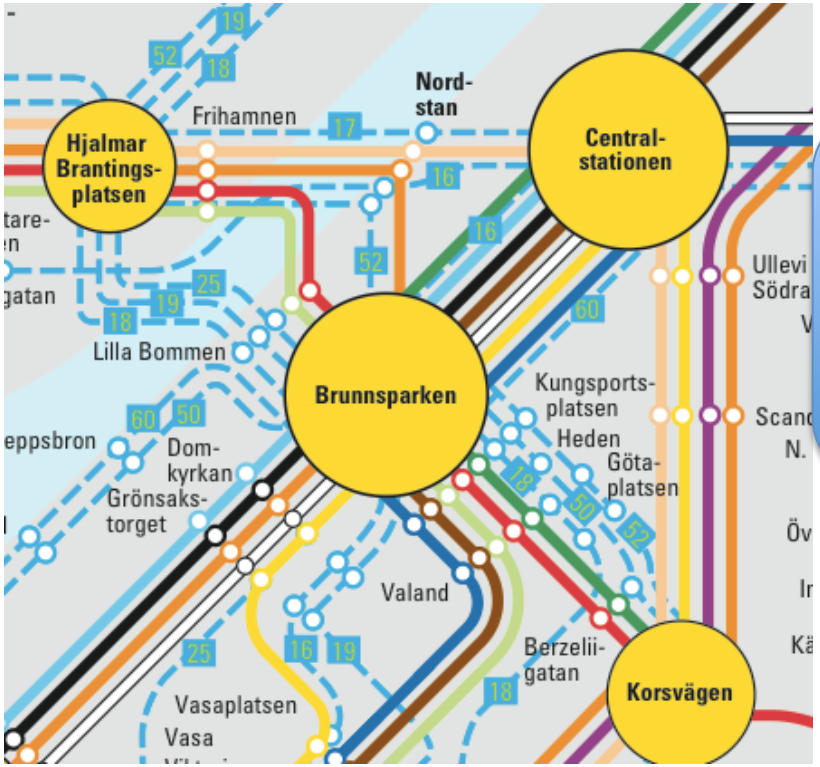
- A graph that contains a cycle is called cyclic, otherwise it is called acyclic



Question

- How many nodes are there in the smallest cycle that you can find in the following snapshot of the Västtrafik map ?

