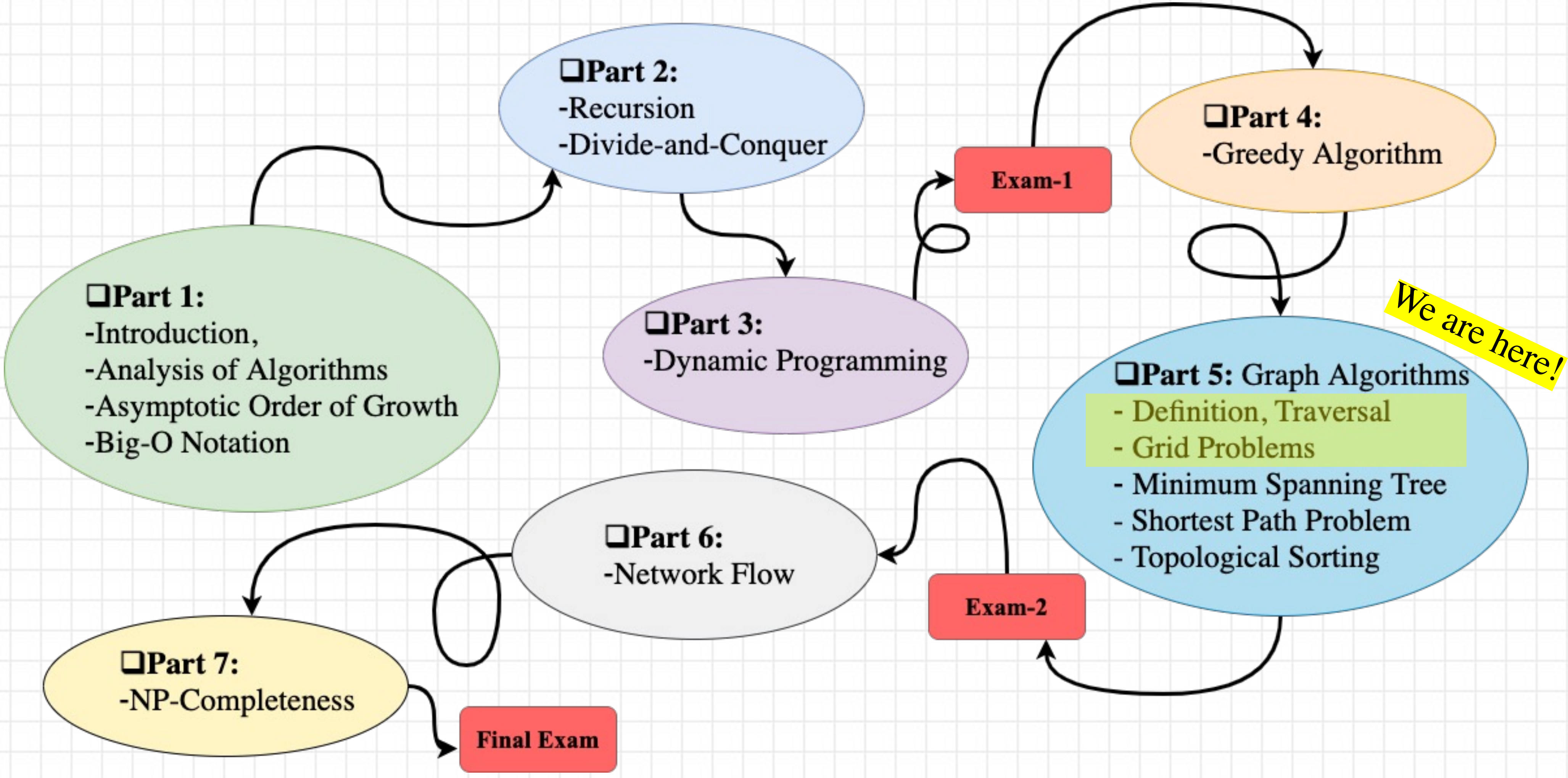
CS-3510: Design and Analysis of Algorithms

Graph Algorithms: Definitions and Traversal

Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology
Summer 2022

Roadmap


❑Part 1:
-Introduction,
-Analysis of Algorithms
-Asymptotic Order of Growth
-Big-O Notation
❑Part 2:
-Recursion
-Divide-and-Conquer
❑Part 3:
-Dynamic Programming
Exam-1
❑Part 4:
-Greedy Algorithm
We are here!
❑Part 5: Graph Algorithms
- Definition, Traversal
- Grid Problems
- Minimum Spanning Tree
- Shortest Path Problem
- Topological Sorting
Exam-2
❑Part 6:
-Network Flow
❑Part 7:
-NP-Completeness
Final Exam

Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (<u>unweighted</u> graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components
- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed <u>weighted</u> graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

Graph

- Review of graph definition and representation
  - Adjacency matrix
  - Adjacency list

- Graph traversal
  - Breadth first search (BFS)
  - Depth first search (DFS)

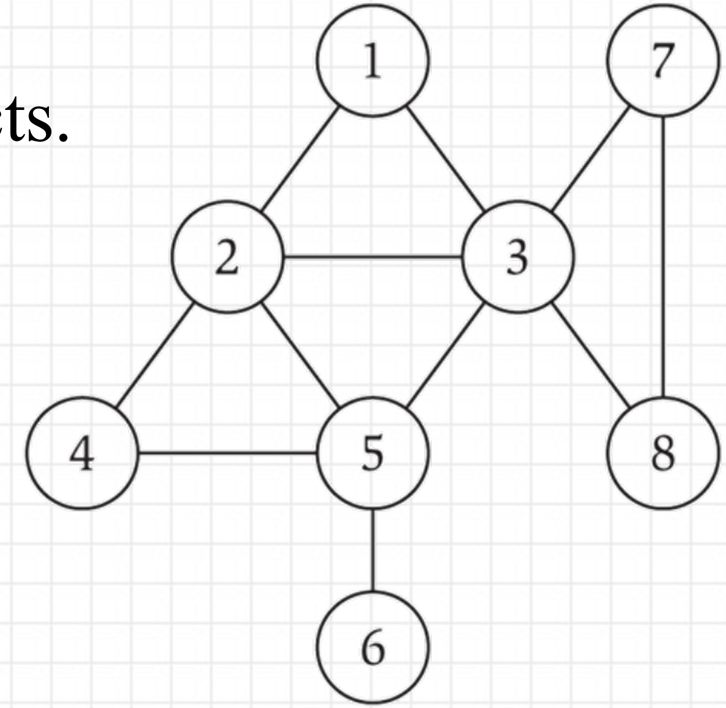
Graph Properties and Terminology Review

- Notation. $G = (V, E)$
  - V = nodes (or vertices).
  - E = edges (or arcs) between pairs of nodes.
  - Captures pairwise relationship between objects.
  - Graph size parameters: $n = |V|$, $m = |E|$.

$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$

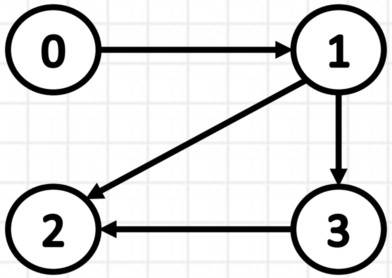
$E = \{ 1–2, 1–3, 2–3, 2–4, 2–5, 3–5, 3–7, 3–8, 4–5, 5–6, 7–8 \}$

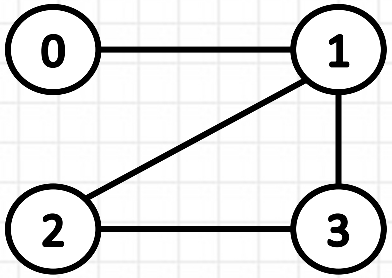
$m = 11, n = 8$



Graph Properties and Terminology Review

- Notation. $G = (V, E)$
  - V = nodes (or vertices). $\{0, 1, 2, \ldots n\text{-}1\}$
  - E = edges (or arcs) between pairs of nodes. $\{e_1, e_2, \ldots e_m\}$ where $e_i = (v_i, v_j)$
  - Captures pairwise relationship between objects.
- Directed vs. undirected

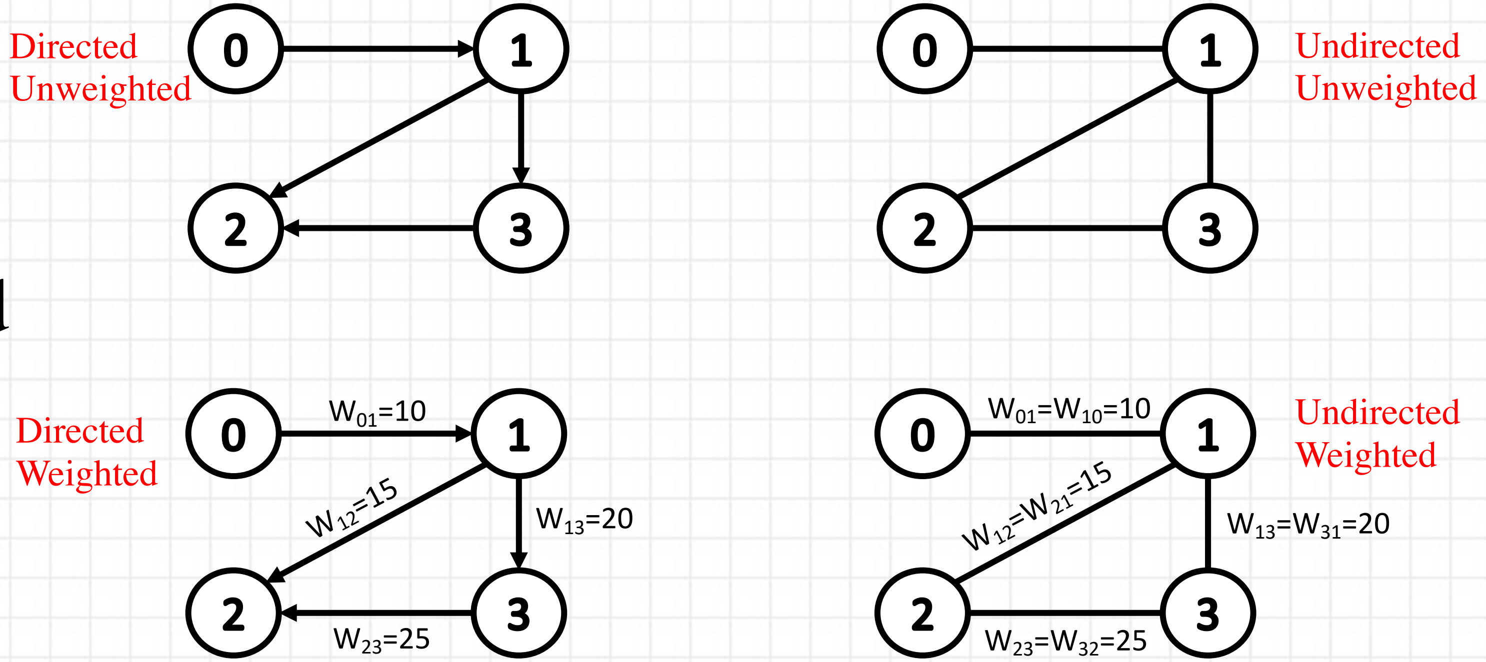




- Weighted vs. unweighted
  - Weights = properties assigned to edges (usually) and/or nodes
  - E.g., distance, cost, time

Graph Properties and Terminology Review

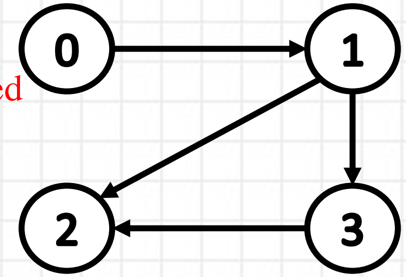
- Notation. $G = (V, E)$
  - V = nodes (or vertices). $\{0, 1, 2, \dots n\text{-}1\}$
  - E = edges (or arcs) between pairs of nodes. $\{e_1, e_2, \dots e_m\}$ where $e_i = (v_i, v_j)$
  - Captures pairwise relationship between objects.
- Directed vs. undirected
  - Directed graph = digraph
- Weighted vs. unweighted

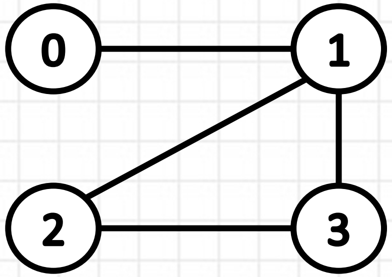


Graph Properties and Terminology Review

- Notation. $G = (V, E)$
  - V = nodes (or vertices). $\{0, 1, 2, \ldots n\text{-}1\}$
  - E = edges (or arcs) between pairs of nodes. $\{e_1, e_2, \ldots e_m\}$ where $e_i = (v_i, v_j)$
  - Captures pairwise relationship between objects.
- Directed vs. undirected
- Weighted vs. unweighted

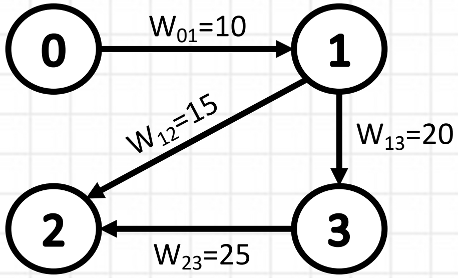
Directed
Unweighted

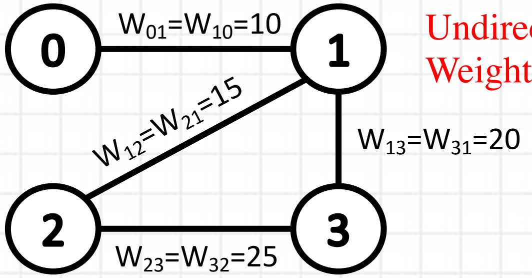




Undirected
Unweighted

Directed
Weighted

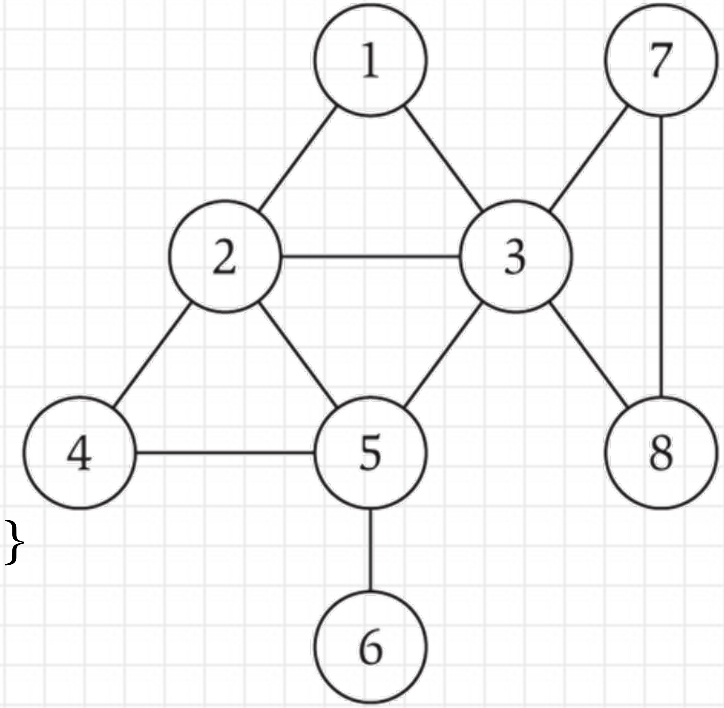




Undirected
Weighted

Graph Properties and Terminology Review

- Notation. $G = (V, E)$
  - V = nodes (or vertices). $\{0, 1, 2, \ldots n-1\}$
  - E = edges (or arcs) between pairs of nodes. $\{e_1, e_2, \ldots e_m\}$ where $e_i = (v_i, v_j)$
  - Captures pairwise relationship between objects.
- Directed vs. undirected
- Weighted vs. unweighted



$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$

$E = \{ 1–2, 1–3, 2–3, 2–4, 2–5, 3–5, 3–7, 3–8, 4–5, 5–6, 7–8 \}$

$m = 11, n = 8$

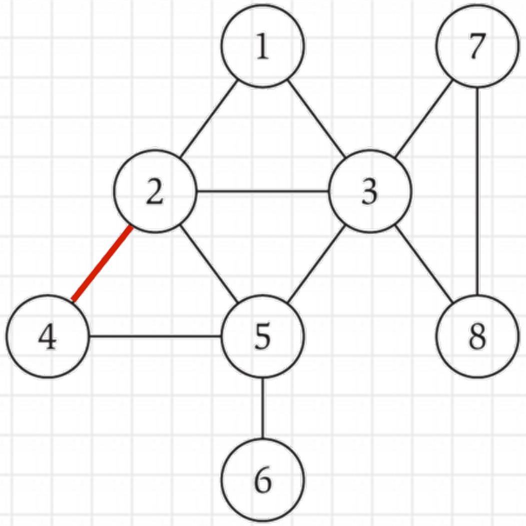
Graph Properties and Terminology Review

- Notation. $G = (V, E)$
  - V = nodes (or vertices).
  - E = edges (or arcs) between pairs of nodes.

- Graph parameters:
  - Graph size parameters: $n = |V|$, $m = |E|$.
  - Degree(i): number of edges on node i
    - In-degree (directed networks): number of incoming links
    - Out-degree (directed networks): the number of outgoing links

Graph Properties and Terminology Review

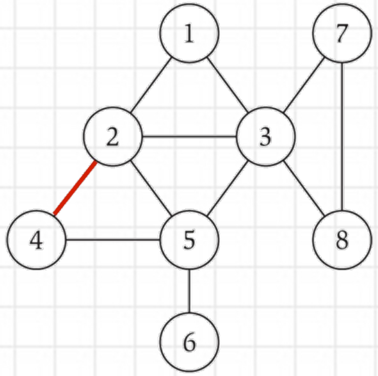
- Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.
  - Two representations of each edge.
  - Space proportional to n^2.
  - Checking if (u, v) is an edge takes $\Theta(1)$ time.
  - Identifying all edges takes $\Theta(n^2)$ time.



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph Properties and Terminology Review

- Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.
  - Two representations of each edge.
  - Space proportional to n^2.
  - Checking if (u, v) is an edge takes $\Theta(1)$ time.
  - Identifying all edges takes $\Theta(n^2)$ time.



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

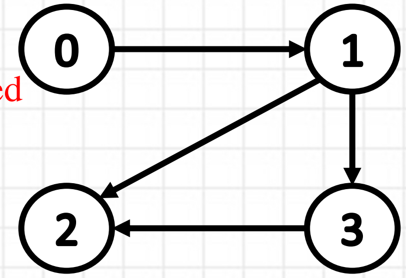
- Notes
  - Weighted graphs → $A_{uv} = w_{uv}$
  - Undirected graphs → $A = A^T$ (symmetric adj. matrix)
    - Duplicate information
  - Inefficient if graphs are sparse (lots of "zero"s)
  - Easy to determine quickly if there is a link between nodes i and j
    - A[i,k] + A[k,j]

Graph Properties and Terminology Review

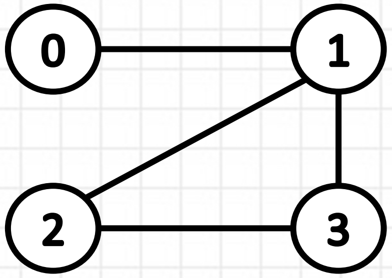
💻 Demo code time!

- Directed vs. undirected

Directed
Unweighted



Graph-1

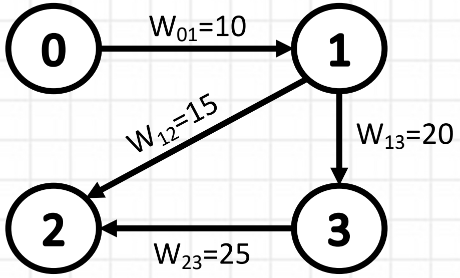


Undirected
Unweighted

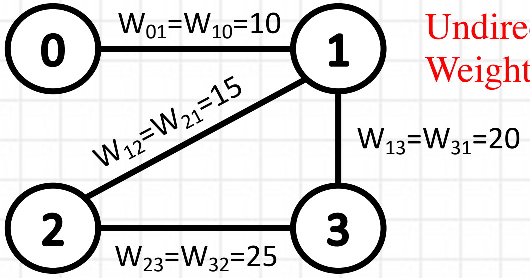
Graph-3

- Weighted vs. unweighted

Directed
Weighted



Graph-2



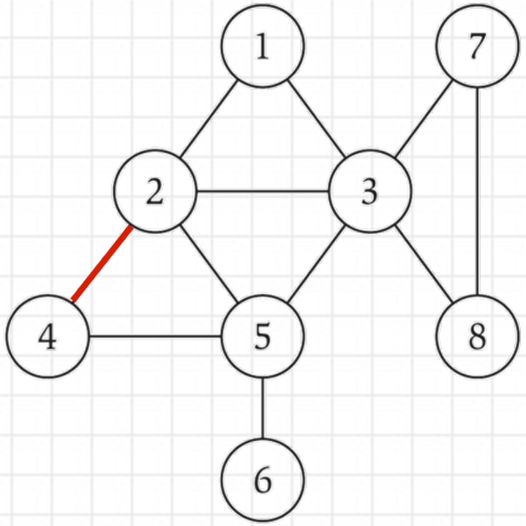
Undirected
Weighted

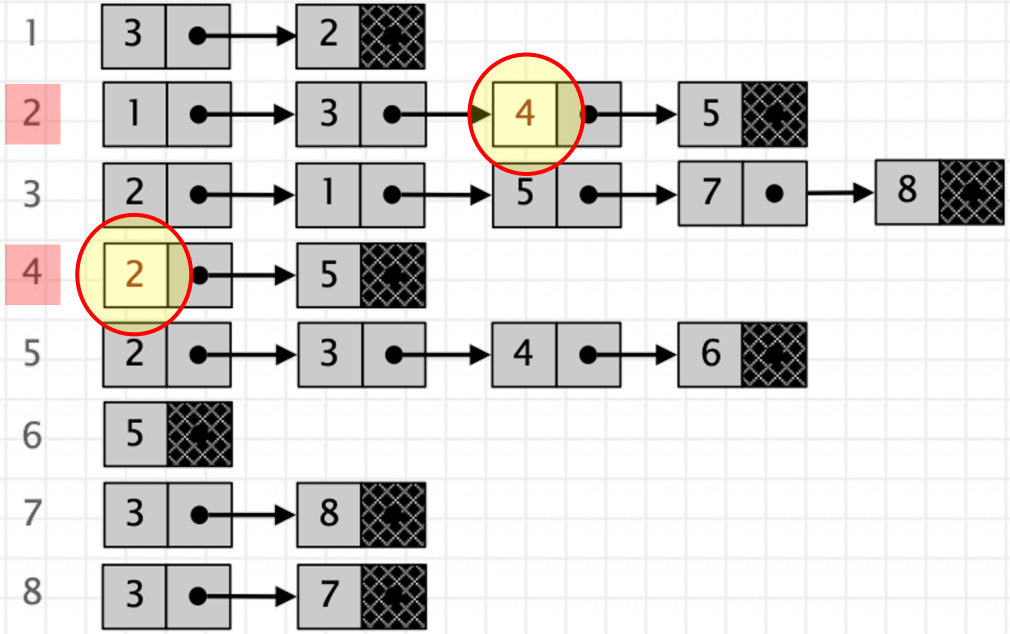
Graph-4

Graph Properties and Terminology Review

- Adjacency lists. Node-indexed array of lists.
  - Two representations of each edge.
  - Space is $\Theta(m + n)$.
  - Checking if (u, v) is an edge takes $O(\text{degree}(u))$ time.

degree = number of neighbors of u

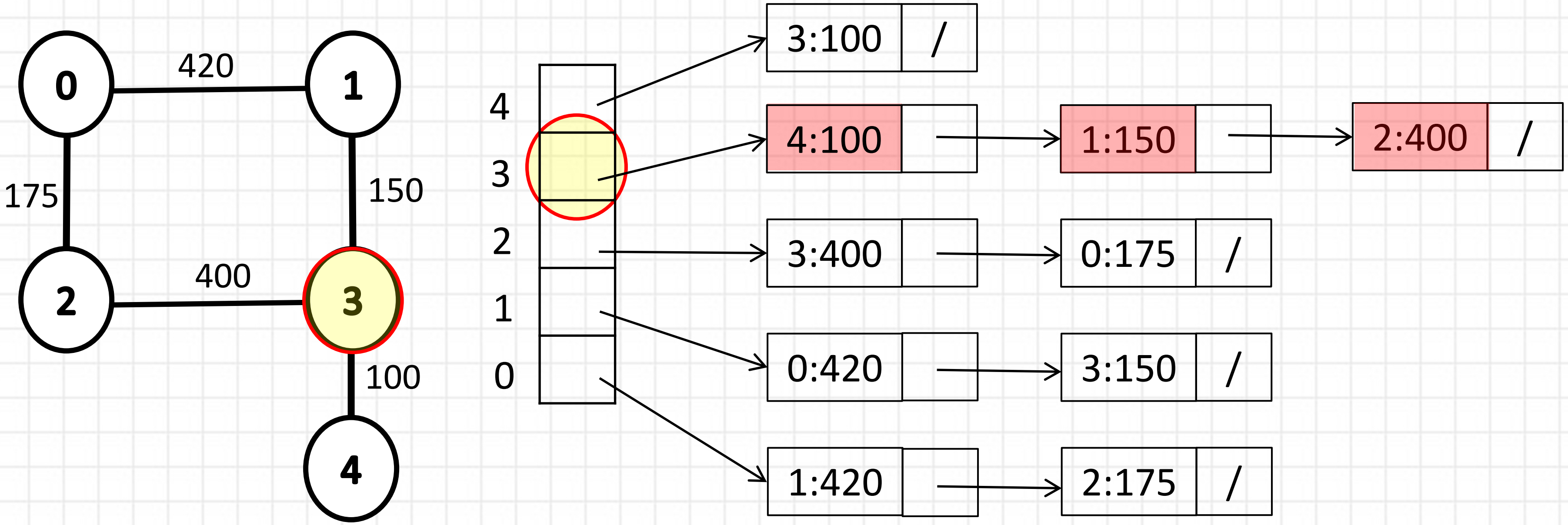




Graph Properties and Terminology Review

- Adjacency lists. Node-indexed array of lists.
  - Two representations of each edge.
  - Space is $\Theta(m + n)$.
  - Checking if (u, v) is an edge takes $O(\text{degree}(u))$ time.

degree = number of neighbors of u

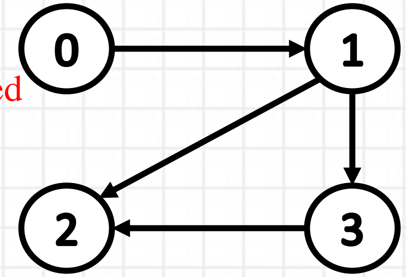


Graph Properties and Terminology Review

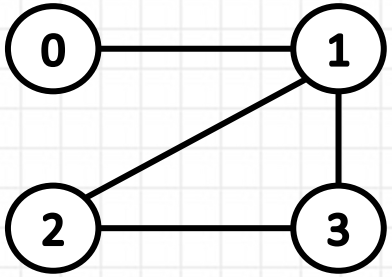
💻 Demo code time!

- Directed vs. undirected

Directed
Unweighted



Graph-1

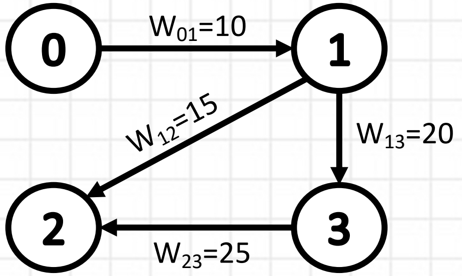


Undirected
Unweighted

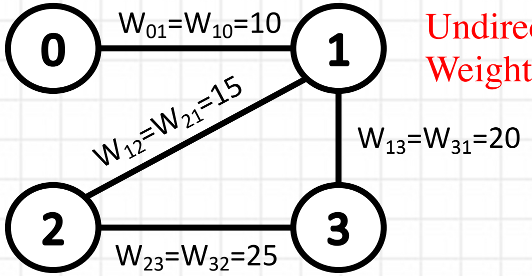
Graph-3

- Weighted vs. unweighted

Directed
Weighted



Graph-2



Undirected
Weighted

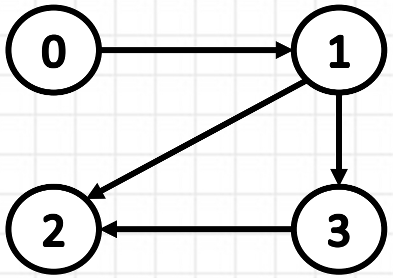
Graph-4

Graph Definition: Summary

- Two common ways to represent graphs
  - Adjacency matrix
  - Adjacency list
- Adjacency matrix
  - Space: n^2 elements for n vertices
  - Easy to check if a link exists between two vertices
- Adjacency list
  - More common representation: most large real-world graphs are sparse
  - Space: Number of edges [2*(number of edges) if undirected] + number of vertices, i.e., (m+n) or (2m+n)
  - Linked list implementation is typically used

Graph Properties and Terminology Review

- Paths and connectivity

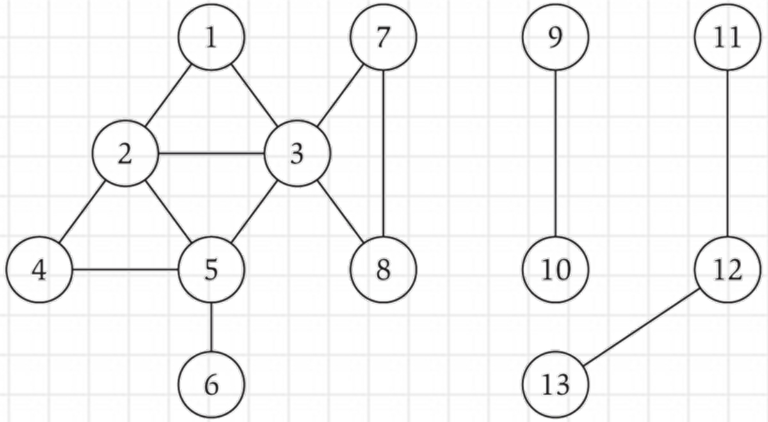


Path1: 0 → 1 → 2
Path2: 0 → 1 → 3 → 2

- Def. A **path** in a directed/undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_k$ with the property that each consecutive pair v_{i-1}, v_i is joined by a different edge in E.

