The Graph Data Structure

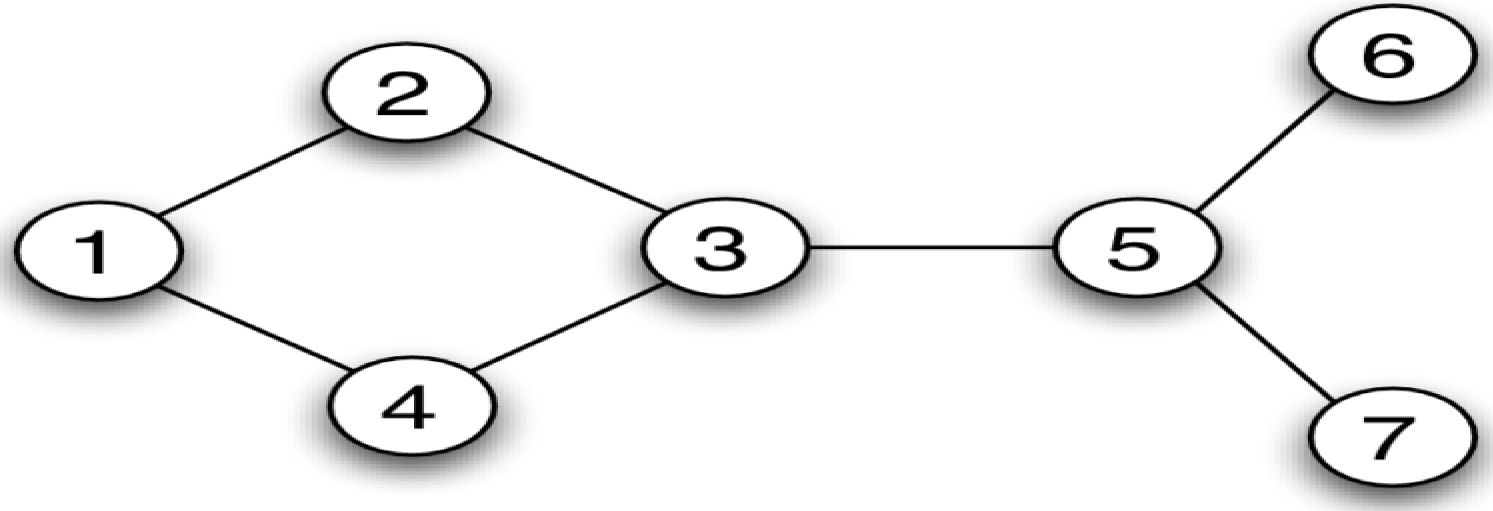
Dr. Amit Praseed

Graphs

- A graph is a data structure that is commonly represented as G = (V, E), where V is a set of vertices and E is a set of edges
- Graphs are commonly used to represent a large number of real world problems
  - Railways, roadways, airline routes, transmission towers etc.
  - Routing traffic over the Internet
  - Representing game outcomes
  - Representing a problem search space

Graph Terminology

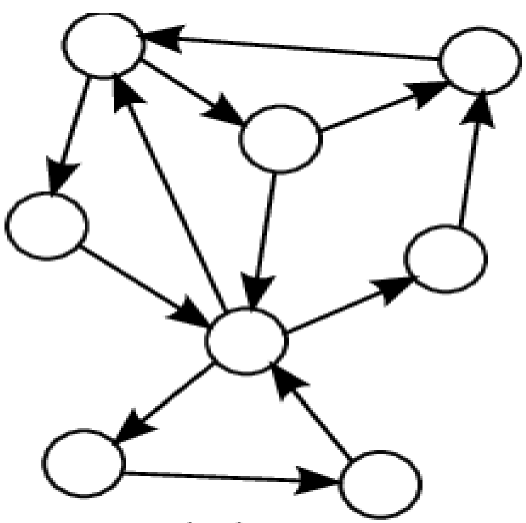
- A graph is a collection of vertices, V and a collection of edges, E
- Every edge $e \in E$ can be represented as $\{u, v\}$ where $u, v \in V$



Graph Terminology

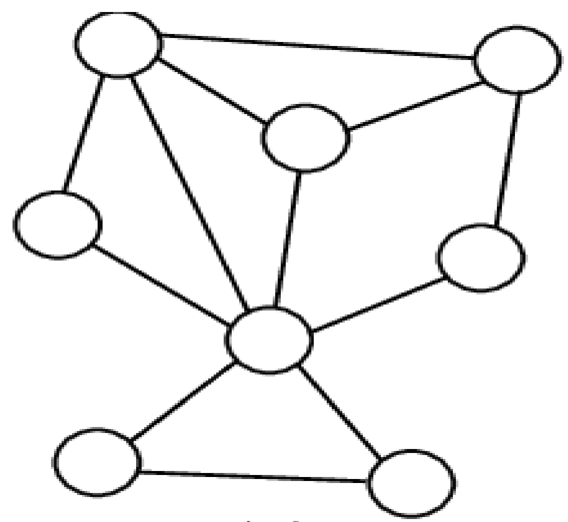
Directed Graphs

- A graph in which the edges are represented as **ordered pairs (u, v)** are called directed graphs
- Edges have direction



Undirected Graphs

- A graph in which the edges are represented as **unordered pairs (u, v)** are called undirected graphs
- Edges do not have direction



Graph Terminology

- Graphs without loops and parallel edges are often called simple graphs; non-simple graphs are sometimes called multigraphs
- For any edge *u-v* in an undirected graph, we call u a neighbor of v and vice versa, and we say that u and v are adjacent.
- The degree of a node is its number of neighbors.
- For any directed edge *u-v*, we call u a predecessor of v, and we call v a successor of u. The in-degree of a vertex is its number of predecessors; the out-degree is its number of successors

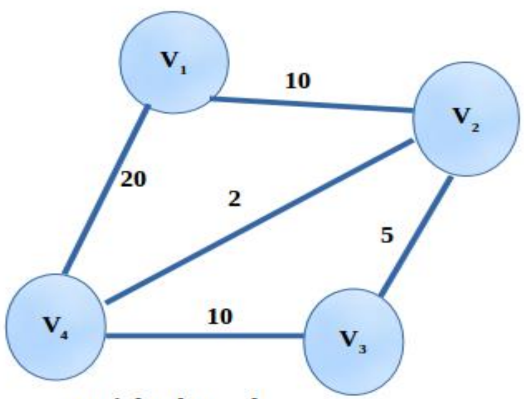
Graph Terminology

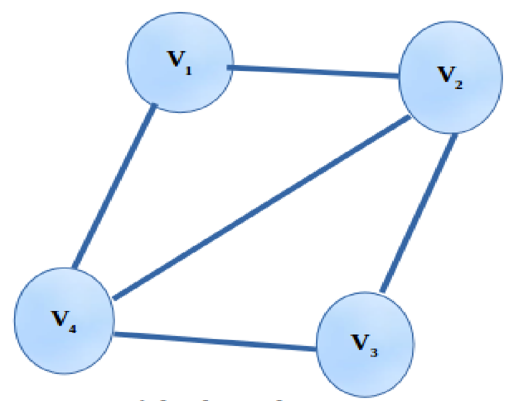
Weighted Graphs

- A graph in which the edges is assigned a numeric value (called weight) are called weighted graphs
- Can represent path length, delay etc.

Undirected Graphs

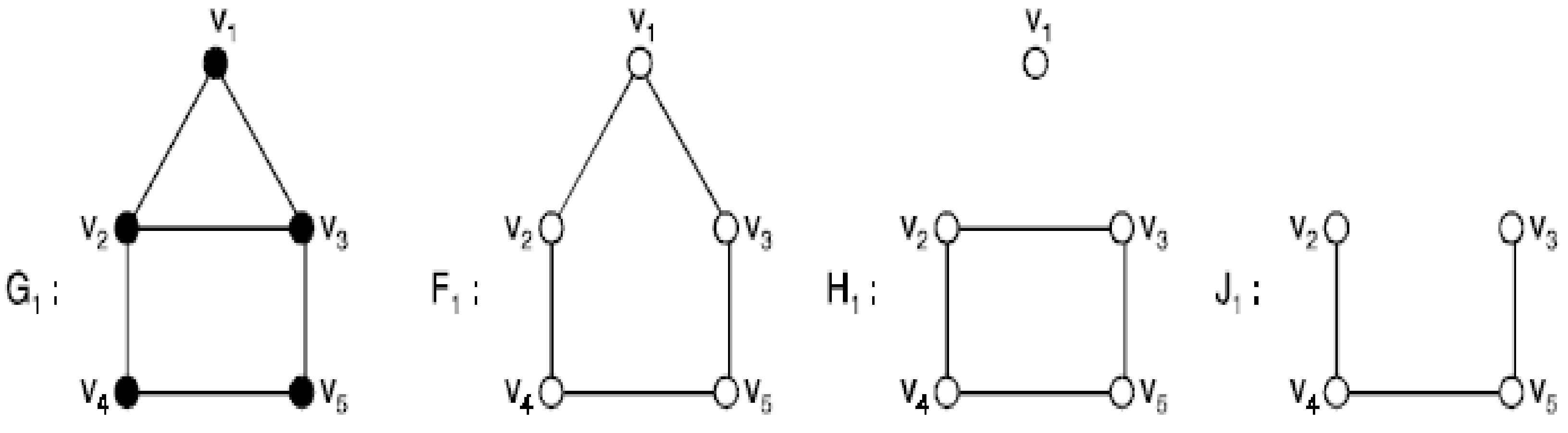
- A graph in which the edges do not have any numeric value associated with them are called unweighted graphs





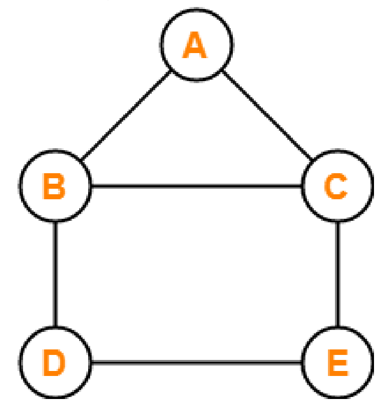
Graph Terminology

- A graph $G' = (V', E')$ is a subgraph of G = (V, E) if $V' \subseteq V$ and $\mathrm{E}' \subseteq E$
  - By definition, G is a subgraph of itself
- A proper subgraph of G is any subgraph other than G itself.



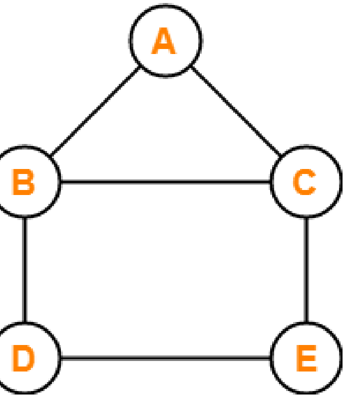
Walks, Trails, Paths

- A walk is a sequence of vertices, where each adjacent pair of vertices are adjacent in G
  - A vertex can be traversed more than once
  - An edge can be used more than once
- Eg: d b a c e d e c is a walk of length 7
- If a walk starts and ends at the same vertex it is called a closed walk
  - Otherwise it is called an open walk



Walks, Trails, Paths

- A path is a walk in which each vertex is visited at most once
  - A vertex cannot be traversed more than once
  - Eg: a b c e d is a path of length 4
- A trail is a walk in which an edge can be trav most once
  - A vertex can be visited more than once
  - An edge can only be visited once

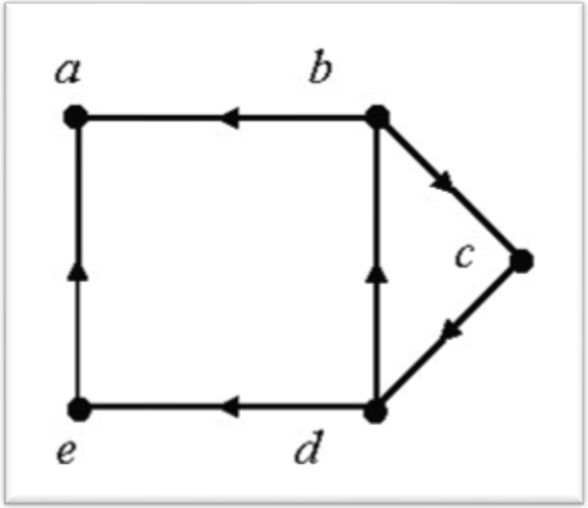


Connected Graphs

- Two vertices u and v in a graph G, v is said to be reachable from u if there exists a path between u and v.
- An undirected graph is connected if every vertex is reachable from every other vertex.
- Every undirected graph consists of one or more components, which are its maximal connected subgraphs; two vertices are in the same component if and only if there is a path between them

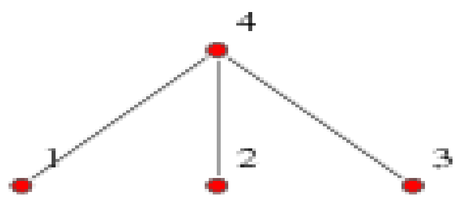
Connected Graphs

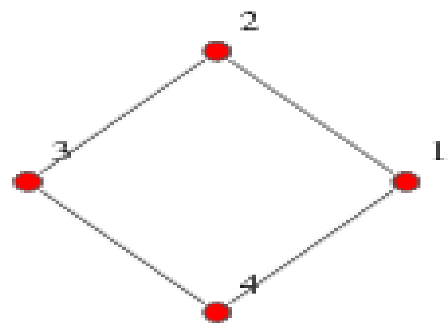
- A directed graph G is said to be strongly connected if there exists a path between any two vertices u and v
  - Note that the path needs to be a directed path
- The given graph is not strongly connected (why?)
- A directed graph G is said to be weakly connected if the graph is not strongly connected, but the underlying undirected graph is connected

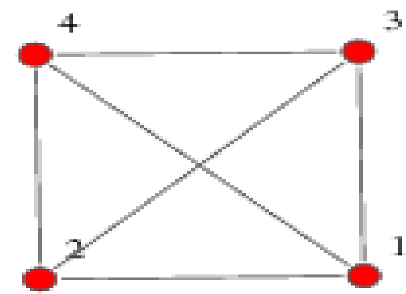


Adjacency Matrix

- An adjacency matrix is a |V| x |V| matrix
  - A[i, j] = 1 if there is an edge from vertex u to vertex v
  - Otherwise, A[i, j] = 0







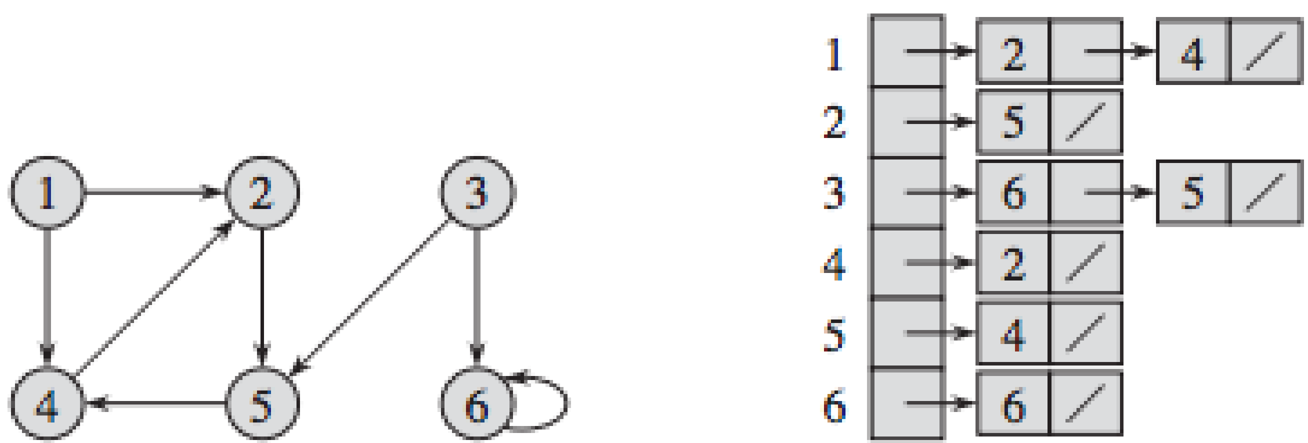
$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency Matrix

- For undirected graphs, the adjacency matrix is always symmetric, meaning A[u, v] = A[v, u] for all vertices u and v and the diagonal entries A[u, u] are all zeros.
- For directed graphs, the adjacency matrix may or may not be symmetric, and the diagonal entries may or may not be zero.
- Given an adjacency matrix
  - We can decide in $\theta(1)$ time whether two vertices are connected by an edge
  - We can also list all the neighbors of a vertex in $\theta(V)$ time.
  - Adjacency Matrices require $\theta(V^2)$ space, regardless of how many edges the graph actually has
- Wastage of space and time for sparse graphs

Adjacency List

- An adjacency list is an array of lists, each containing the neighbors of one of the vertices
  – For undirected graphs, each edge u-v is stored twice; for directed graphs, each edge u-v is stored only once



(a) (b)

Adjacency List

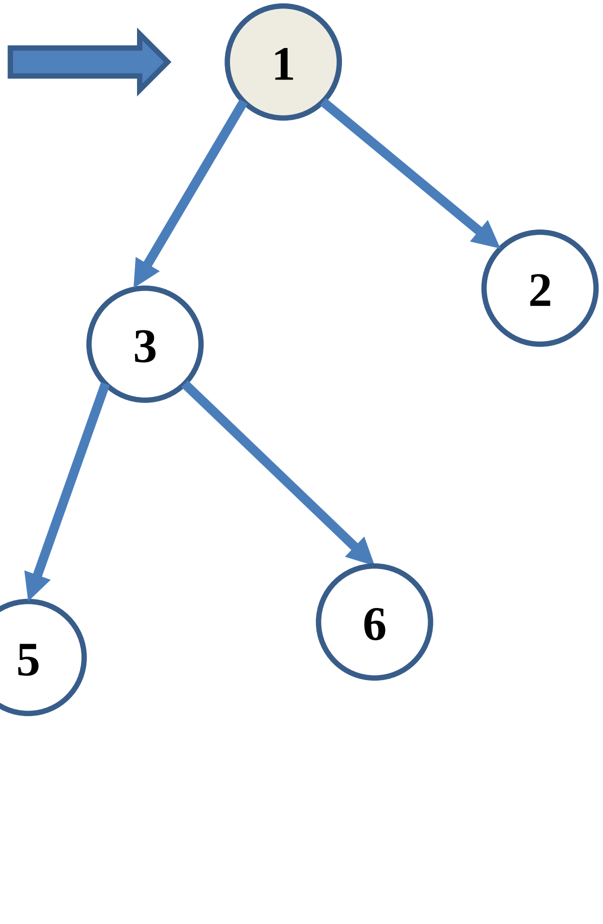
- Given an adjacency list
  - We can list the neighbors of a node v in O(1 + deg(v)) time
  - We can determine whether u-v is an edge in O(1 + deg(u)) time
  - We can traverse the graph in O(V + E) time
  - Adjacency lists require a space complexity of O(V + E)
- Adjacency lists are usually preferred for representing sparse graphs, but dense graphs can be more efficiently represented using adjacency matrices

Depth First Search

- Depth First Search (DFS) is a graph traversal algorithm
- As the name suggests, DFS explores one path in a graph completely before exploring a new one
- When DFS hits a dead end on a path, it backtracks and starts exploring a new path from the previous node
- This behaviour is suggestive of a LIFO data structure – STACK
- DFS can be implemented easily with recursion which uses an implicit stack

