Chapter 2:
Exploring the Labyrinth

Panos Louridas

Athens University of Economics and Business
Real World Algorithms
A Beginners Guide
The MIT Press

Outline

Definition

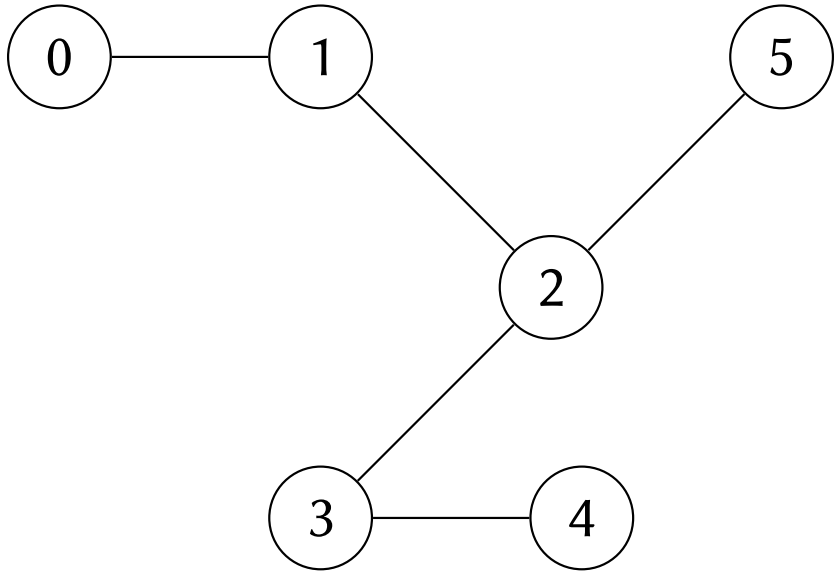
The Labyrinth Exploration Problem

- We are inside a labyrinth.
- The labyrinth consists of rooms that are connected with corridors.

Question

How can we visit all the rooms in the labyrinth?

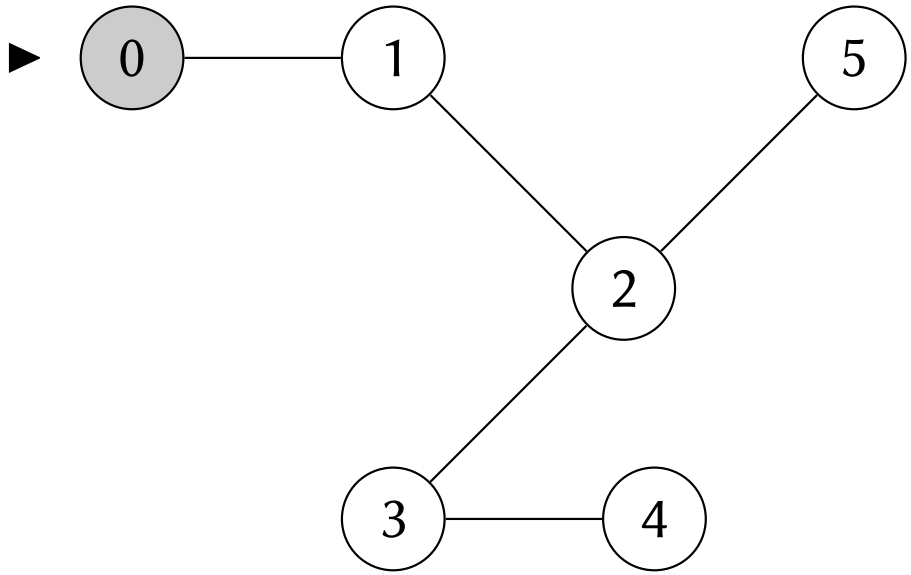
Example



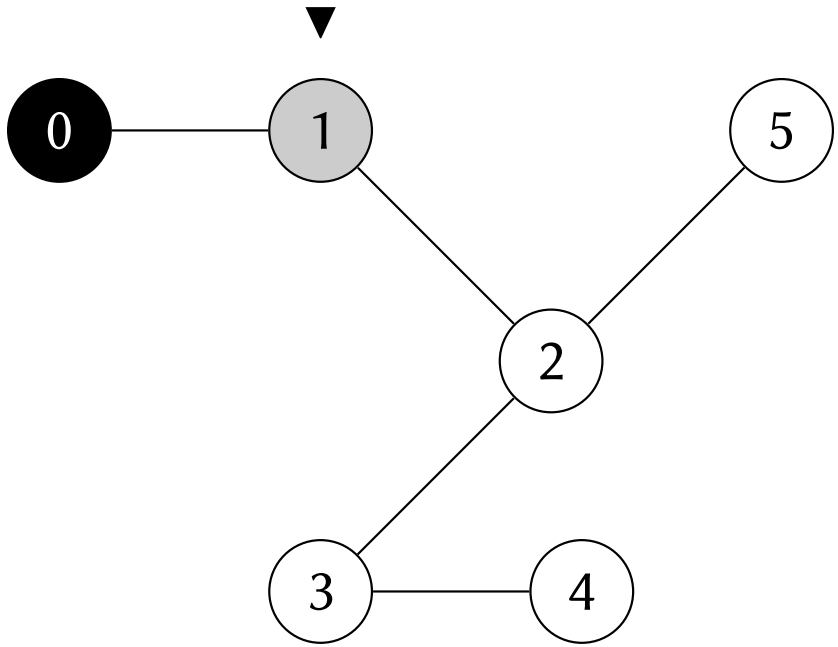
Hand on the Wall Strategy

- We place our hand on the wall.
- We move across the rooms and the corridors without lifting our hand from the wall.

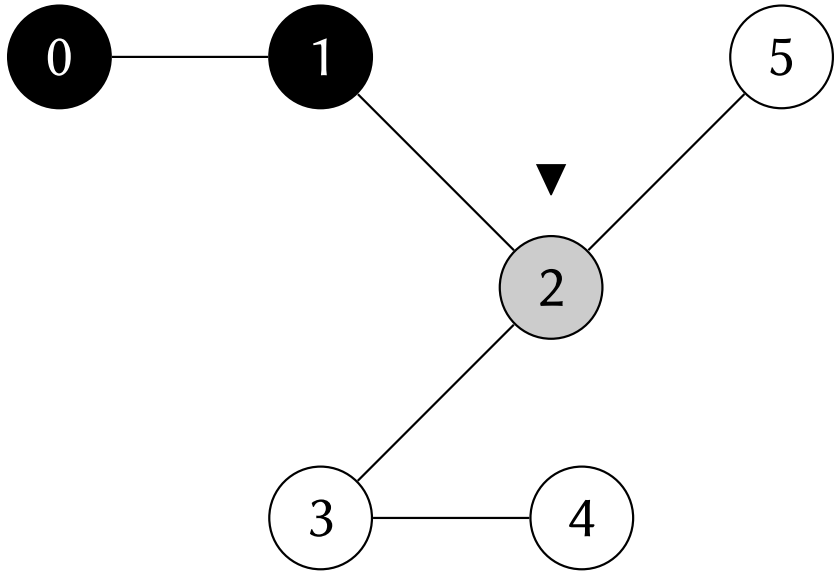
Example: It Works (1)



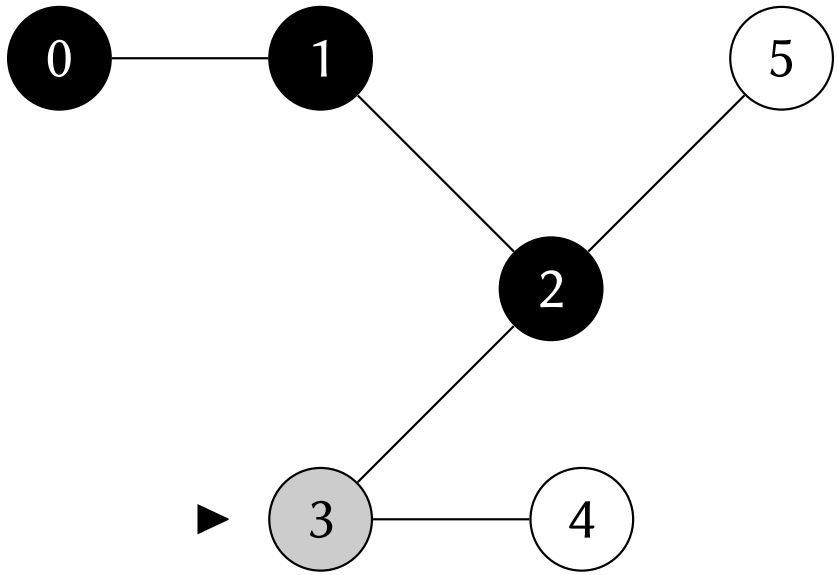
Example: It Works (2)



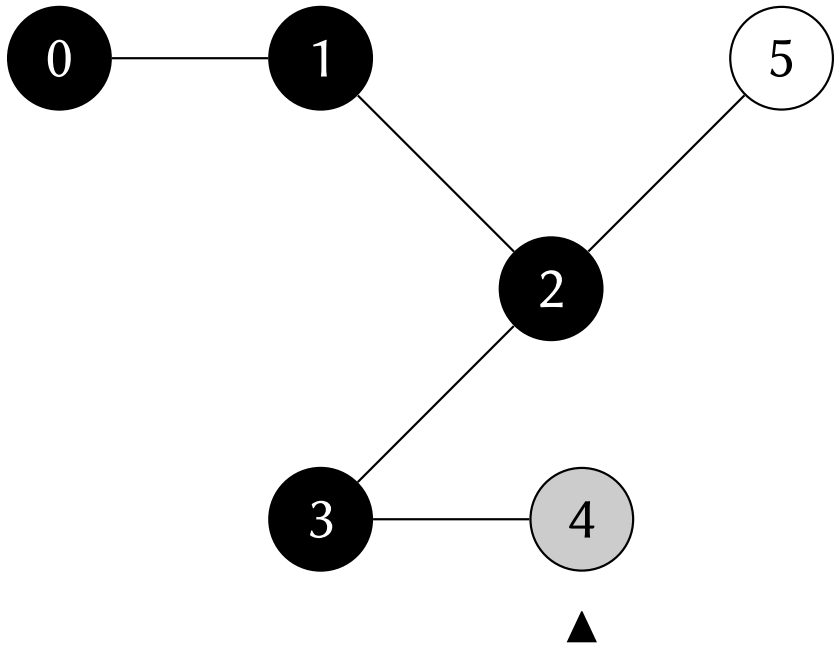
Example: It Works (3)



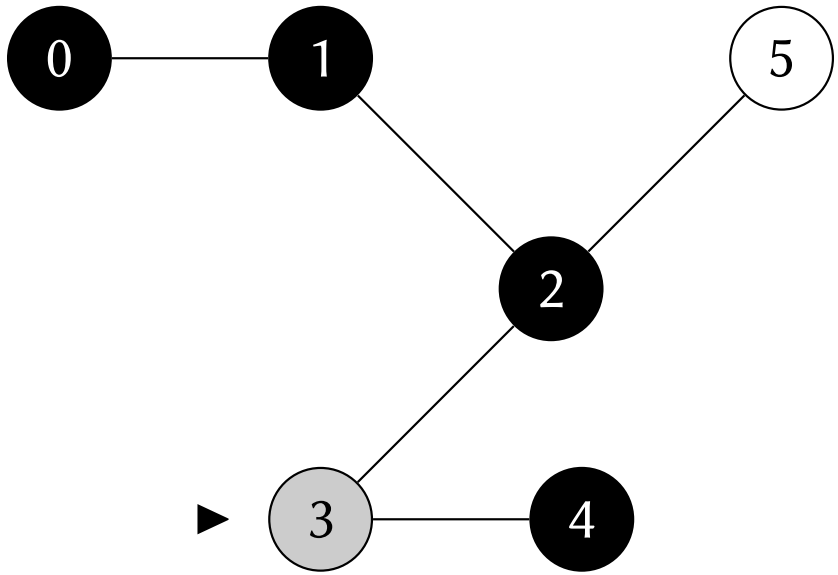
Example: It Works (4)



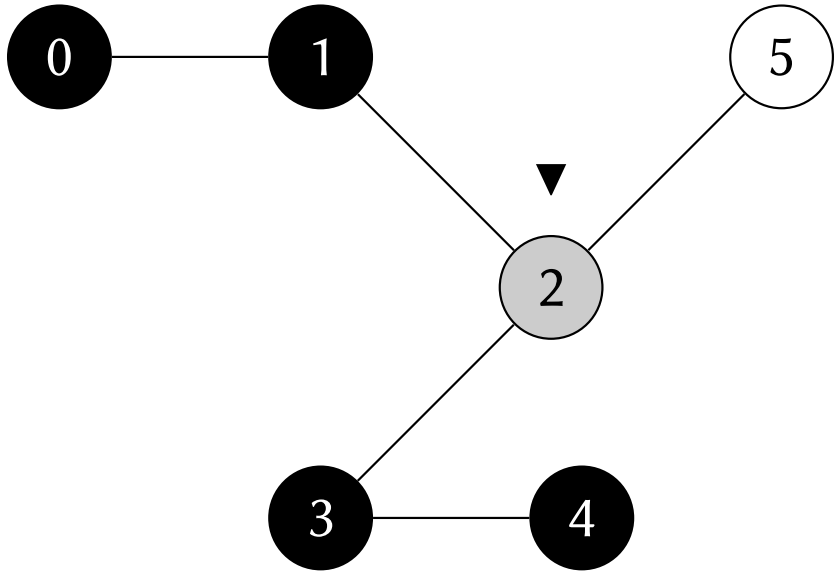
Example: It Works (5)



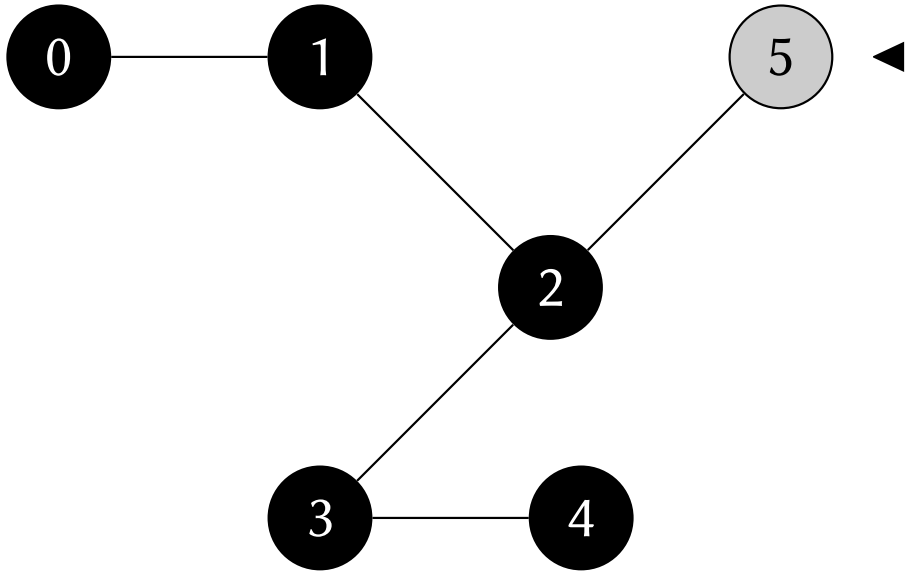
Example: It Works (6)



