APPLICATION IN DESIGNING ALGORITHMS: GRAPH ALGORITHMS

GROUP 12

Lê Quang Thiên Phúc
Lý Nguyên Thùy Linh

TABLE OF CONTENTS

01 Overview

02 Graph's Algorithms

- Topological Sort
- Tarjan's Algorithm
- Ford - Fulkerson's Algorithm

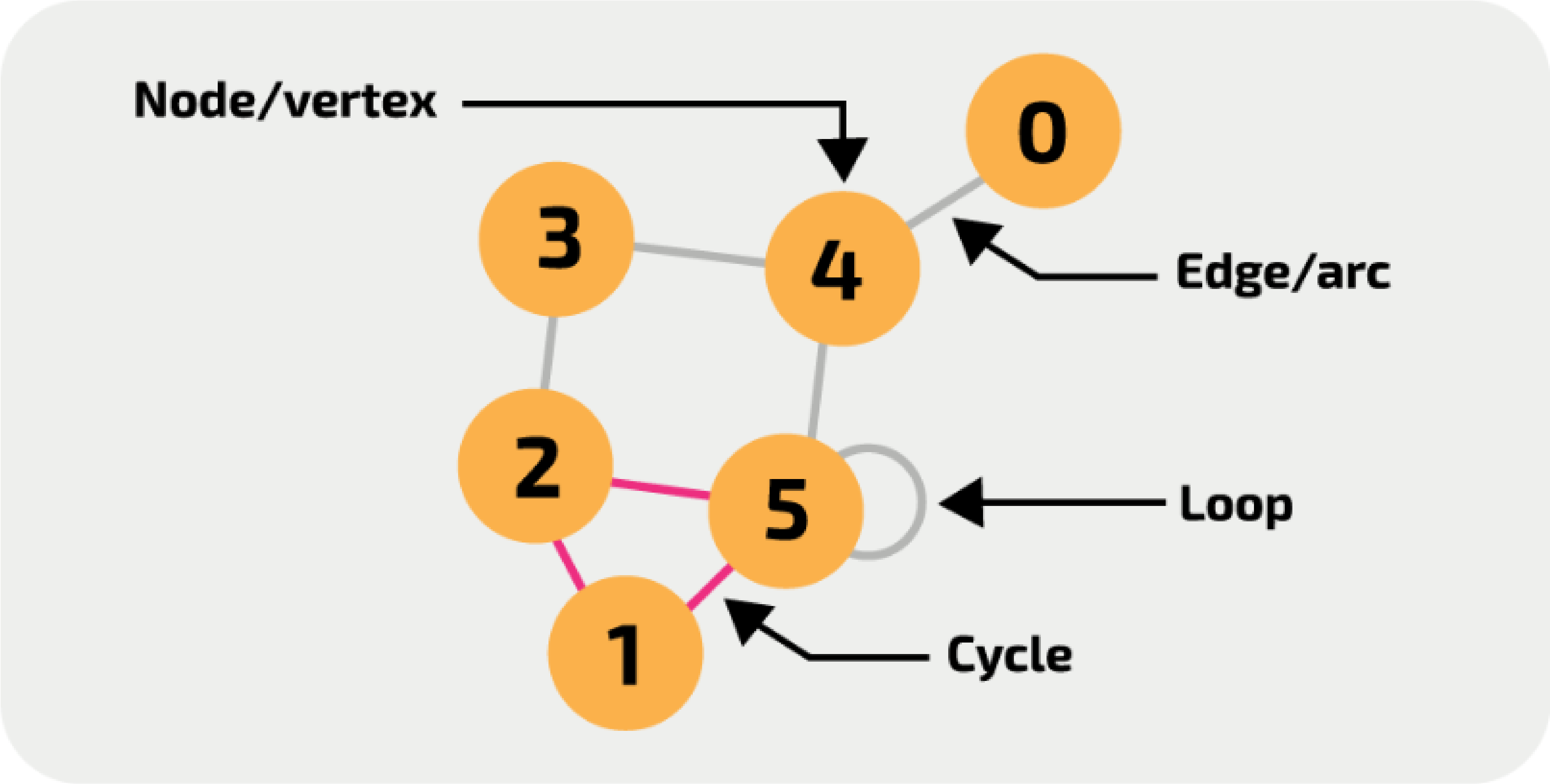
Part 1: OVERVIEW

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
disc	• 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

Algorithm

- Brute Force
- Backtracking
- Divide & Conquer
- Dynamic Programming
- Comparison

Part 2: GRAPH'S ALGORITHMS

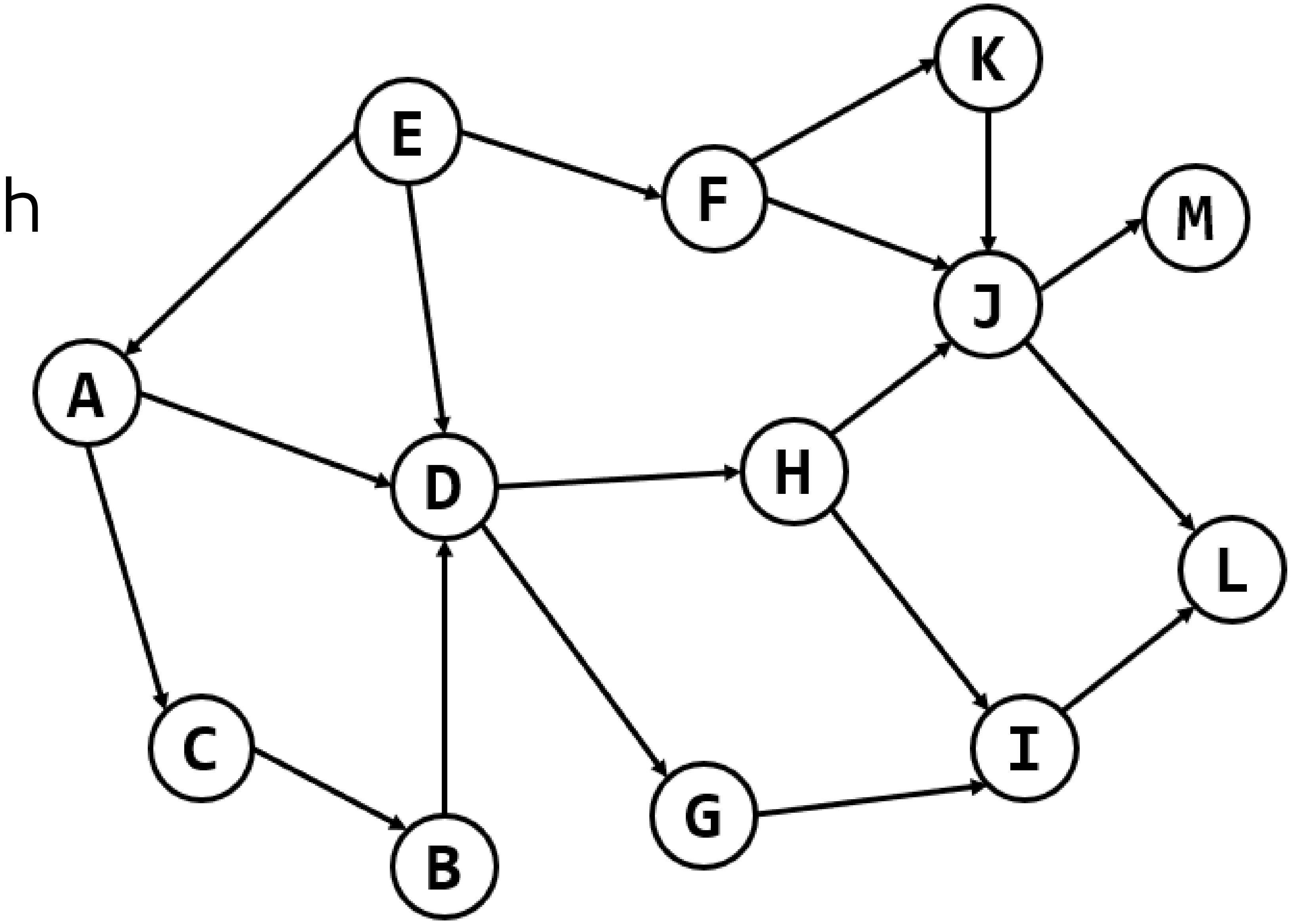
Problem Statement

Algorithm

Topological Sort — Problem Statement

Given a directed acyclic graph, find an disc of vertices so that for every directed edge u-v, vertex u comes before vertex v (finding a topological disc).

Input: a directed acyclic graph
Output: a topological disc
Constraint:
|V|≤3*10^4, |E|≤2*10^5



Problem Statement

Algorithm

- Backtracking
- Divide & Conquer
- Dynamic Programming
- Comparison

Topological Sort

Brute Force

- Let C be any order of the vertices.

→ The number of orders is **|V|!**.

→ To check if C is valid: **$O(|V|^2(|V|+|E|))$** (check if all pairs are valid).

❓ **How about other approaches?**

Backtracking

Problem Statement

Algorithm

- Brute Force
-
- Divide & Conquer
- Dynamic Programming
- Comparison

Idea: Iteratively insert vertices into a solution.

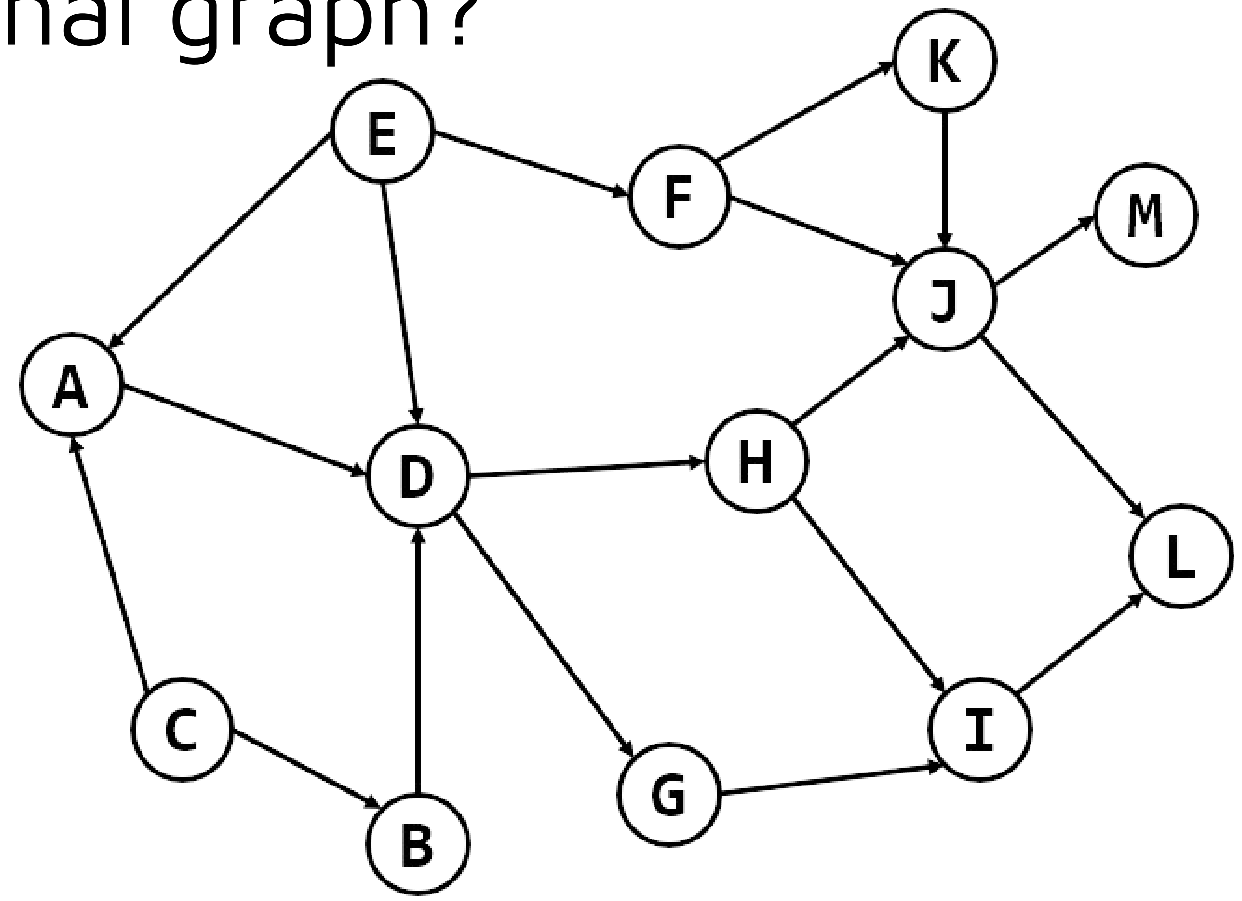
- If a newly added vertex can reach any vertices already in the solution, try another vertex.