- Def. A path is **simple** if all nodes are distinct.

- Def. An undirected graph is **connected** if for every pair of nodes u and v, there is a path between u and v.

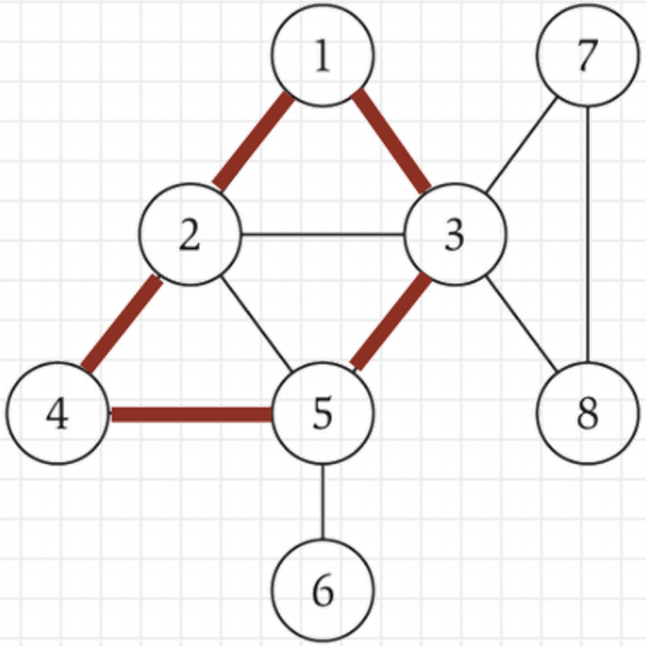


Graph Properties and Terminology Review

- Cycles

- Def. A **cycle** is a path $v_1, v_2, \ldots, v_k$ in which $v_1 = v_k$ and $k \geq 2$.

- Def. A cycle is **simple** if all nodes are distinct (except for v_1 and v_k).

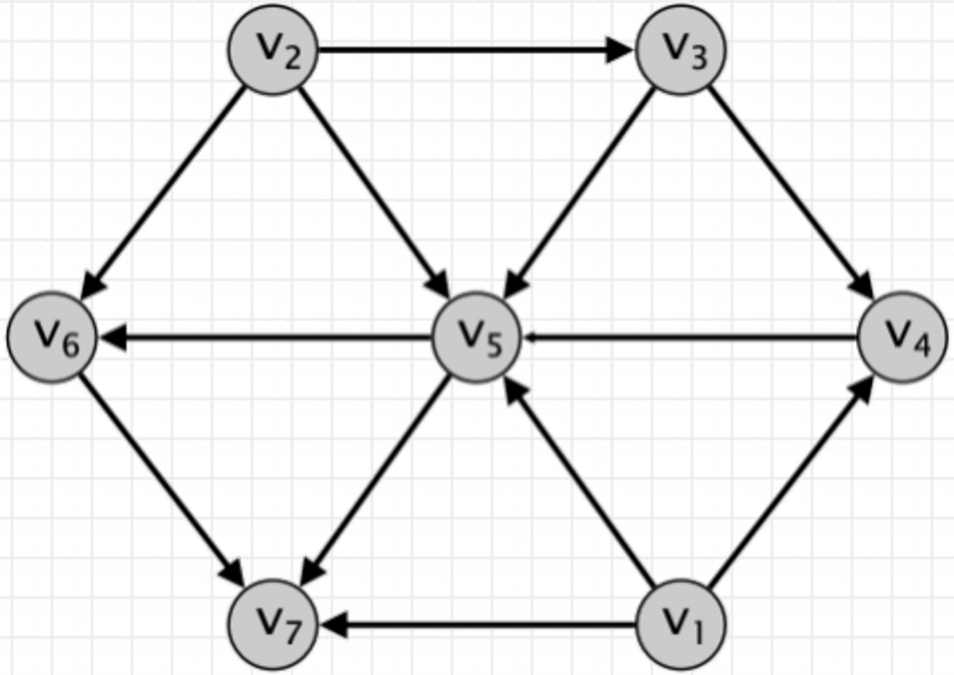


Cycle C = 1 → 2 → 4 → 5 → 3 → 1

Graph Properties and Terminology Review

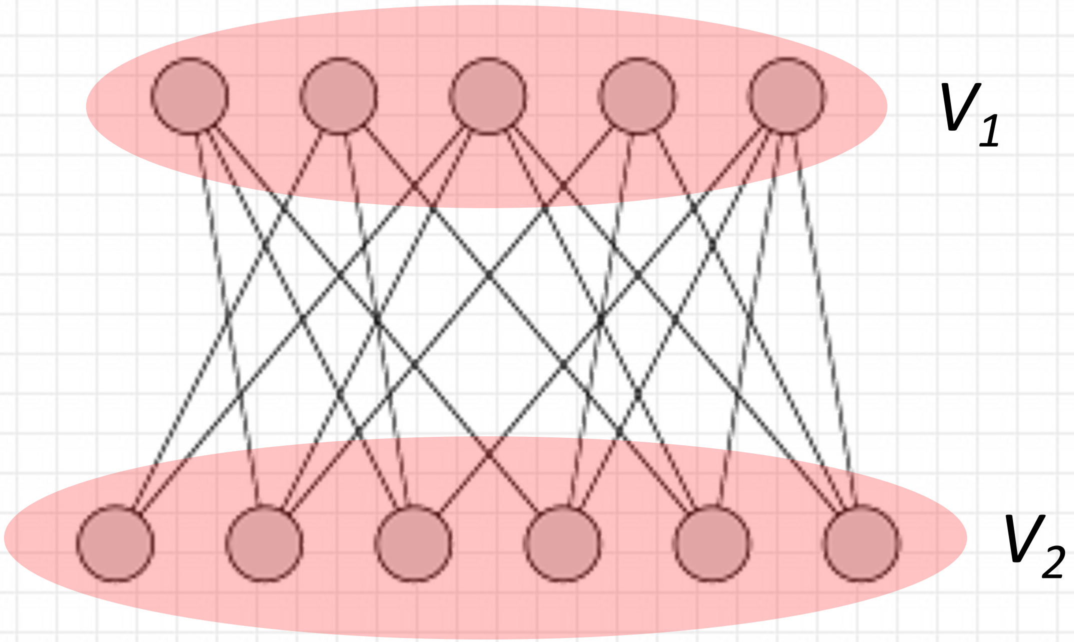
- Def. A directed acyclic graphs (DAG) is a directed graph that contains no directed cycles.

- We'll re-visit this later!



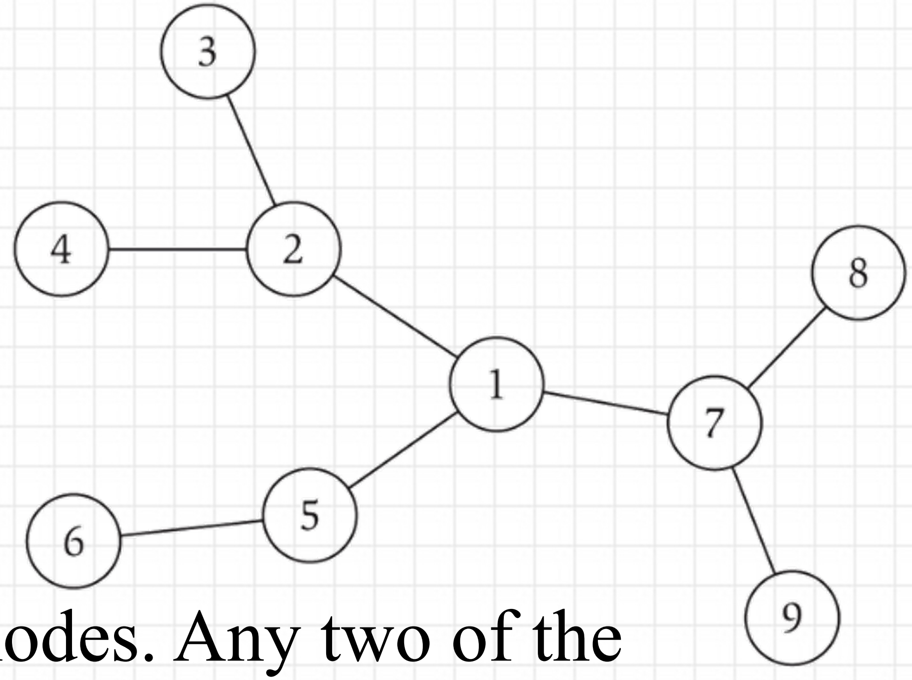
Graph Properties and Terminology Review

- Def. A bipartite graph is an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2.



Graph Properties and Terminology Review

- Trees
- Def. An undirected graph is a tree if
  - it is connected and
  - does not contain a cycle.



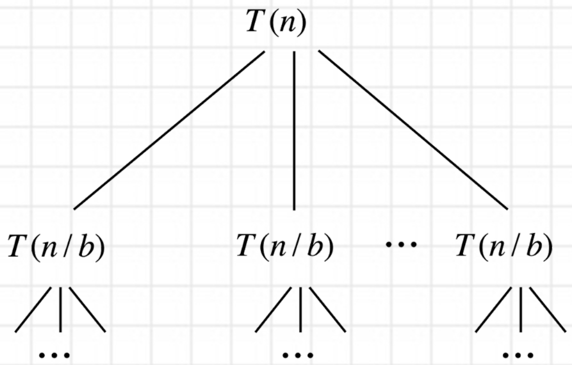
- Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third one:
  1. G is connected.
  2. G does not contain a cycle.
  3. G has $n - 1$ edges.

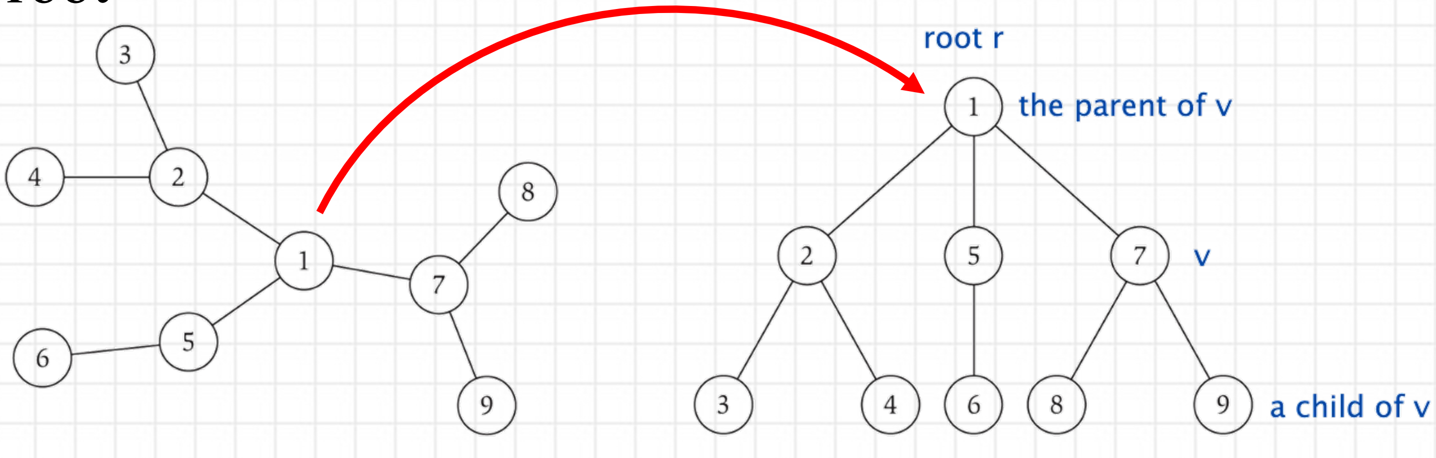
Graph Properties and Terminology Review

- Trees
- Def. An undirected graph is a tree if
  - it is connected and does not contain a cycle.
- Trees can be considered as special cases of graphs (trees ⊆ graphs)
- All graph algorithms can also be applied to trees
  - (with or without some modifications/simplifications)
- We are mostly interested in particular types of trees
  - Binary search trees ⊆ Binary trees ⊆ N-ary trees ⊆ Rooted trees
  - Recursion trees
  - Minimum spanning tree (MST)

Graph Properties and Terminology Review

- Rooted trees
- Given a tree T, choose a root node r and orient each edge away from r.
  - One vertex designated as the root





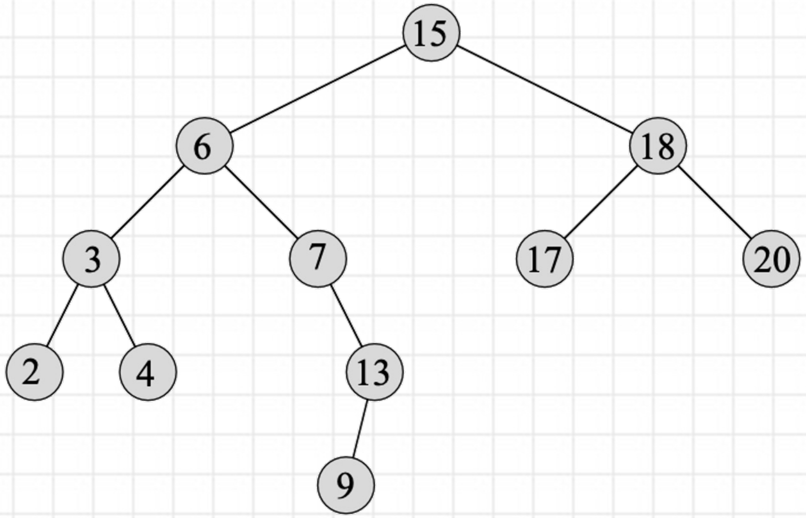
a tree

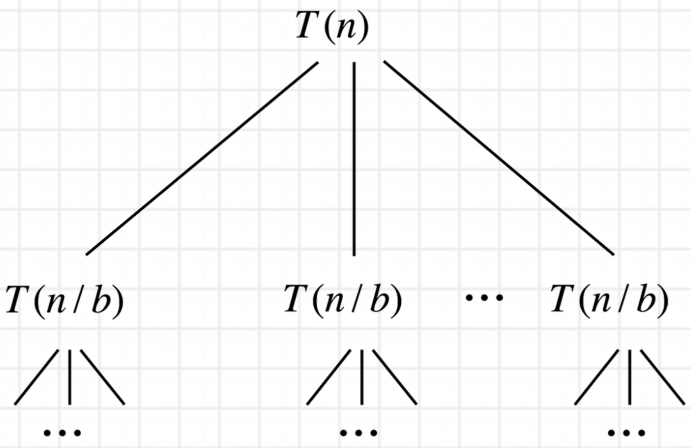
the same tree, rooted at 1

  - Ex. binary tree, binary search tree, recursion trees

Graph Properties and Terminology Review

- Rooted trees
- Given a tree T, choose a root node r and orient each edge away from r.
  - One vertex designated as the root
  - Ex. binary tree, binary search tree, recursion trees





Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list

- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components

- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

Graph Traversal

- Connectivity and Traversal

- s-t connectivity problem. Given two nodes *s* and *t*, is there a path between *s* and *t* ? (is t reachable from s?)
- s-t shortest path problem. Given two nodes *s* and *t*, what is the length of a shortest path between *s* and *t* ?
- [Strongly] connected component is a set of vertices all reachable from each other (mutually reachable)
- Connected component problem. Find all nodes reachable from *s*.

- Applications
  - Facebook, mutual friends
  - Maze traversal
  - Fewest hops in a communication network

Graph Traversal

- Traversal = Exploring = Searching
- A graph needs to be traversed in order to determine some properties

- Breadth-first search (BFS)
  - Shortest path (unweighted graphs)
  - Testing bipartiteness
  - Tree traversal (level-order)
  - Connected components
- Depth-first search (DFS)
  - Topological sorting
  - Tree traversal (in-order, pre-order, post-order)
  - Connected components

Graph Traversal

- Traversal = Exploring = Searching
- A graph needs to be traversed in order to determine some properties

- Breadth-first search (BFS)
  - Shortest path (unweighted graphs)
  - Testing bipartiteness
  - Tree traversal (level-order)
  - Connected components
- Depth-first search (DFS)
  - Topological sorting
  - Tree traversal (in-order, pre-order, post-order)
  - Connected components

	Implementation	Data Structure
BFS	Iterative	Queue (FIFO)
DFS	Recursive Iterative	(not explicitly required → execution stack) Stack (LIFO)

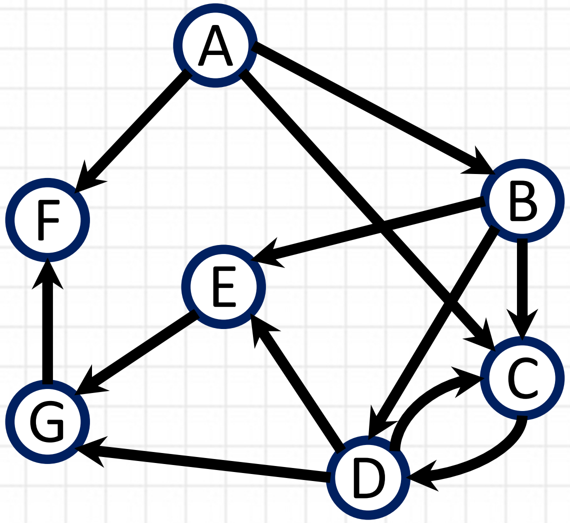
Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex "s"
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v
- <u>Iterative</u> implementation.
- Needs <u>queue</u> data structure

- Traversal = Exploring = Searching (visiting vertices one-by-one)

Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v




```
BFS(G, s)
  for each vertex u ∈ G.V − {s}
      u.color = WHITE
      u.d = ∞
      u.π = NIL
  s.color = GRAY
  s.d = 0
  s.π = NIL
  Q = ∅
  ENQUEUE(Q, s)
  while Q ≠ ∅
      u = DEQUEUE(Q)
      for each v ∈ G.Adj[u]
          if v.color == WHITE
              v.color = GRAY
              v.d = u.d + 1
              v.π = u
              ENQUEUE(Q, v)
      u.color = BLACK
```

white := unvisited node

distance from source

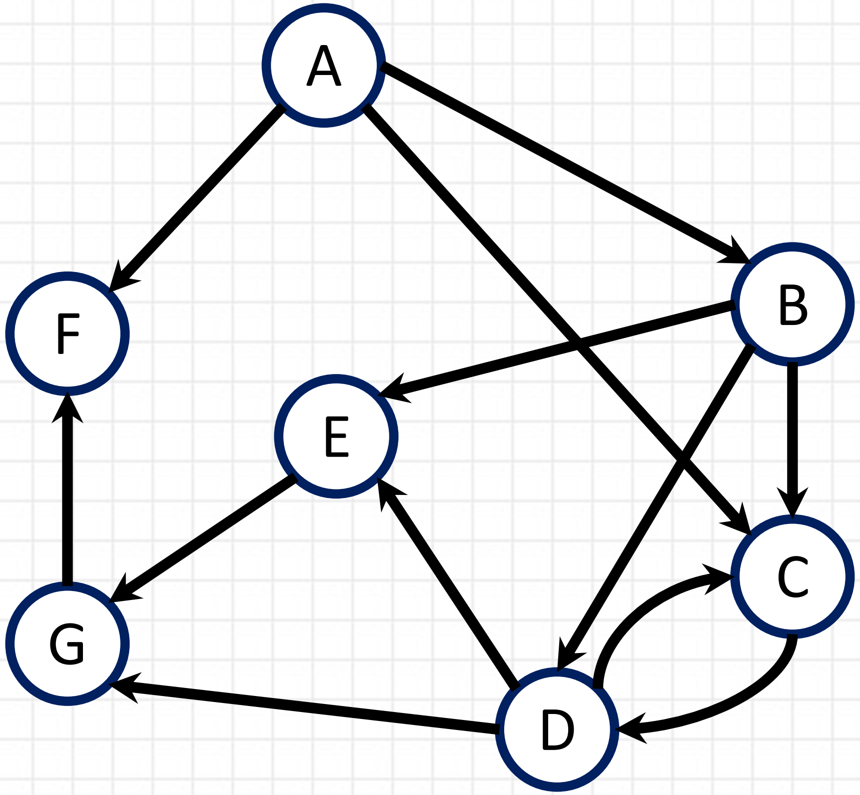
parent

gray := visited node

black := visited & all unvisited neighbors added to the queue

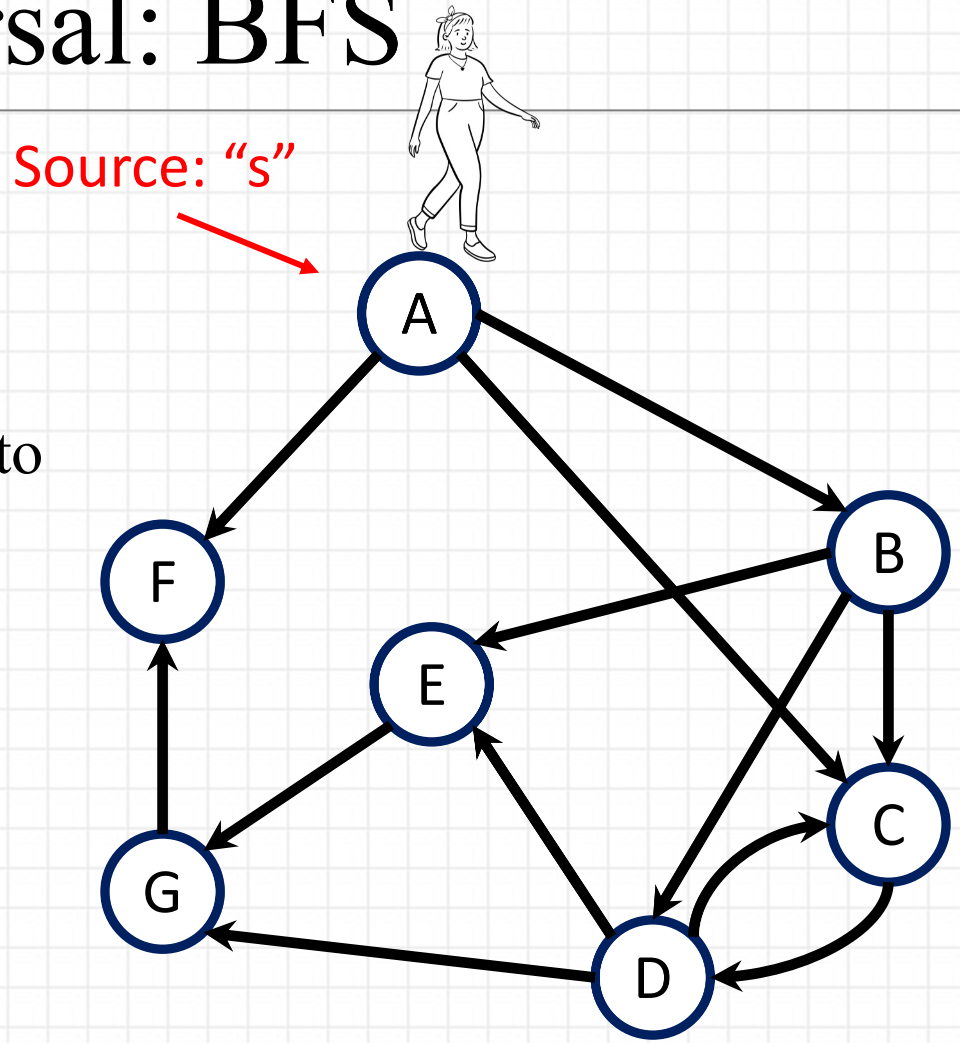
Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v



Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

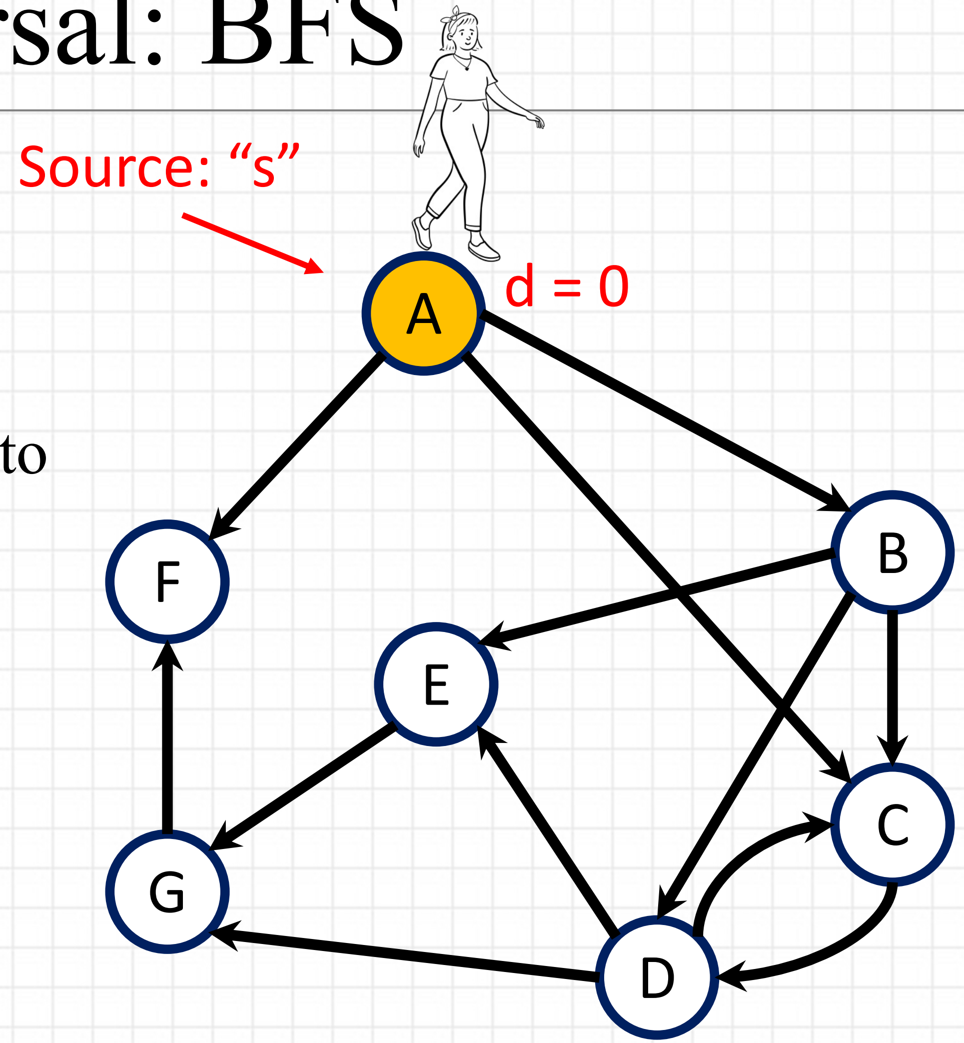


- Queue = {}
- Visited = {}

Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

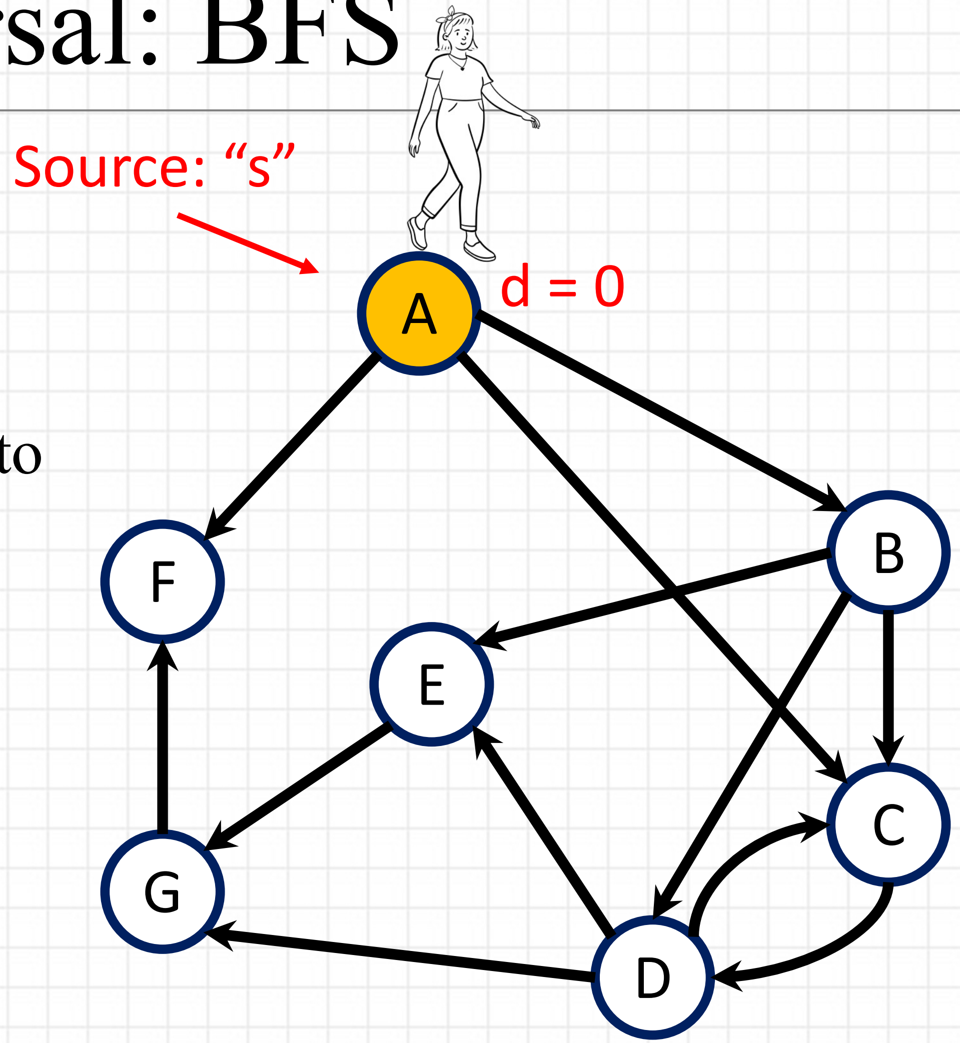
- Queue = {A}
- Visited = {}



Graph Traversal: BFS

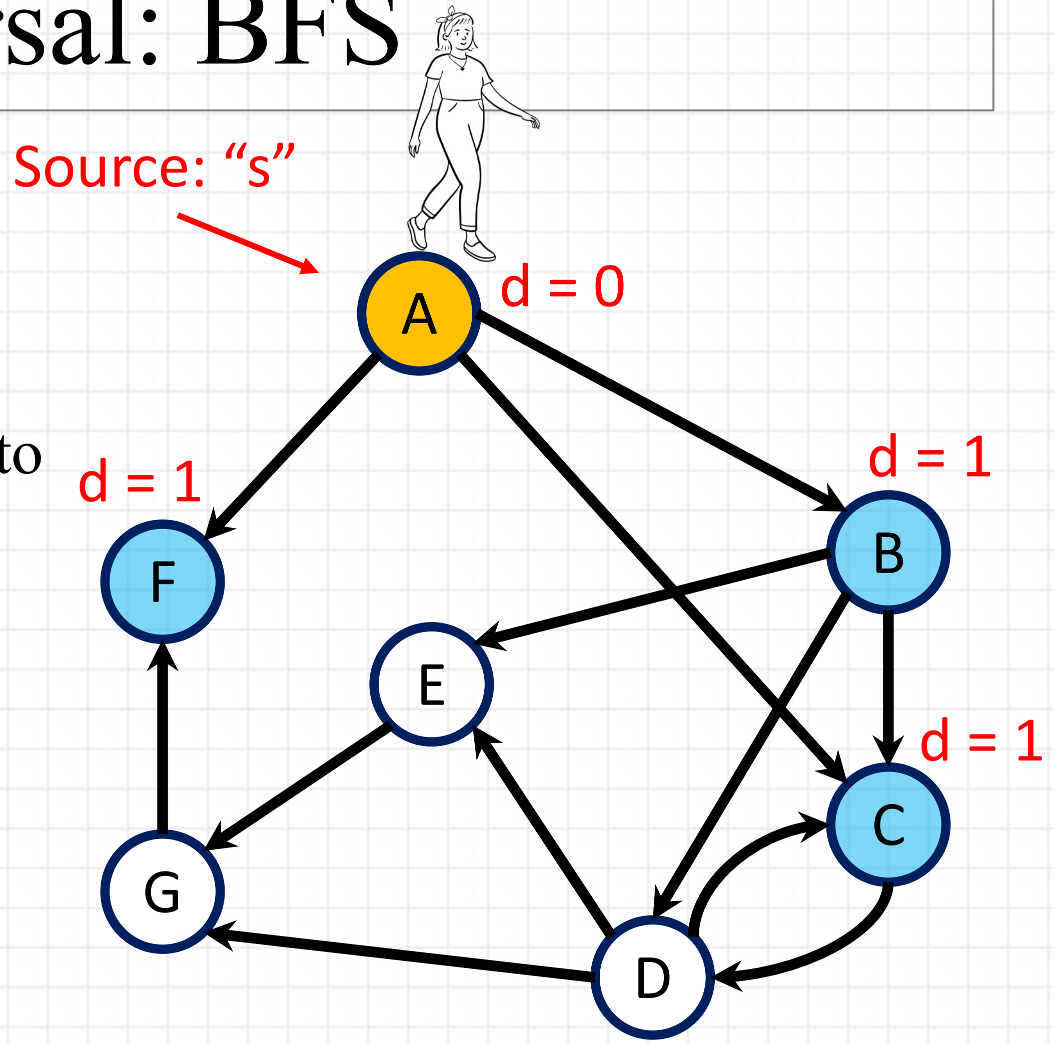
- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