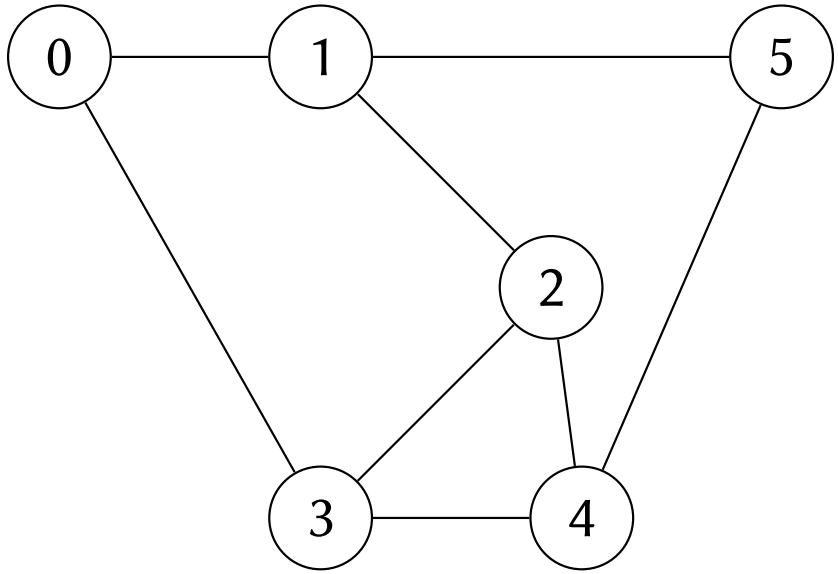
Example: It Works (7)



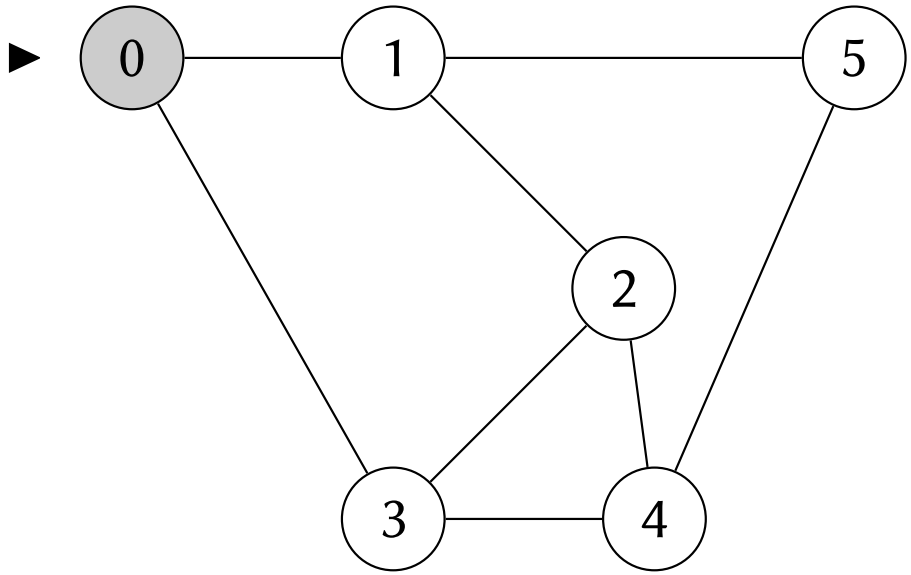
Example: It Works (8)



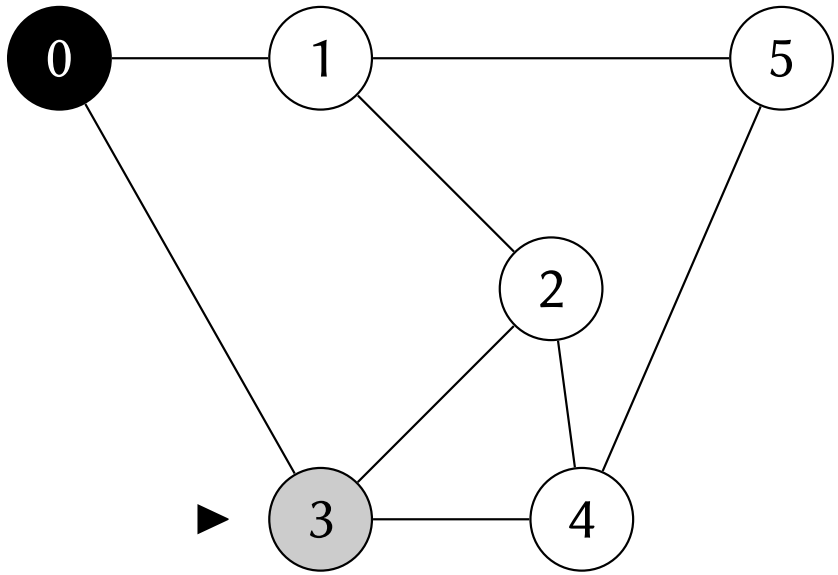
Example: It Does Not Work... (1)



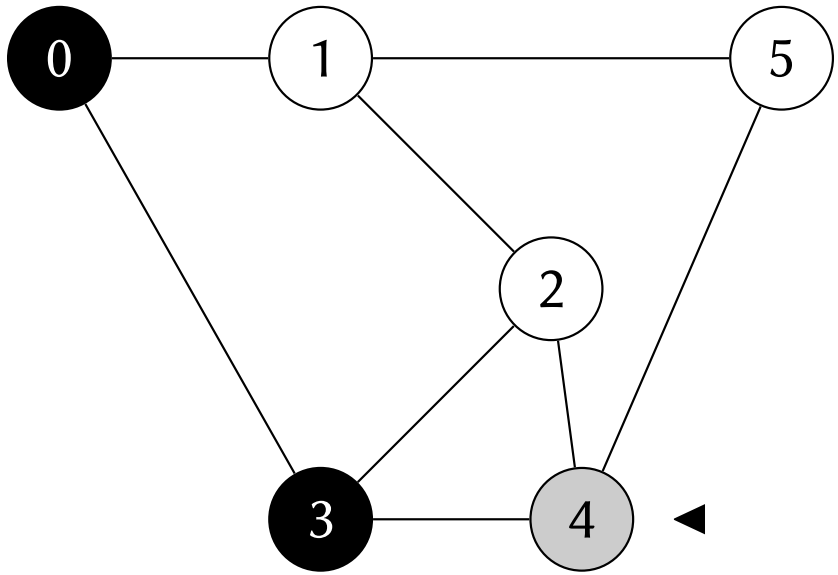
Example: It Does Not Work...(2)



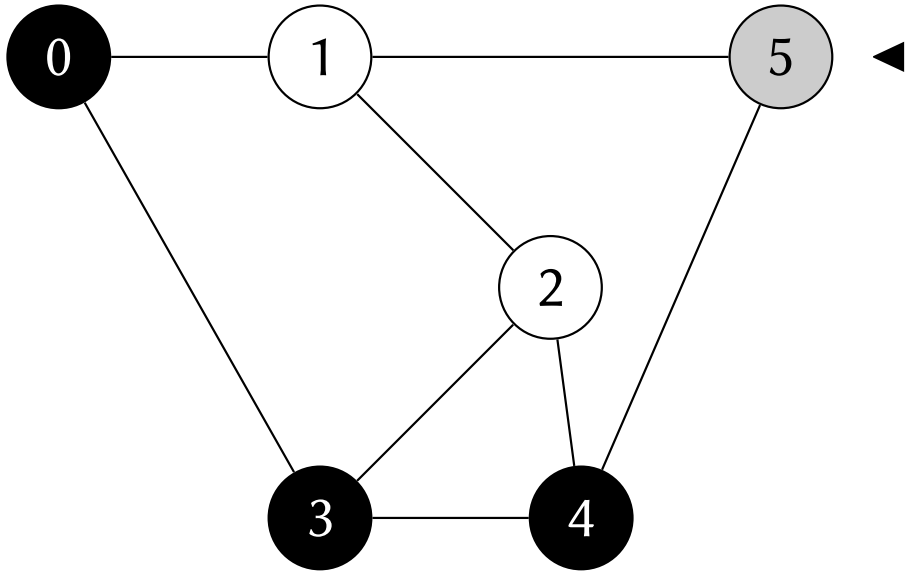
Example: It Does Not Work...(3)



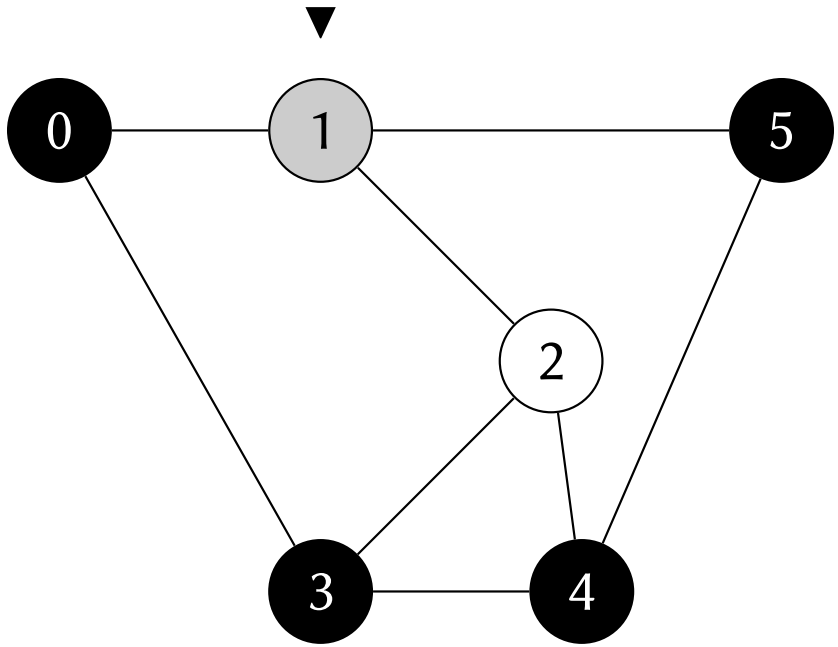
Example: It Does Not Work...(4)



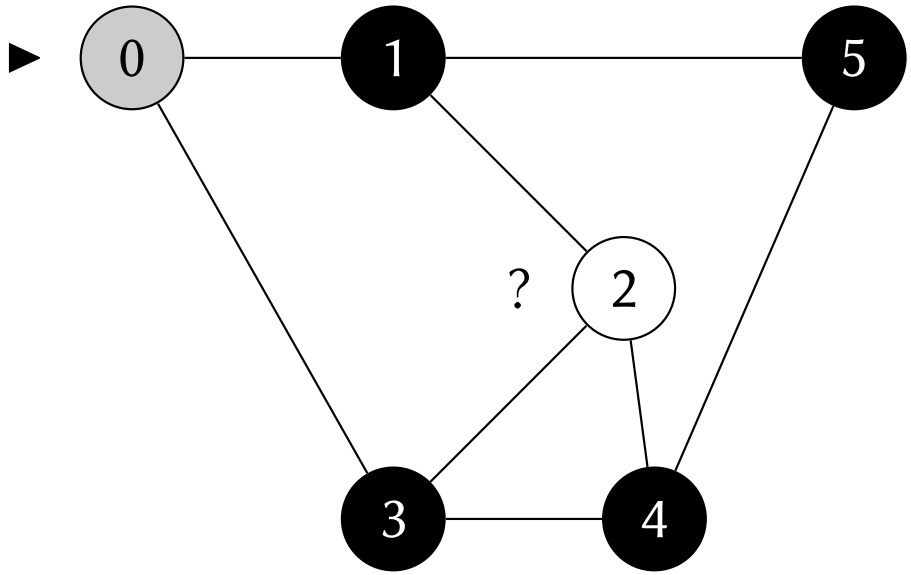
Example: It Does Not Work...(5)



Example: It Does Not Work...(6)



Example: It Does Not Work...(7)



Outline

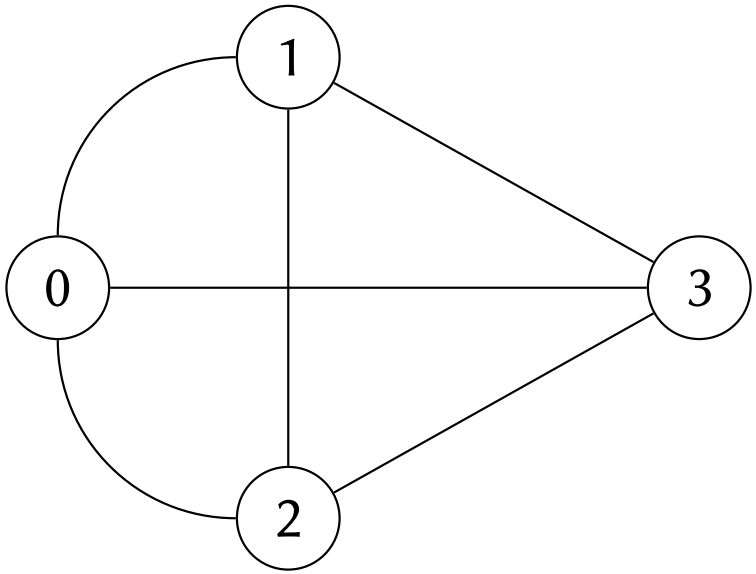
Description

- A *graph* consists of *nodes* or *vertices* and *links* or *edges* connecting them.
- The links may or may not be directed.
- For a graph G we write $G = (V, E)$, where V is the set of vertices and E is the set of edges.

Undirected Graphs

- A graph whose links are not directed is an *undirected graph*.
- In this case, the set of edges consists of sets $\{x, y\}$ where x and y are connected vertices.
- We usually write (x, y) instead of $\{x, y\}$. The order of x and y does not matter.

Undirected Graph Example



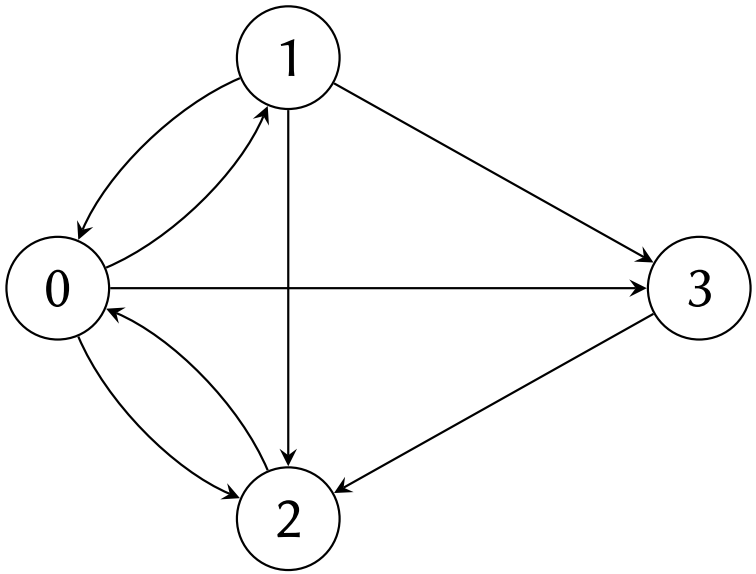
$$V = \{0, 1, 2, 3\}$$

$$\begin{aligned} E &= \{\{0,1\},\{0,2\},\{0,3\},\{1,2\},\{1,3\},\{2,3\}\} \\ &= \{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\} \end{aligned}$$

Directed Graphs

- A graph whose edges are directed is a *directed graph*, or *digraph* for short.
- In this case, the set of edges consists of tuples (x, y), where the order of x and y is important.