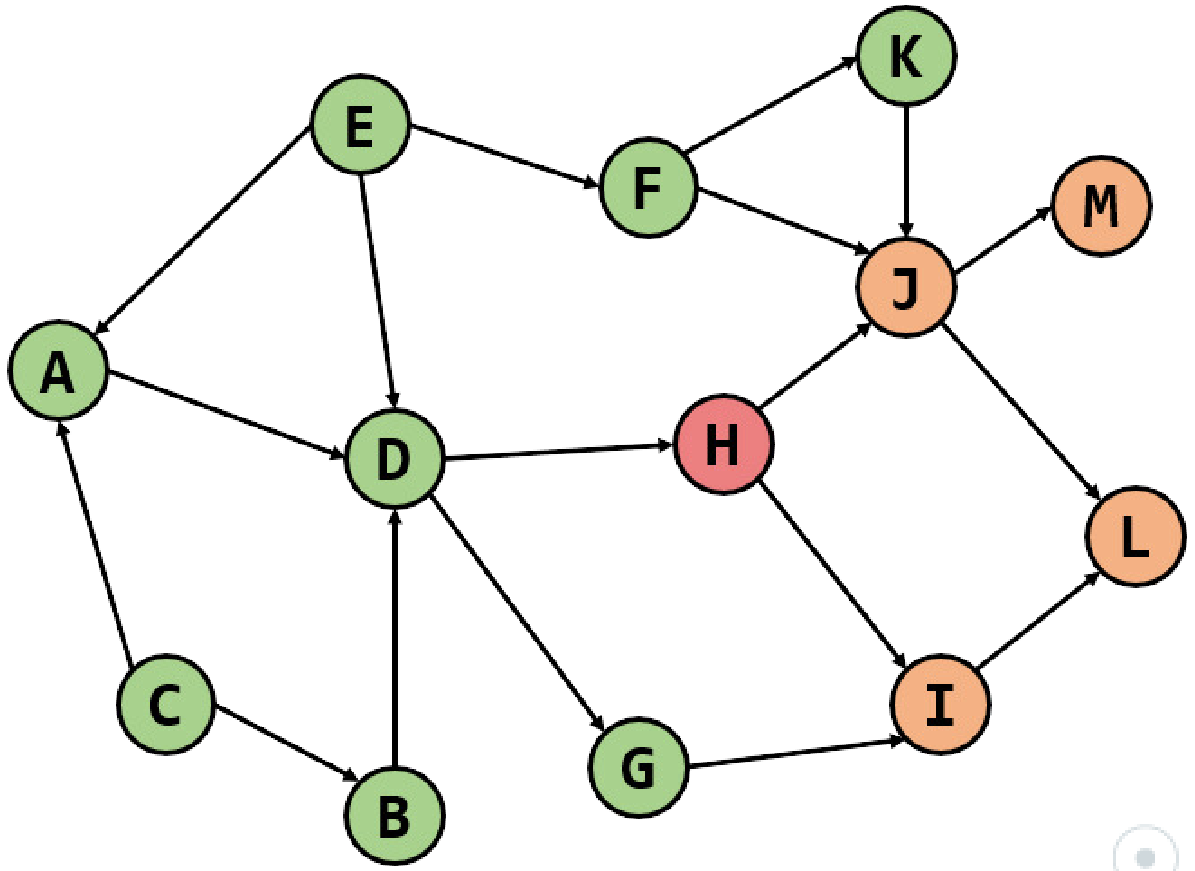
→ This will reduce some solution of brute force but have the same complexity.

Part 2: GRAPH'S ALGORITHMS

Topological Sort | Divide & Conquer

What happens if we remove a vertex, e.g H, from the original graph?





C	B	E	F	K	A	D	G	H	I	J	L	M

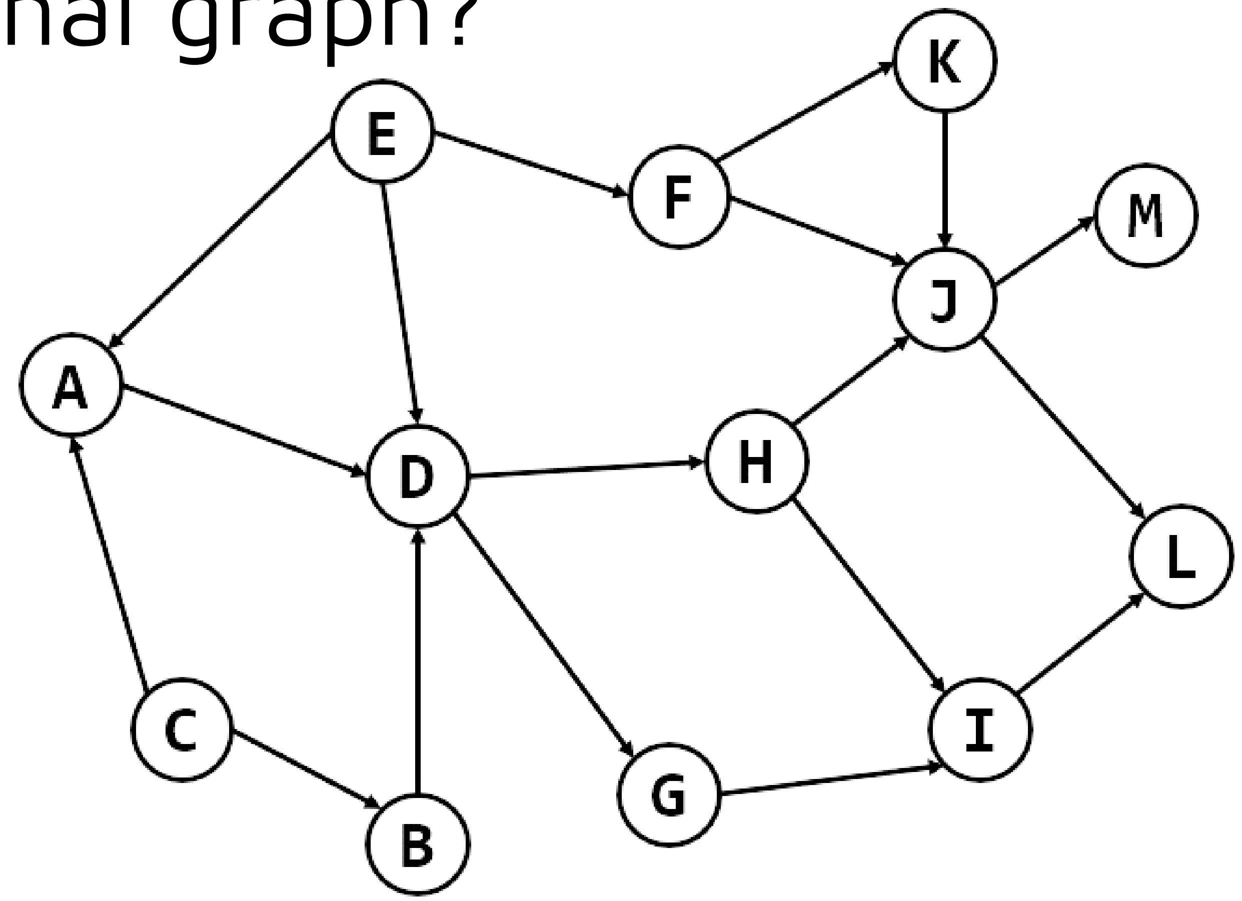
→ This will divide the graph into two independent parts: the orange part, which can be reached from H, and the green part, which cannot be reached from H.

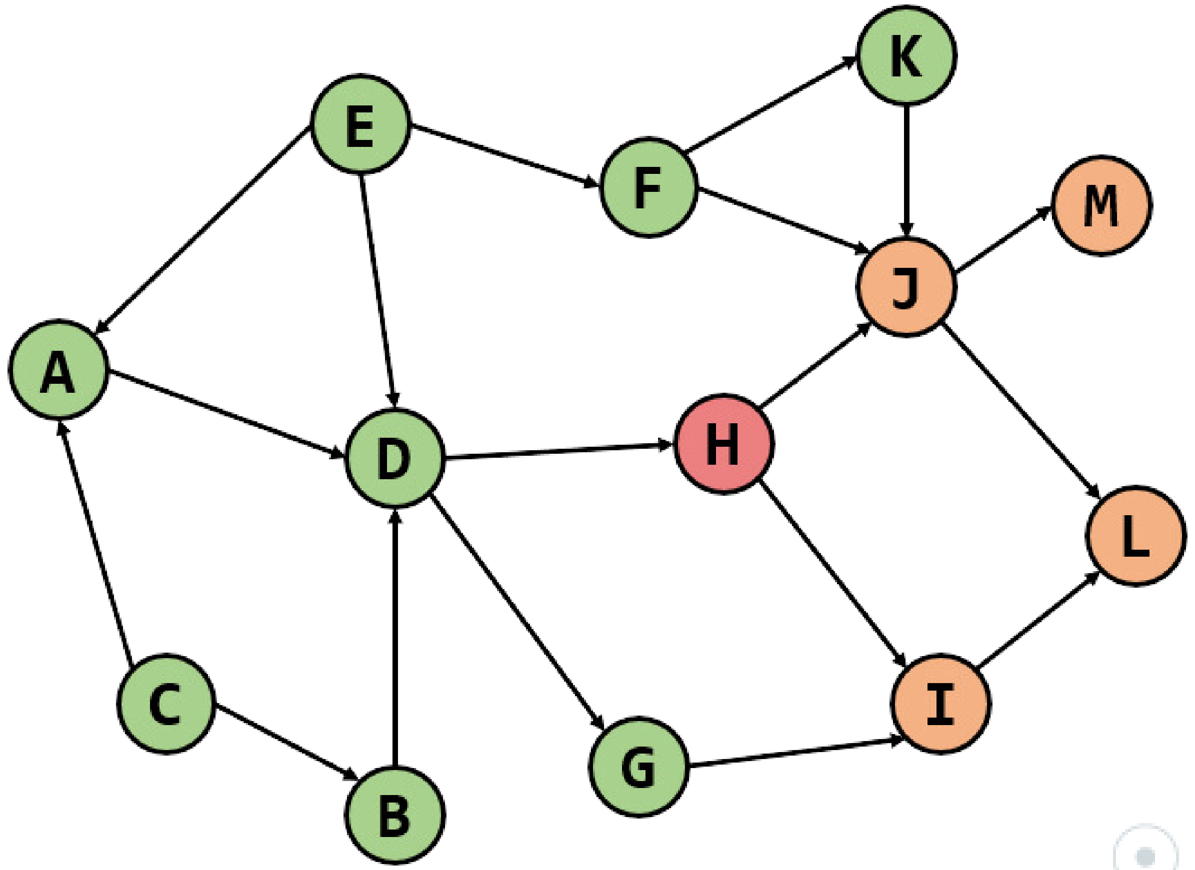
Part 2: GRAPH'S ALGORITHMS

Topological Sort

Divide & Conquer

What happens if we remove a vertex, e.g H, from the original graph?