- Queue = {A}
- Visited = {A}



Graph Traversal: BFS

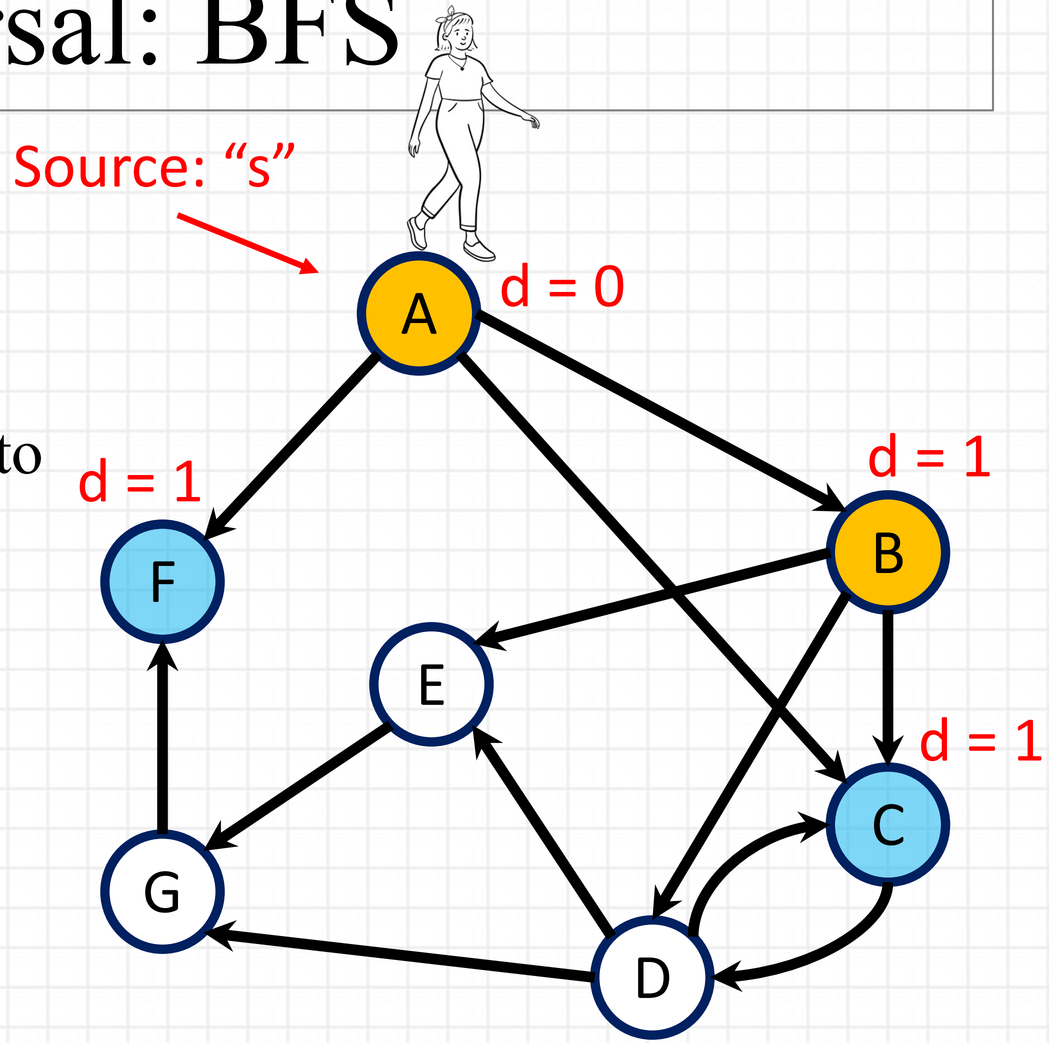
- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v



- Queue = {~~A~~, B, C, F}
- Visited = {A, B, C, F}

Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex "s"
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

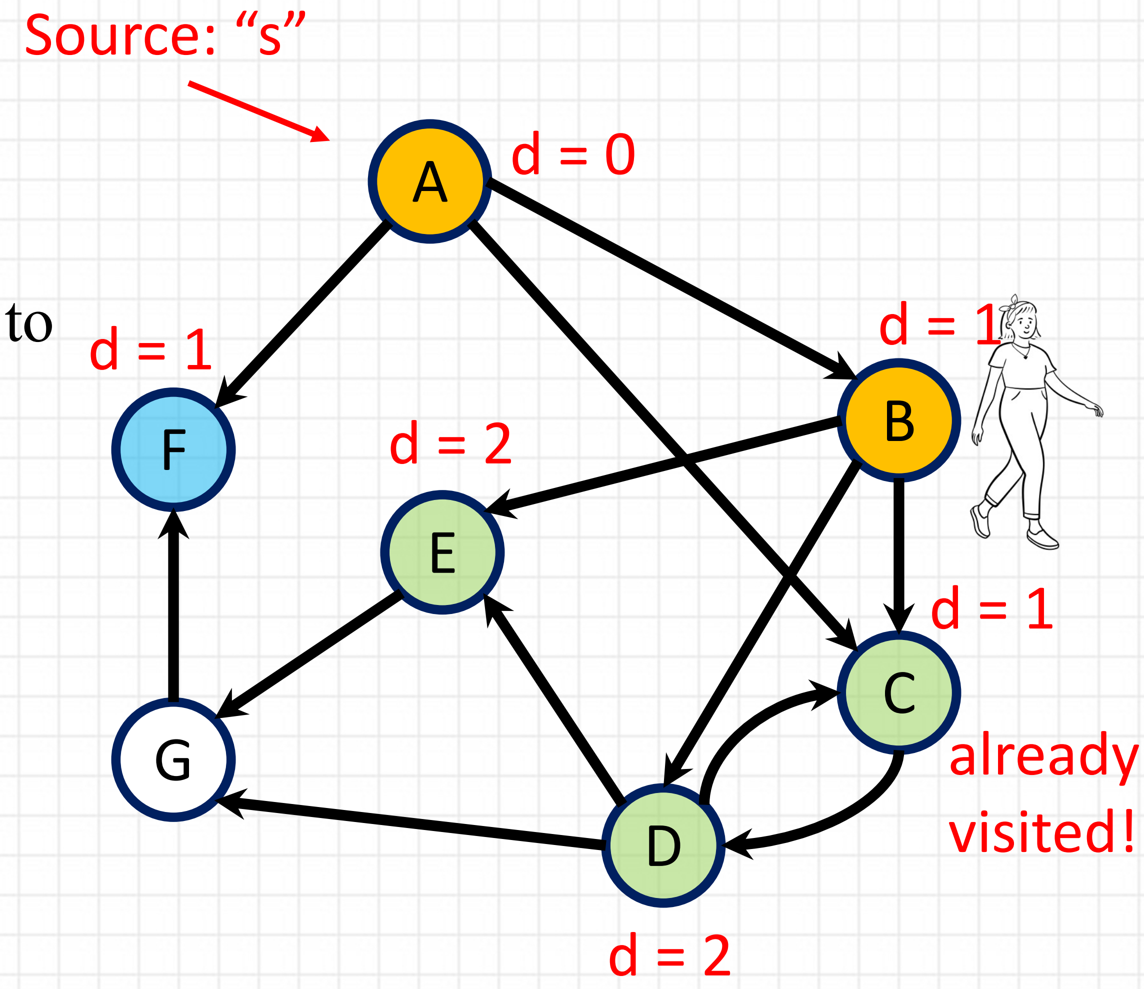


- Queue = {~~A~~, ~~B~~, C, F}
- Visited = {A, B, C, F}

Graph Traversal: BFS

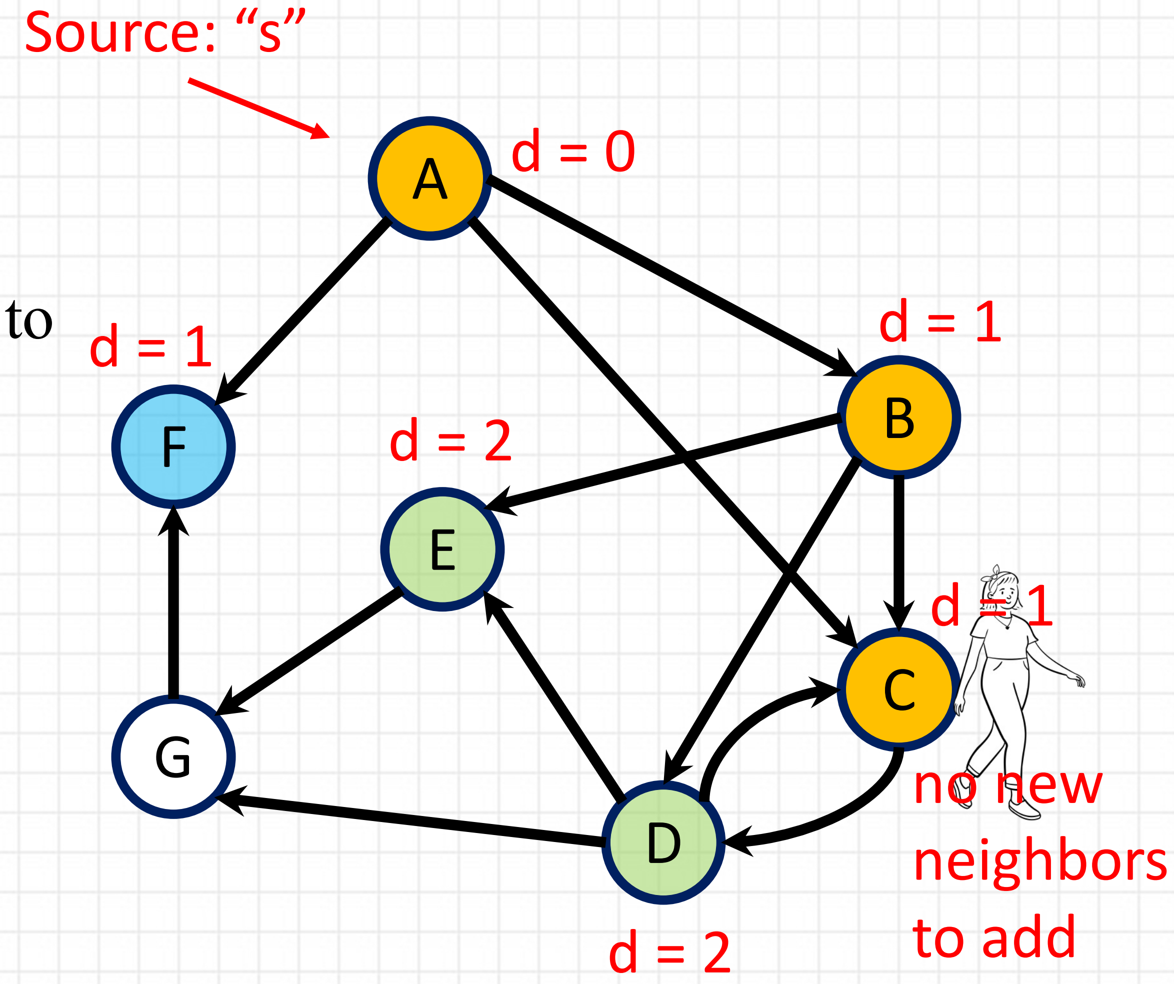
- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

- Queue = {~~A~~, ~~B~~, C, F, D, E}
- Visited = {A, B, C, F, D, E}



Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

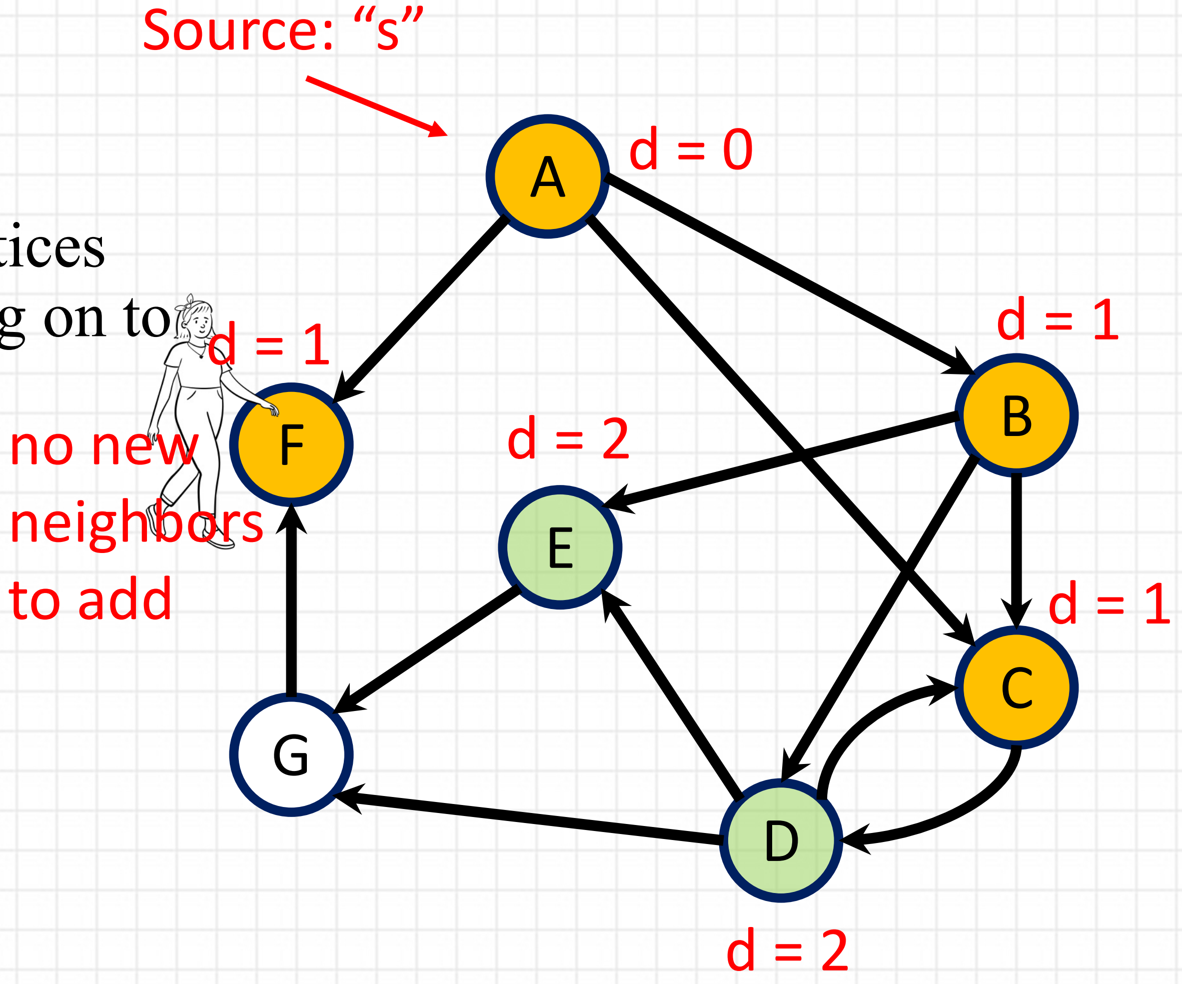


- Queue = {~~A~~, ~~B~~, ~~C~~, F, D, E}
- Visited = {A, B, C, F, D, E}

Graph Traversal: BFS

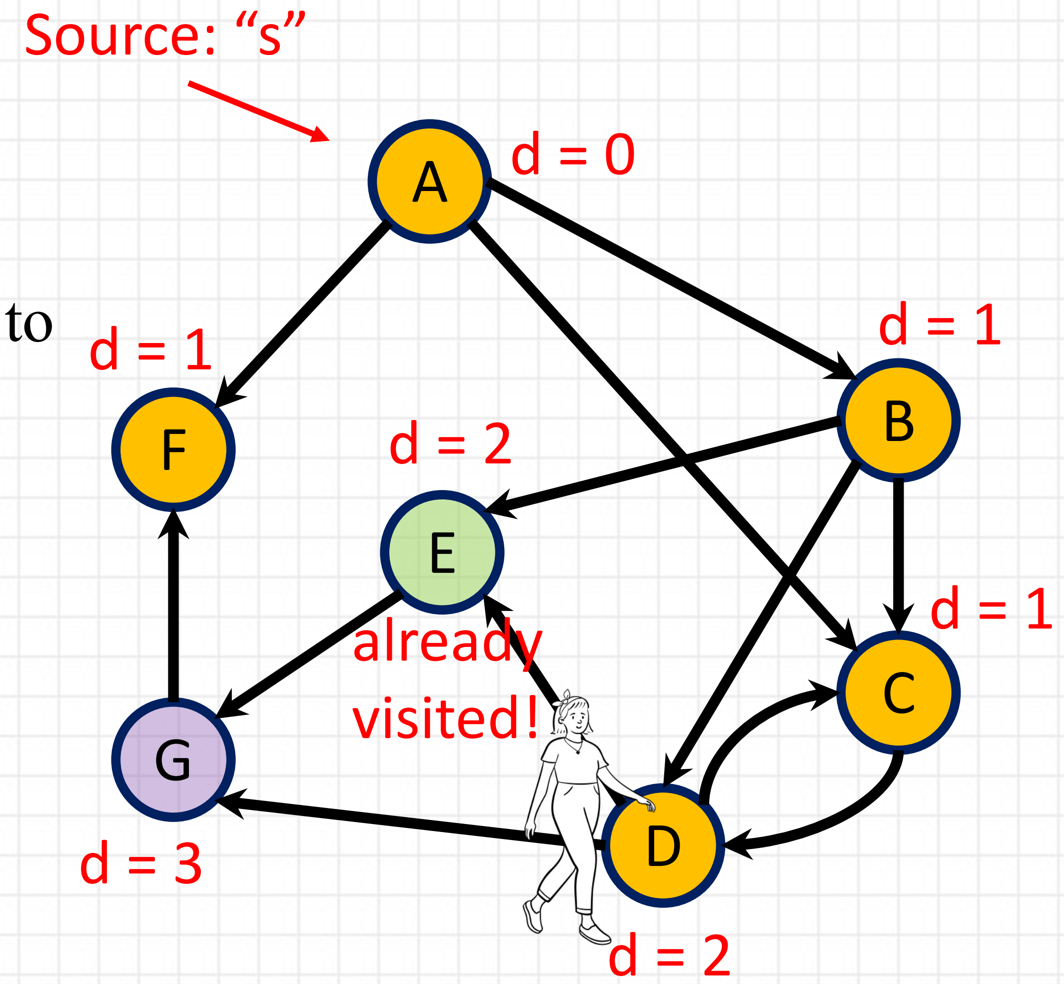
- An efficient graph traversal procedure
- BFS starts from a source vertex "s"
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

- Queue = {~~A~~, ~~B~~, ~~C~~, ~~F~~, D, E}
- Visited = {A, B, C, F, D, E}



Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex "s"
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

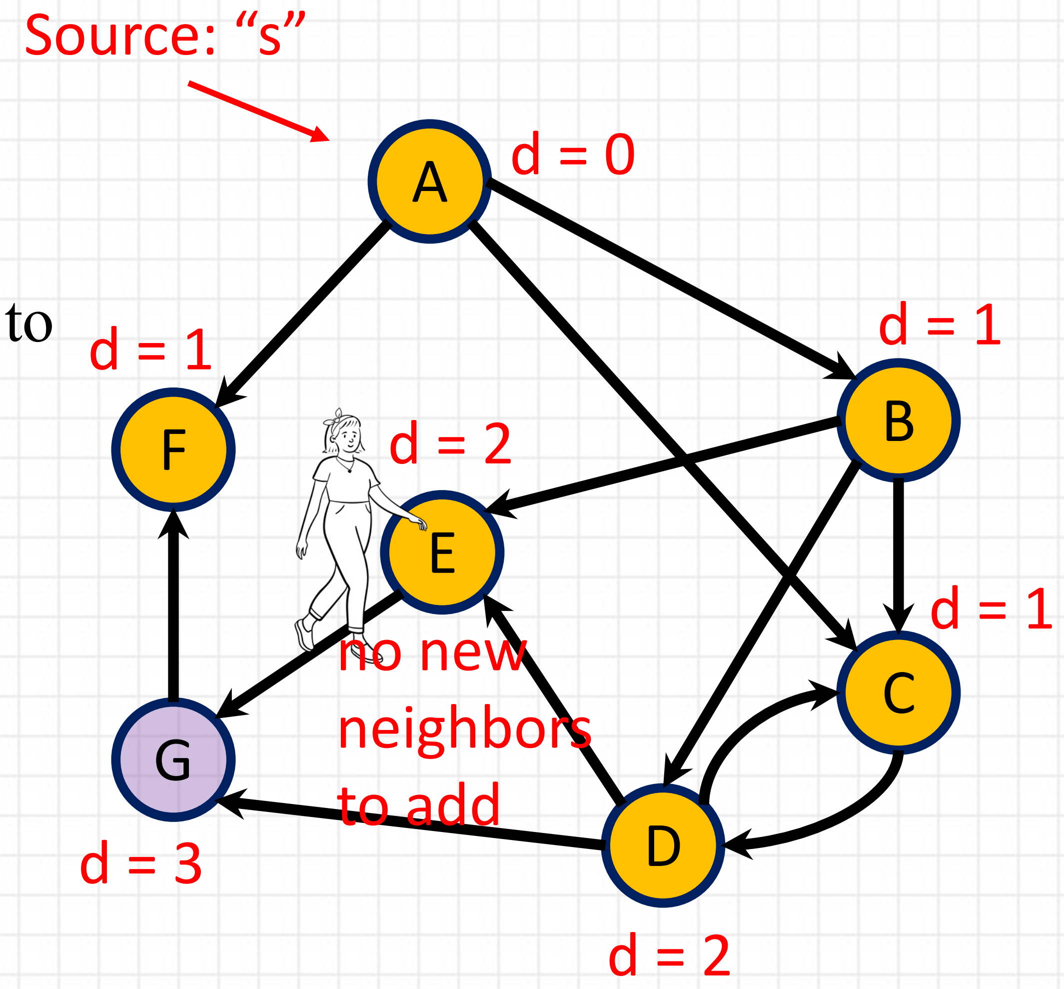


- Queue = {~~A~~, ~~B~~, ~~C~~, ~~F~~, ~~D~~, E, G}
- Visited = {A, B, C, F, D, E, G}

Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

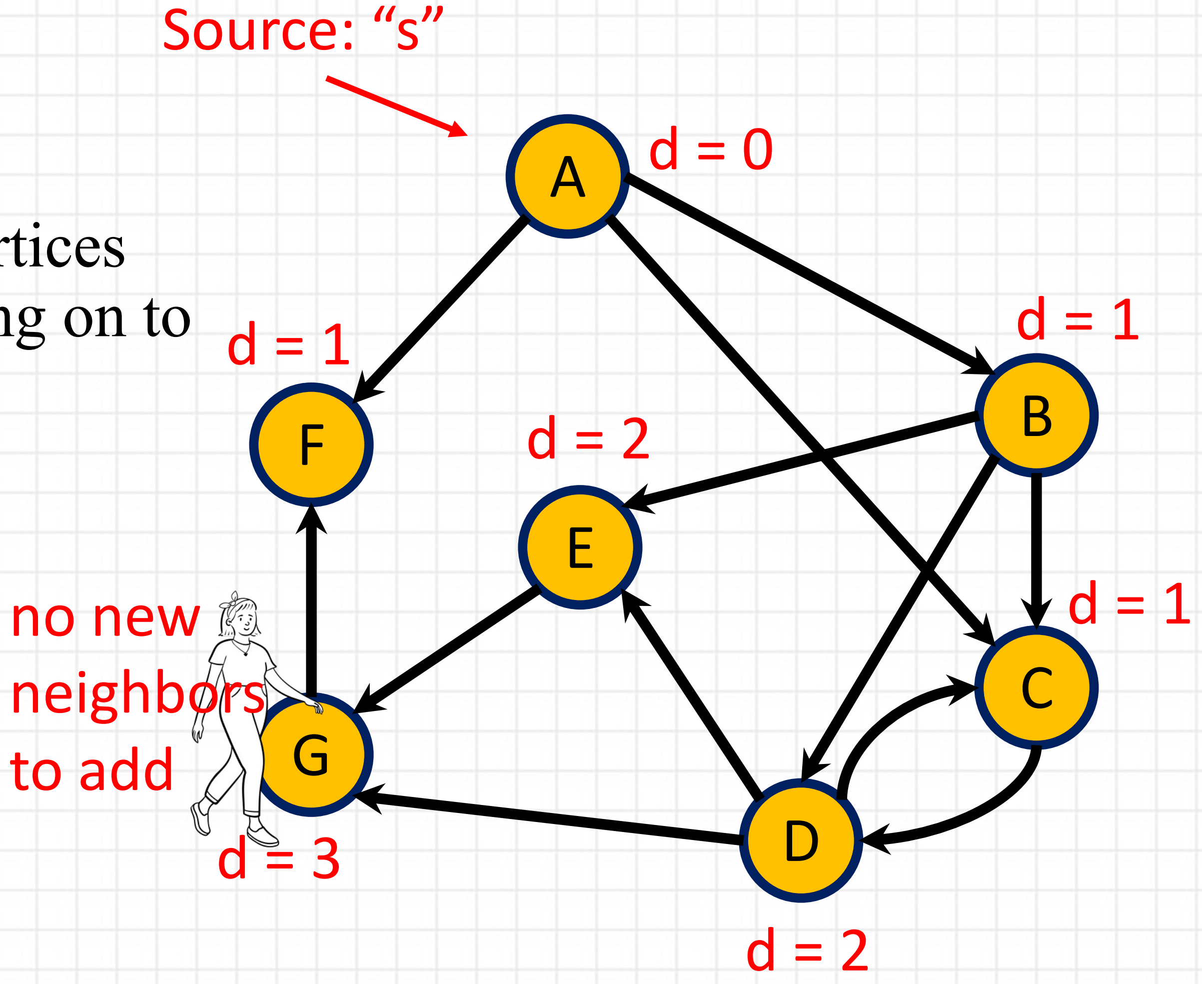
- Queue = {~~A~~, ~~B~~, ~~C~~, ~~F~~, ~~D~~, ~~E~~, G}
- Visited = {A, B, C, F, D, E, G}



Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex "s"
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

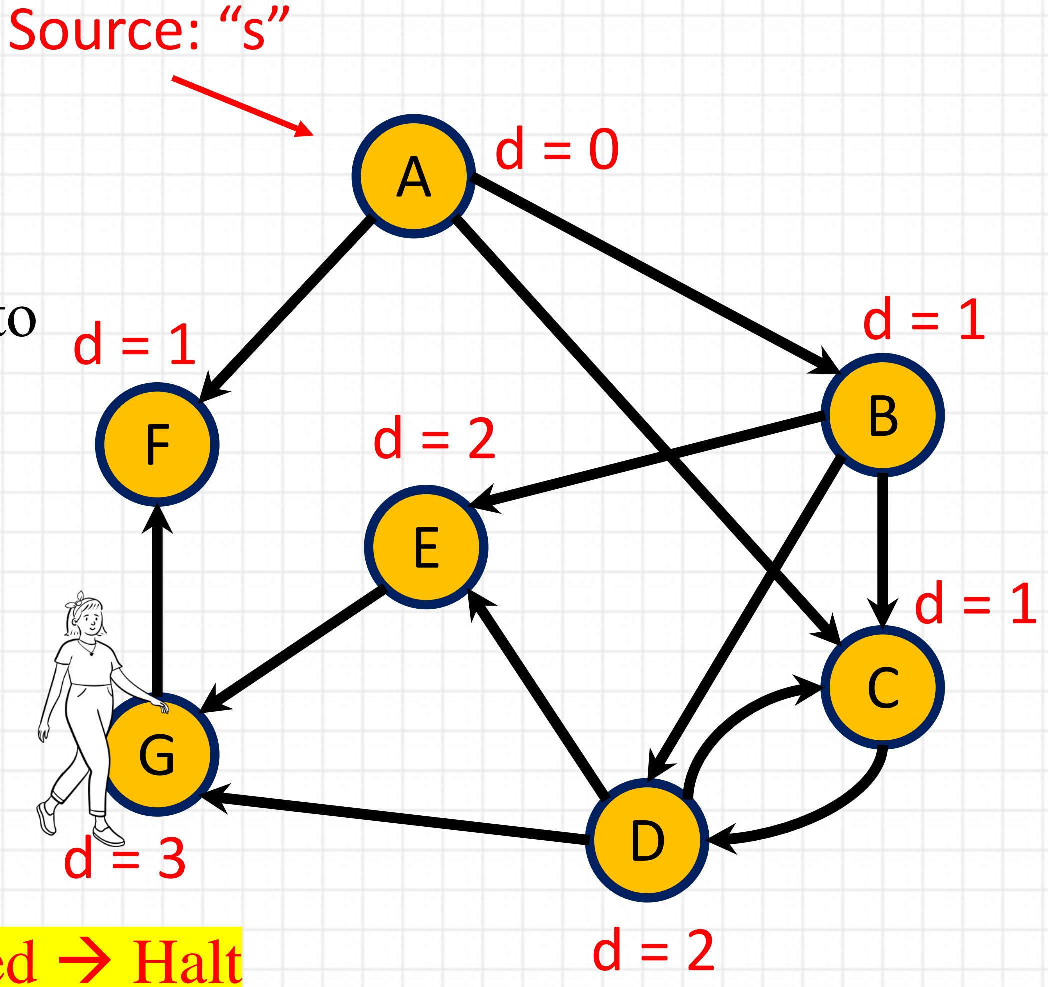
- Queue = {~~A~~, ~~B~~, ~~C~~, ~~F~~, ~~D~~, ~~E~~, ~~G~~}
- Visited = {A, B, C, F, D, E, G}



Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

- Queue = {~~A~~, ~~B~~, ~~C~~, ~~F~~, ~~D~~, ~~E~~, ~~G~~}
- Visited = {A, B, C, F, D, E, G}



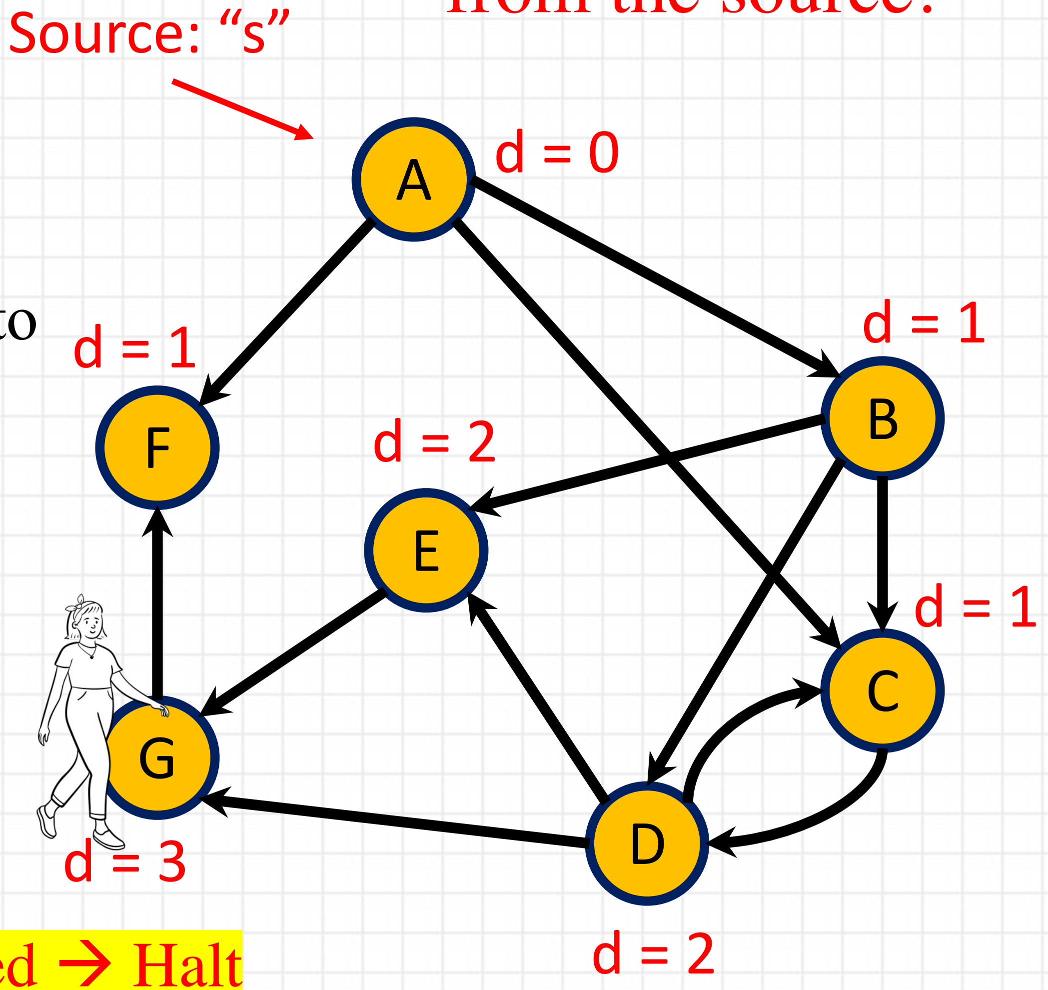
Nothing left in the queue → All nodes are visited → Halt

Graph Traversal: BFS

Note d always presents the "shortest distance" from the source!

- An efficient graph traversal procedure
- BFS starts from a source vertex "s"
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

- Queue = {~~A~~, ~~B~~, ~~C~~, ~~F~~, ~~D~~, ~~E~~, ~~G~~}
- Visited = {A, B, C, F, D, E, G}

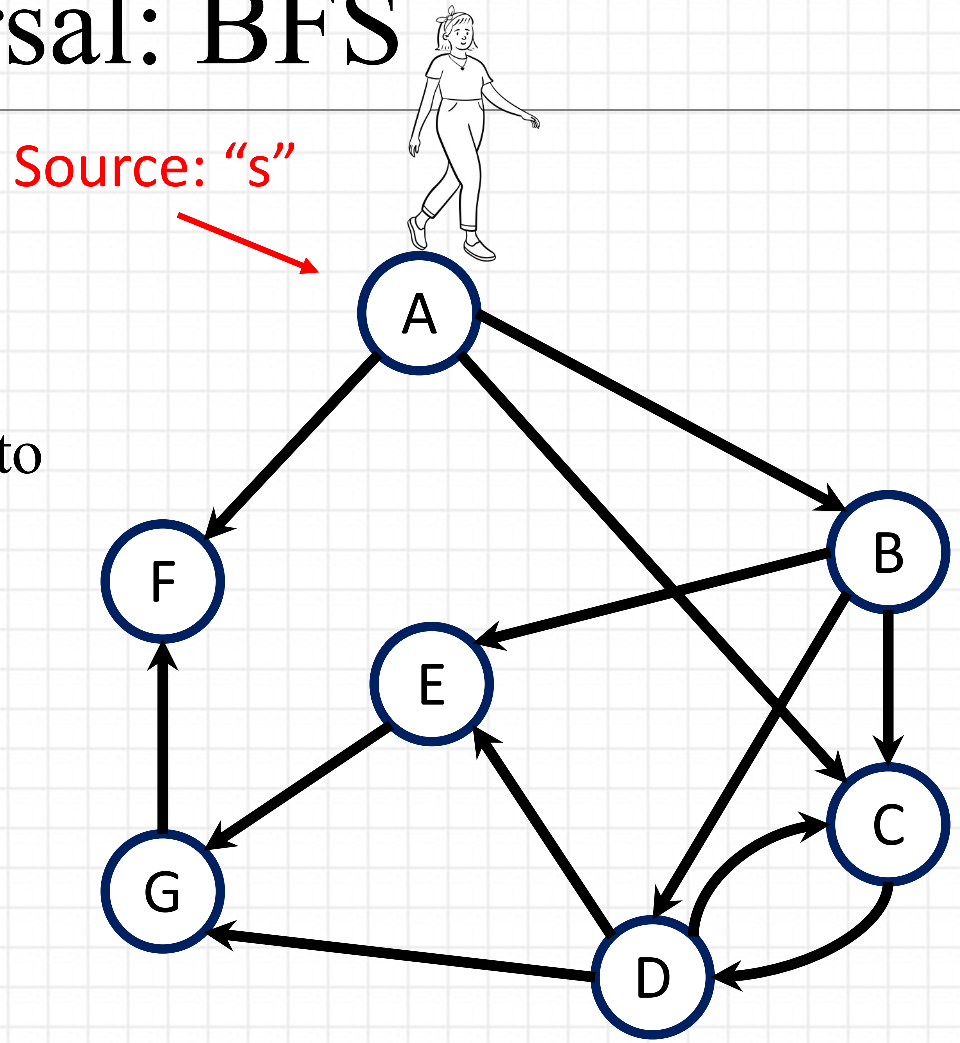


Nothing left in the queue → All nodes are visited → Halt

Graph Traversal: BFS

- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

- Queue = {}
- Visited = {}



Demo code time!

Graph Traversal: BFS

- BFS runs in O(|V| + |E|) time
- The worst case is when the graph is connected.
  - Each vertex is added to the queue and removed from it <u>exactly once</u>
  - Each adjacency list is used <u>exactly once</u>

Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path.
- No explicit storage of vertices is required (BFS needs a queue)
- However, calls for each vertex build up on the execution stack (recursive implementation)
- An iterative implementation is possible using an explicit stack data structure.

- Traversal = Exploring = Searching (visiting vertices one-by-one)

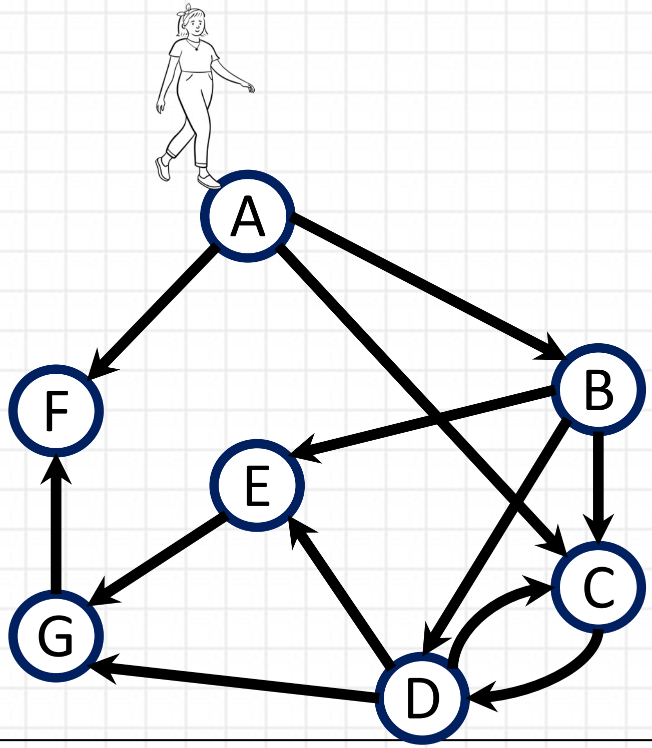
A Note about Recursive Algorithms

- In general, recursive algorithms can be used in various setups:
  - Backtracking
    - Ex. Enumerating all subsets of a given set or array
    - Usually (not always!), in these cases we can expect an exponential runtime $O(a^n)$, where a is the number of possible options to choose at each step which is equal to the number branches after each node in the recursion tree.
  - Divide-and-Conquer (D&C)
  - Dynamic programming (DP)
  - Traversing a graph or tree using the depth-first search (DFS) approach

Do you remember this slide?

Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

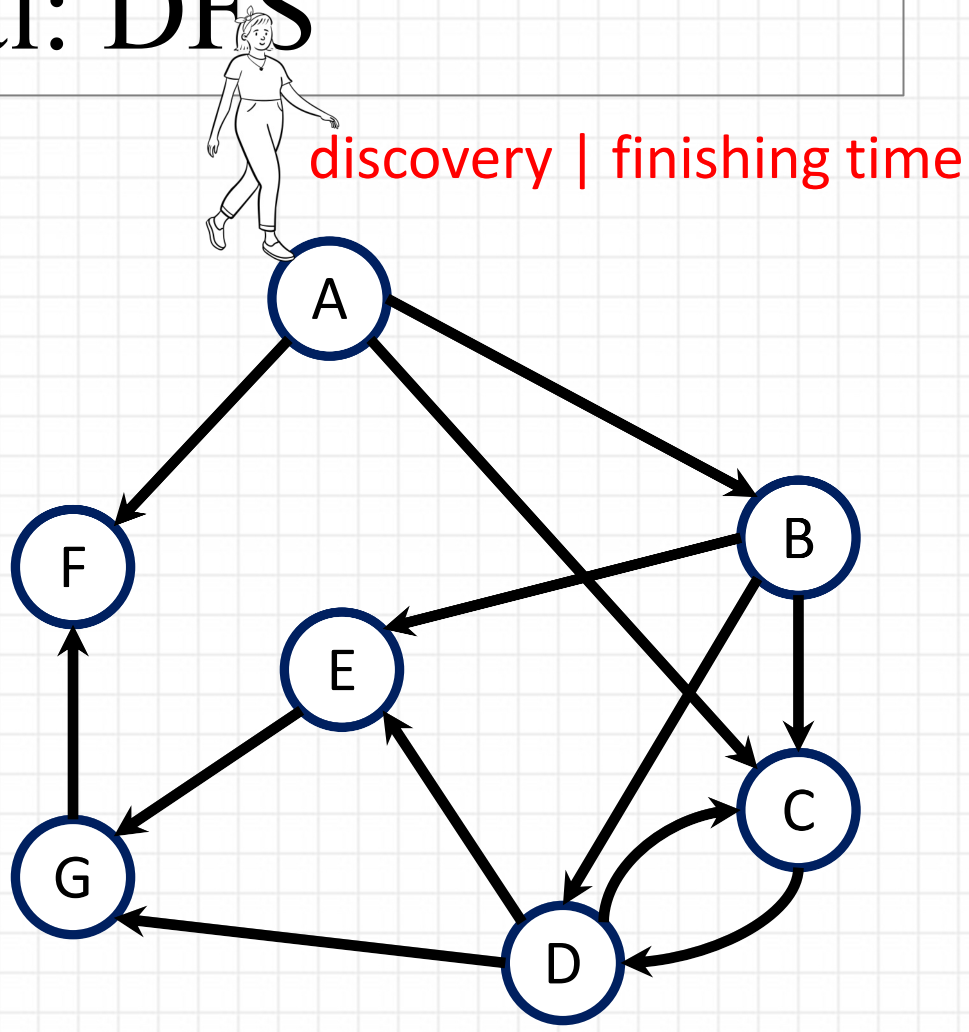



```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1                  // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]            // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                  // blacken u; it is finished
time = time + 1
u.f = time
```

Graph Traversal: DFS

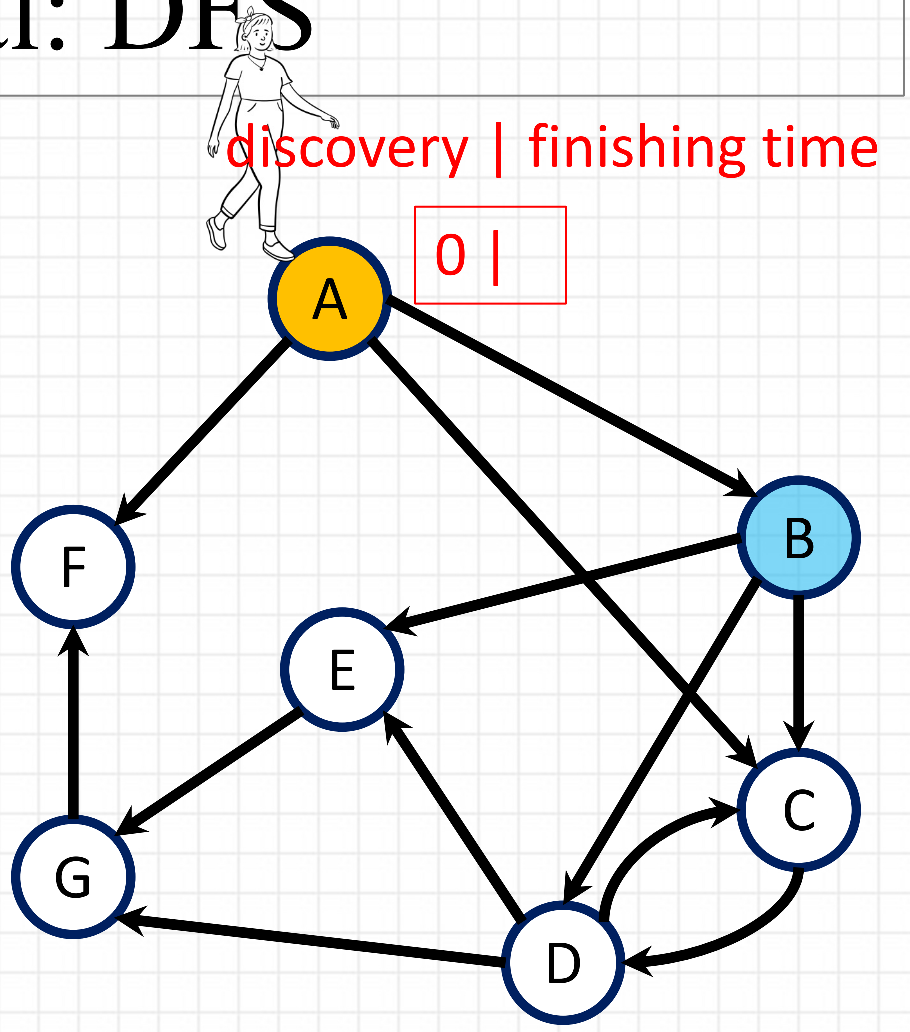
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



- Stack = {A}
- Visited = {A}

Graph Traversal: DFS

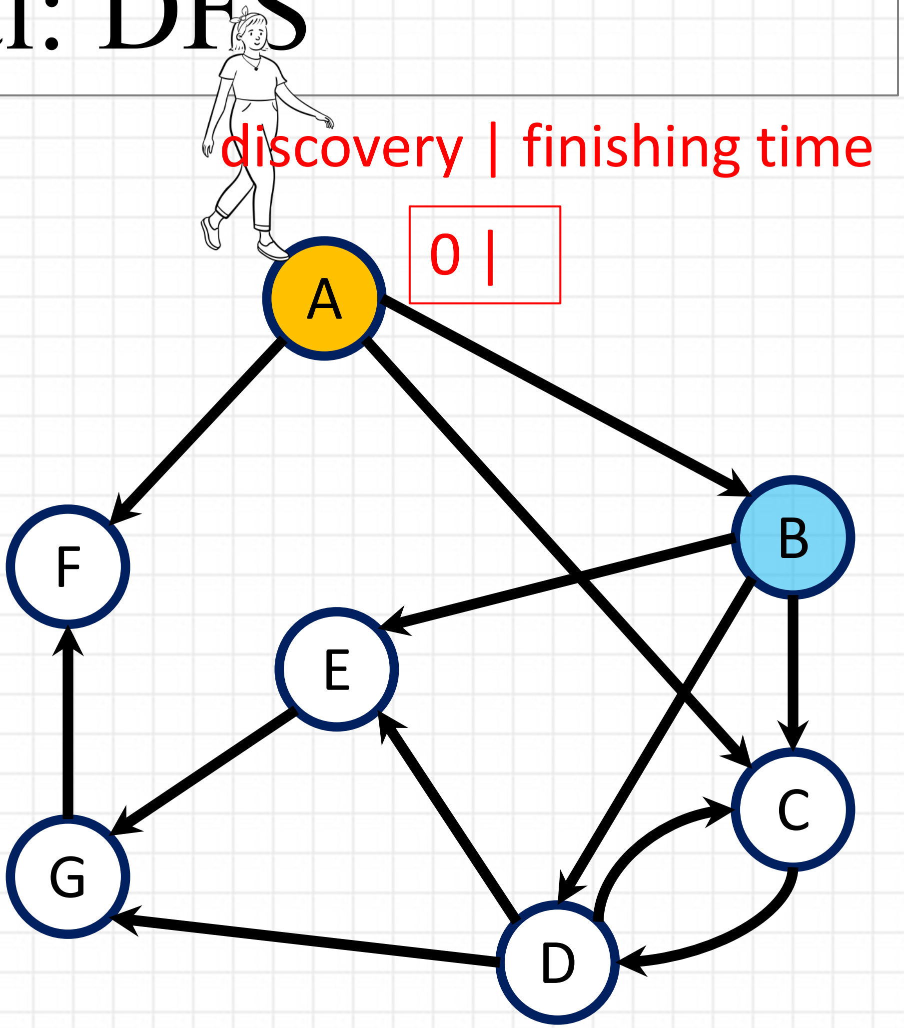
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



- Stack = {A, B}
- Visited = {A}

Graph Traversal: DFS

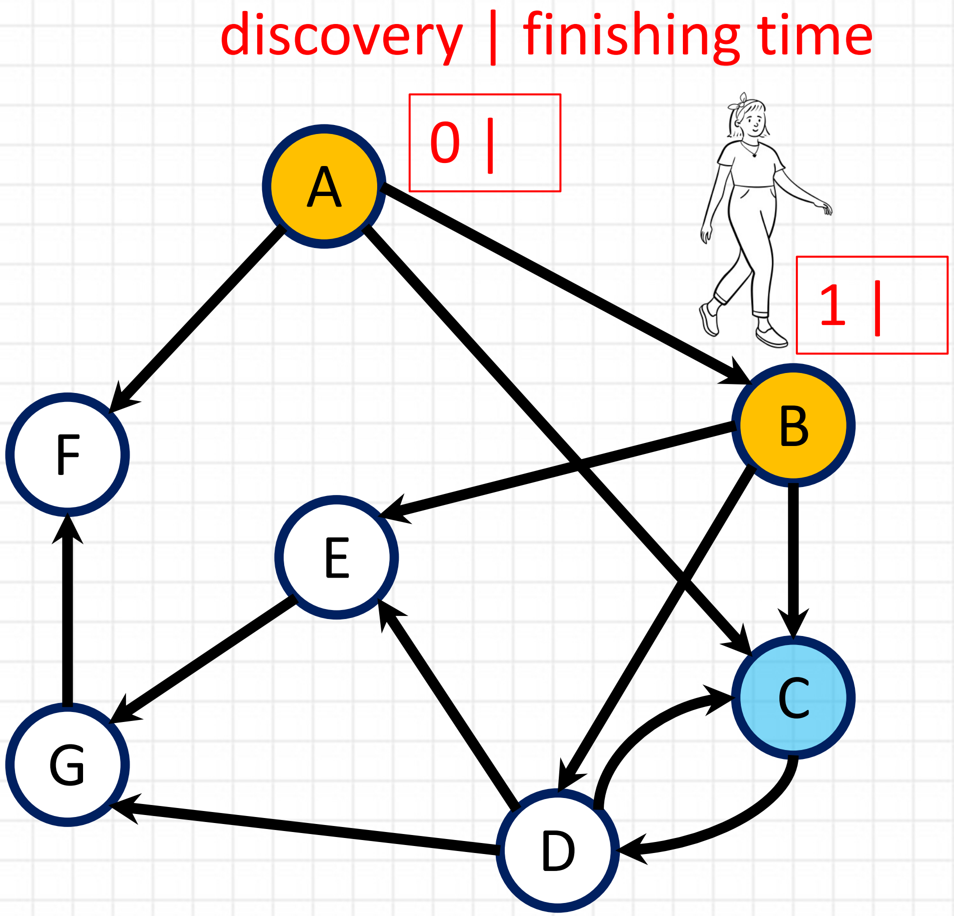
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



- Stack = {A, B}
- Visited = {A}

Graph Traversal: DFS

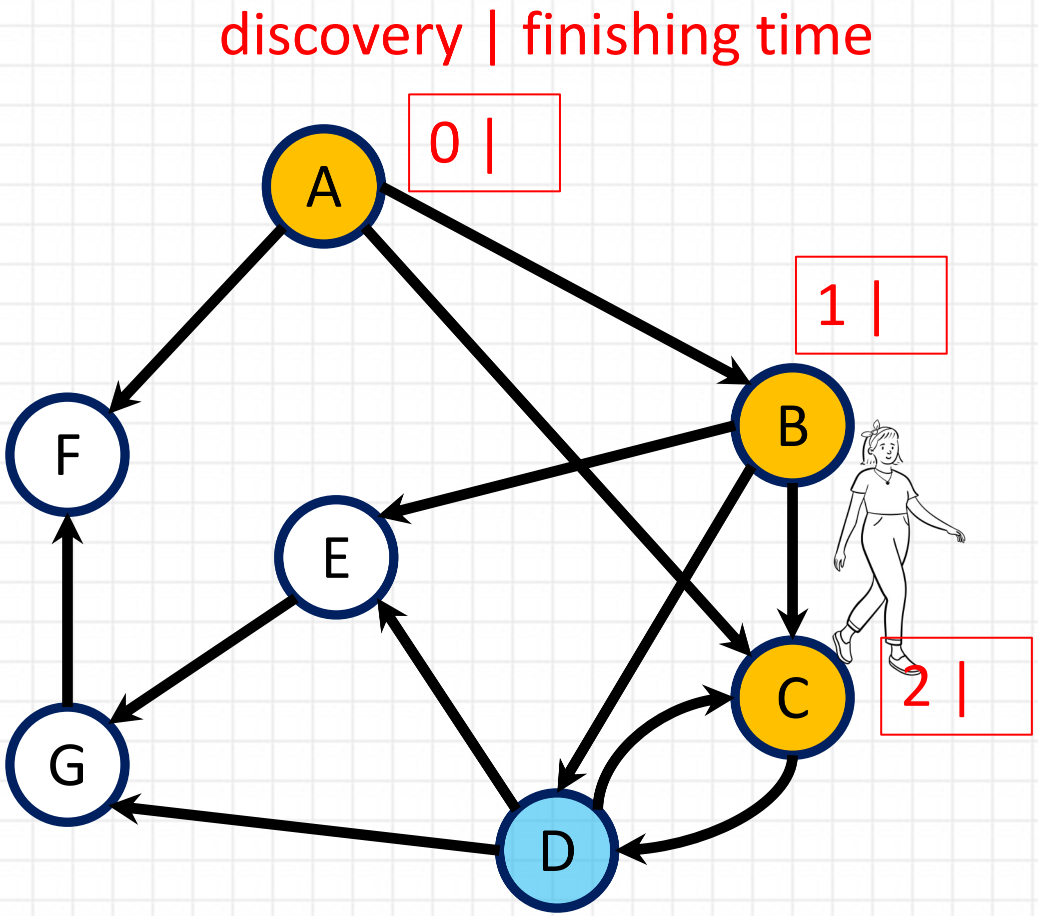
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