Directed Graph Example



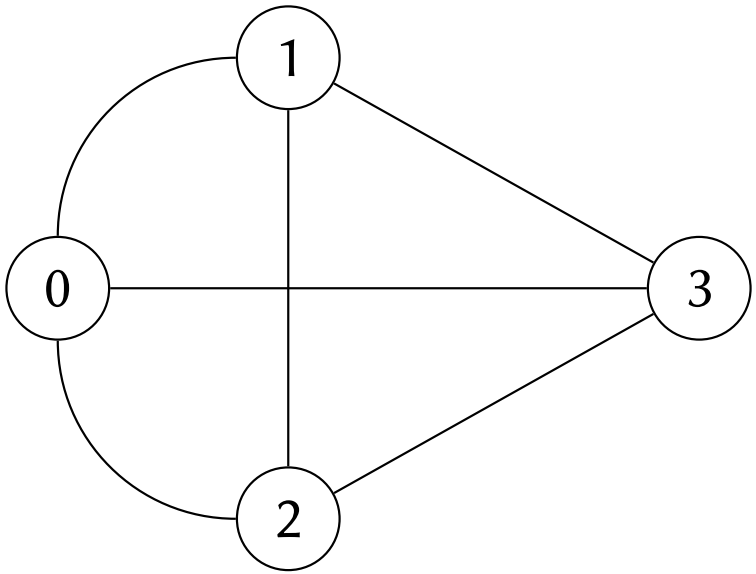
$$V = \{0, 1, 2, 3\}$$

$$\begin{aligned} E = \{&(0, 1), (0, 2), (0, 3), \\ &(1, 0), (1, 2), (1, 3), \\ &(2, 0), \\ &(3, 2)\} \end{aligned}$$

Paths

- A sequence of links such that each consecutive pair of links share an edge is called a *path*.
- The number of links in a path is its *length*.
- If there exists a path between two vertices, they are *connected*.

Path Example

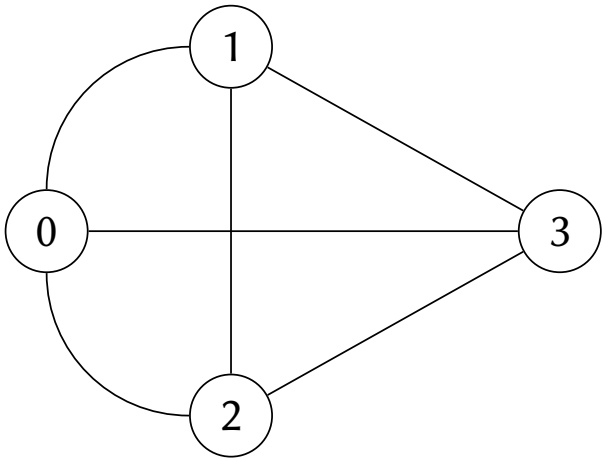


- The path from node 0 to node 3 through node 1 or node 2 has length 2.
- The path from node 0 to node 3 without an intermediary node has length 1.

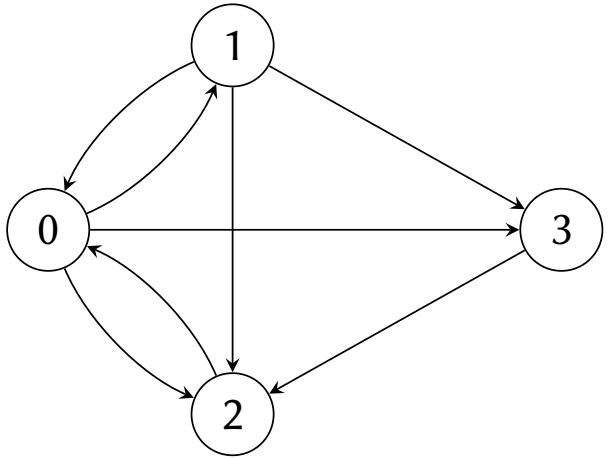
Degree of a Node

- The number of edges adjacent to a node is the node's *degree*.
- In directed graphs, the number of incoming links is its *in-degree* and the number of outgoing links is its *out-degree*.

Example of Node Degrees



Node 0 has degree 3.



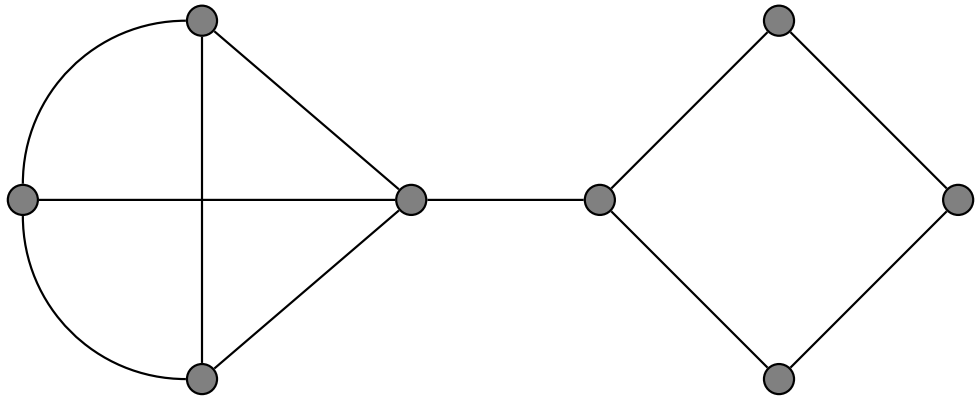
Node 0 has in-degree 2 and out-degree 3.

Applications of Graphs

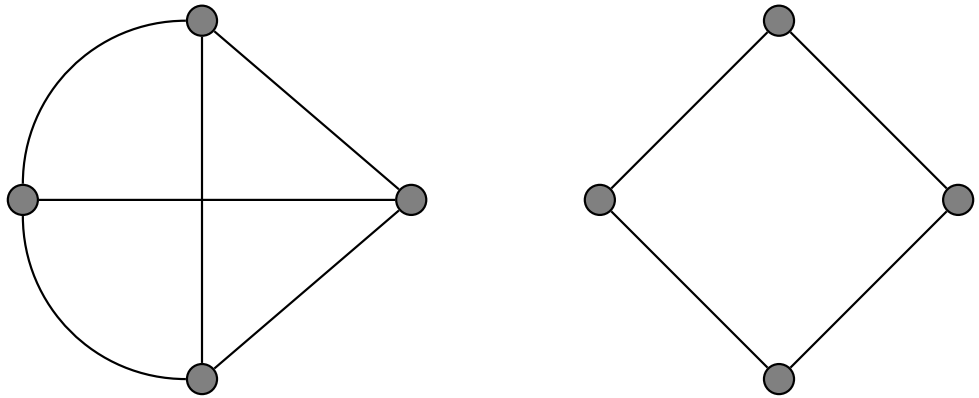
- Networks (computers, communications, transport).
- Internet, world wide web.
- Electronic circuits.
- Metabolic networks.
- Social networks.
- Time scheduling.
- And many more.

Outline

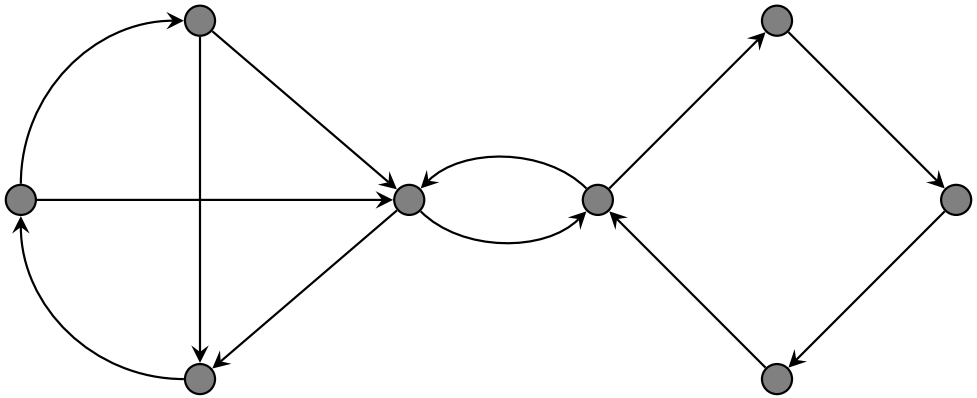
Connected Graph



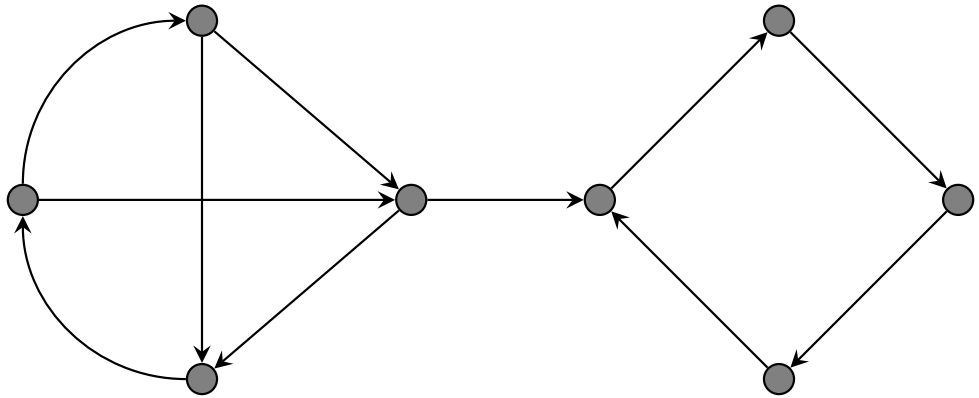
Disconnected Graph



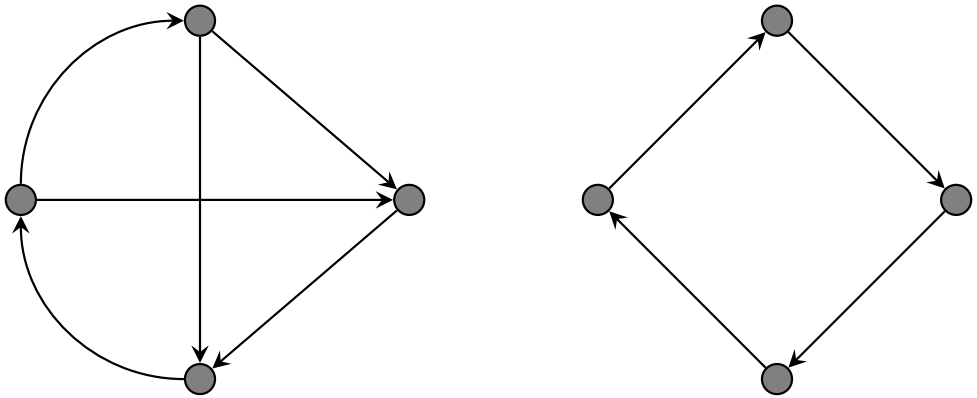
Strongly Connected Graph



Weakly Connected Graph



Disconnected Directed Graph



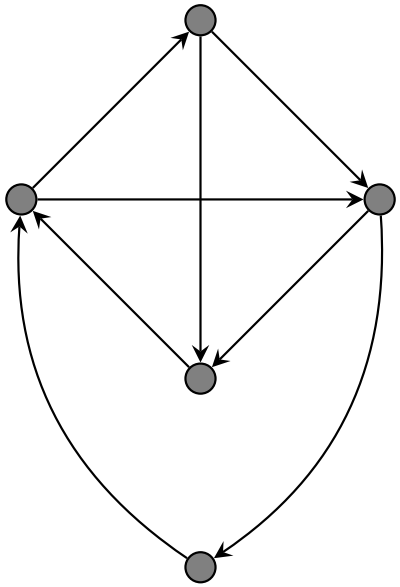
Connected Components

Definition

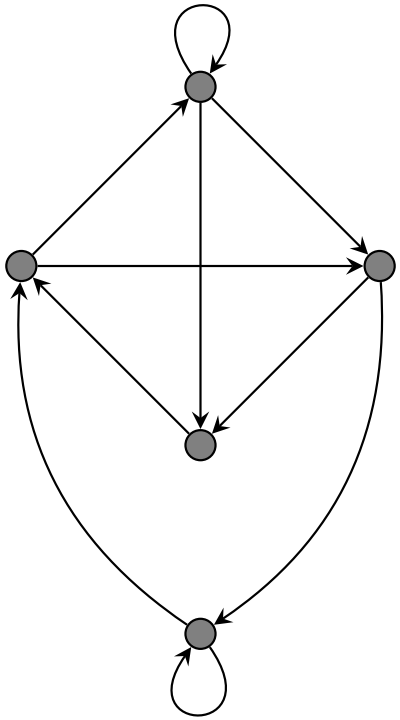
A subset of a graph whose nodes are connected is called a *connected component*.

A graph is (strongly) connected if it consists of a single connected component.

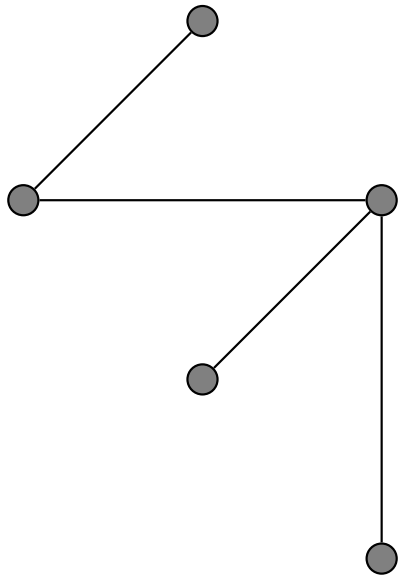
Directed Cyclic Graph



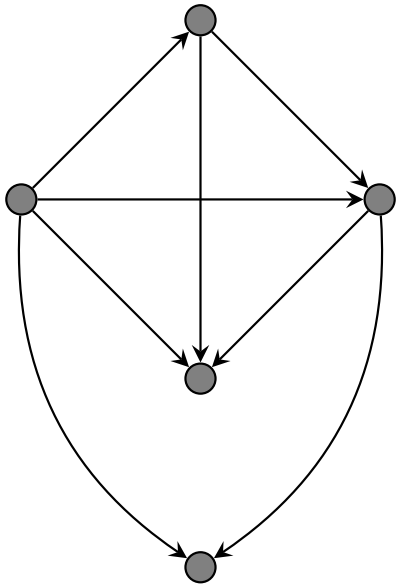
Directed Cyclic Graph with Loops



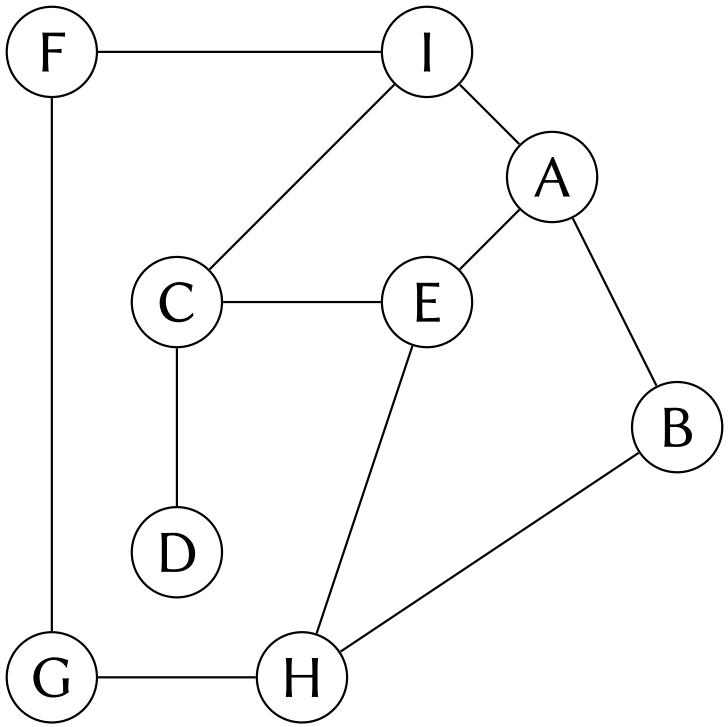
Undirected Acyclic Graph



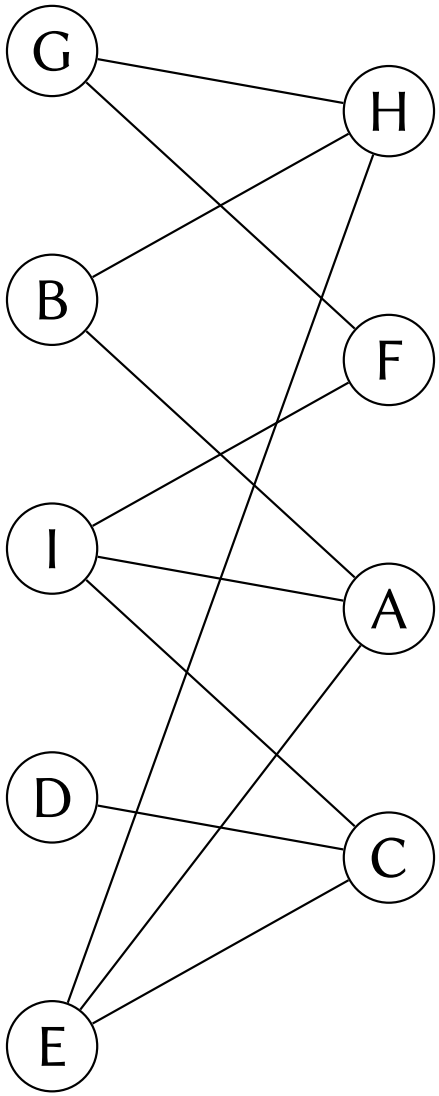
Directed Acyclic Graph—dag)



Bipartite Graph



Bipartite Graph (Contd.)



