香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Graph Algorithms

Overview

- Basic Definitions and Applications
- Graph Connectivity and Graph Traversal
- Connectivity in Directed Graphs
- DAGs & Topological Ordering

Basic Definitions and Applications

Definition

- A graph: a group of vertices and edges that are used to connect these vertices

- Definition: A graph G can be defined as an ordered set $G(V, E)$
  - V represents the set of vertices/nodes
  - E represents the set of edges which are used to connect V

Applications: Social Network

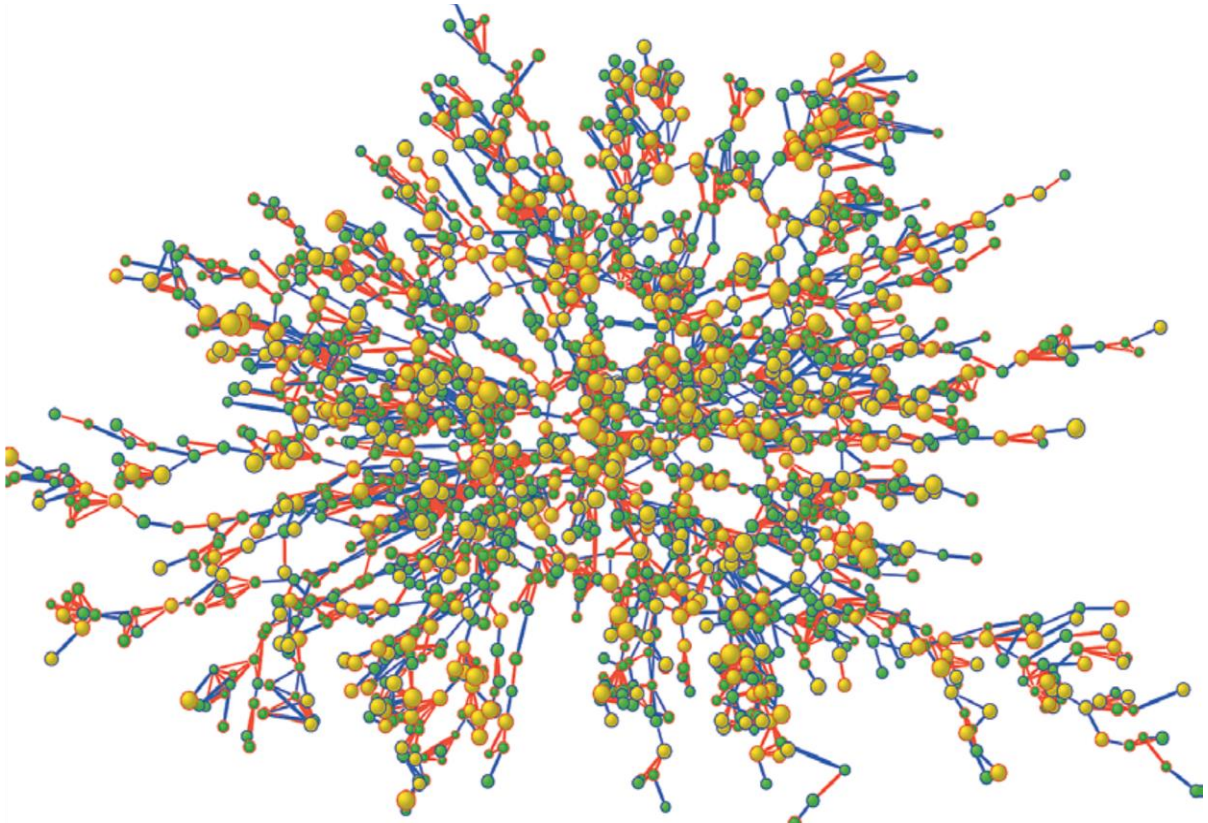
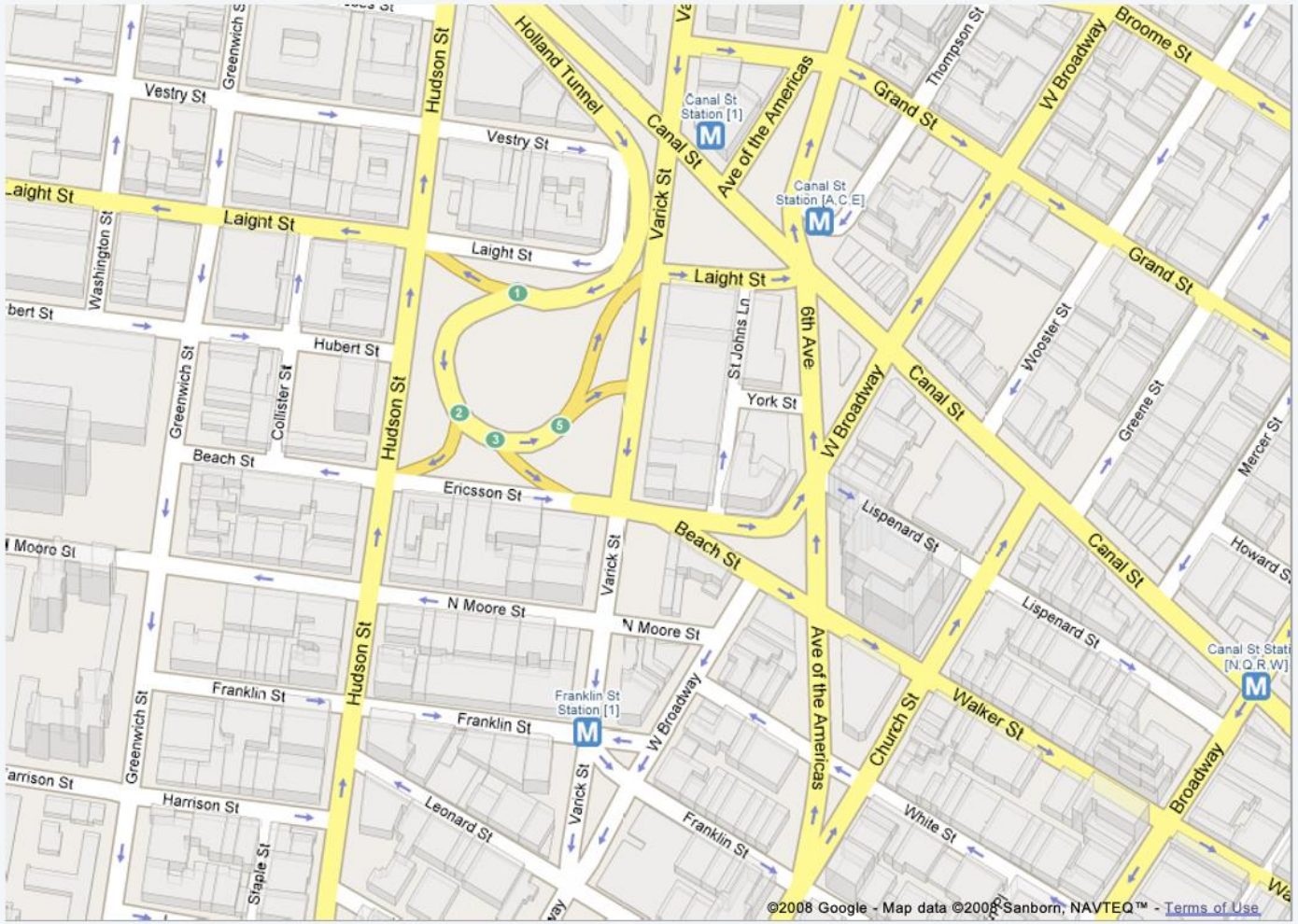


Figure 1. Largest Connected Subcomponent of the Social Network in the Framingham Heart Study in the Year 2000.

Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person's body-mass index. The interior color of the circles indicates the person's obesity status: yellow denotes an obese person (body-mass index, ≥30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.

Road Network

Node = intersection; edge = street.



More Applications

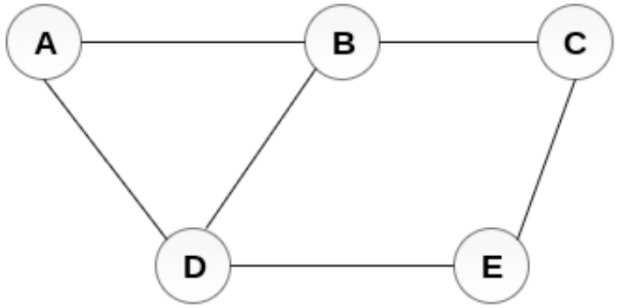
graph	node	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

Sequential Representation

- Use adjacency matrix to store the mapping represented by vertices and edges
- In adjacency matrix, the rows and columns are represented by the graph vertices
- For a graph having n vertices, the adjacency matrix will have a dimension $n \times n$

Sequential Representation

- Undirected: an entry A_{ij} in the adjacency matrix will be 1 if there exists an edge between v_i and v_j.



Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

Linked Representation

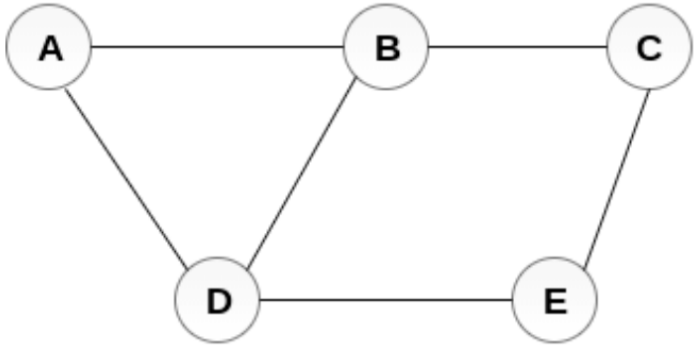
- An adjacency list is used to store the Graph into the computer's memory

- An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node

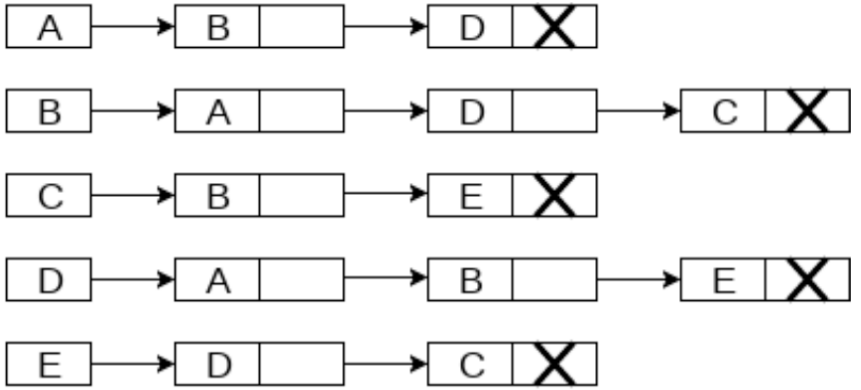
- If all the adjacent nodes are traversed, then store the NULL in the pointer field of last node of the list

Linked Representation

- Undirected: The sum of the lengths of adjacency lists is equal to the twice of the number of edges



Undirected Graph



Adjacency List

Terminology

- Path: a sequence of edges connecting initial node v_0 to terminal node v_n
- Closed Path: A path where the initial node is same as terminal node, i.e., $v_0 = v_n$
- Simple Path: all the nodes of the path are distinct, with the exception $v_0 = v_n$
- Closed Simple Path: a simple path with $v_0 = v_n$
- Cycle: a path which has no repeated edges or vertices except the first and last vertices
- Adjacent Nodes: two nodes u and v are connected via an edge e
  - the nodes u and v are also called as neighbors
- Degree of a Node: the number of edges that are connected with the node
  - A node with degree 0 is called as isolated node

Terminology

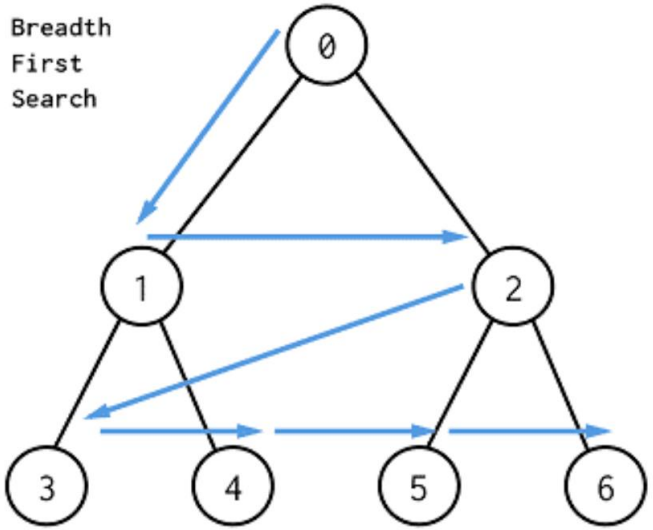
- Connected Graph: a graph in which a path exists between every two vertices u and v in V
  - There are no isolated nodes in connected graph
- Complete Graph: a graph in which there is an edge between each pair of vertices
  - A complete graph contain $n(n-1)/2$ edges where n is the number of nodes in the graph
- Weighted Graph: each edge is assigned with some data such as length or weight
  - The weight of an edge e, $w(e)$, must be positive indicating the cost of traversing the edge
- Digraph: each edge of the graph is associated with some direction
  - The traversing can be done only in the specified direction

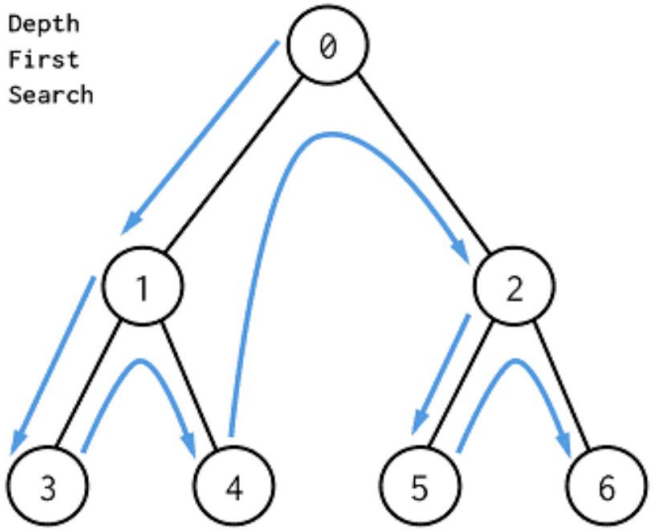
Graph Traversal

Connectivity

- $s-t$ connectivity problem: Given two nodes s and t, is there a path between s and t?
- $s-t$ shortest path problem: Given two nodes s and t, what is the length of a shortest path between s and t?
- Applications
  - Friendster
  - Maze traversal
  - Kevin Bacon number
  - Fewest hops in a communication network

Graph Traversal



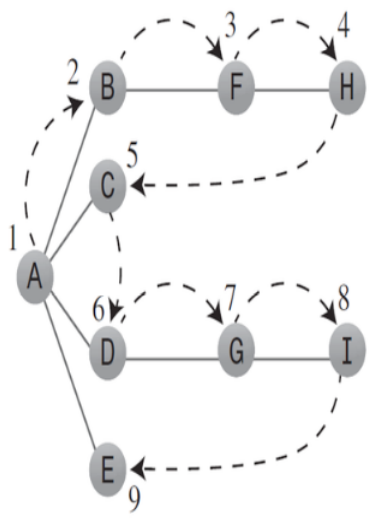


- Traversing the graph means examining all the nodes and vertices of the graph
- Two standard methods to traverse graphs
  - Breadth First Search
  - Depth First Search

Depth First Search (Stack-Based)

- DFS: starts with the initial node, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.
  - Step 1: SET STATUS = 1 (ready state) for each node in G
  - Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
  - Step 3: Repeat Steps 4 and 5 until STACK is empty
  - Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
  - Step 5: Push on the stack all the neighbours of N with STATUS = 1 and set their STATUS = 2
  - Step 6: EXIT

Depth First Search (Stack-Based)



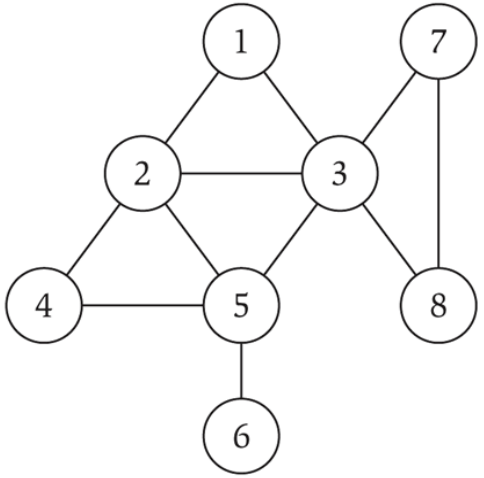
Event	Stack
	A
Visit & Pop A	
Push A's unvisited neighbours	EDCB
Visit & Pop B	EDC
Push B's unvisited neighbours	EDCF
Visit & Pop F	EDC
Push F's unvisited neighbours	EDCH
Visit & Pop H	EDC
Push H's unvisited neighbours	EDC
Visit & Pop C	ED
Push C's unvisited neighbours	ED
Visit & Pop D	E
Push D's unvisited neighbours	EG
Visit & Pop G	E
Push G's unvisited neighbours	EI
Visit & Pop I	E
Push I's unvisited neighbours	E
Visit & Pop E	
Done	

- Rule 1: Push every unvisited neighbour (if there is one) of the current vertex on the stack
- Rule 2: If you can't carry out Rule 1 because there are no more unvisited vertices, pop a vertex from the stack (if possible), mark it visited, and make it the current vertex
- Rule 3: If you can't carry out Rule 1&2 because the stack is empty, you're done

Time complexity $O(m+n)$, with an adjacency list

Depth First Search (Recursion-Based)

DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)



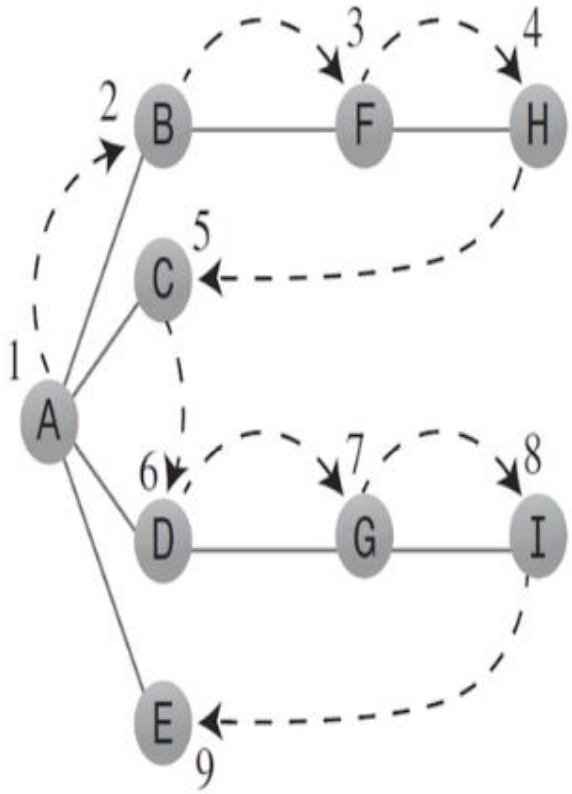
Assume that we follow neighbours with smaller ID first

DFS-recursive($G, 1$) = $[1, 2, 4, 5, 6, \mathbf{3}, 7, 8]$

Time complexity $O(n + m)$, when implemented using an adjacency list.

