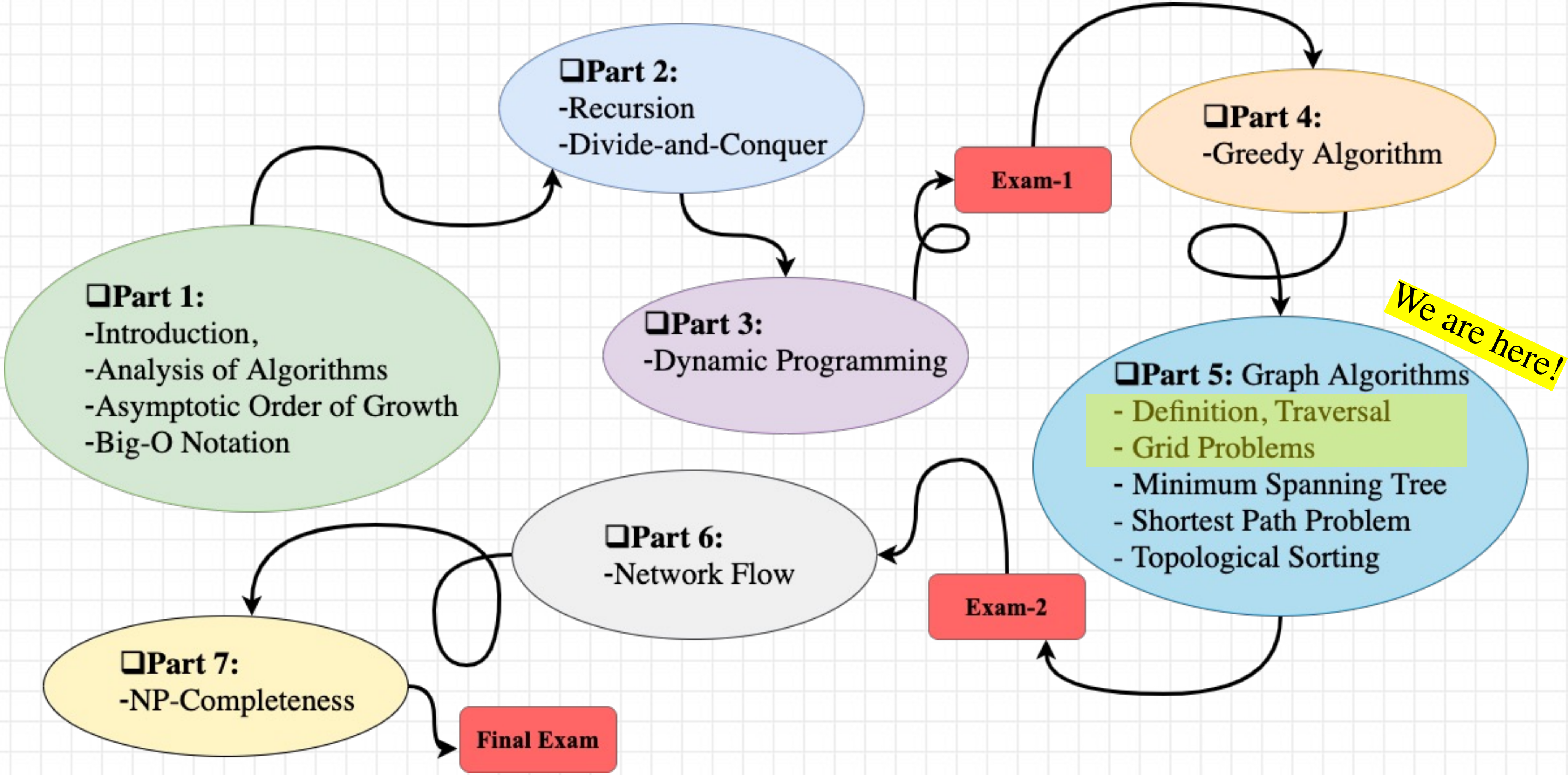
CS-3510: Design and Analysis of Algorithms

Graph Algorithms: Traversal Applications I

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)