C	B	E	F	K	A	D	G	H	I	J	L	M

We can build the whole solution in a single container.

→ First, we build the orange part, then insert H, then build the green part. That can be done by using DFS.

Part 2: GRAPH'S ALGORITHMS

Problem Statement

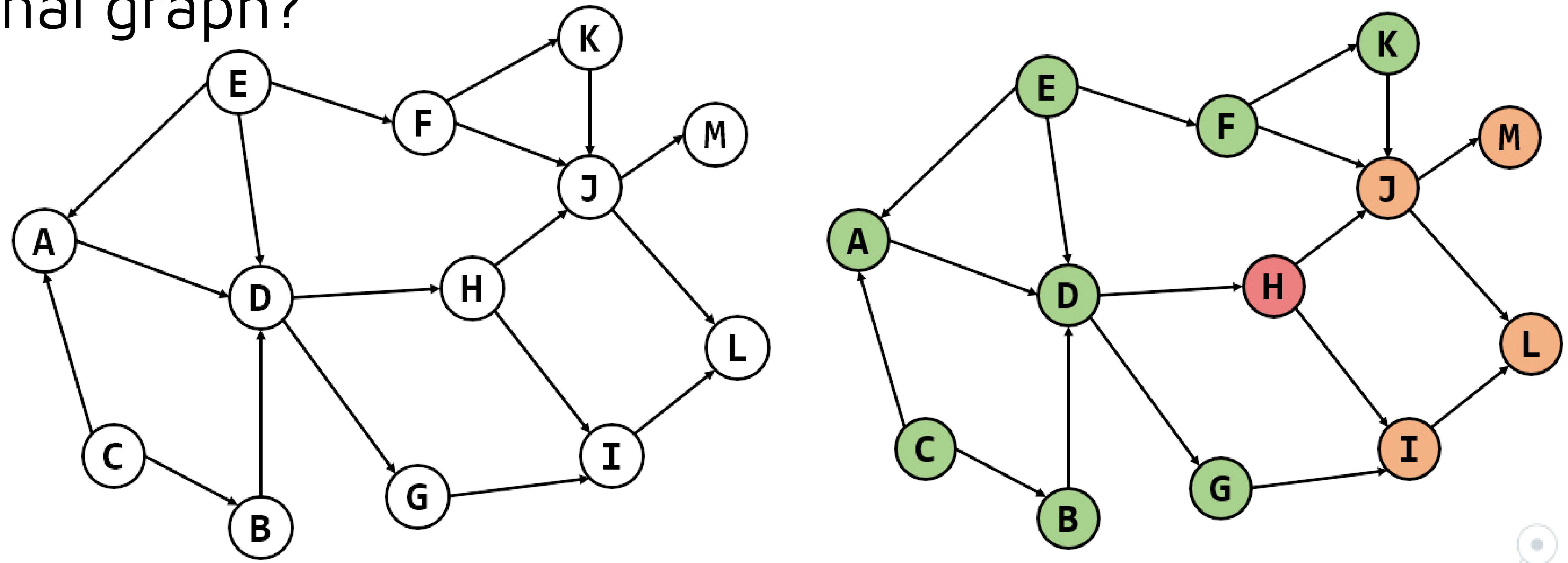
Algorithm

- Brute Force
- Backtracking
-
- Dynamic Programming
- Comparison

Topological Sort

Divide & Conquer

What happens if we remove a vertex, e.g H, from the original graph?



C	B	E	F	K	A	D	G	H	I	J	L	M

→ Time Complexity: **O(|V|+|E|)**

Part 2: GRAPH'S ALGORITHMS

Topological Sort

Divide & Conquer

Problem Statement

Algorithm

- Brute Force
- Backtracking
-
- Dynamic Programming
- Comparison

Pseudocode:

```
ts <- empty list
for each unvisited vertex u:
  topo_sort(u)
function topo_sort(root):
  visited[root] <- True
  for each ajacent vertex v of root:
    if not visited[v]:
      topo_sort(v)
  insert root at the beginning of ts
```

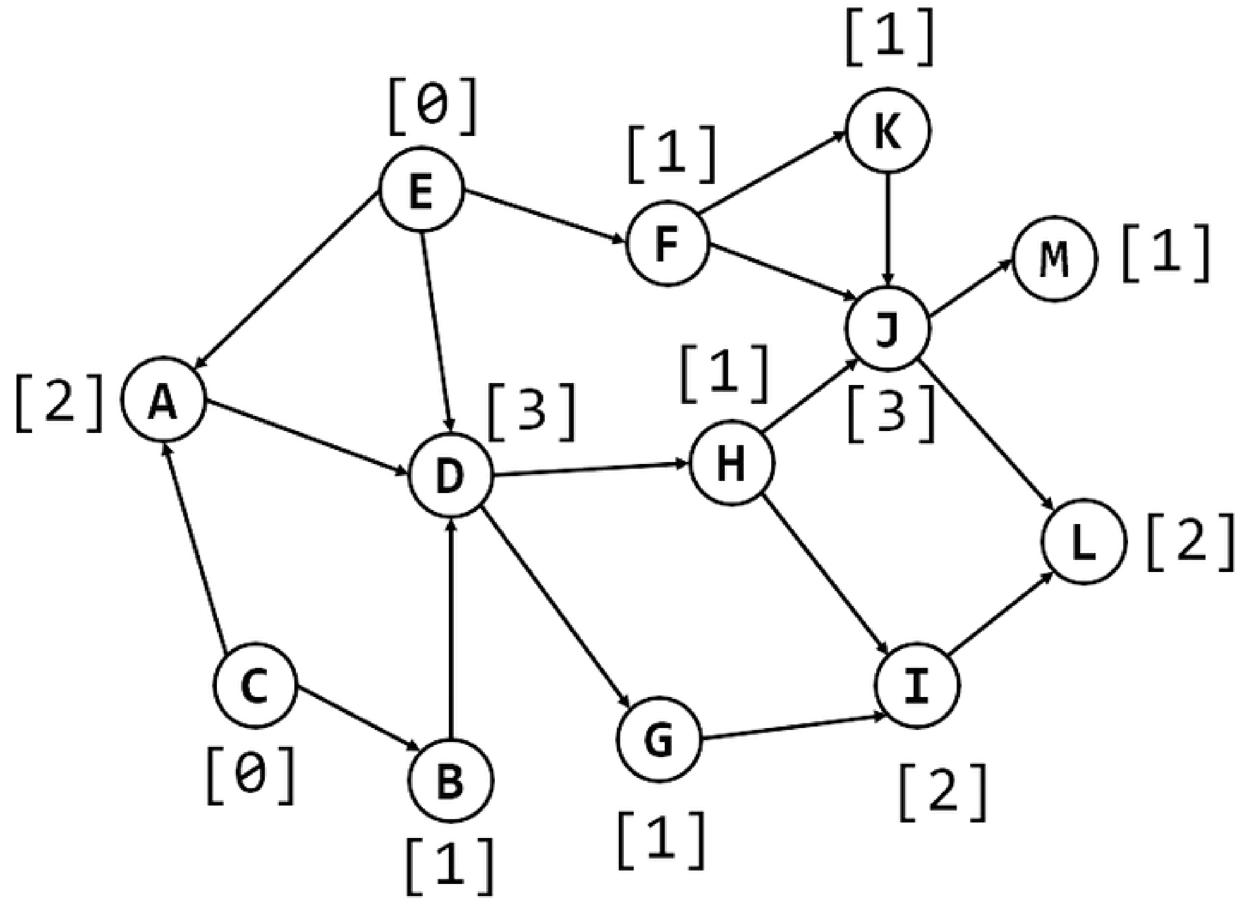
Topological Sort   Dynamic Programming

Problem Statement

Algorithm

- Brute Force
- Backtracking
- Divide & Conquer
-
- Comparison

- Noticing that a vertex can be put at the beginning of a topological order if and only if its in-degree is 0



Queue: E C

- In this case, both E and C can be put at the beginning of a topological order.

Part 2: GRAPH'S ALGORITHMS

Topological Sort

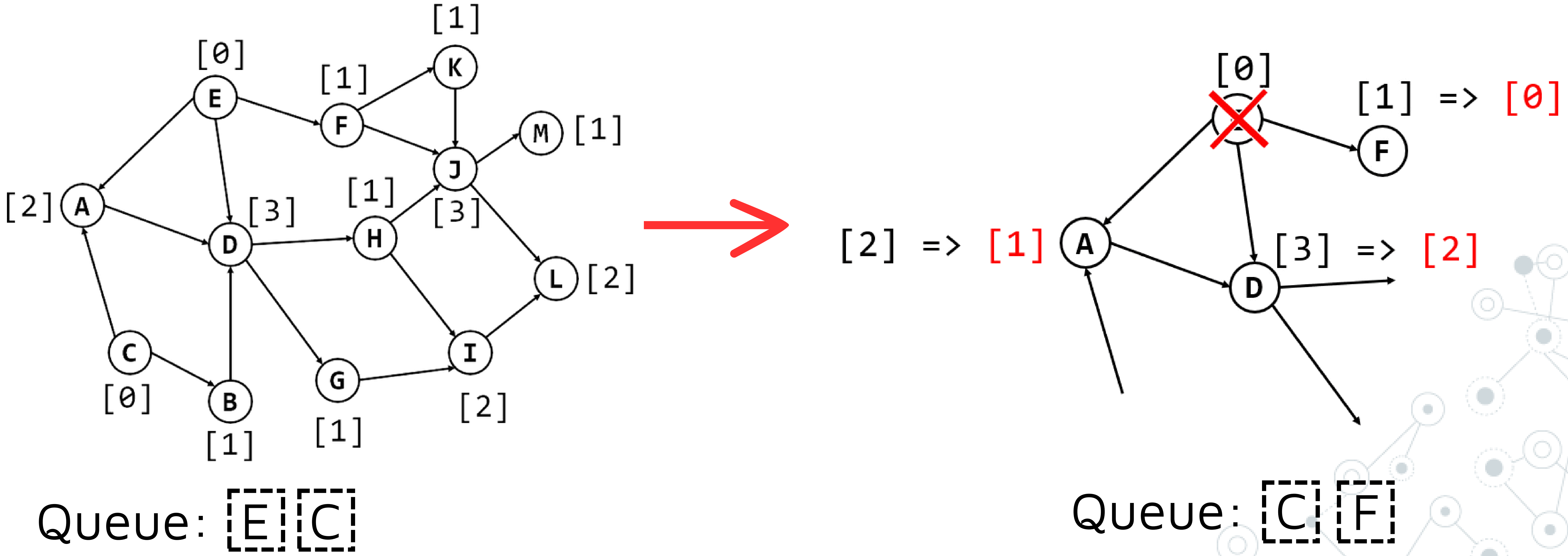
Dynamic Programming

Problem Statement

Algorithm

- Brute Force
- Backtracking
- Divide & Conquer
-
- Comparison

Now if we remove E from the graph (insert it to the topological order), we can efficiently calculate the in-degrees of its neighbors.