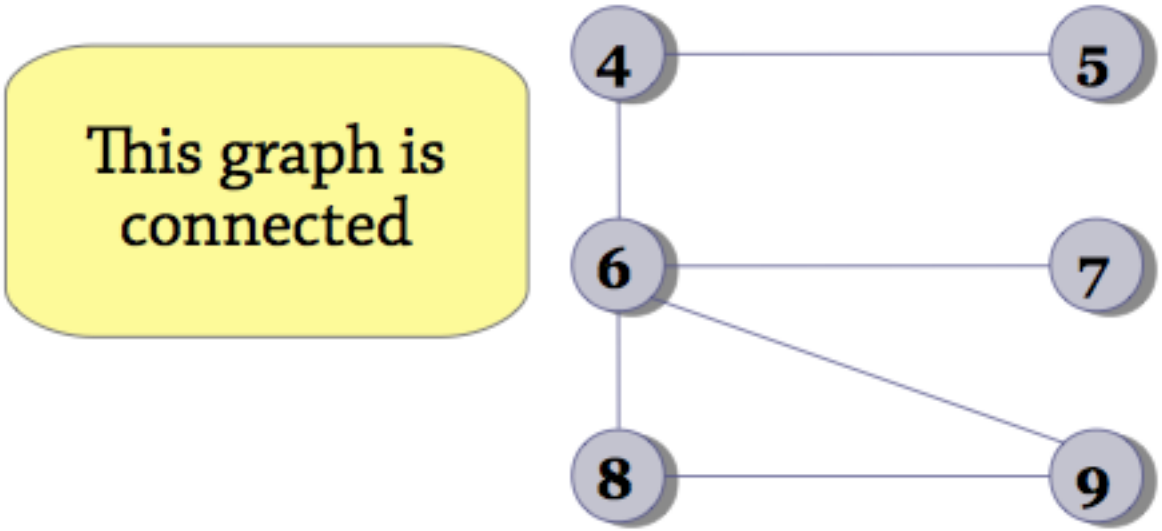
1. **1**
2. **2**
3. **3**
4. **4**



govote.at
Code 442748

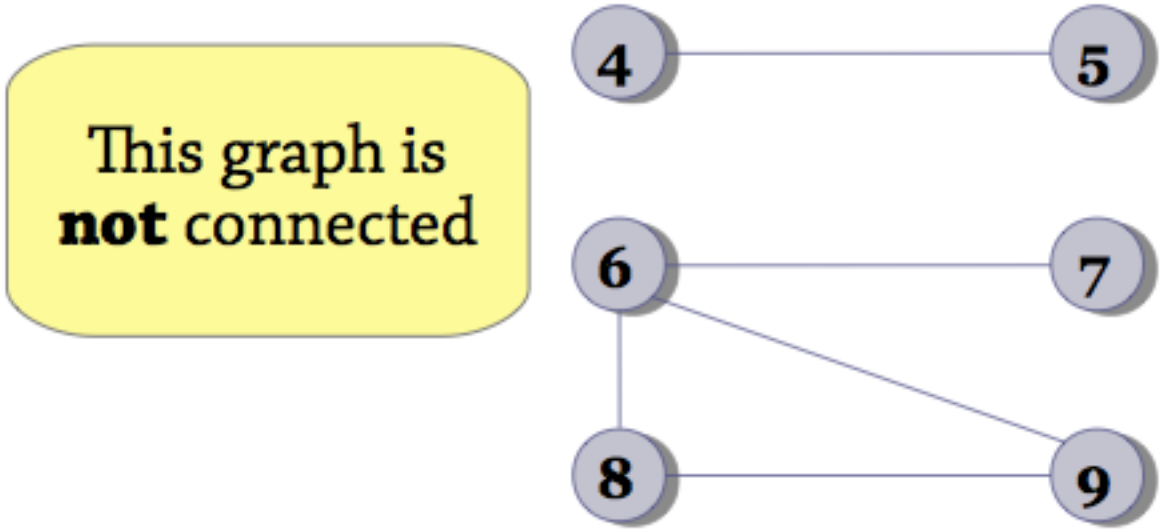
Paths and Cycles

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



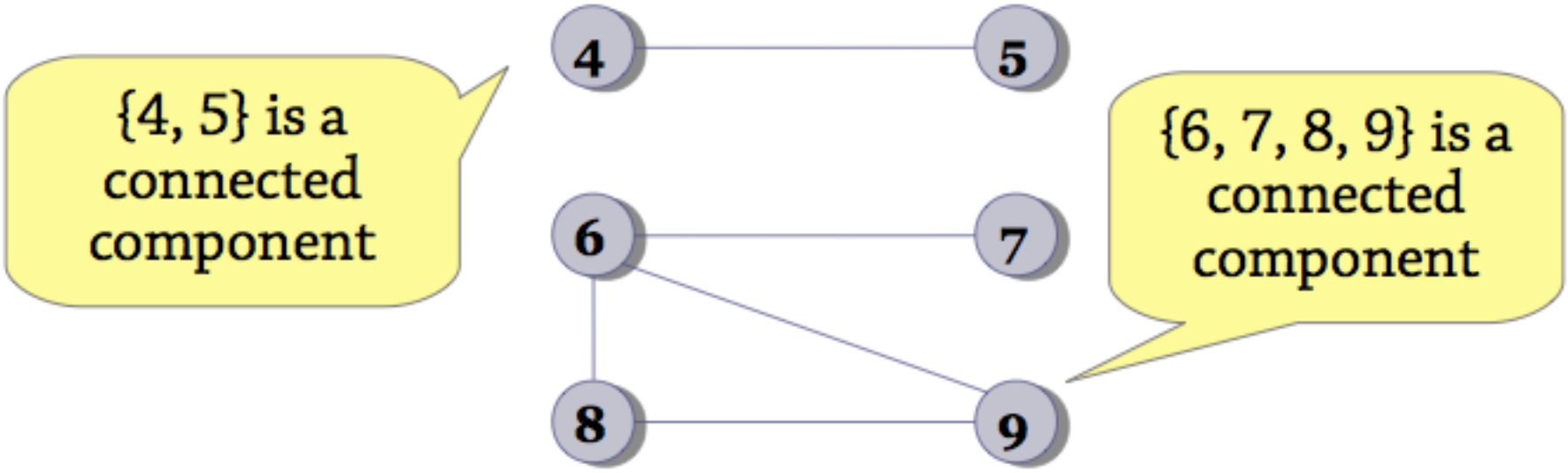
Paths and Cycles

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

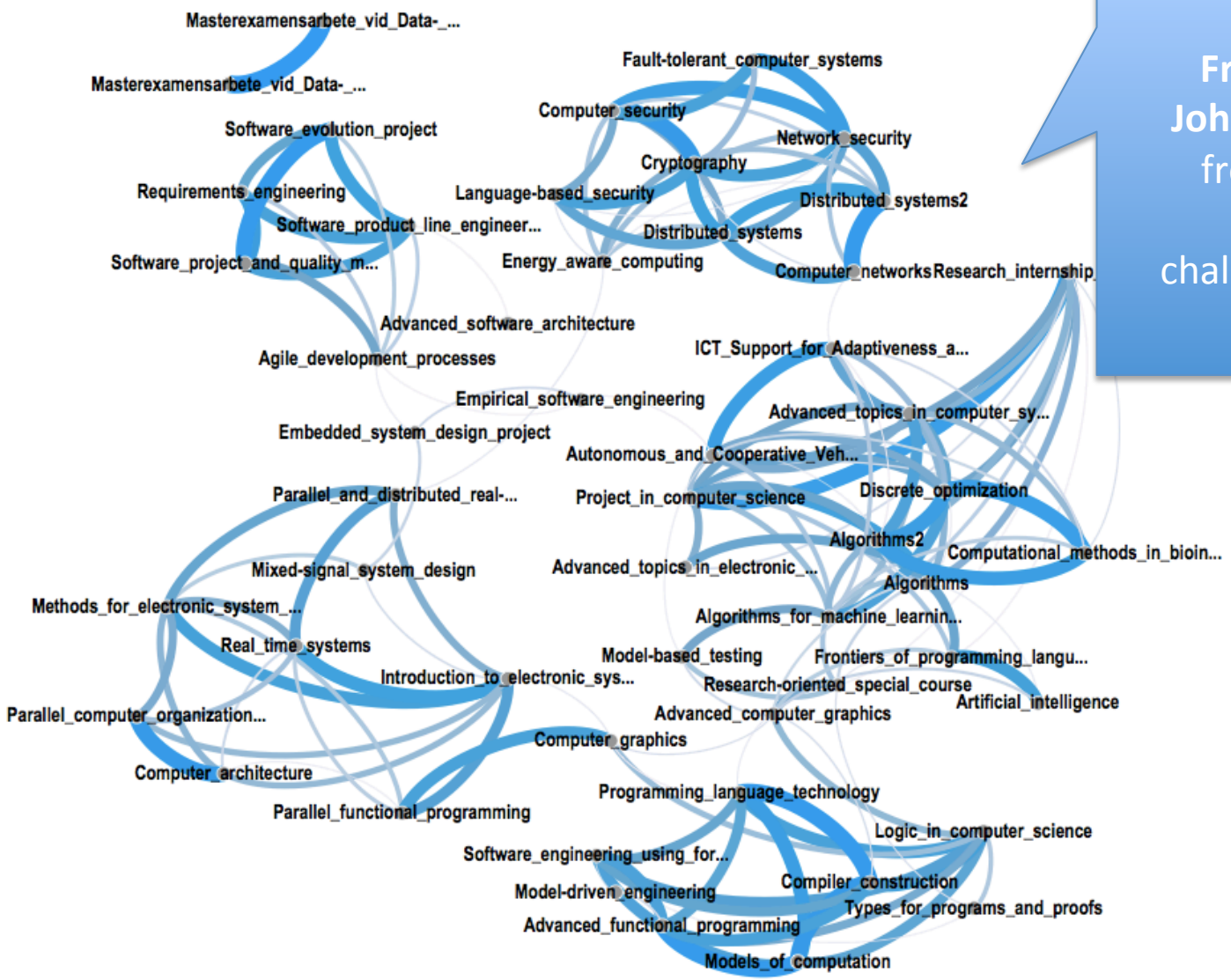


Paths and Cycles

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



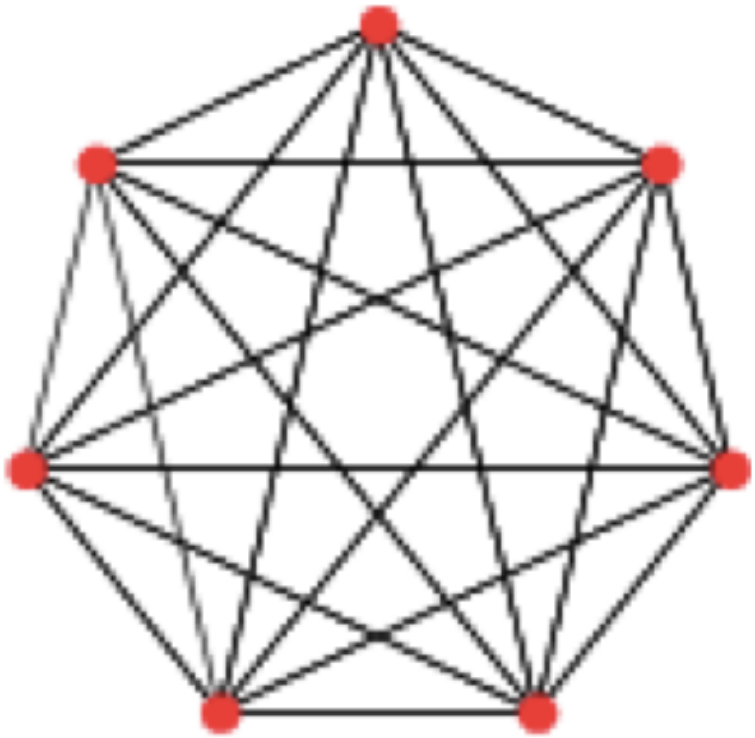
Paths and Cycles



Fredrik Johansson
frejohk
@
chalmers.se

More Graphs

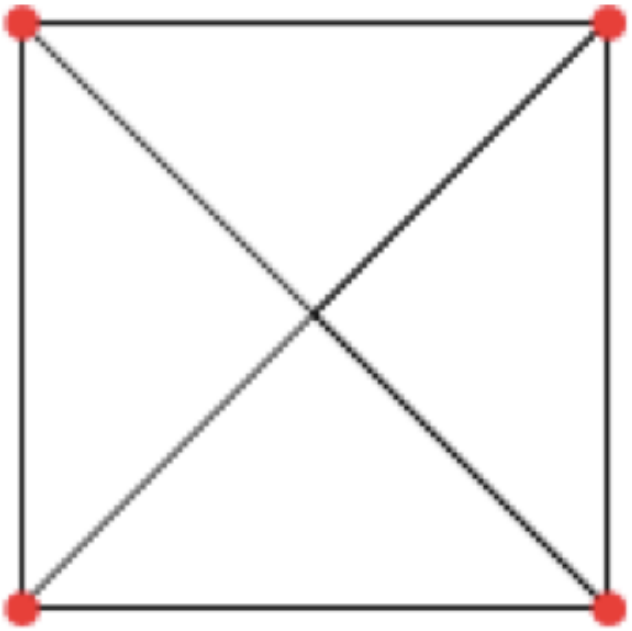