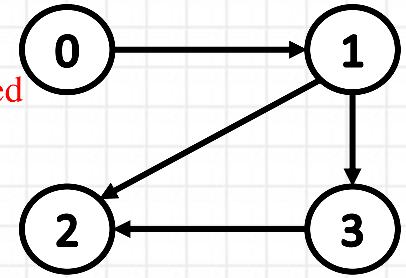
Graph Properties and Terminology Review

```python
# adjacency matrix:

graph1 = [
      [0, 1, 0, 0],
      [0, 0, 1, 1],
      [0, 0, 0, 0],
      [0, 0 ,1, 0]
]

graph2 = [
      [0, 10,  0,  0],
      [0,  0, 15, 20],
      [0,  0,  0,  0],
      [0,  0, 25,  0]
]

graph3 = [
      [0, 1, 0, 0],
      [1, 0, 1, 1],
      [0, 1, 0, 1],
      [0, 1 ,1, 0]
]

graph4 = [
      [ 0, 10,  0,  0],
      [10,  0, 15, 20],
      [ 0, 15,  0, 25],
      [ 0, 20, 25,  0]
]
```

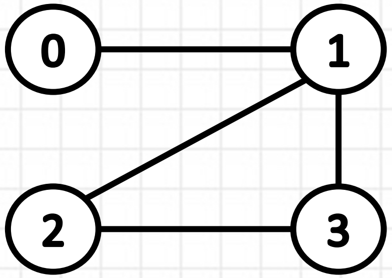
```python
# adjacency list --> dictionary/hashmap

graph1 = {
    0: [1],
    1: [2, 3],
    2: [],
    3: [2]
}

graph2 = {
    0: {1: 10},
    1: {2: 15, 3: 20},
    2: {},
    3: {2: 25}
}

graph3 = {
    0: [1],
    1: [0, 2, 3],
    2: [1, 3],
    3: [1, 2]
}

graph4 = {
    0: {1: 10},
    1: {1: 10, 2: 15, 3: 20},
    2: {1: 15, 3: 25},
    3: {1: 20, 2: 25}
}
```

💻 Demo code time!

Directed
Unweighted



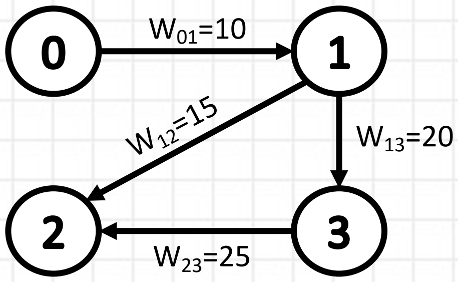
Graph-1



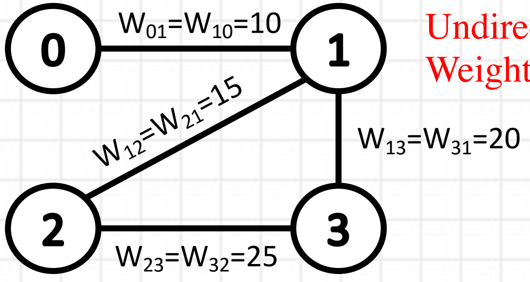
Undirected
Unweighted

Graph-3

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

- 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 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

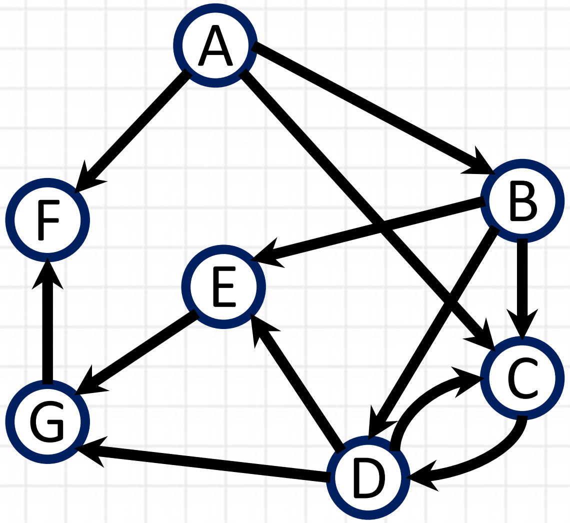
- Traversal = Exploring = Searching
- A graph needs to be traversed in order to determine some properties