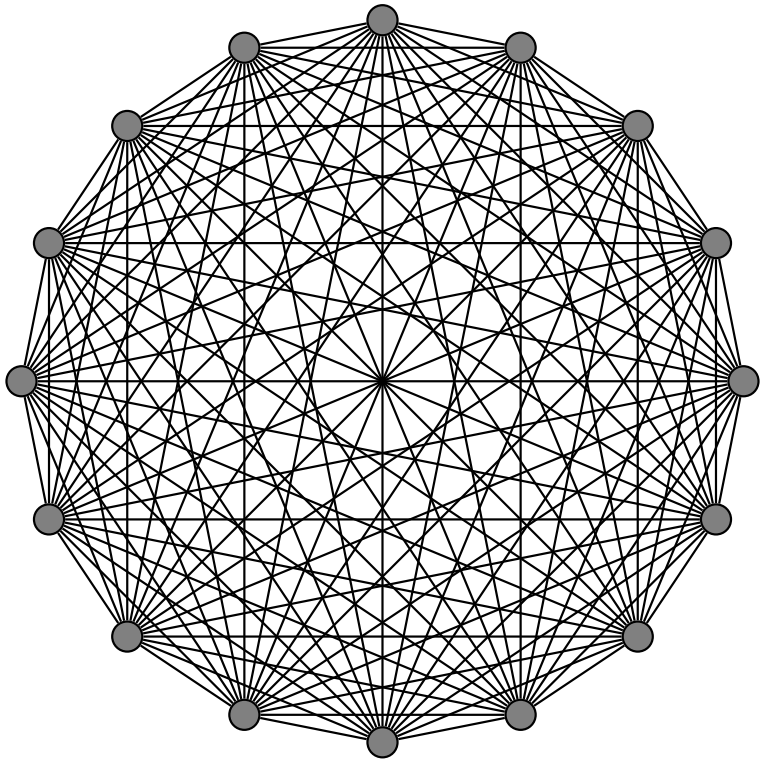
Dense and Sparse Graphs

Definition

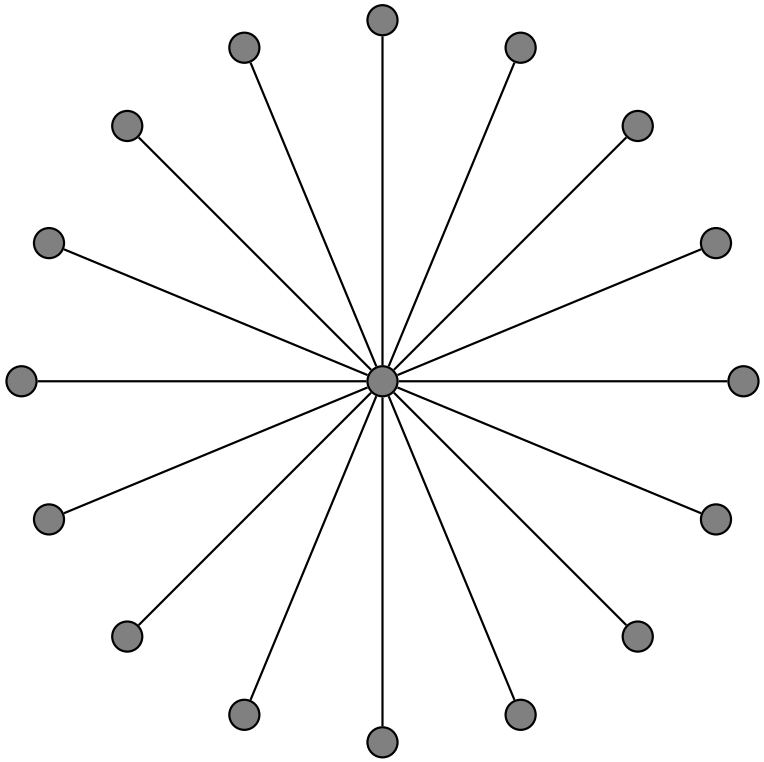
If the number of links in a graph is close to n^2, where n is the number of nodes, then the graph is *dense*. Otherwise it is *sparse*.

Why n^2? Because the number of all possible links is $n(n-1)/2$.

Complete Graph



Sparse Graph



Sparse Graph Example

- Suppose we have a social network with 7 billion nodes, $n = 7 \times 10^9$.
- Suppose that each node has 1,000 friends.
- The number of links is 7×10^{12}, or 7 trillion.
- The number $n(n-1)/2$ for $n = 7 \times 10^9$ is approximately 2.5×10^{19}, or 25 quintillion, many times over 7 trillion.

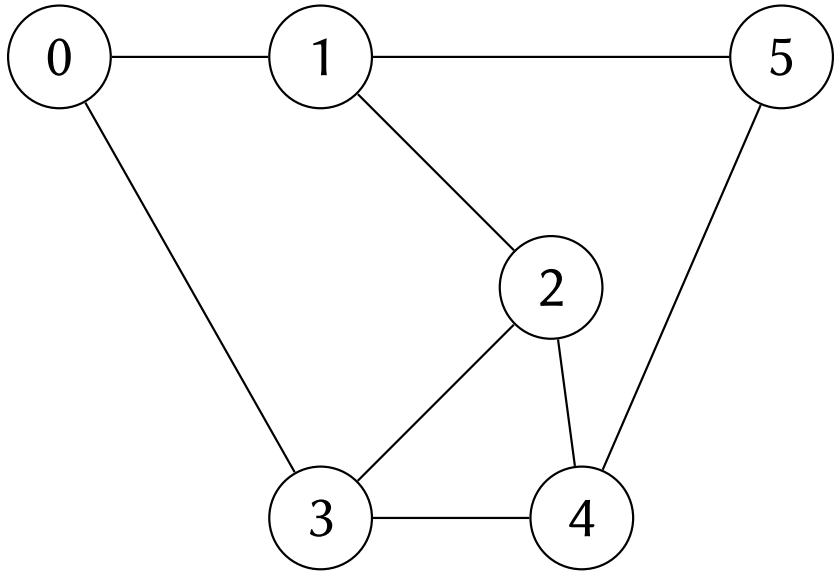
Outline

Adjacency Matrix

Definition

Suppose we have a graph $G = (V, E)$, where V is the set of vertices and E the set of edges. The *adjacency matrix* of G is a matrix whose (i, j) cell is 1 if there is a link between nodes i and j, otherwise it is 0.

Adjacency Matrix Example



Adjacency Matrix Example (Contd.)

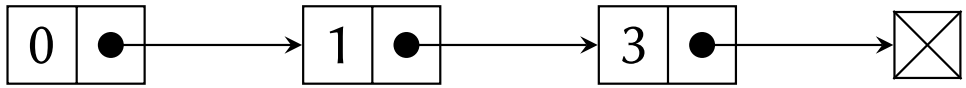
	0	1	2	3	4	5
0	0	1	0	1	0	0
1	1	0	1	0	0	1
2	0	1	0	1	1	0
3	1	0	1	0	1	0
4	0	0	1	1	0	1
5	0	1	0	0	1	0

Lists

Definition

- A list is a data structure whose elements consist of two parts.
- The first part contains the data describing each element.
- The second part contains a link to the next element in the list.
- A list has a head, containing its first element.
- When an element does not have a next element, we say that it points to NULL.
- The elements of a list are called *nodes*.

List Example



List Creation and Insertion

- `CreateList()`, creates and returns a new empty list.
- `InsertListNode`(L, p, n), inserts a new node n in the list L after node p; the node n must already exist and `InsertListNode` just puts it inside the list.
- `InsertInList`(L, p, d), creates a node with data d inserts it inside list L after node p.

InsertInList

Concerning `InsertInList`(L, p, d):

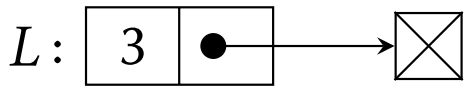
- If p is NULL, then the node containing d goes to the beginning of the list.
- It returns a pointer to the newly inserted node. In this way we can insert elements one after the other:
  $p \leftarrow$ `InsertInList`(L, p, i)
  $p \leftarrow$ `InsertInList`(L, p, j)
  etc.

Insertion in the Beginning of a List

L: [ ]

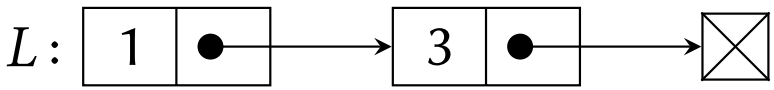
$L \leftarrow$ `CreateList()`

Insertion in the Beginning of a List (Contd.)



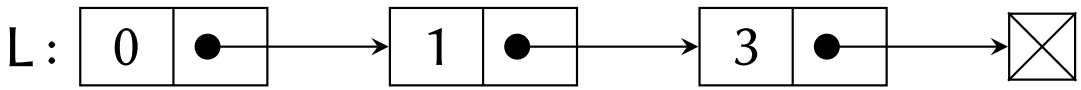
$p \leftarrow$ `InsertInList`$(L,$ `NULL`$, 3)$

Insertion in the Beginning of a List (Contd.)



`InsertInList`(L, NULL, 1)

Insertion in the Beginning of a List (Contd.)



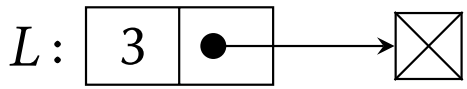
InsertInList(L, NULL, 0)

Insertion at the End of a List

L: [ ]

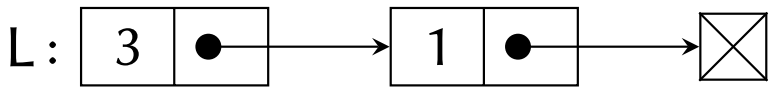
$L \leftarrow$ `CreateList()`

Insertion at the End of a List (Contd.)



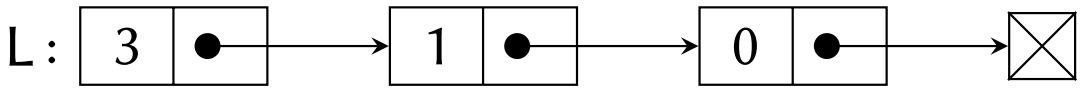
$p \leftarrow$ `InsertInList`$(L,$ `NULL`$, 3)$

Insertion at the End of a List (Contd.)



$p \leftarrow \texttt{InsertInList}(L, p, 1)$

Insertion at the End of a List (Contd.)

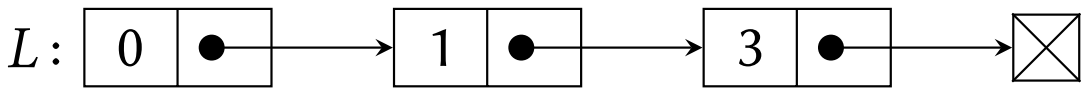


`InsertInList`$(L, p, 0)$

Removing Items

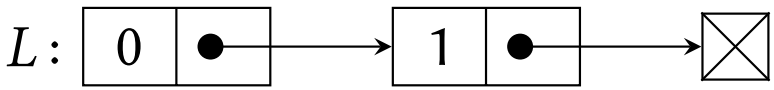
- `RemoveListNode`(L, p, r), removes node r from list L and returns it; p is its predecessor in the list. If r is not in the list, it returns NULL.
- `RemoveFromList`(L, d), removes from list L the first node with data d. If no such node exists, it returns NULL.

Removing Items Example



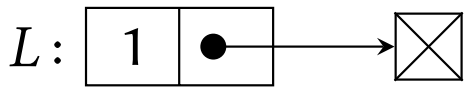
`RemoveFromList`$(L, 3)$

Removing Items Example (Contd.)



`RemoveFromList`$(L, 0)$

Removing Items Example (Contd.)



RemoveFromList(L, 1)

Removing Items Example (Contd.)

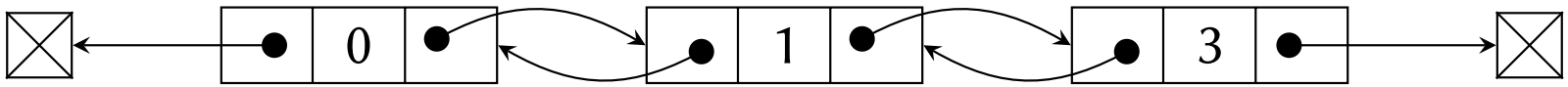
L: [ ]

List Search

- `GetNextListNode`(L, p), returns the node following p in list L. If p is the last node, it returns NULL. If p is NULL, it returns the head of the list. The node is not removed from the list.
- `SearchInList`(L, d), searches in list L for the first node that contains d and returns it, if it is found, without removing it from the list. If it is not found, it returns NULL.

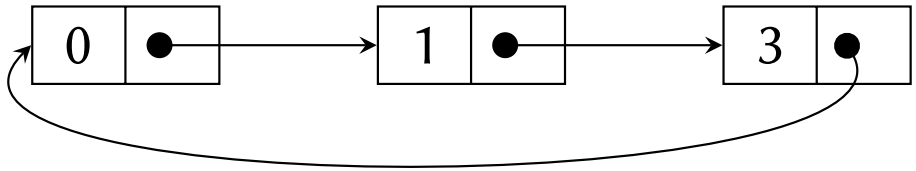
Doubly Linked List

In a *doubly linked list*, its elements point back (each to its previous element), as well as to their next element.

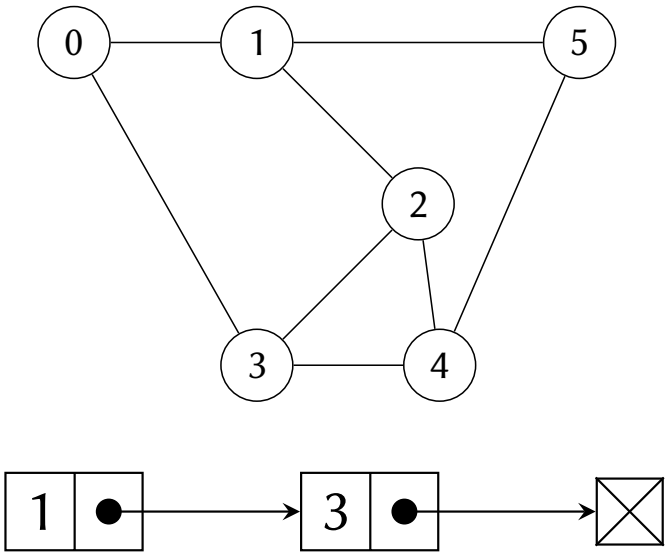


Circular Lists

In a *circular list* the last element of the list points to its first element instead of NULL.