- Stack = {A, B, C}
- Visited = {A, B}

Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

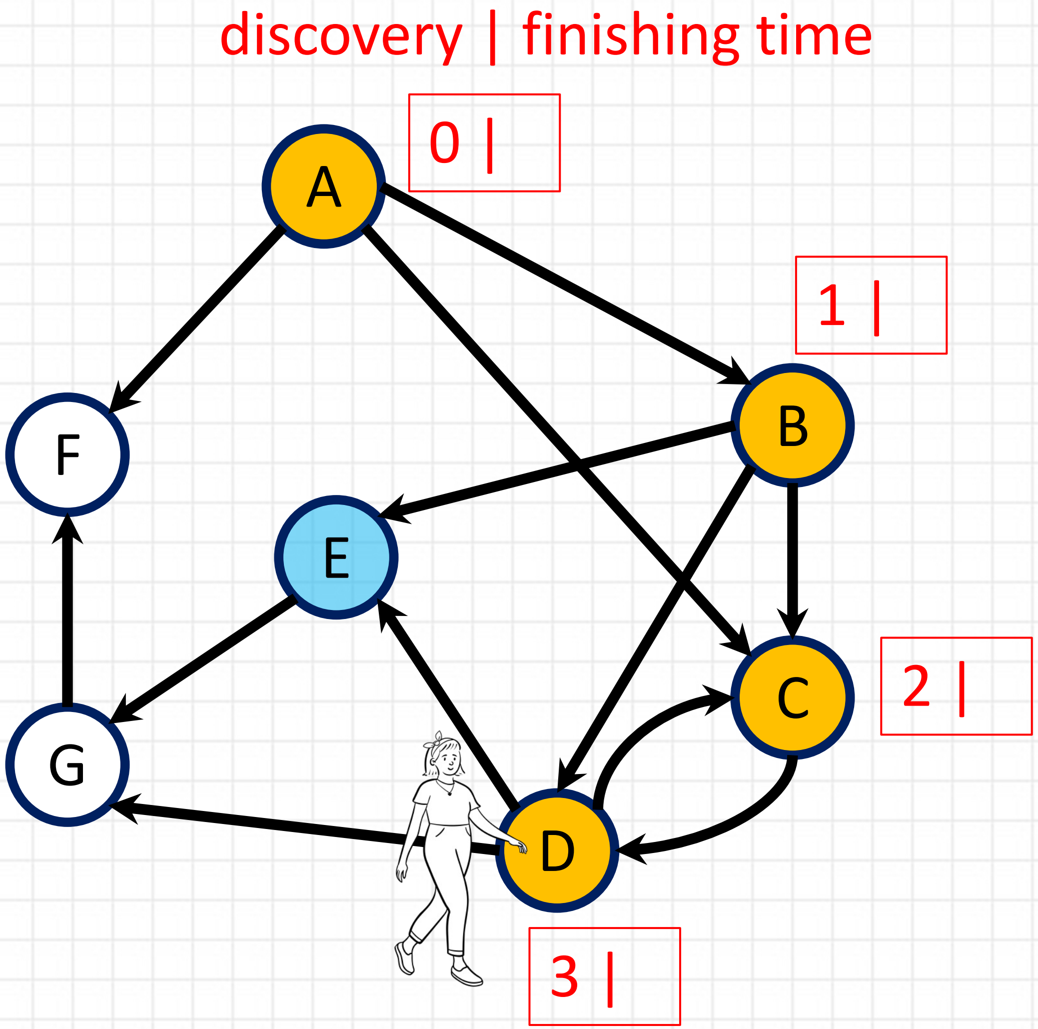


- Stack = {A, B, C, D}
- Visited = {A, B, C}

Graph Traversal: DFS

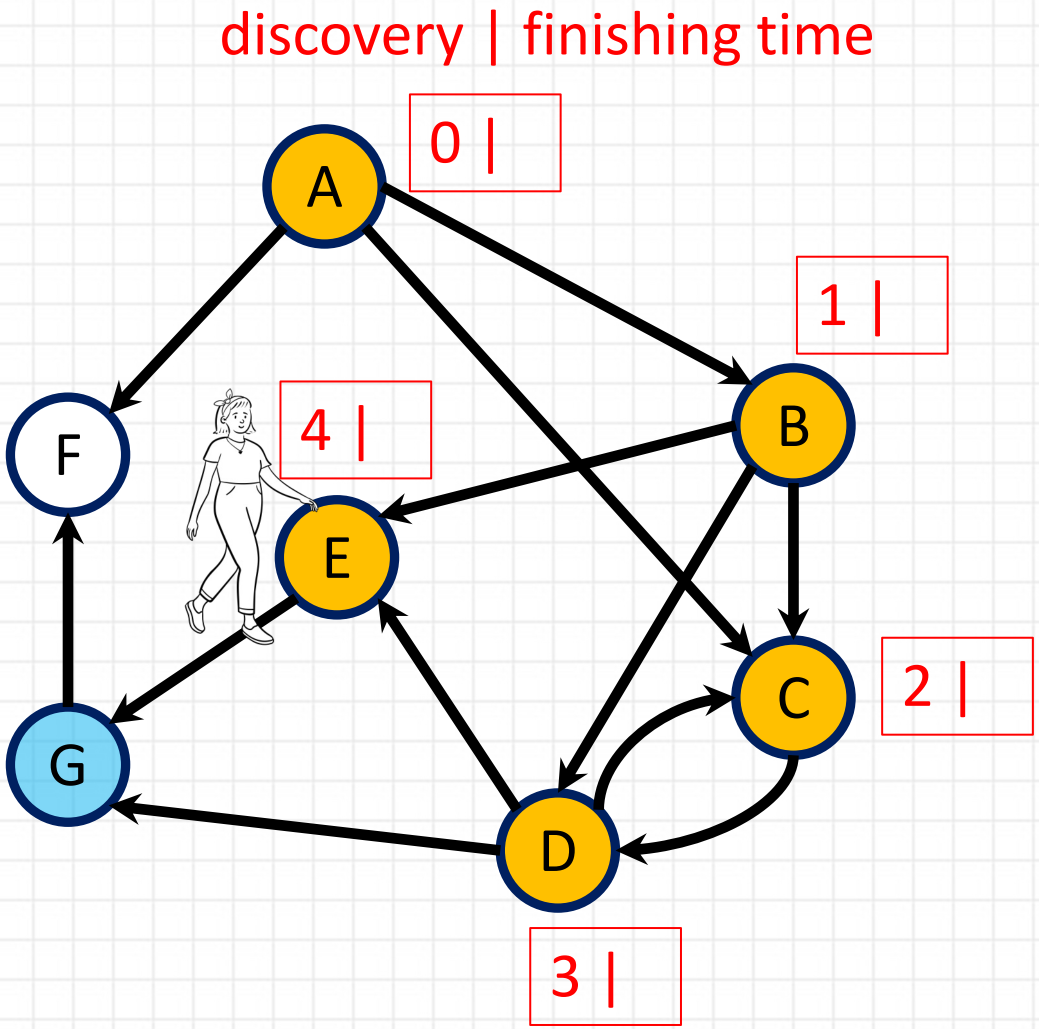
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

- Stack = {A, B, C, D, E}
- Visited = {A, B, C, D}



Graph Traversal: DFS

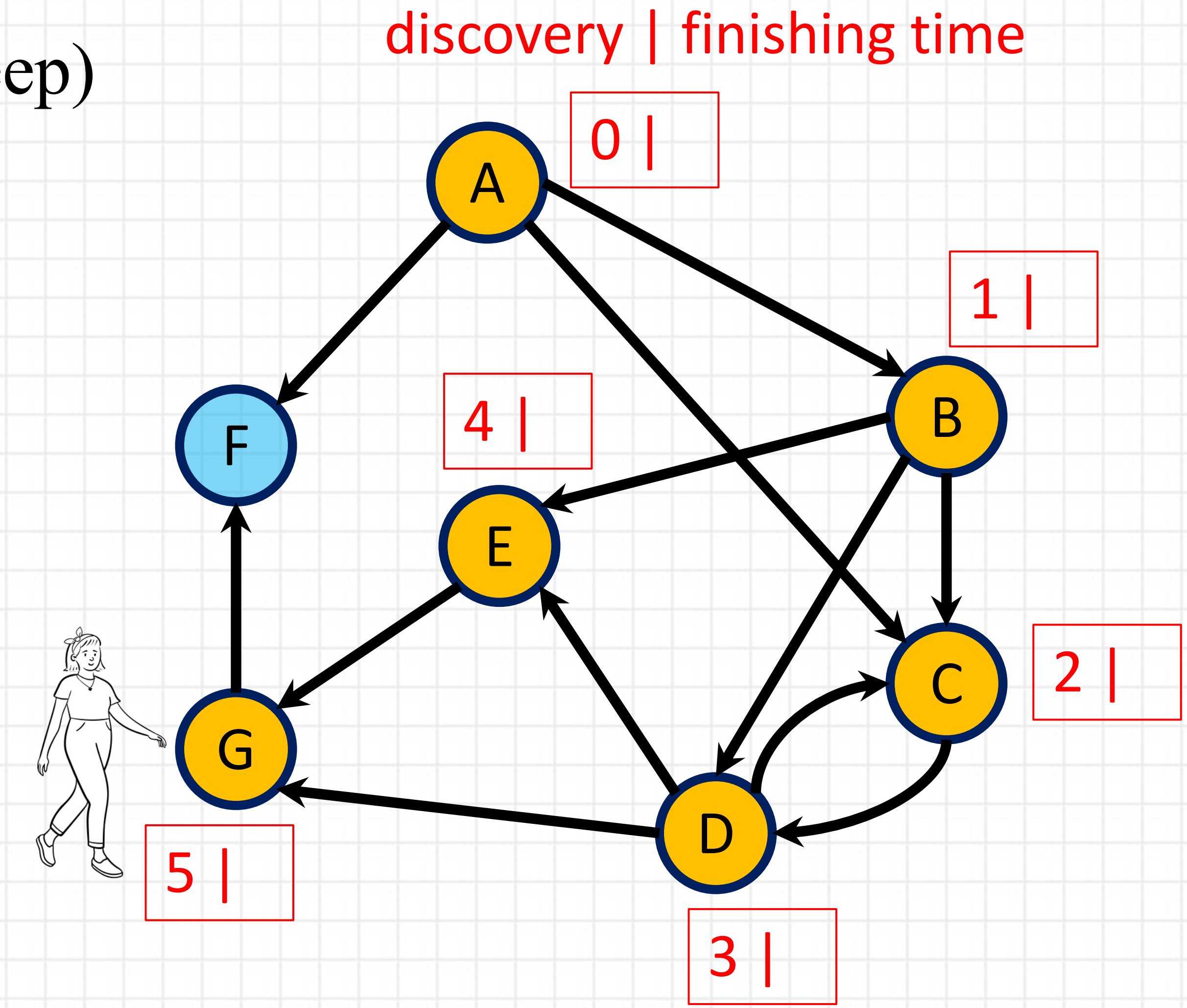
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



- Stack = {A, B, C, D, E, G}
- Visited = {A, B, C, D, E}

Graph Traversal: DFS

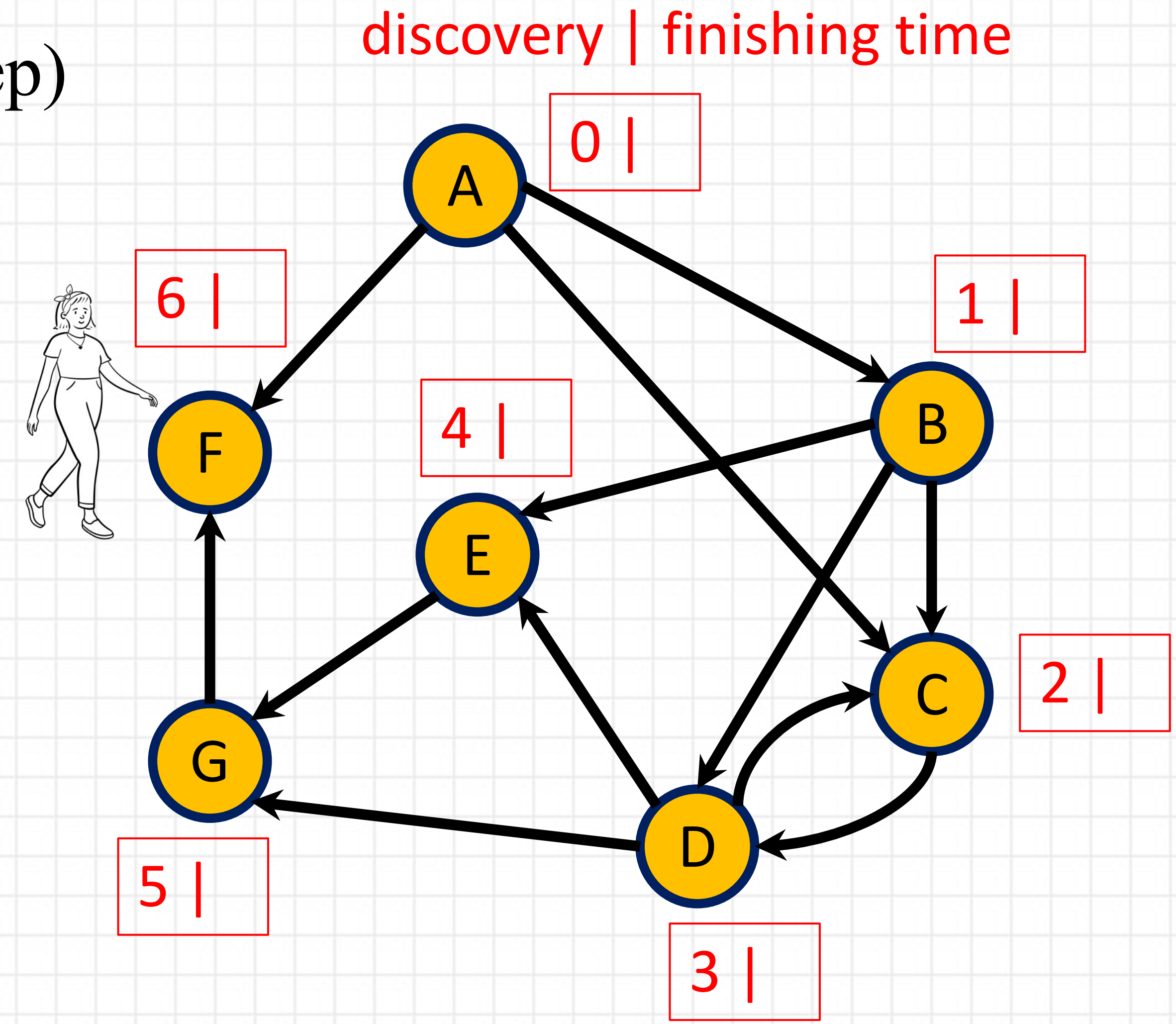
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



- Stack = {A, B, C, D, E, G, F}
- Visited = {A, B, C, D, E, G}

Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



- Stack = {A, B, C, D, E, G, F}
- Visited = {A, B, C, D, E, G, F}

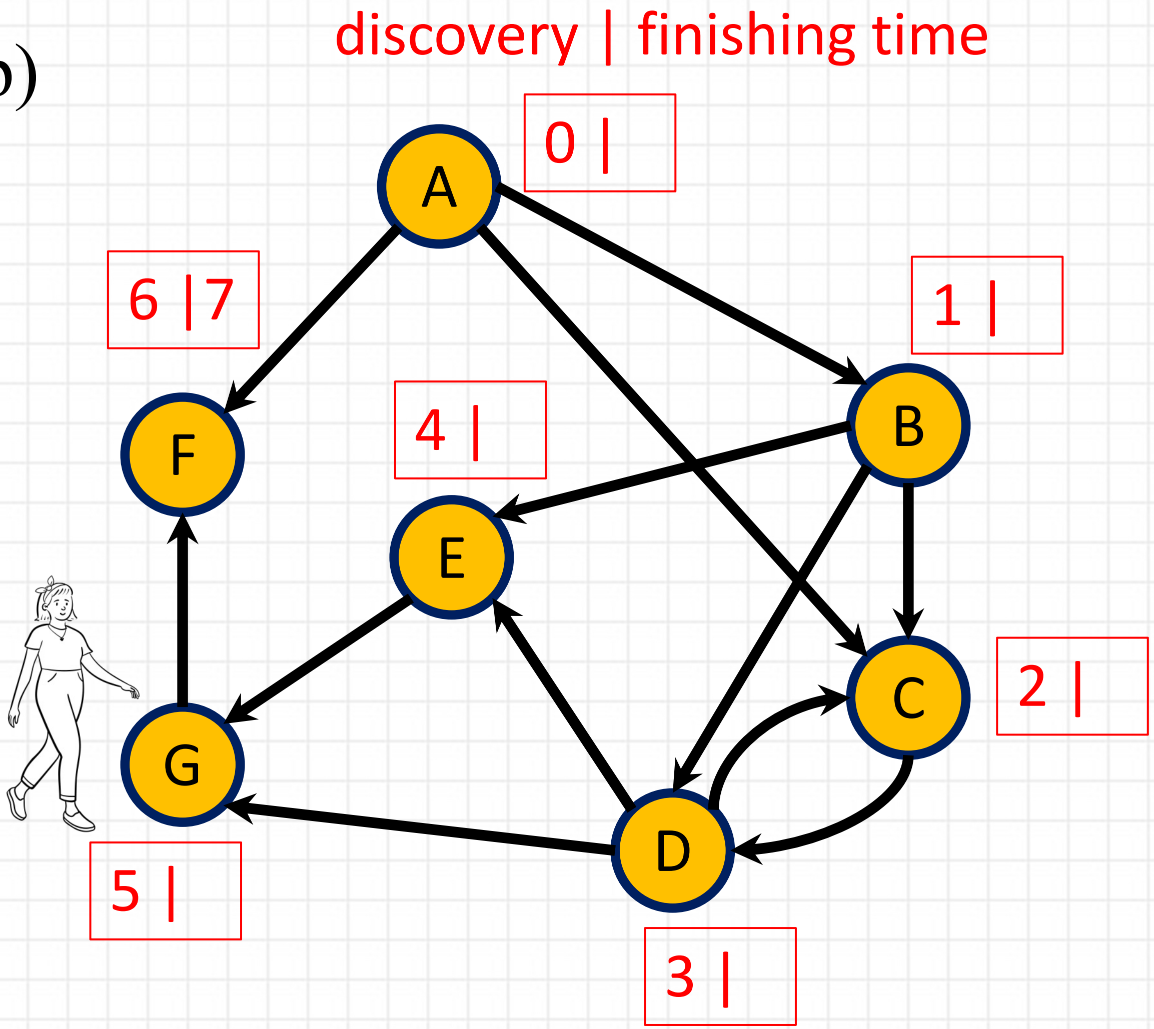
Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

Pop

- Stack = {A, B, C, D, E, G, ~~F~~}
- Visited = {A, B, C, D, E, G, F}

- No more path to explore→ backtrack



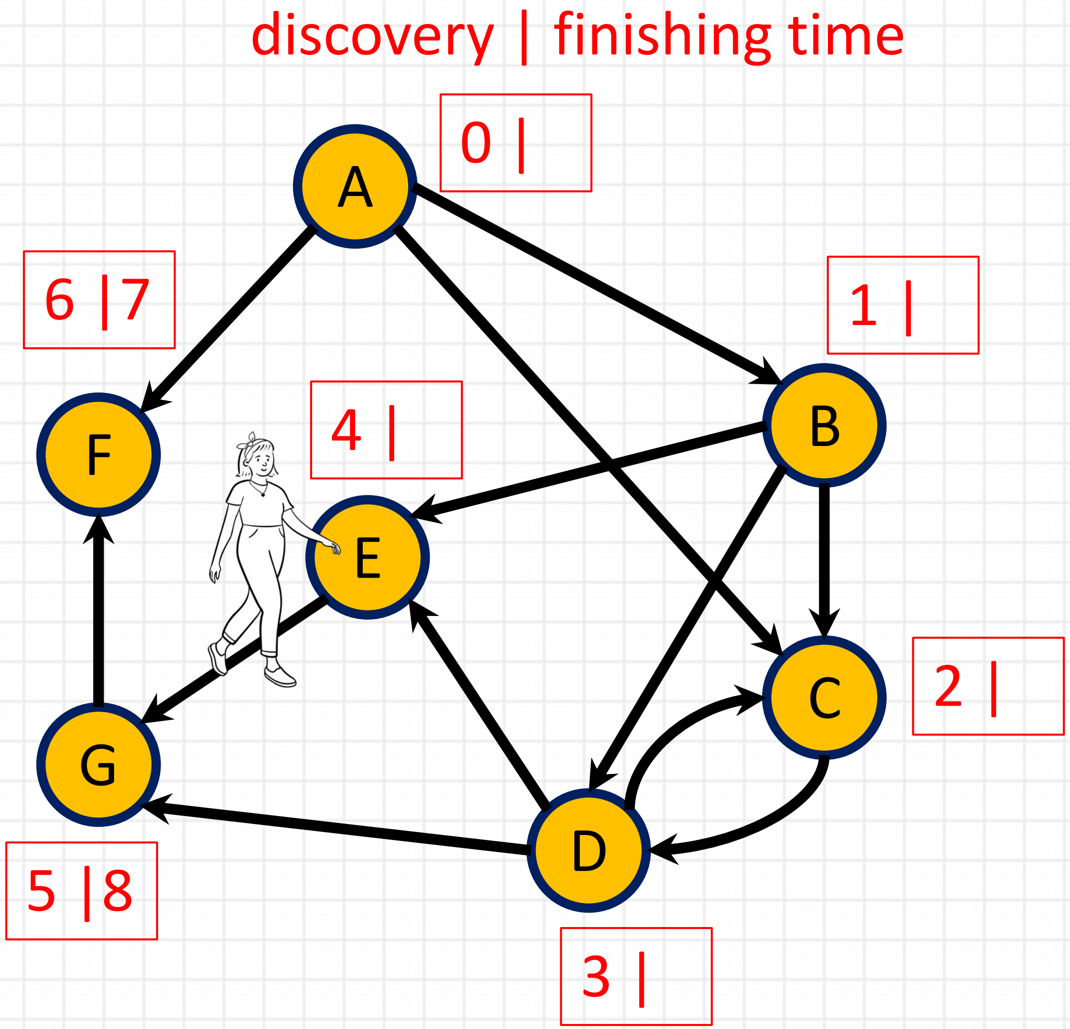
Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

- Stack = {A, B, C, D, E, ~~G~~, ~~F~~}
- Visited = {A, B, C, D, E, G, F}

Pop

- No more path to explore→ backtrack



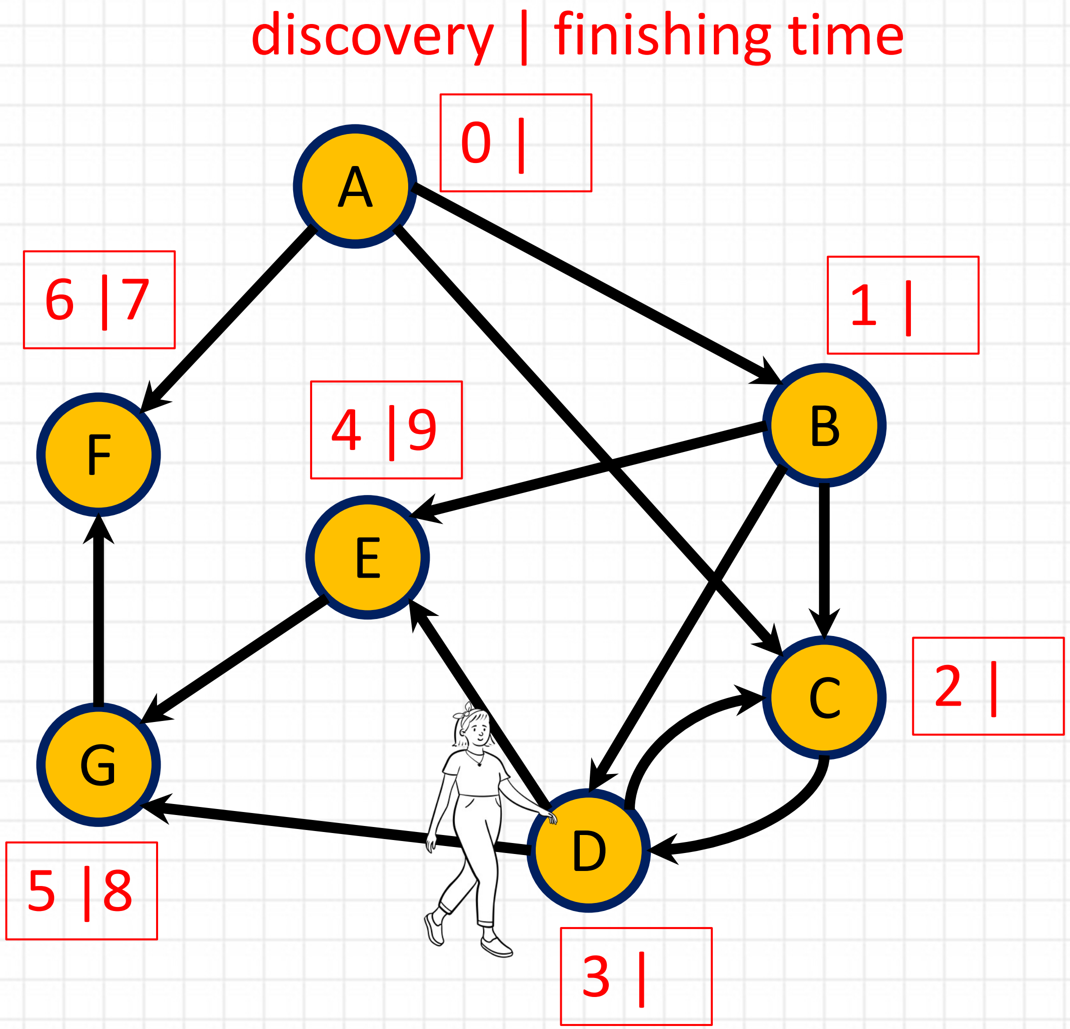
Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

Pop

- Stack = {A, B, C, D, ~~E, G, F~~}
- Visited = {A, B, C, D, E, G, F}

- No more path to explore→ backtrack



Graph Traversal: DFS

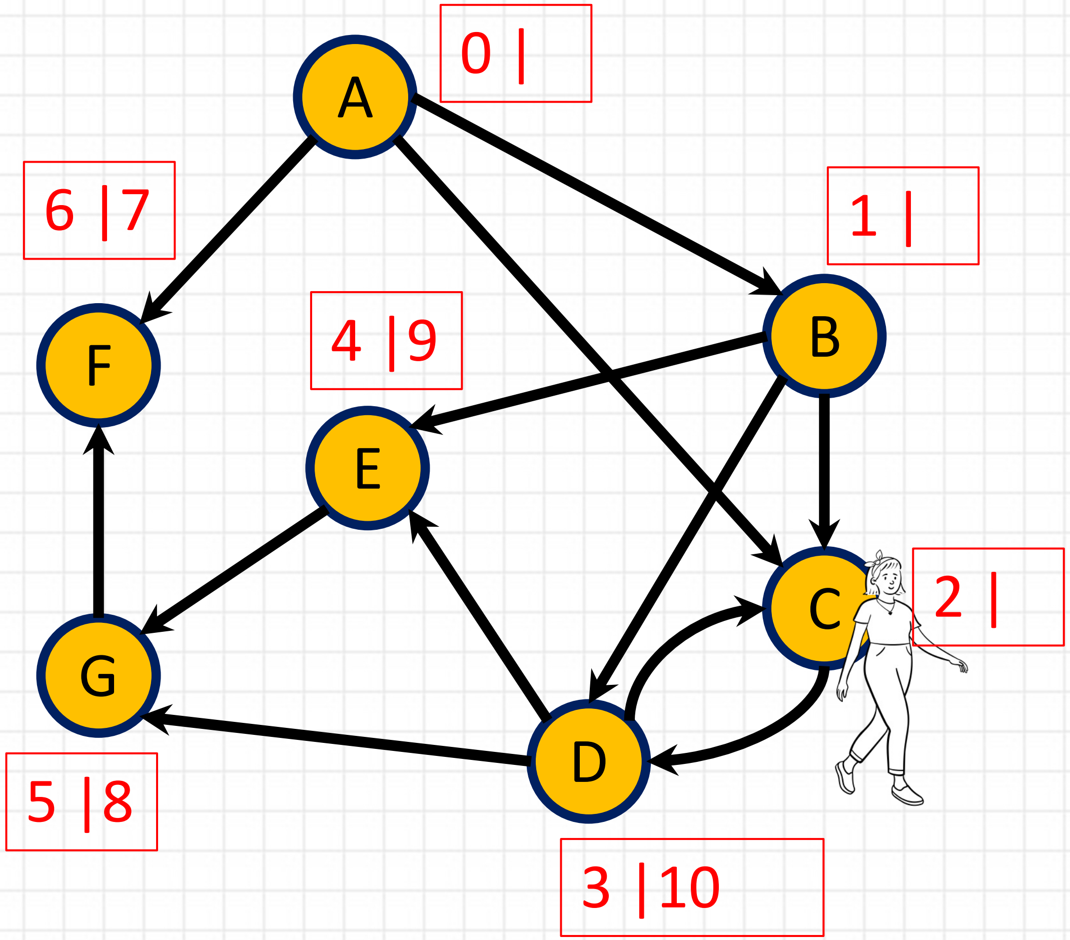
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

Pop

- Stack = {A, B, C, ~~D~~, ~~E~~, ~~G~~, ~~F~~}
- Visited = {A, B, C, D, E, G, F}

- No more path to explore→ backtrack

discovery | finishing time



Graph Traversal: DFS

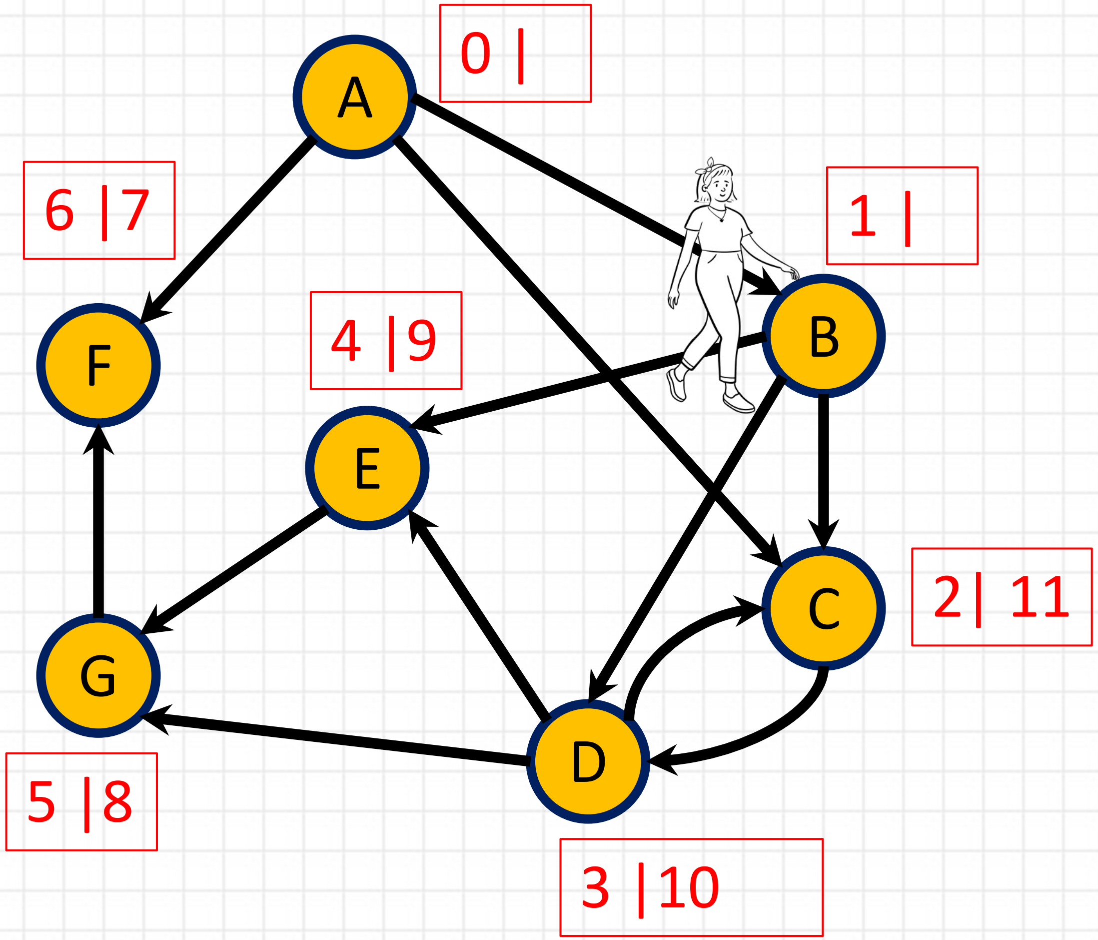
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