Traversals

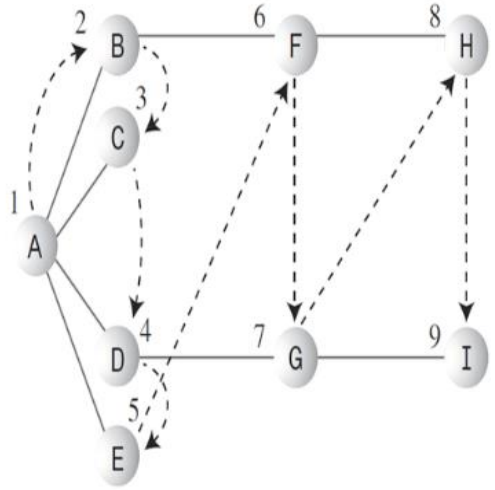
Now try to implement the two DFS traversal algorithms (20 min)



Breadth First Search

- BFS: starts traversing the graph from root node and explores all the neighbours. Then, it selects the nearest node and explore all the unexplored nodes. It follows the same process for each of the nearest node until it finds the goal.
  - Step 1: SET STATUS = 1 (ready state) for each node in G
  - Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
  - Step 3: Repeat Steps 4 and 5 until QUEUE is empty
  - Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
  - Step 5: Enqueue all neighbours of N in the ready state (STATUS = 1) and set their STATUS = 2
  - Step 6: EXIT

Breadth First Search



Event	Queue (Front to Rear)
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	
Done	

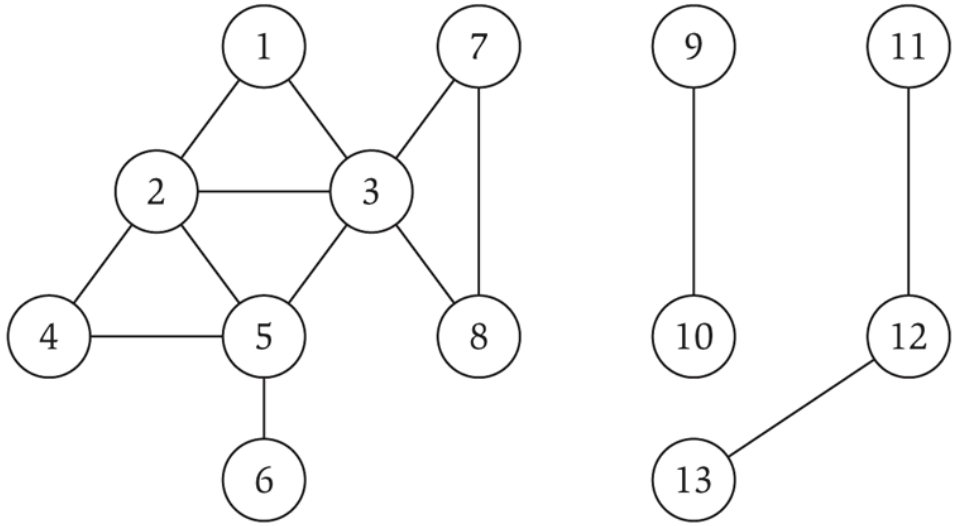
- Rule 1: Visit the next unvisited vertex (if there is one) that's adjacent to the current vertex, and insert it into the queue
- Rule 2: If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it the current vertex
- Rule 3: If you can't carry out Rule 1&2 because the queue is empty, you're done

Time complexity $O(m+n)$, when implemented using an adjacency list

Graph Connectivity

Connected Component

- Connected component: find all nodes reachable from s



Connected component containing node 1 is [1, 2, 3, 4, 5, 6, 7, 8]

Connected Component

- Connected component: find all nodes reachable from s

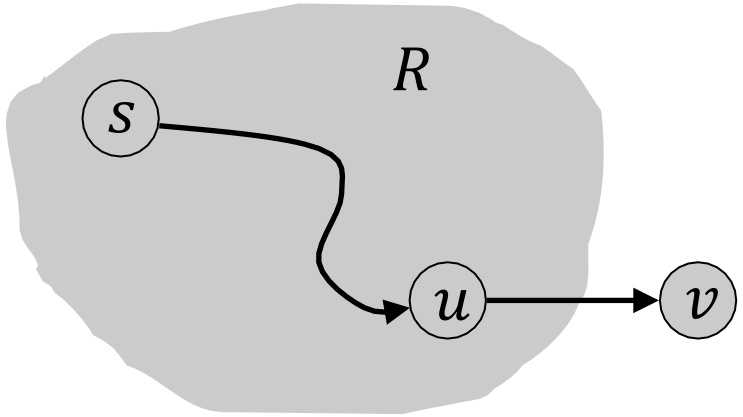
R will consist of nodes to which s has a path

Initially $R = \{s\}$

While there is an edge (u, v) where $u \in R$ and $v \notin R$

  Add v to R

Endwhile



it's safe to add v

Theorem. Upon termination, R is the connected component containing s

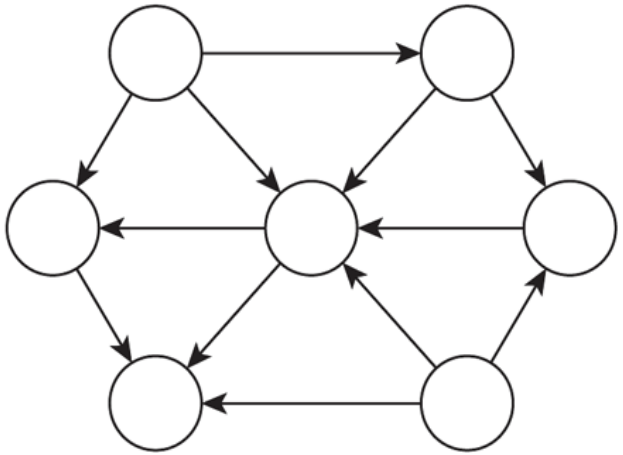
- BFS = explore in order of distance from s
- DFS = explore in a different way

Connectivity in Directed Graphs

Directed Graph

Notation: $G = (V, E)$

- Edge (u, v) leaves node u and enters node v



Ex. Web graph: hyperlink points from one web page to another

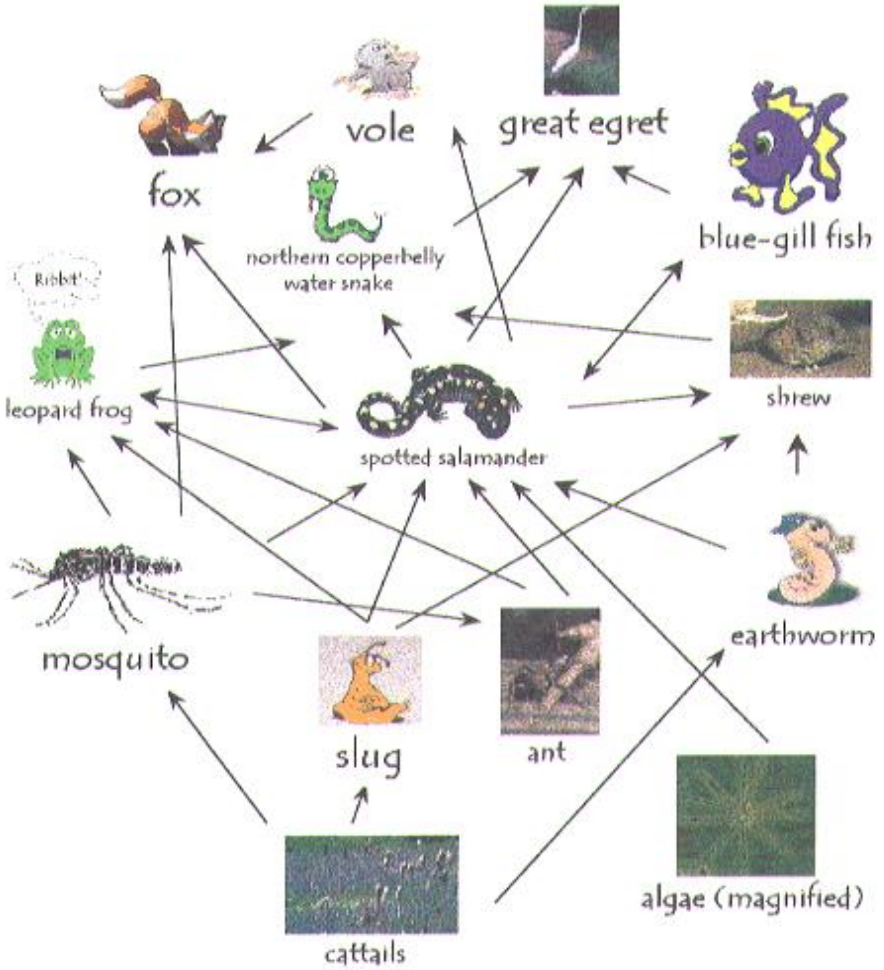
- Orientation of edges is crucial
- Modern web search engines exploit hyperlink structure to rank web pages by importance