Adjacency List



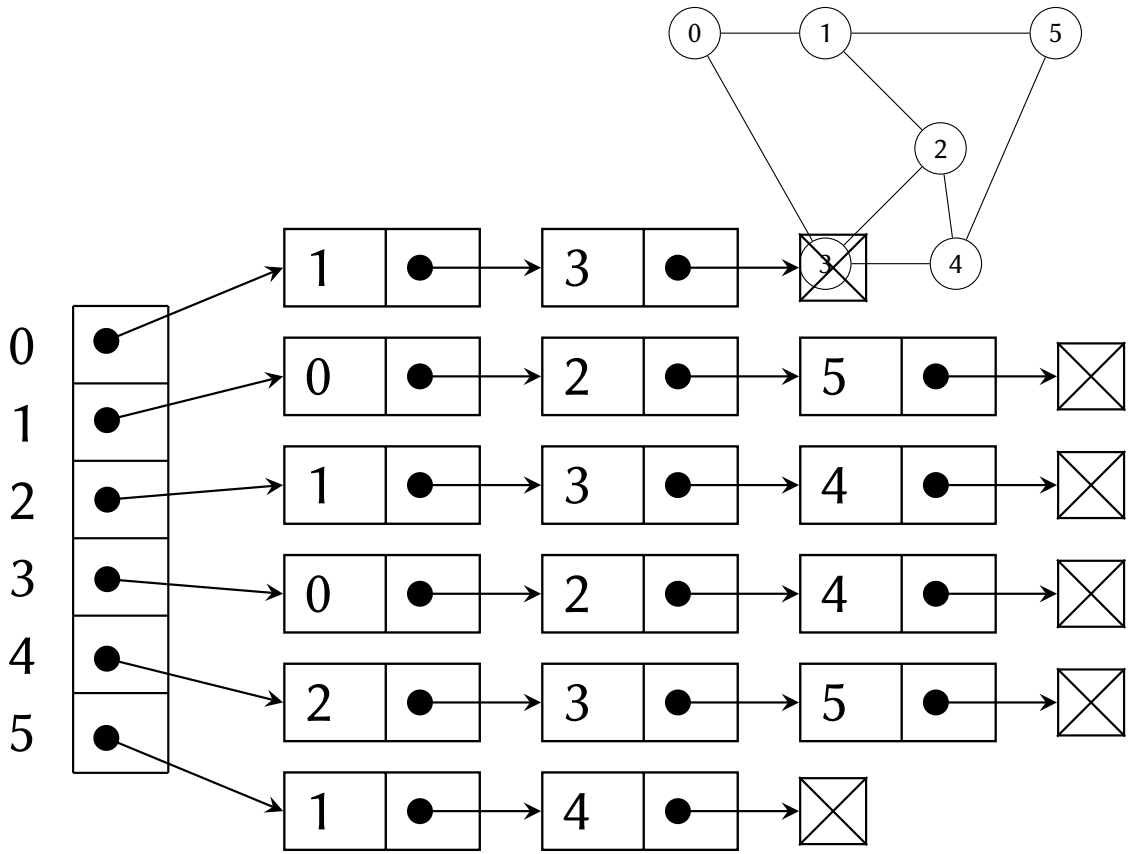
Definition

An adjacency list is a list containing the neighbors of a node.

Adjacency List Graph Representation

- We use an adjacency list for each graph node.
- We use a matrix A.
- In A, element $A[i]$ points to the head of the adjacency list of node i, or it points to NULL if the node has no neighbors.

Adjacency List Graph Representation Example



Comparison of Graph Representations

- Given a graph $G = (V, E)$, its adjacency matrix has $|V|^2$ elements.
- To represent the same graph with adjacency lists we need $|V| + |E|$ elements.
- To determine if a node is a neighbor of another node we need time $O(1)$ with an adjacency matrix.
- To determine if a node is a neighbor of another node we need time $O(|V|)$ with adjacency lists.
- With adjacency lists we gain in space, but we lose in time: that is a *space-time tradeoff*.

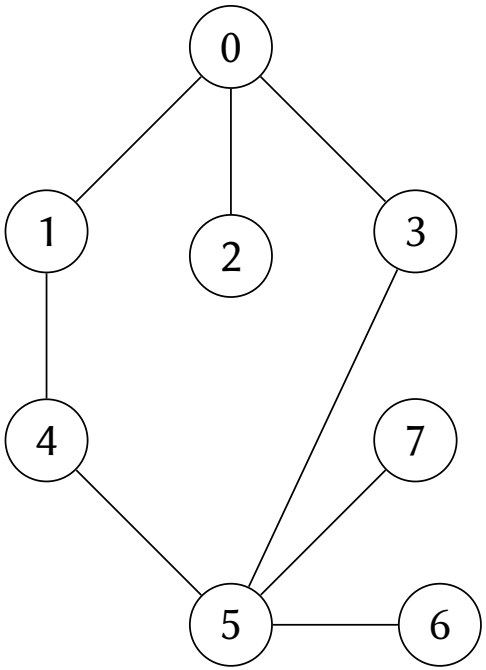
Outline

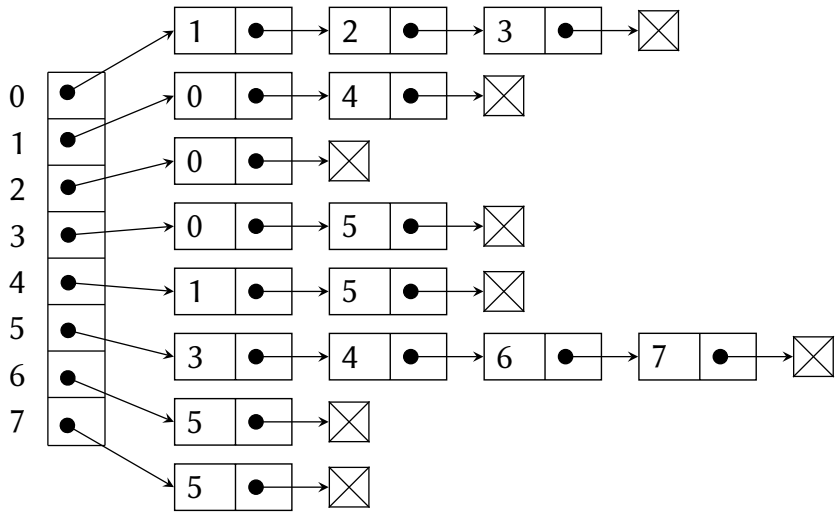
Depth-First Search

Definition

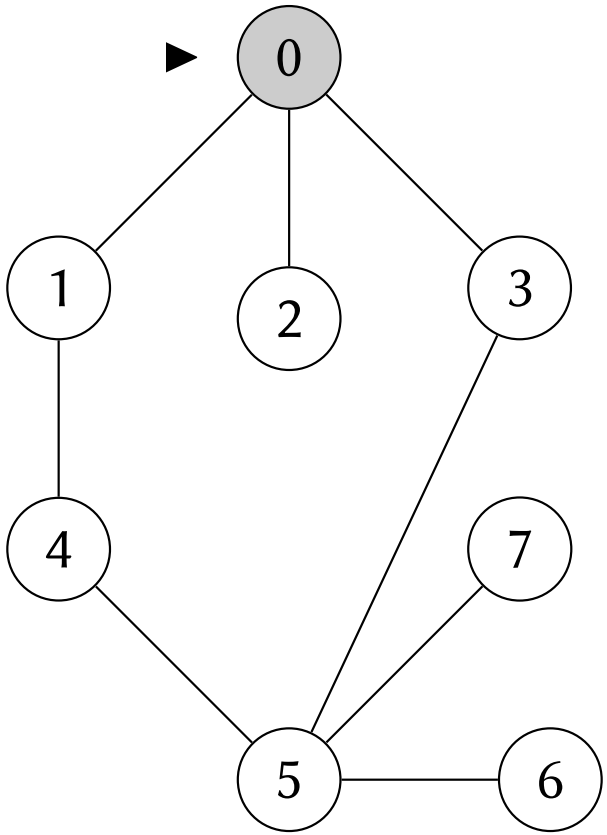
In depth-first search, we visit a node, we note that we have visited it, and we visit the first of its unvisited neighbors. We repeat the process on that neighbor.

Depth-First Example (1)

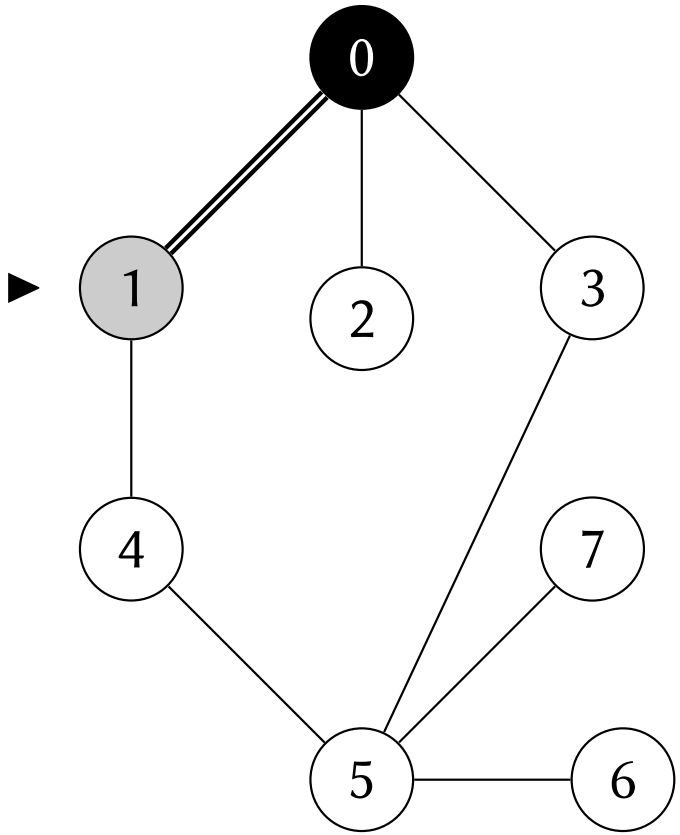




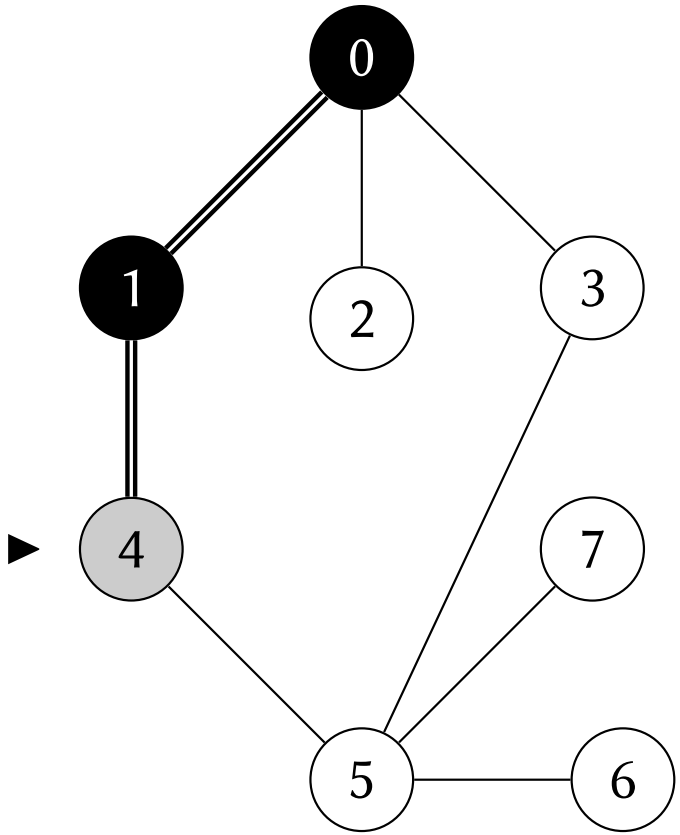
Depth-First Example (2)



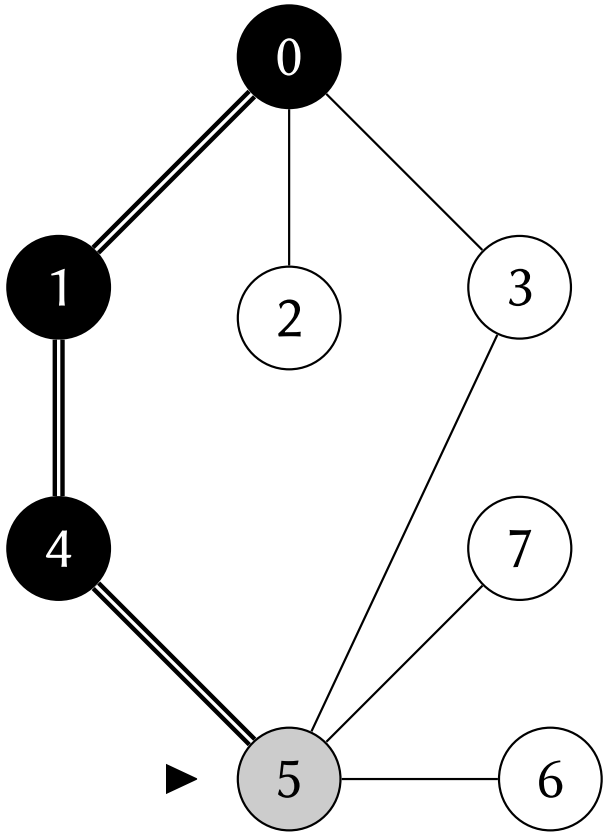
Depth-First Example (3)



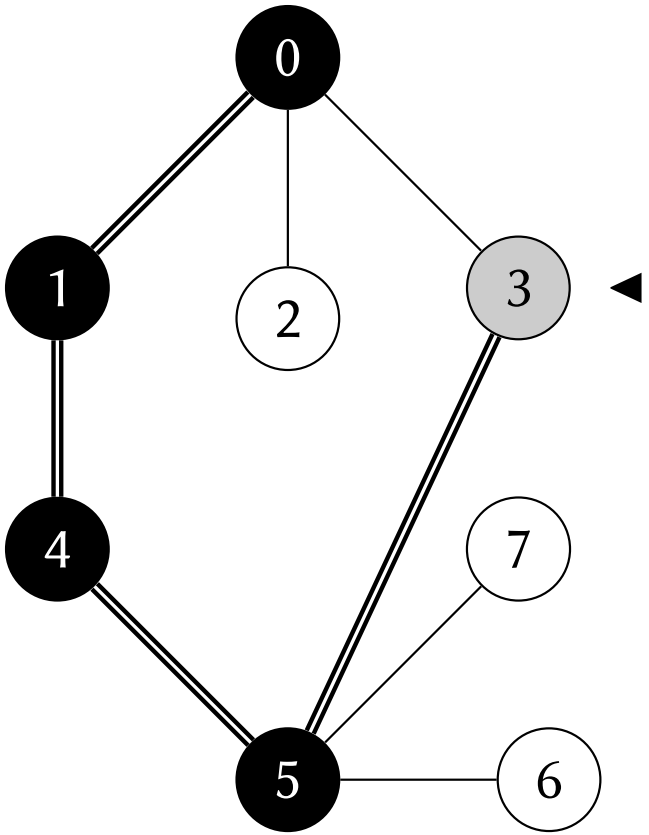
Depth-First Example (4)