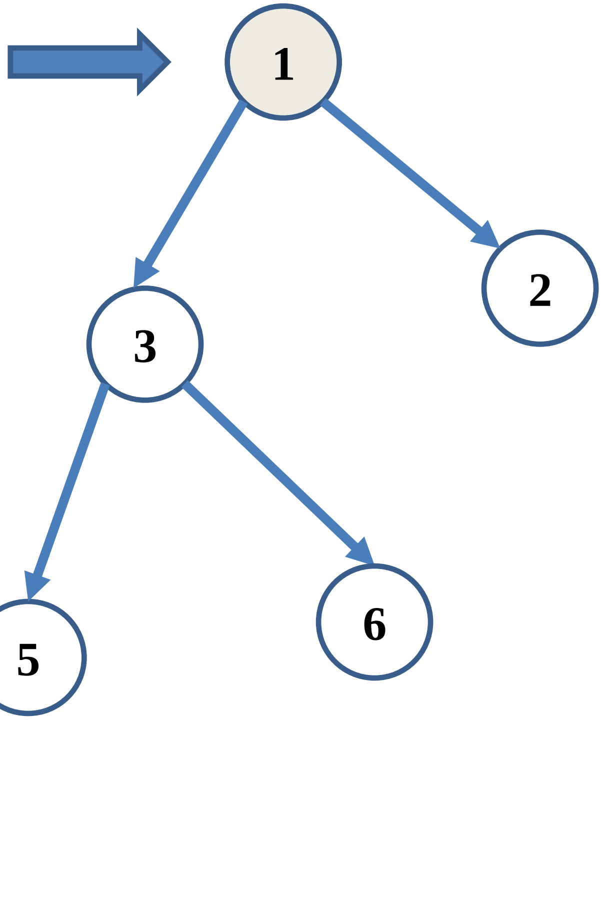
Depth First Search

DFS is a graph traversal algorithm



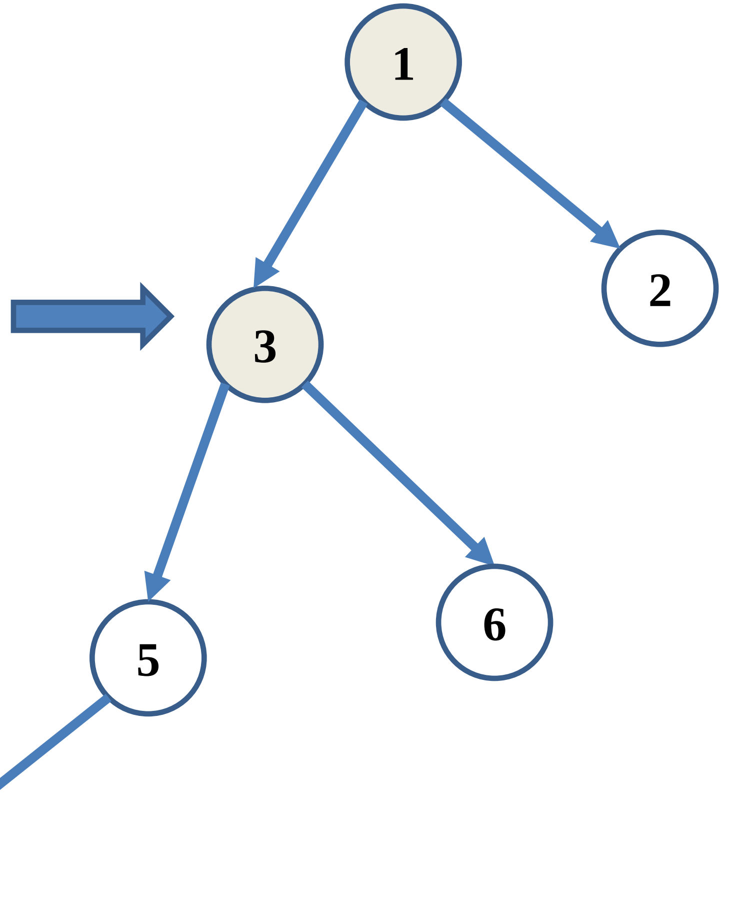
Depth First Search

DFS explores one path completely before moving on to another



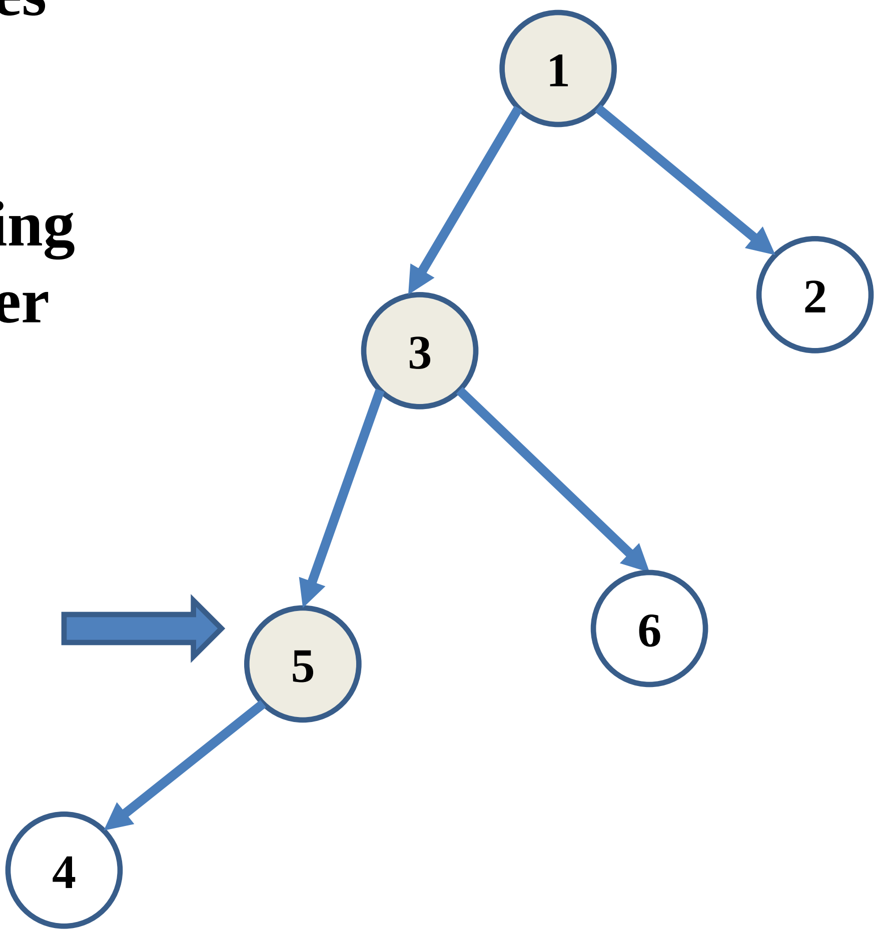
Depth First Search

DFS explores one path completely before moving on to another



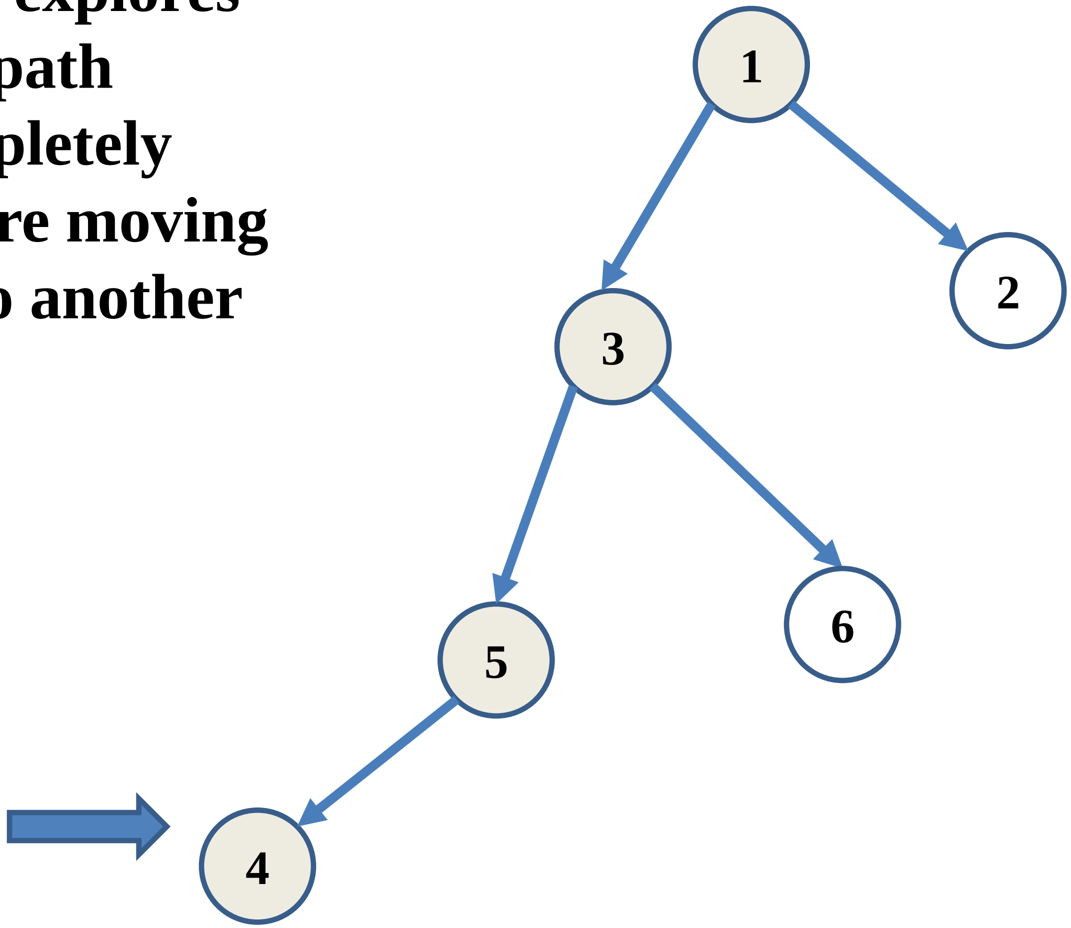
Depth First Search

DFS explores one path completely before moving on to another



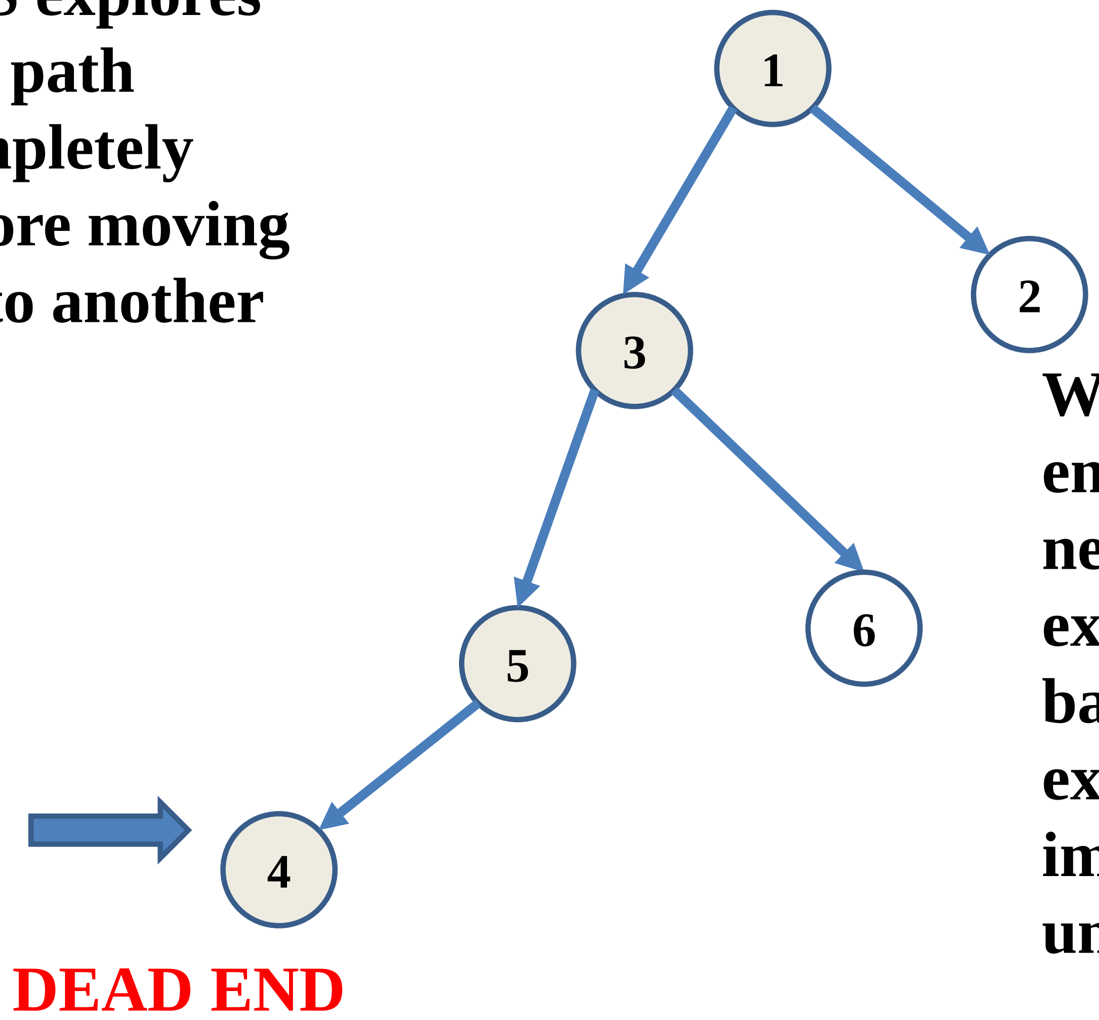
Depth First Search

DFS explores one path completely before moving on to another



Depth First Search

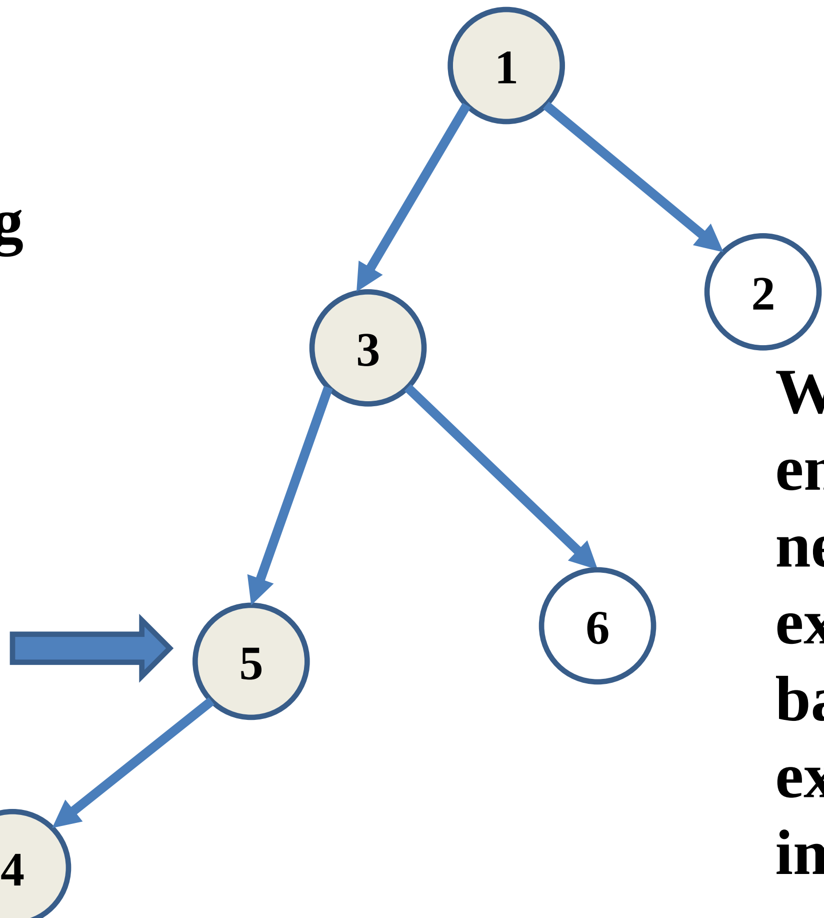
DFS explores one path completely before moving on to another



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

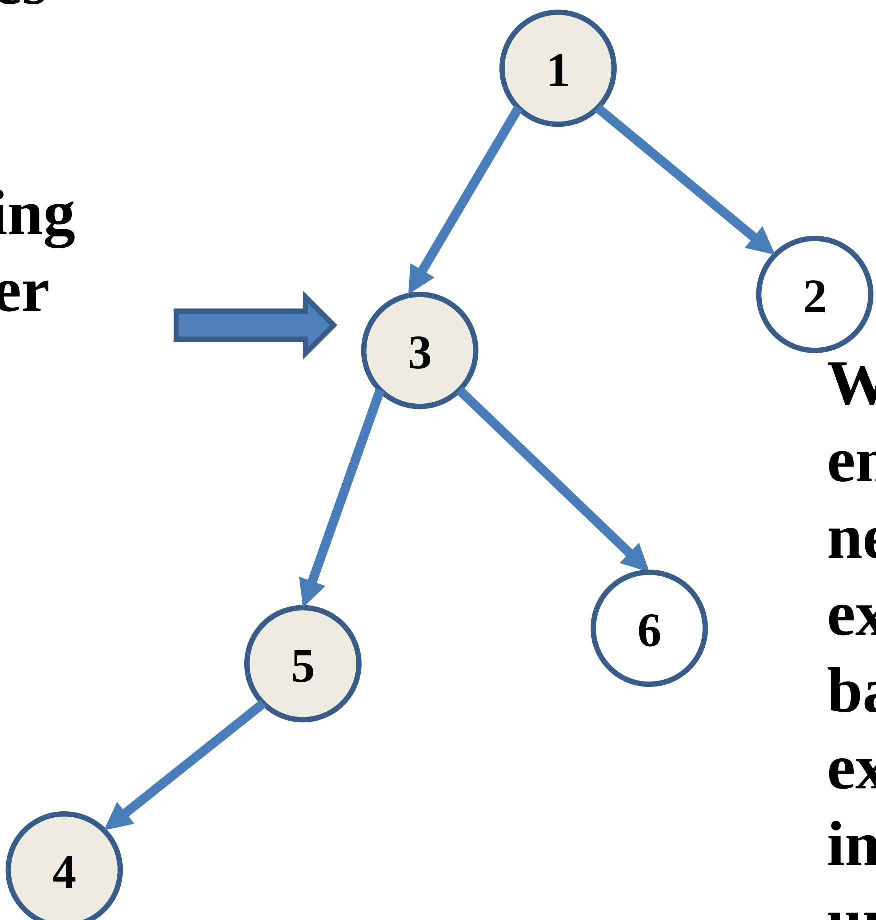
DFS explores one path completely before moving on to another



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

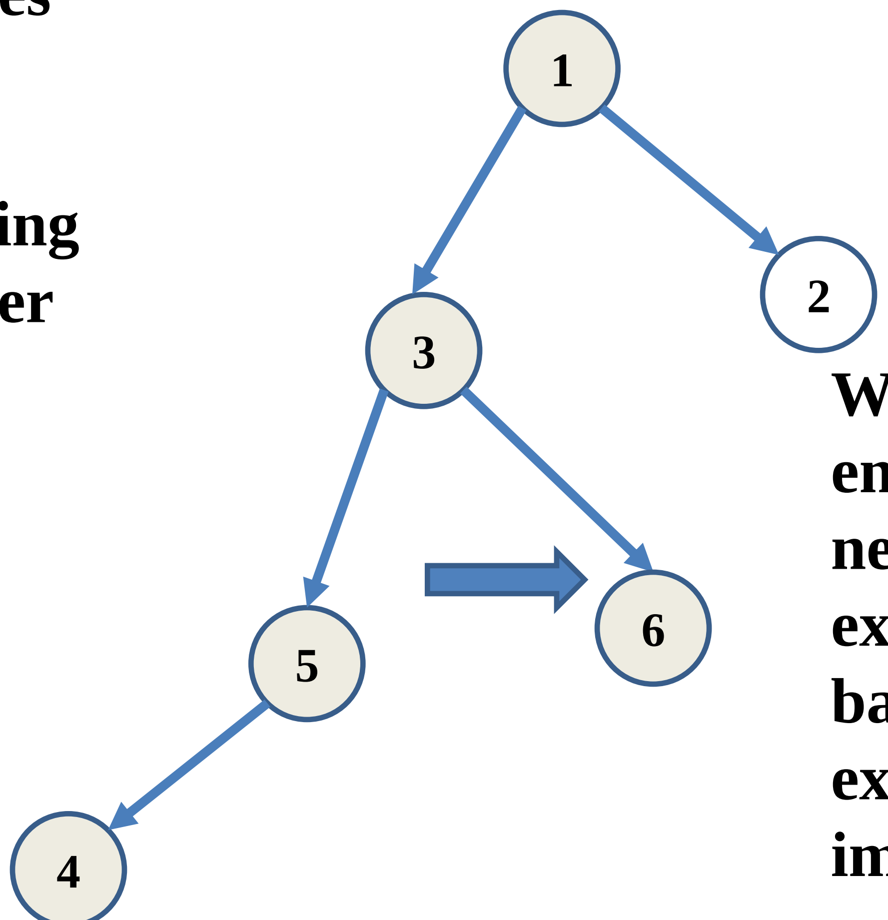
DFS explores one path completely before moving on to another



