APPLICATION IN DESIGNING ALGORITHMS: GRAPH ALGORITHMS

GROUP 12

Lê Quang Thiên Phúc
Lý Nguyên Thùy Linh

TABLE OF CONTENTS

Part 1: OVERVIEW

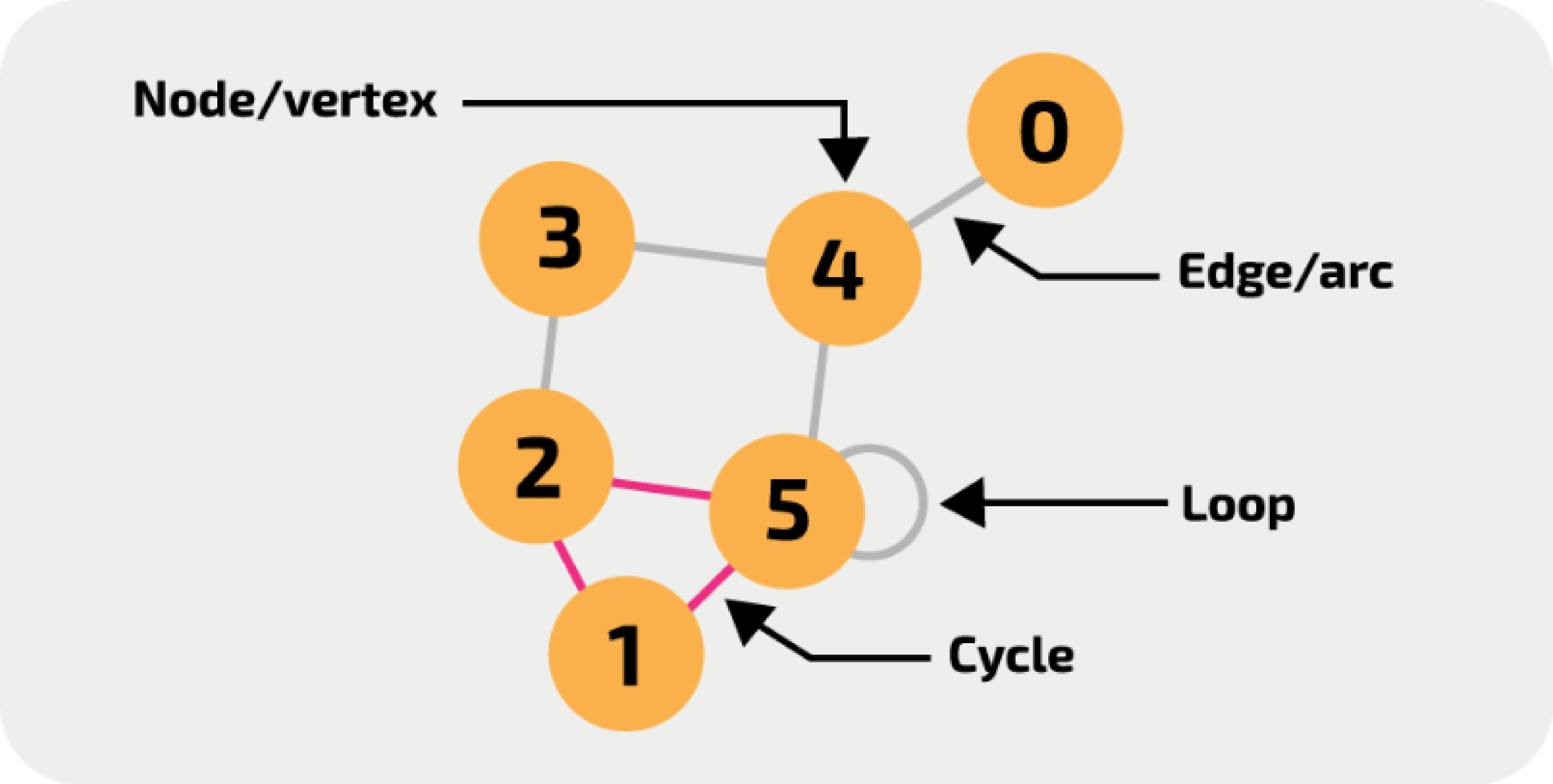
Part 1: OVERVIEW

?

What is a GRAPH?

Part 1: OVERVIEW

What is a GRAPH?



Part 1: OVERVIEW

What are differences between geometry and graph?

Part 1: OVERVIEW

Traversing Algorithm

- DFS
- BFS

Part 1: OVERVIEW

	DFS	BFS
Order	• explores as deeply as possible along a branch before backtracking • processes from farther to nearer	• explores vertices level by level • processes from nearer to farther
Approach	Backtracking	???
Data structure	stack	queue
Specific use cases	• ???	• finding shortest path (in a map, a network, a puzzle,...)
Common use cases	Better if the graph is wide	Better if solutions are shallow
Time complexity	O(|V| + |E|) |V|: numbers of vertices |E|: numbers of edges	

Part 2: GRAPH'S ALGORITHMS

Part 2: GRAPH'S ALGORITHMS

Topological Sort

- Problem Statement
- Technical Definition
- Algorithm

Part 2: GRAPH'S ALGORITHMS

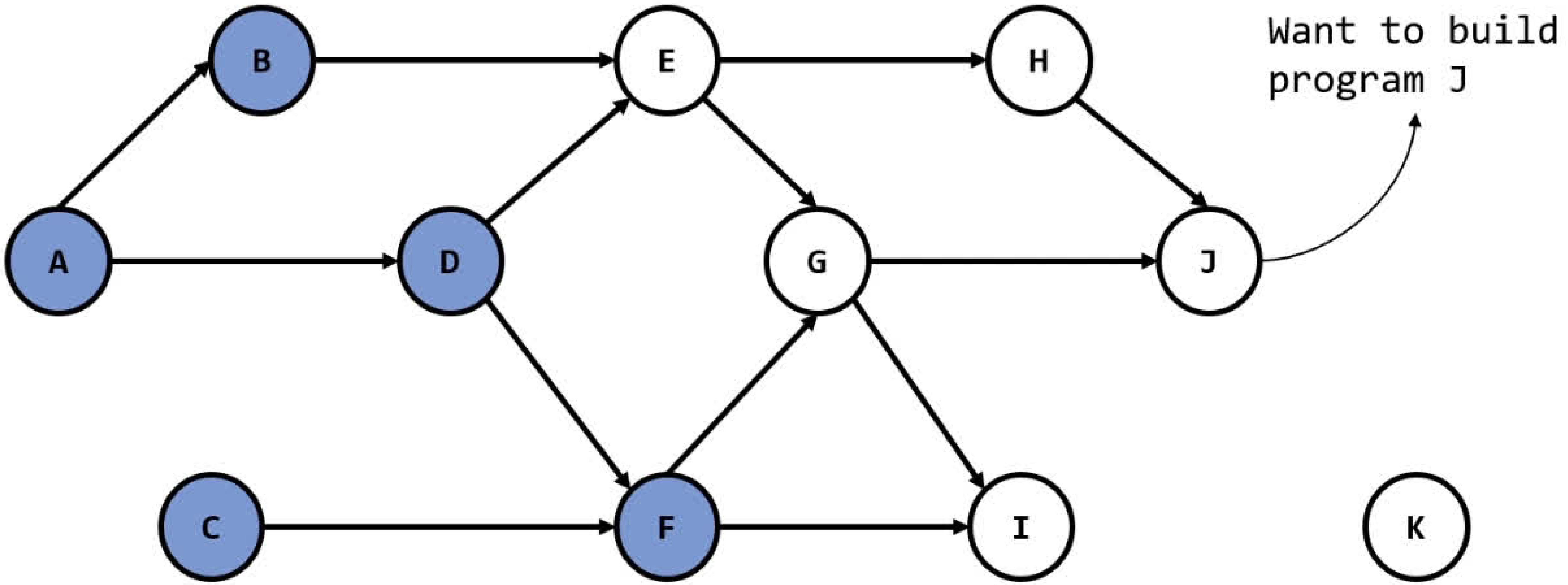
Topological Sort

Problem Statement

Technical Definition

Algorithm

Program build dependencies: a program cannot be built unless its dependencies are built first.



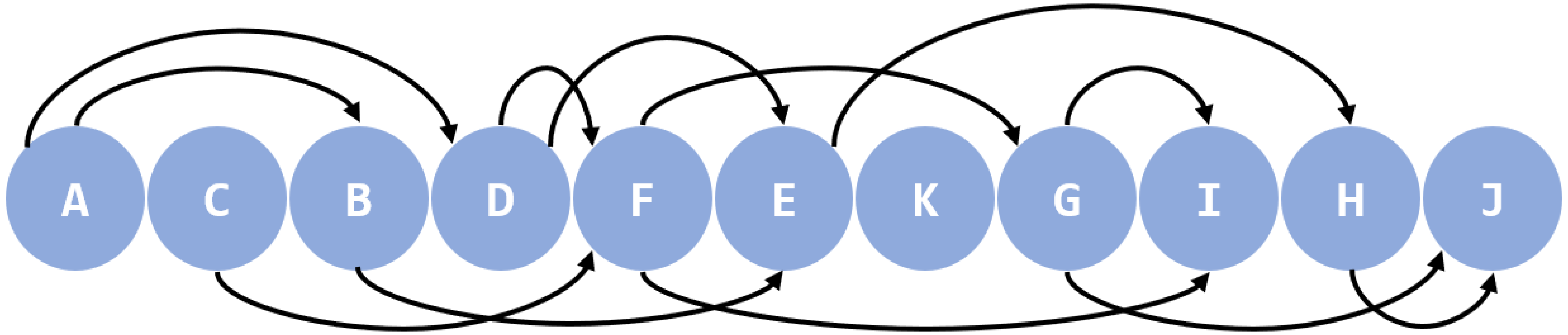
Part 2: GRAPH'S ALGORITHMS

Topological Sort

Problem Statement

Technical Definition

Algorithm



A **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge (u,v) from vertex u to vertex v, u comes before v in the ordering.

The **topological sort** algorithm can find a topological ordering in **O(V + E)** time!

NOTE: Topological orderings are not unique!

Part 2: GRAPH'S ALGORITHMS

Topological Sort

Problem Statement

Technical Definition

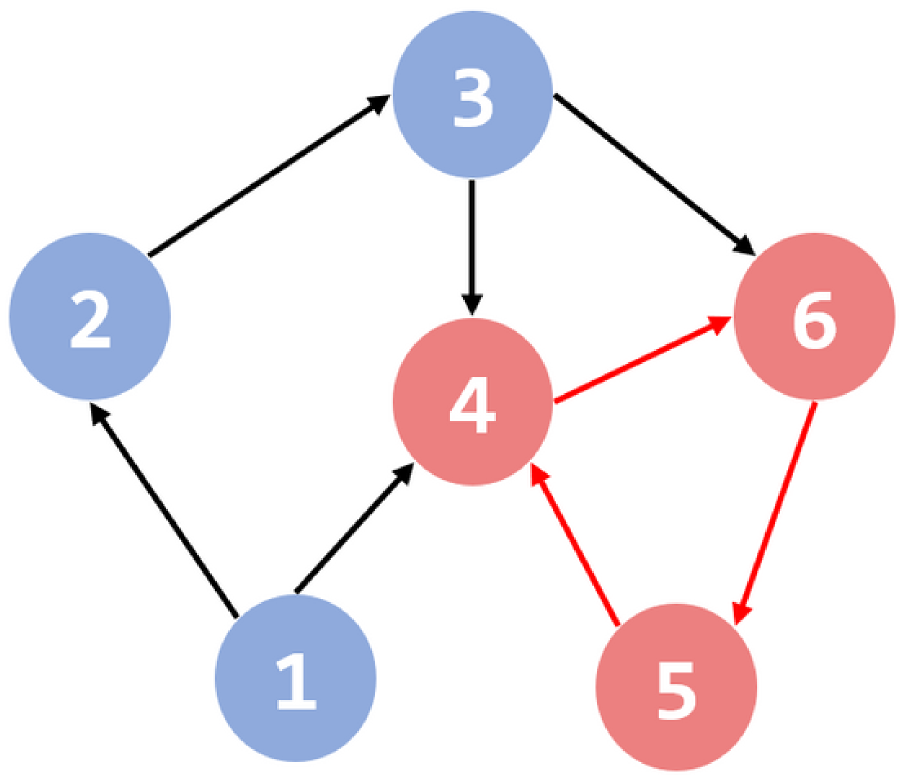
Algorithm

Does every graphs have topological order?