- 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

	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



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

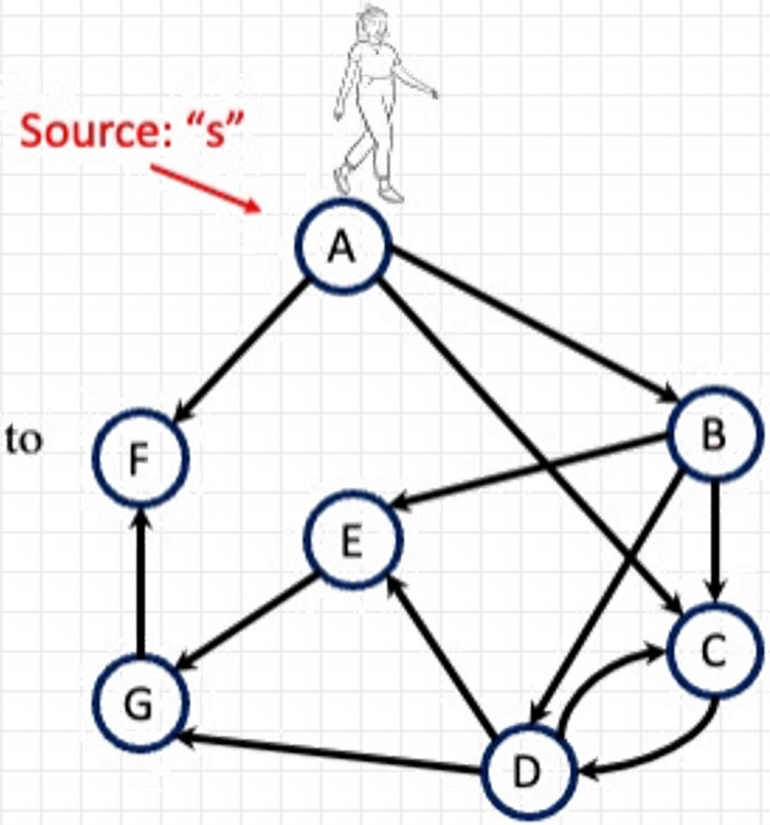
gray := visited node

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

Graph Traversal: BFS

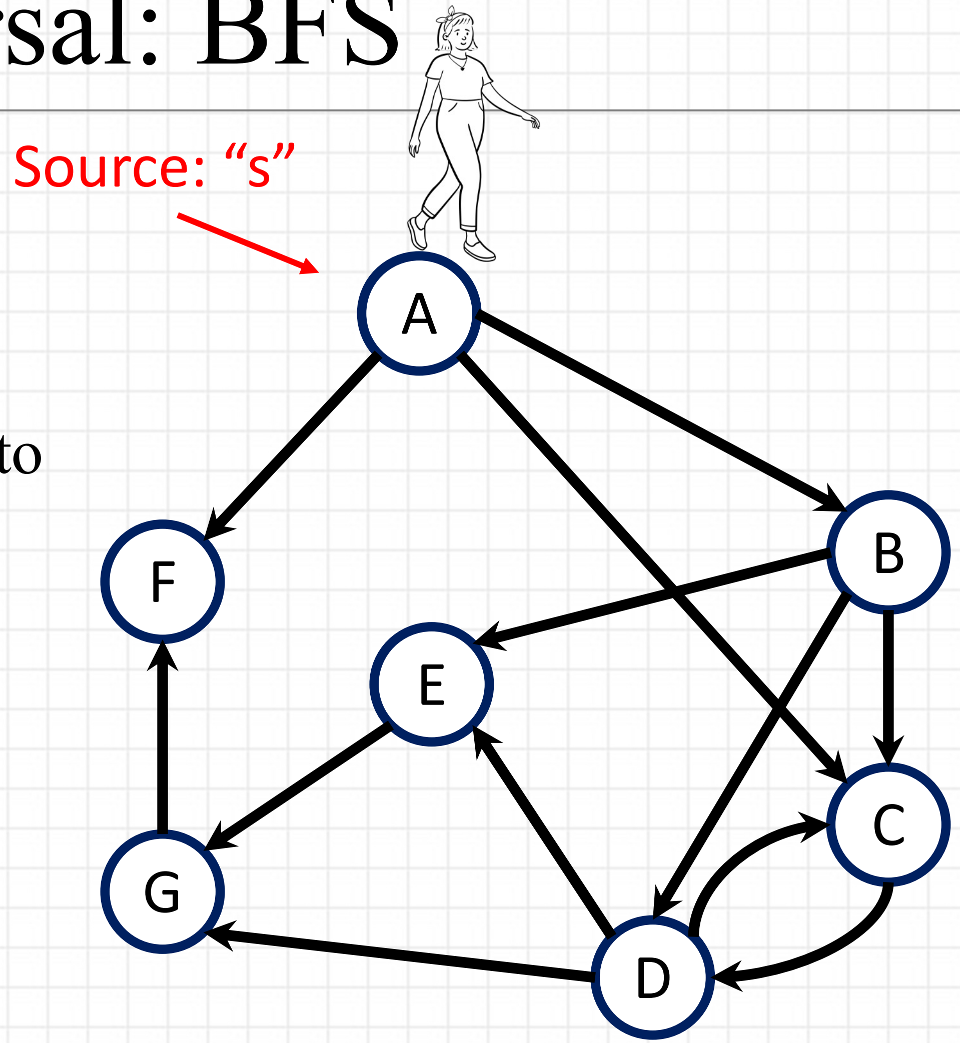
- An efficient graph traversal procedure
- BFS starts from a source vertex “s”
- At each vertex u, all neighbors, i.e., vertices v adjacent to u are visited before moving on to vertices adjacent to some v