- A graph where there is an edge between any two nodes is called a **complete graph**



Question

- How many cycles are there in the following complete graph ?

1. **4**
2. **5**
3. **6**
4. **7**



govote.at
Code 683175

More Graphs

Let's consider the following notation for graphs:

The graph G = (V,E) where

V – represents the set of vertices

E – represents the edges of the graph

When |E| is O(|V|^2) the graph is **dense**, otherwise it is **sparse**

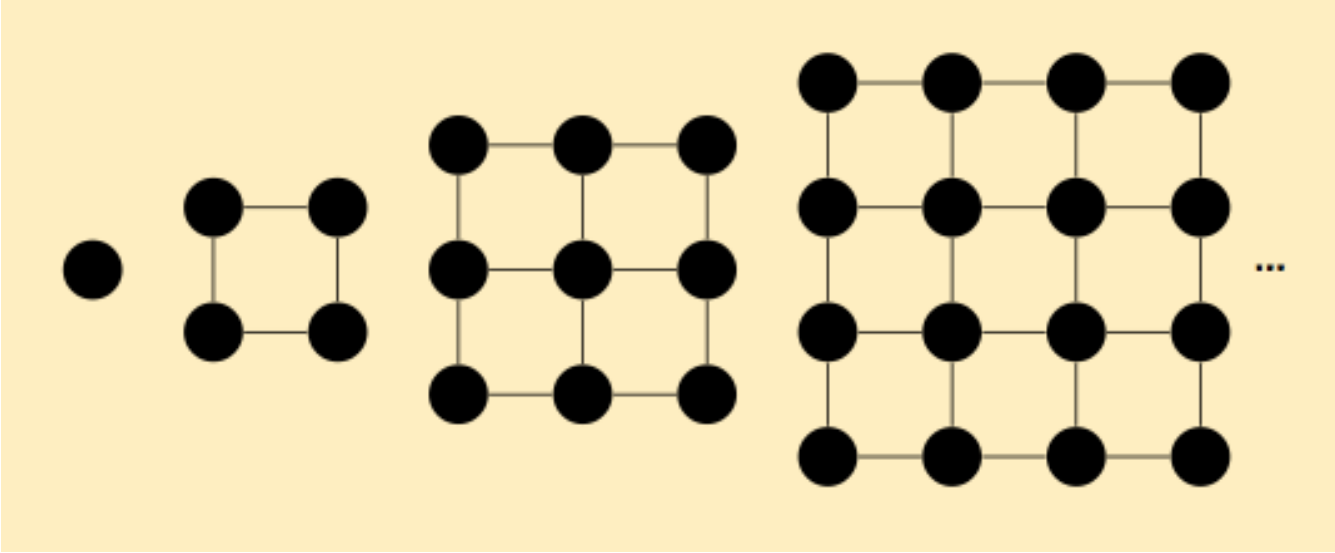
- Complete graphs are dense, because

  |E| = |V| (|V| - 1) – maximum number of edges

Question

Is the following kind of graph dense ?