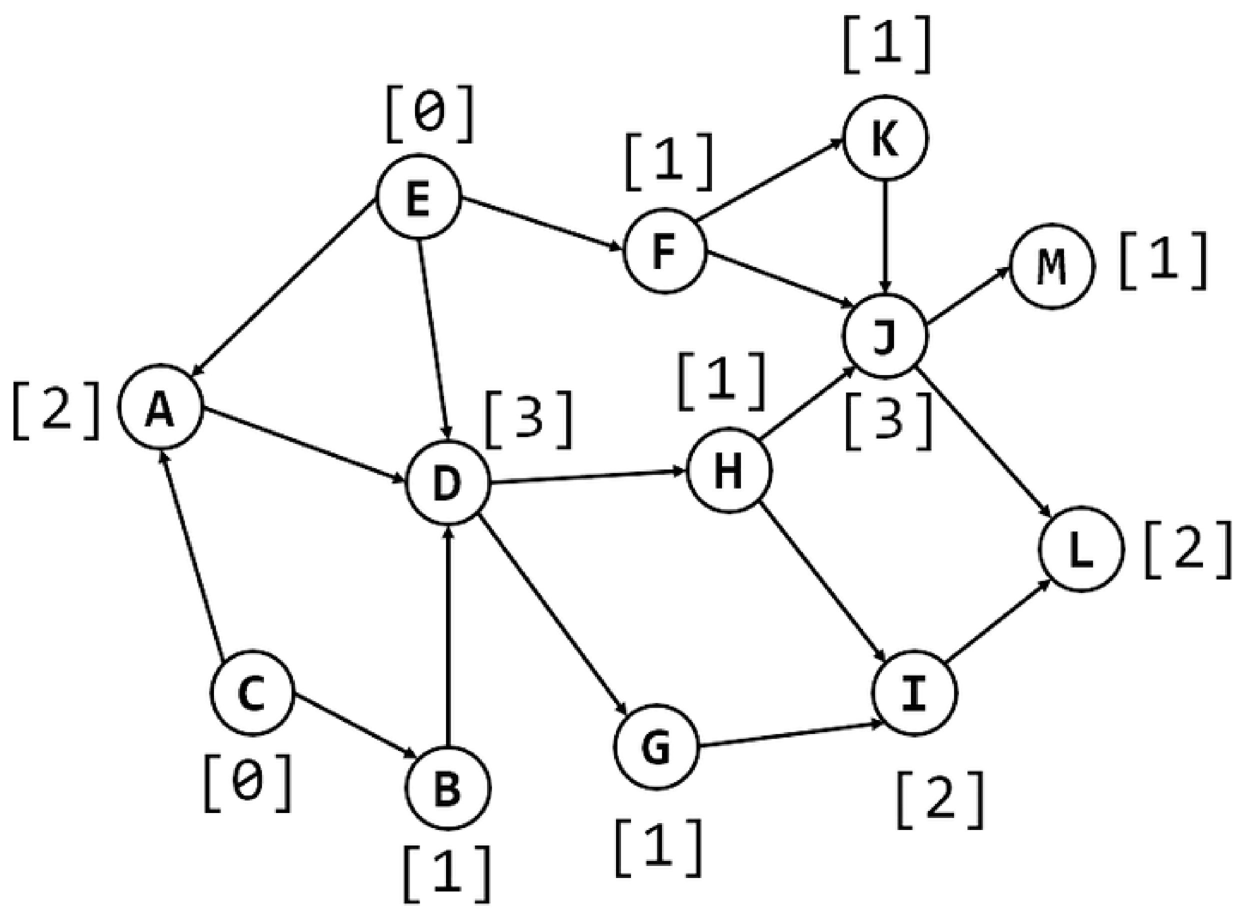
- F now have in degree of 0. Both C and F can be put right after E

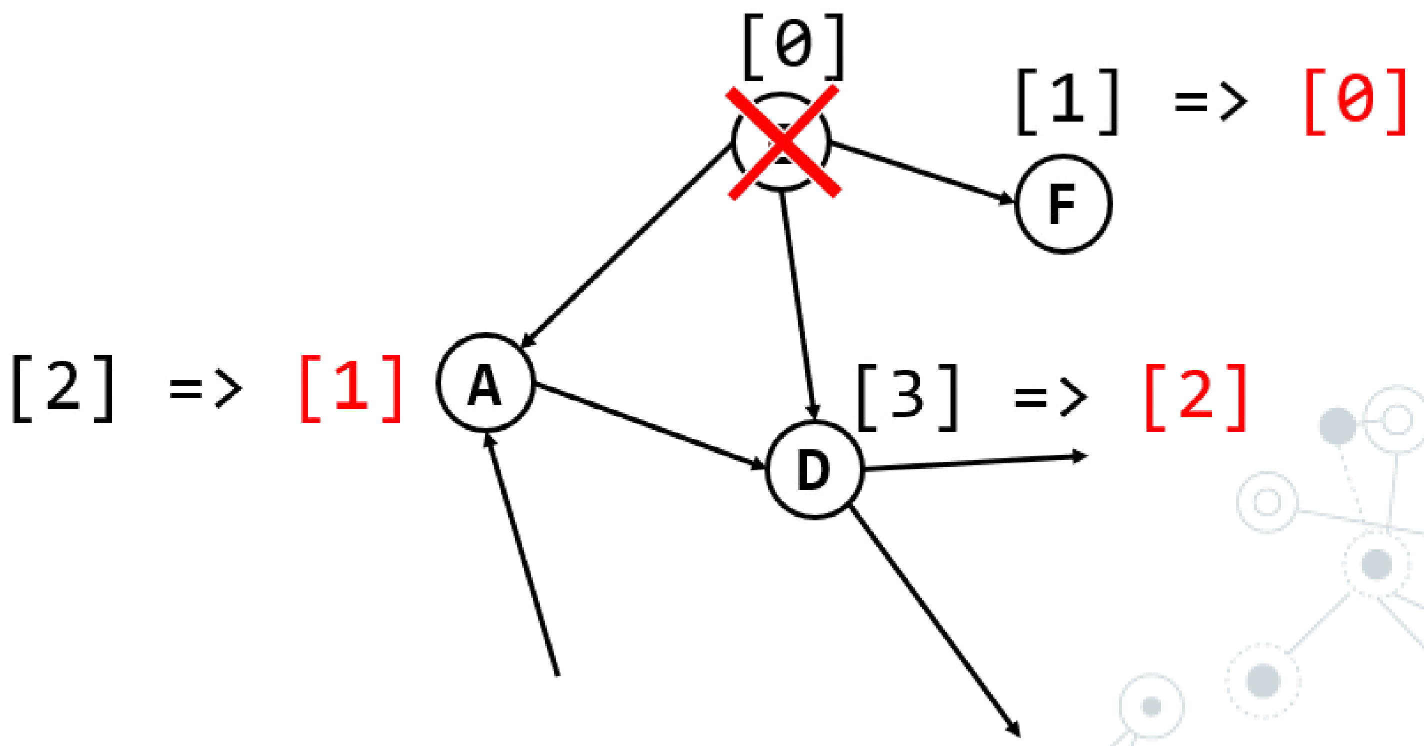
Part 2: GRAPH'S ALGORITHMS

Topological Sort

Dynamic Programming

Now if we remove E from the graph (insert it to the topological order), we can efficiently calculate the in-degrees of its neighbors.





Queue: E C

Queue: C F

The number of such calculations is the number of edges. Time Complexity: **O(|V|+|E|)**

Part 2: GRAPH'S ALGORITHMS

Topological Sort

Dynamic Programming

Problem Statement

Algorithm

- Brute Force
- Backtracking
- Divide & Conquer
-
- Comparison

Pseudocode:

```
in_deg <- a list of size |V| with all 0
ts <- empty list
zero_deg <- empty list
for each vertex u:
  for each child v of u:
    in_deg[v] <- in_deg[v]+1
  for each vertex u:
    if in_deg[u]=0:
      insert u to zero_deg
  while zero_deg is not empty:
    u <- any element of zero_deg
    insert u to ts
    for each child v of u:
      in_deg[v]←in_deg[v]-1
      if in_deg[v]=0:
        insert v to zero_deg
```

Problem Statement

Algorithm

- Brute Force
- Backtracking
- Divide & Conquer
- Dynamic Programming

Topological Sort

Comparison

DAC	DP
• Run faster for large graphs • May fail to detect cycles without further improvement	• Run slow for large graphs • Can always to detect cycles without further improvement

Part 2: GRAPH'S ALGORITHMS

Tarjan’s Algorithm

Problem Statement

Algorithm

- Problem Analysis
- Finding Bridges
- Finding Articulation points

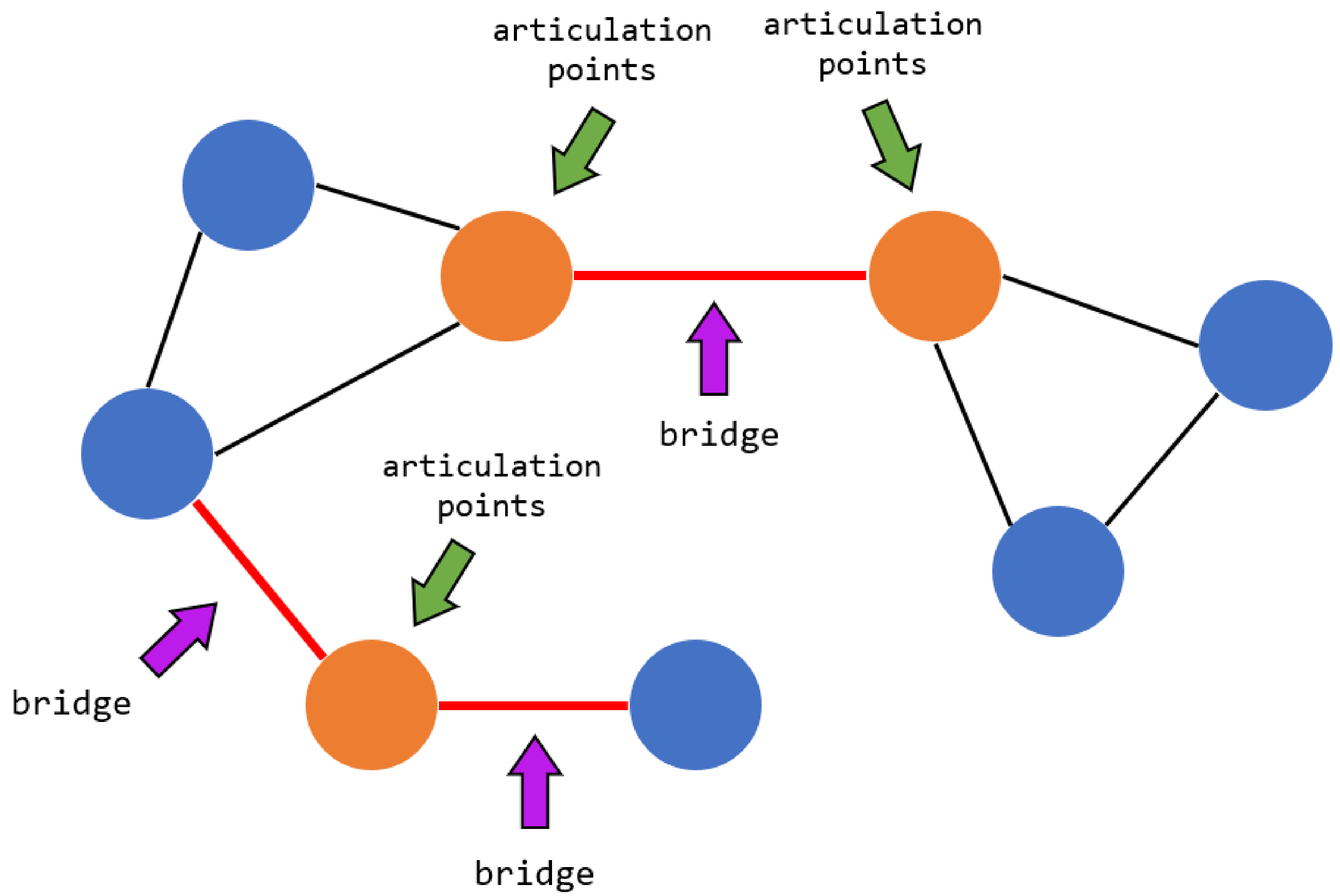
Tarjan's Algorithm Problem Statement

Given an undirected graph, find all bridges and articulation points in the graph.