- 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 = {}
- 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)

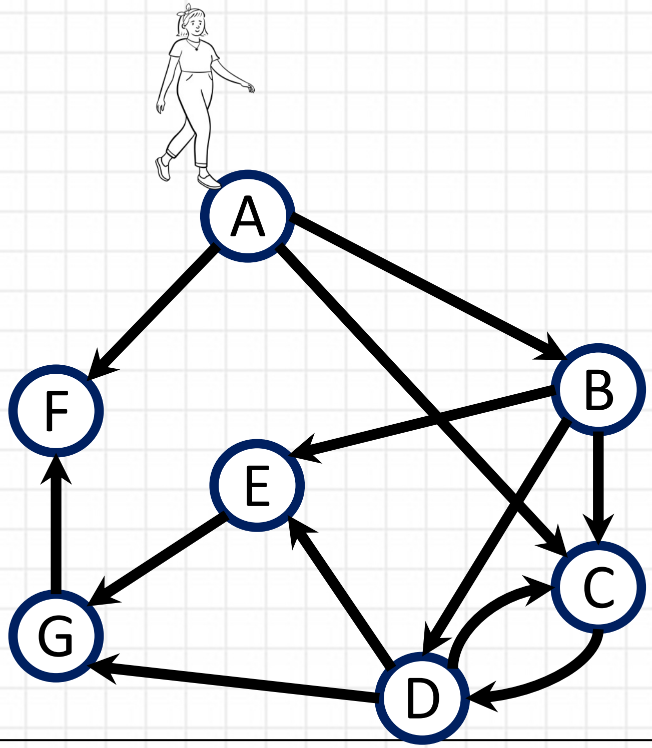
Graph Traversal: DFS

- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path.
- Traversal = Exploring = Searching (visiting vertices one-by-one)
- Analogy:

Exploring a maze:
"Visited" set → A piece of chalk
"Stack" → ball of string
  Push: unwind the string to try new path
  Pop: rewind the string to return to previous junction