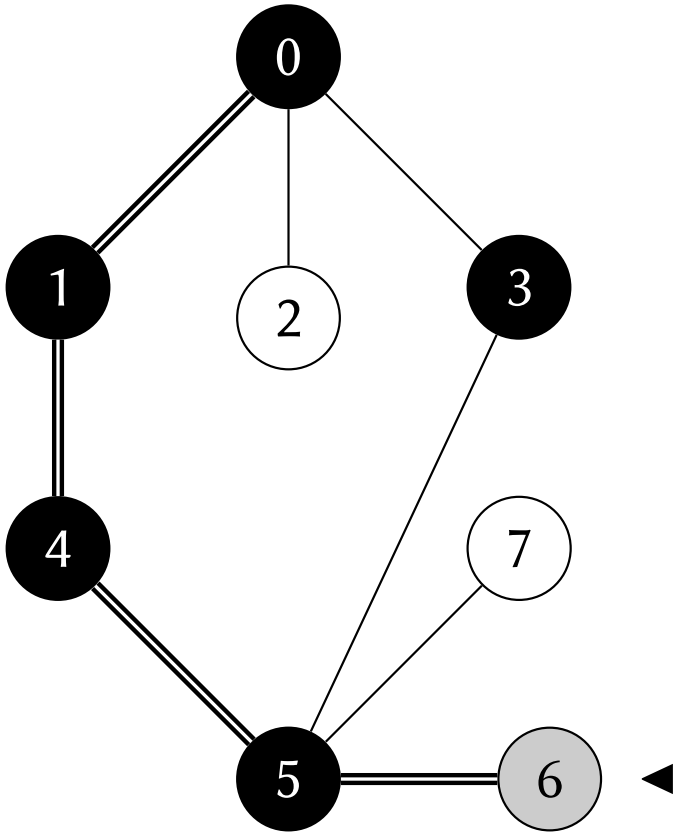
Depth-First Example (5)



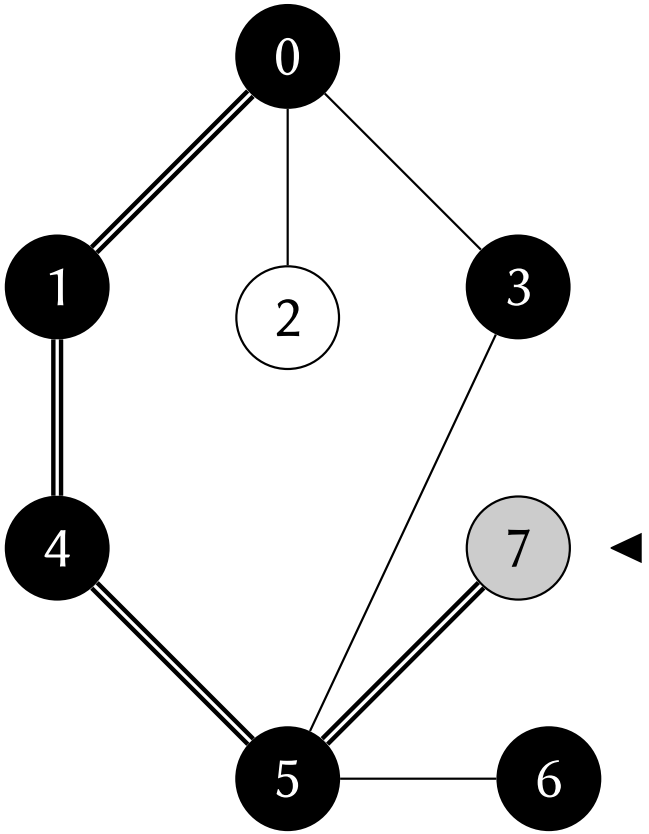
Depth-First Example (6)



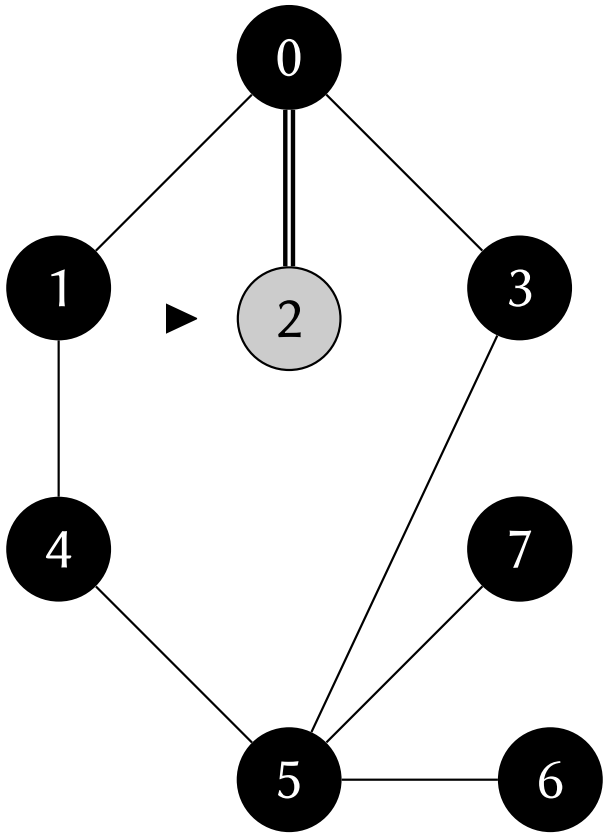
Depth-First Example (7)



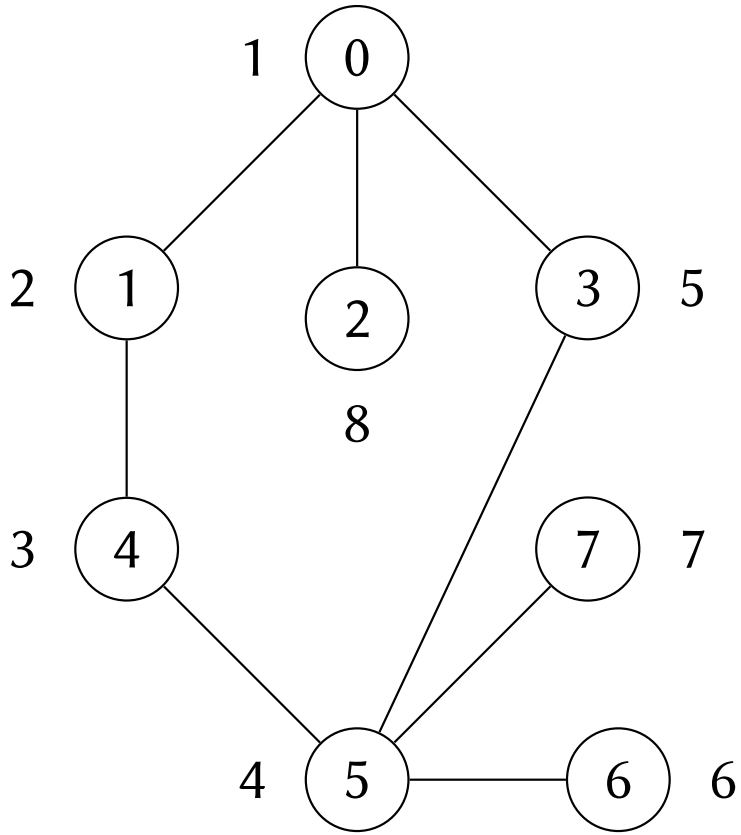
Depth-First Example (8)



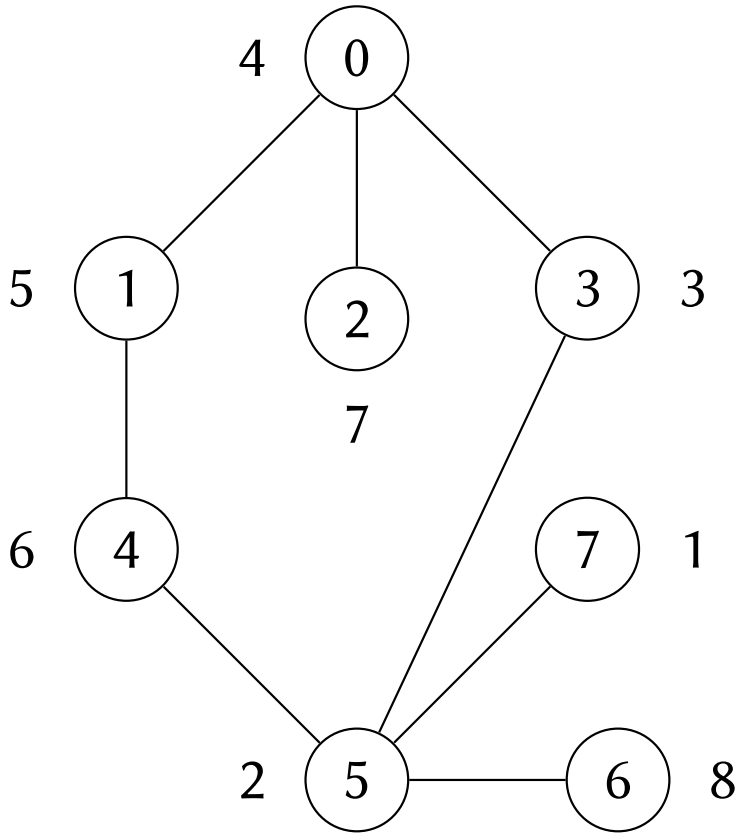
Depth-First Example (9)



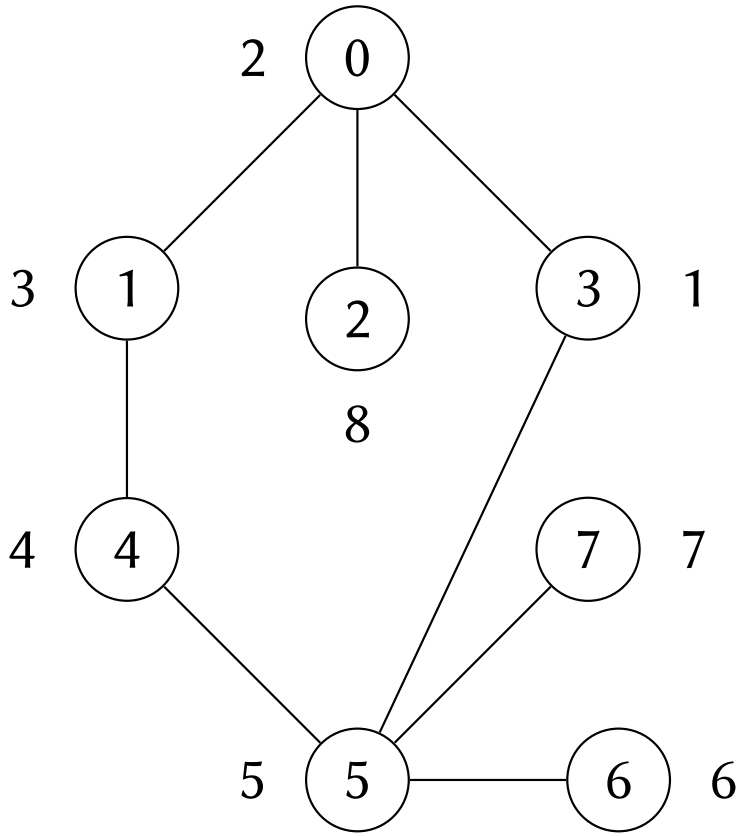
Depth-First Example—Overview



Depth-First Search from Different Starting Nodes



Starting from node 7.



Starting from node 3.

Depth-First Search Algorithm

Algorithm: Recursive graph depth-first search.

DFS(G, *node*)

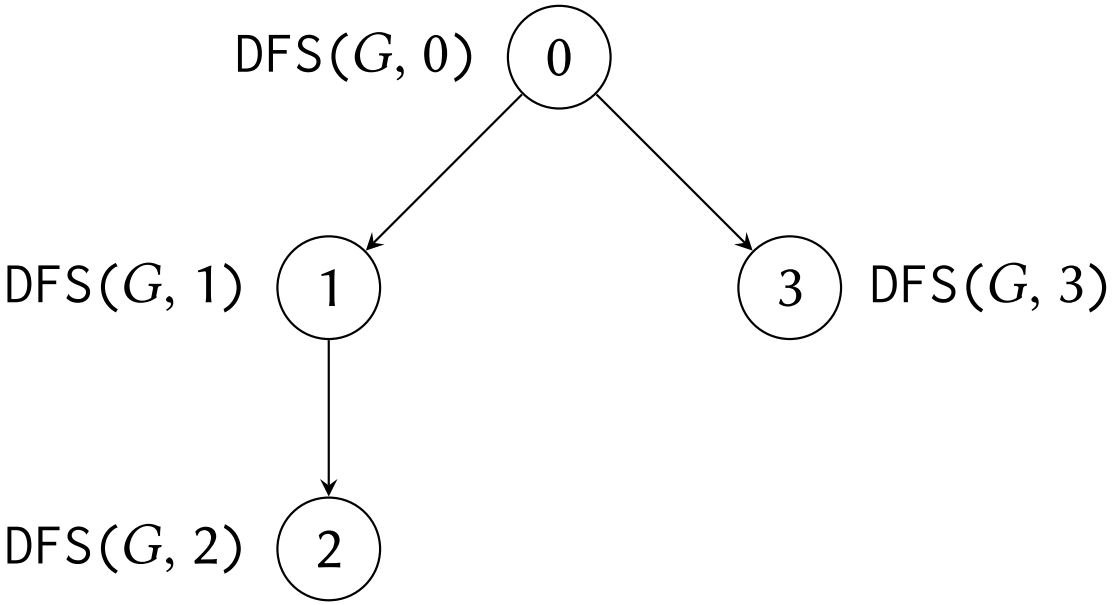
Input: $G = (V, E)$, a graph
node, a node in G

Data: *visited*, an array of size $|V|$

Result: *visited*[i] is TRUE if we have visited node i, FALSE otherwise

```
1  visited[node] ← TRUE
2  foreach v in AdjacencyList(G, node) do
3      if not visited[v] then
4          DFS(G, v)
```

Depth-First Search in Depth



DFS(G, 0)
- DFS(G, 1)
  - DFS(G, 2)
- DFS(G, 3)

Recursion

Algorithm: Factorial function.