Input: an undirected graph

Output: a list of bridges, a list of articulation points

Constraint: $|V| \le 3*10^4$, $|E| \le 2*10^5$



Tarjan's Algorithm

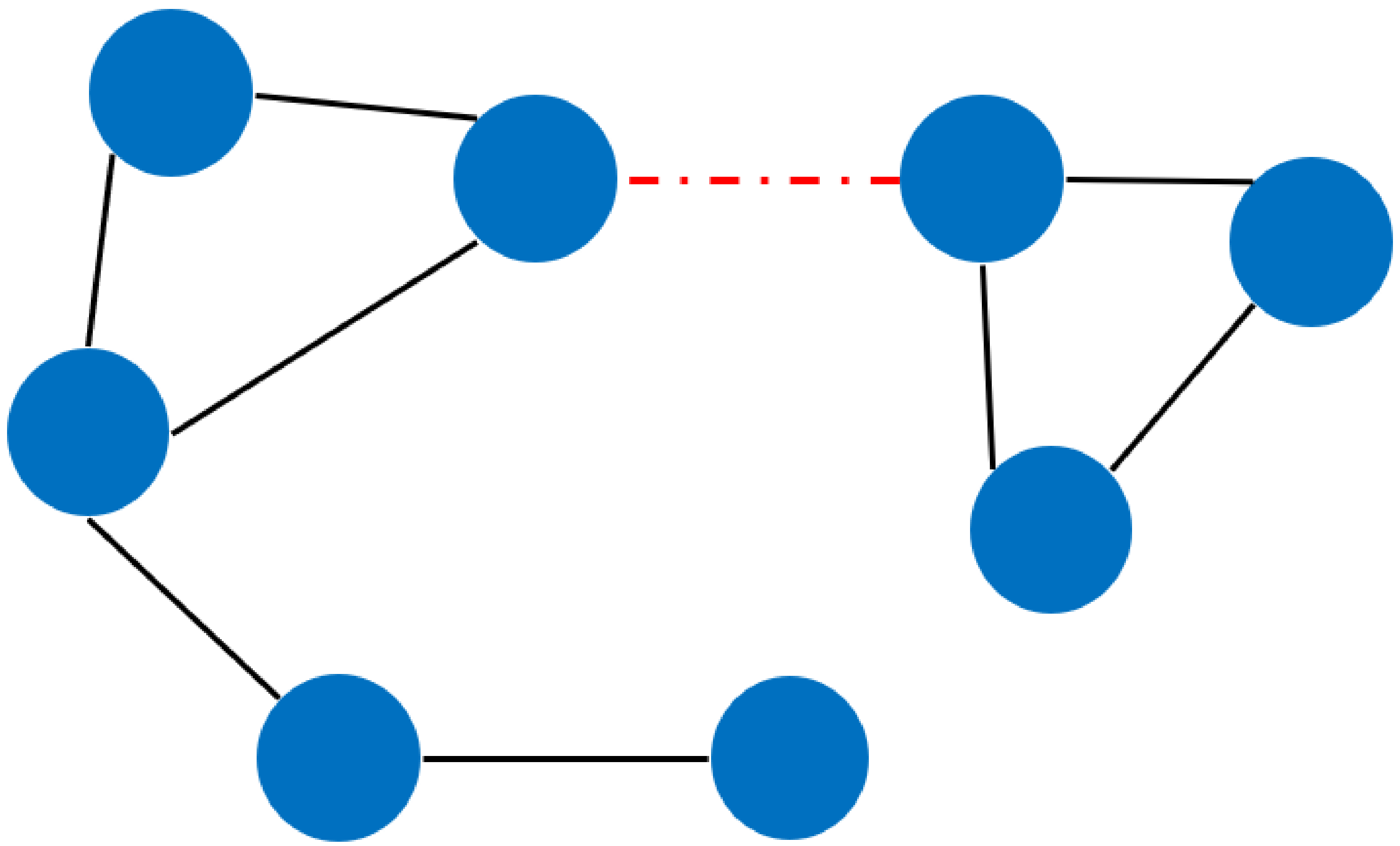
Problem Analysis

Problem Statement

Algorithm

- Finding Bridges
- Finding Articulation points

Subproblem: given an edge in the graph, determine if it is a bridge or not



Counting number of connected components before and after removing an edge → **O(|V|+|E|)**

All edges → **O(|E|(|V|+|E|))**

Problem Statement

Algorithm

Finding Bridges

Finding Articulation points

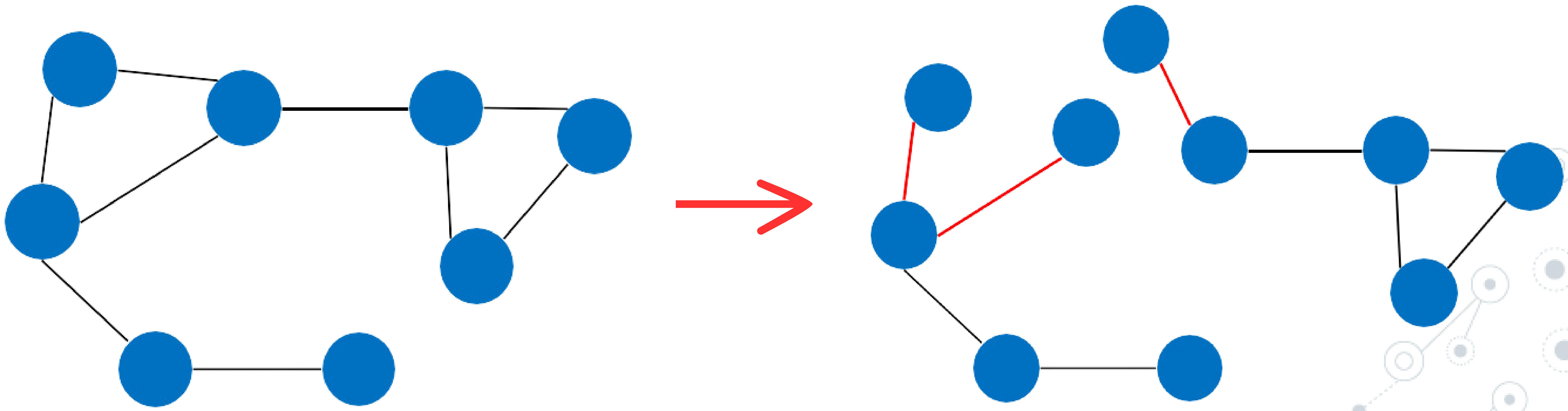
Tarjan's Algorithm

Problem Analysis

Better approach?

- Optimization problem: Greedy Approach ✗
- Independent subproblems: Backtracking

⇒ Non - bridges may becomes bridges.



→ Backtracking ✗

- Optimal Structures?

Part 2: GRAPH'S ALGORITHMS

Problem Statement

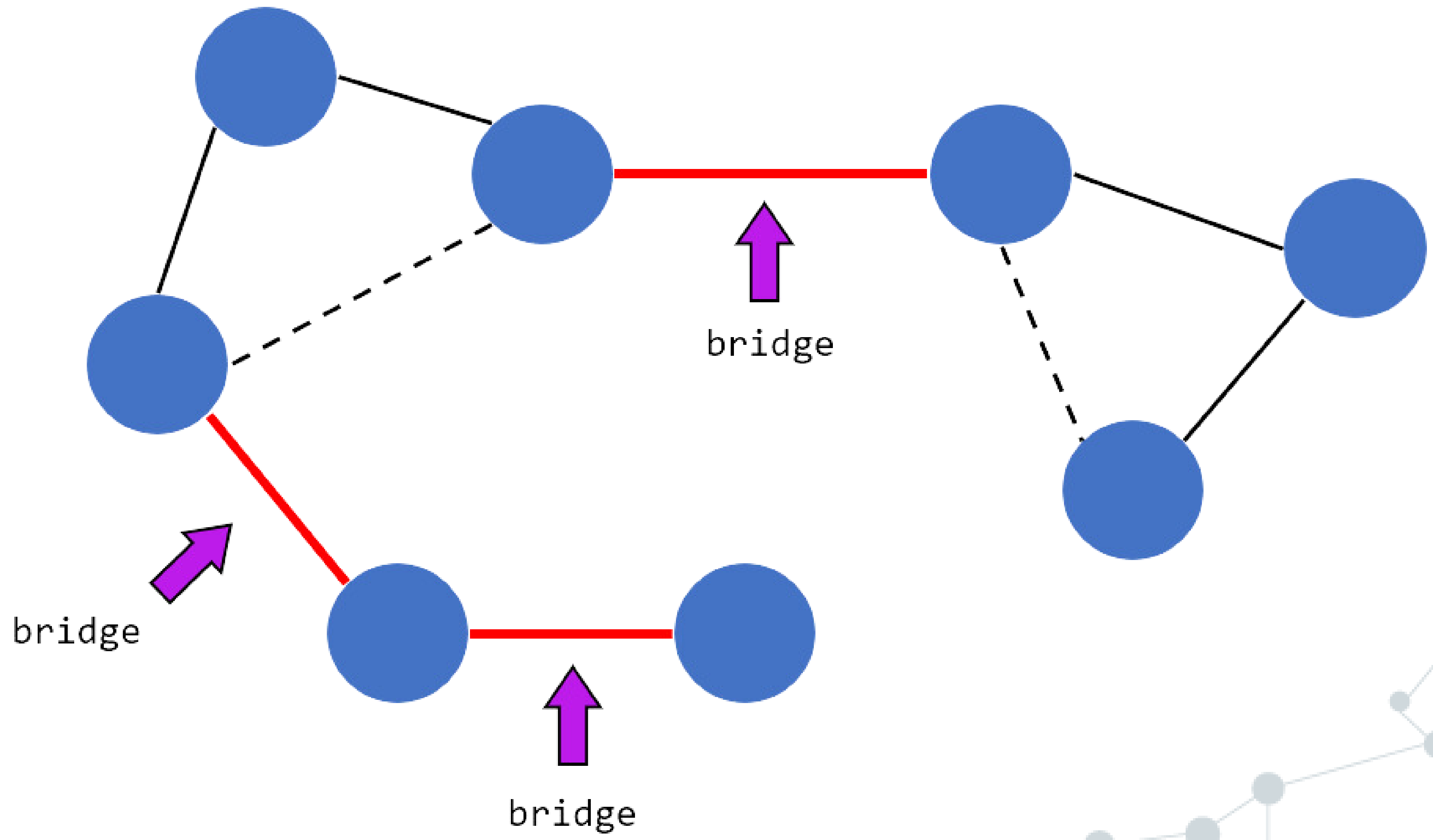
Algorithm

- Problem Analysis
-
- Finding Articulation points

Tarjan's Algorithm

Finding Bridges

On the DFS tree, vertex u is the parent of vertex v:
⇒ edge (u, v) is **a bridge** if and only if v **doesn't have any other links** to u or any ancestors of u.