Graph Traversal: DFS

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

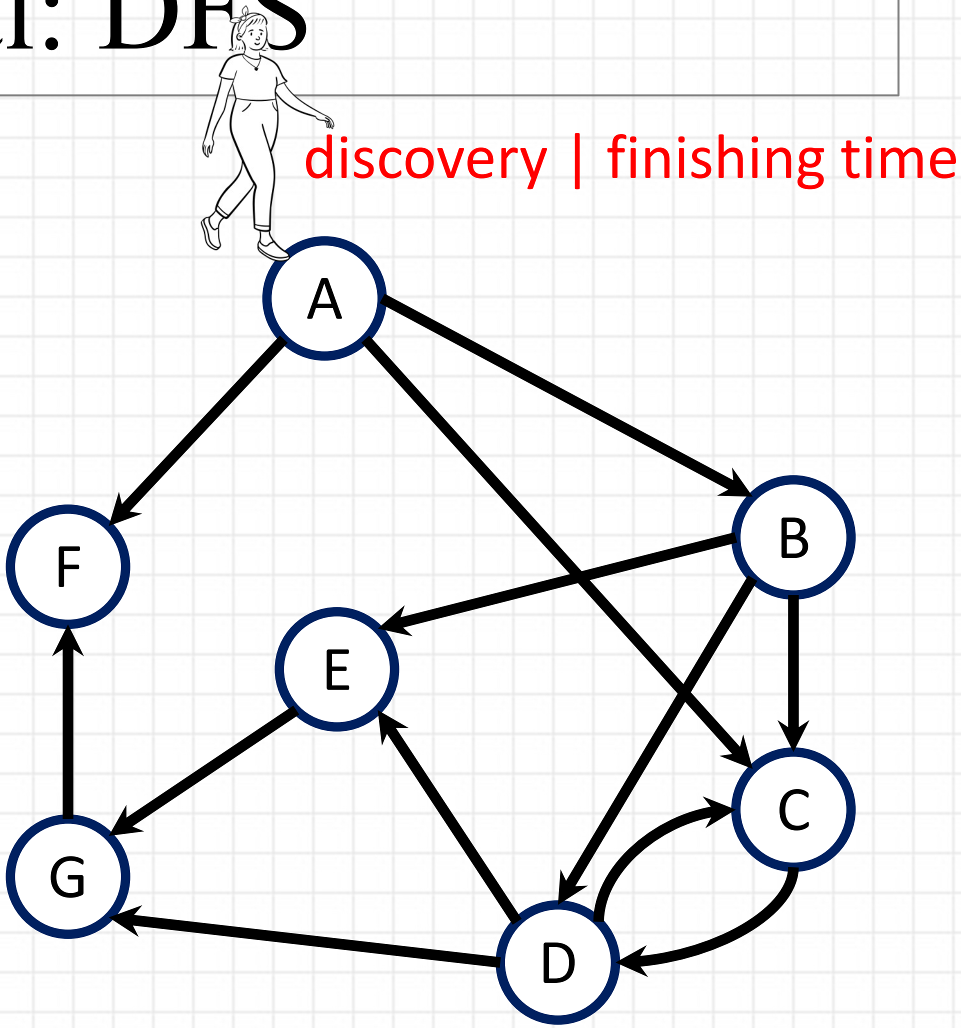



```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
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