Application: Ecological Food Web

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Food web graph

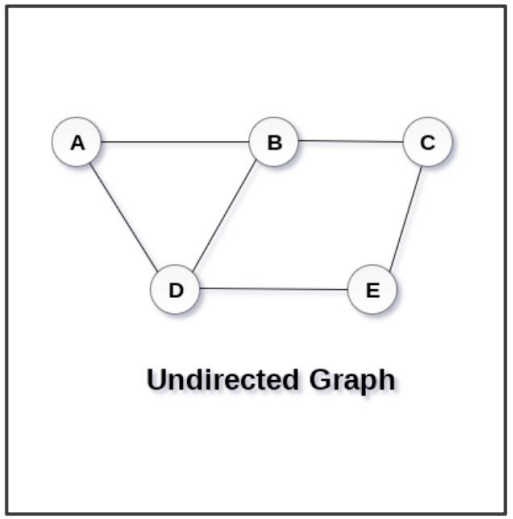
- Node = species
- Edge = from prey to predator



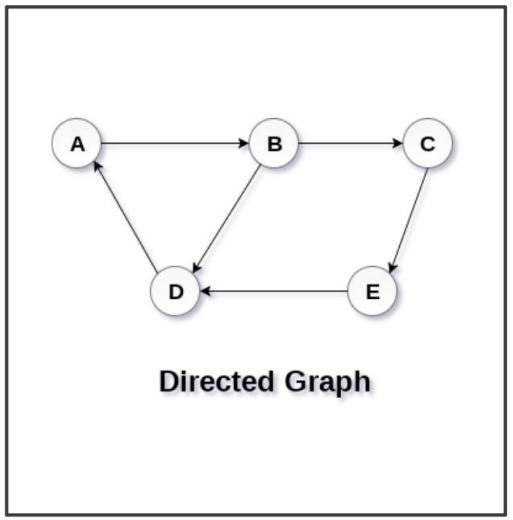
More Applications

directed graph	node	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Undirected V.S. Directed



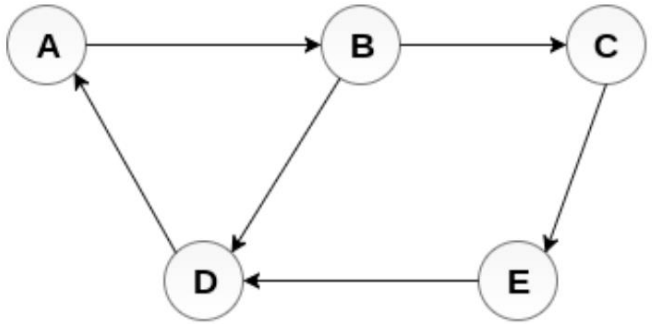
- In an undirected graph, edges are not associated with the directions with them
  - If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B



- In a directed graph, edges form an ordered pair
  - Edges represent a specific path from some vertex A to another vertex B
  - Node A is called initial node while node B is called terminal node

Sequential Representation

- Directed: an entry A_{ij} in the adjacency matrix will be 1 if there exists an edge directly from v_i to v_j



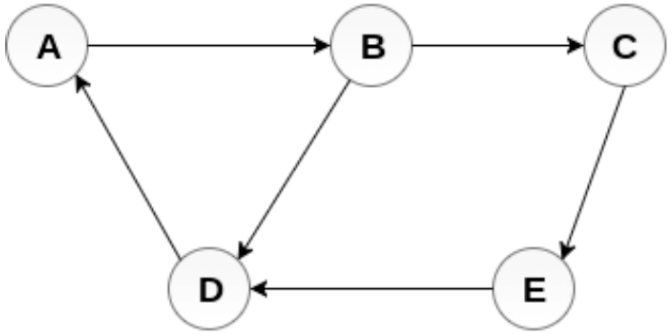
Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

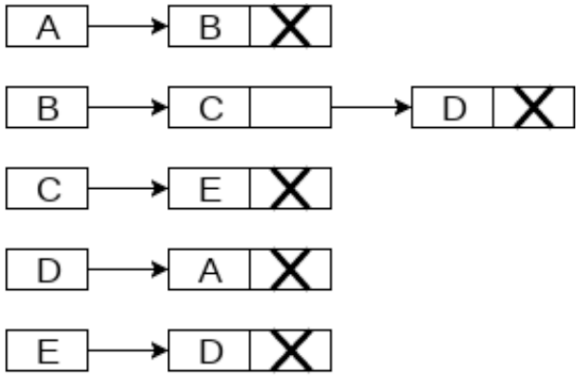
Adjacency Matrix

Linked Representation

- Directed: The sum of the lengths of adjacency lists is equal to the number of edges



Directed Graph



Adjacency List

Graph Search

- Directed reachability: Given a node s, find all nodes reachable from s
- Directed $s \rightsquigarrow t$ shortest path problem: Given two nodes s and t, what is the length of a shortest path from s to t?

coming soon!

- Graph Traversal: BFS and DFS extend naturally to directed graphs

Strong Connectivity

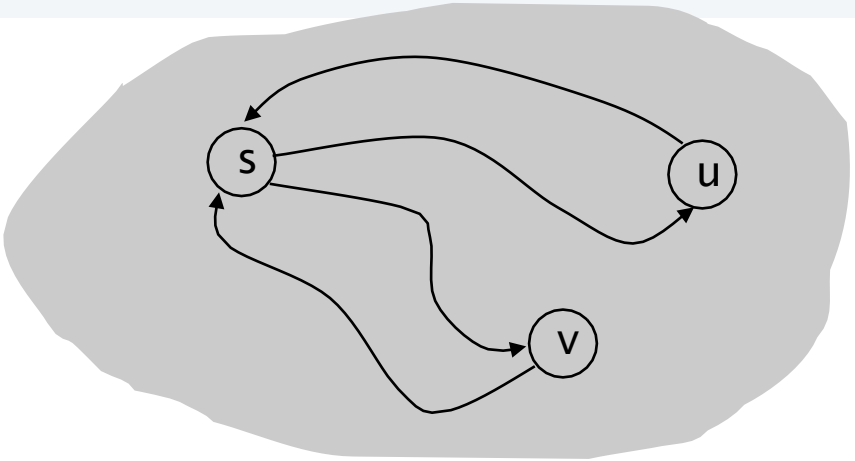
- Def. Nodes u and v are mutually reachable if there is both a path from u to v and also a path from v to u
- Def. A graph is strongly connected if every pair of nodes is mutually reachable
- Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node

Pf. $\Rightarrow$ Follows from definition

Pf. $\Leftarrow$ Path from u to v: concatenate $u \rightsquigarrow s$ path with $s \rightsquigarrow v$ path

Path from v to u: concatenate $v \rightsquigarrow s$ path with $s \rightsquigarrow u$ path ▪

ok if paths overlap



Strong Connectivity: Algorithm

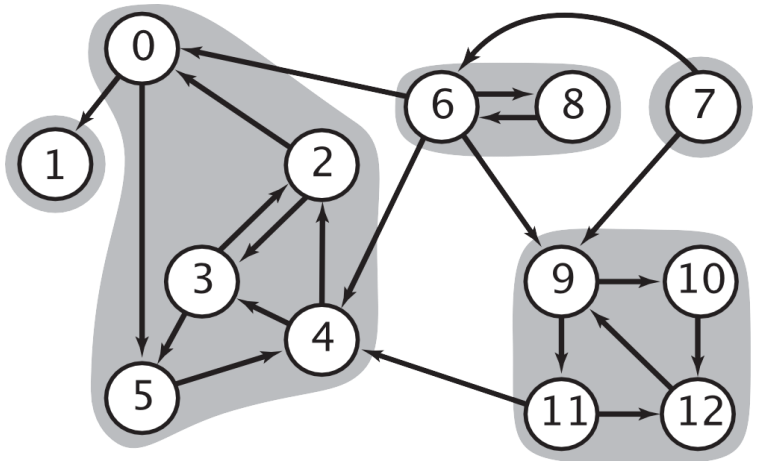
- Theorem. Can determine if G is strongly connected in $O(m + n)$ time

Pf.

- Pick any node s
- Run BFS from s in G
- Run BFS from s in G reverse
- Return true iff all nodes reached in both BFS executions
- Correctness follows immediately from previous lemma ▪

Strong Components

- Def. A strong component is a maximal subset of mutually reachable nodes



Theorem. [Tarjan 1972] Can find all strong components in $O(m+n)$ time

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

Abstract. The value of depth-first search or "backtracking" as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirect graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2, and k_3, where V is the number of vertices and E is the number of edges of the graph being examined.