1. Yes
2. No



govote.at
Code 649102

Implementing a Graph – Take 1

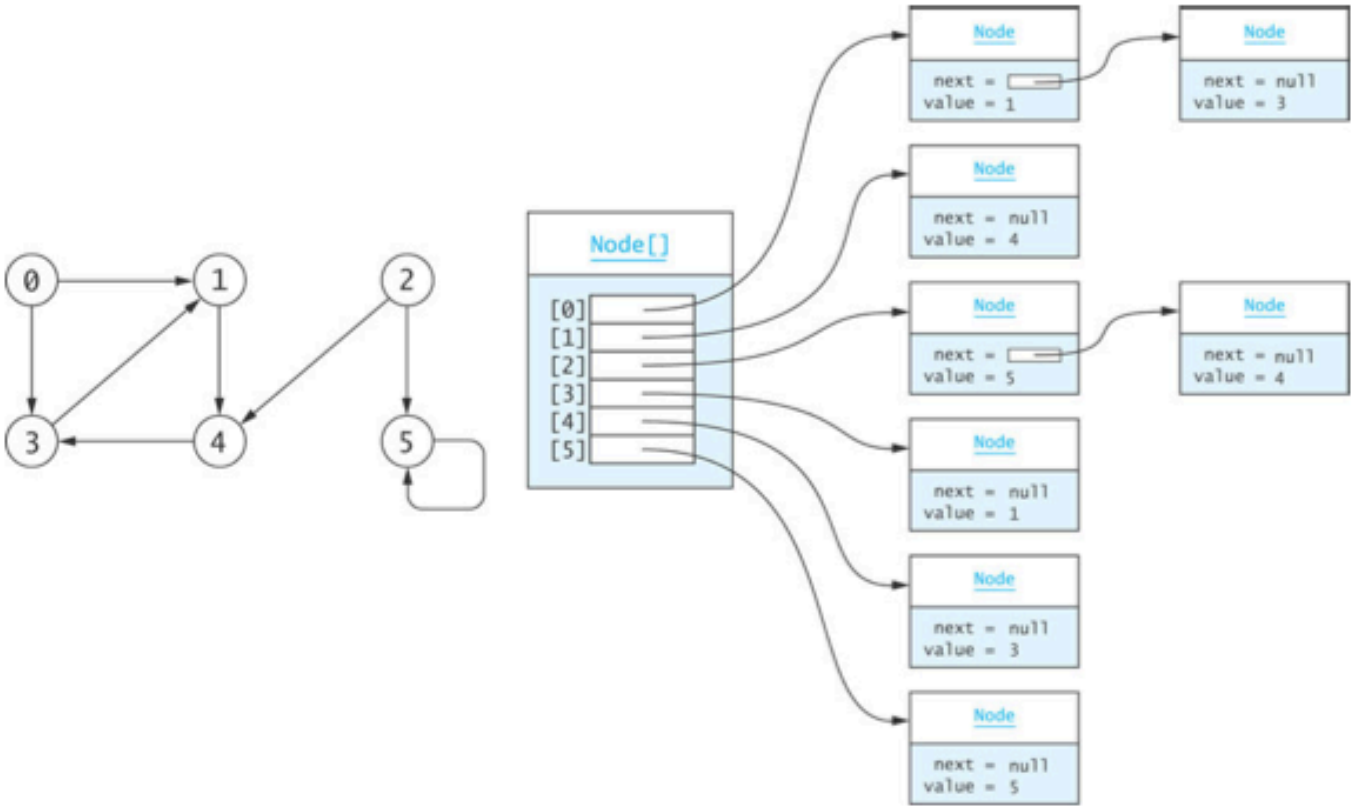
Adjacency lists

-keep a list of all nodes in the graph

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

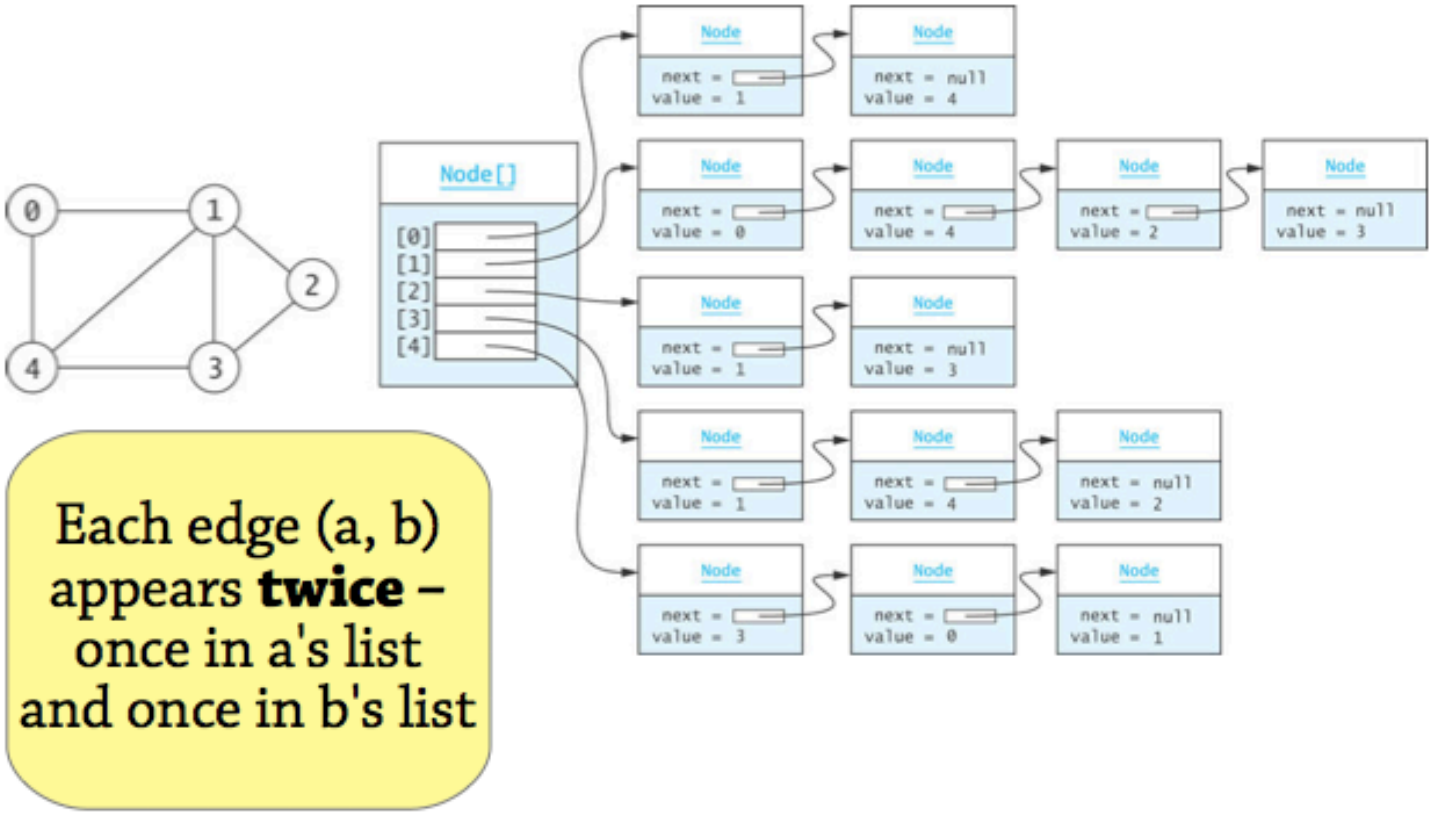
Implementing a Graph – Take 1

Adjacency lists – directed graph



Implementing a Graph – Take 1

Adjacency lists – undirected graph



Implementing a Graph – Take 2

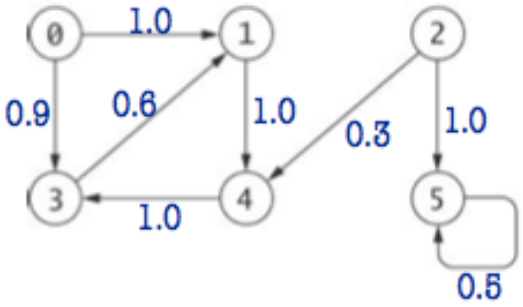
Adjacency matrix

- 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

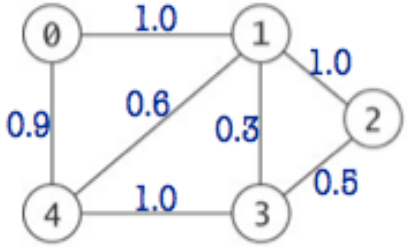
Implementing a Graph – Take 2

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	

Implementing a Graph – Best Way

- 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 time 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 time complexity of O(|V|^2)

- **Winner**:

  + adjacency lists for sparse graphs

  + unclear for dense graphs

Implementing a Graph – Best Way

- 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 bool)
  - 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

- 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

Breadth-First Search

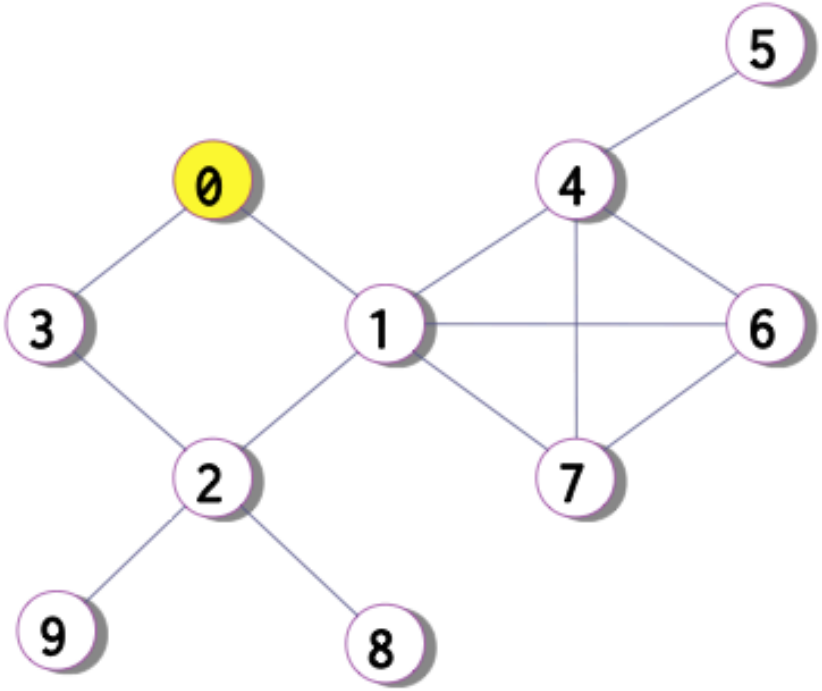
- Implementing BFS:
  - we maintain a queue of nodes that we are going to visit next
  - initially, the queue contains the start node
  - we repeat the following process:
    - remove a node from the queue
    - visit it
    - find all nodes adjacent to the visited node and add them to the queue, **unless** they have been visited or added to the queue already

Breadth-First Search

Queue:

0

Visit order:



Initially, queue contains start node

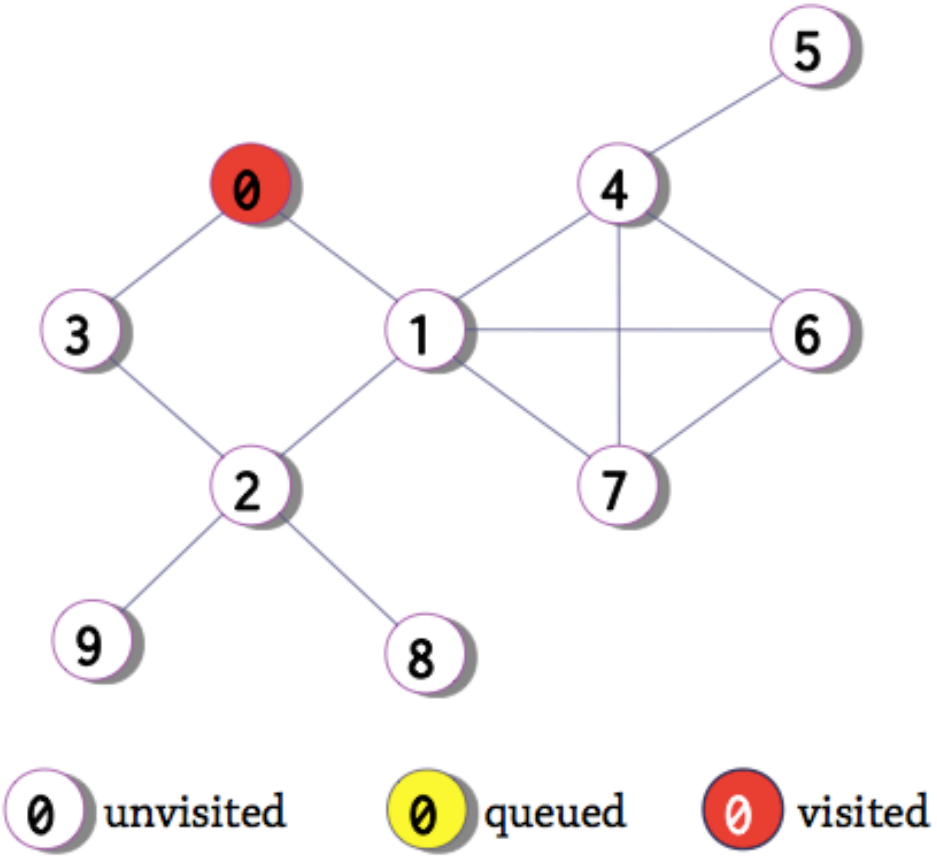
0 unvisited 0 queued 0 visited

Breadth-First Search

Queue:

Visit order:
0

Step 1:
remove node
from queue
and visit it



Breadth-First Search

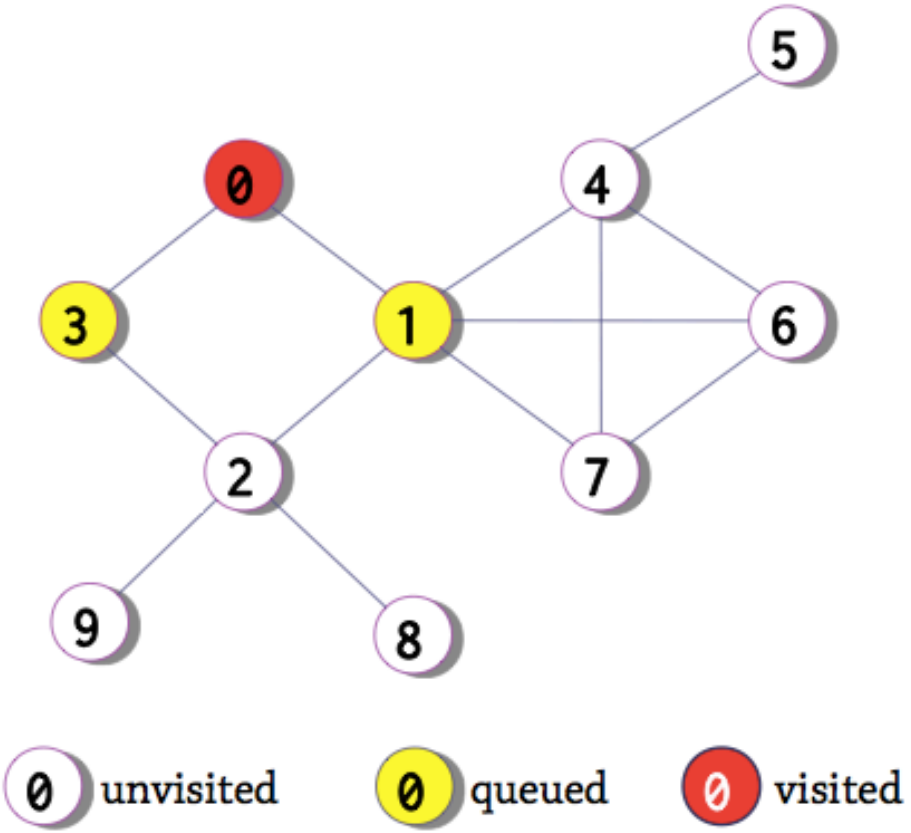
Queue:

3 1

Visit order:

0

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



Breadth-First Search

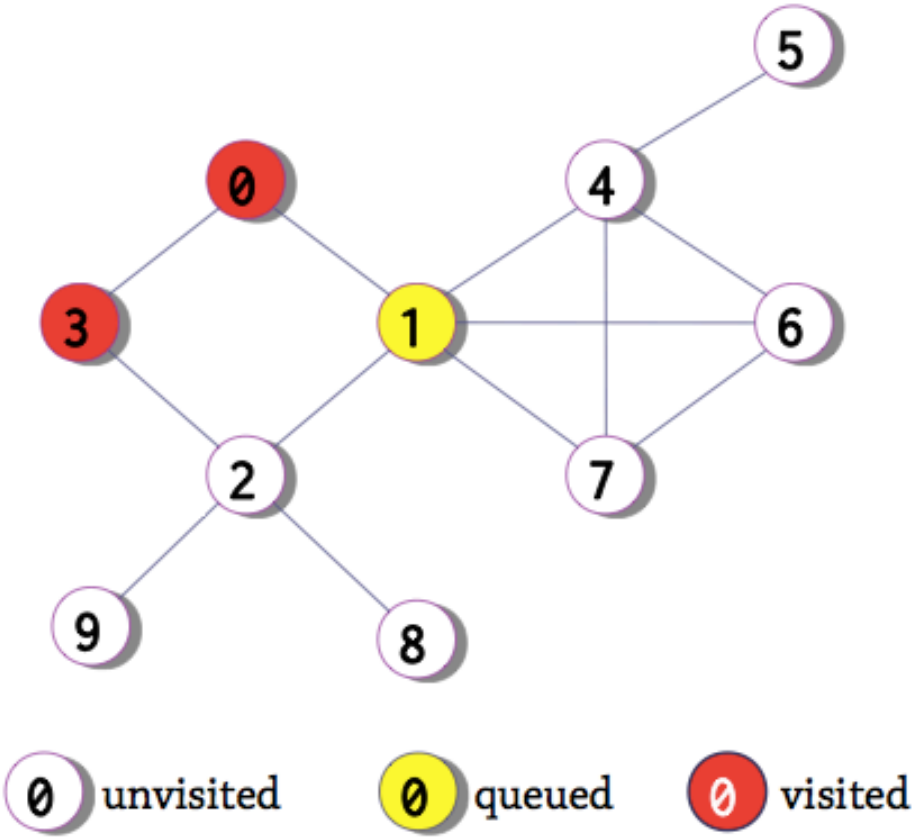
Queue:

1

Visit order:

0 3

Step 1:
remove node
from queue
and visit it



Breadth-First Search

Fast forward to the end

(demonstration on the board)