Problem Statement

Algorithm

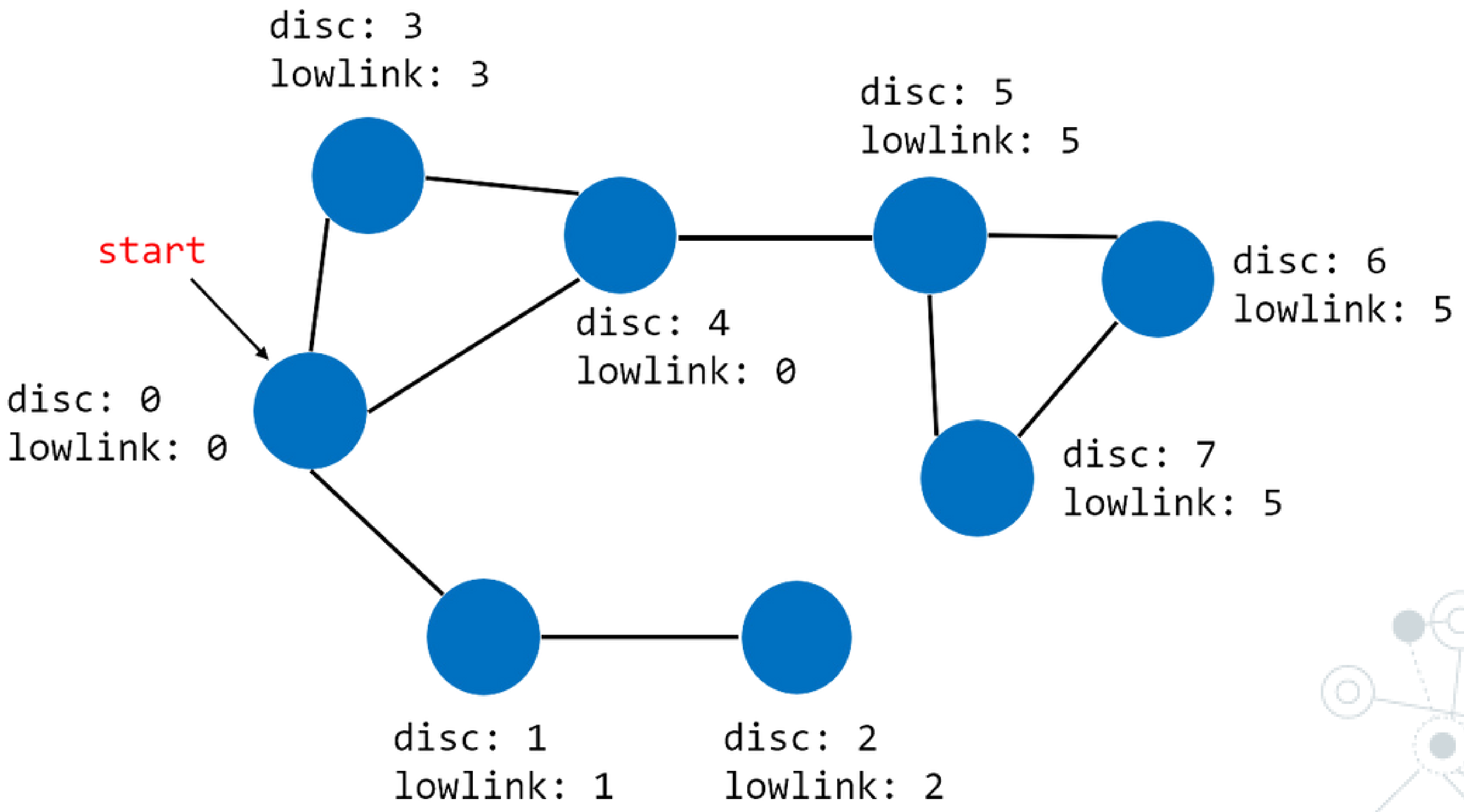
- Problem Analysis
- Finding Articulation points

Tarjan's Algorithm

Finding Bridges

Giving each vertex an id showing the time we have discovered it

⇒ we can find the earliest vertex that a certain vertex can link to using the following formulas:



lowlink[u] = disc[u]

for each neighbor v of u:

    lowlink[u] = min(disc[v], lowlink[u]) if v is visited and not parent

    lowlink[u] = min(lowlink[v], lowlink[v]) if v is not visited

→ Dynamic programming

Time complexity: **O(|V|+|E|)**

Tarjan’s Algorithm

Finding Bridges

Problem Statement

Algorithm

- Problem Analysis
-
- Finding Articulation points

Pseudocode:

```
function bridges():
  visited <- a list of size |V| with all 0
  disc <- 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[u]:
      bridge_helper(u)
  return bridge_count
```

Part 2: GRAPH'S ALGORITHMS

Tarjan's Algorithm

Finding Bridges

Problem Statement

Algorithm

- Problem Analysis
-
- Finding Articulation points

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

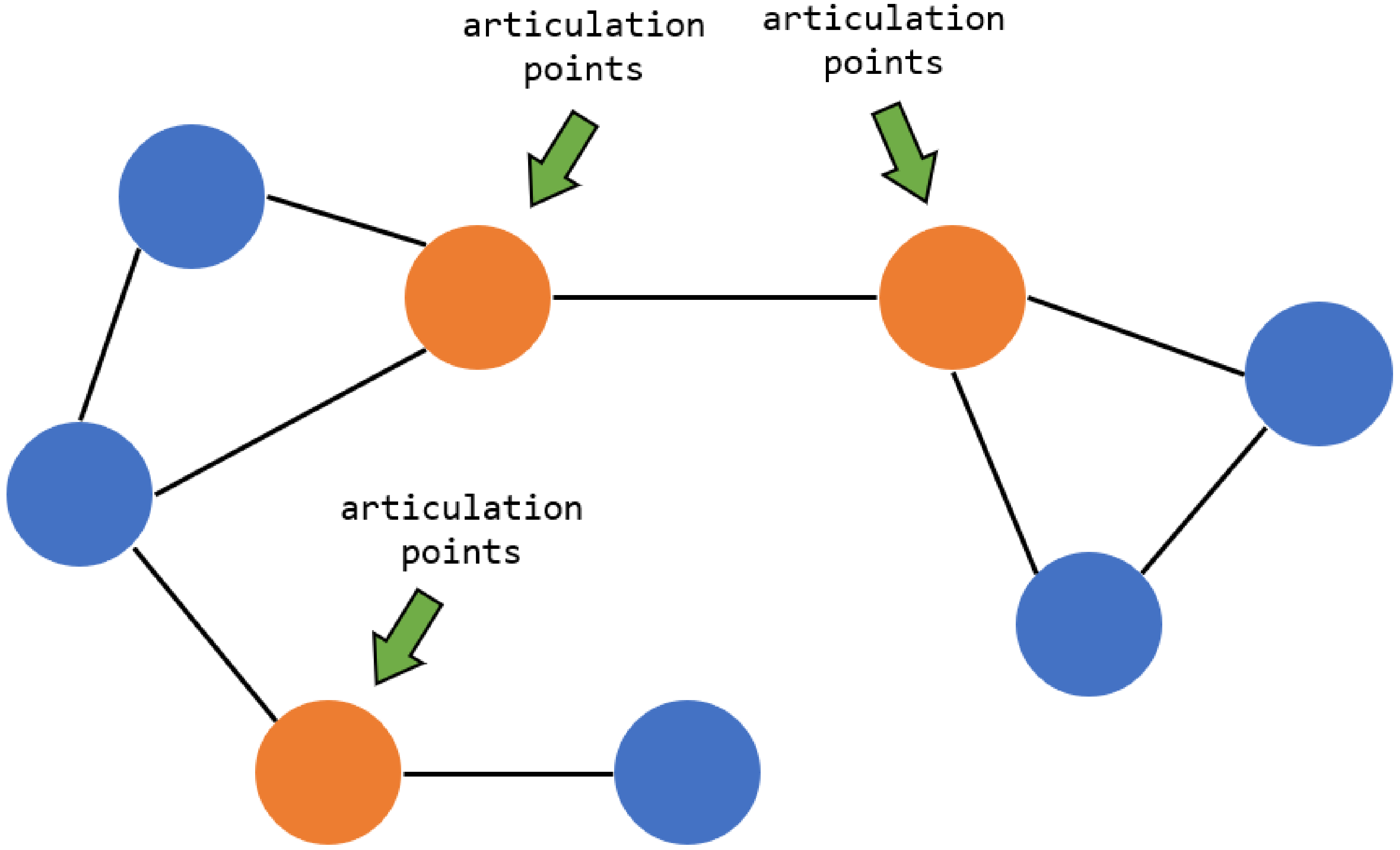
Tarjan's Algorithm

Finding Articulation points

Same idea as finding articulation points.

A vertex u is an articulation point if and only if:

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



Tarjan's Algorithm Finding Articulation points

Pseudocode:

```
function ap():
  visited <- a list of size |V| with all Ø
  disc <- 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 u in the graph:
    if not visited[u]:
      branches <- Ø
      root <- u
      ap_helper(u)
  return ap_count
```