Tarjan's Algorithm: Overview

1. **Initialization**:
    1. Assign a unique index to each node, initialize as undefined
    2. Assign a lowlink value to each node, initialize as undefined
    3. Create an empty stack to keep track of nodes in the current search path
    4. Create an empty list to store the strongly connected components (SCCs)
2. **Depth-First Search (DFS) Loop**:
    1. For each node v in the graph:
        1. If v has not been visited:
            1. Call the strongConnect function on v
3. **strongConnect Function**:
    1. Set the index of v to the current global index
    2. Set the lowlink of v to the current global index
    3. Push v onto the stack
    4. Mark v as being on the stack
    5. Increment the global index
4. **Explore Adjacent Nodes**:
    1. For each adjacent node u of v:
        1. If u has not been visited:
            1. Recursively call strongConnect(u)
            2. Update the lowlink of v to the minimum of v.lowlink and u.lowlink
        2. If u is on the stack:
            1. Update the lowlink of v to the minimum of v.lowlink and u.index
5. **Identify SCC**:
    1. If the lowlink of v is equal to its index:
        1. Pop nodes from the stack until v is popped
        2. Each popped node is part of a new SCC
        3. Add the popped nodes to the list of SCCs
6. **Output**:
    1. After all nodes have been processed, the list of SCCs contains all the strongly connected components of the graph

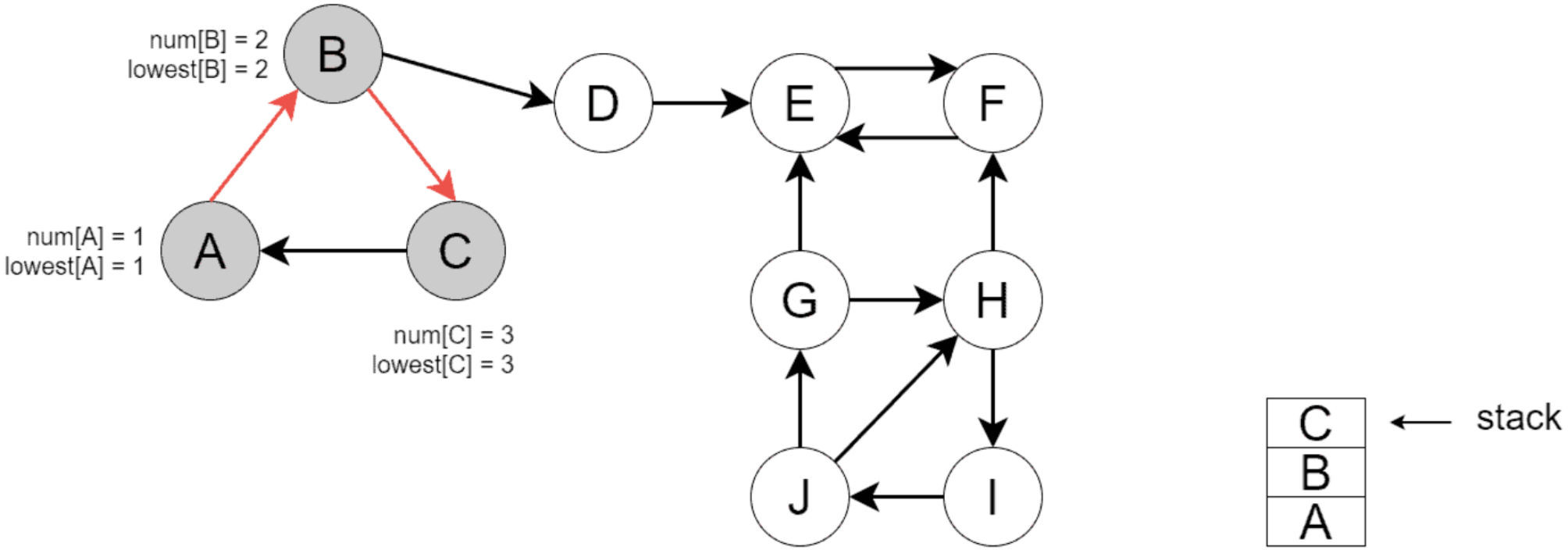
Tarjan's Algorithm: Pseudocode

```
// GLOBAL VARIABLES
//    num <- global array of size V initialized to -1
//    lowest <- global array of size V initialized to -1
//    visited <- global array of size V initialized to false
//    processed <- global array of size V initialized to false
//    s <- global empty stack
//    i <- 0
```

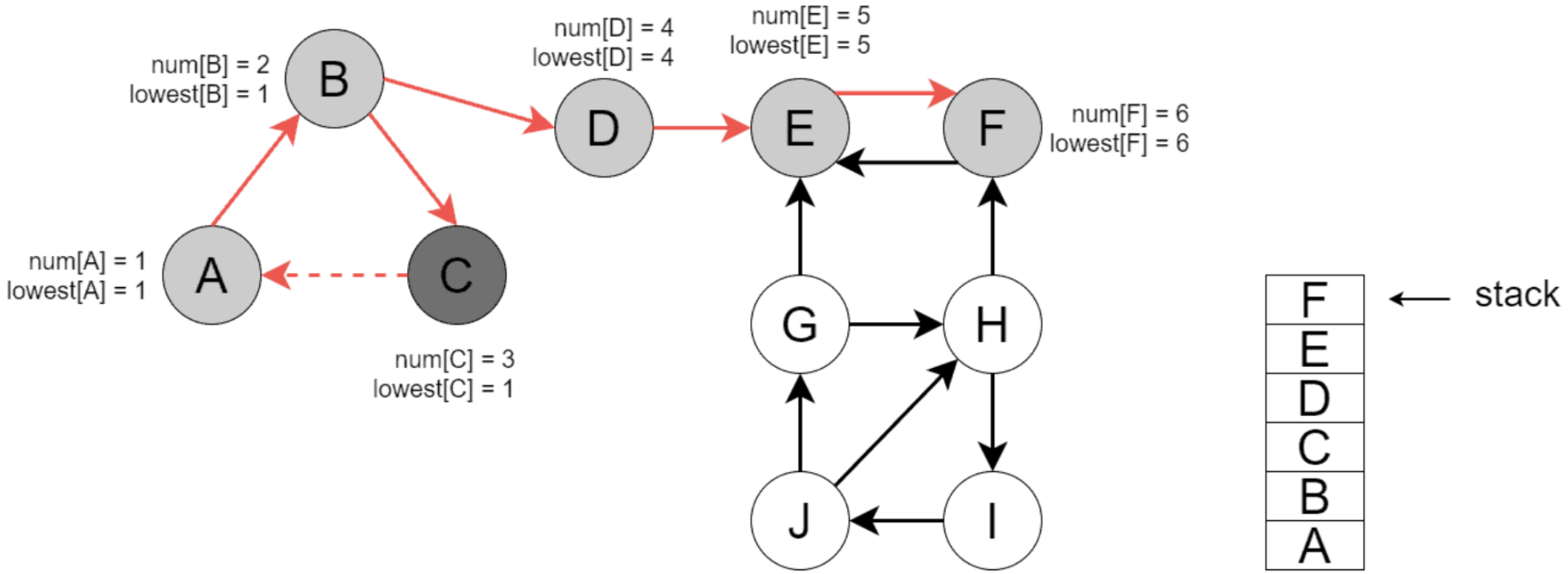
```
algorithm TarjanAlgorithm(G):
    // INPUT
    //    G = the graph
    // OUTPUT
    //    SCCs of G are found

    visted <- an empty global visited map
    for v in G.V:
        if visited[v] = false:
            // global variables are accessible from within DFS
            DFS(G, v)
```

```
algorithm DFS(G, v):
    // INPUT
    //    G = the graph
    //    v = the current vertex
    // OUTPUT
    //    Vertices reachable from v are processed, their SCCs are reported

    num[v] <- i
    lowest[v] <- num[v]
    i <- i + 1
    visited[v] <- true
    s.push(v)

    for u in G.neighbours[v]:
        if visited[u] = false:
            DFS(G, u)
            lowest[v] <- min(lowest[v], lowest[u])
        else if processed[u] = false:
            lowest[v] <- min(lowest[v], num[u])

    processed[v] <- true

    if lowest[v] = num[v]:
        scc <- an empty set
        sccVertex <- s.pop()

        while sccVertex != v:
            scc.add(sccVertex)
            sccVertex <- s.pop()

        scc.add(sccVertex)

        Process the found scc in the desired way

    return
```