No!

The only type of graph which has valid topological order is a Directed Acyclic Graph (DAG) - directed edges and no cycle!

Q: How do I verify if my graph has a cycle?



A:

Tarjan`s Algorithm strongly connected components

Kahn`s Algorithm

...

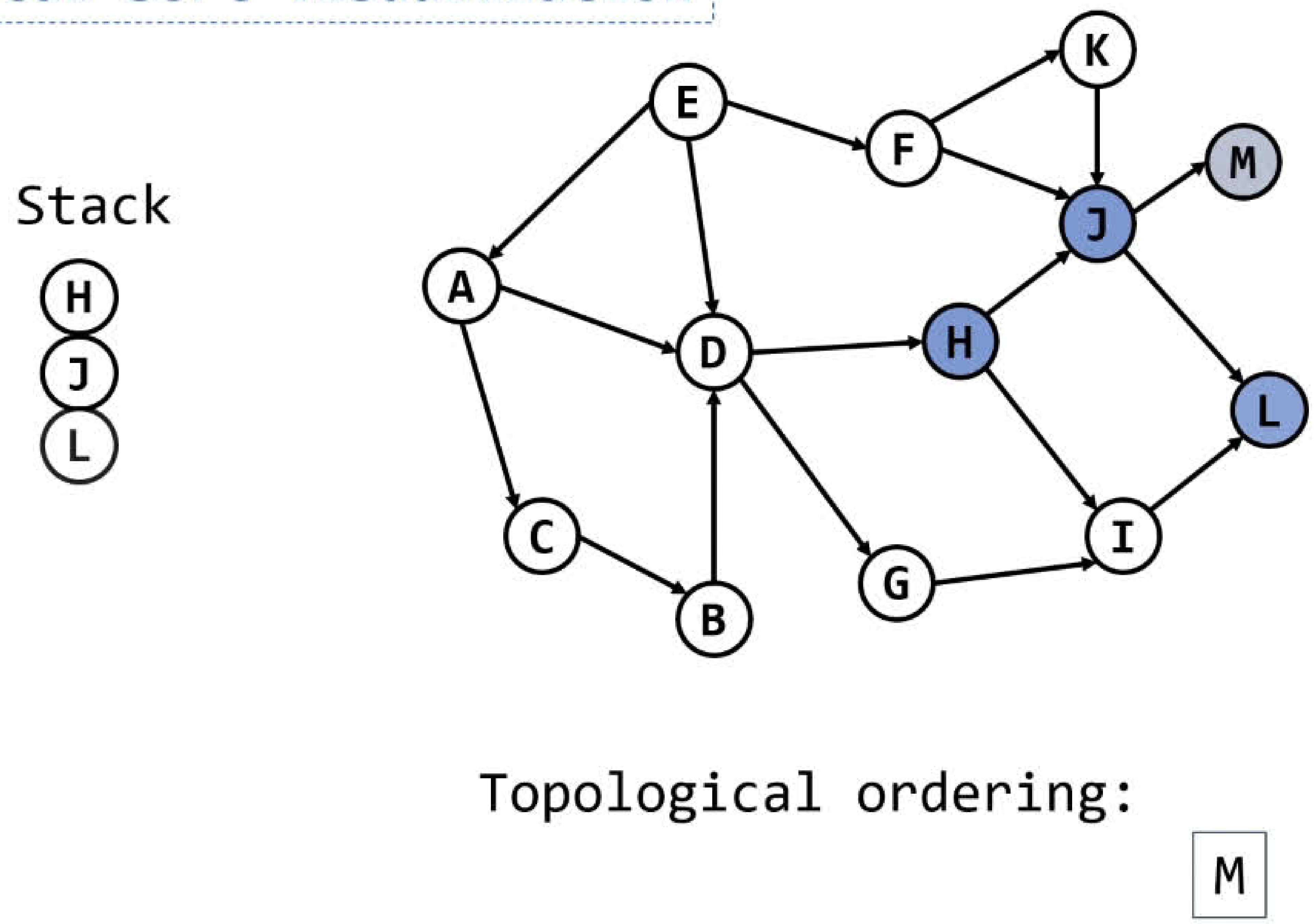
Topological Sort

1. Pick an unvisited node.
2. Beginning with the selected node, start a DFS on only unvisited nodes.
3. On the recursive callback of the DFS, add the current node to the topological ordering in reverse order.

Topological Sort

Problem Statement
Technical Definition
Algorithm

Topological Sort Visualization



Topological Sort

Problem Statement

Technical Definition

Algorithm

```
function topo_sort():
  visited <- a list of size |V| with all 0
  result <- an empty list
  for each vertex v in the graph:
    if not visited[v]:
      topo_sort_helper(v)

  function topo_sort_helper(v):
    visited[v] <- 1
    for u in adjacent vertices of v:
      if not visited[u]:
        topo_sort_helper(v)
    add v to the beginning of result

  return result in reverse
```

Part 2: GRAPH'S ALGORITHMS

Tarjan's Algorithm

Directed Graph -
Strongly connected components

Undirected Graph -
Bridges and Articulation Points

Part 2: GRAPH'S ALGORITHMS

Tarjan's Algorithm in directed graph

Some Definations

Problem Statement

Algorithm

- The Stack Invariant
- New low-link update condition
- Overview
- Visualization
- Pseudocode

Part 2: GRAPH'S ALGORITHMS

Tarjan’s Algorithm in directed graph

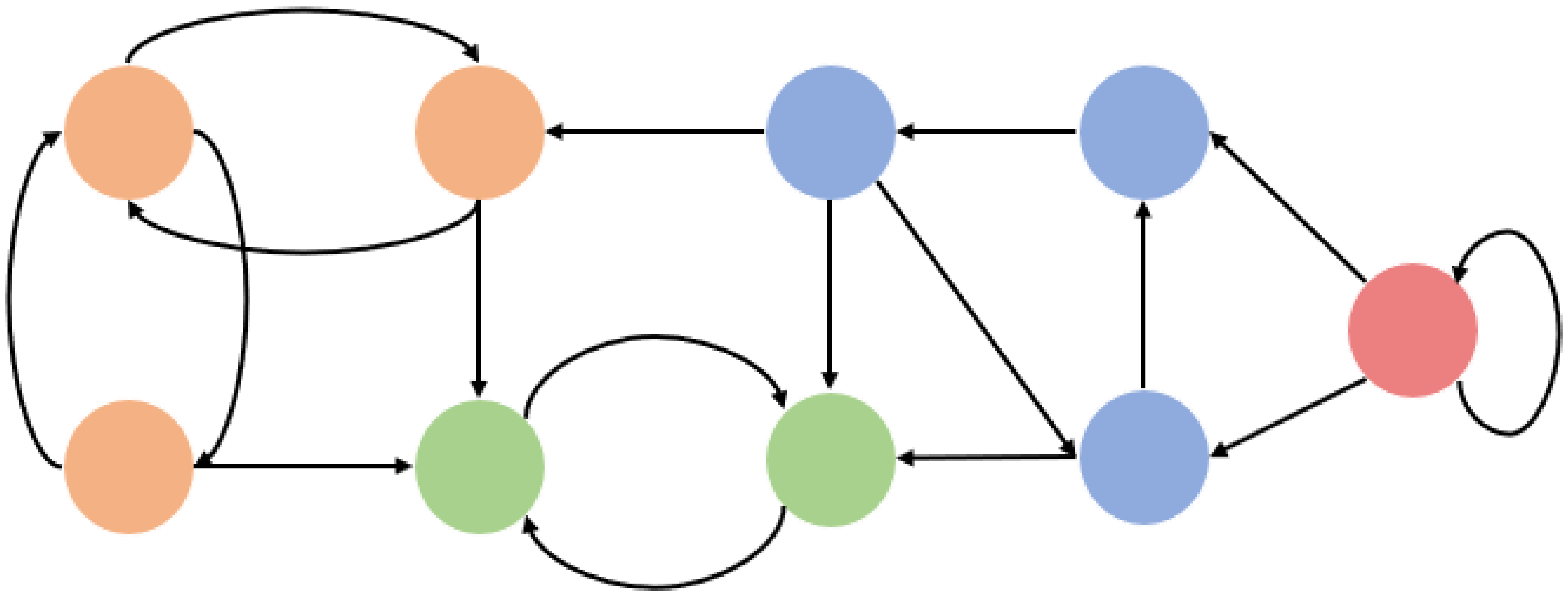
Some Definations

Problem Statement

Algorithm

What are Strongly Connected Components (SCCs)?

SCCs can be thought of as self - contained cycle within a directed graph where every vertex in a given cycle can reach every other vertex in the same cycle.