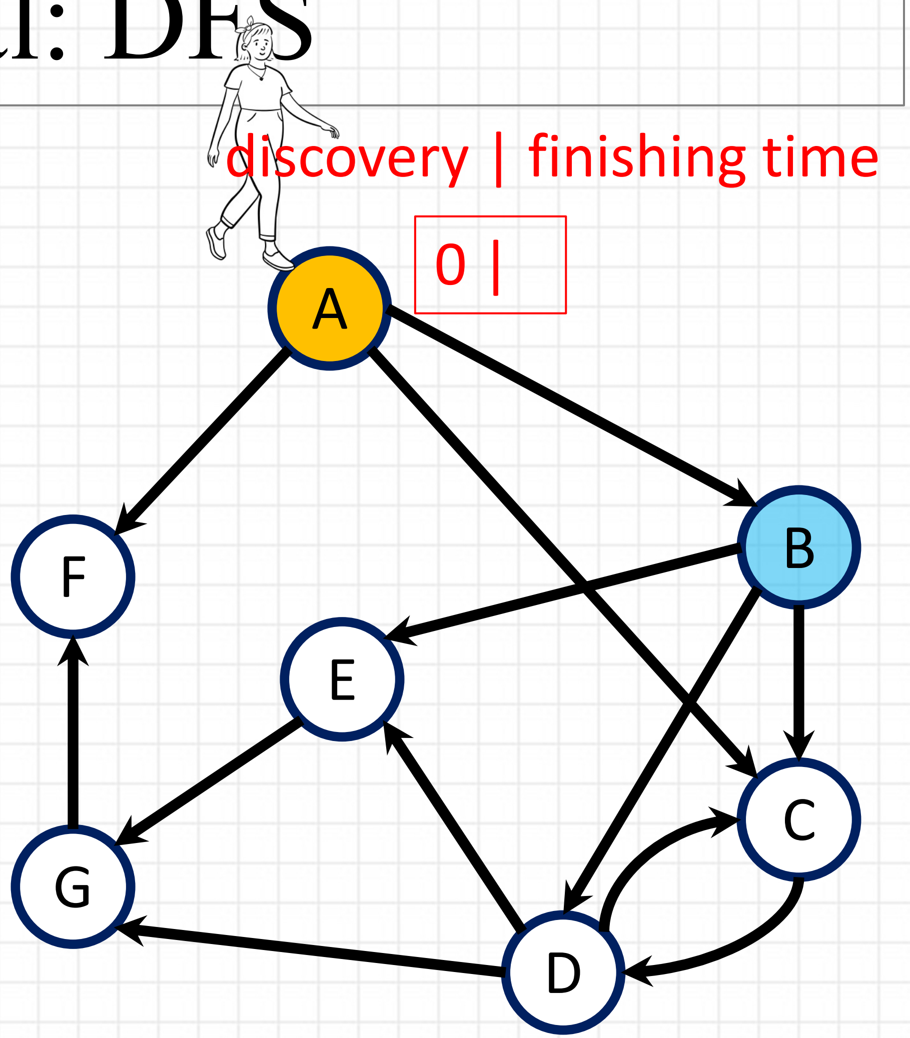
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



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

Graph Traversal: DFS

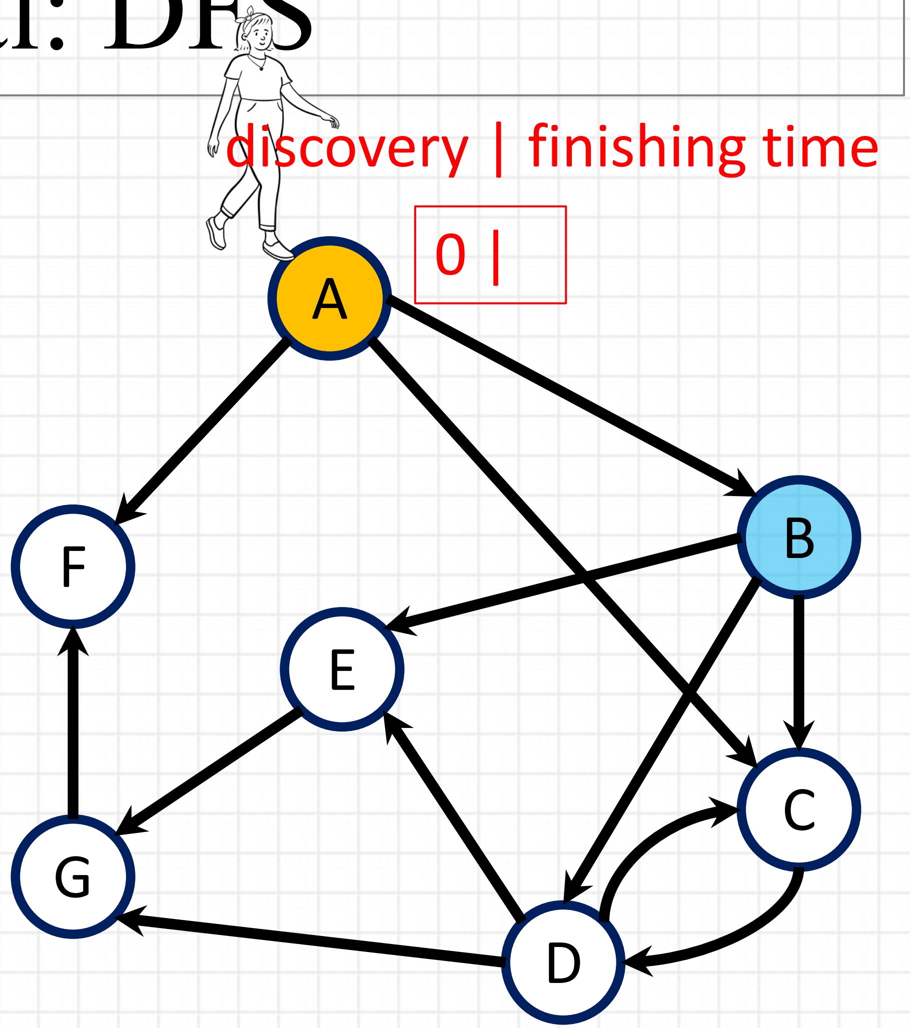
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