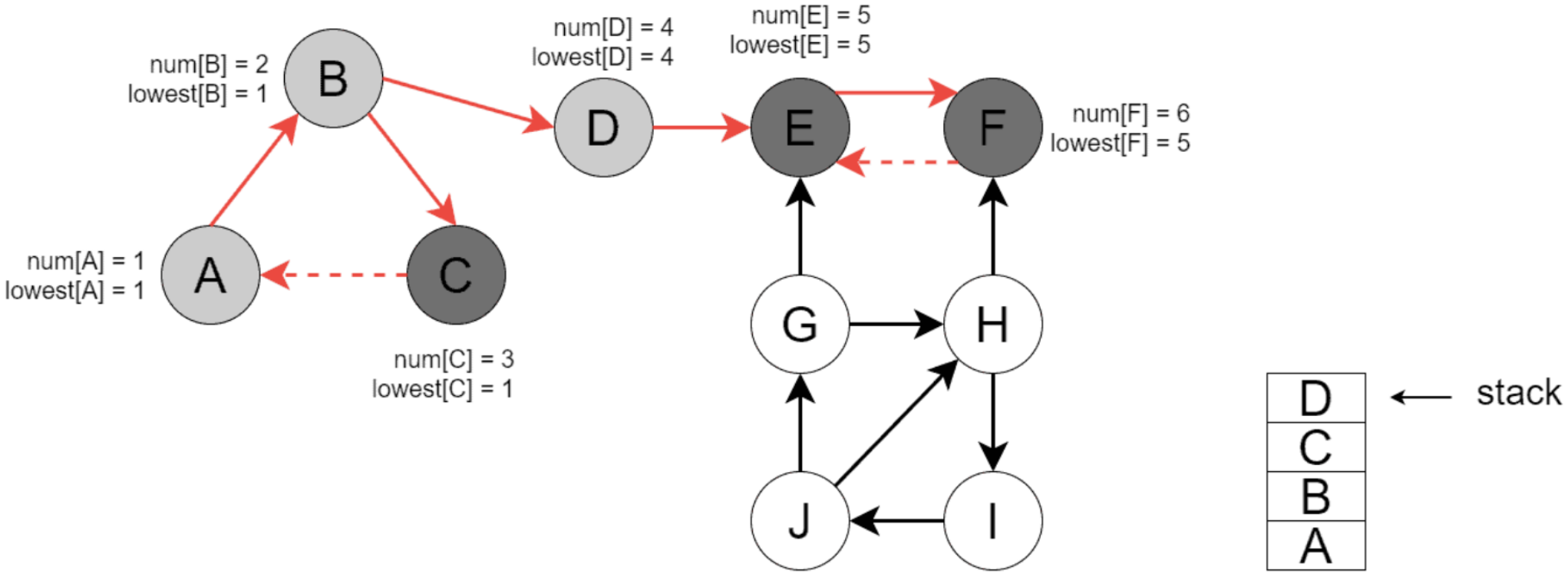
Tarjan's Algorithm: An Example



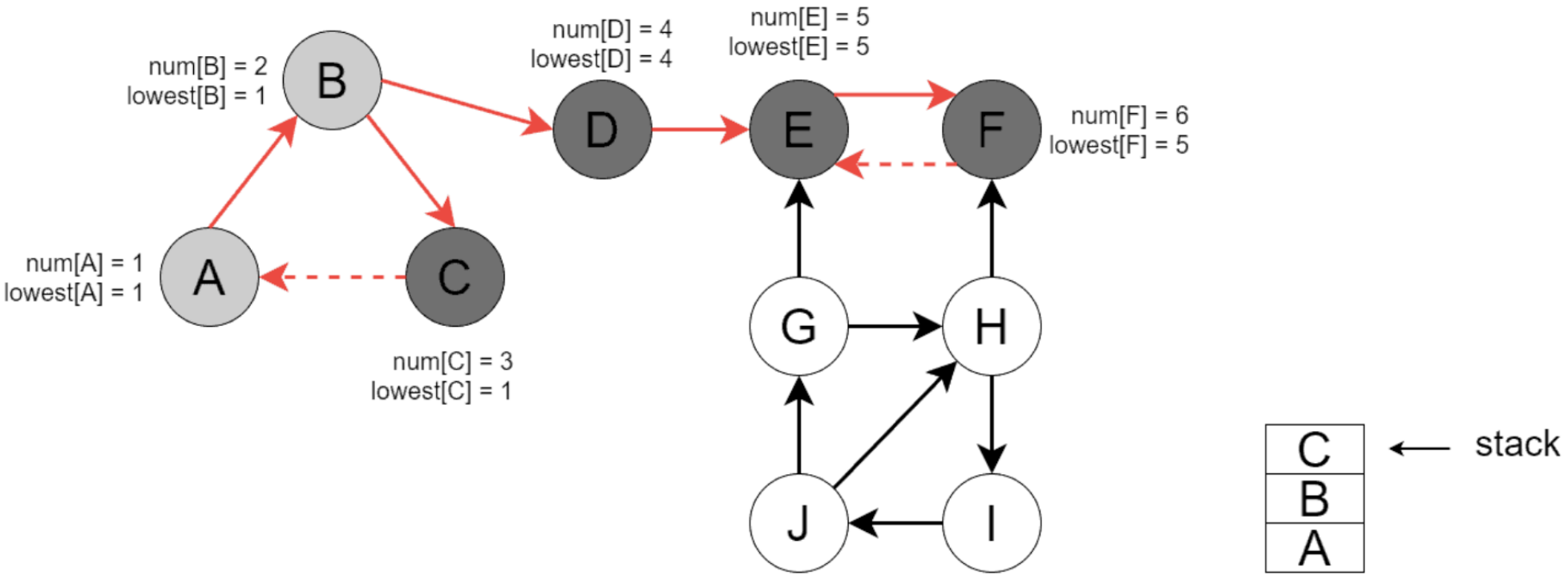
Tarjan's Algorithm: An Example



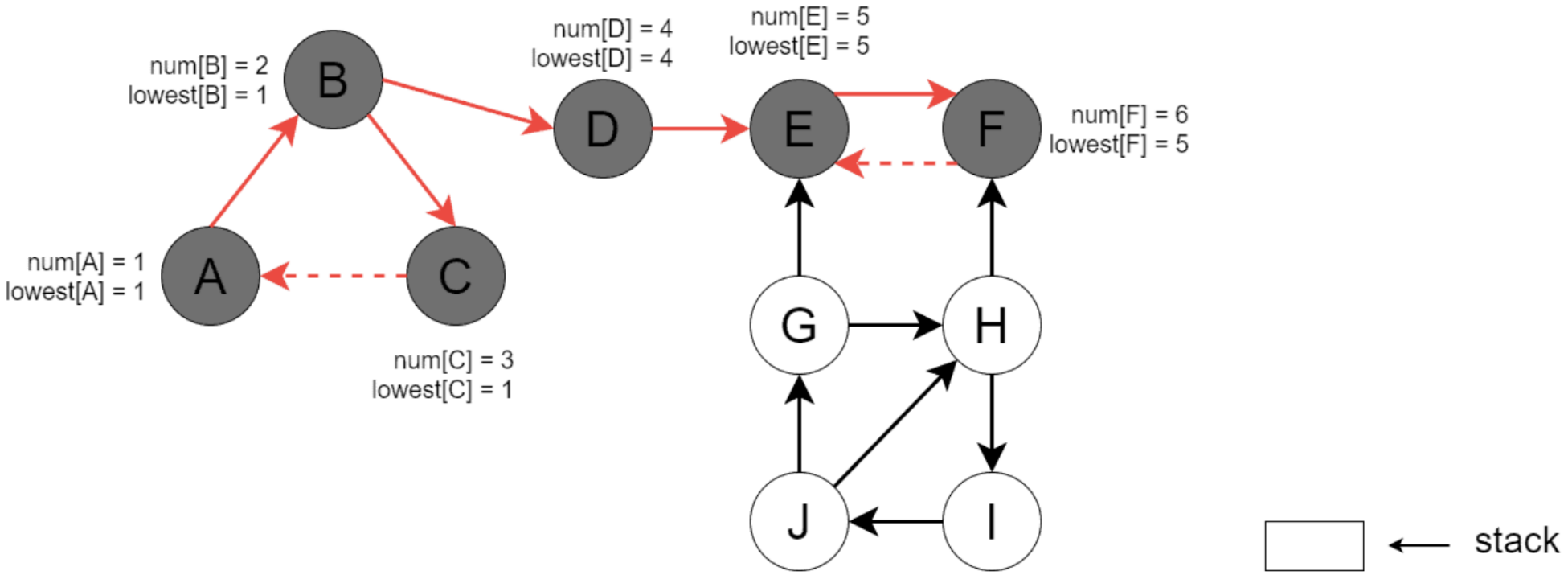
Tarjan's Algorithm: An Example



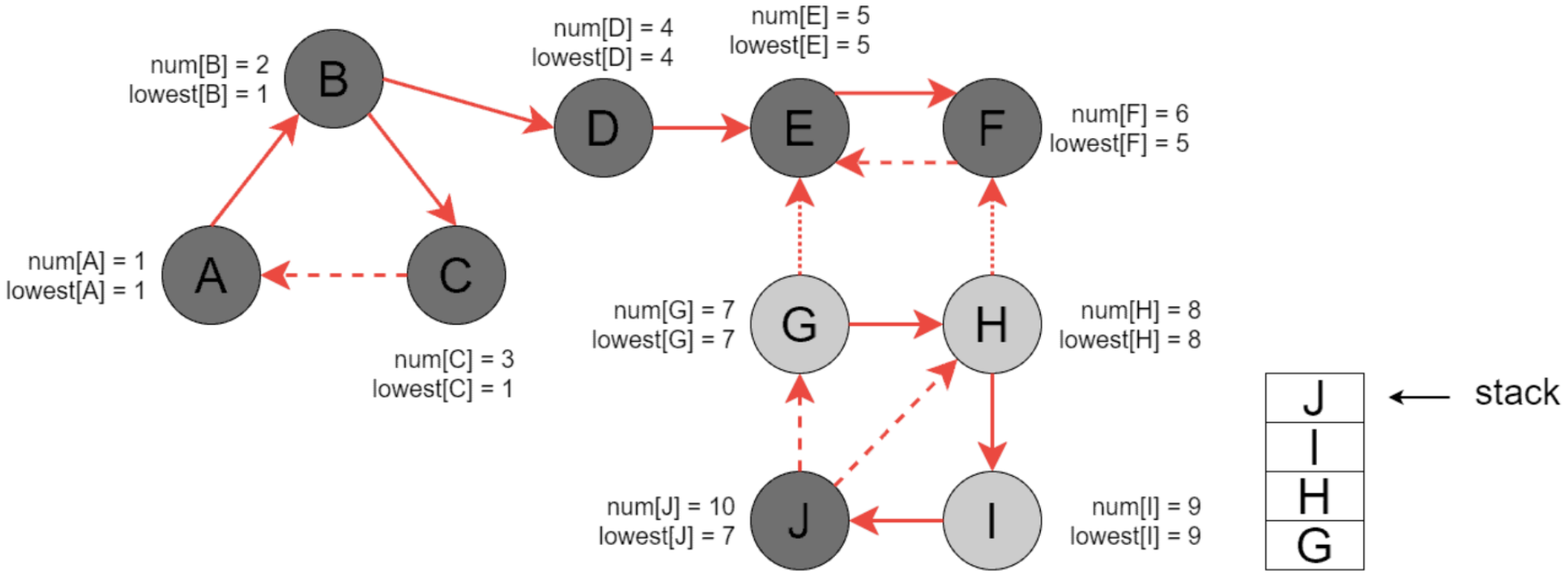
Tarjan's Algorithm: An Example



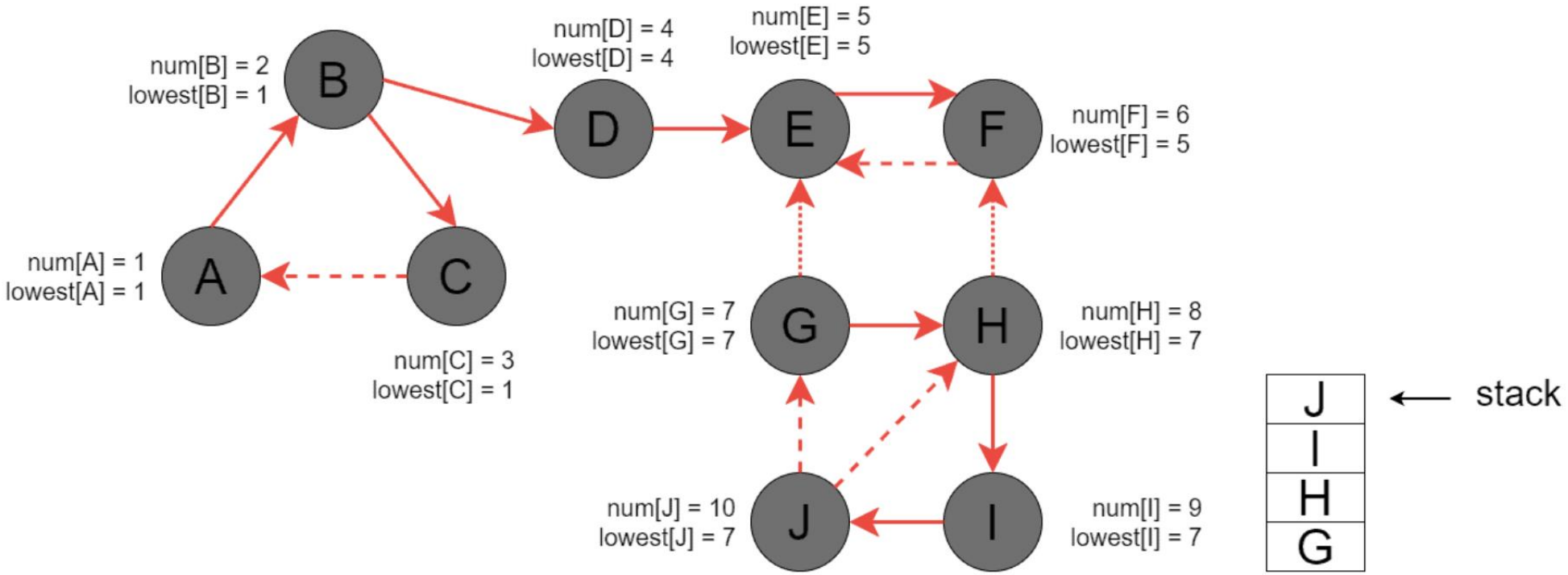
Tarjan's Algorithm: An Example



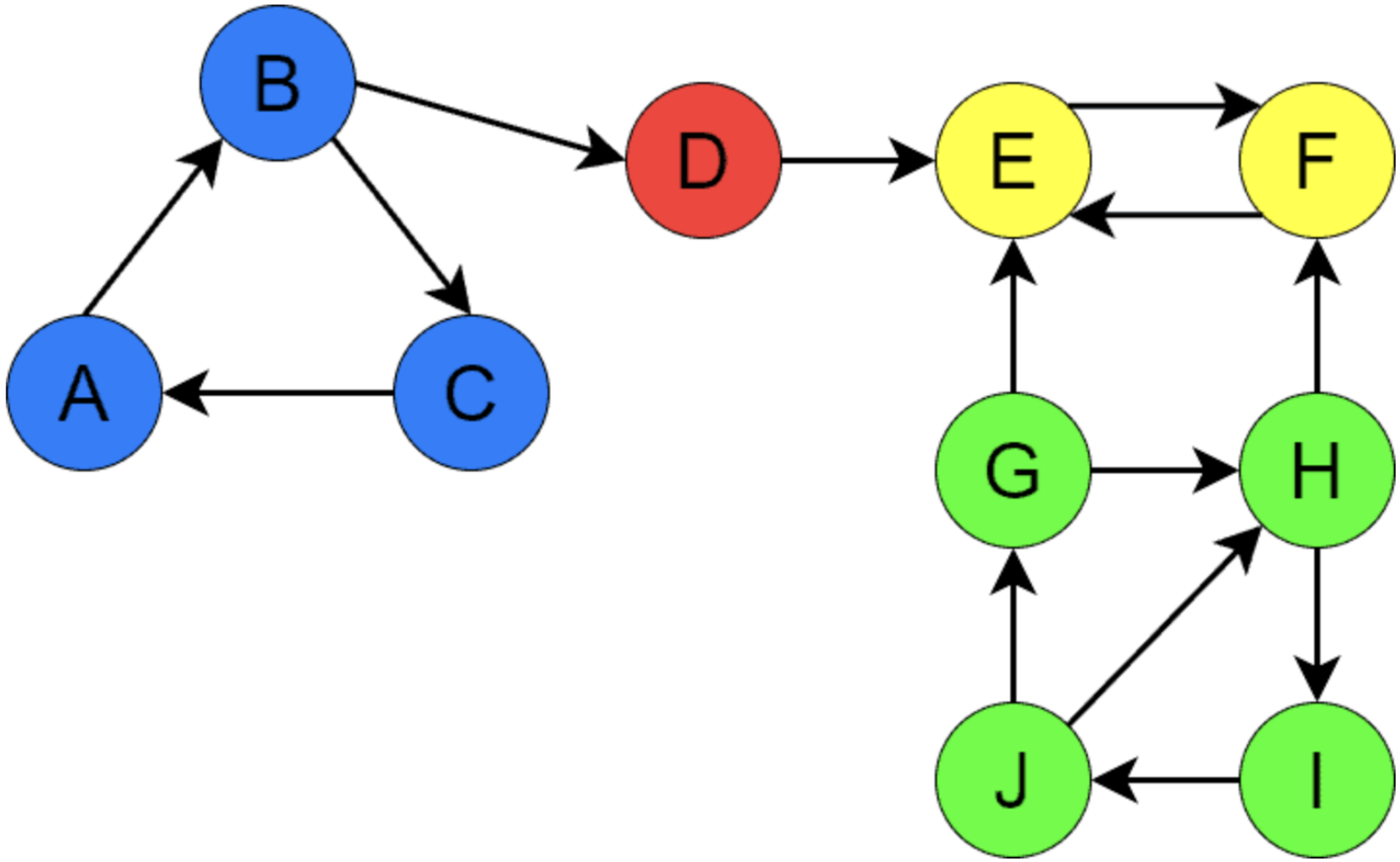
Tarjan's Algorithm: An Example



Tarjan's Algorithm: An Example