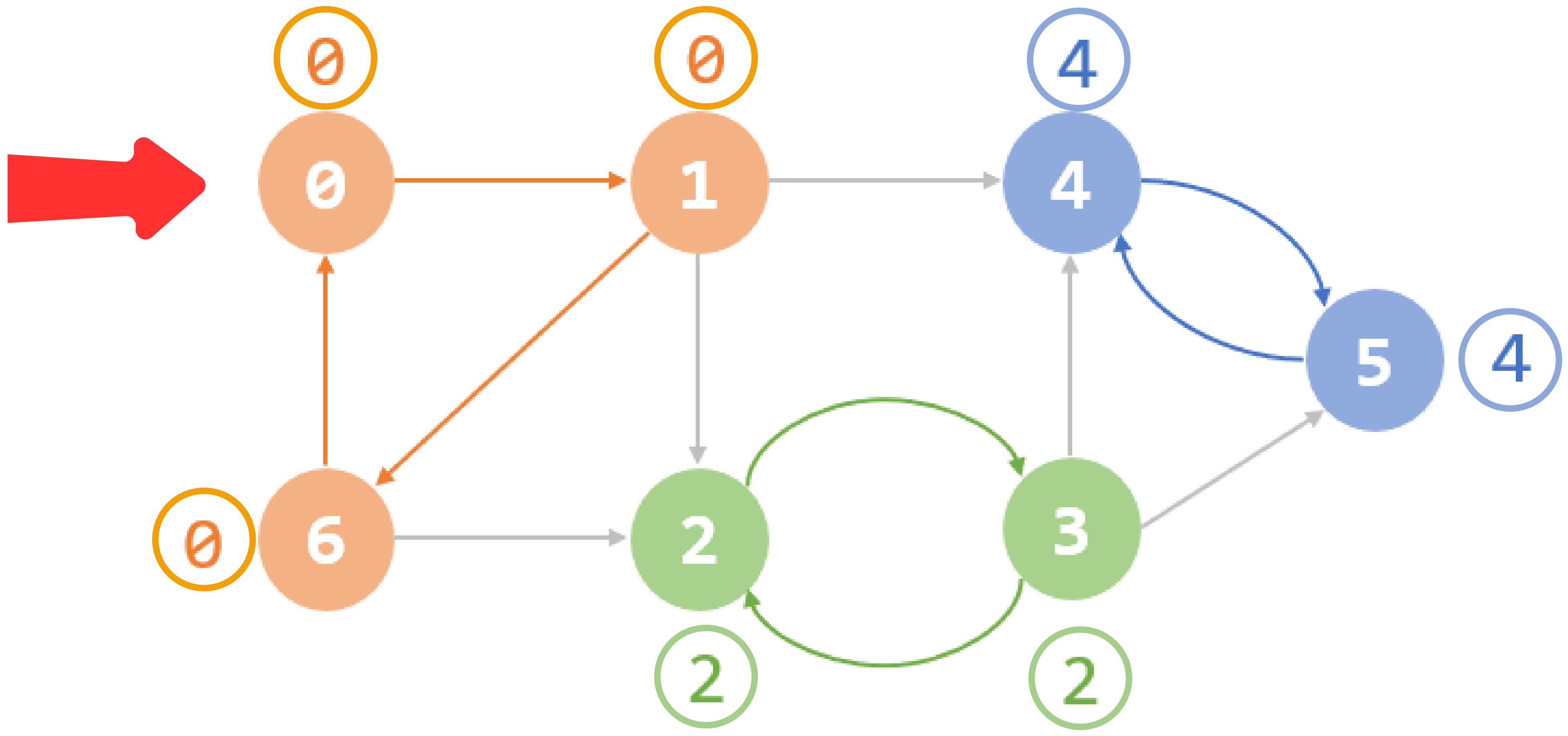


Tarjan's Algorithm in directed graph

Some Definations
Problem Statement
Algorithm

Low - link values

The low - link value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

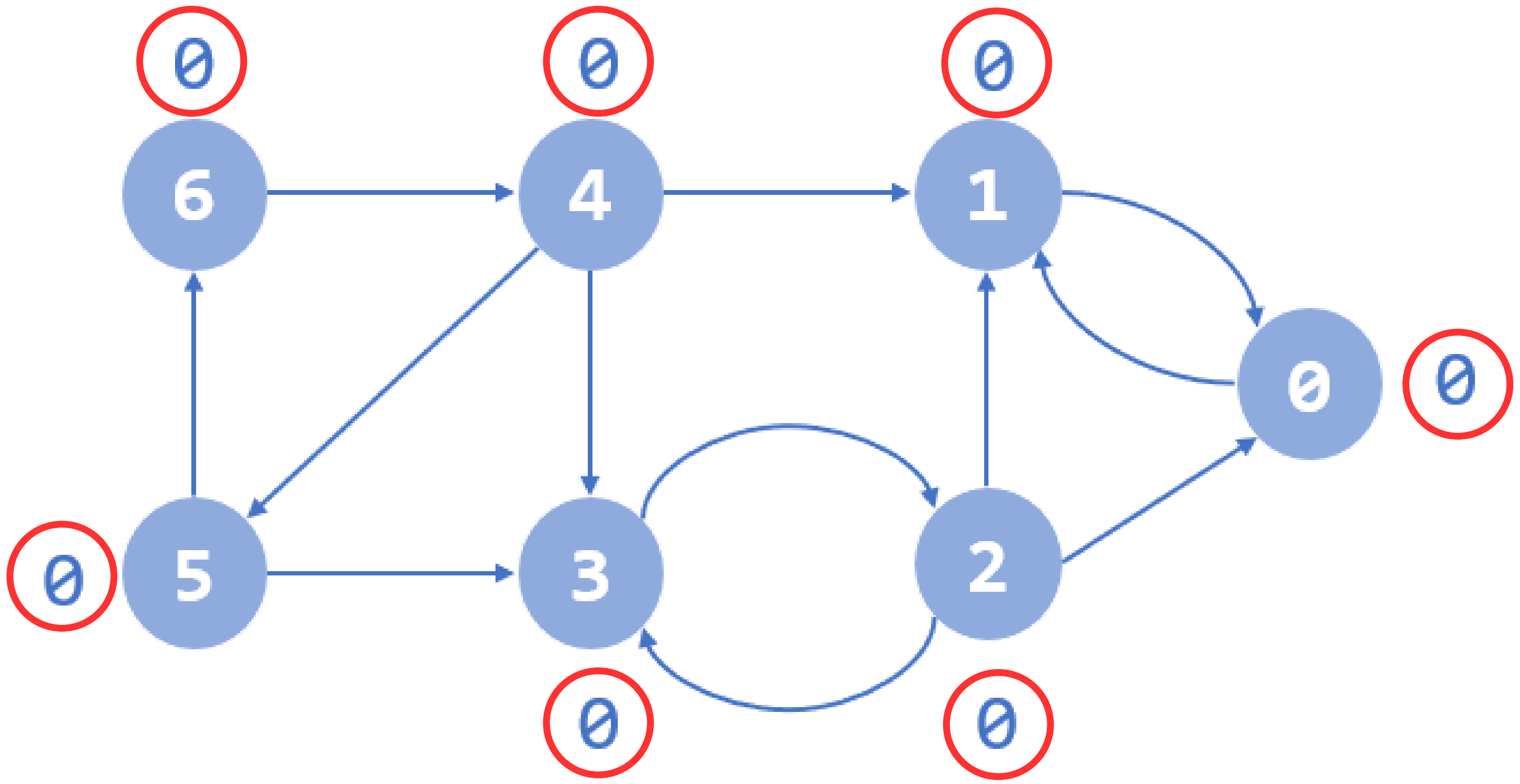


Tarjan's Algorithm in directed graph

Low - link values

The low - link value of a node is the smallest [lowest] node id reachable from that node when doing a DFS (including itself).

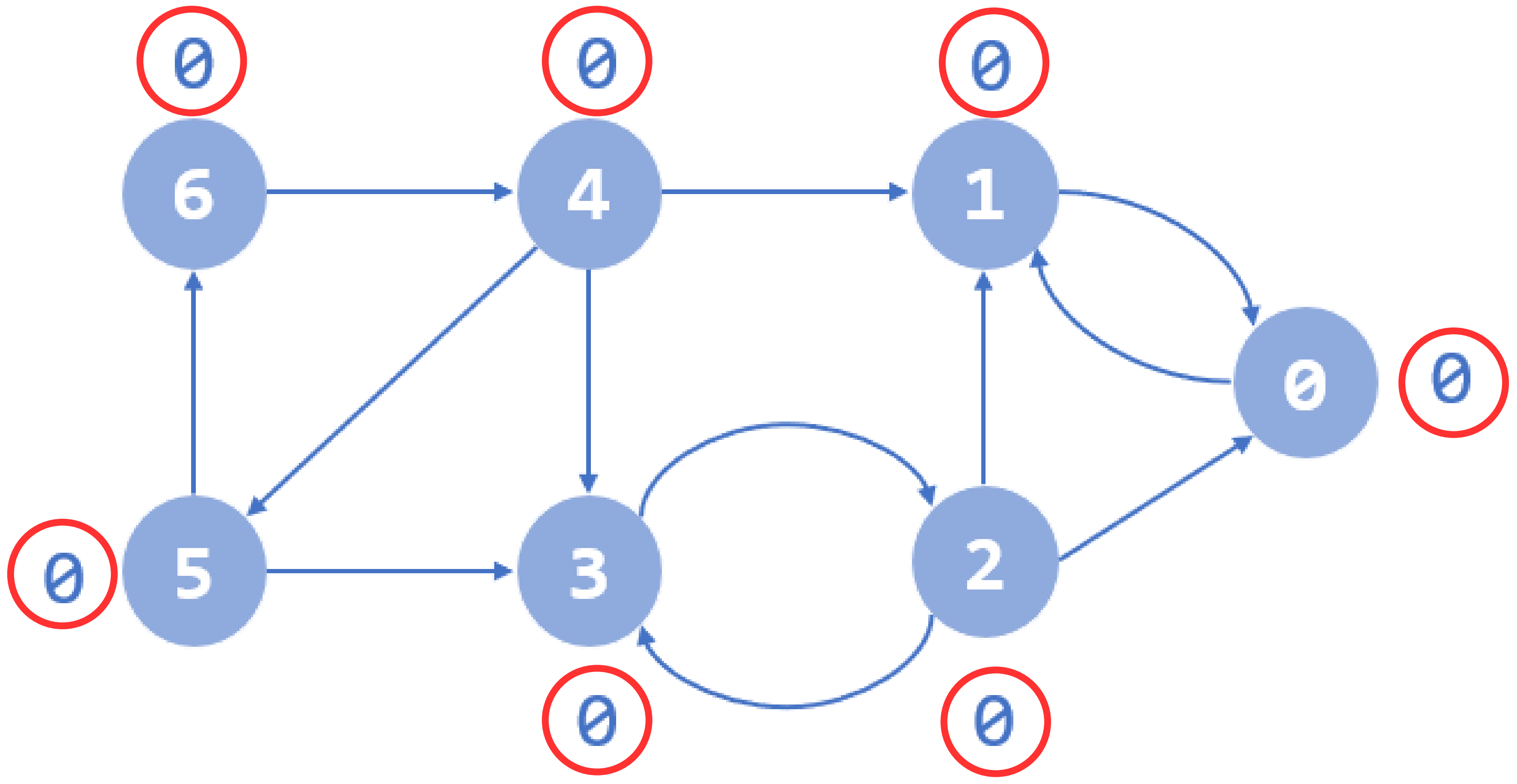
All low link values are the same but there are multiple SCCs!



Tarjan's Algorithm in directed graph

Low - link values

The **low - link value** of a node is the smallest [lowest] node id **reachable** from that node when doing a DFS (including itself).



Depending on where the DFS starts, and the order in which nodes/edges are visited, the low-link values for identifying SCCs could be **wrong**.

Tarjan's Algorithm

Tarjan's Algorithm in directed graph

Some Definations

Problem Statement

Algorithm

- New low-link update condition
- Overview
- Visualization
- Pseudocode

The Stack Invariant

To cope with the random traveral order of the DFS, Tarjan's Algorithm maintains a set (often as a stack) of valid nodes from which to update low-link values from.

STACK/SET

- **nodes are added as they're explored for the first time**
- **nodes are removed each time a SCC is found**

Tarjan’s Algorithm in directed graph

New low-link update condition

- u and v are nodes in a graph
- currently exploring u

To update u’s low-link value to node v’s low-link value:

- there has to be a path of edges from u to v
- node v must be on the stack

Part 2: GRAPH'S ALGORITHMS

Tarjan's Algorithm in directed graph

Some Definations

Problem Statement

Algorithm

- The Stack Invariant
- New low-link update condition
-
- Visualization
- Pseudocode

Overview

- Mark all nodes as unvisited.
- Start DFS.
- Upon visiting a node assign it an id and a low-link value, also mark current nodes as visited and add them to stack.
- On DFS backtrack, if the previous node is on the stack => min(current node's low-link value, last node's low-link value) *
- After visiting all neighbors, if the current node started a connected component **

=> pop nodes off stack until current node is reached.