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

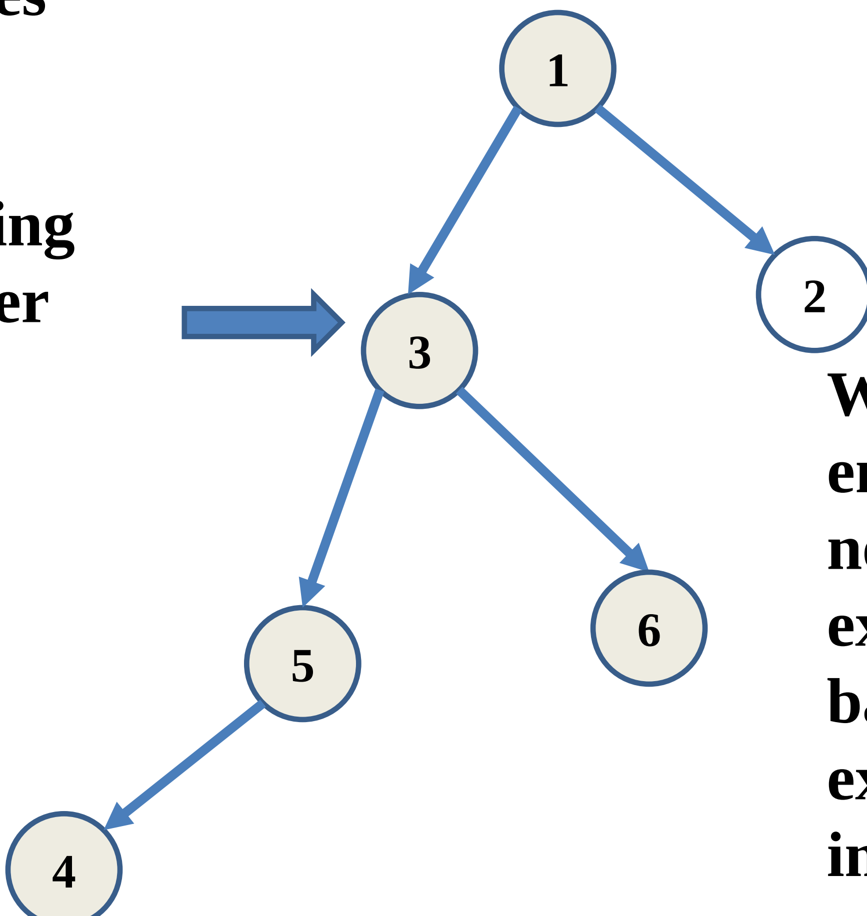
DFS explores one path completely before moving on to another



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

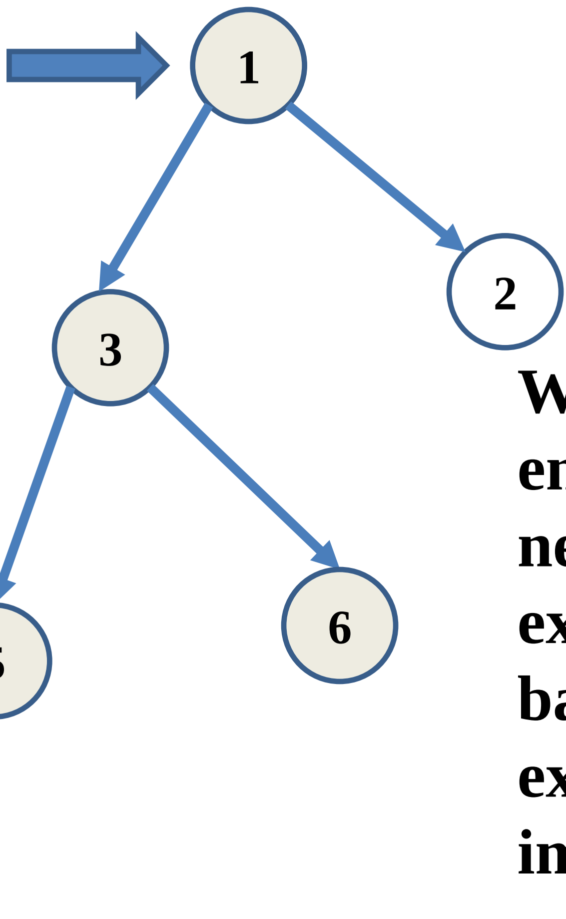
DFS explores one path completely before moving on to another



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

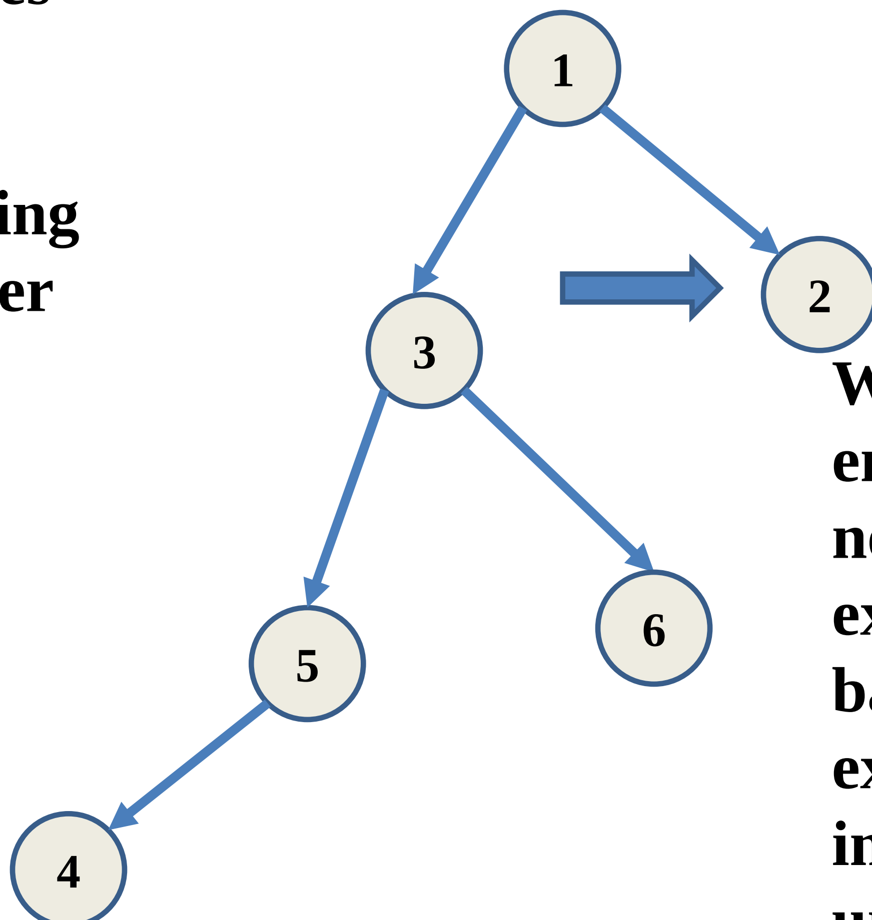
DFS explores one path completely before moving on to another



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

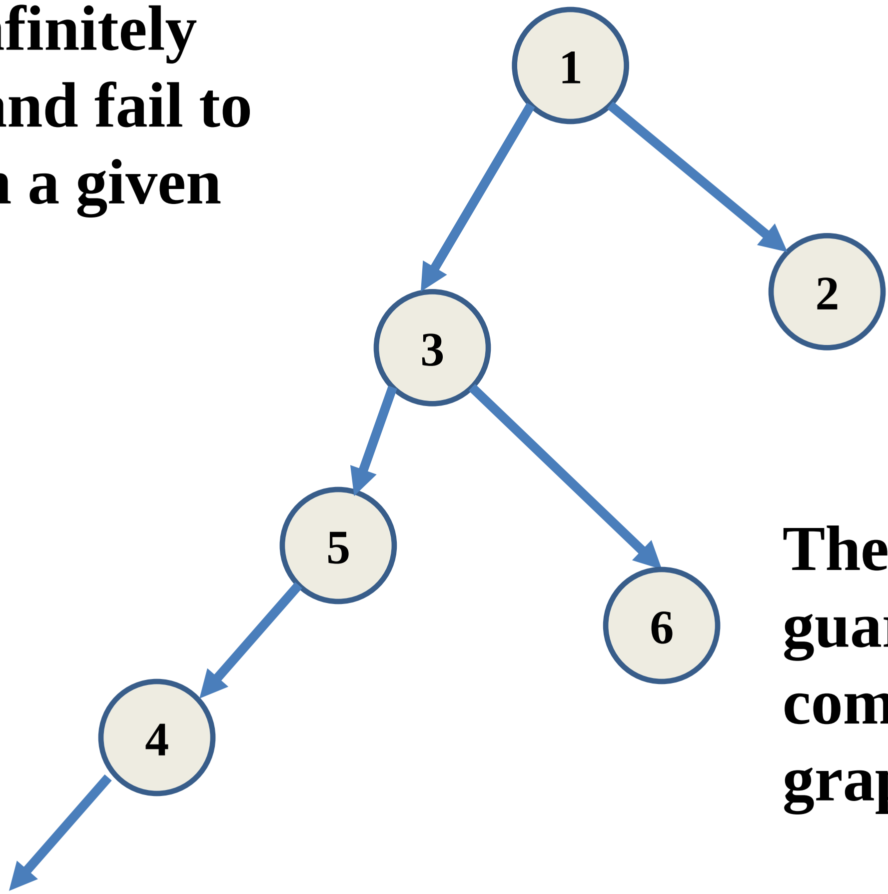
DFS explores one path completely before moving on to another



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Completeness of DFS

DFS could get stuck exploring infinitely long paths and fail to terminate in a given time

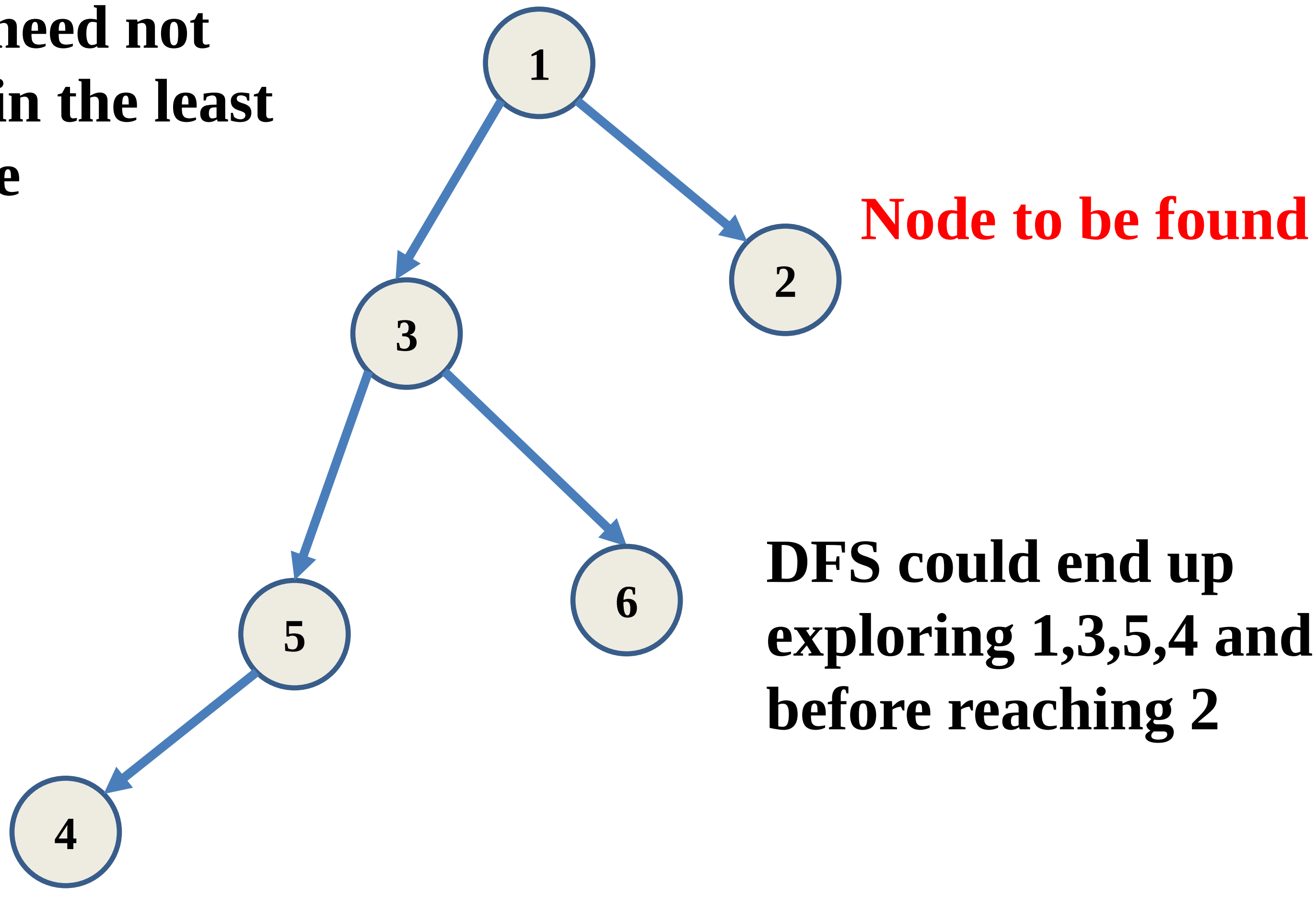


The DFS algorithm is guaranteed to be complete for finite graphs

Potentially continues to ∞

Optimality of DFS

DFS algorithm is not optimal - it need not find a node in the least possible time



DFS could end up exploring 1,3,5,4 and 6 before reaching 2

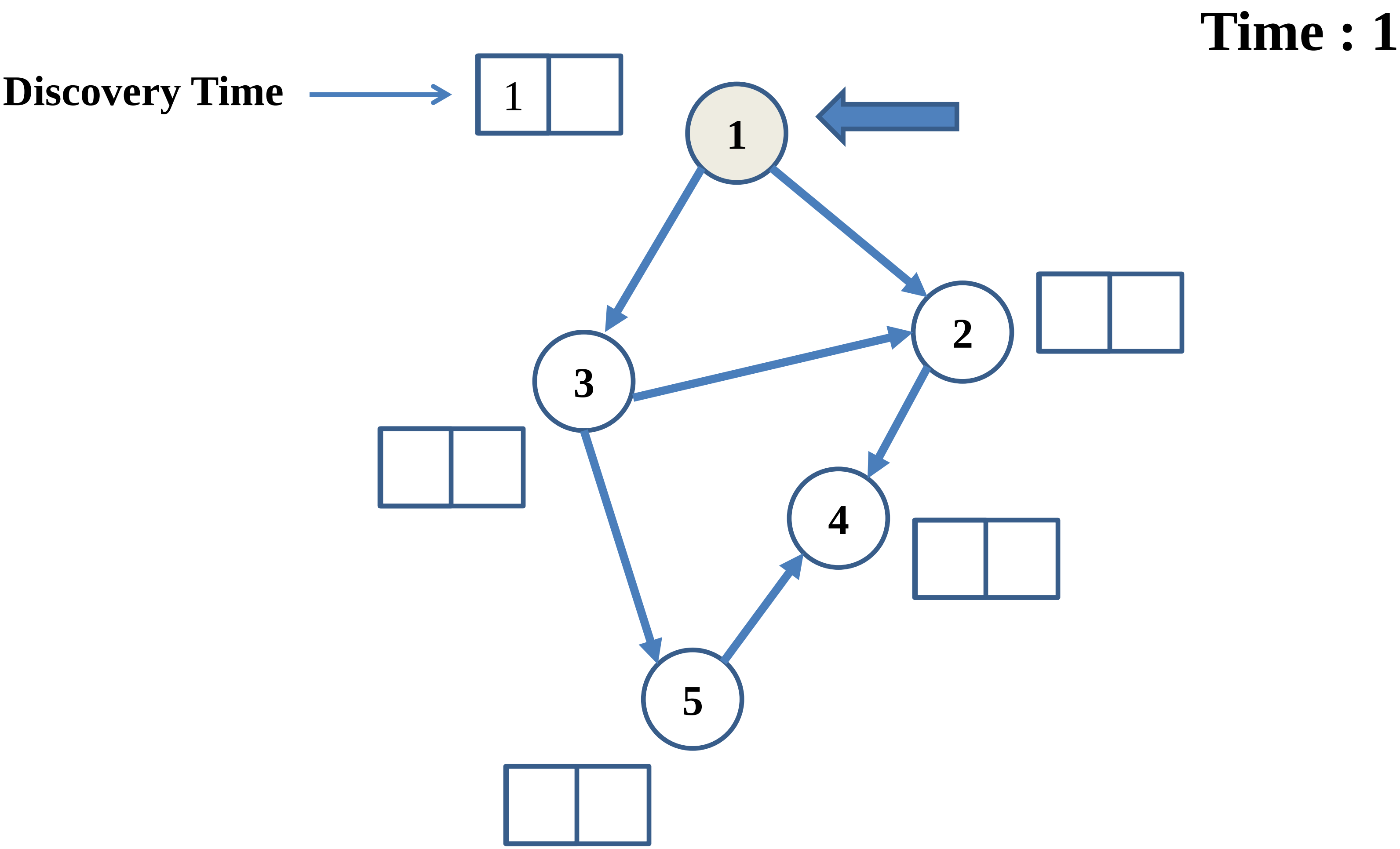
Applications of DFS

- Cycle detection
- Path Detection
- Finding strongly connected components
- Job Scheduling with dependencies (Topological Sort)

Associated Notations

- **Nodes are assigned colours**
  - WHITE : The node has not been visited
  - GRAY : The node has been visited, but all of its branches have not been visited completely
  - BLACK : A node and its branches have been explored completely
- Often, two other values are associated with a node
  - **Discovery Time (d)** : The time at which the node becomes gray
  - **Finishing Time (f)** : The time at which the node becomes black
- **Predecessor Subgraph** : Each time a node is visited, its parent is noted to construct the **DFS tree / forest**