Tarjan's Algorithm: An Example



Tarjan's Algorithm

- Tarjan's algorithm is a modification of the DFS traversal. Hence, the complexity of the algorithm is linear: $O(n+m)$
  - To achieve the mentioned complexity, we must use the adjacency list representation of the graph

- Tarjan's algorithm for finding strongly connected components in directed graphs. It's an optimal linear time algorithm

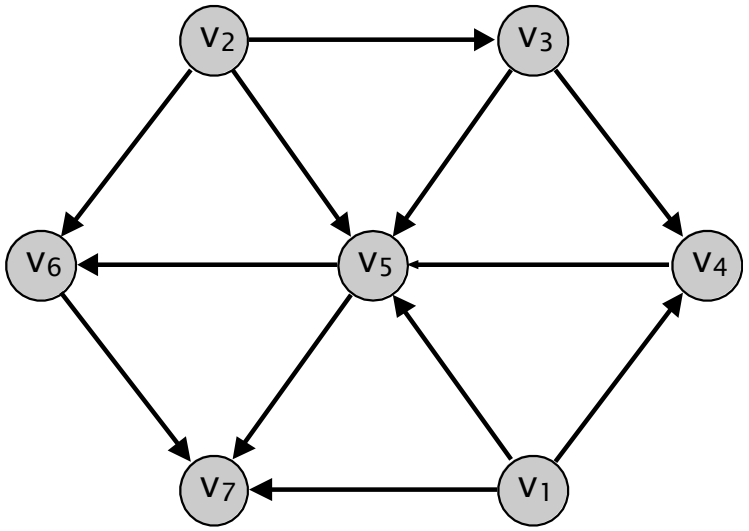
- More Tarjan's algorithms, have a try if you are interested!