*: this allows low-link values are maintained throughout cycles.

**: if its id == its low-link value

Part 2: GRAPH'S ALGORITHMS

Tarjan's Algorithm in directed graph

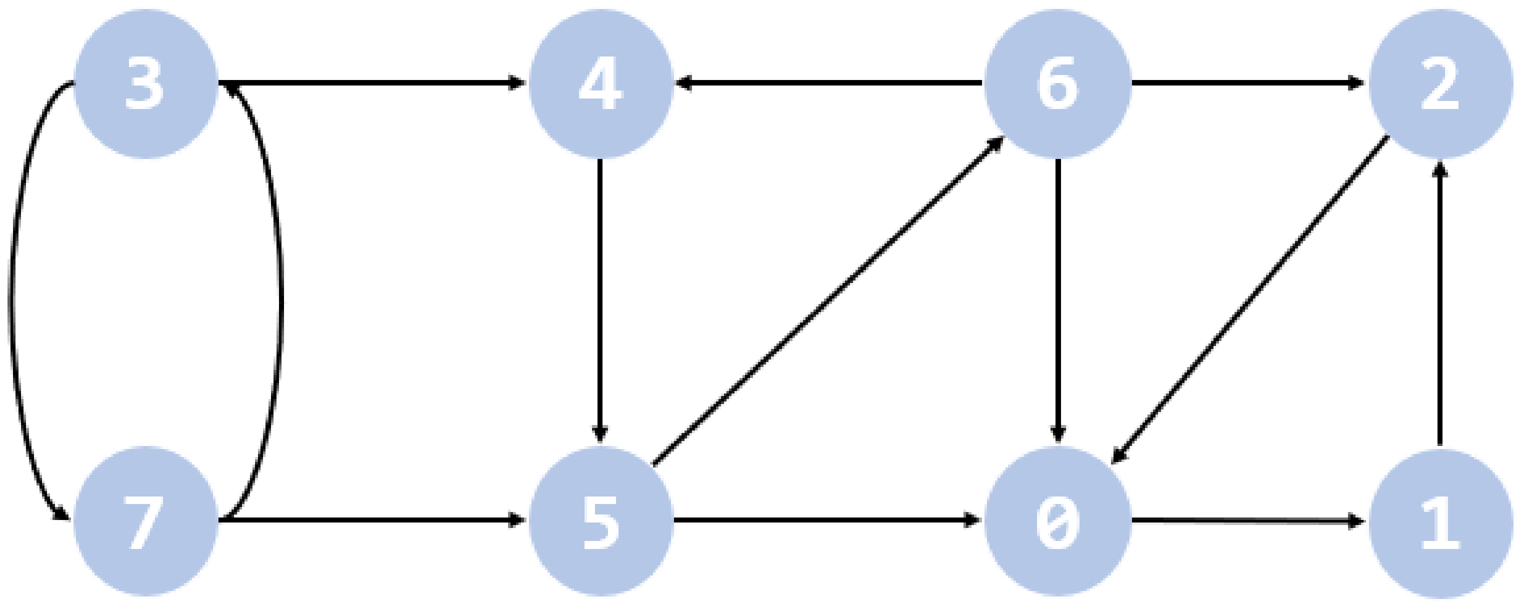
- Some Definations
- Problem Statement
- **Algorithm**
  - The Stack Invariant
  - New low-link update condition
  - Overview
  -
  - Pseudocode

Visualization

Tarjan's Algorithm Visualization

Unvisited   Visiting neighbors   Visiting all neighbors

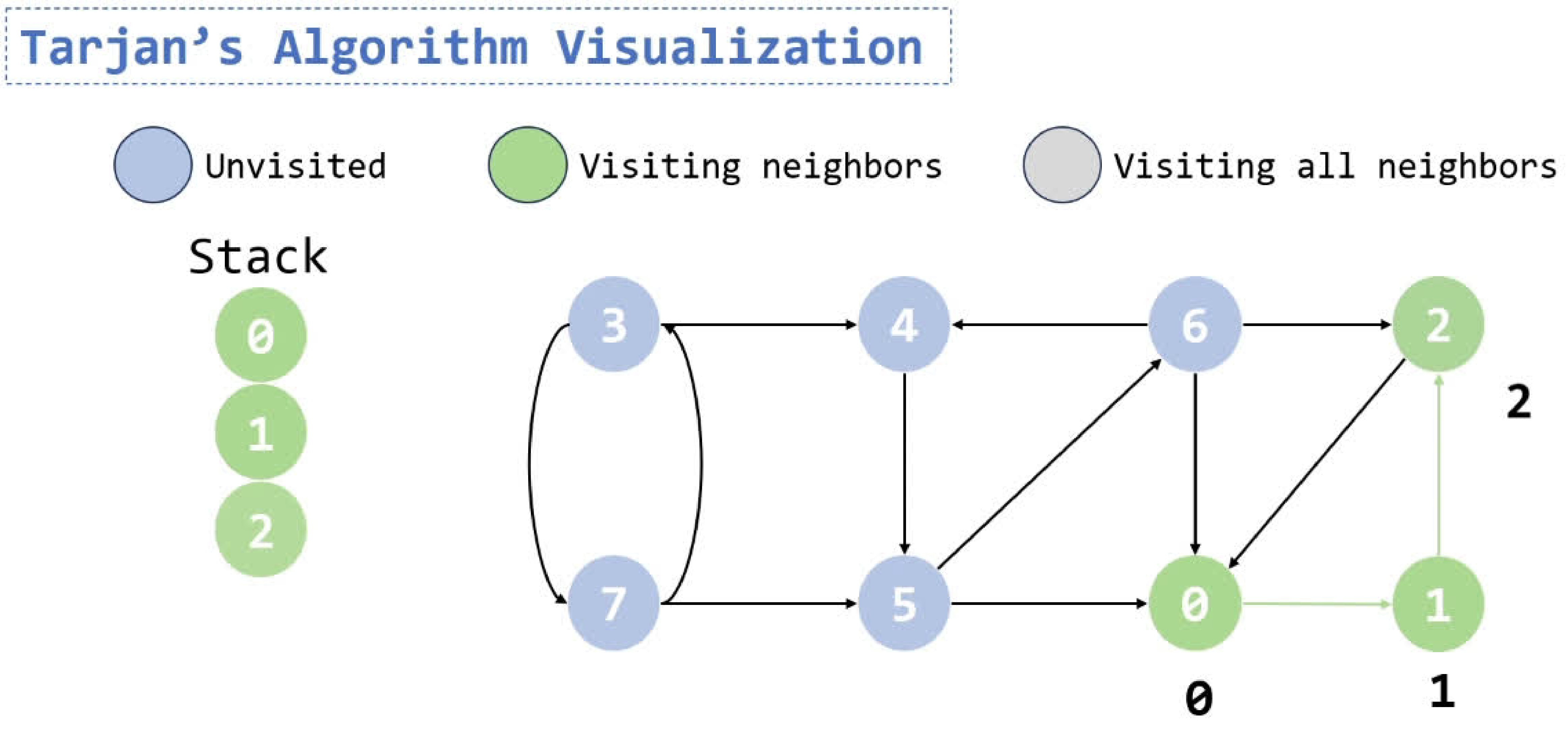
Stack



If a node is green or gray
=> it is on the stack
=> can update its low-link value

Tarjan’s Algorithm in directed graph

Visualization



Tarjan's Algorithm in directed graph

Some Definations

Problem Statement

Algorithm

- The Stack Invariant
- New low-link update condition
- Overview
- Visualization

Pseudocode

```
function scc():
  visited <- a list of size |V| with all Ø
  order <- a list of size |V| with all -1
  low_link <- a list of size |V| with all -1
  current <- Ø
  st <- an empty stack
  on_st <- a list of size |V| with all Ø
  scc_count <- Ø
  for each vertex v in the graph:
    if not visited[v]:
      scc_helper(v)
  return scc_count
```

Part 2: GRAPH'S ALGORITHMS

Tarjan's Algorithm in directed graph

Pseudocode

```
function scc_helper(v):
  add v into st
  visited[v] <- 1
  on_st[v] <- 1
  order[v] <- current
  low_link[v] <- current
  current <- current + 1
  for each adjacent vertex u of v:
    if not visited[u]:
      scc_helper(u)
      low_link[v] <- min(low_link[v], low_link[u])
    else if on_st[u]:
      low_link[v] <- min(low_link[v], order[u])
  if low_link[v] = order[v]:
    repeat:
      u <- extract top from st
      on_st[u] <- Ø
    until u = v
    scc_count <- scc_count + 1
```

Tarjan's Algorithm in undirected graph

- Some Definations
- Conditions
- Algorithm
  - Overview
  - Visualization
  - Pseudocode

Part 2: GRAPH'S ALGORITHMS

Tarjan’s Algorithm in undirected graph

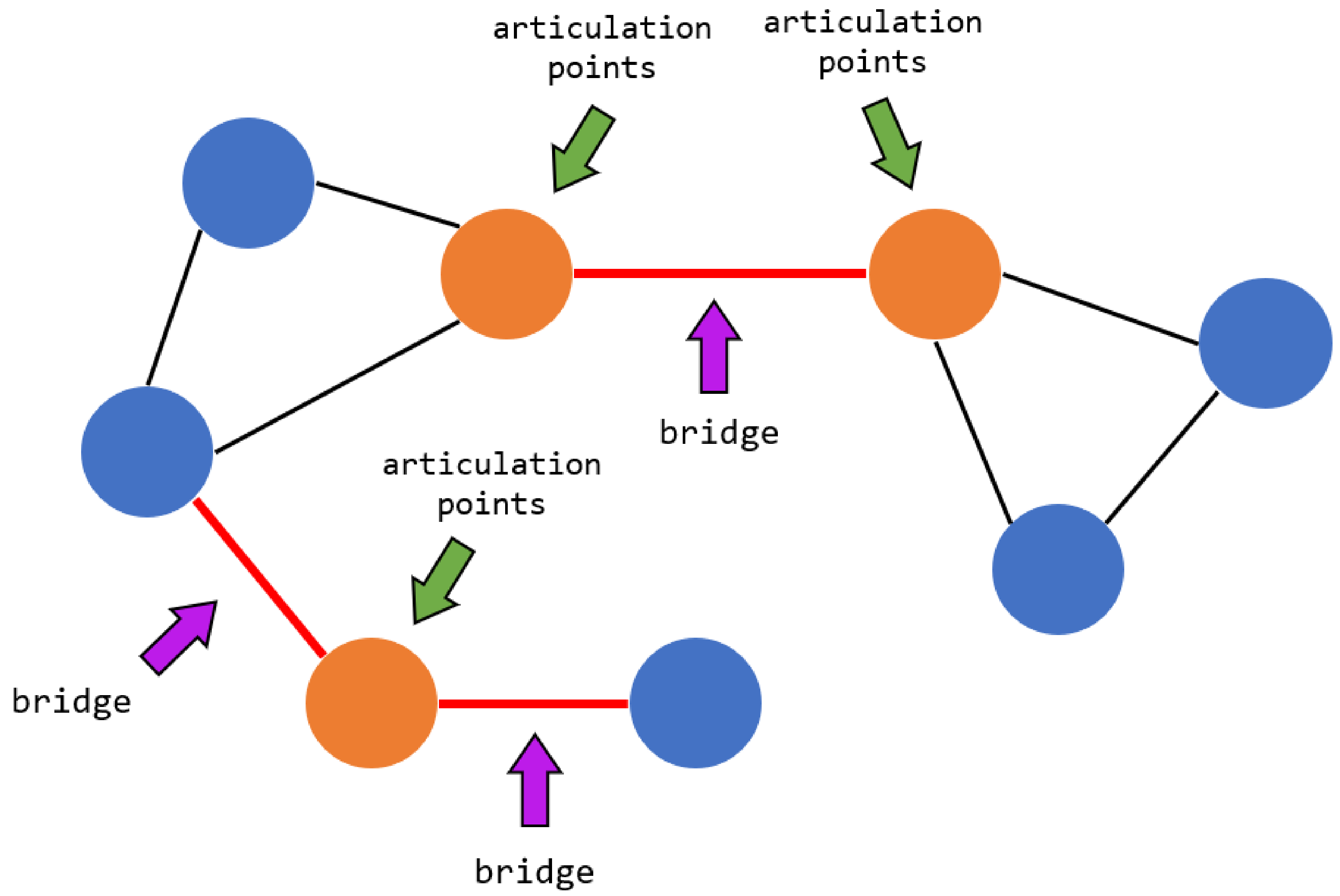
Some Definations

Condition

Algorithm

What are Bridges and Articulation points?