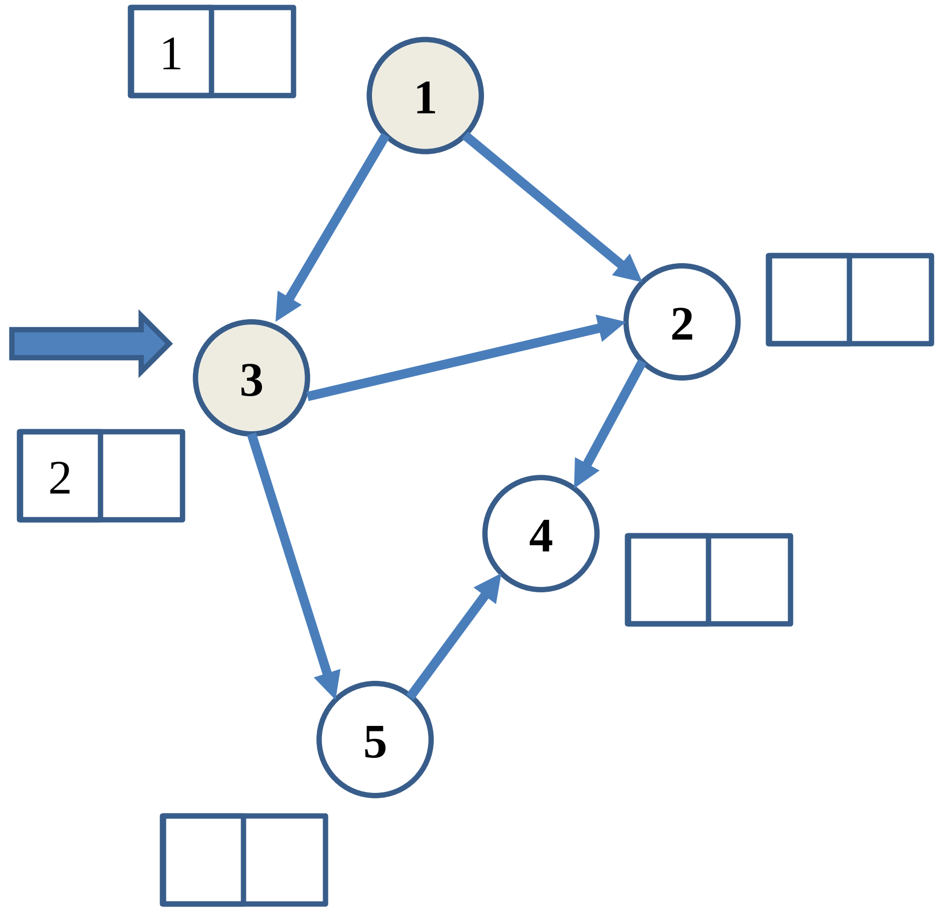
DFS in Action



DFS in Action

Time : 2

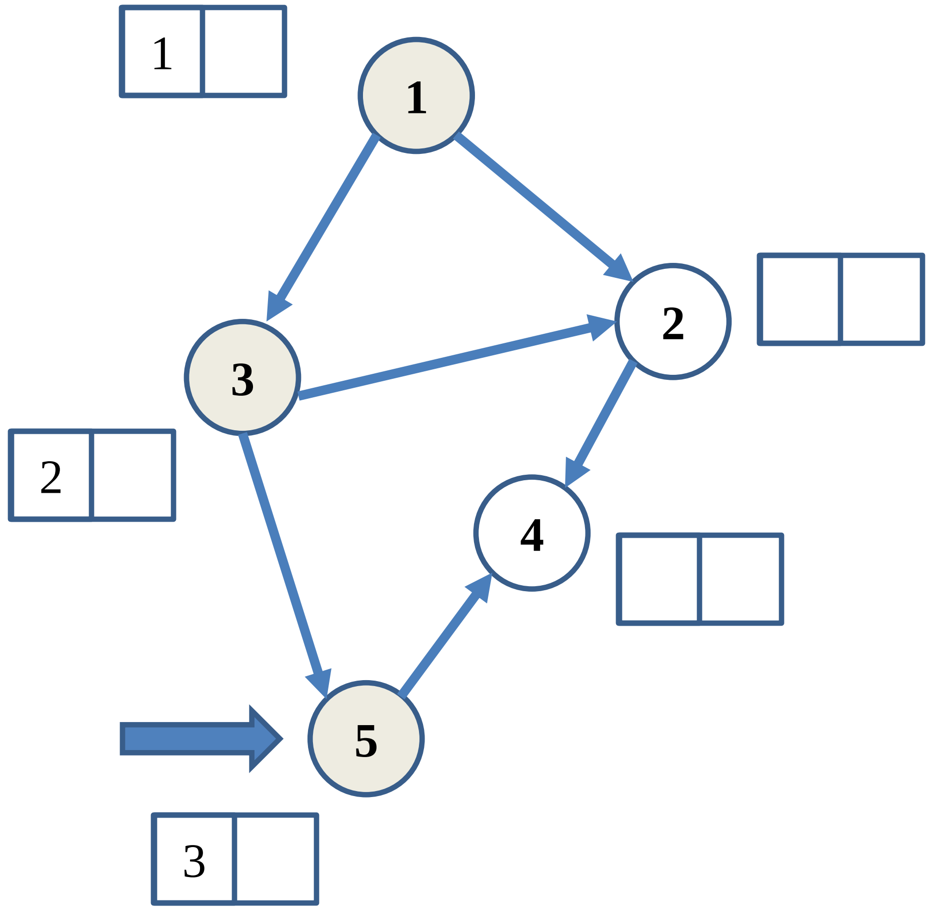
Π(3) = 1



DFS in Action

Time : 3

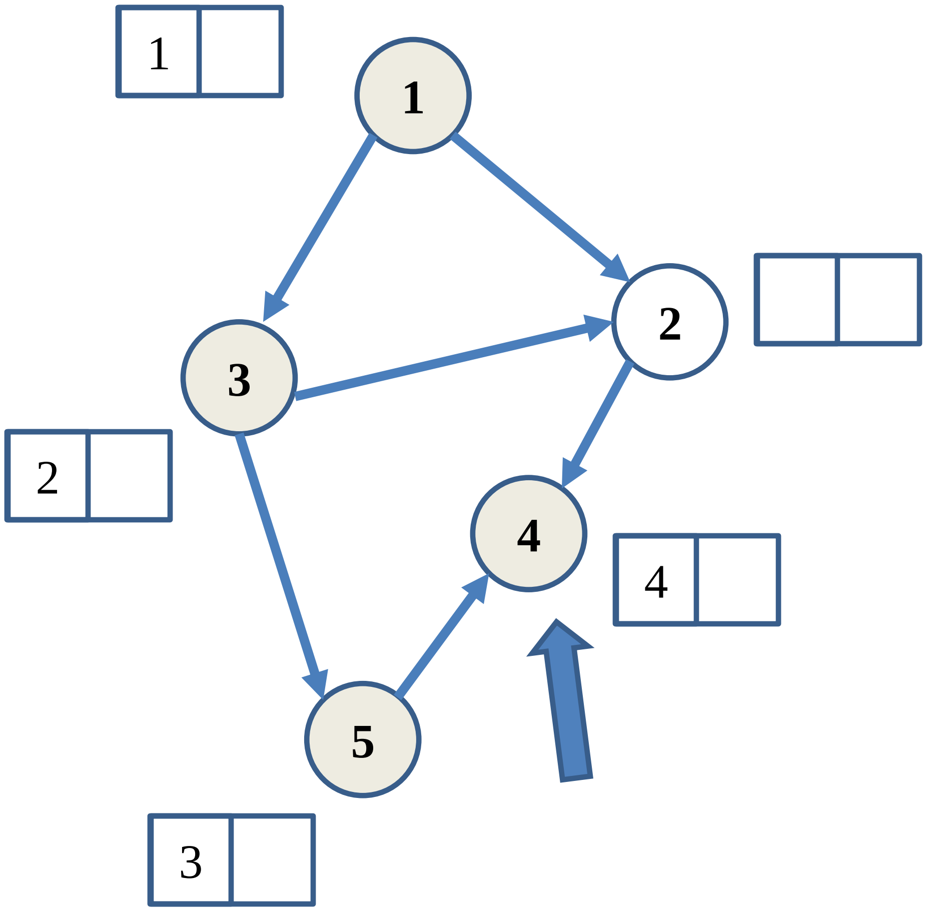
Π(3) = 1
Π(5) = 3



DFS in Action

Time : 4

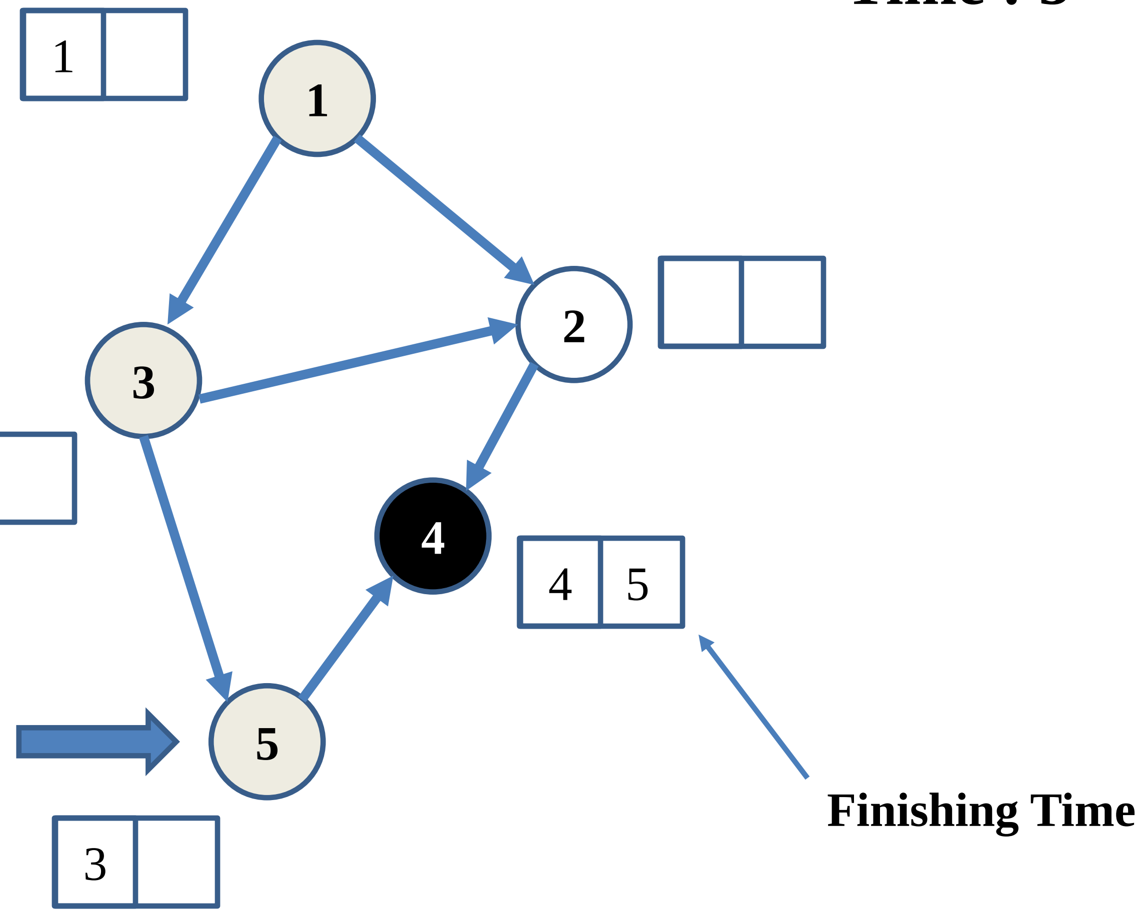
Π(3) = 1
Π(5) = 3
Π(4) = 5



DFS in Action

Time : 5

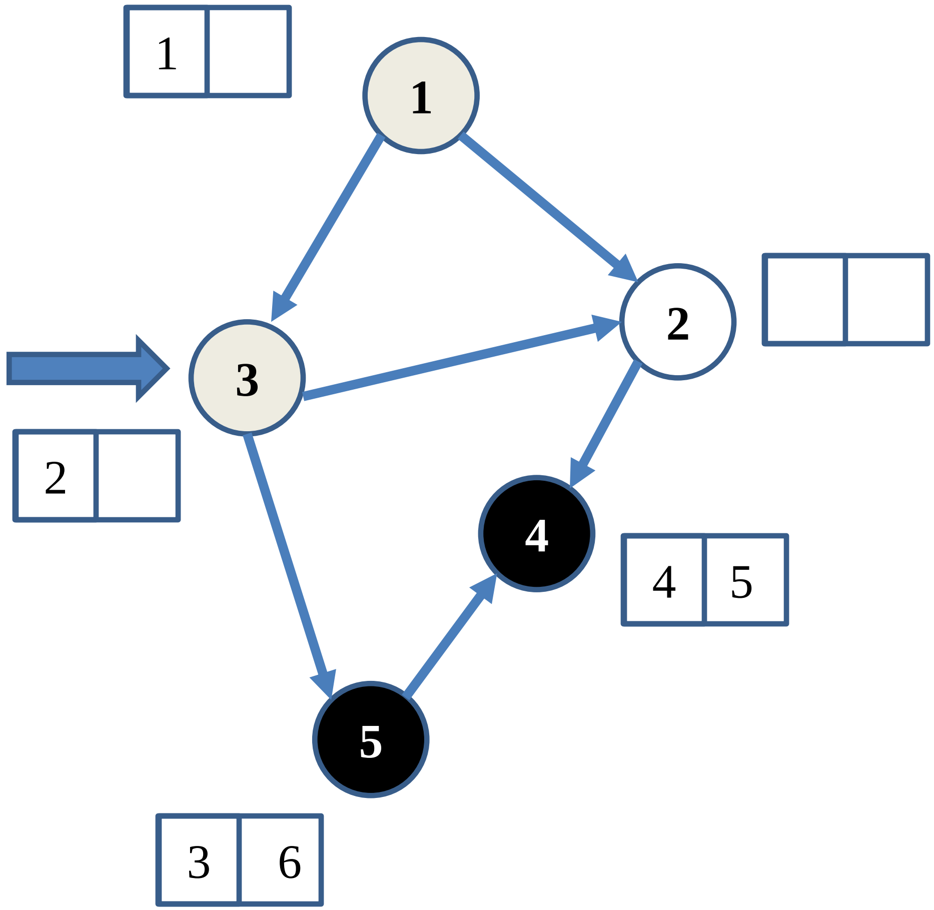
Π(3) = 1
Π(5) = 3
Π(4) = 5



DFS in Action

Time : 6

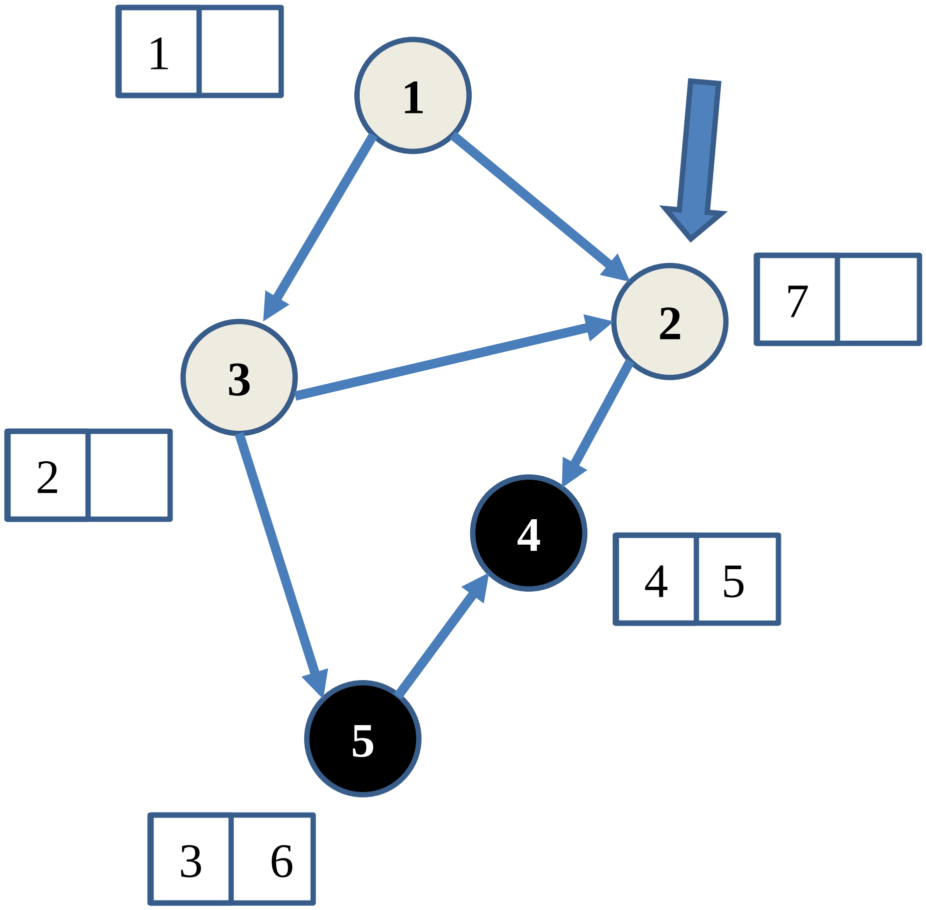
Π(3) = 1
Π(5) = 3
Π(4) = 5



DFS in Action

Time : 7

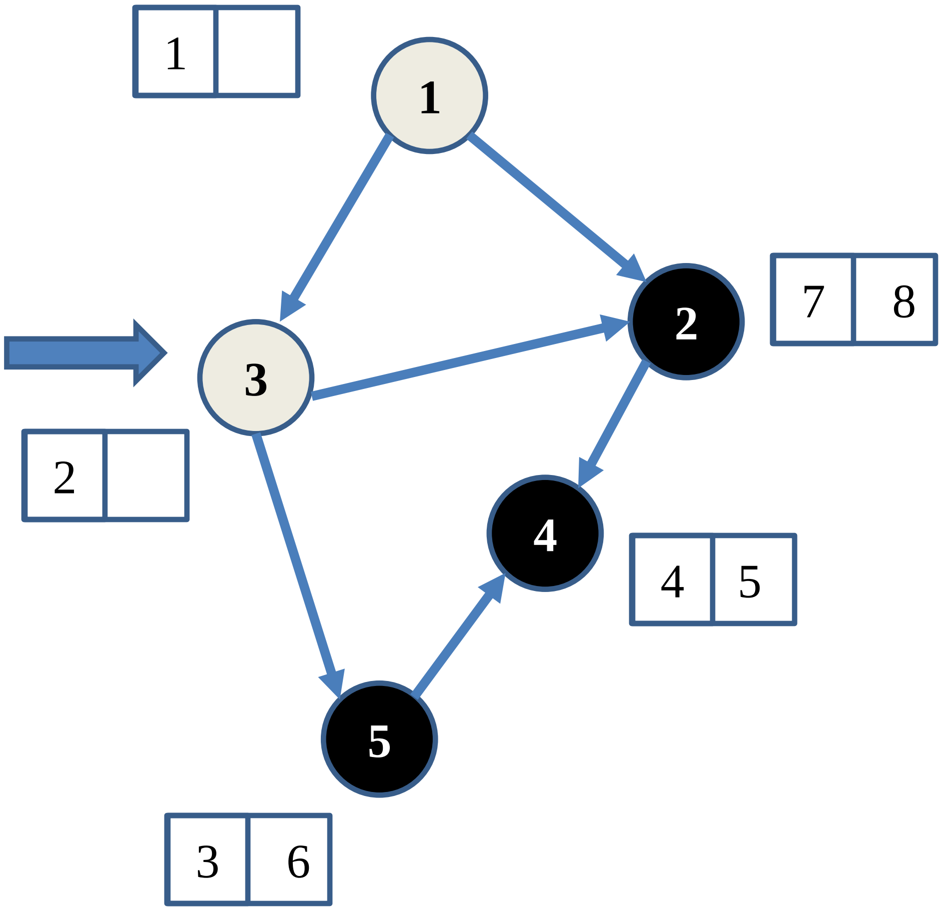
Π(3) = 1
Π(5) = 3
Π(4) = 5
Π(2) = 3



DFS in Action

Time : 8

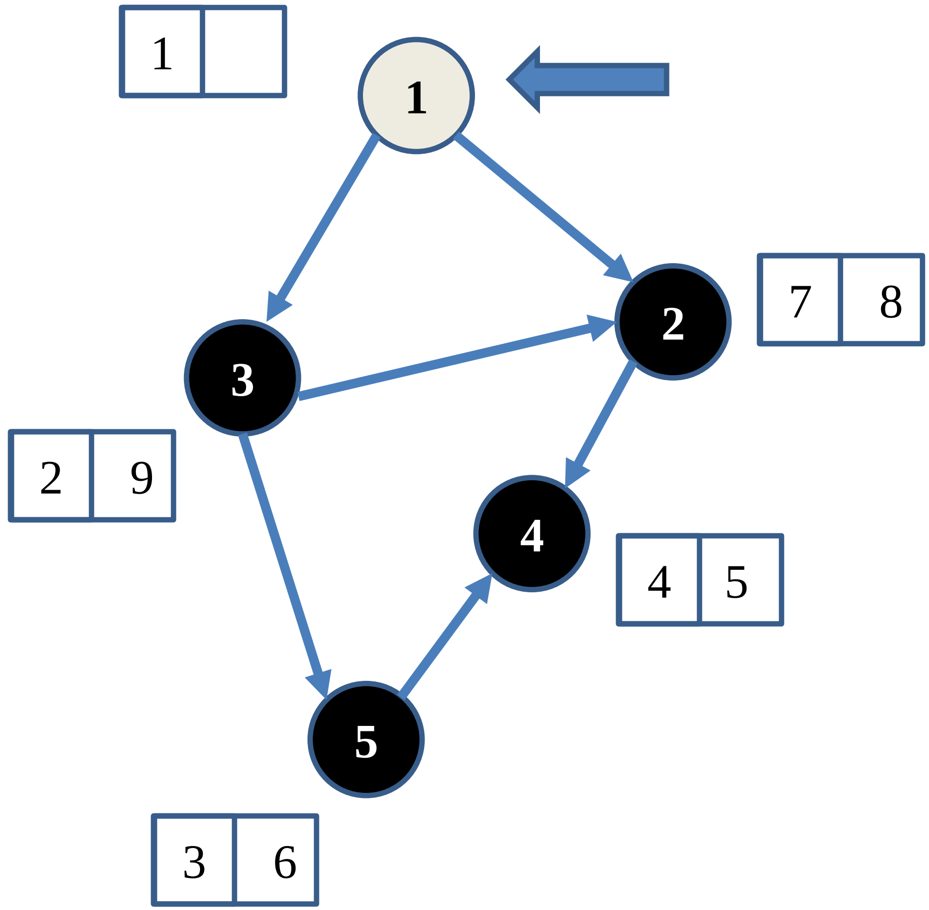
Π(3) = 1
Π(5) = 3
Π(4) = 5
Π(2) = 3



DFS in Action

Time : 9

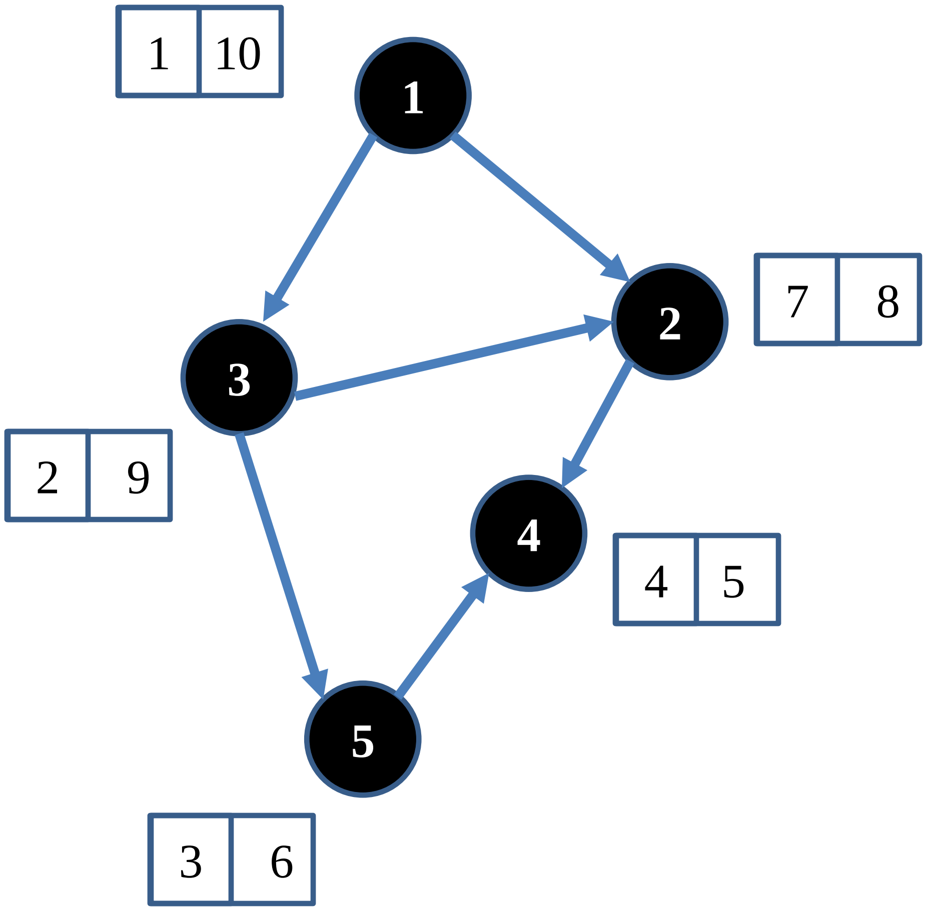
Π(3) = 1
Π(5) = 3
Π(4) = 5
Π(2) = 3



DFS in Action

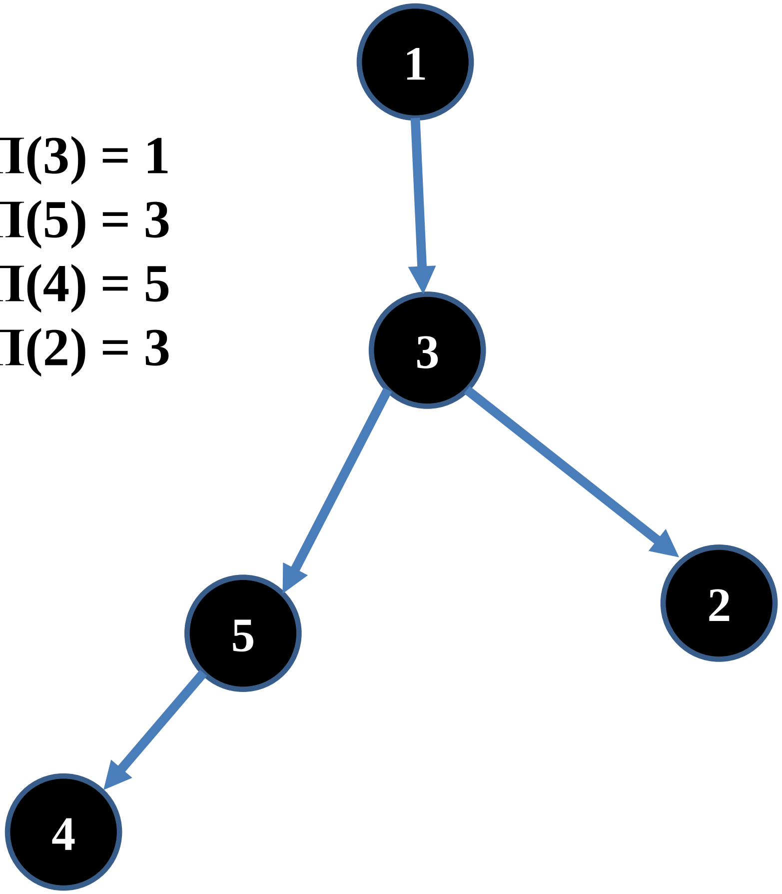
Time : 10

Π(3) = 1
Π(5) = 3
Π(4) = 5
Π(2) = 3



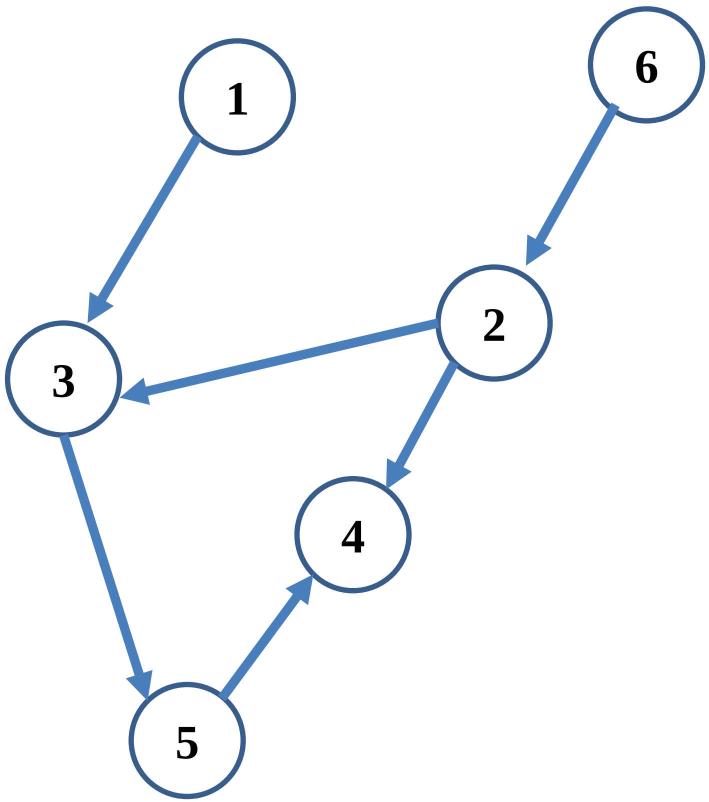
DFS Tree

$\Pi(3) = 1$
$\Pi(5) = 3$
$\Pi(4) = 5$
$\Pi(2) = 3$



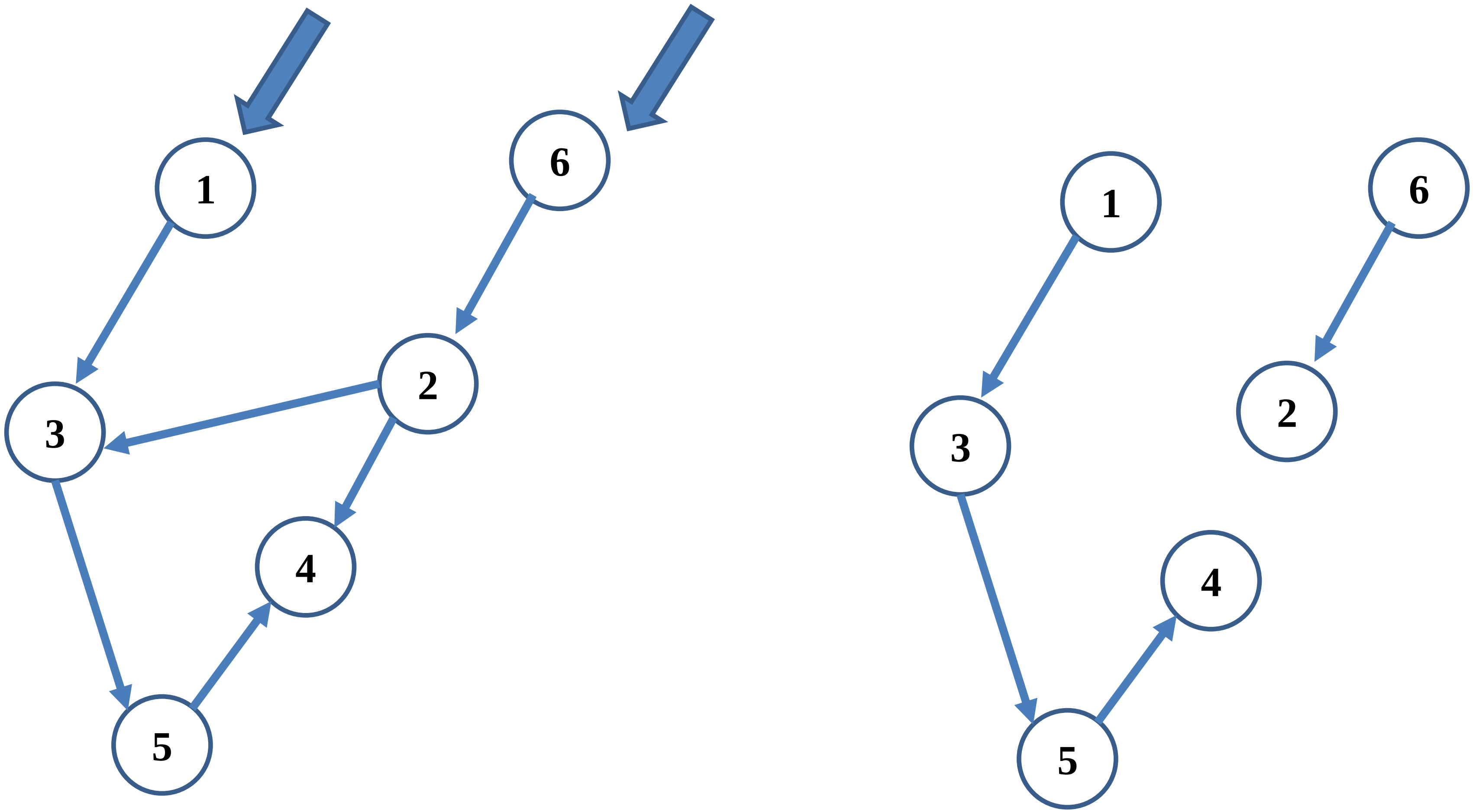
- DFS need not always form a single tree for a graph
- Sometimes, DFS may need to be performed multiple times from different starting nodes to discover all the nodes in the graph
- This gives rise to a number of DFS trees, collectively called a **DFS Forest**

Need for Multiple DFS Runs



- Nodes 2 and 6 are not reachable from 1
- Nodes 1 is not reachable from 6
- There is no starting node such that all the vertices in the graph are reachable