Part 2: GRAPH'S ALGORITHMS

Problem Statement

Algorithm

- Problem Analysis
- Finding Bridges

Tarjan's Algorithm Finding Articulation points

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

Part 2: GRAPH'S ALGORITHMS

Ford - Fulkerson's Algorithm

Problem Statement

Algorithm

- Problem Analysis
- Greedy Approach

Ford - Fulkerson's Algorithm

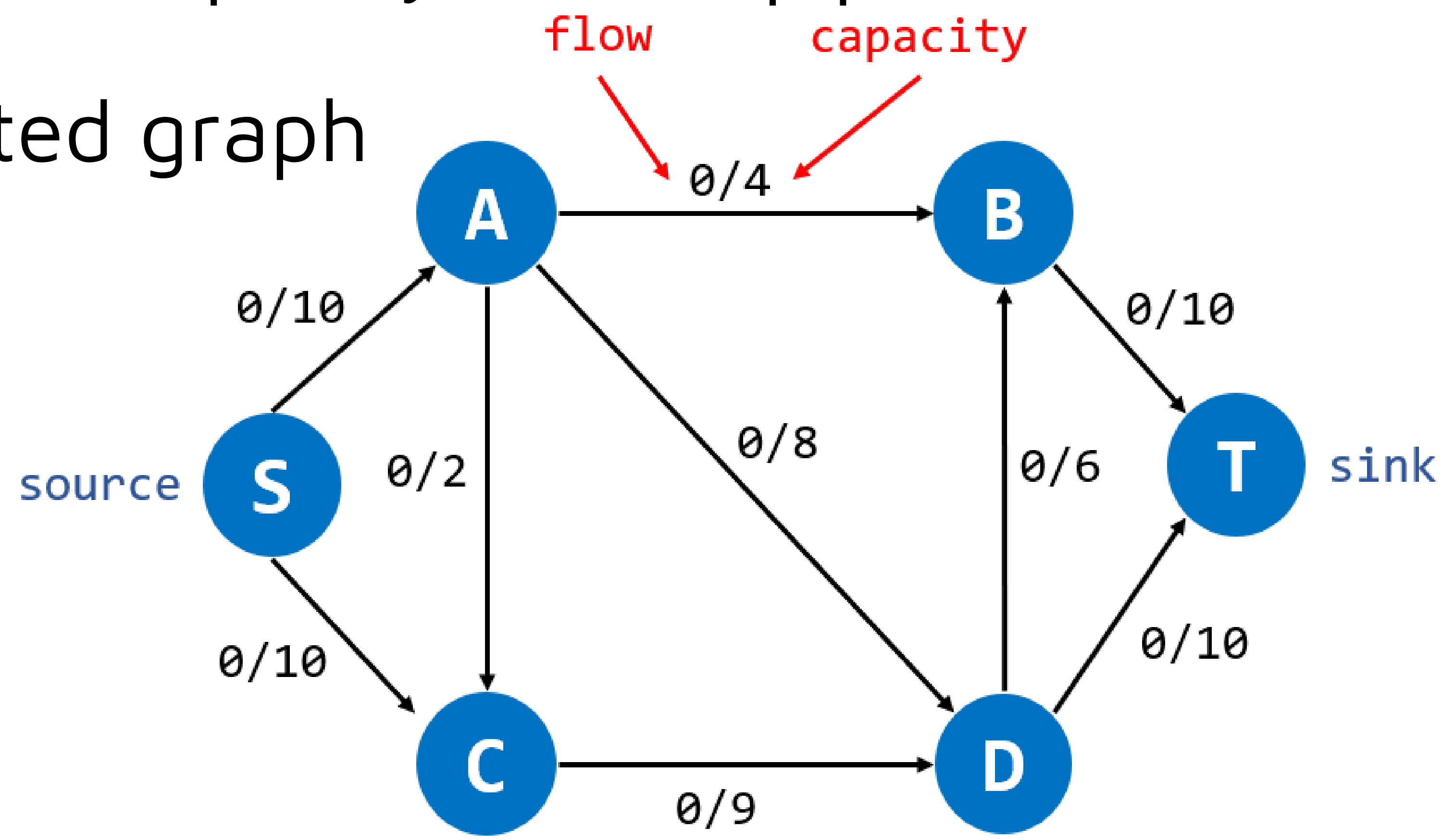
Problem Analysis

Problem Statement
Algorithm
Greedy Approach

Given a pipe network with a source, a sink and capacities of each pipe. Find the maximum amount of water can be sent from source to sink so that:

- total water flowing from source = total water flowing to sink
- total water flowing to v = total water flowing from v (v is not source or sink)
- water flowing through a pipe ≤ capacity of that pipe

Input: a positive-weighted directed graph
Output: max flow value (mfv)
Constraint: mfv ≤ 500, |V|≤500



Brute force approach

Let C be the combination of all possible flow value for each edges.

- The number of combinations is the product of all capacities.

To check if C is valid ⇒ **O(|V|+|E|)**.

The total complexity: **O((|V|+|E|)*product of capacities)**

Backtracking approach

- We may not be able to check for validity after each insertion

Problem Statement

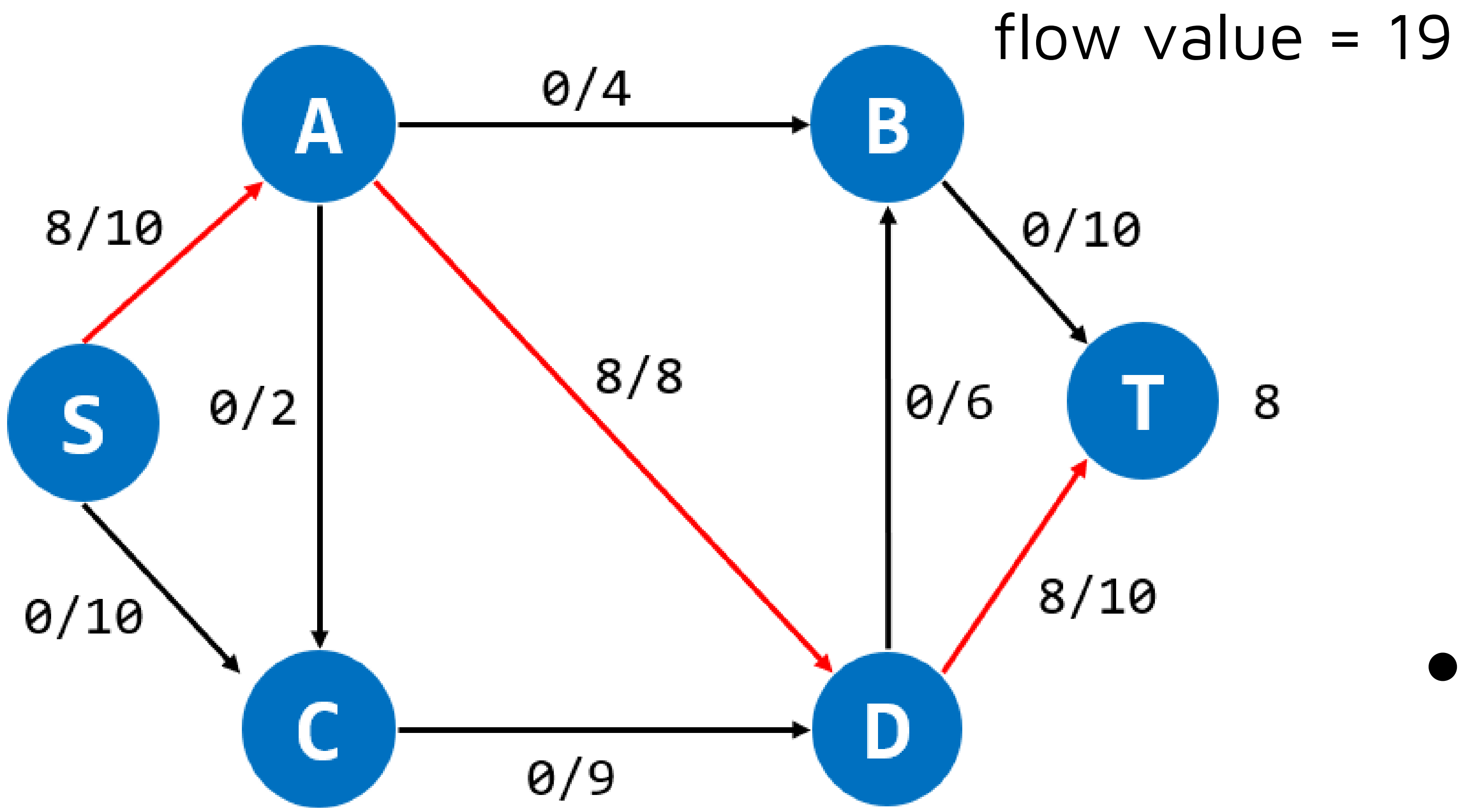
Algorithm

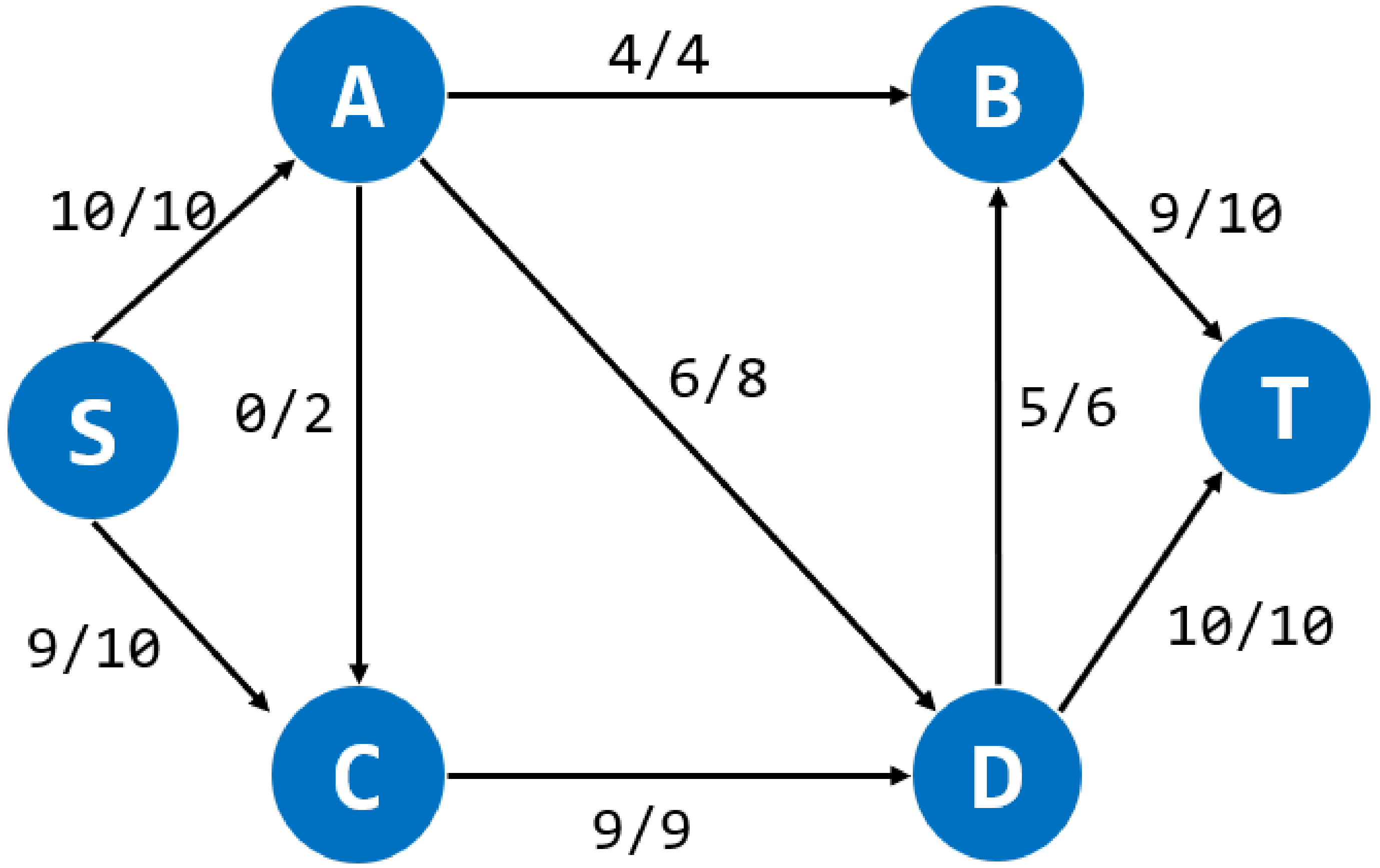
- Problem Analysis

Ford - Fulkerson's Algorithm

Greedy Approach

- We can continuingly add water into the network until we cannot do so anymore.