A Bridge / an Articulation point is an edge / a vertex which, when removed, increases the number of connected components in the graph.



Some Definations

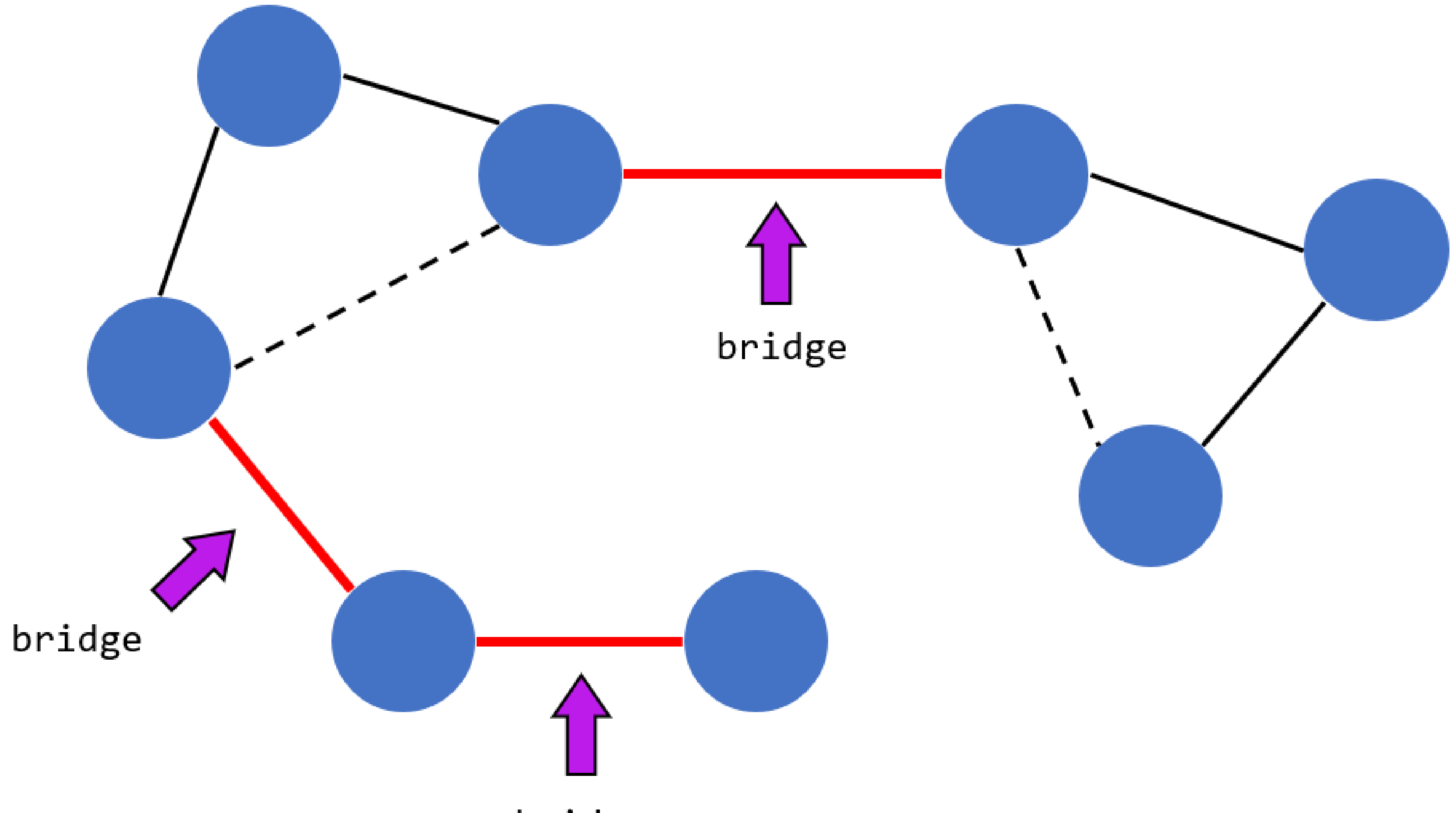
Conditions

Algorithm

Tarjan’s Algorithm in undirected graph

How do we know if an edge is a bridge?

If on the DFS tree, vertex v is the parent of vertex u, the edge connecting u and v is a bridge if and only if u doesn’t have any other links to v or any ancestors of v.

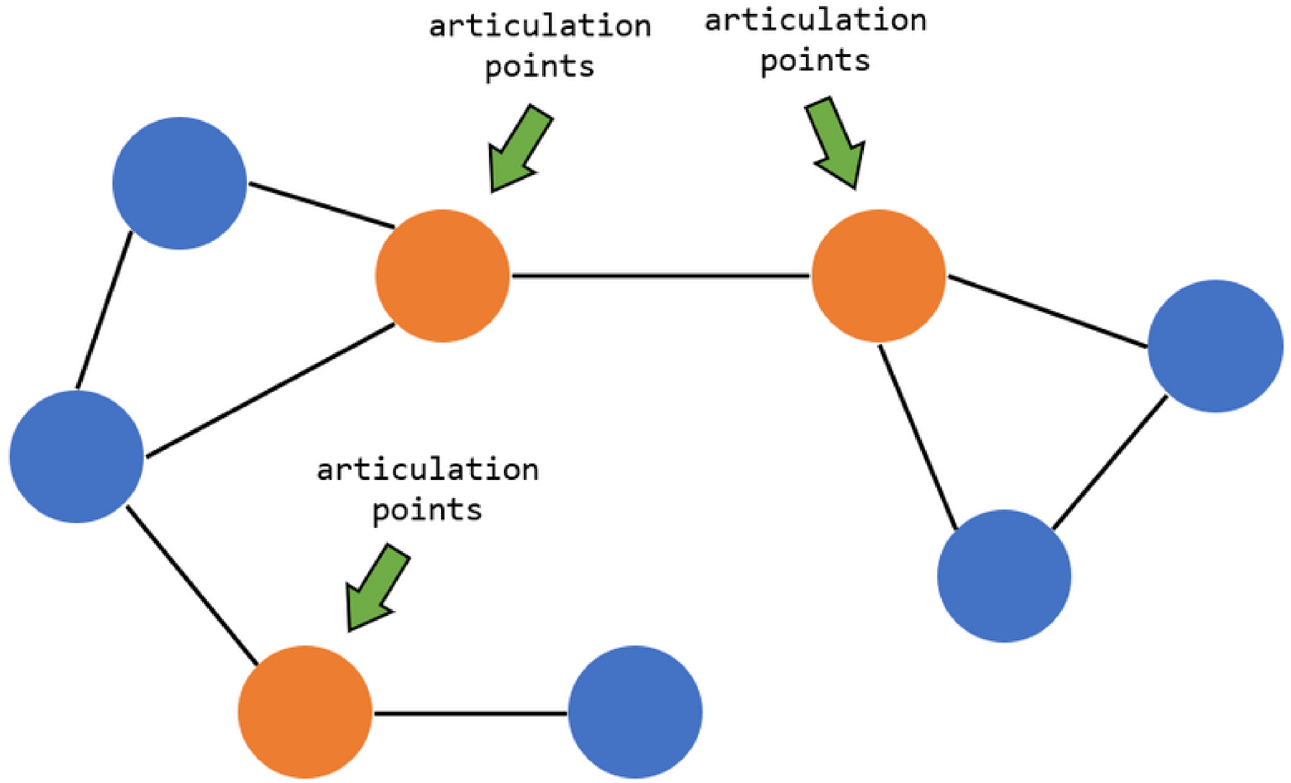


Tarjan's Algorithm in undirected graph

How do we know if a vertex is an articulation point?

A vertex v of the DFS tree is an articulation point if and only if one of the followings happens:

- v is the root and have more than 1 child
- v is not the root and a child of v does not have any other links to any ancestors of v



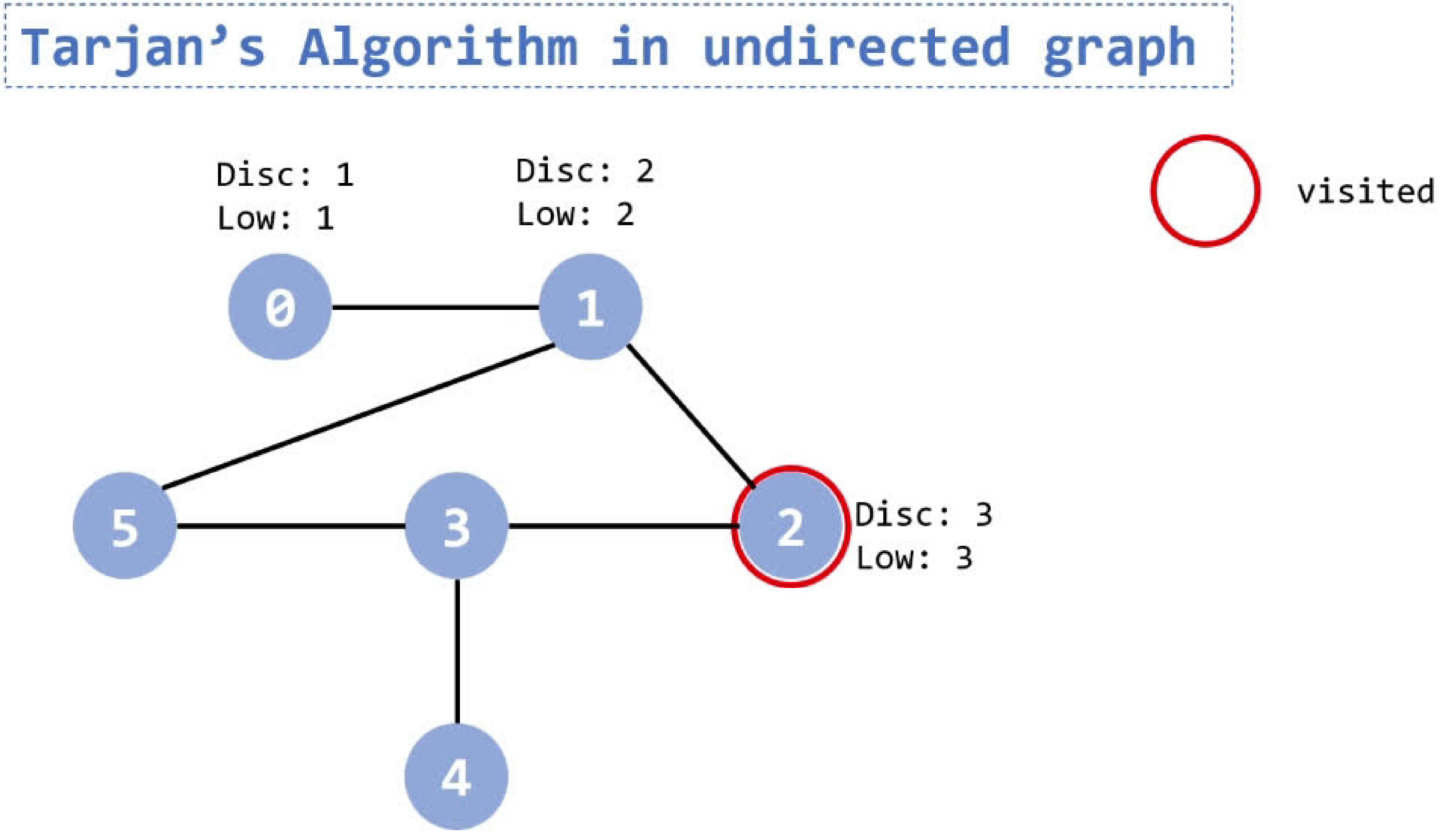
Tarjan's Algorithm in undirected graph

Overview

- If we consider the DFS tree as a DAG, we can use the same approach except that now we want to find the lowest link possible so the stack is not needed.
- We will use an array to keep track of the parent of each vertex.