- We need to run DFS from at least two vertices – say 1 and 6

DFS Forest



If DFS is run from node 1 and then from node 6, we get two DFS trees (a DFS forest)

DFS Algorithm

DFS(G)

  for each *vertex u* ***in*** *V[G]*

    colour[u] = WHITE

    Π[u]= NIL

  time=0

  for each *vertex u* ***in*** *V[G]*

    if *colour[u]= WHITE*

      DFS_VISIT(u)

DFS_VISIT Algorithm

```
DFS_VISIT(u)
    colour[u] = GRAY
    time = time+1
    d[u] = time
    for each v in Adj[u]
        if colour[v]=WHITE
            Π[v] = u
            DFS_VISIT(v)
    colour[u] = BLACK
    time = time+1
    f[u] = time
```

Time Complexity of DFS

- Every node is explored EXACTLY ONCE --- $\Theta(|V|)$
- For every node u, DFS explores all the edges in Adj[u]
- When summed over all the nodes in the graph, this amounts to the number of edges in the graph

$$\sum_{u \in V} Adj[u] = \Theta(|E|)$$

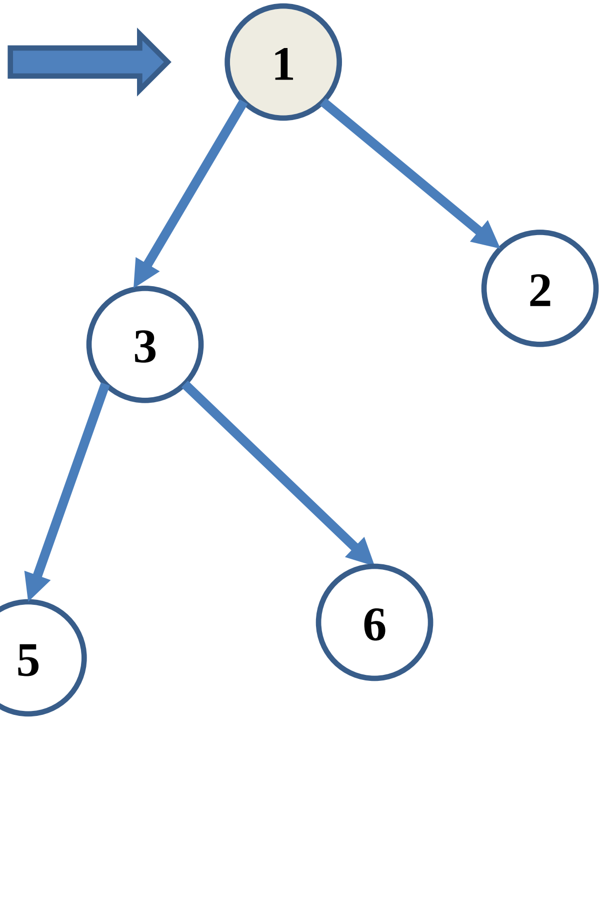
- Thus, **total complexity of DFS is $\Theta(|V| + |E|)$**

Breadth First Search

- Breadth First Search (BFS) is another graph traversal algorithm
- As the name suggests, BFS explores all the neighbours of a node before exploring any other node
- When BFS hits a dead end on a path, it backtracks and starts exploring a new path from the previous node
- This behaviour is suggestive of a FIFO data structure – QUEUE

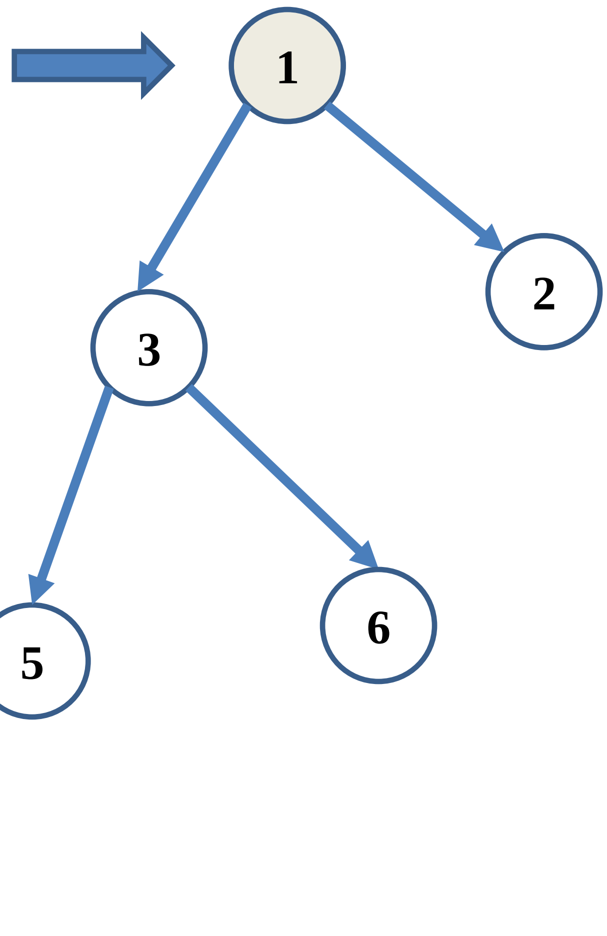
Breadth First Search

BFS is a graph traversal algorithm



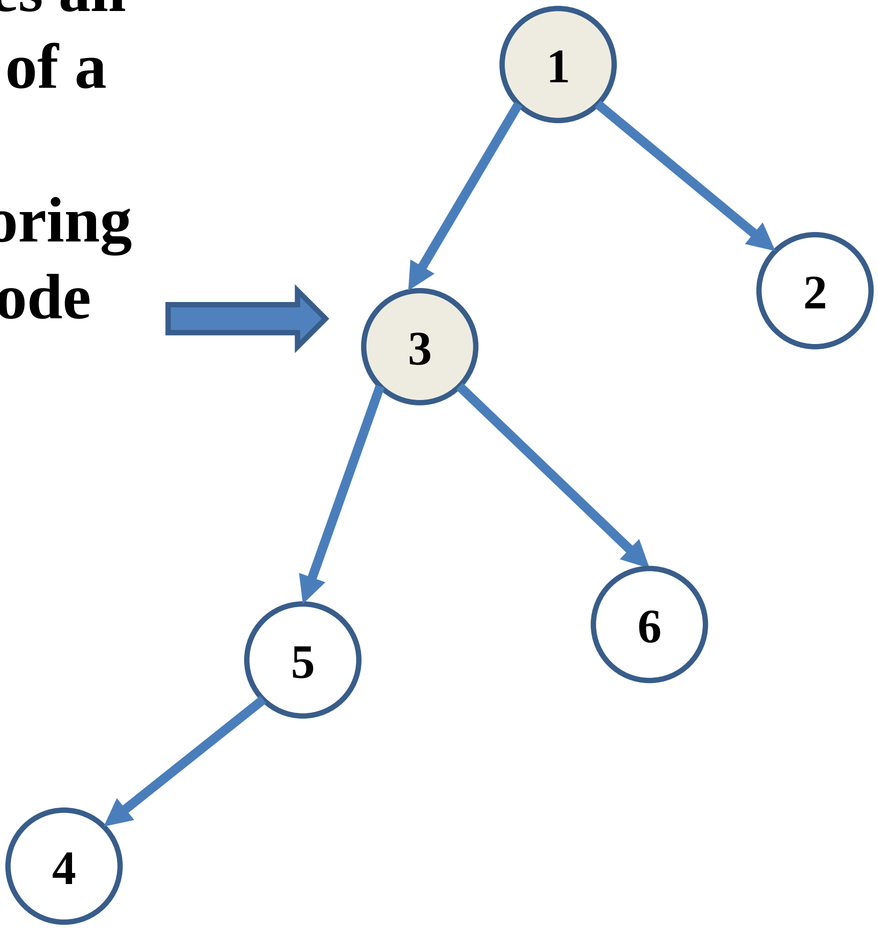
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