Pop

- Stack = {A, B, ~~C~~, ~~D~~, ~~E~~, ~~G~~, ~~F~~}
- Visited = {A, B, C, D, E, G, F}

- No more path to explore→ backtrack

discovery | finishing time



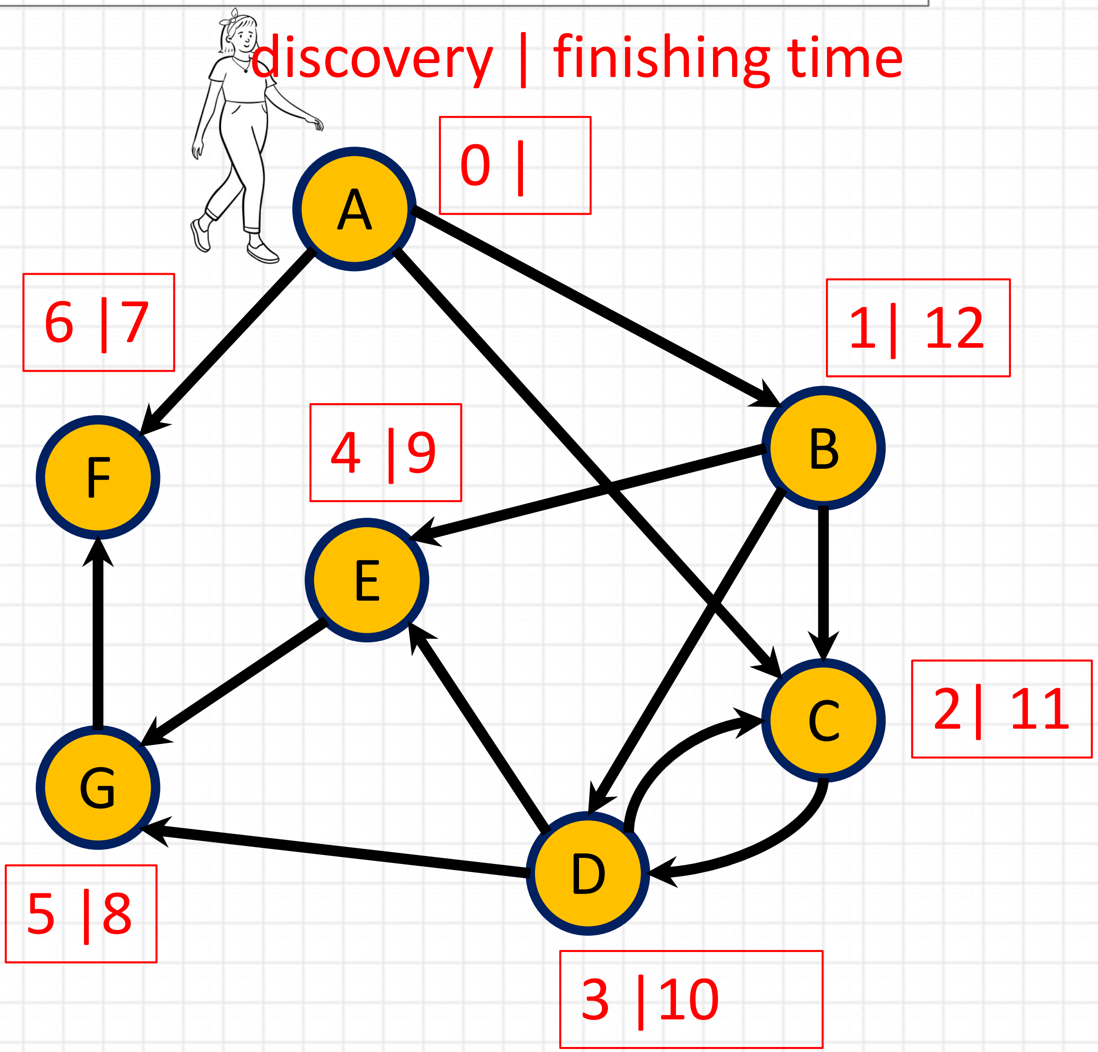
Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

Pop

- Stack = {A, ~~B~~, ~~C~~, ~~D~~, ~~E~~, ~~G~~, ~~F~~}
- Visited = {A, B, C, D, E, G, F}

- No more path to explore→ backtrack

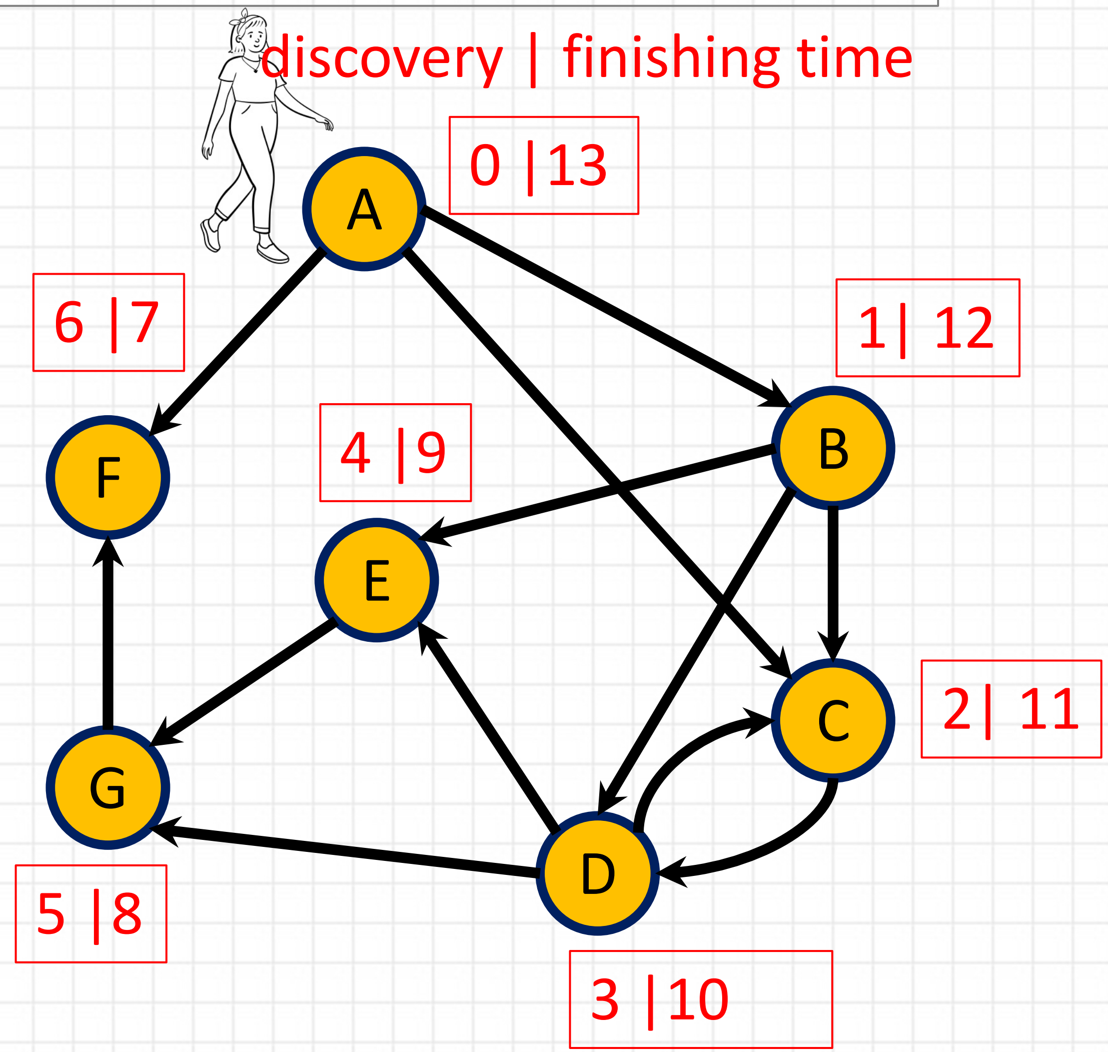


Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

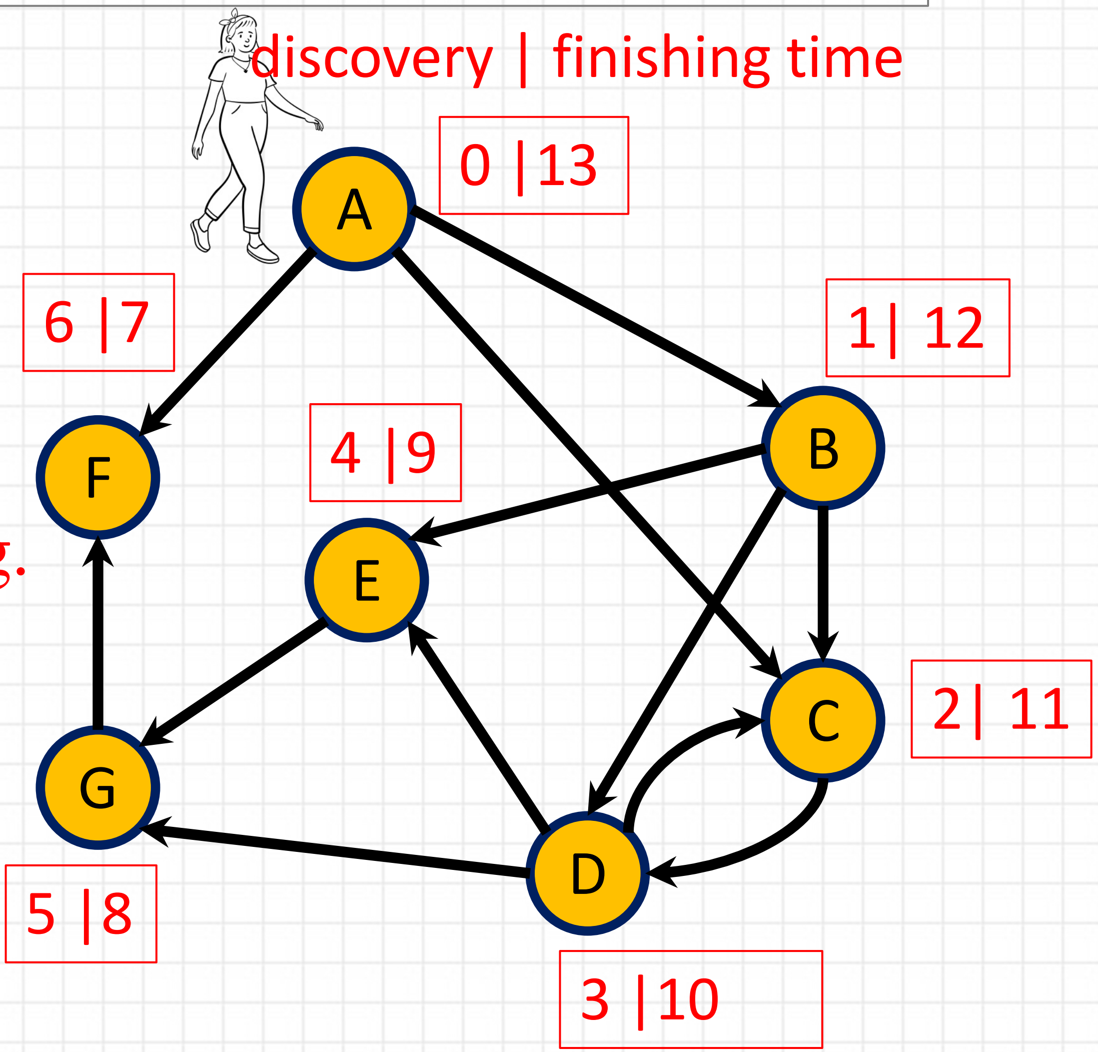
Pop

- Stack = {~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~E~~, ~~G~~, ~~F~~}
- Visited = {A, B, C, D, E, G, F}
- No more path to explore → backtrack
- No more element in the stack → Halt



Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Note in this example we were able to reach all nodes without any backtracking. But this is not usually the case in many examples!

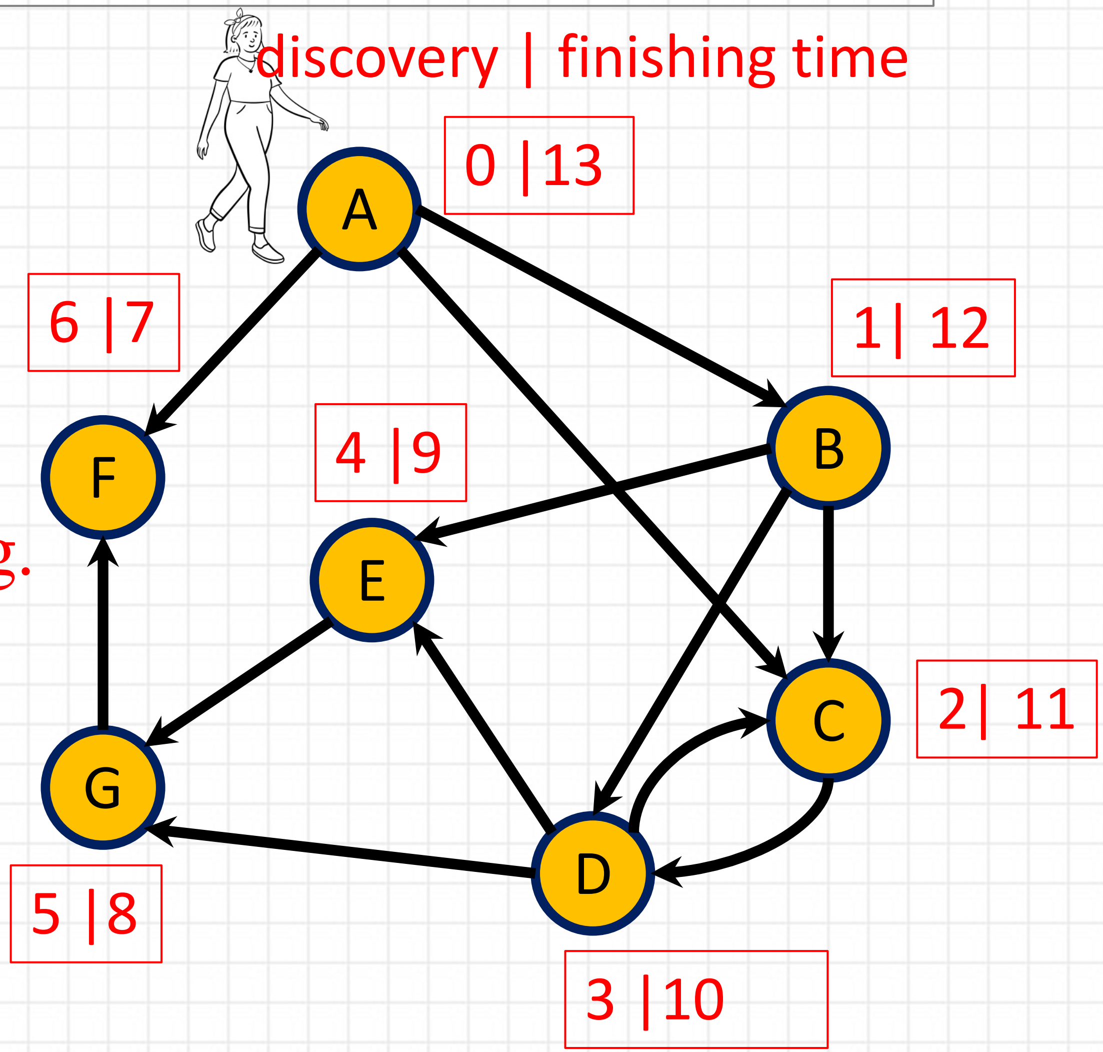


Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

- Note in this example we were able to reach all nodes without any backtracking. But this is not usually the case in many examples!

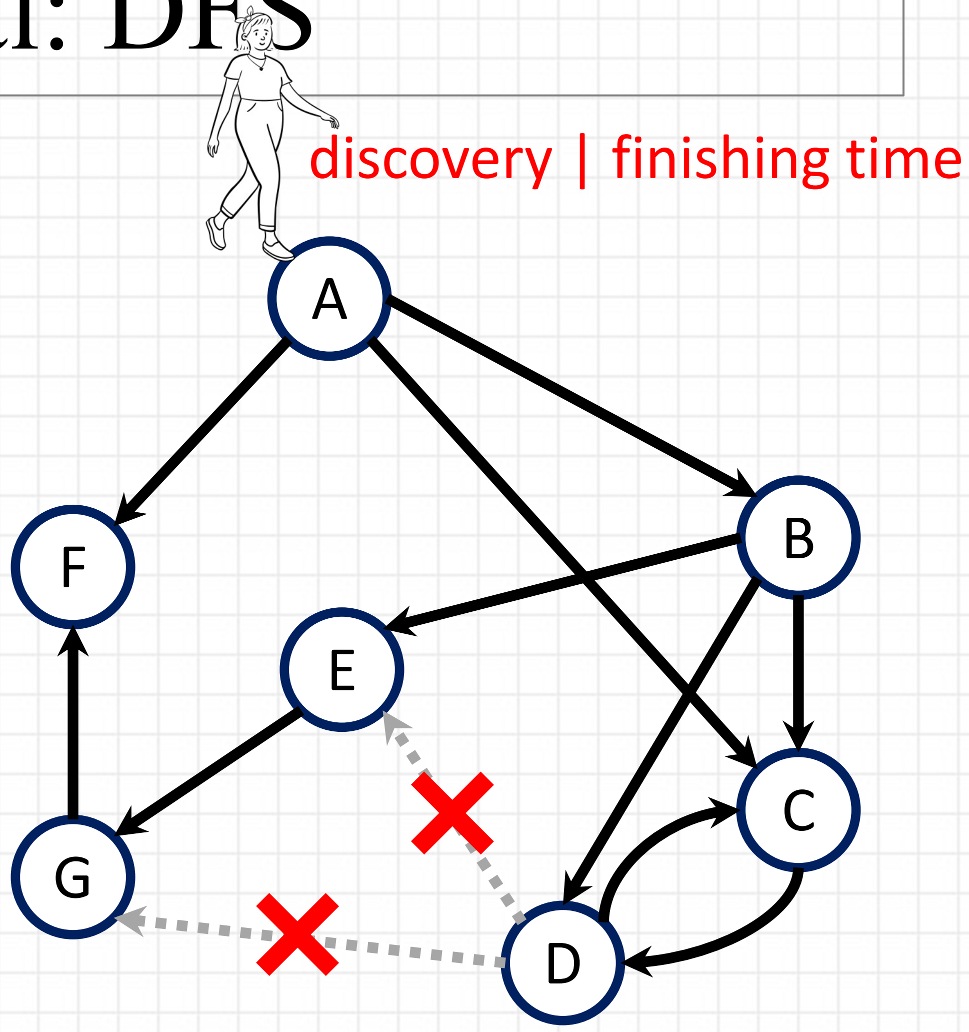
- → Consider the same example, with minor difference:



Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

- Stack = {A}
- Visited = {A}

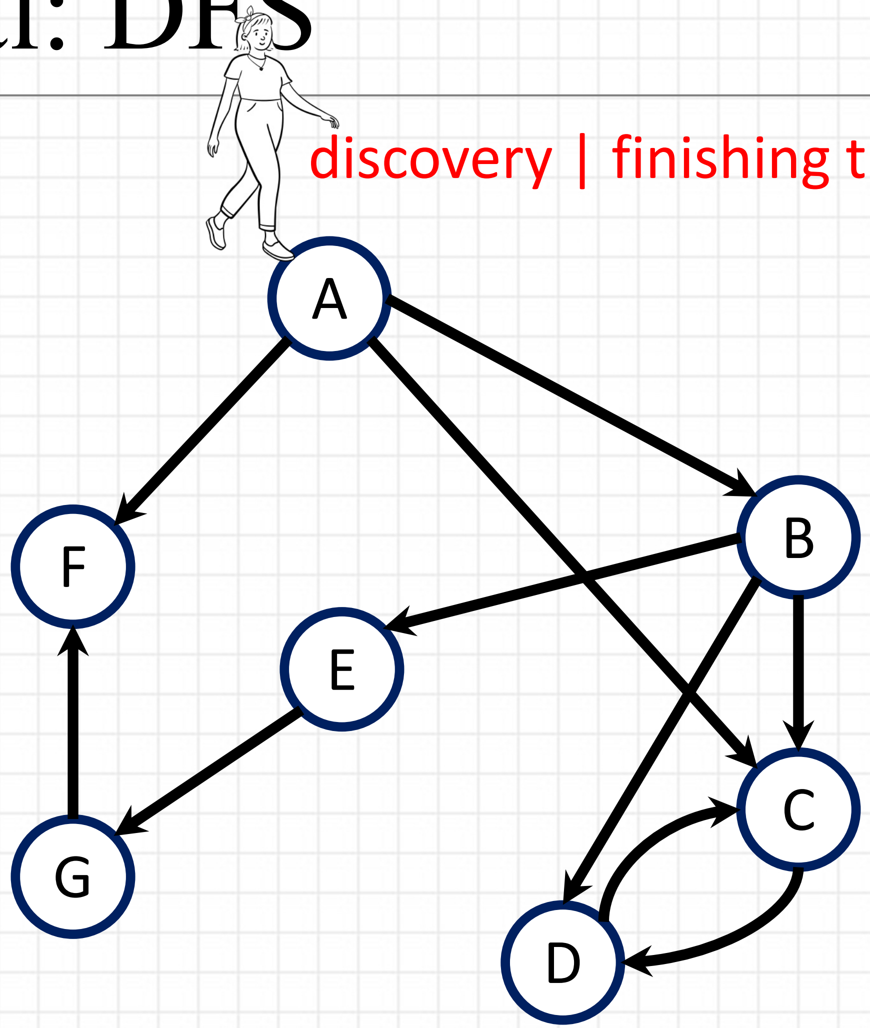


Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

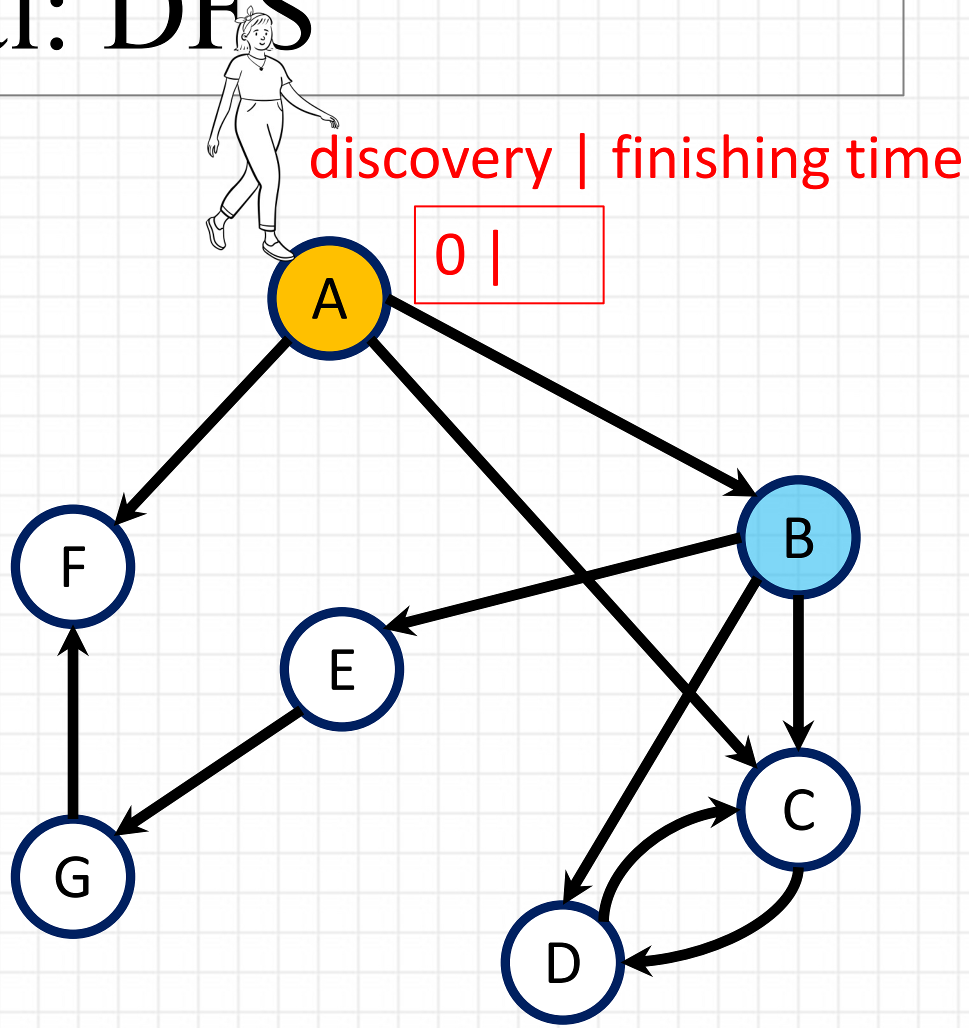
- Stack = {A}
- Visited = {A}

discovery | finishing time



Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

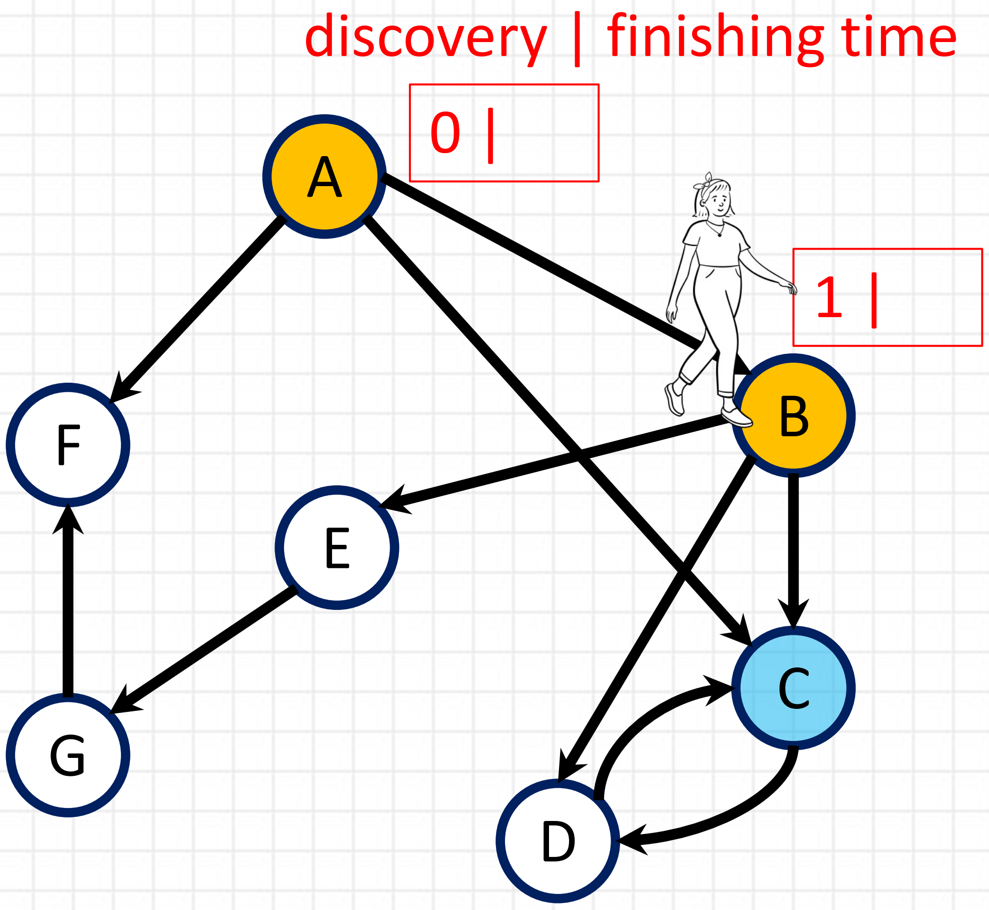


- Stack = {A, B}
- Visited = {A}

Graph Traversal: DFS

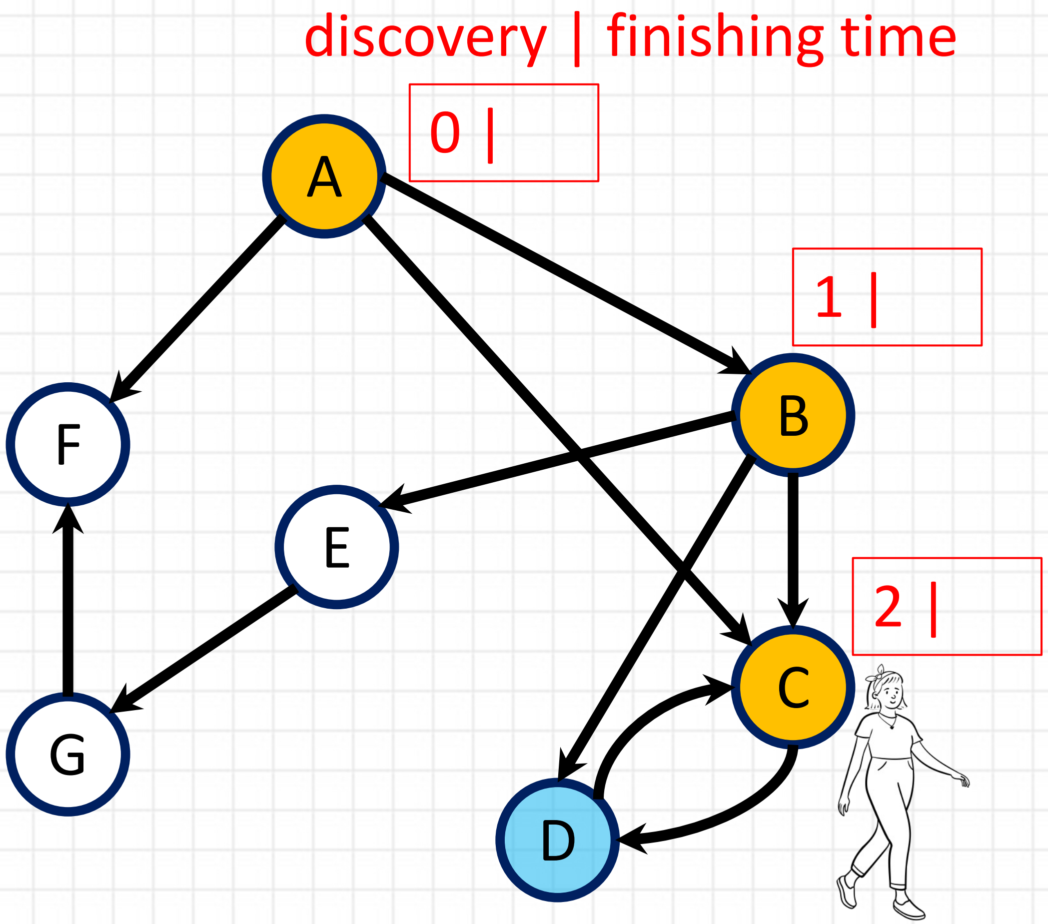
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