Tarjan's Algorithm in undirected graph

Visualization



Tarjan’s Algorithm in undirected graph

Some Definations
Conditions
Algorithm
Overview
Visualization

Pseudocode

```
function bridges():
  visited <- a list of size |V| with all 0
  order <- a list of size |V| with all -1
  low_link <- a list of size |V| with all -1
  parent <- a list of size |V| with all -1
  current <- 0
  bridge_count <- 0
  for each vertex v in the graph:
    if not visted[v]:
      bridge_helper(v)
  return bridge_count
```

Part 2: GRAPH'S ALGORITHMS

Tarjan's Algorithm in undirected graph

Pseudocode

```
function bridge_helper(v):
  visited[v] <- 1
  order[v] <- current
  low_link[v] <- current
  current <- current + 1
  for each adjacent vertex u of v:
    if not visited[u]:
      parent[u] <- v
      bridge_helper(u)
      low_link[v] <- min(low_link[v], low_link[u])
      if low_link[u] > order[v]:
        bridge_count <- bridge_count + 1
    else if parent[v] != u
      low_link[v] <- min(low_link[v], order[u])
```

Tarjan’s Algorithm in undirected graph

Pseudocode

```
function ap():
  visited <- a list of size |V| with all Ø
  order <- a list of size |V| with all -1
  low_link <- a list of size |V| with all -1
  parent <- a list of size |V| with all -1
  current <- Ø
  ap_count <- Ø
  branches <- Ø
  root <- -1
  for each vertex v in the graph:
    if not visited[v]:
      branches <- Ø
      root <- v
      ap_helper(v)
  return ap_count
```

Part 2: GRAPH'S ALGORITHMS

Some Definations

Conditions

Algorithm

- Overview
- Visualization

Tarjan's Algorithm in undirected graph

Pseudocode

```
function ap_helper(v):
  visited[v] <- 1
  order[v] <- current
  low_link[v] <- current
  current <- current + 1
  is_ap <- Ø
  for each adjacent vertex u of v:
    if not visited[u]:
      parent[u] <- v
      ap_helper(u)
      if v = root:
        branches <- branches + 1
        if branches > 1:
          is_ap = 1
      else:
        low_link[v] <- min(low_link[v], low_link[u])
        if low_link[u] >= order[v]:
          is_ap = 1
    else if parent[v] != u
      low_link[v] <- min(low_link[v], order[u])
  ap_count <- ap_count + is_ap
```

Part 2: GRAPH'S ALGORITHMS

Ford - Fulkerson's Algorithm

Some Definations

- Capacity, Source, Sink and Flow
- Augmenting path
- Residual Graph

Algorithm

- Updating capacity
- Finding augmenting paths
- Overview
- Visualization
- Pseudocode

Part 2: GRAPH'S ALGORITHMS

Ford - Fulkerson's Algorithm

Some Definations

- Capacity, Source, Sink and Flow
- Augmenting path
- Residual Graph

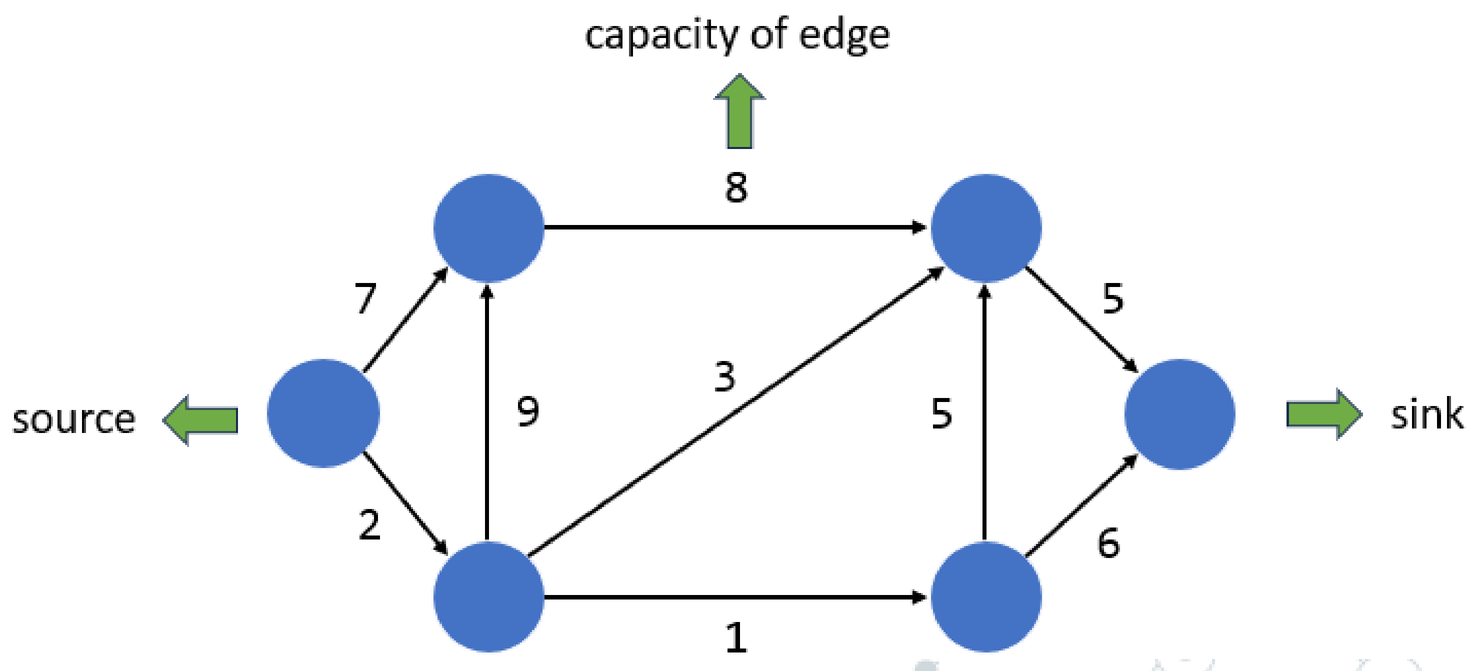
Algorithm

Capacity: the maximum amount of substance that can pass through a certain edge

Source / sink: sender / receiver of substance

Flow: a way by which substance flowing from source to sink, satisfying the following constraints:

- total substance from source = total substance to sink
- total substance to v = total substance from v (v is not source or sink) substance through an edge ≤ capacity of that edge



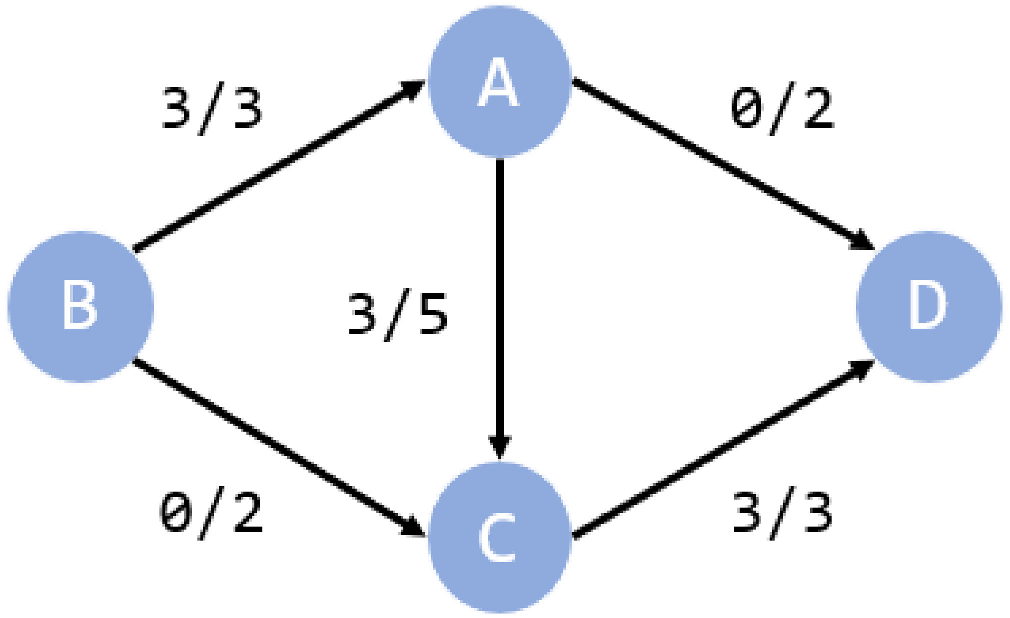
Part 2: GRAPH'S ALGORITHMS

Ford - Fulkerson's Algorithm

Some Definations

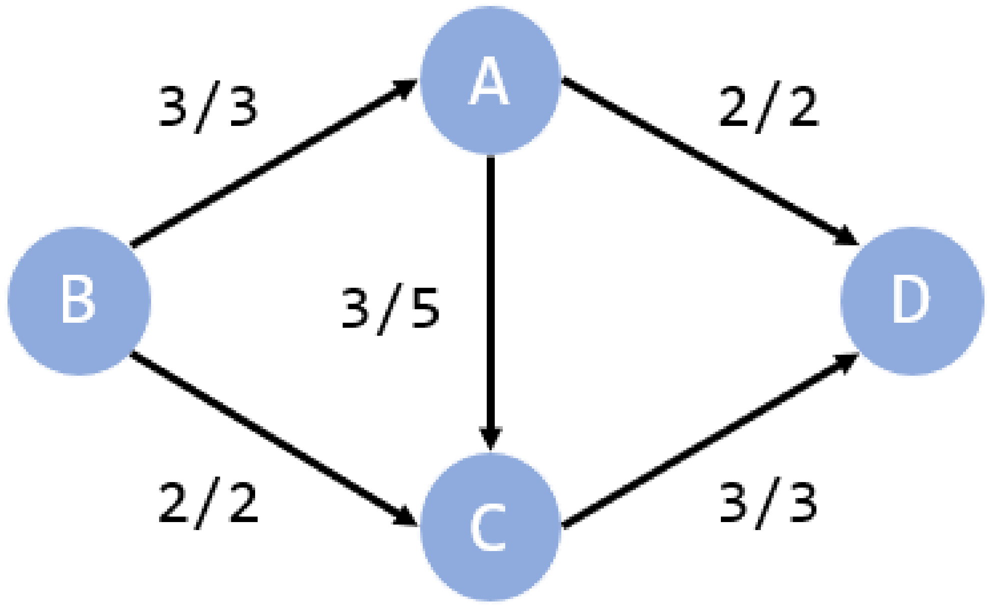
- Capacity, Source, Sink and Flow
- Augmenting path
- Residual Graph