DAG & Topological Ordering

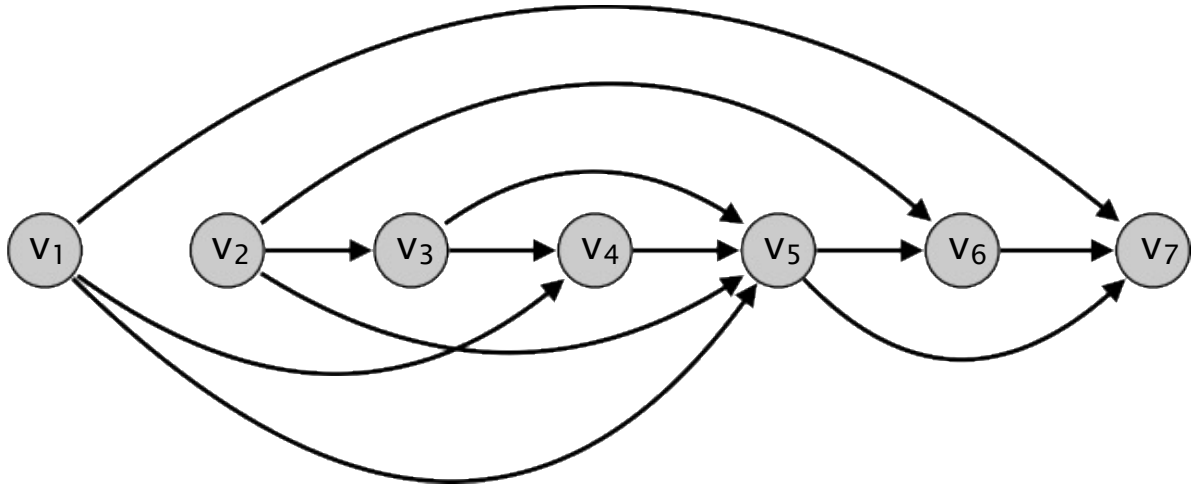
Directed Acyclic Graphs

- Def. A DAG is a directed graph that contains no directed cycles

- Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$

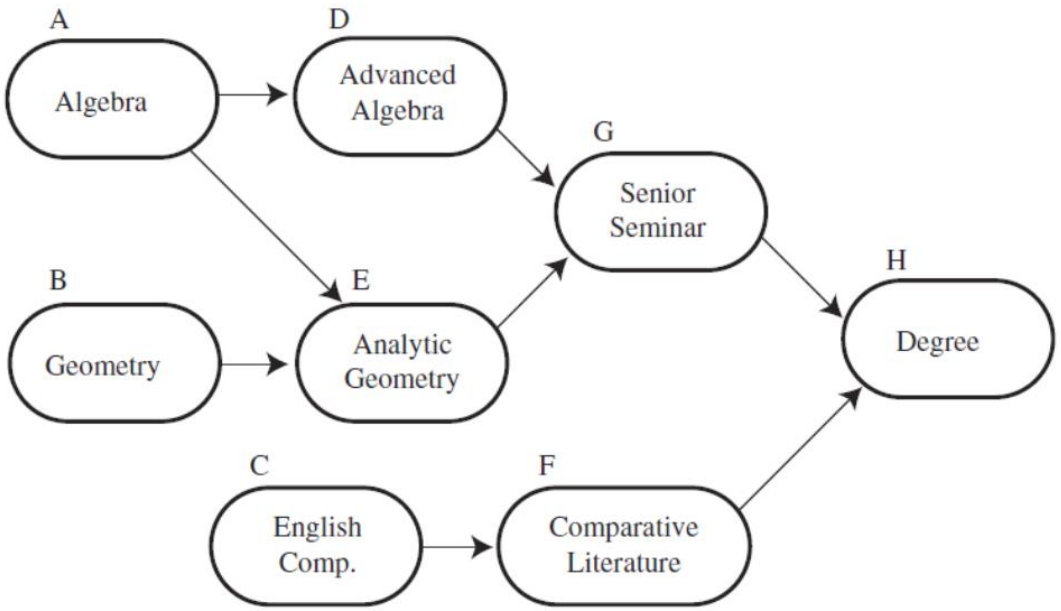


a DAG



a topological ordering

Precedence Constraints



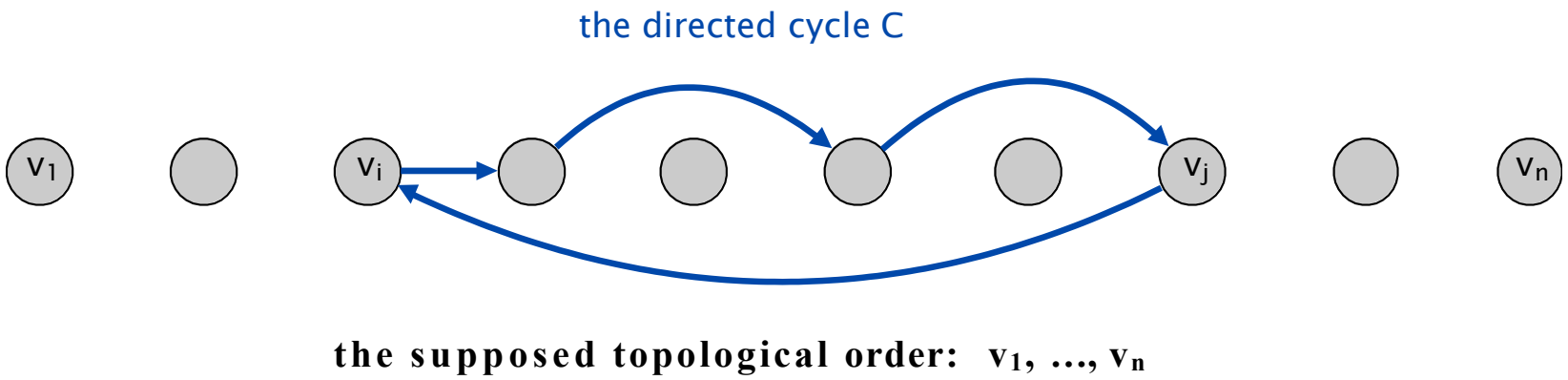
- Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j
- Applications
  - Course prerequisite graph: course v_i must be taken before v_j
  - Compilation: module v_i must be compiled before v_j
  - Pipeline of computing jobs: output of job v_i needed to determine input of job v_j

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG

Pf. [by contradiction]

- Suppose that G has a topological order $v_1, v_2, \ldots, v_n$ and that G also has a directed cycle C
- Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge
- By our choice of i, we have $i < j$
- On the other hand, since (v_j, v_i) is an edge and $v_1, v_2, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction ▪



Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG

Q. Does every DAG have a topological ordering?

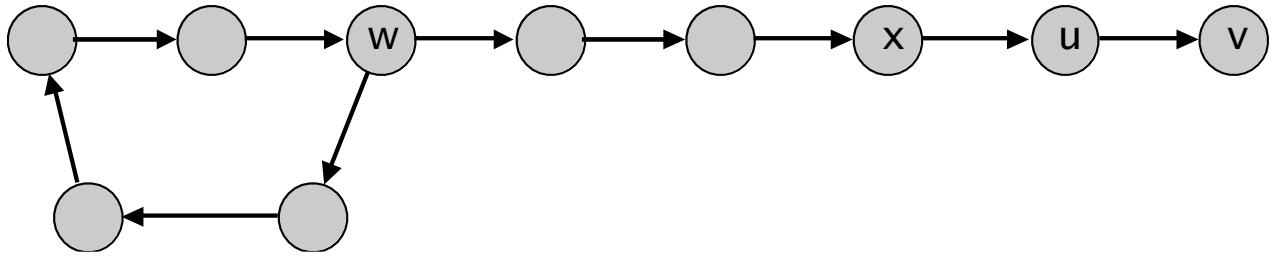
Q. If so, how do we compute one?

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no entering edges

Pf. [by contradiction]

- Suppose that G is a DAG and every node has at least one entering edge
- Pick any node v, and begin following edges backward from v. Since v has at least one entering edge (u, v) we can walk backward to u
- Then, since u has at least one entering edge (x, u), we can walk backward to x
- Repeat until we visit a node, say w, twice
- Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle ▪



Directed Acyclic Graphs

Lemma. If *G* is a DAG, then *G* has a topological ordering

Pf. [by induction on n]

- Base case: true if $n = 1$
- Given DAG on $n > 1$ nodes, find a node v with no entering edges
- $G - \{v\}$ is a DAG, since deleting v cannot create cycles
- By inductive hypothesis, $G - \{v\}$ has a topological ordering
- Place v first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since v has no entering edges ▪

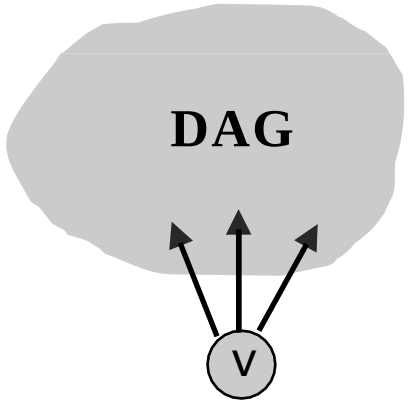
To compute a topological ordering of G:

Find a node v with no incoming edges and order it first

Delete v from G

Recursively compute a topological ordering of $G-\{v\}$

  and append this order after v



Topological Sorting Algorithm

- Theorem. Algorithm finds a topological order in $O(m + n)$ time
- Pf.
  - Maintain the following information:
    - *count*(w) = remaining number of incoming edges
    - S = set of remaining nodes with no incoming edges
  - Initialization: $O(m + n)$ via single scan through graph
  - Update: to delete v
    - *remove v from S*
    - *decrease count(w) for all edges from v to w; and add w to S if count(w) hits* 0
    - *this is O(1) per edge* ∎

- Topological-sort cannot handle graphs with cycles!

Summary

- Graphs definition
- Graphs representation
- Graph search algorithms
- Connected components in directed/undirected graphs
- Tarjan’s Algorithm
- DAGs and Topological orders

The End