- Stack = {A, B, C}
- Visited = {A, B}



Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

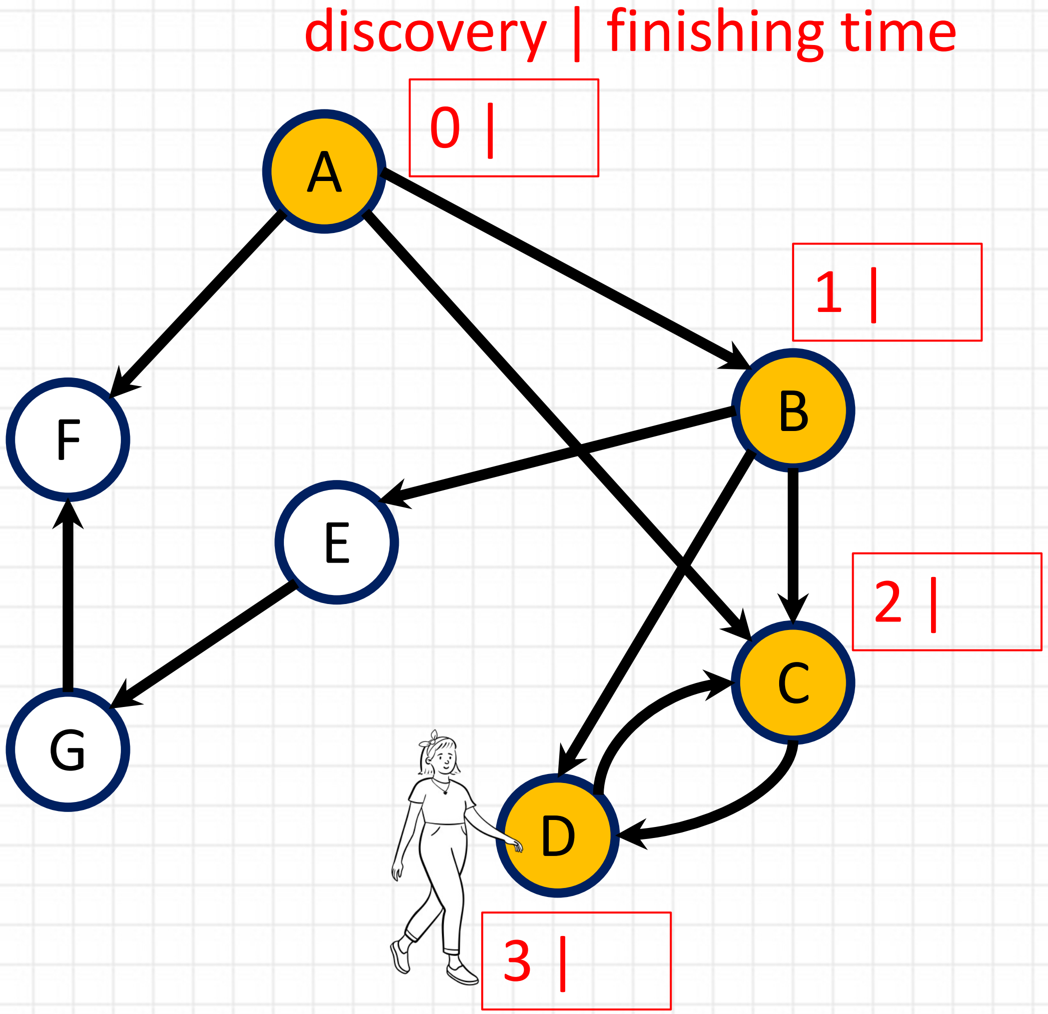


- Stack = {A, B, C, D}
- Visited = {A, B, C}

Graph Traversal: DFS

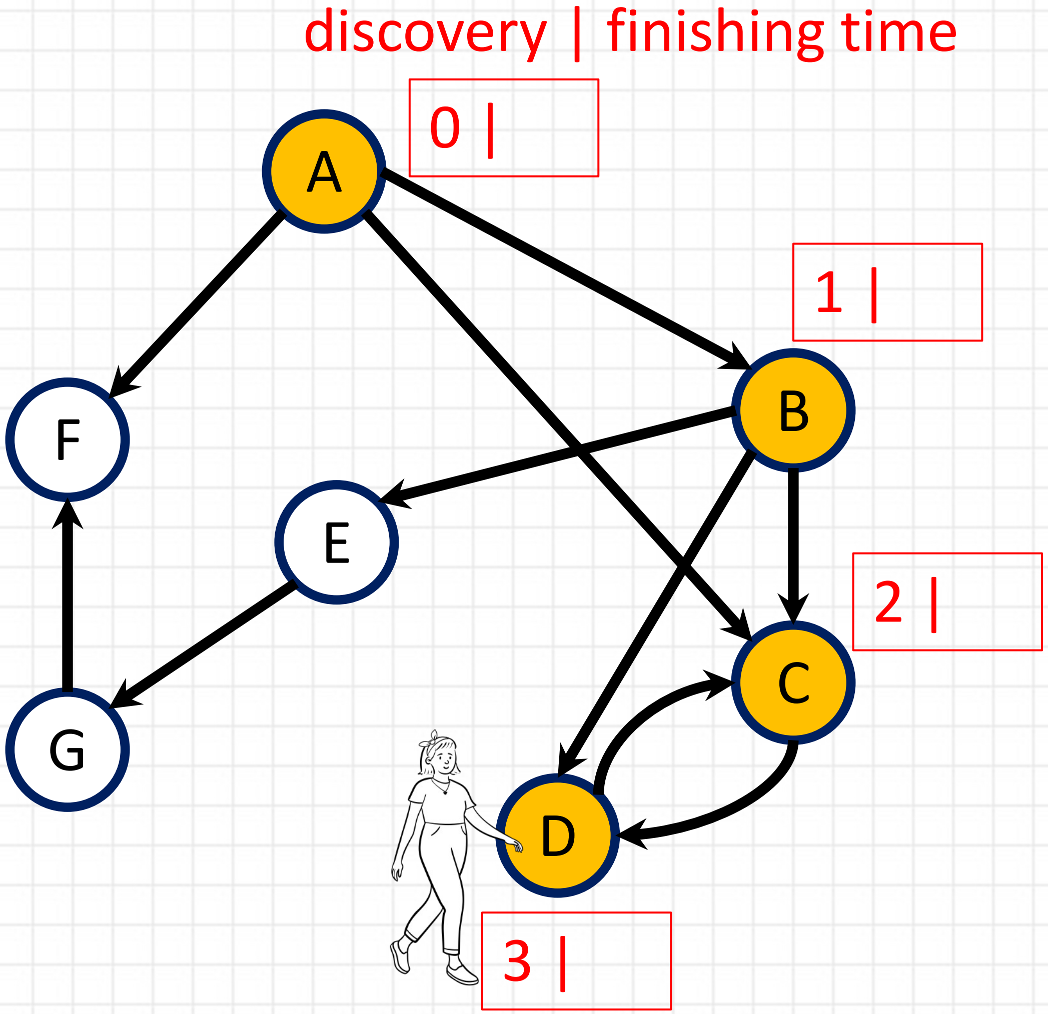
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

- Stack = {A, B, C, D}
- Visited = {A, B, C, D}



Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →
- Stack = {A, B, C, D}
- Visited = {A, B, C, D}
- No more path to explore → backtrack

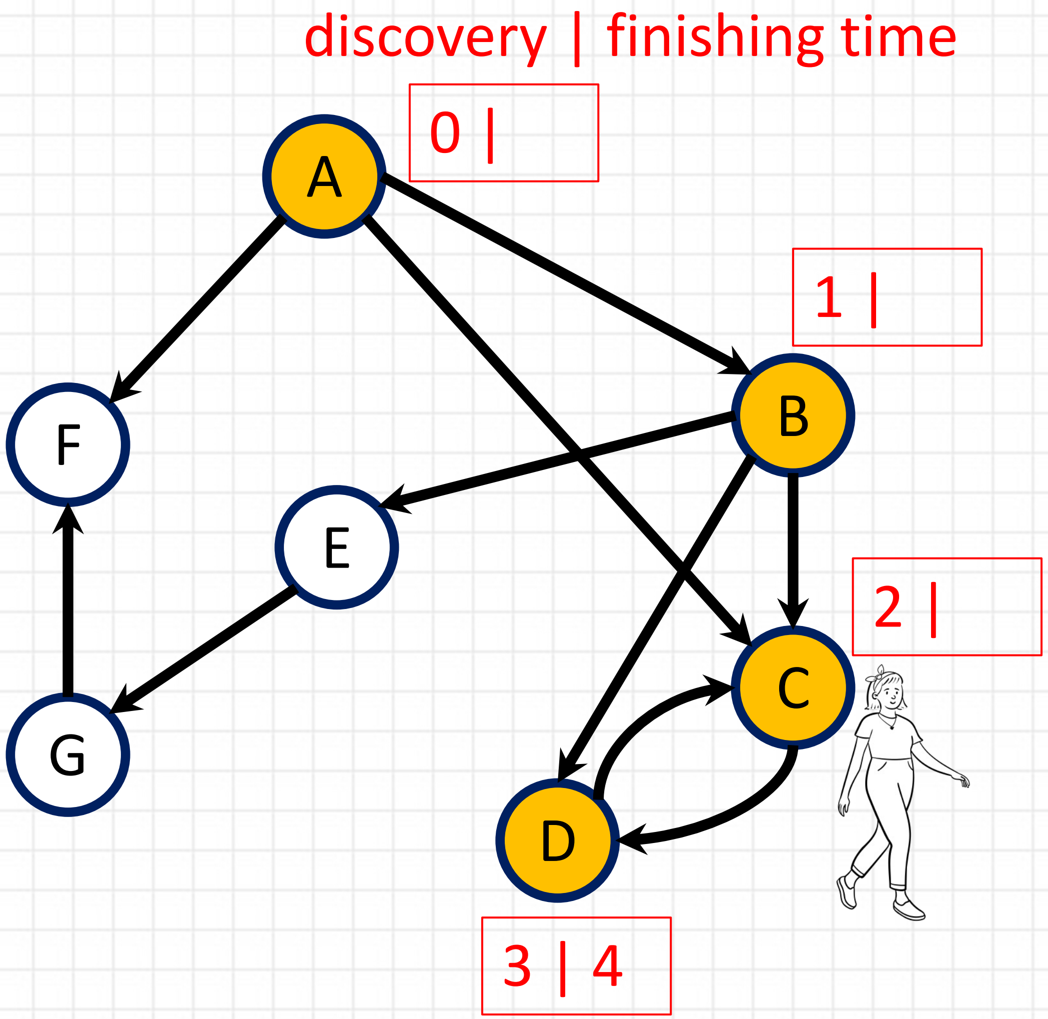


Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

Pop

- Stack = {A, B, C, ~~D~~}
- Visited = {A, B, C, D}
- No more path to explore → backtrack

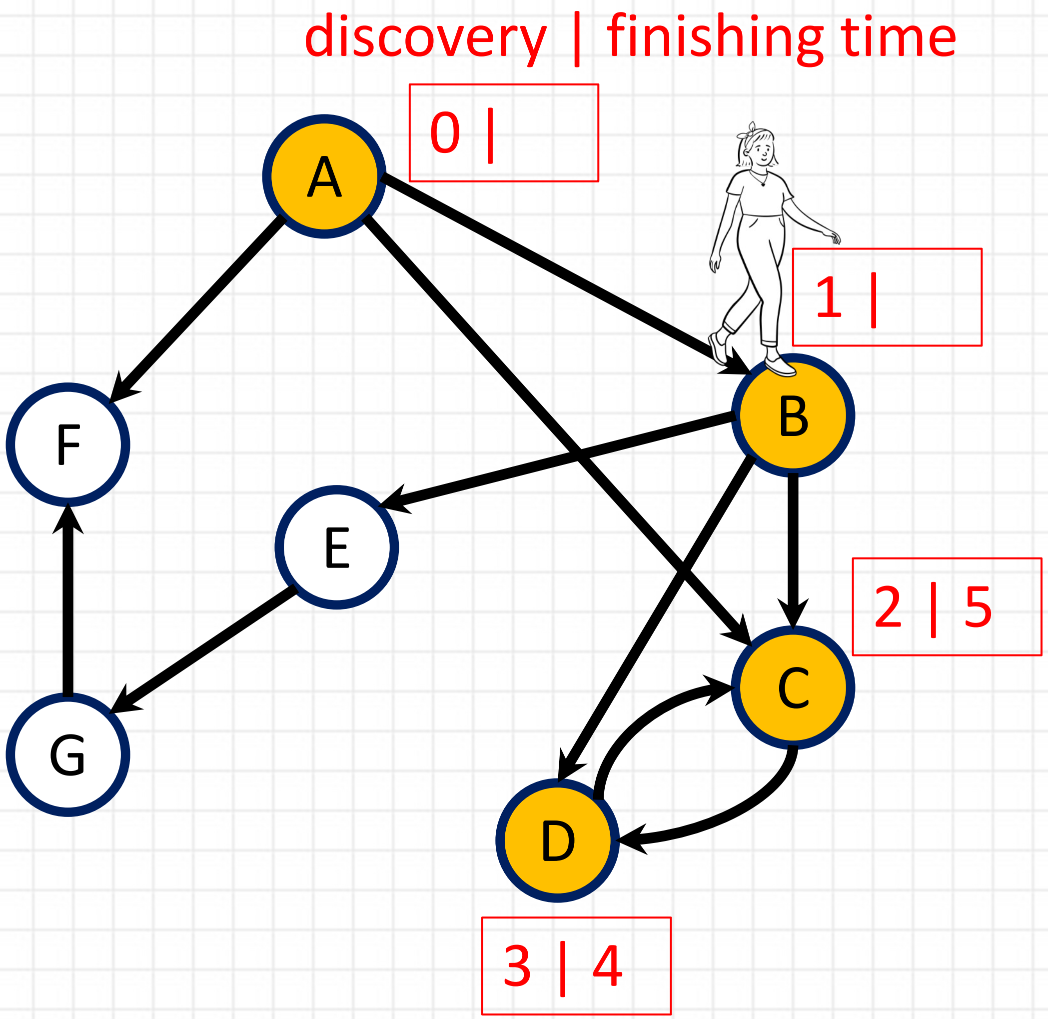


Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

Pop

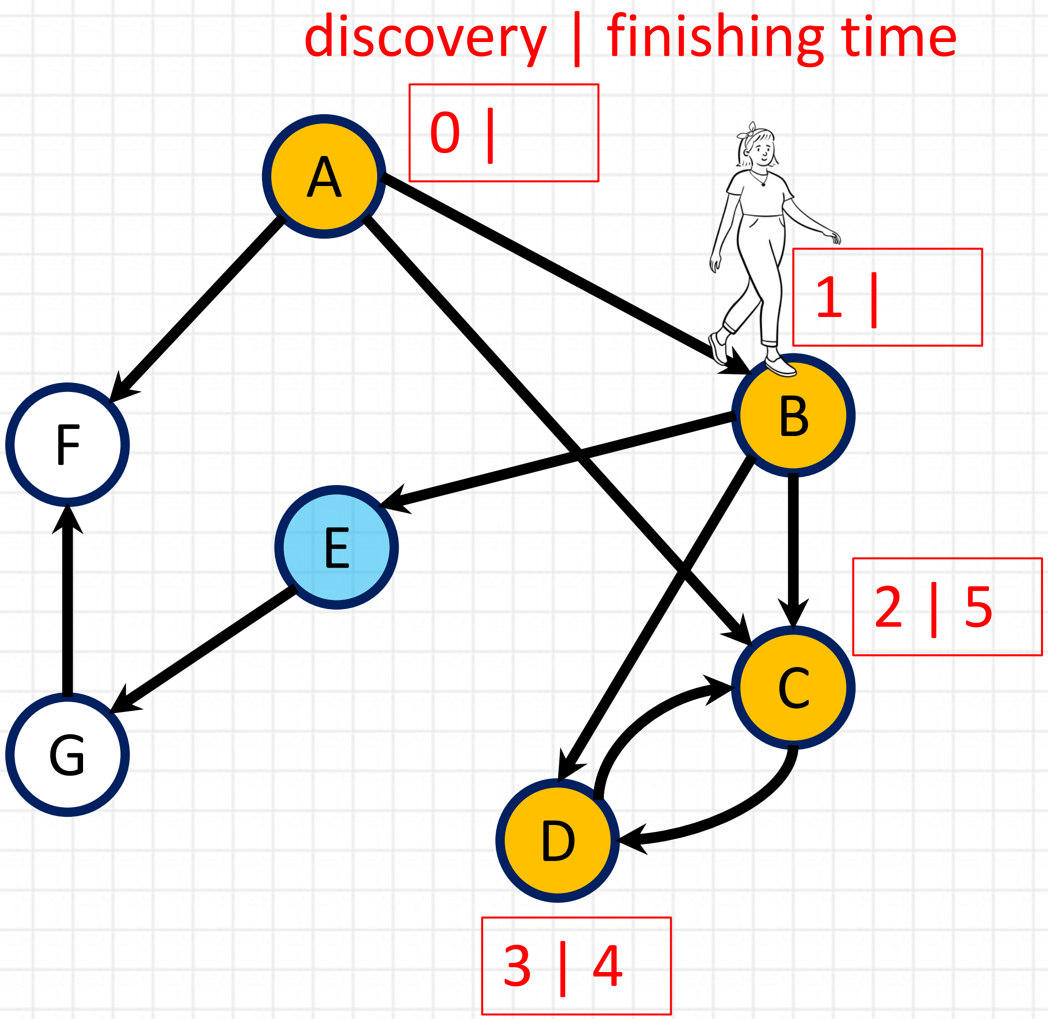
- Stack = {A, B, ~~C~~, ~~D~~}
- Visited = {A, B, C, D}
- No more path to explore → backtrack



Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

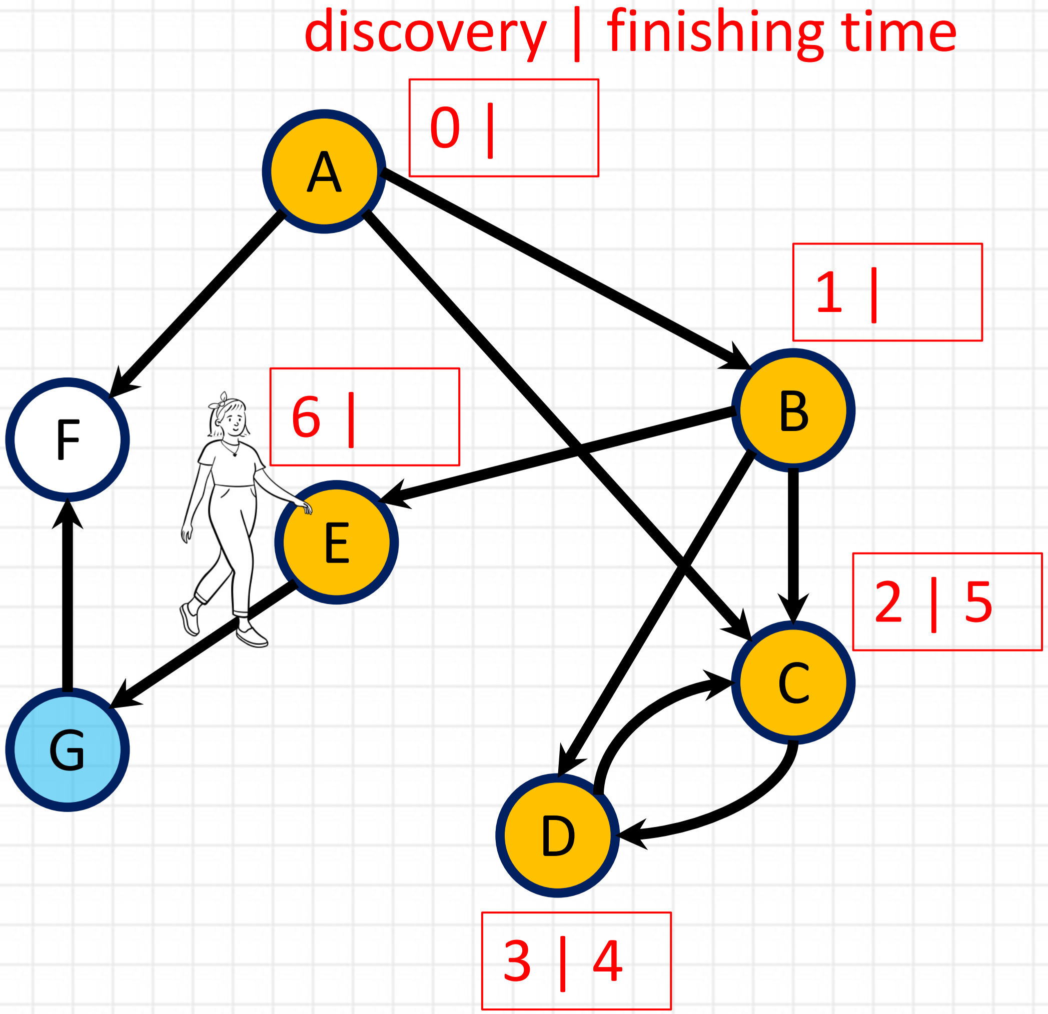
- Stack = {A, B, ~~C~~, ~~D~~, E}
- Visited = {A, B, C, D}



Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

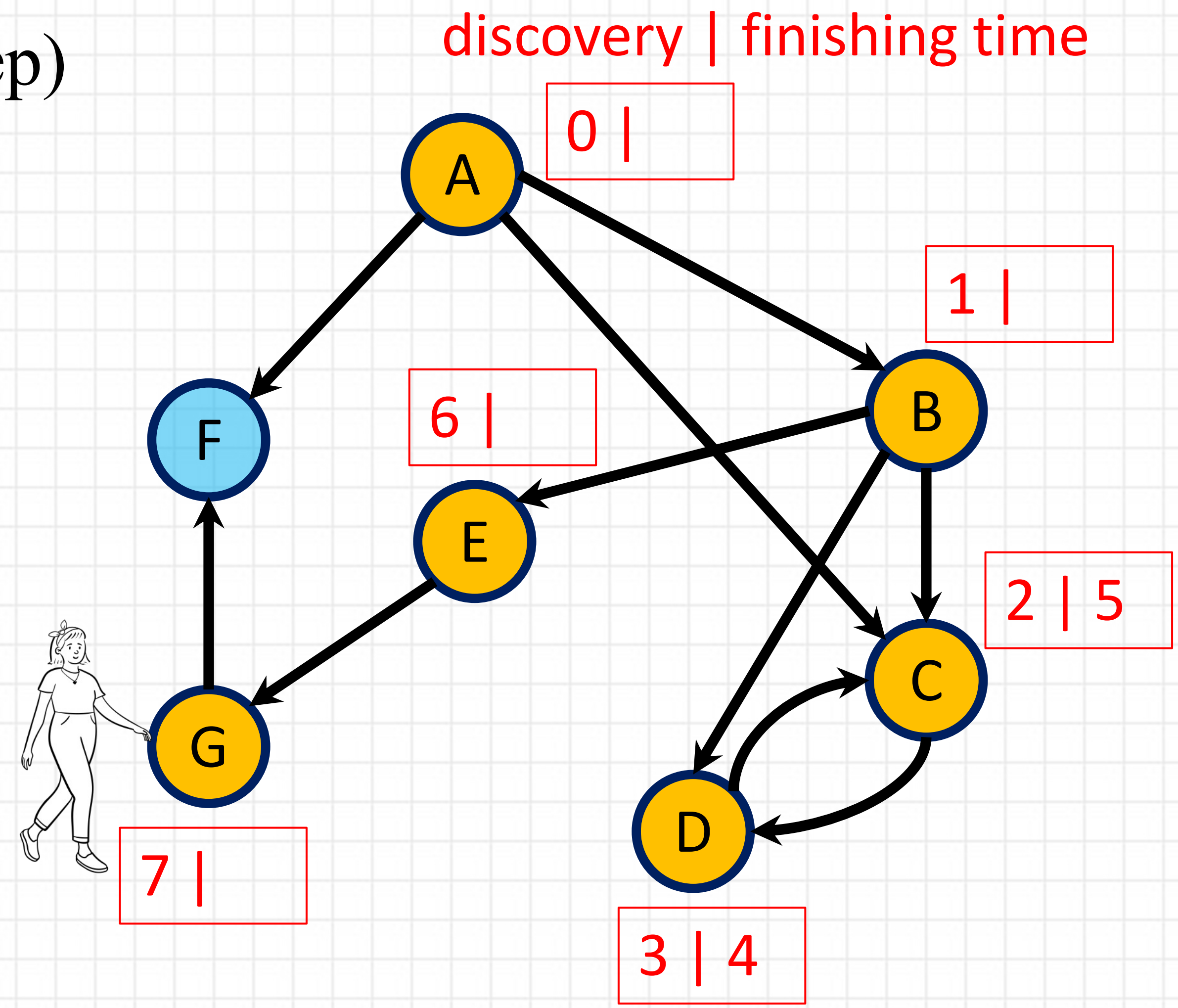
- Stack = {A, B, ~~C~~, ~~D~~, E, G}
- Visited = {A, B, C, D, E}



Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

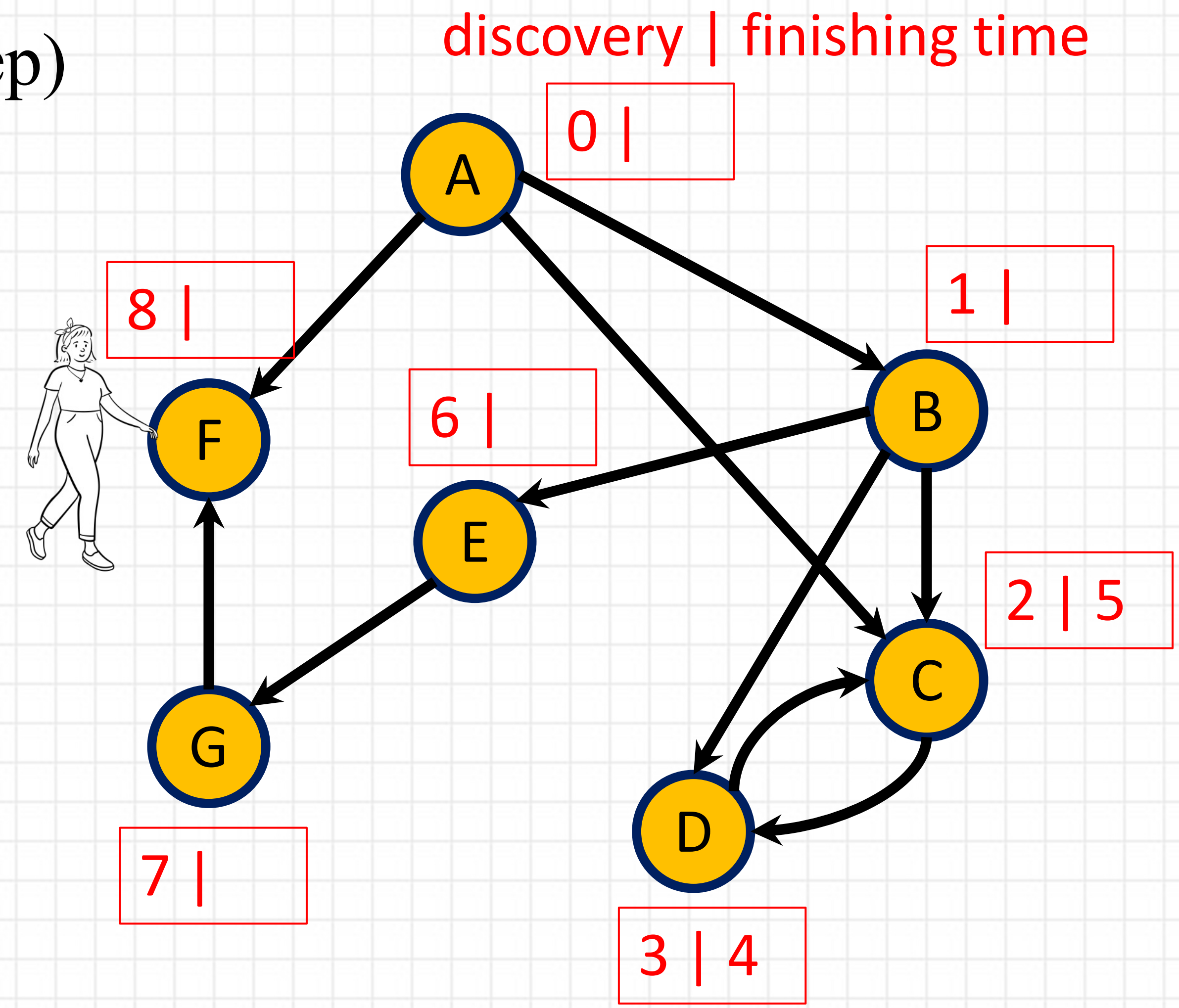
- Stack = {A, B, ~~C~~, ~~D~~, E, G, F}
- Visited = {A, B, C, D, E, G}



Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

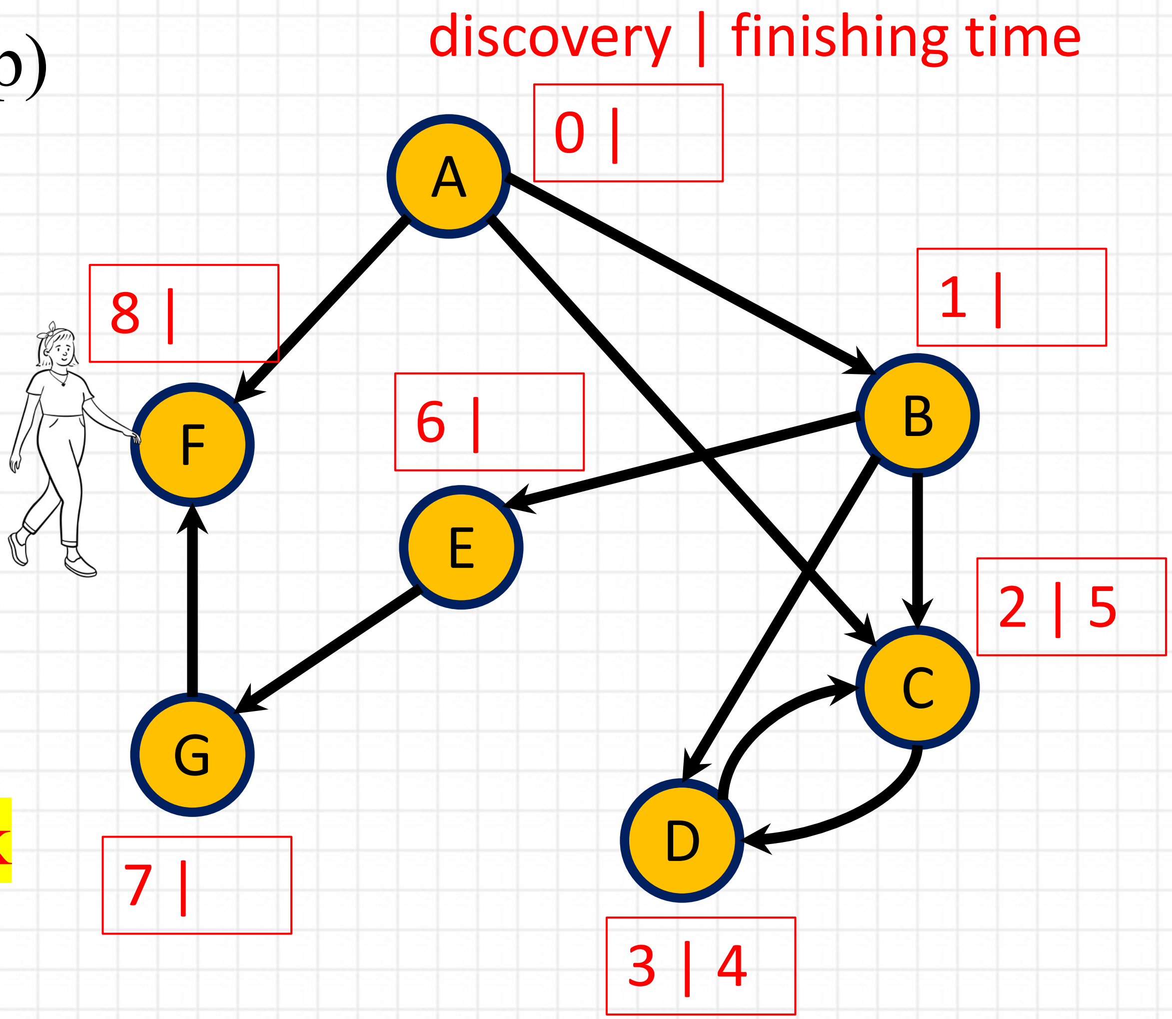
- Stack = {A, B, ~~C~~, ~~D~~, E, G, F}
- Visited = {A, B, C, D, E, G, F}



Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

- Stack = {A, B, ~~C~~, ~~D~~, E, G, F}
- Visited = {A, B, C, D, E, G, F}
- No more path to explore → backtrack

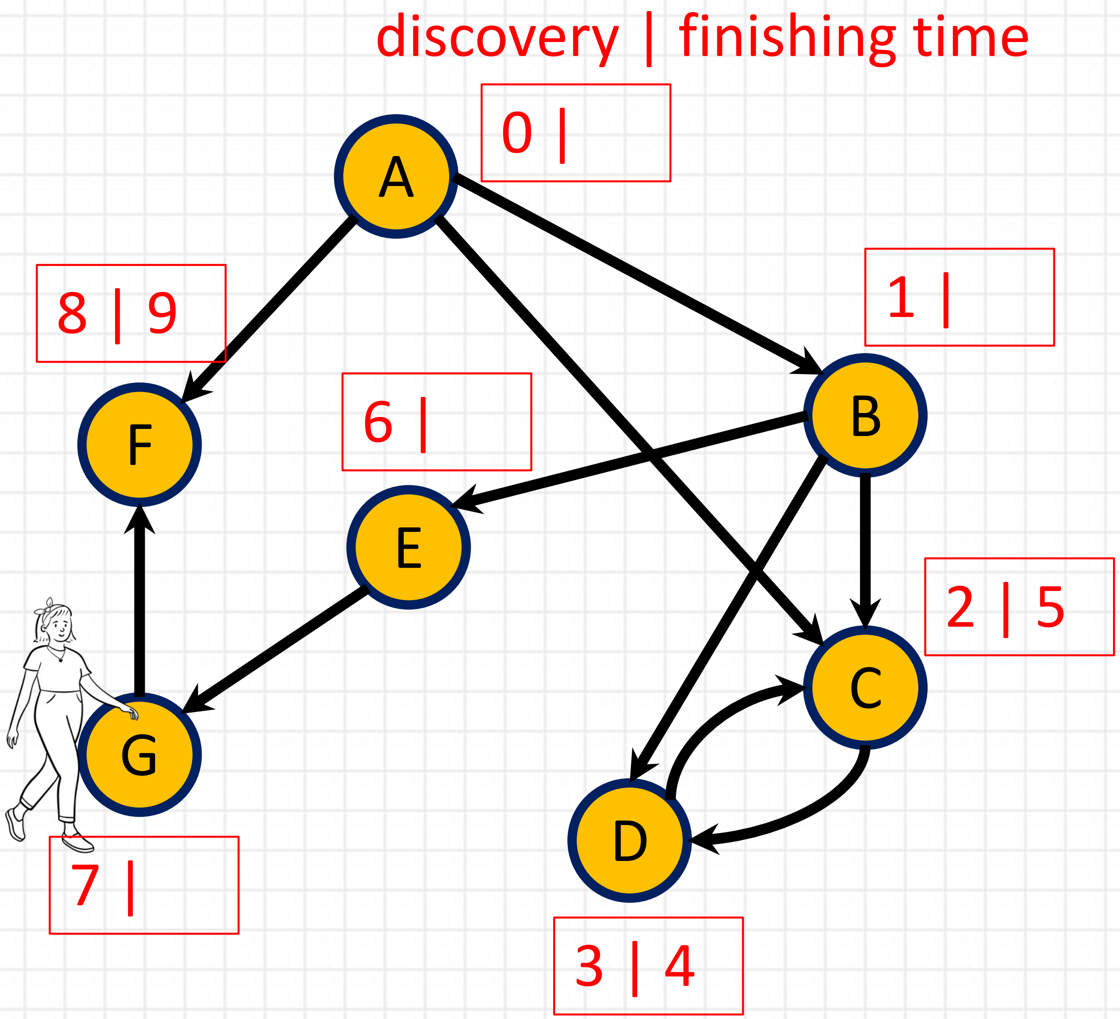


Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

Pop

- Stack = {A, B, ~~C~~, ~~D~~, E, G, ~~F~~}
- Visited = {A, B, C, D, E, G, F}
- No more path to explore → backtrack

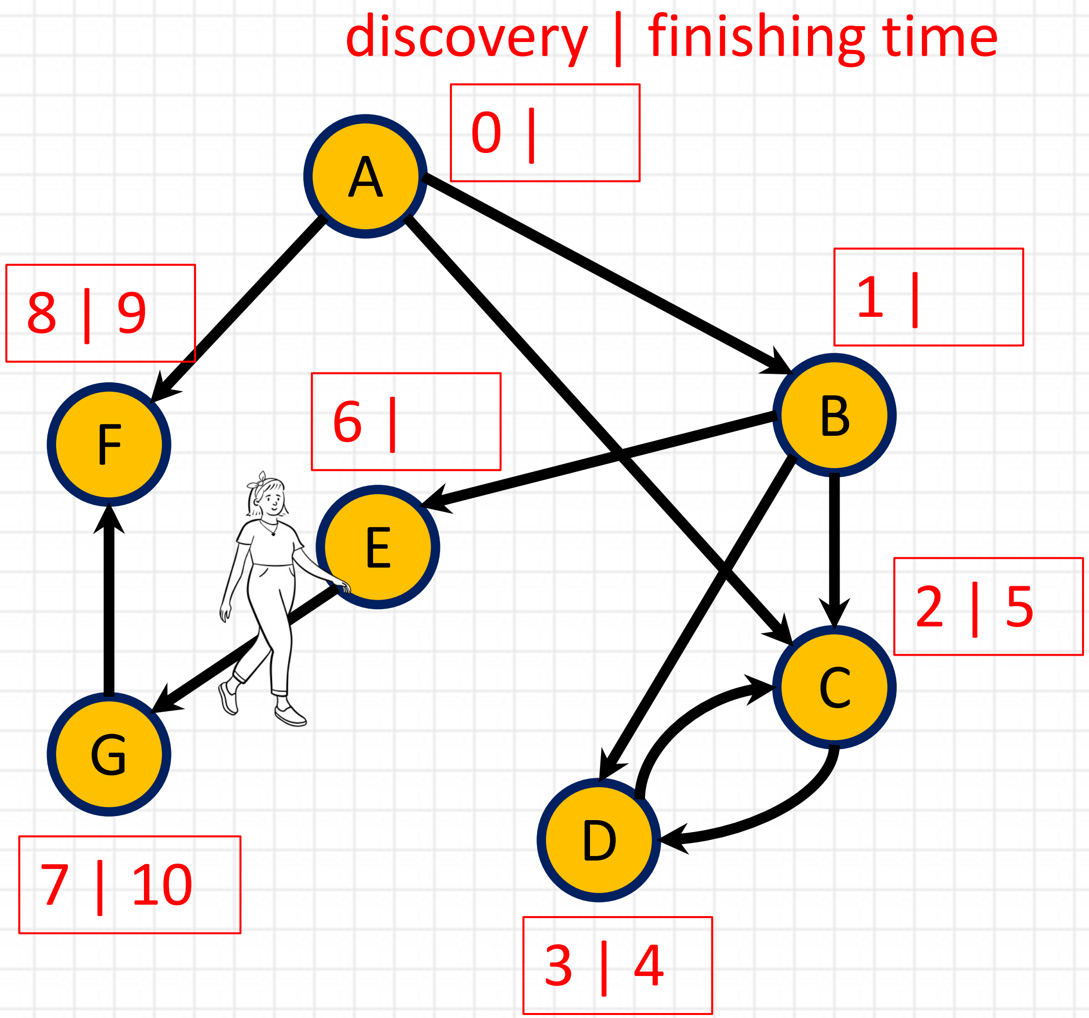


Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

Pop

- Stack = {A, B, ~~C~~, ~~D~~, E, ~~G~~, ~~F~~}
- Visited = {A, B, C, D, E, G, F}
- No more path to explore → backtrack

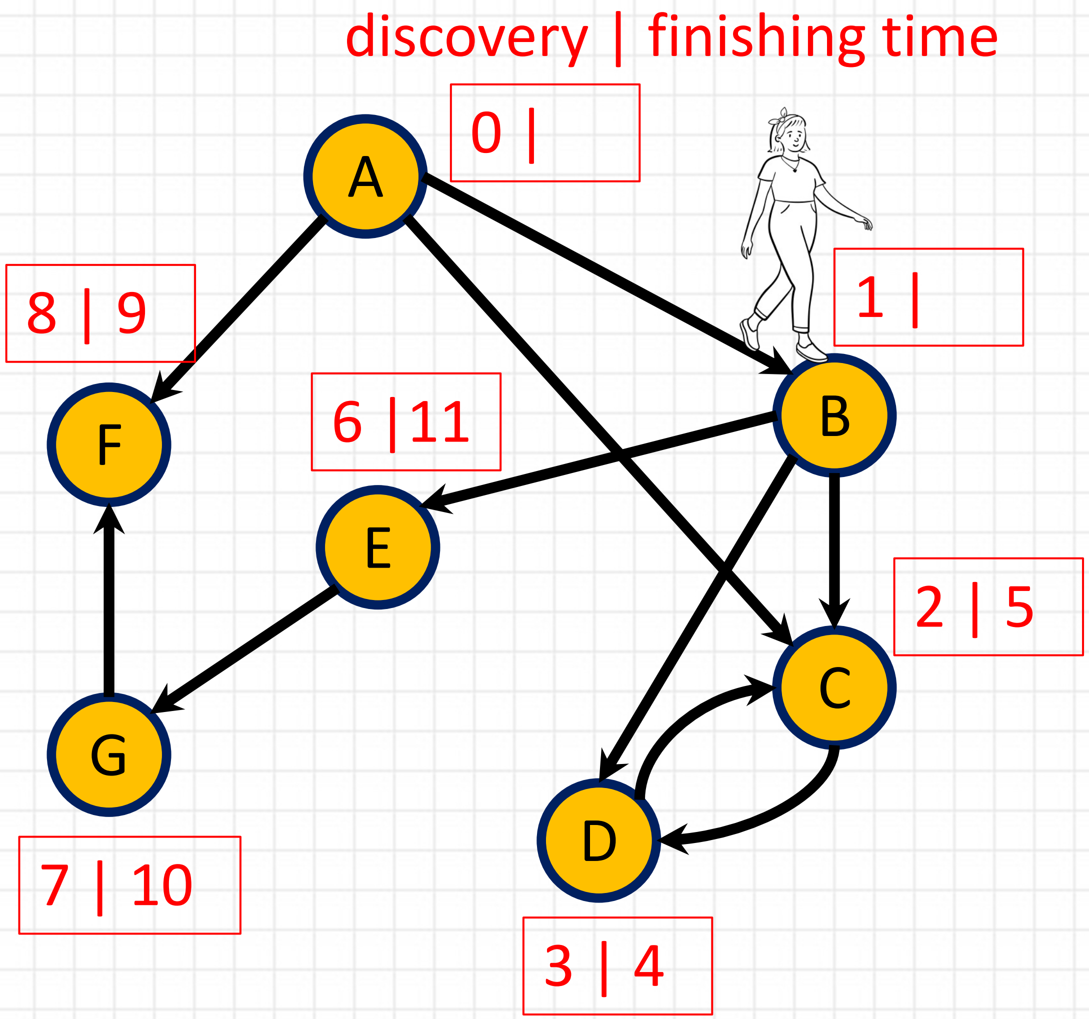


Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

Pop

- Stack = {A, B, ~~C~~, ~~D~~, ~~E~~, ~~G~~, ~~F~~}
- Visited = {A, B, C, D, E, G, F}
- No more path to explore → backtrack

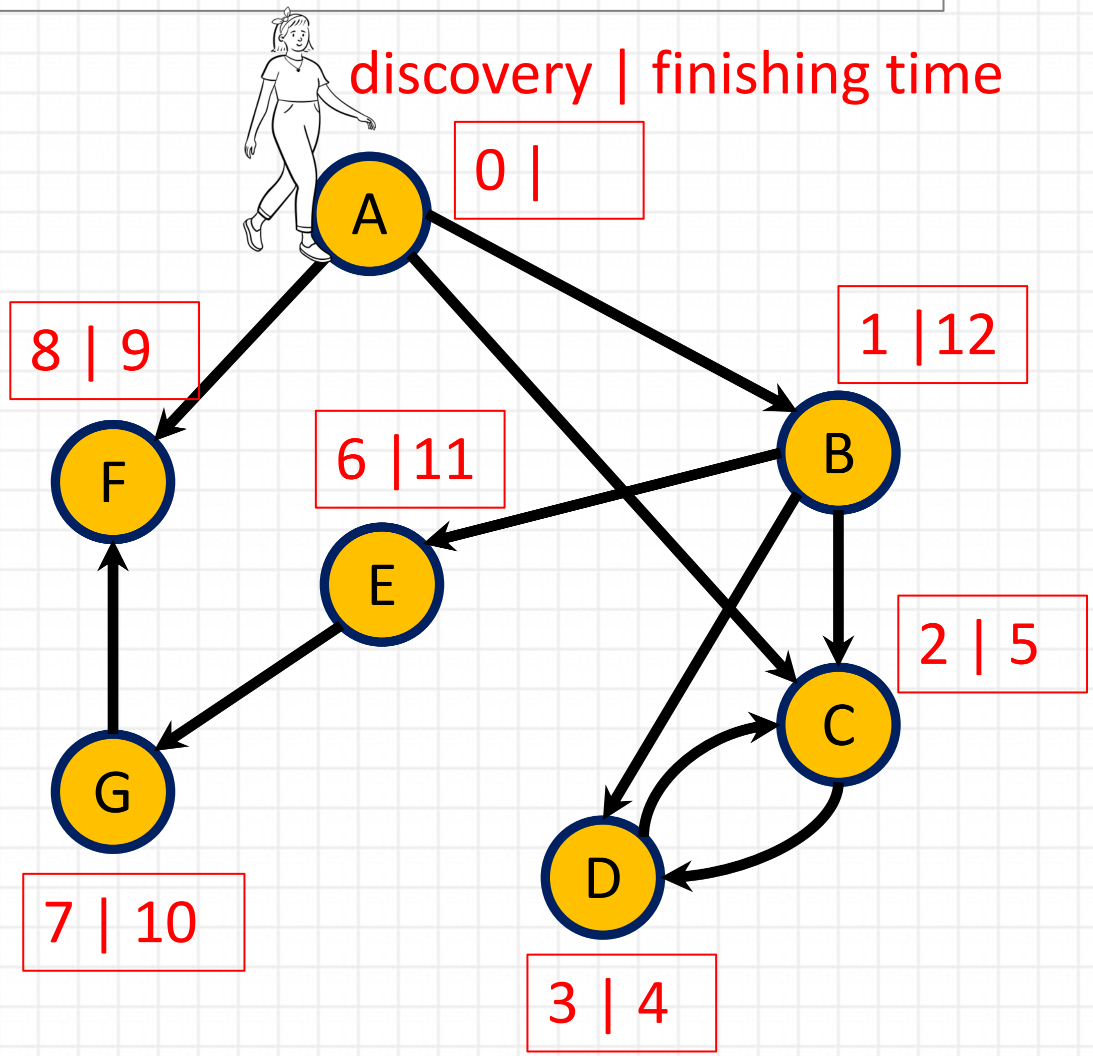


Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

Pop

- Stack = {A, ~~B~~, ~~C~~, ~~D~~, ~~E~~, ~~G~~, ~~F~~}
- Visited = {A, B, C, D, E, G, F}
- No more path to explore → backtrack



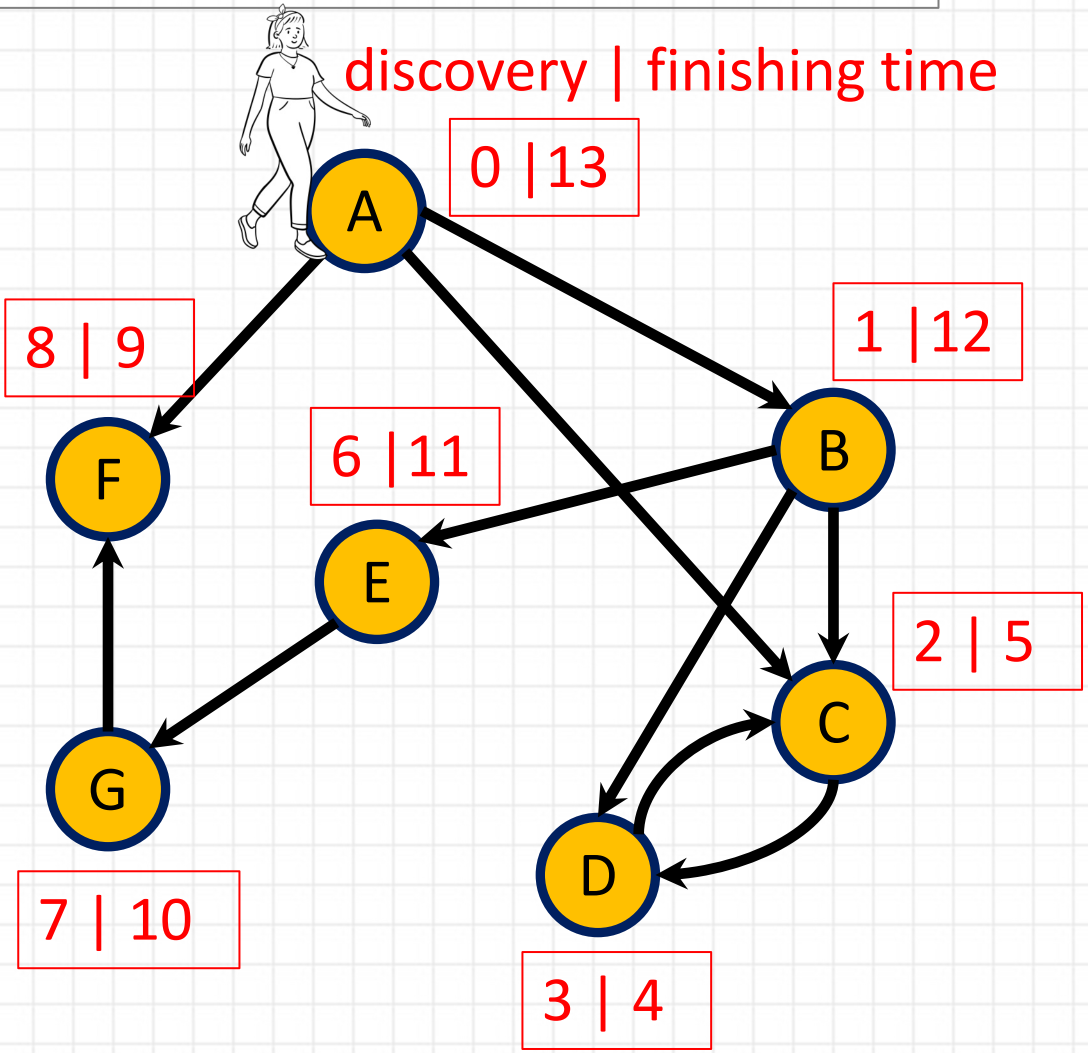
Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

Pop

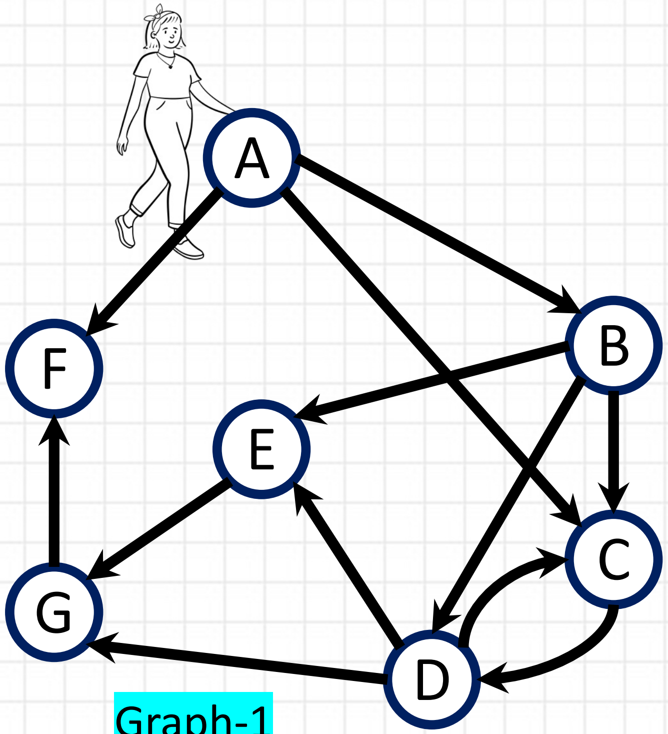
- Stack = {~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~E~~, ~~G~~, ~~F~~}
- Visited = {A, B, C, D, E, G, F}

Nothing left to explore → empty stack→ Halt
All nodes are visited, and we reach to the root

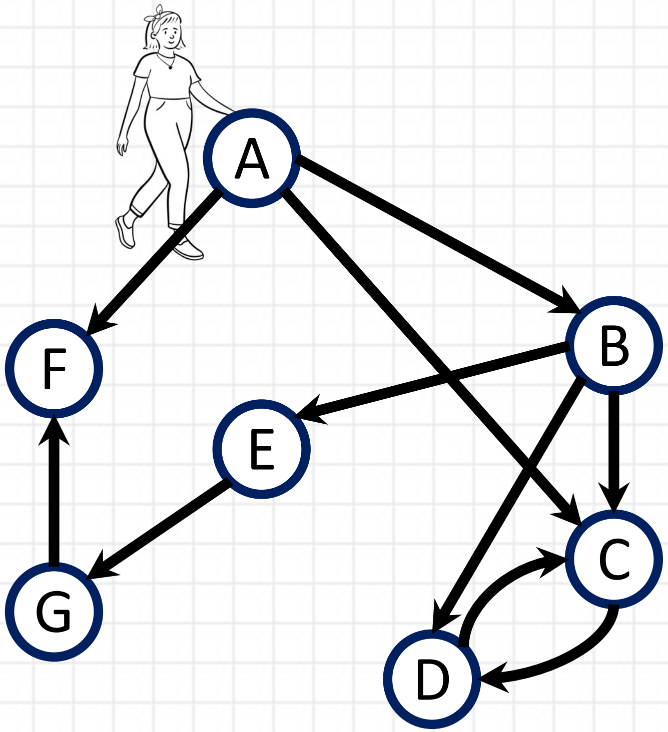


Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



Graph-1



Graph-2

Demo code time!

Graph Traversal: DFS

- DFS also runs in $O(|V| + |E|)$ time
- DFS is called exactly once per vertex
- Each adjacency list is used exactly once

	Implementation	Data Structure	Running Time
BFS	Iterative	Queue (FIFO)	$O(\|V\| + \|E\|)$
DFS	Recursive Iterative	(not explicitly required → execution stack) Stack (LIFO)	$O(\|V\| + \|E\|)$

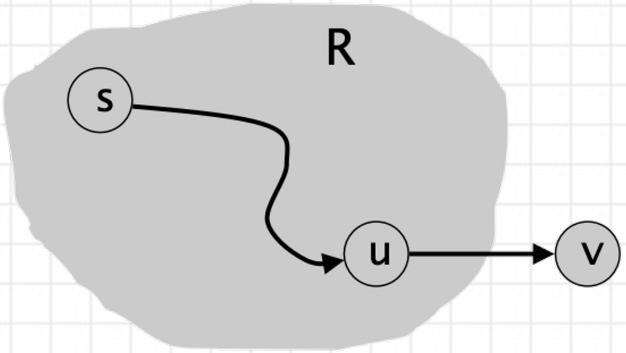
Graph

- Graph definition and representation
  - Adjacency matrix
  - Adjacency list
- Graph traversal
  - Breadth first search (BFS)
    - Shortest path (unweighted graphs)
    - Testing bipartiteness
    - Tree traversal (level-order)
    - Connected components
  - Depth first search (DFS)
    - Topological sorting
    - Tree traversal (in-order, pre-order, post-order)
    - Connected components
- Graph problems/algorithms
  - Minimum spanning tree (MST)
    - Kruskal (greedy)
    - Prim (greedy)
  - Shortest path (directed weighted graphs)
    - Dijkstra (greedy)
    - Bellman-Ford (dynamic programming)
    - Floyd-Warshall (dynamic programming)
  - Flow network
    - Max-flow min-cut theorem
    - Ford-Fulkerson algorithm

Graph Traversal: Connected Component

- <u>Connected component problem</u>. 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 ∈ R and v ∉ R
  Add v to R
Endwhile
```



it's safe to add v

- Upon termination, *R* is the connected component containing *s*.
  - BFS
  - DFS

References

- The lecture slides are mainly based on the suggested textbooks and the corresponding published lecture notes:

  - CLRS: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.
  - KT: Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.
  - DPV: Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Algorithms, McGraw-Hill Higher Education., 2008.
  - Slides by Kevin Wayne. Copyright © 2005 Pearson-Addison Wesley.