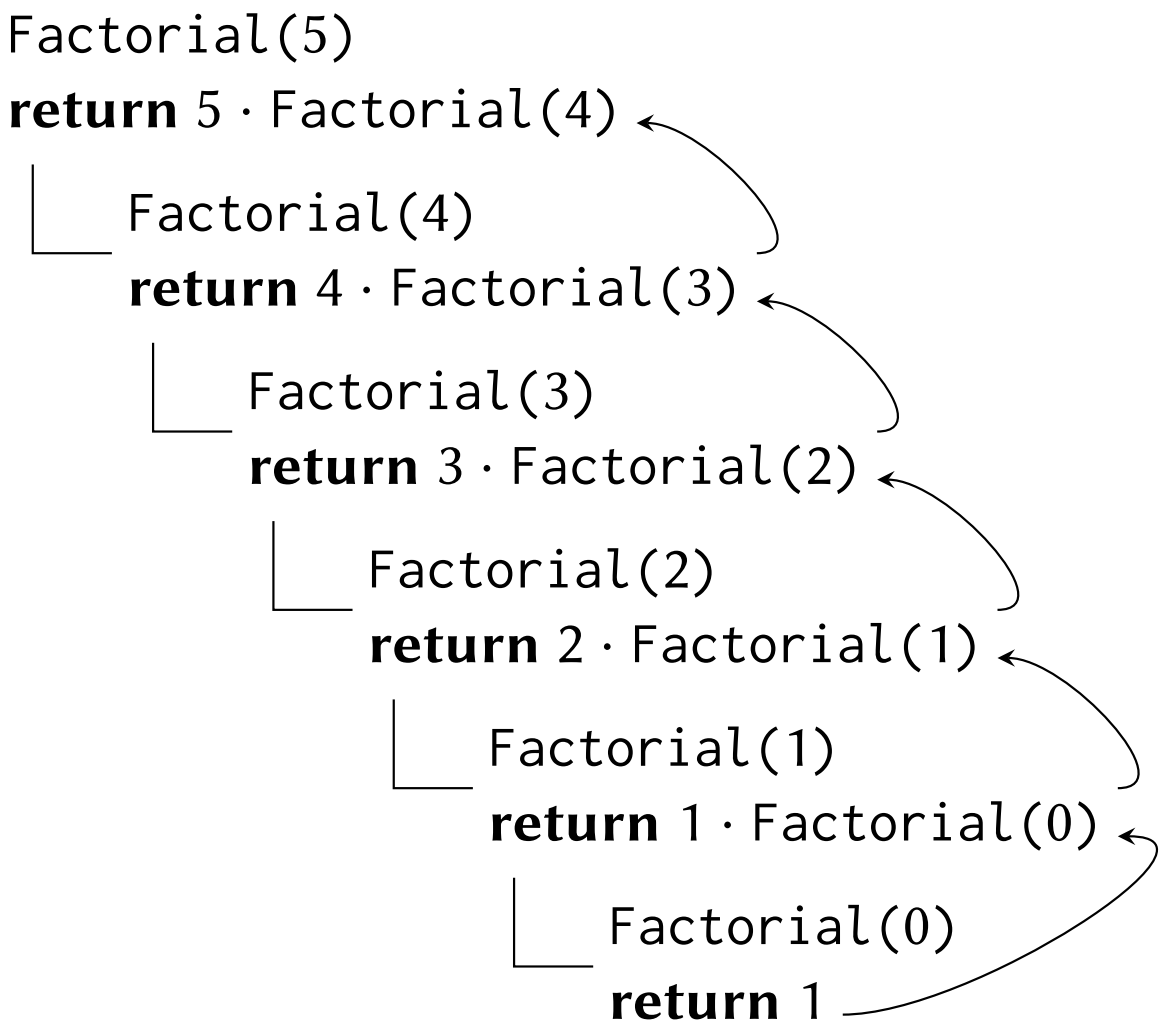
`Factorial`$(n) \longrightarrow !n$

Input: n, a natural number

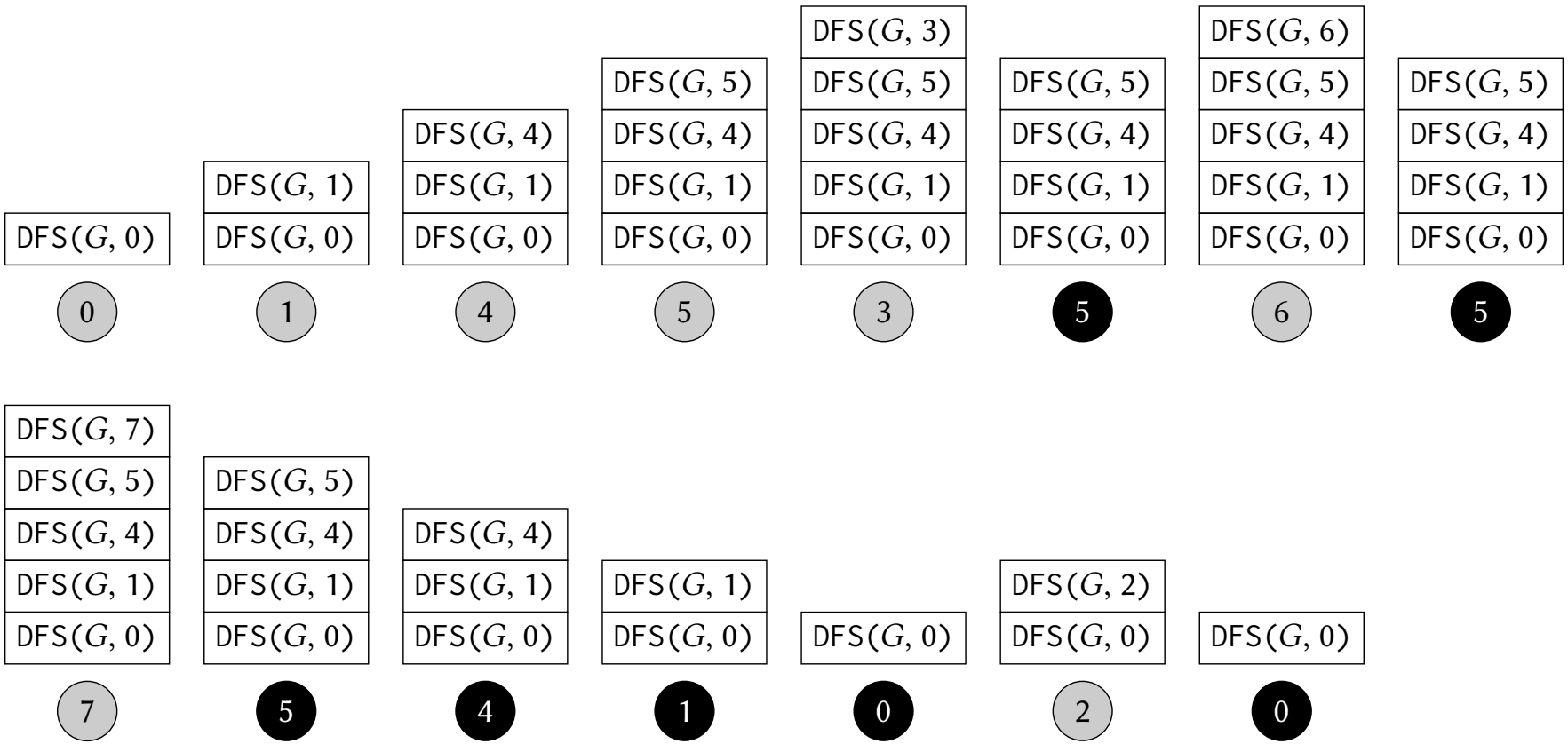
Output: $n!$, the factorial of n

```
1  if n = 0 then
2      return 1
3  else
4      return n · Factorial(n − 1)
```

Factorial Call Trace



Stack Evolution of Depth-First Search



Depth-First Algorithm with a Stack

Algorithm: Graph depth-first search with a stack.

`StackDFS`(G, *node*) $\longrightarrow$ *visited*

Input: $G = (V, E)$, a graph
node, the starting vertex in G

Output: *visited*, an array of size $|V|$ such that *visited*$[i]$ is TRUE if we have visited node i, FALSE otherwise

```
1  S ← CreateStack()
2  visited ← CreateArray(|V|)
3  for i ← 0 to |V| do
4      visited[i] ← FALSE
5  Push(S, node)
6  while not IsStackEmpty(S) do
7      c ← Pop(S)
8      visited[c] ← TRUE
9      foreach v in AdjacencyList(G, c) do
10         if not visited[v] then
11             Push(S, v)
12 return visited
```

Depth-First Search Algorithm

Algorithm: Recursive graph depth-first search.

DFS(G, *node*)

Input: $G = (V, E)$, a graph
node, a node in G

Data: *visited*, an array of size $|V|$

Result: *visited*[i] is TRUE if we have visited node i, FALSE otherwise

```
1  visited[node] ← TRUE
2  foreach v in AdjacencyList(G, node) do
3      if not visited[v] then
4          DFS(G, v)
```

Depth-First Search Complexity

- Line 4 is executed $|V|$ times, once per node.
- Line 3 is executed once per link, so $|E|$ times.
- Therefore the overall complexity is $\Theta(|V| + |E|)$.

Breadth-First Search

Definition

In breadth-first search we visit first all the neighbors of a node, and then we visit their neighbors.

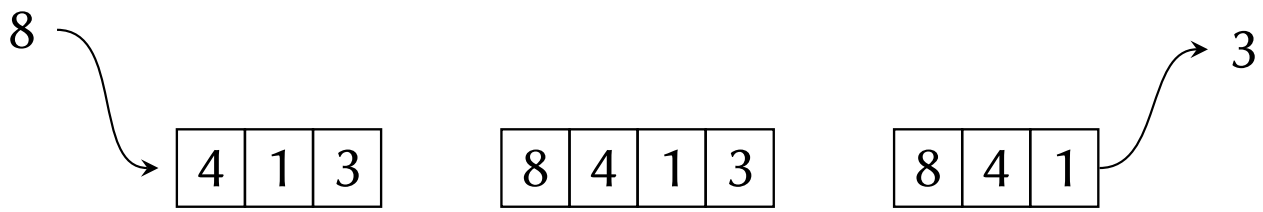
Queues

A queue is a data structure with two ends, a *head* and a *tail*, with the following functions:

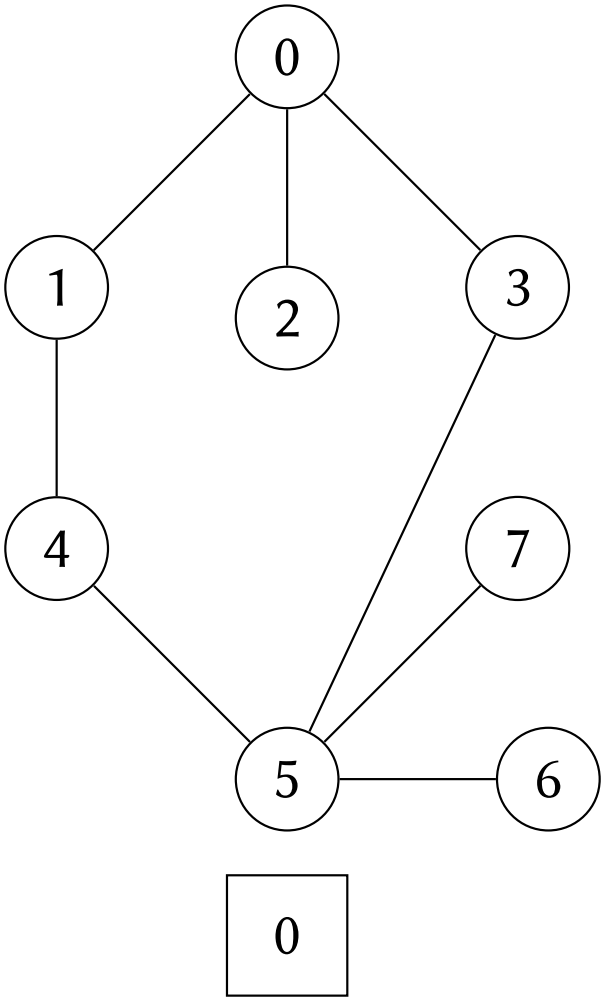
- `CreateQueue()`, creates an empty queue.
- `Enqueue`(Q, i), adds item i at the end of queue Q.
- `Dequeue`(Q) removes an item from the head of queue Q; if it is empty, we get an error.
- `IsQueueEmpty`(Q) returns TRUE if the queue is empty, otherwise it returns FALSE.

A queue is a First In First Out (FIFO) data structure.

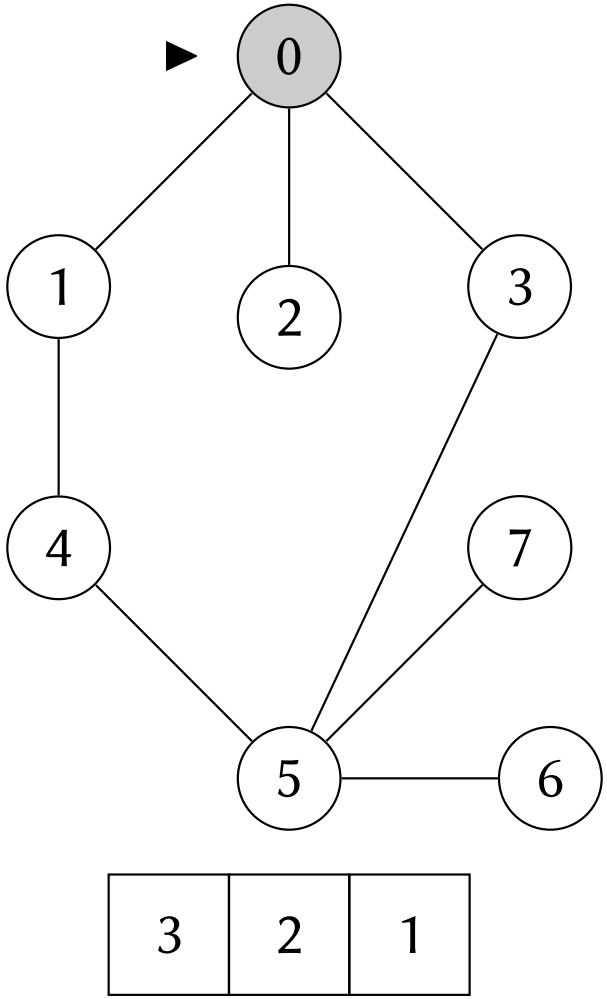
Queue Example



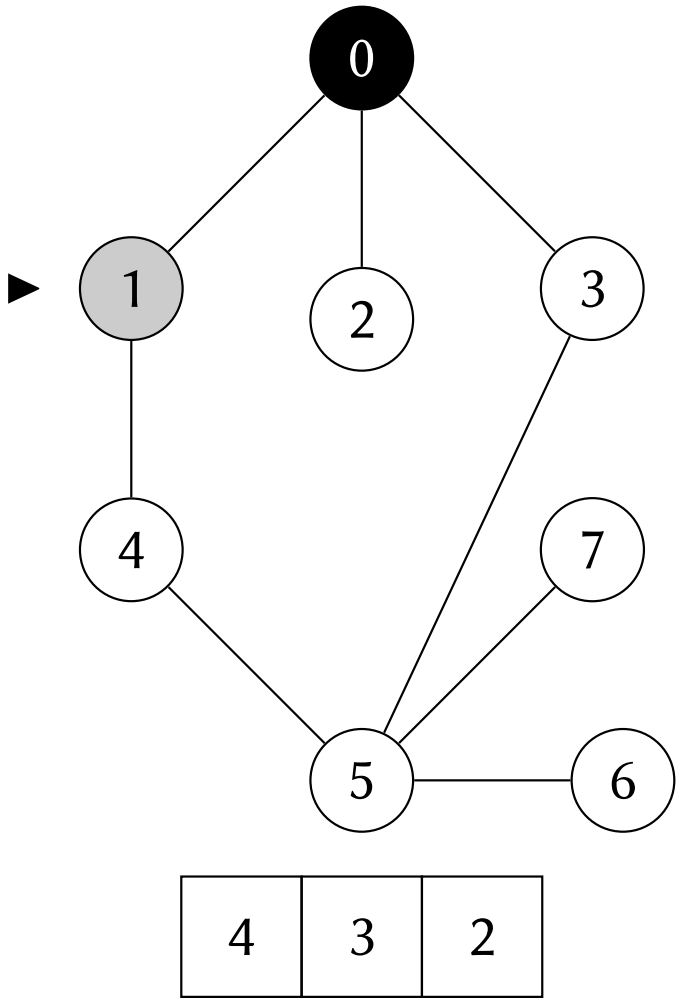
Breadth-First Example (1)