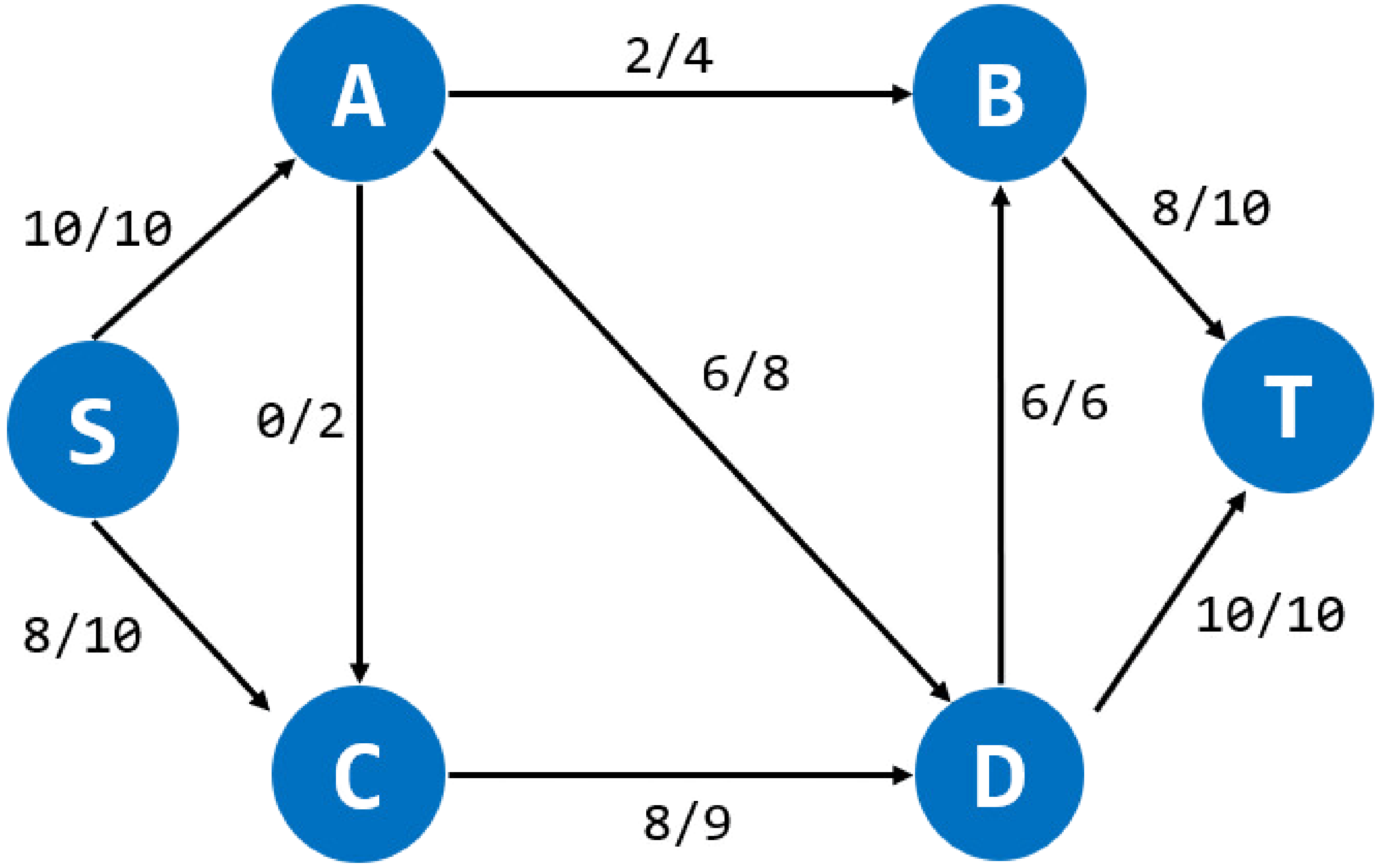
- It is proved that at that point, a maximum flow is found.

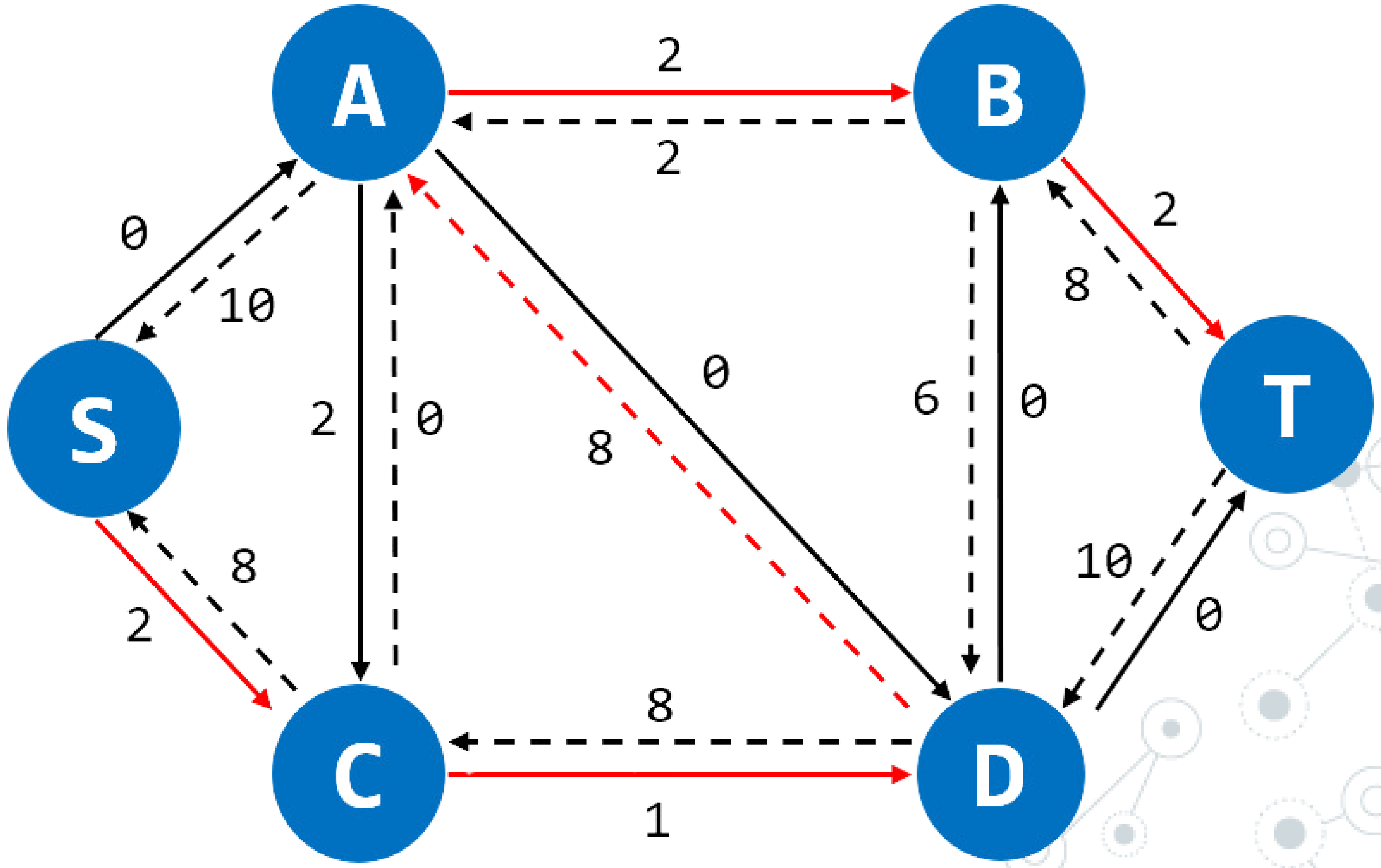
→ At each iteration, we just need to find a way from source to sink and add the amount of water equal to the bottleneck.
However, there is a problem...

Ford - Fulkerson's Algorithm

Greedy Approach

The current flow has "blocked" any water to be sent. How can we adjust it?





By introducing reverse edges (which tell us the number of subtances currently in the original edges), we can let some water to flow into another pipe.

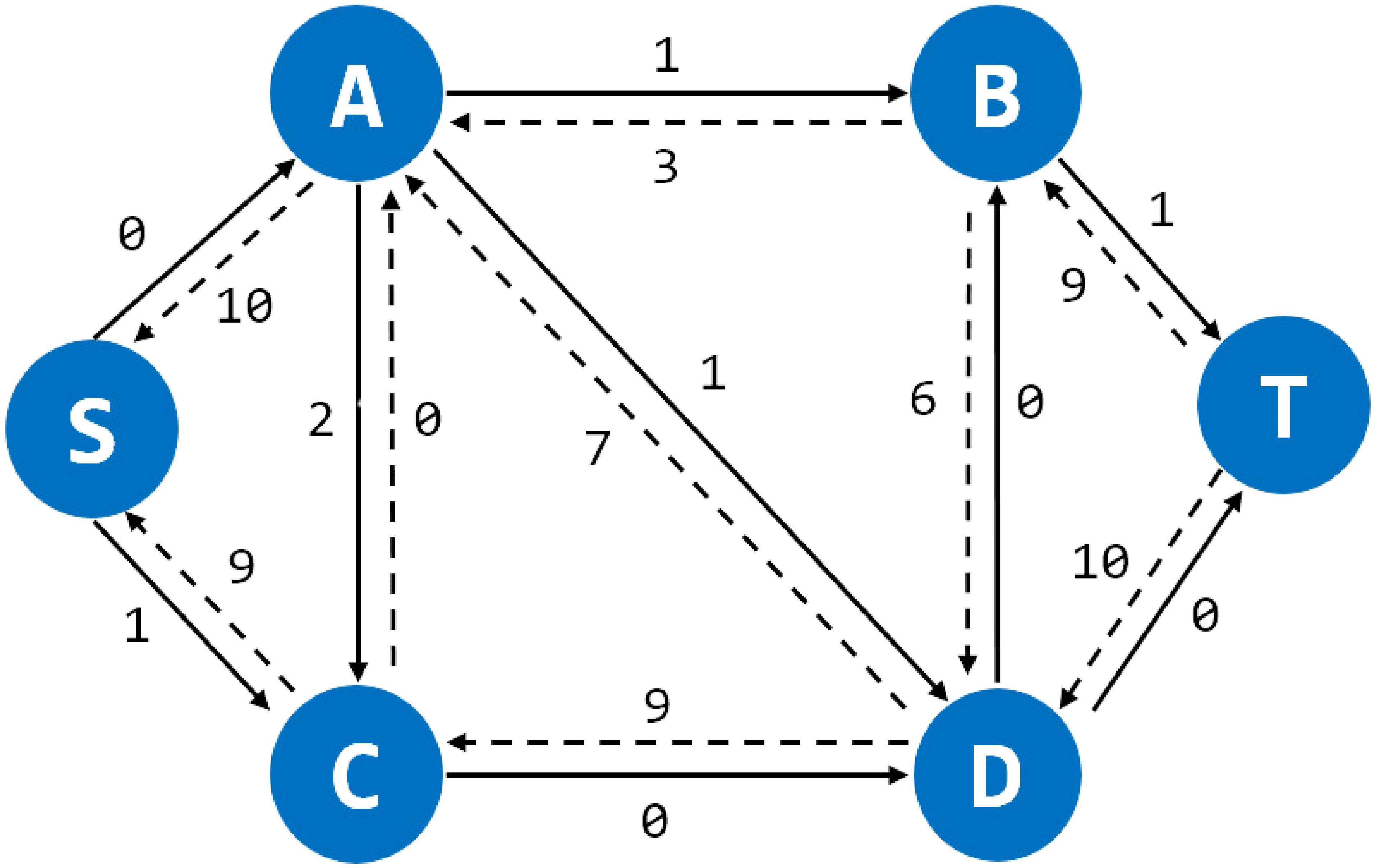
Part 2: GRAPH'S ALGORITHMS

Ford - Fulkerson's Algorithm Greedy Approach

Problem Statement

Algorithm

- Problem Analysis



In the worst case: we have to traverse the whole network and can only add 1 more unit of water to the flow.

→ Time complexity: **O(mfv*(|V|+|E|)**

Part 2: GRAPH'S ALGORITHMS

Ford - Fulkerson’s Algorithm Greedy Approach

Problem Statement

Algorithm

- Problem Analysis

Pseudocode:

```
for each edge (u, v) of the graph:
  add edge (v, u) of capacity 0 to the graph
function ford_fulkerson():
  flow_value <- 0
  visited <- a list of size |V| with all 0
  parent <- a list of size |V| with all -1
  bottleneck <- infinity
  while find_aug_path(source): # find a way from source to sink
    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 0
    parent <- a list of size |V| with all -1
  return max_flow
```

THANK YOU FOR LISTENING!