Algorithm



flow value = 3

Flow value: the total amount of substance flowing from source to sink



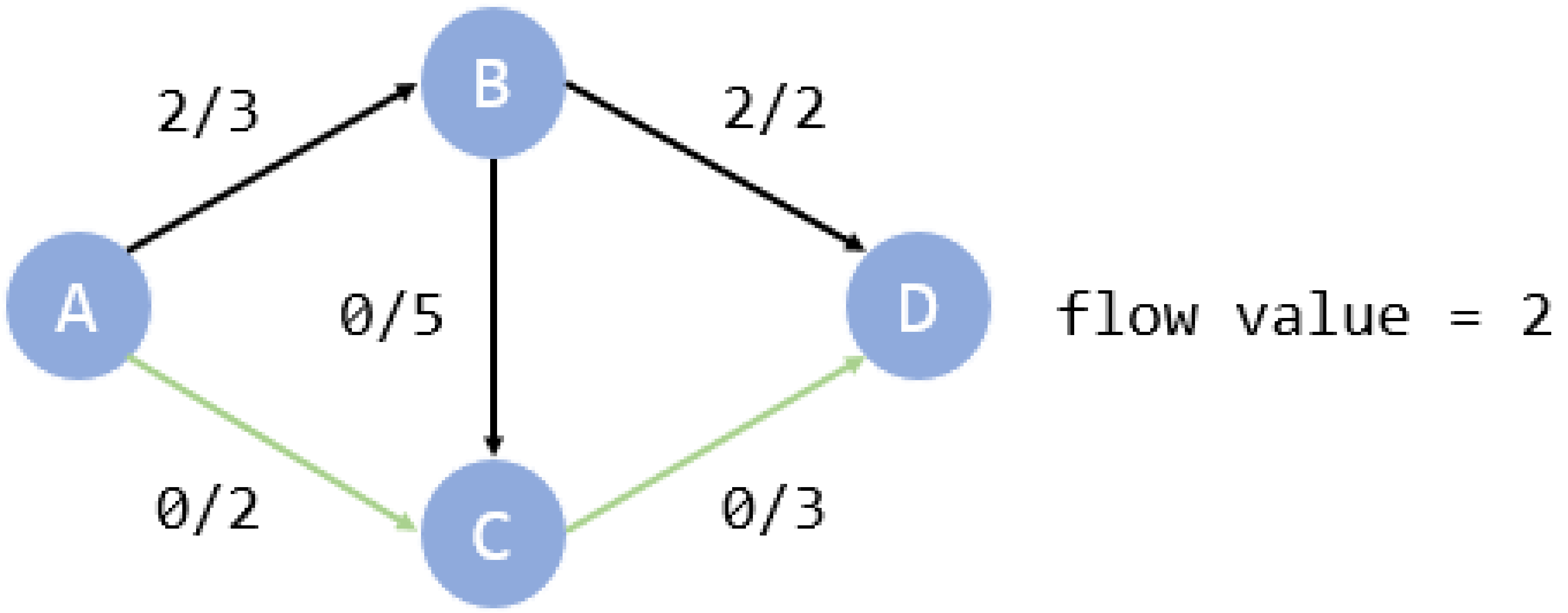
flow value = 5

Maximum flow: the flow which maximizes flow value

Ford - Fulkerson's Algorithm

Augmenting path

Augmenting path is a path from source to sink where more substance can be sent. The amount of substance must not be greater than the bottleneck of the path.



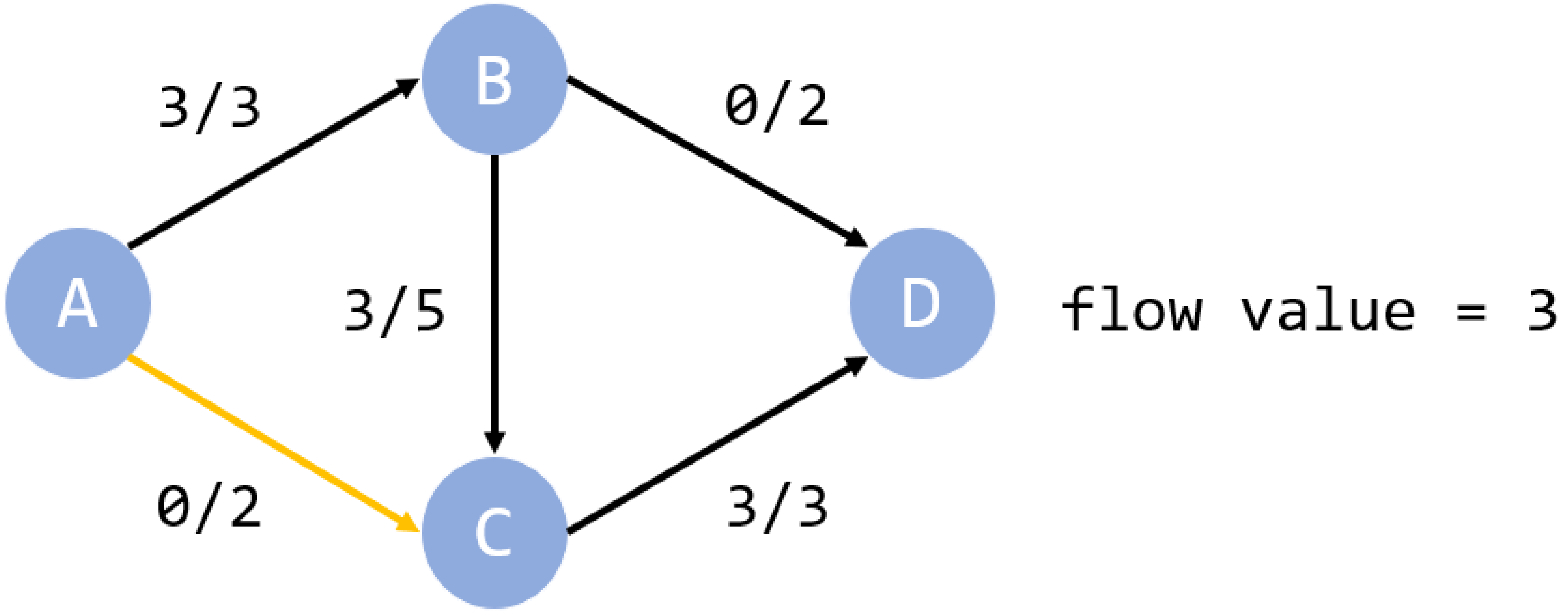
Ford - Fulkerson's Algorithm

Residual graph

Some Definations
- Capacity, Source, Sink and Flow
- Augmenting path
- Residual Graph

Algorithm

Assuming that the current flow **blocks all augmenting path** when we haven't yet found the maximum flow value. How do we cope with this?



Part 2: GRAPH'S ALGORITHMS

Ford - Fulkerson's Algorithm

Some Definations

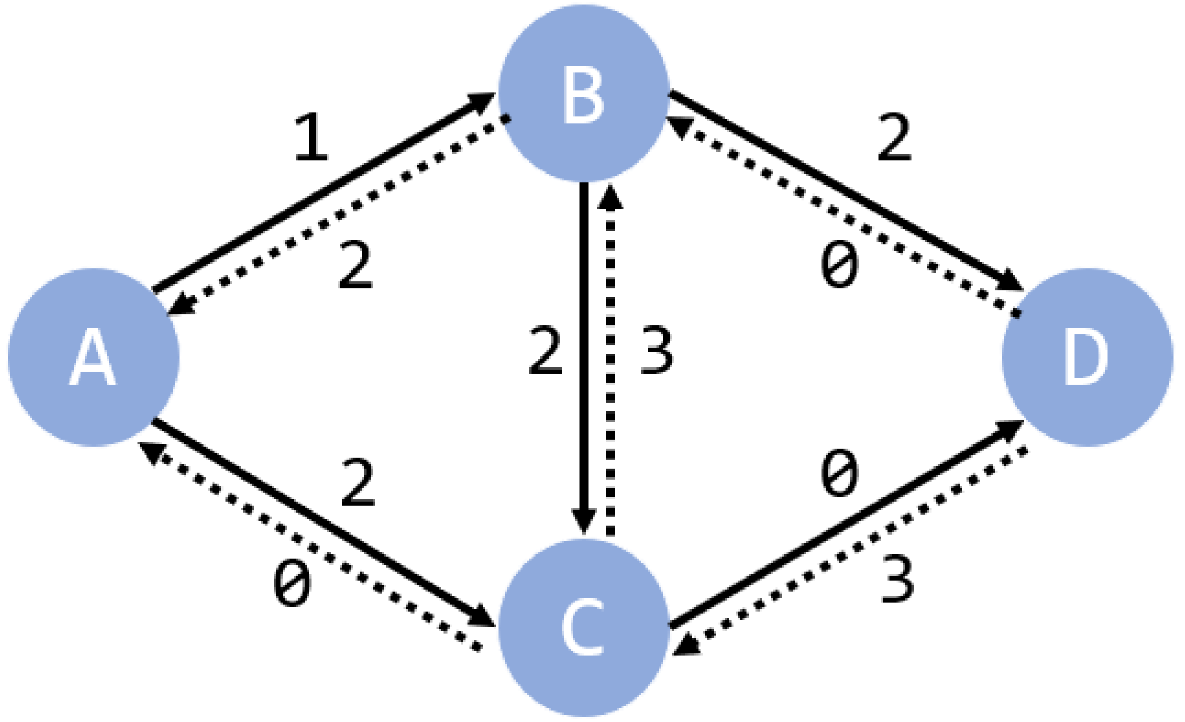
- Capacity, Source, Sink and Flow
- Augmenting path
- Residual Graph

Algorithm

Residual graph

Assuming that the current flow blocks all augmenting path when we haven't yet found the maximum flow value. How do we cope with this?

By adding reverse edges. The reverse edges allow us to "undo" bad path choices by "forcing" the substance to flow on other paths.