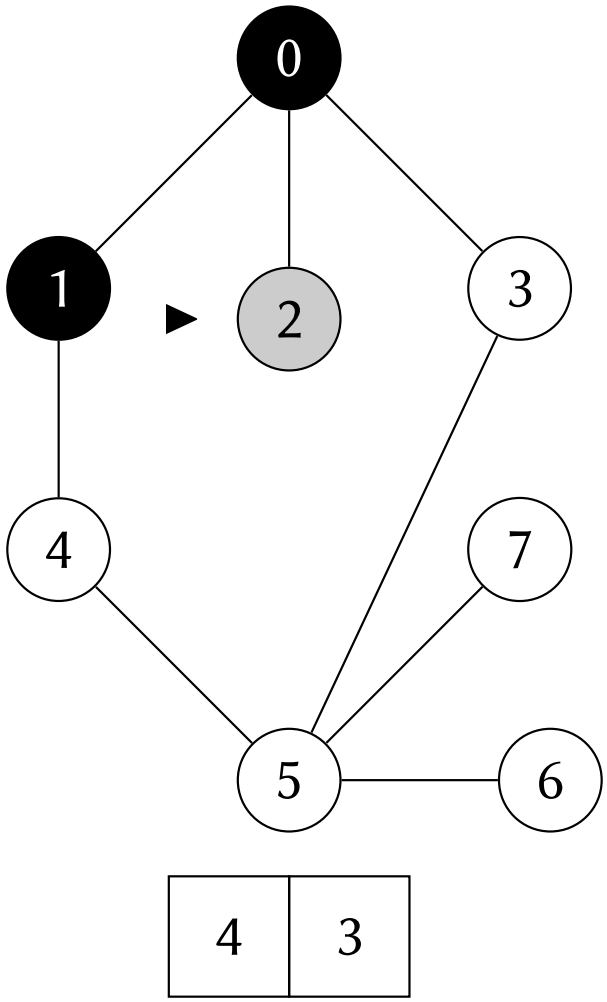
Breadth-First Example (2)



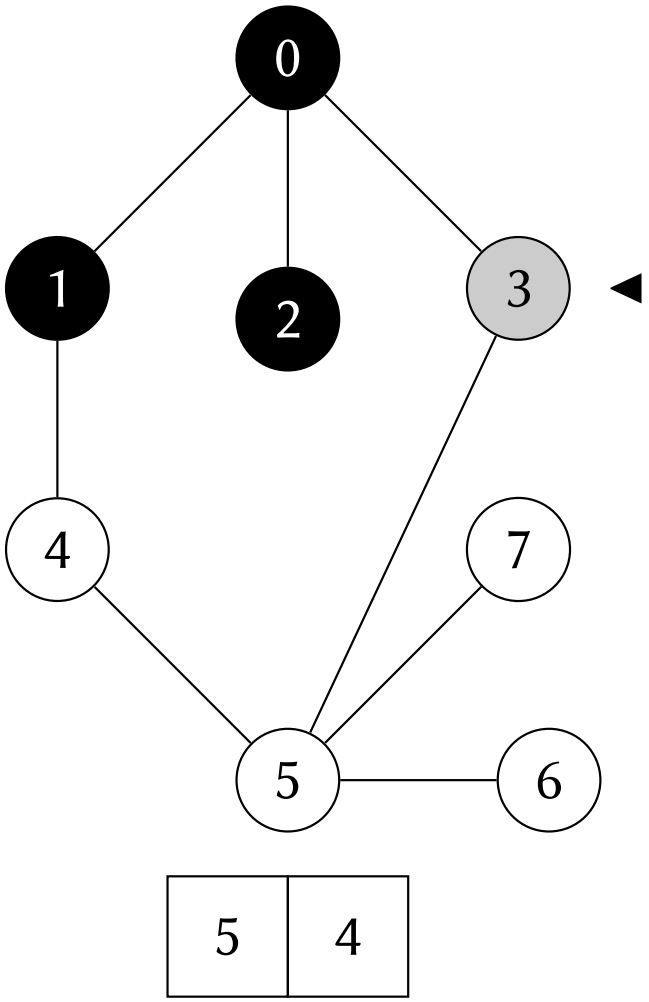
Breadth-First Example (3)



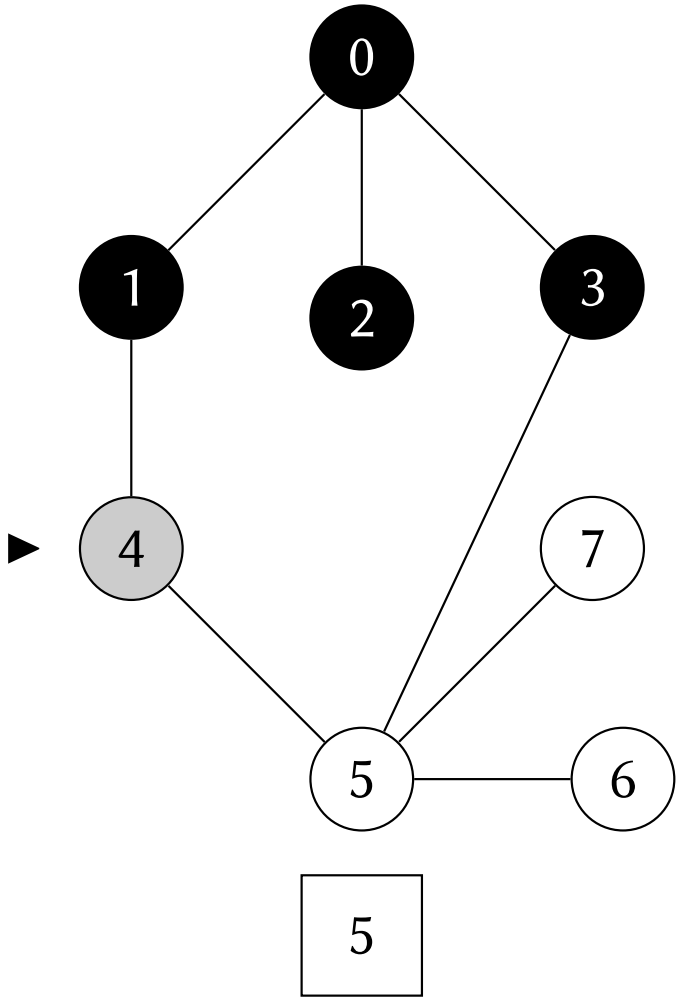
Breadth-First Example (4)



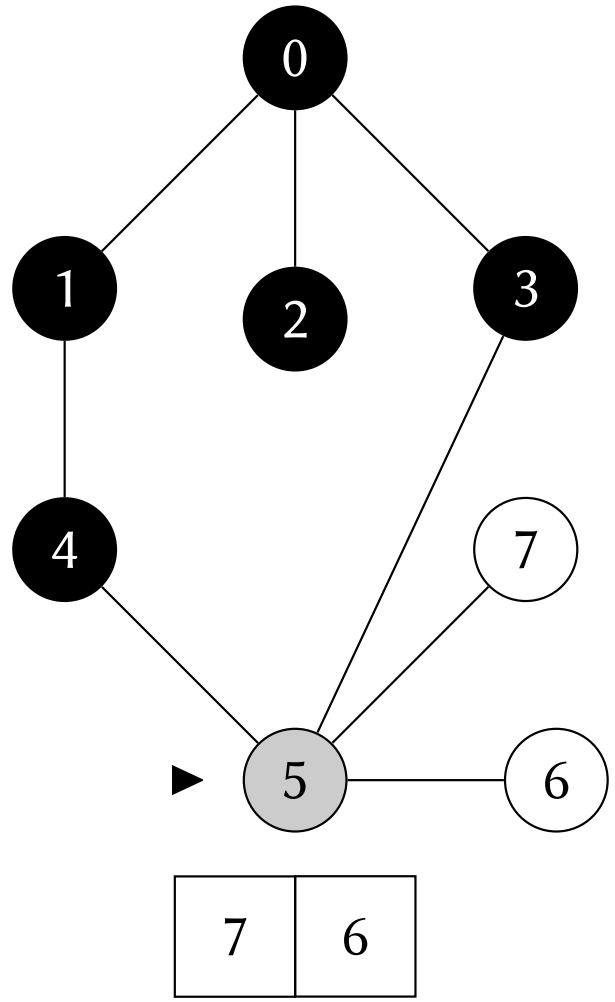
Breadth-First Example (5)



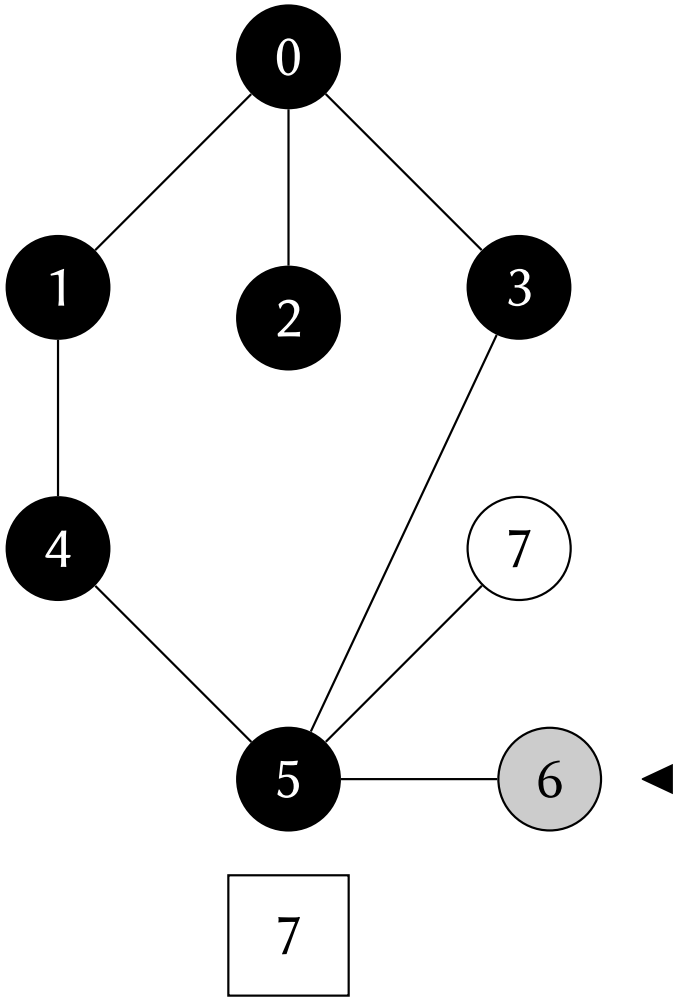
Breadth-First Example (6)



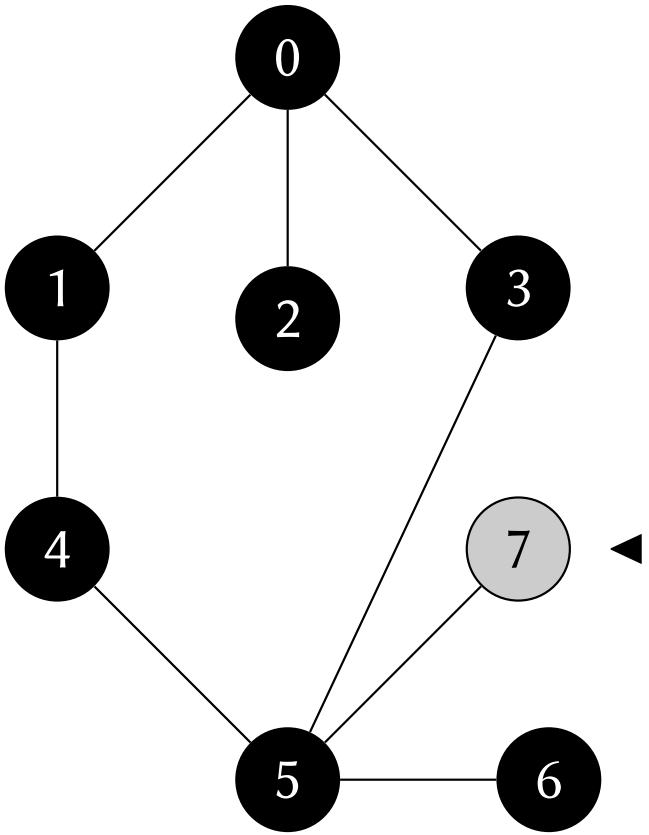
Breadth-First Example (7)



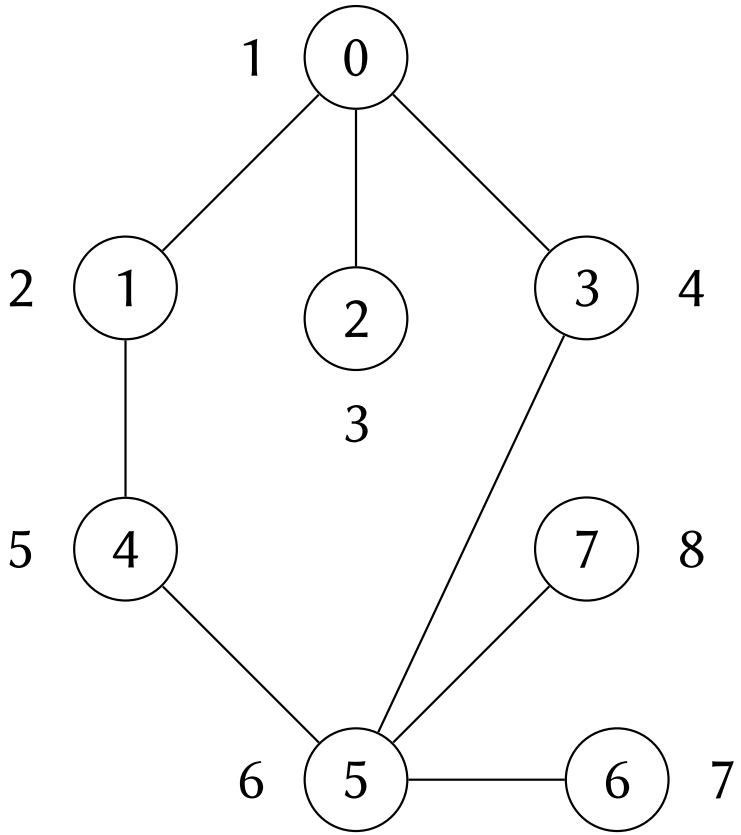
Breadth-First Example (8)



Breadth-First Example (9)



Breadth-First Search Example: Overview



	0		
0:	3	2	1
1:	4	3	2
2:	4	3	
3:	5	4	
4:	5		
5:	7	6	
6:	7		
7:			

Breadth-First Search Algorithm

Algorithm: Graph breadth-first search

BFS(G, *node*) $\rightarrow$ *visited*

Input: $G = (V, E)$, a graph
node, the starting vertex in G

Output: *visited*, an array of size $|V|$ such that *visited*$[i]$ is TRUE if we have visited node i, FALSE otherwise

```
1  Q ← CreateQueue()
2  visited ← CreateArray(|V|)
3  inqueue ← CreateArray(|V|)
4  for i ← 0 to |V| do
5      visited[i] ← FALSE
6      inqueue[i] ← FALSE
7  Enqueue(Q, node)
8  inqueue[node] ← TRUE
9  while not IsQueueEmpty(Q) do
10     c ← Dequeue(Q)
11     inqueue[c] ← FALSE
12     visited[c] ← TRUE
13     foreach v in AdjacencyList(G, c) do
14         if not visited[v] and not inqueue[v] then
15             Enqueue(Q, v)
16             inqueue[v] ← TRUE
17 return visited
```

Breadth-First Search Complexity

- Line 9 is executed $|V|$ times.
- The loop starting at line 13 will be executed once for each link, so $|E|$ times in total.
- Therefore the overall complexity is $\Theta(|V| + |E|)$.