Breadth-First Search

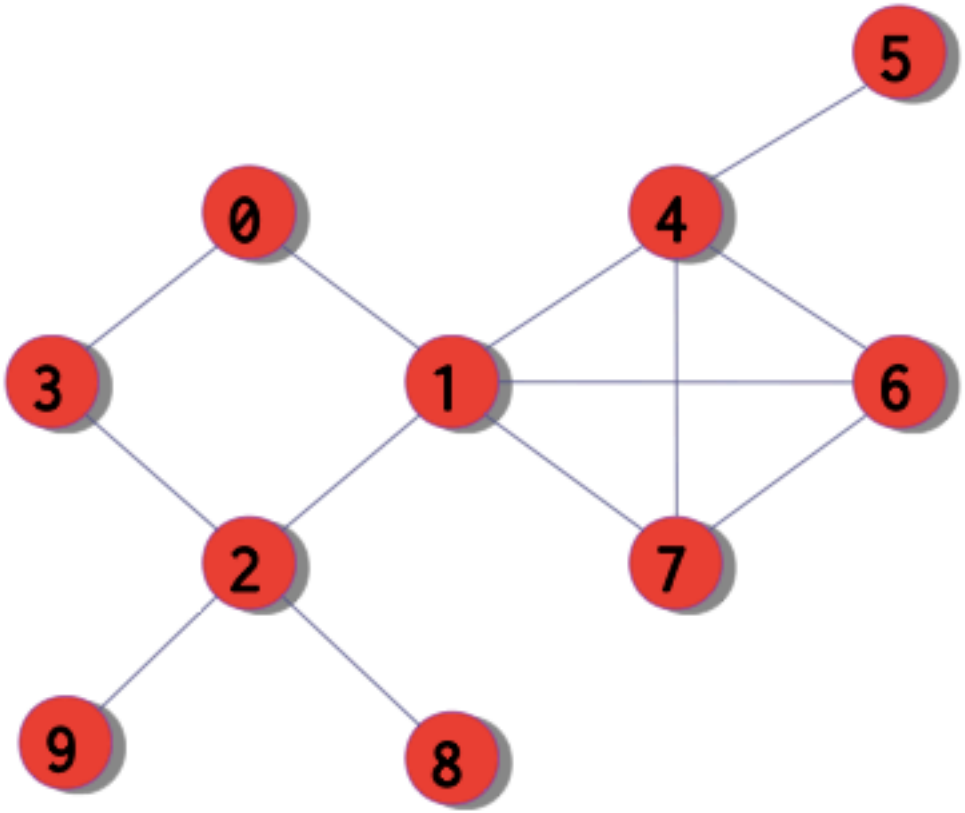
Queue:

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!**



0 unvisited 0 queued 0 visited

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 ☺

Question

- Is it possible to obtain DFS by replacing the queue from BFS with another data structure ?

1. Yes
2. No

govote.at
Code 916362

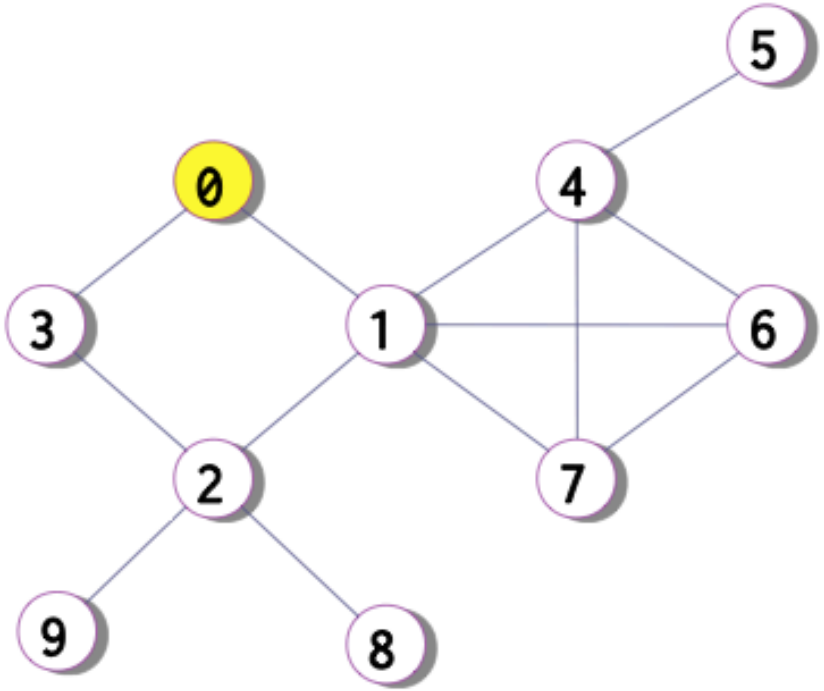
Depth-First Search

Stack:

0

Visit order:

Initially, stack contains start node



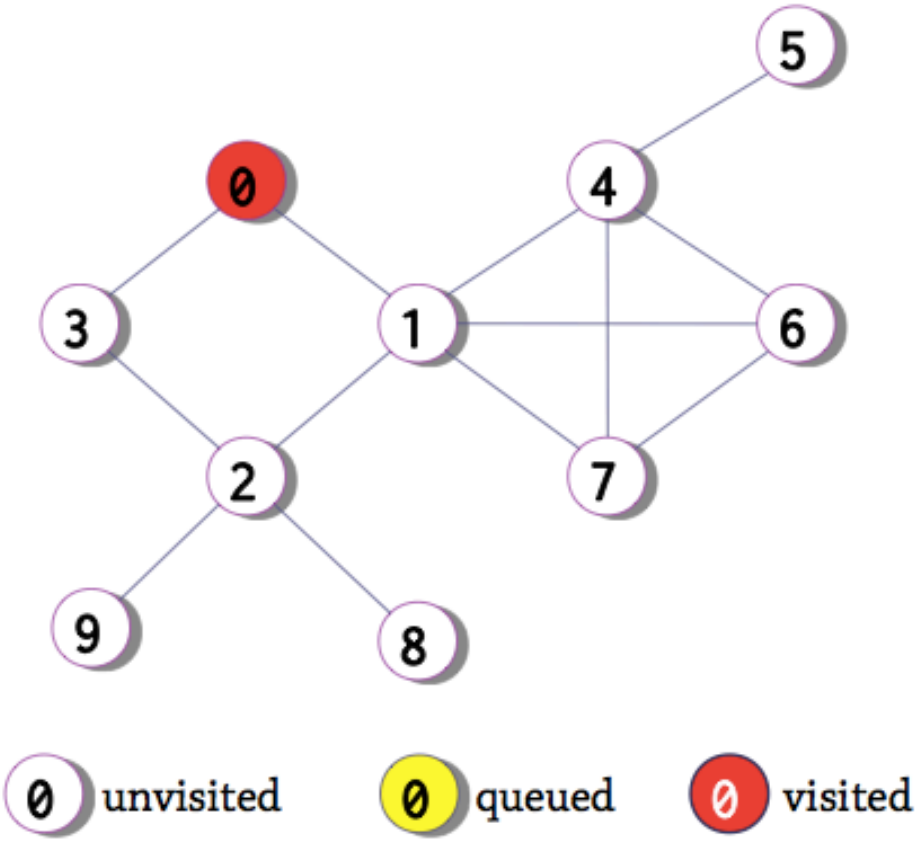
0 unvisited 0 queued 0 visited

Depth-First Search

Stack:

Visit order:
0

Step 1:
remove node
from stack
and visit it



Depth-First Search

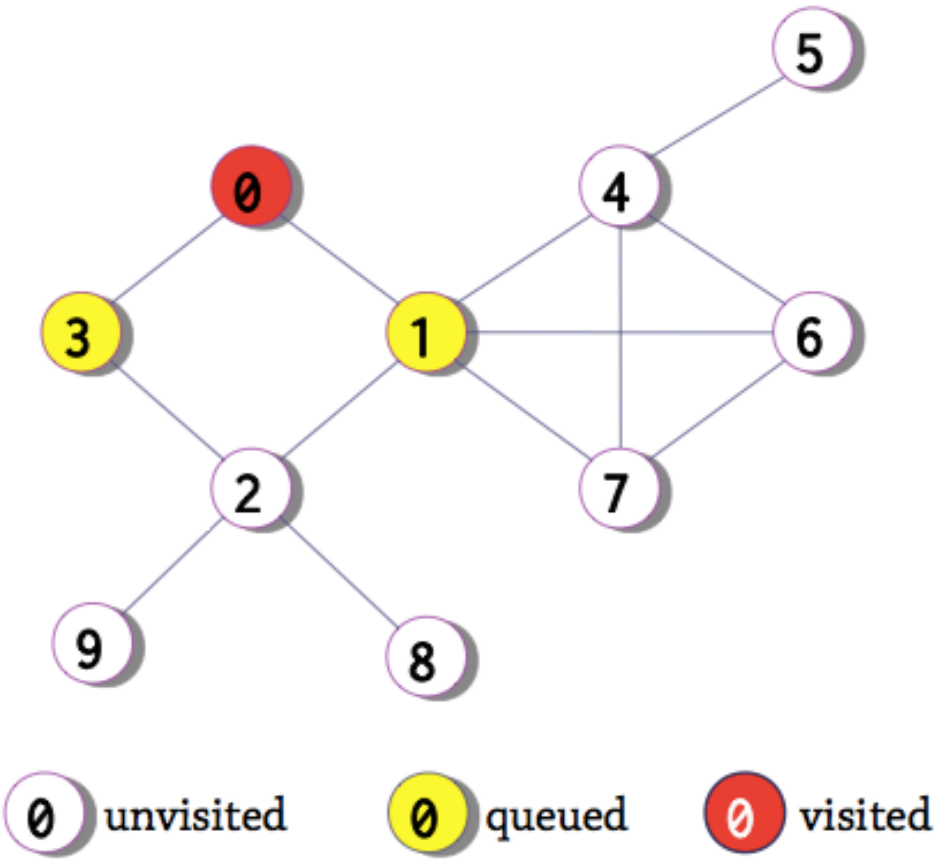
Stack:

3 1

Visit order:

0

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



Breadth-First Search

Fast forward to the end

(demonstration on the board)

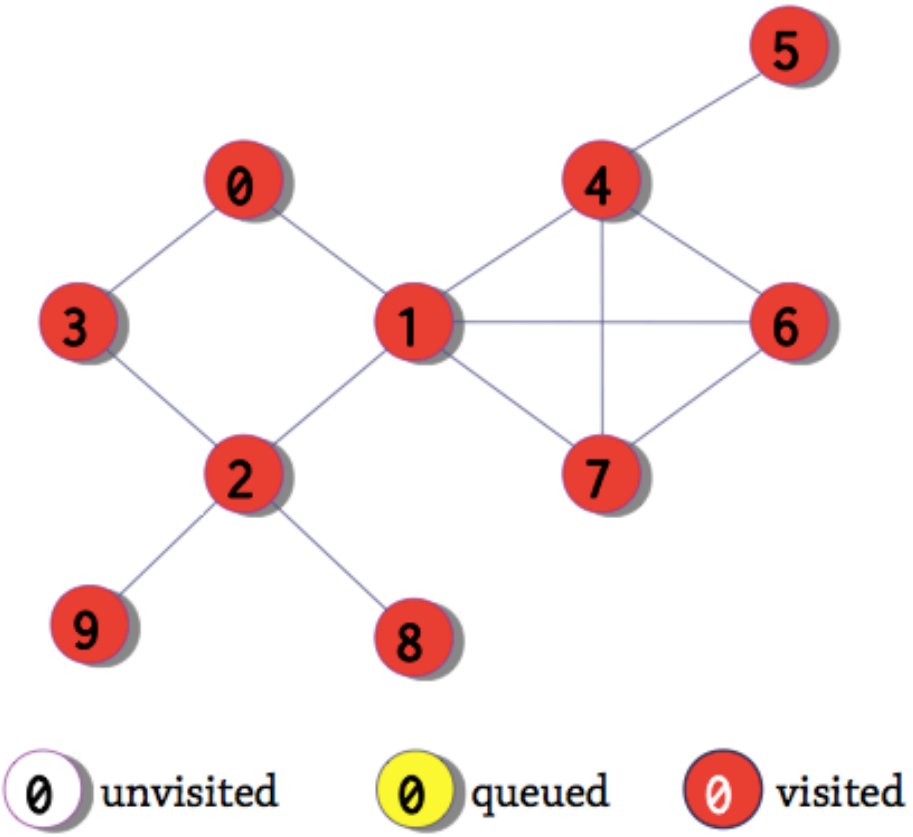
Depth-First Search

Stack:

Visit order:

0 1 6 4 5

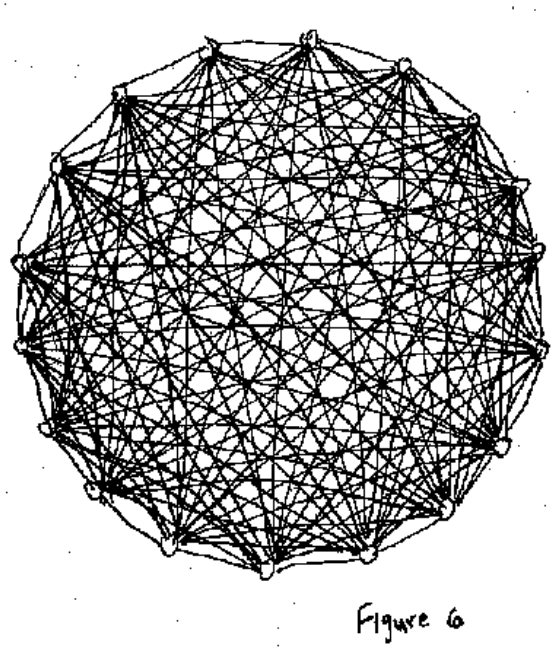
7 2 8 9 3



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

To Do



Read from the book:

+ 9.1 – 9.3

Implement:

+ graphs in your favourite programming language

Fun with graphs:

+ famous graph problem: **The Seven Bridges of Königsberg**

http://www.mathsisfun.com/activity/seven-bridges-konigsberg.html