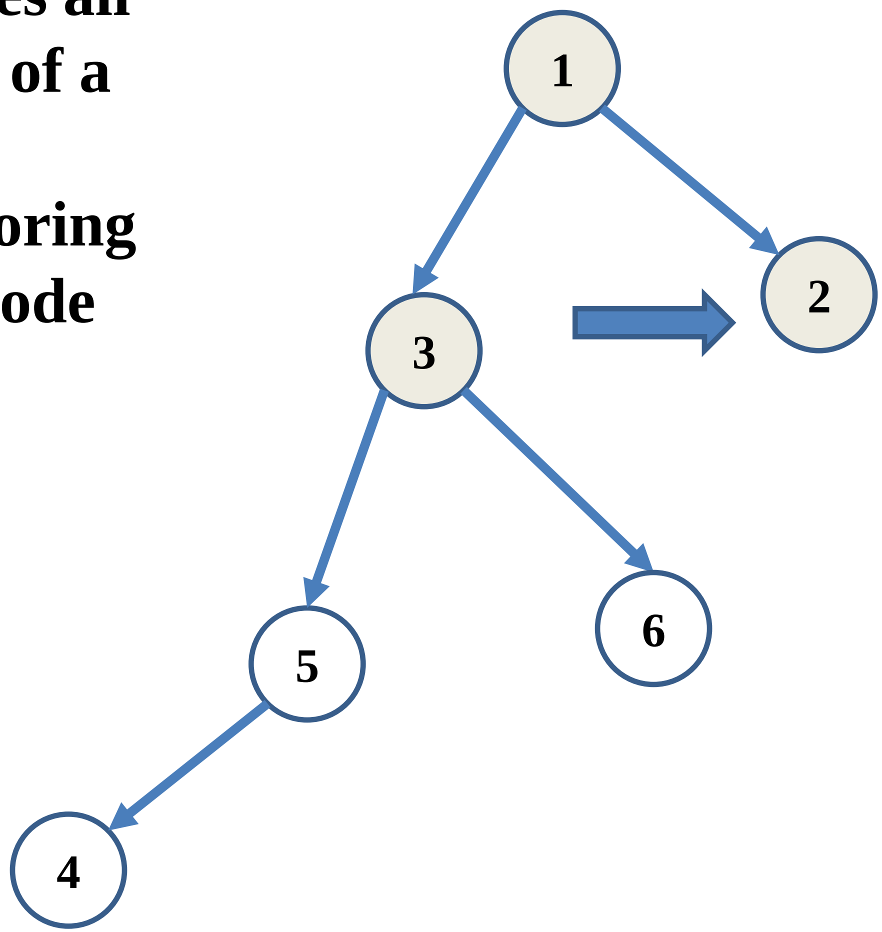
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



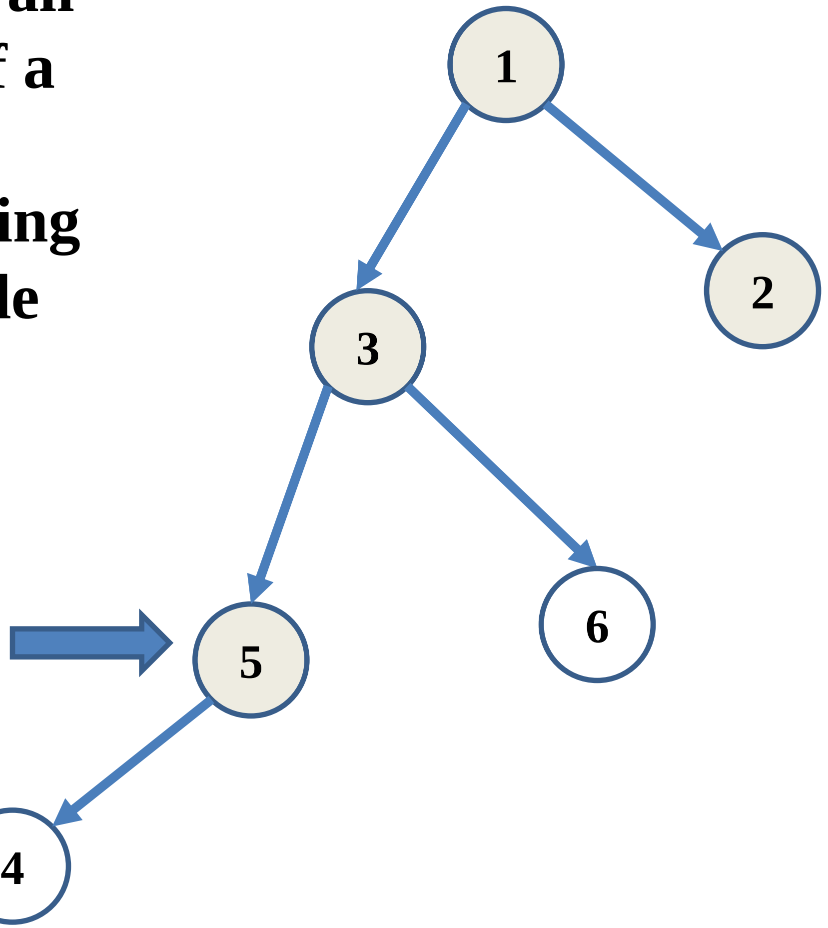
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



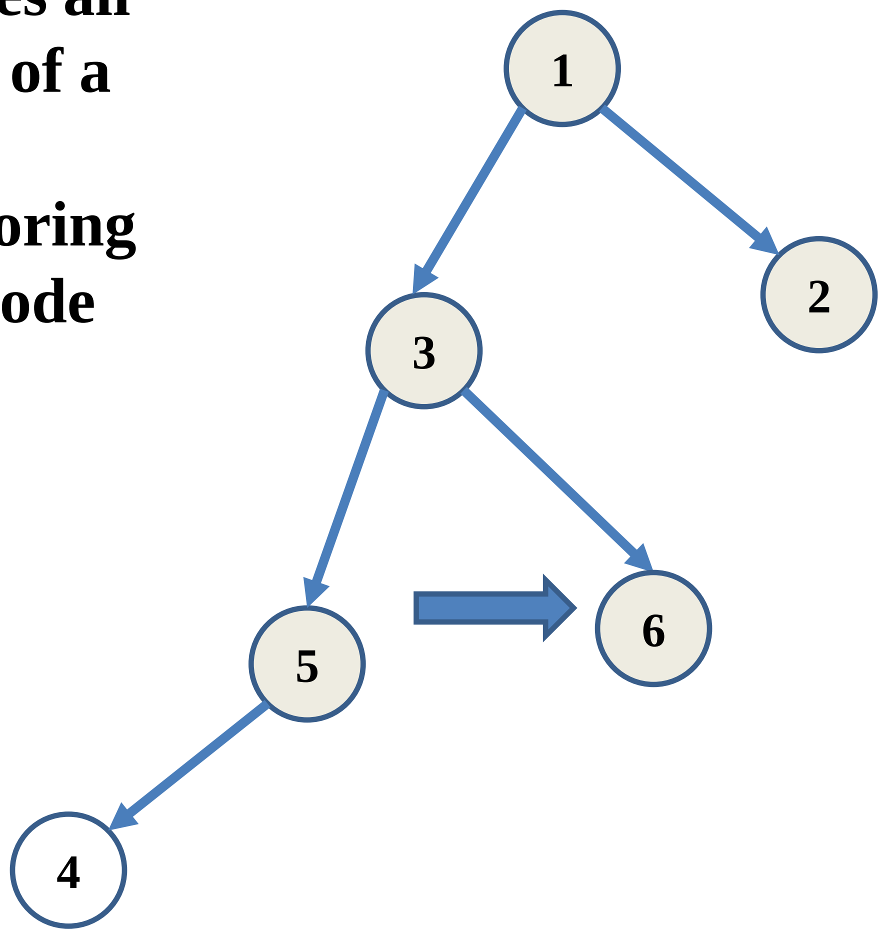
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



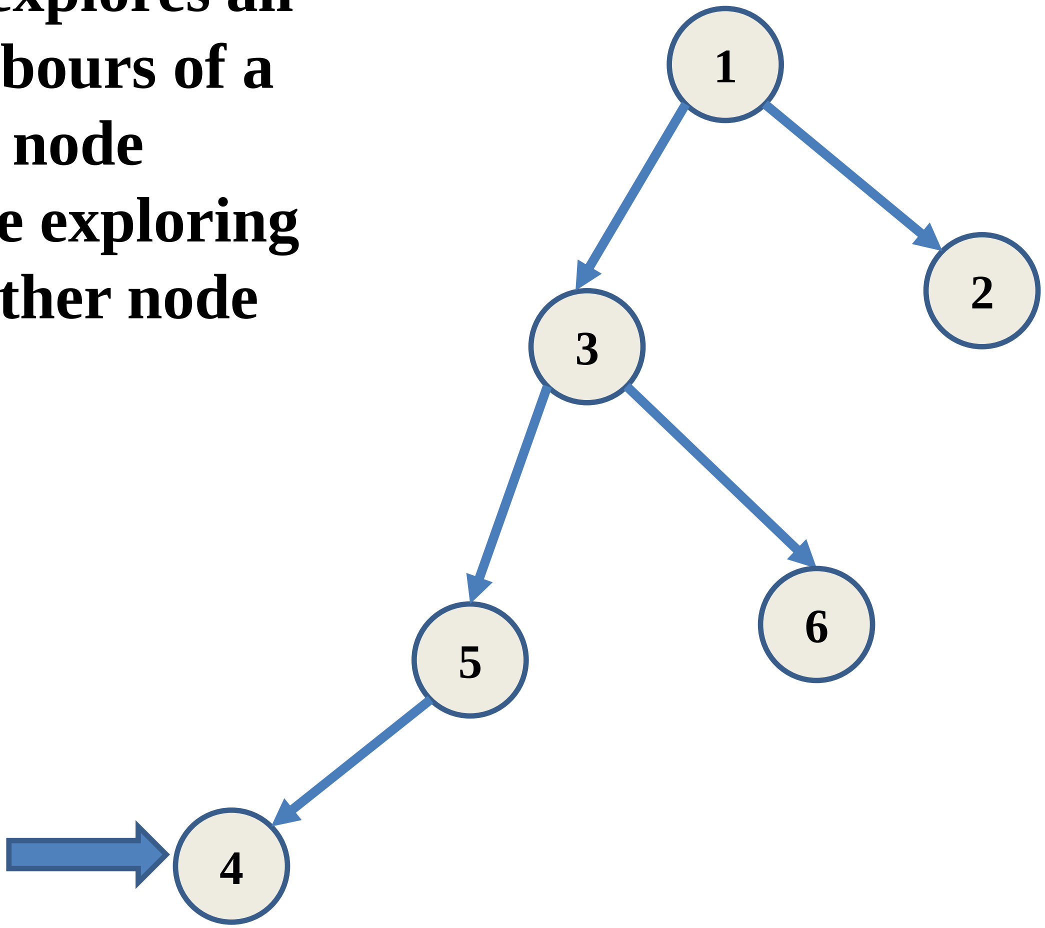
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



Breadth First Search

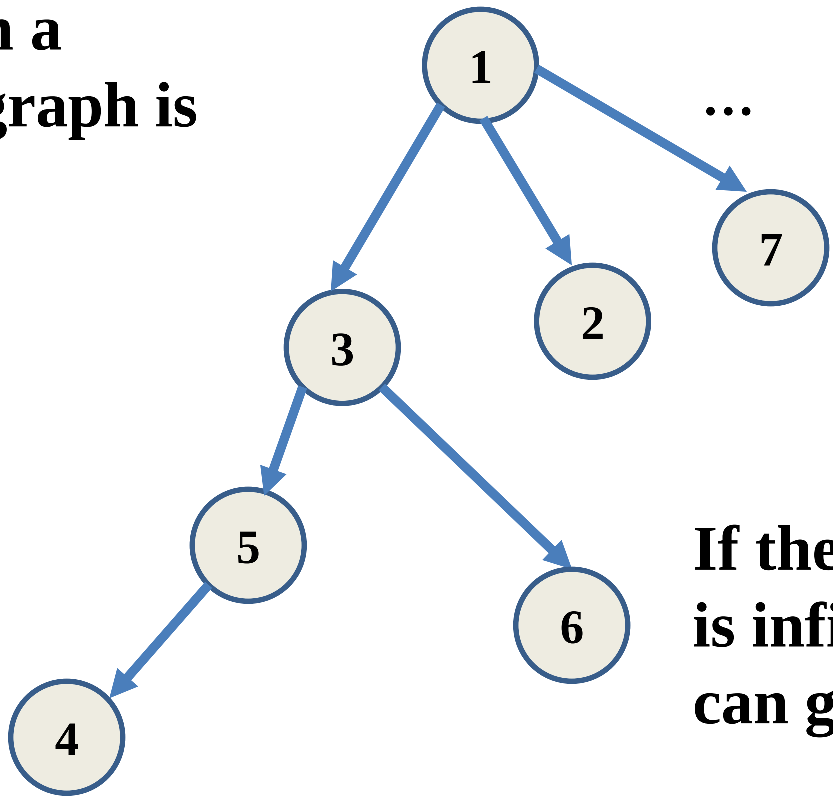
BFS explores all neighbours of a given node before exploring any other node



Completeness of BFS

BFS algorithm terminates with a solution if the graph is finite

Potentially continues to ∞

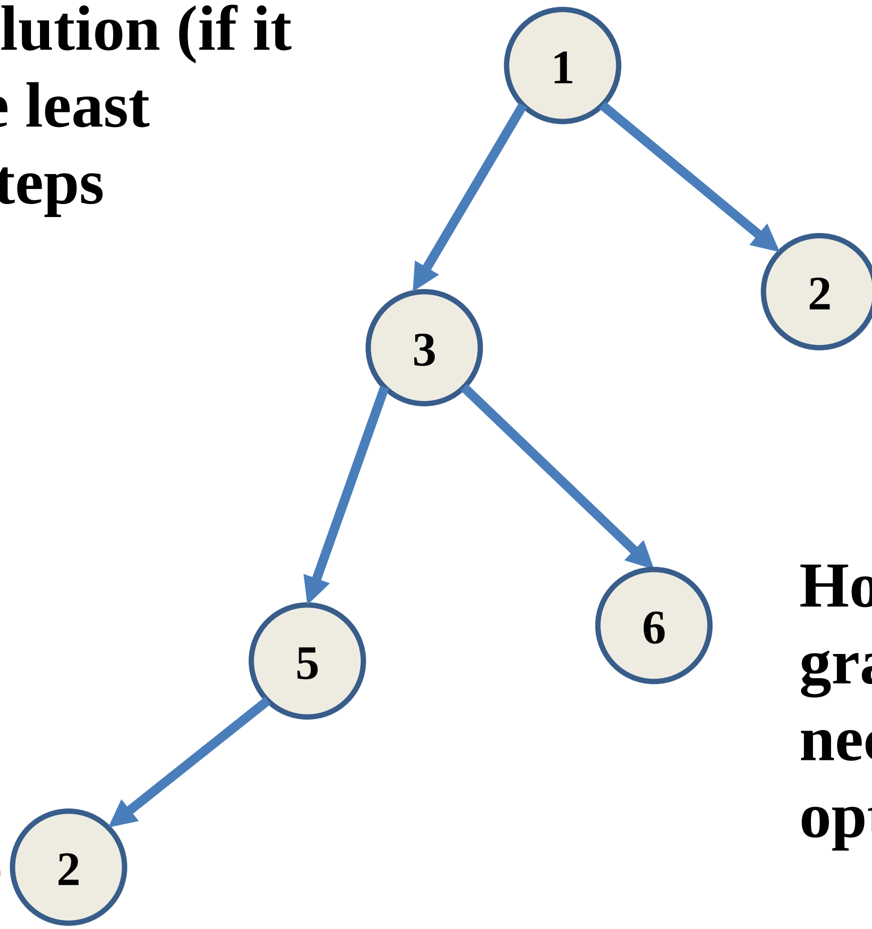


If the branching factor is infinite, then BFS can get stuck in a loop

Optimality of BFS

BFS algorithm will find the a solution (if it exists) in the least number of steps