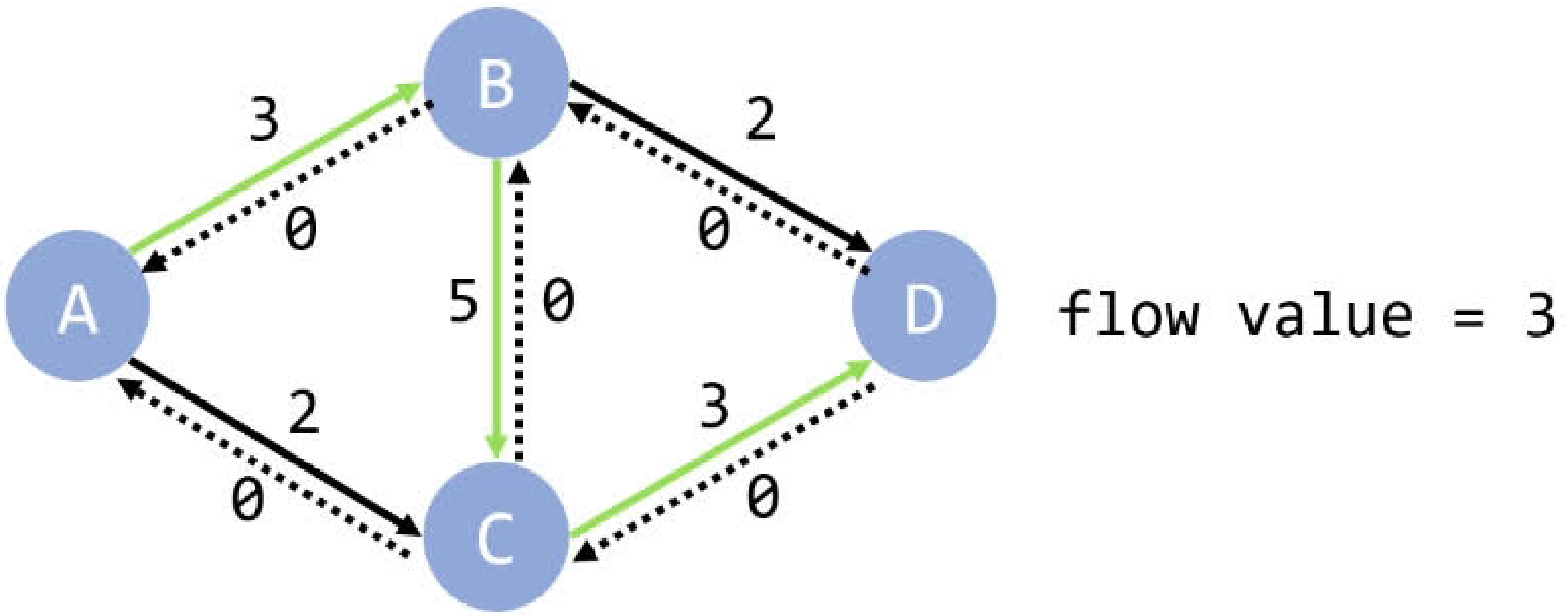
flow value = 2

Ford - Fulkerson's Algorithm

Updating capacity

Letting substance passing through an edge reduce its capacity but allow more substance to be forced out.

=> decrease capacity of the edge and increase capacity of the reverse edge.



Ford - Fulkerson's Algorithm

Updating capacity

Some Definations

Algorithm

Finding augmenting paths

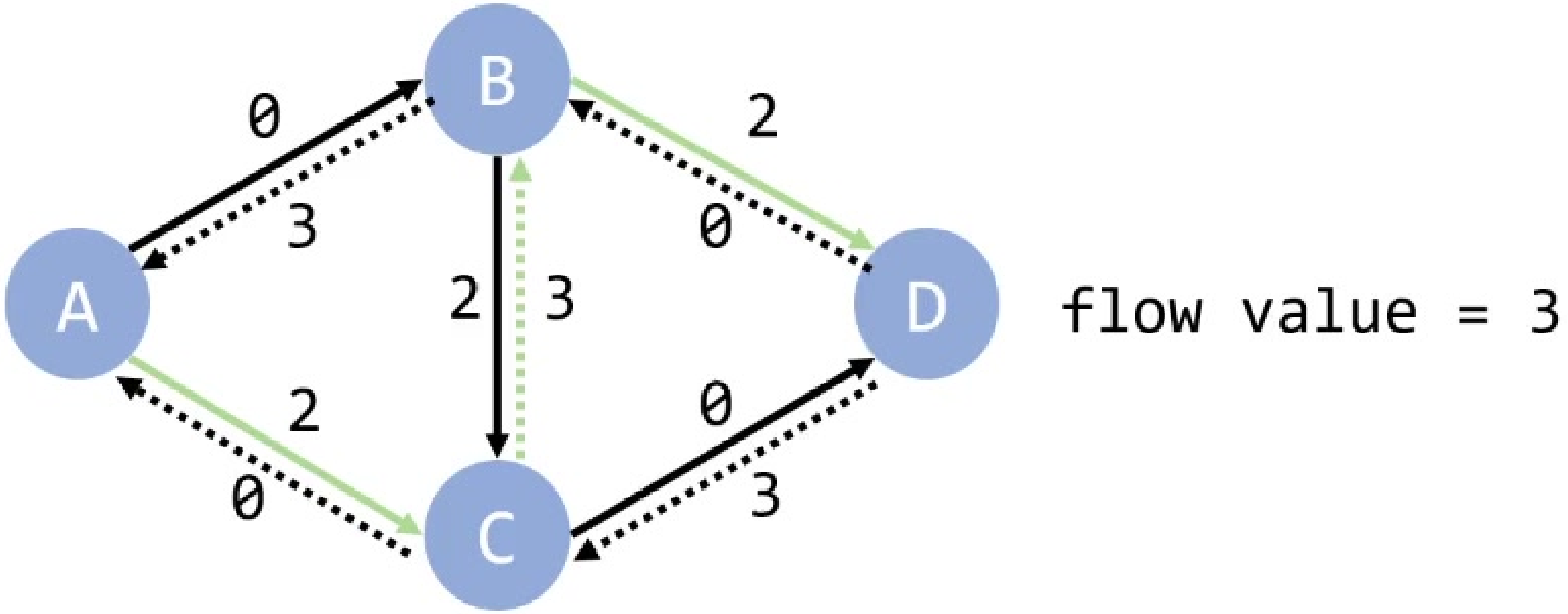
Overview

Visualization

Pseudocode

By constrast, force some substance out of an edge increase its capacity but leave the edges with less substance.

=> update capacity the opposite way.



Ford - Fulkerson's Algorithm

Finding augmenting paths

- **Find a path from source to sink which does not have any edges with capacity 0. Here we use DFS.**
- **Other methods of finding paths can also be used. If we use BFS, we have Edmond - Karp's algorithm.**

Ford Fulkerson's Algorithm

Overview

Ford - Fulkerson's Algorithm repeatedly finds augmenting paths using DFS and sends flow along the paths. When no more augmenting paths can be found, the algorithm ends.

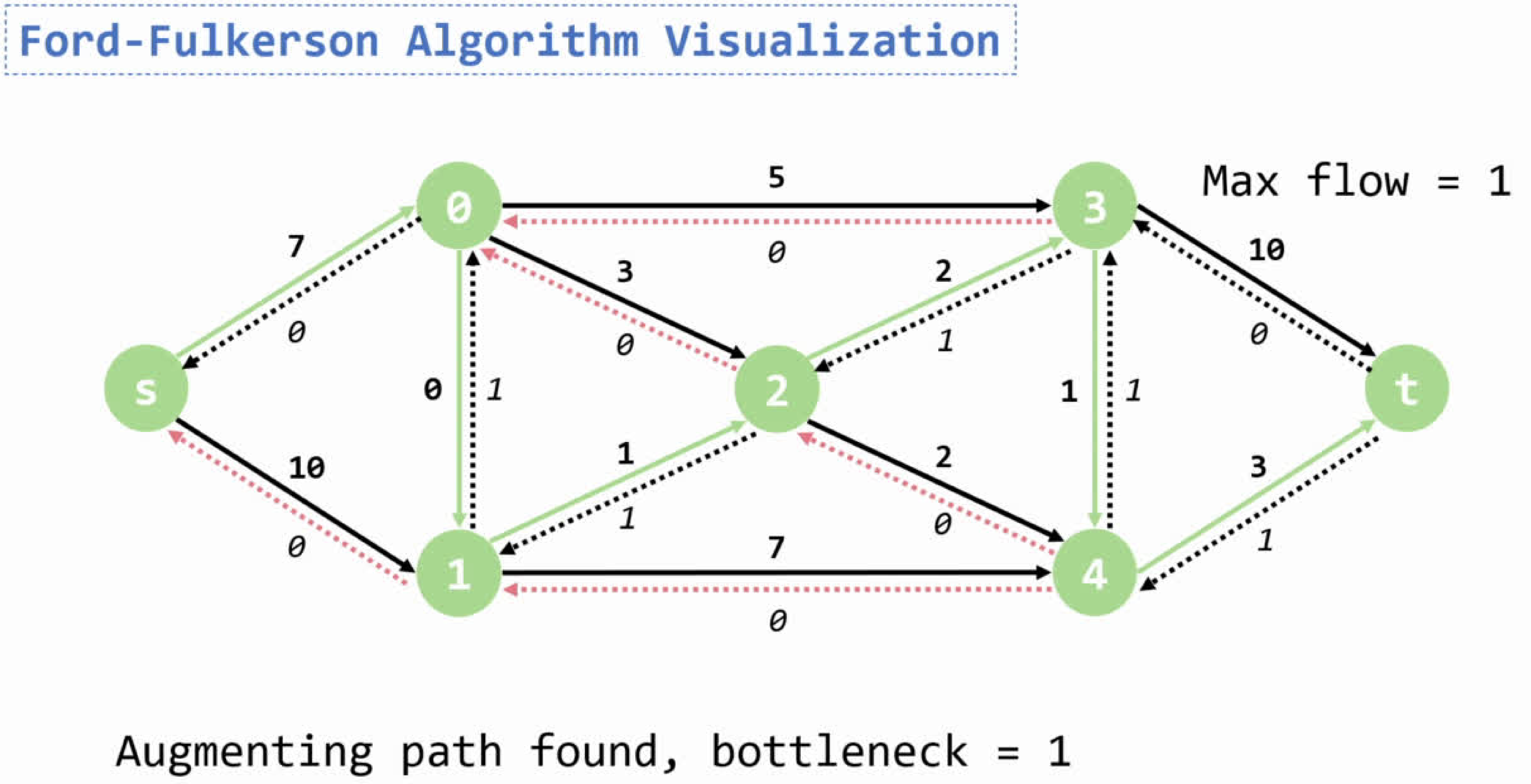
Part 2: GRAPH'S ALGORITHMS

Some Definations

Algorithm

- Updating capacity
- Finding augmenting paths
- Overview
-
- Pseudocode

Ford Fulkerson's Algorithm Visualization



Ford Fulkerson's Algorithm

Pseudocode

```
pre-condition:
  for each edge (u, v) of the graph:
    add edge (v, u) of capacity Ø to the graph
function ford_fulkerson():
  flow_value <- Ø
  visited <- a list of size |V| with all Ø
  parent <- a list of size |V| with all -1
  bottleneck <- infinity
  while find_aug_path(source):
    s <- sink
    while s != source:
      graph[parent[s]][s] <- graph[parent[s]][s] - bottleneck
      graph[s][parent[s]] <- graph[s][parent[s]] + bottleneck
    s <- parent[s]
    flow_value += bottleneck
    visited <- a list of size |V| with all Ø
    parent <- a list of size |V| with all -1
  return max_flow
```

Ford Fulkerson's Algorithm

Pseudocode

```
function find_aug_path(v)
  visited[v] <- 1
  for each adjacent vertex u of v and capacity c of connecting
edge:
    if not visited[u] and c > 0:
      parent[u] <- v
      if u = sink or find_aug_path(u):
        bottleneck <- min(bottleneck, c)
        return True
    return False
```

THANK YOU FOR LISTENING!