Key = 2



Present here

Present here

However, in a weighted graph, BFS need not necessarily give an optimal solution

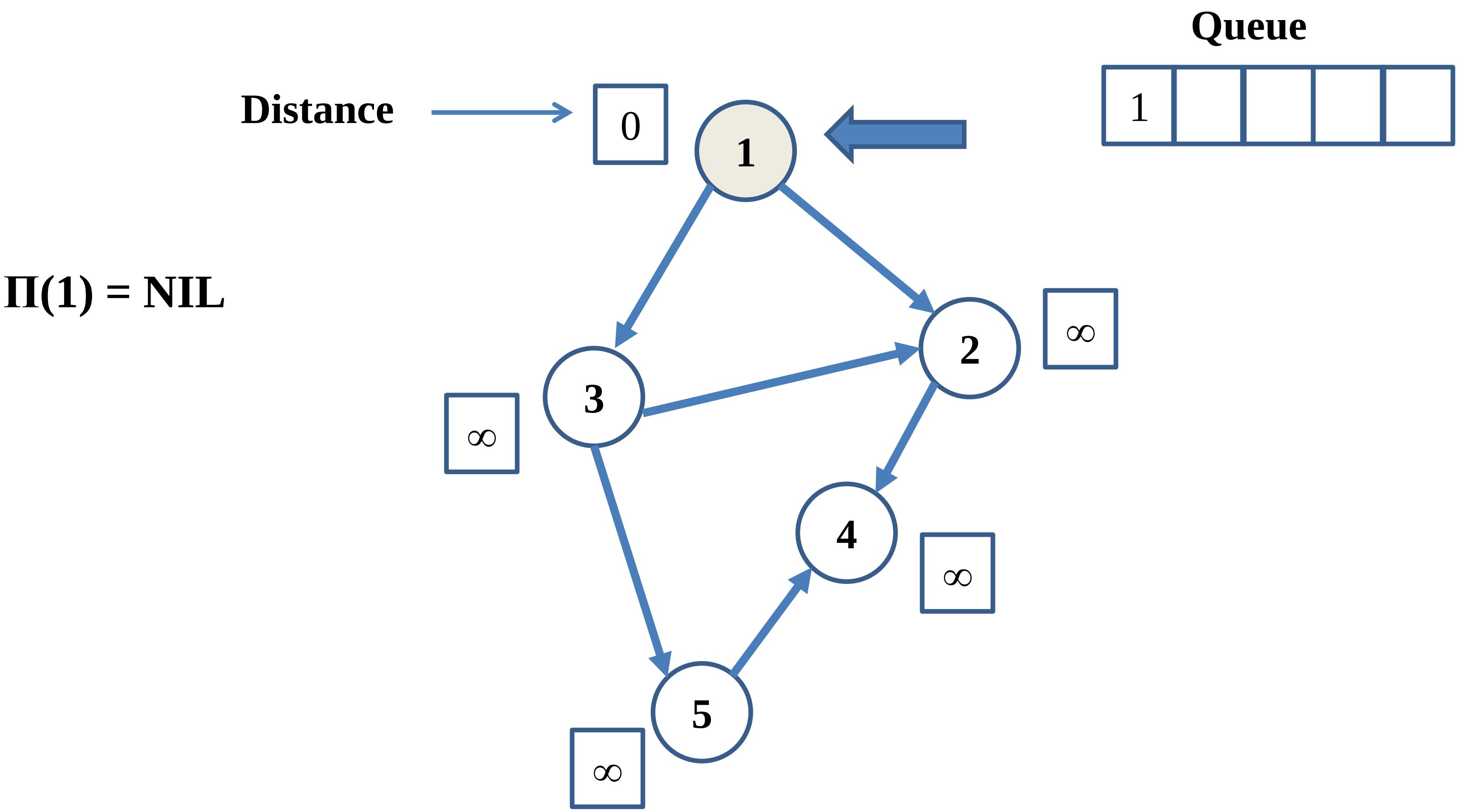
Applications of BFS

- Cycle detection
- Path Detection
- Finding strongly connected components

Associated Notations

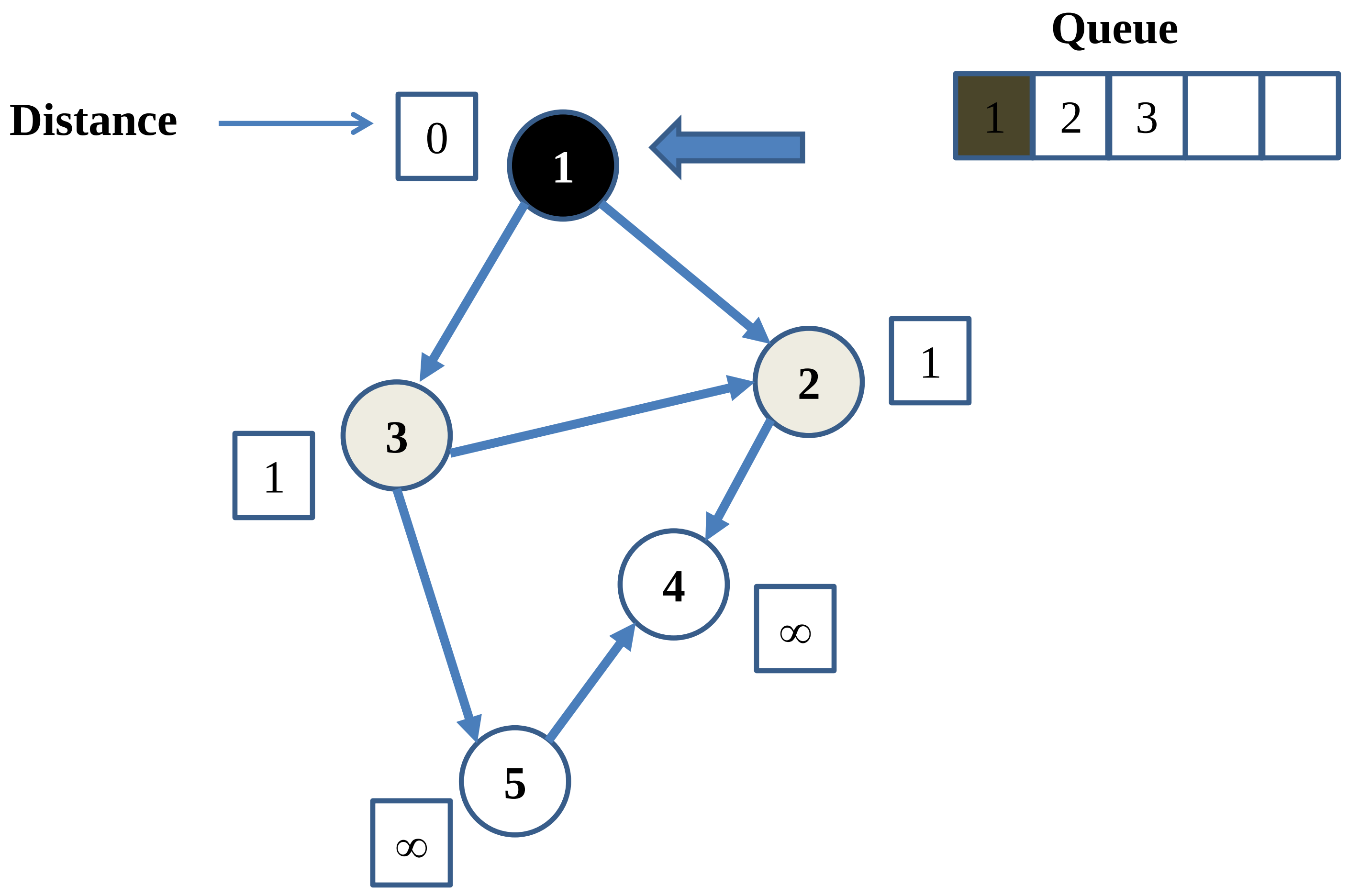
- **Nodes are assigned colours**
  - WHITE : The node has not been visited
  - GRAY : The node has been visited, but all of its branches have not been visited completely
  - BLACK : A node and its branches have been explored completely
- Every node is also assigned a distance value which is the number of steps it took to reach that node from the source vertex

BFS in Action

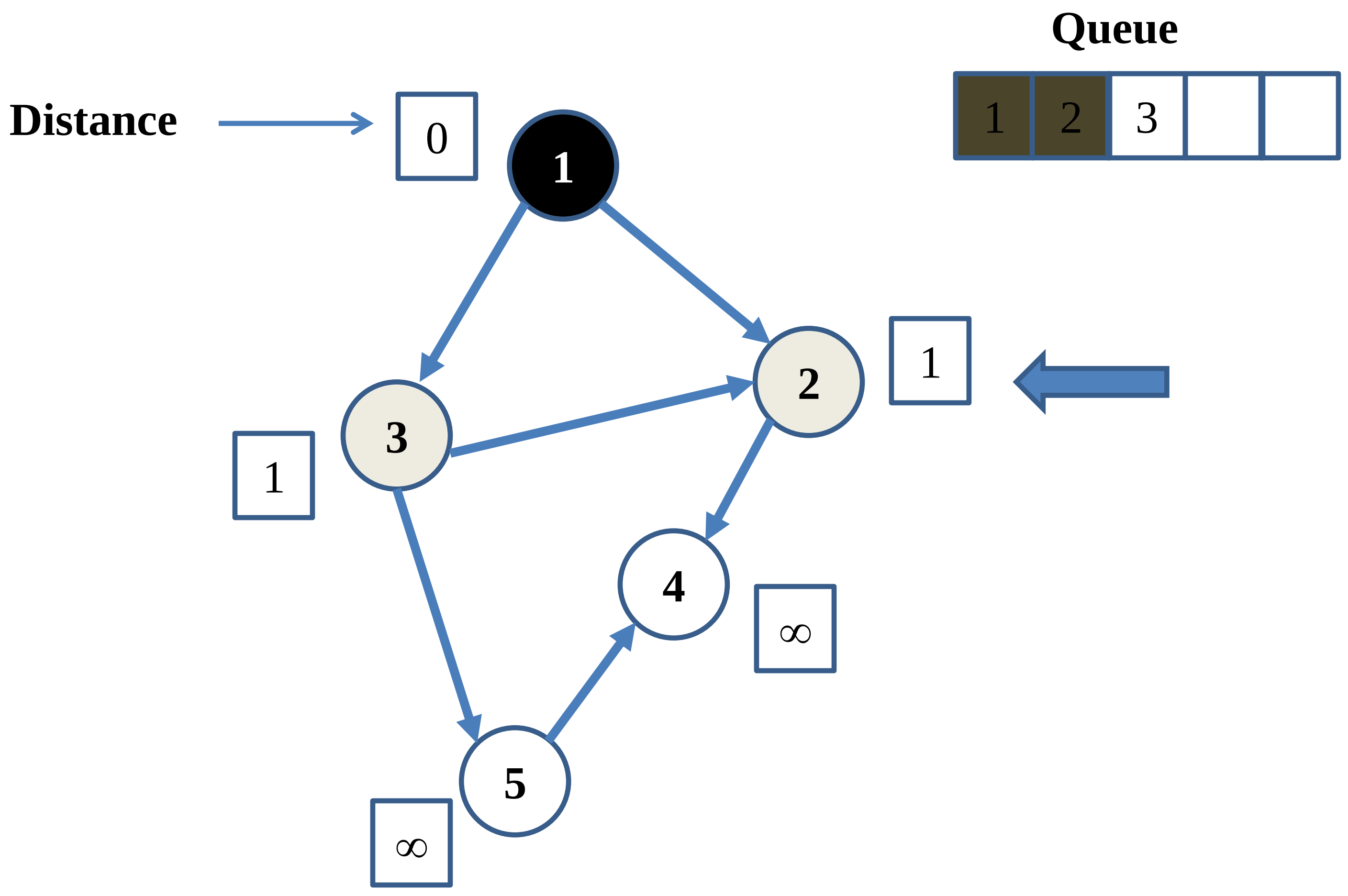


BFS in Action

Π(1) = NIL
Π(2) = 1
Π(3) = 1

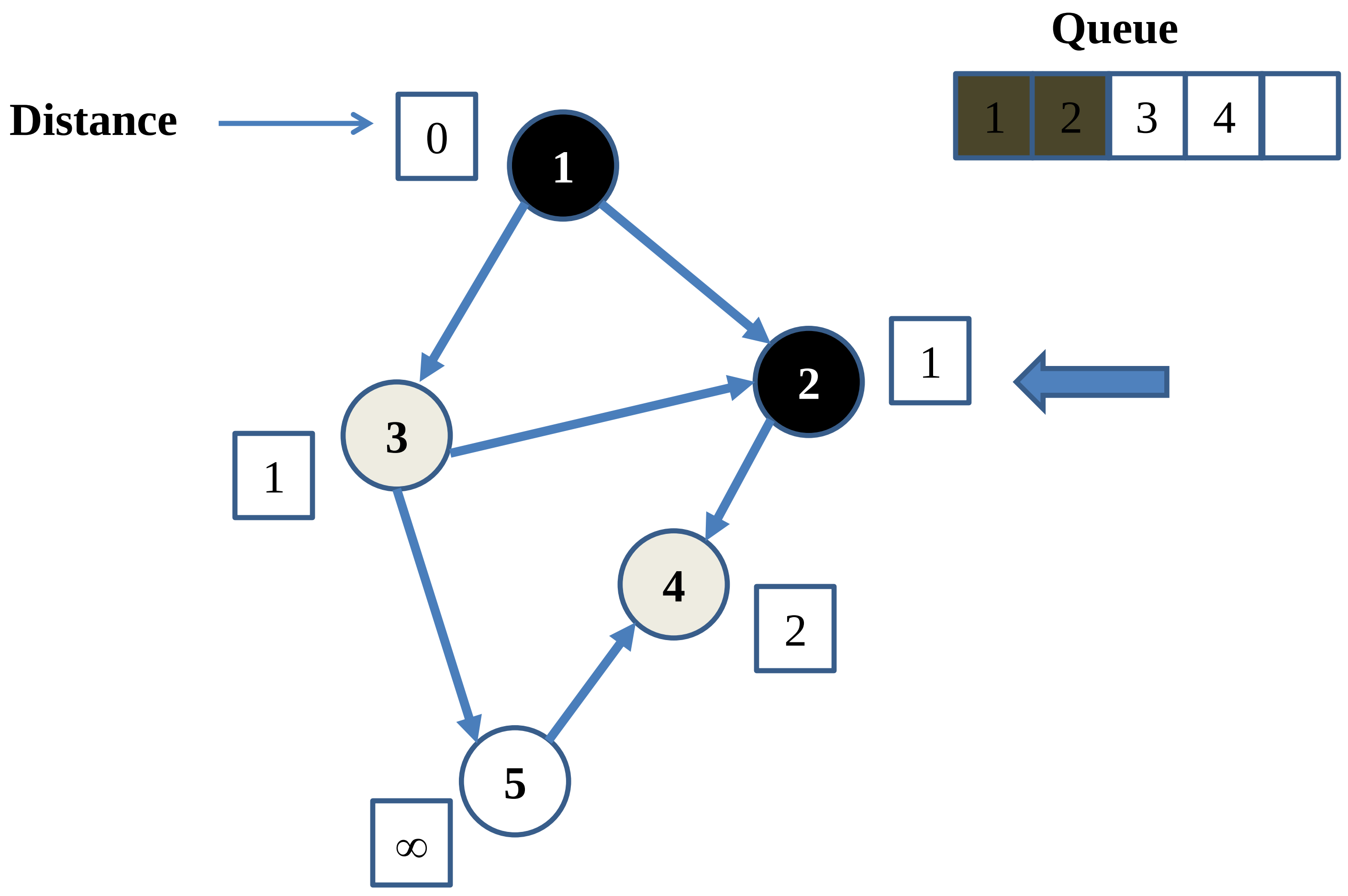


BFS in Action



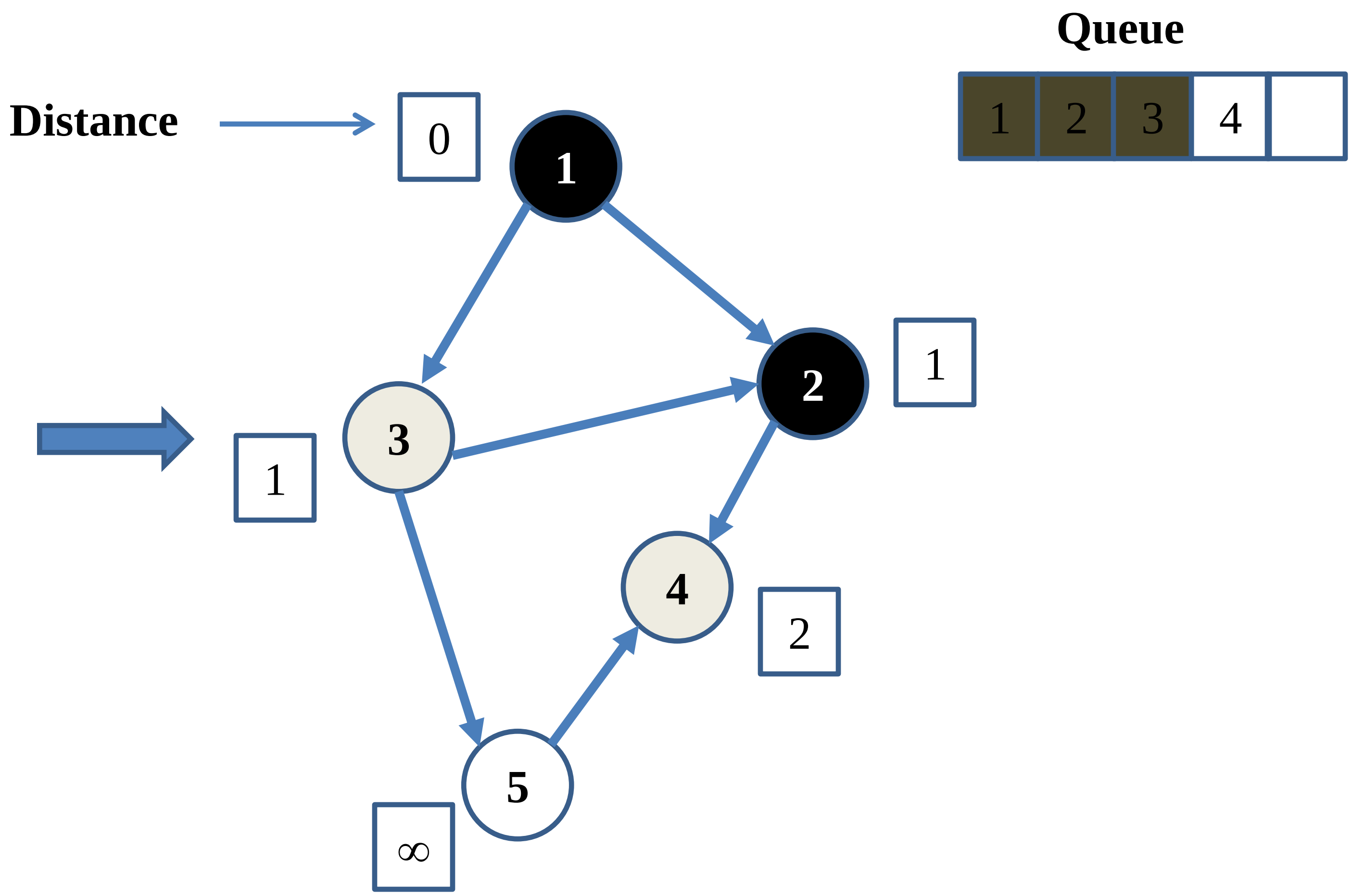
Π(1) = NIL
Π(2) = 1
Π(3) = 1

BFS in Action



Π(1) = NIL
Π(2) = 1
Π(3) = 1
Π(4) = 2

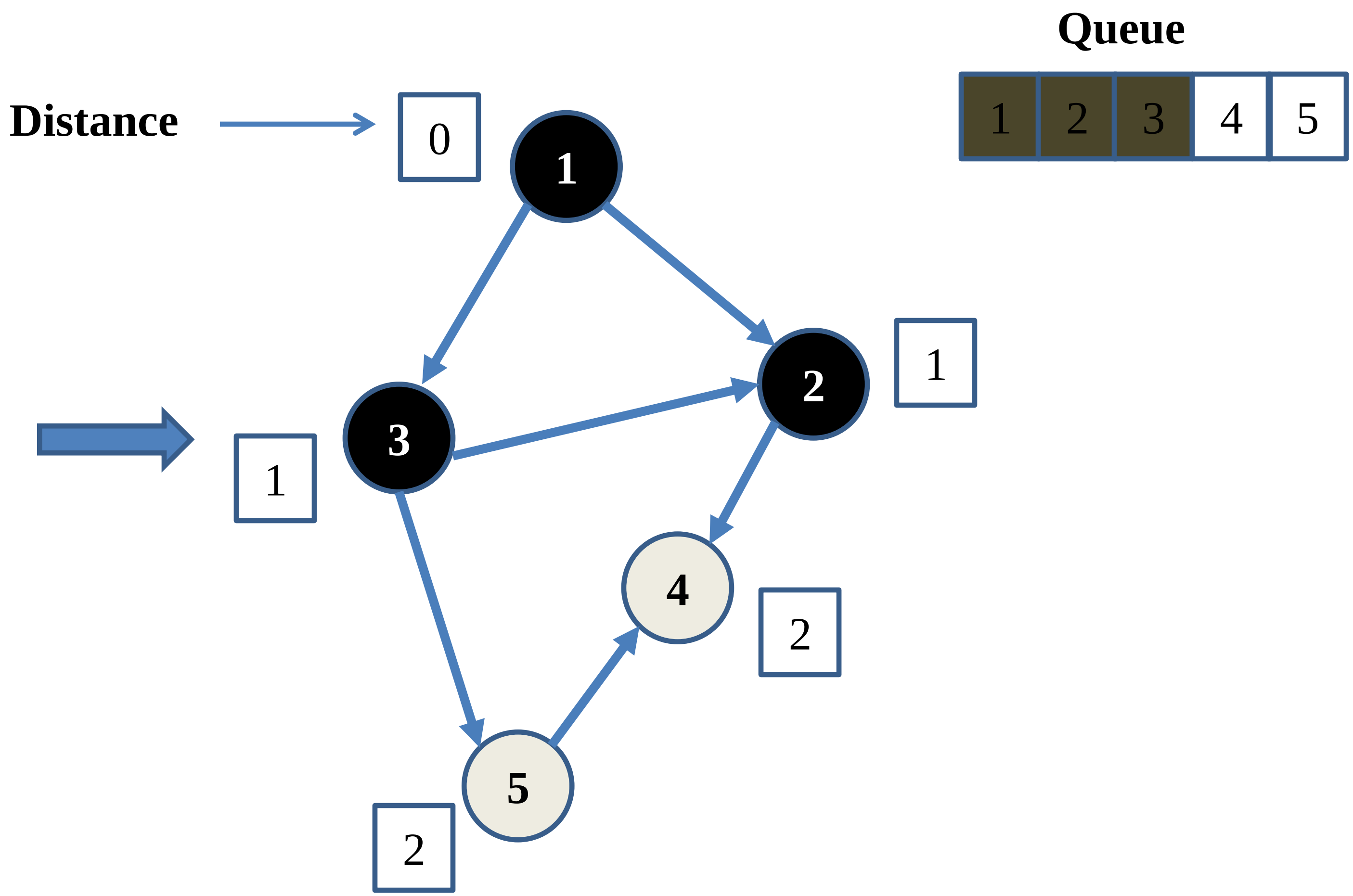
BFS in Action



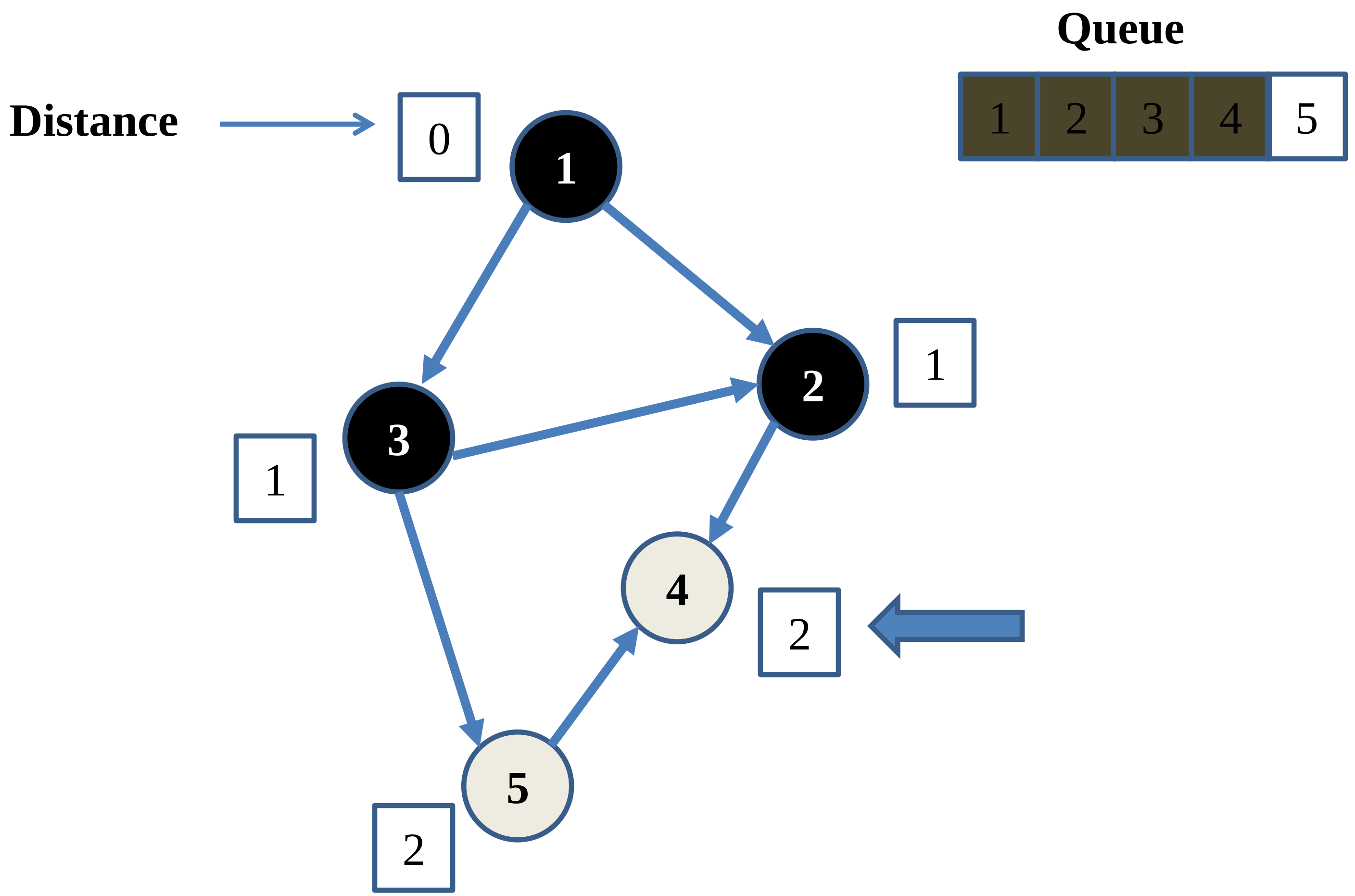
Π(1) = NIL
Π(2) = 1
Π(3) = 1
Π(4) = 2

BFS in Action

Π(1) = NIL
Π(2) = 1
Π(3) = 1
Π(4) = 2
Π(5) = 3

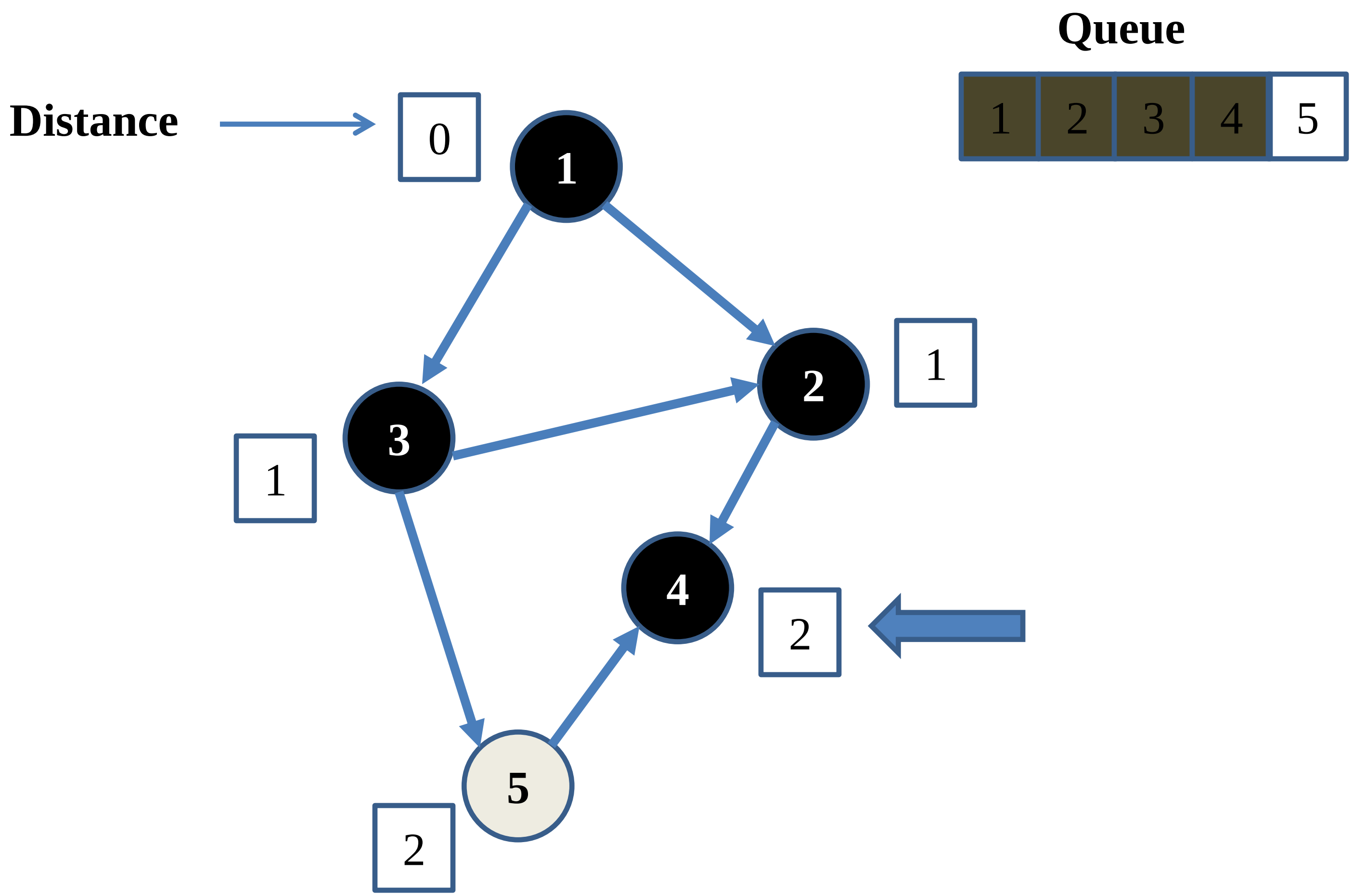


BFS in Action



Π(1) = NIL
Π(2) = 1
Π(3) = 1
Π(4) = 2
Π(5) = 3

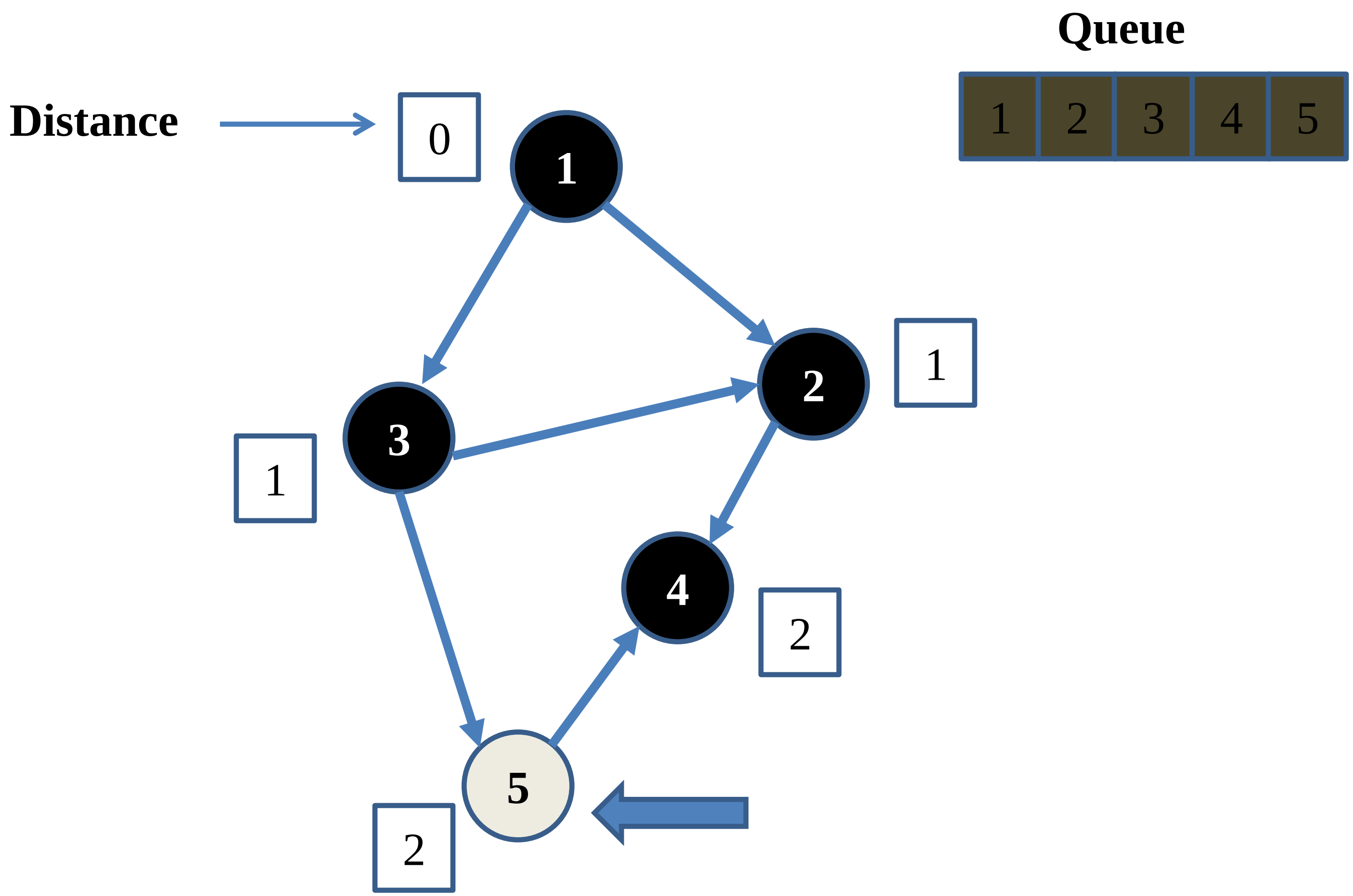
BFS in Action



Π(1) = NIL
Π(2) = 1
Π(3) = 1
Π(4) = 2
Π(5) = 3

BFS in Action

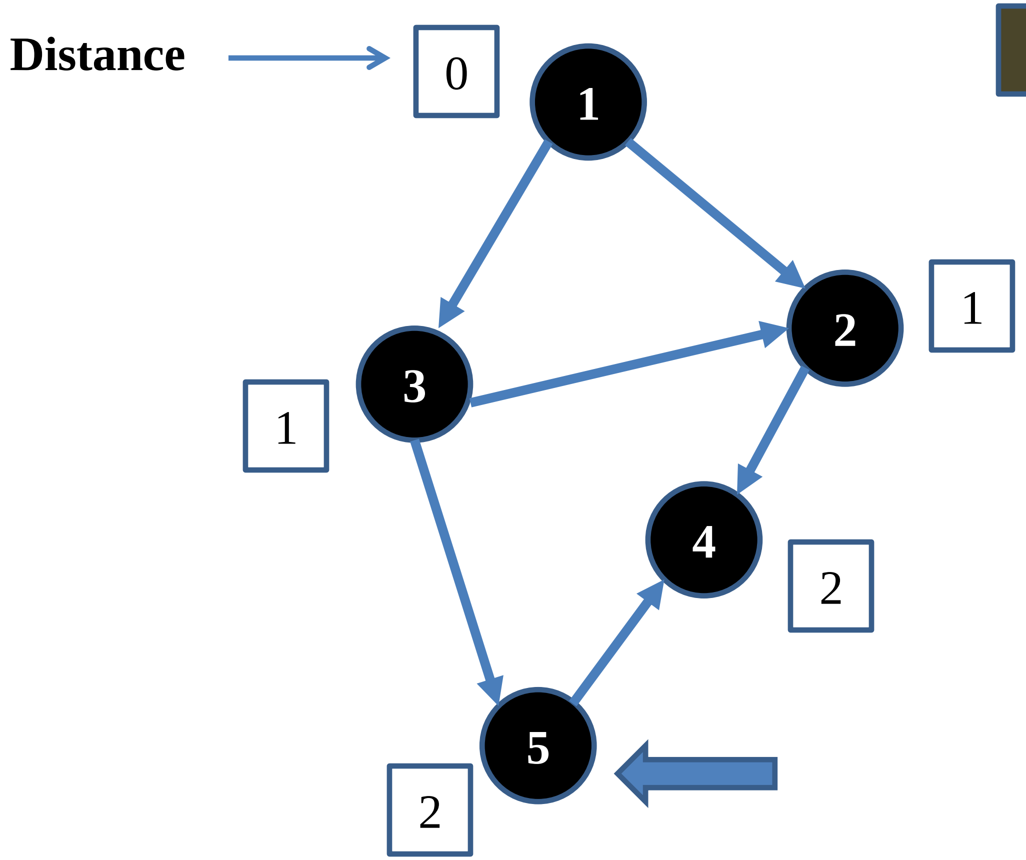
Π(1) = NIL
Π(2) = 1
Π(3) = 1
Π(4) = 2
Π(5) = 3



BFS in Action

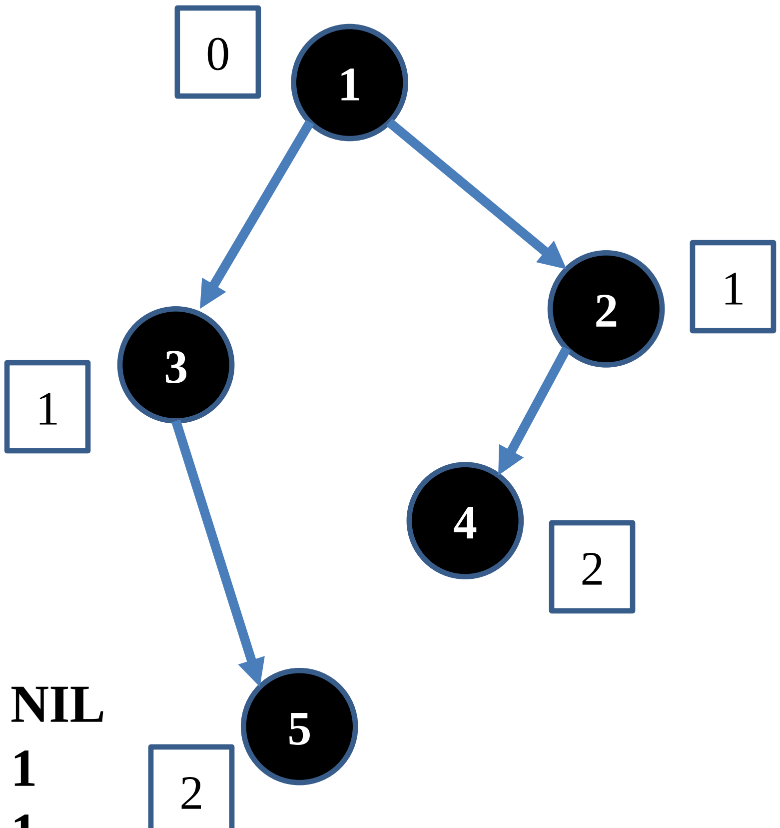
Queue

1	2	3	4	5



Π(1) = NIL
Π(2) = 1
Π(3) = 1
Π(4) = 2
Π(5) = 3

BFS Tree



- The breadth first tree gives the nodes reachable from the source node
- For an unweighted graph, it also shows the shortest path from the source vertex to every other vertex in the graph

Π(1) = NIL
Π(2) = 1
Π(3) = 1
Π(4) = 2
Π(5) = 3

BFS Algorithm

BFS(G, s)

for each *vertex u* **in** $V[G] - \{s\}$

$colour[u] = WHITE$

$\Pi[u] = NIL$

$d[u] = \infty$

$colour[s] = GRAY$

$d[s] = 0$

$\Pi[u] = NIL$

$Q = \phi$

ENQUEUE (Q, s)

BFS Algorithm

while $Q \neq \phi$

    u=DEQUEUE()

    for each *v* ***in*** *Adj[u]*

        if *colour[v]=WHITE*

            $\Pi[v] = u$

            $d[v] = d[u] + 1$

            ENQUEUE(v)

    colour[u] = BLACK

Time Complexity of BFS

- Every node is explored EXACTLY ONCE --- $\Theta(|V|)$
- For every node u, BFS explores all the edges in Adj[u]
- When summed over all the nodes in the graph, this amounts to the number of edges in the graph

$$\sum_{u \in V} Adj[u] = \Theta(|E|)$$

- Thus, **total complexity of BFS is also $\Theta(|V| + |E|)$**

Thank You