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

Graph Traversal: DFS

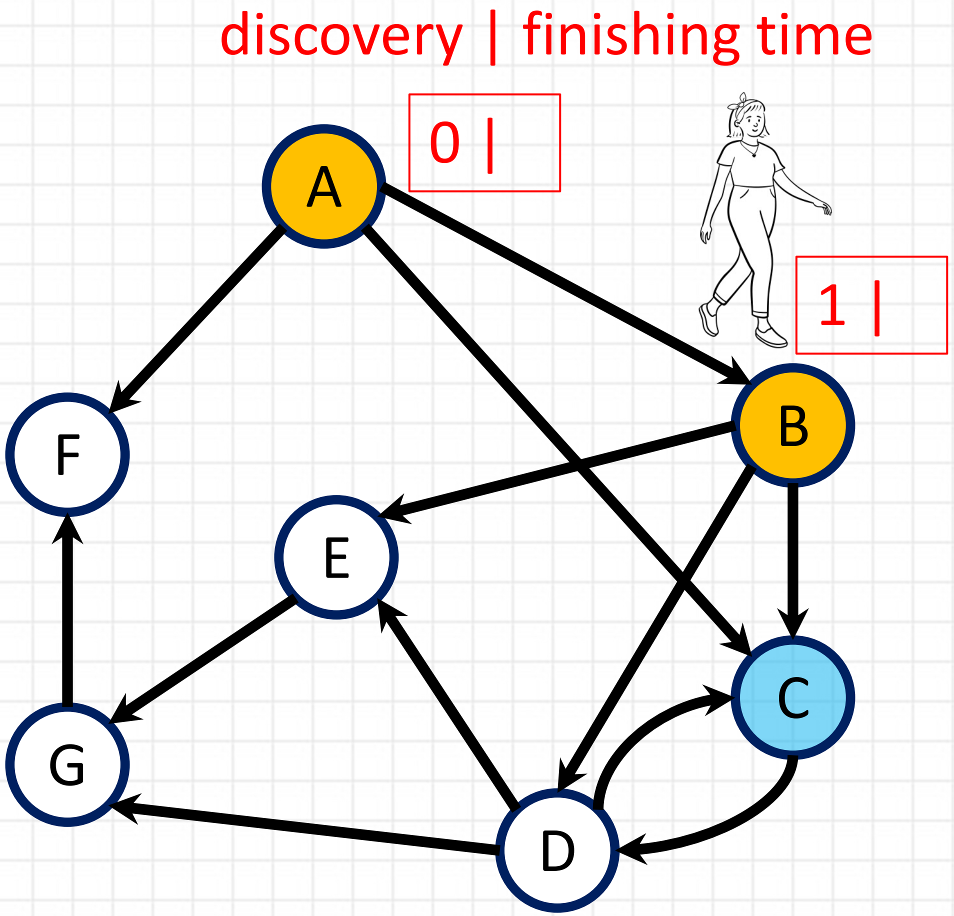
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



- 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

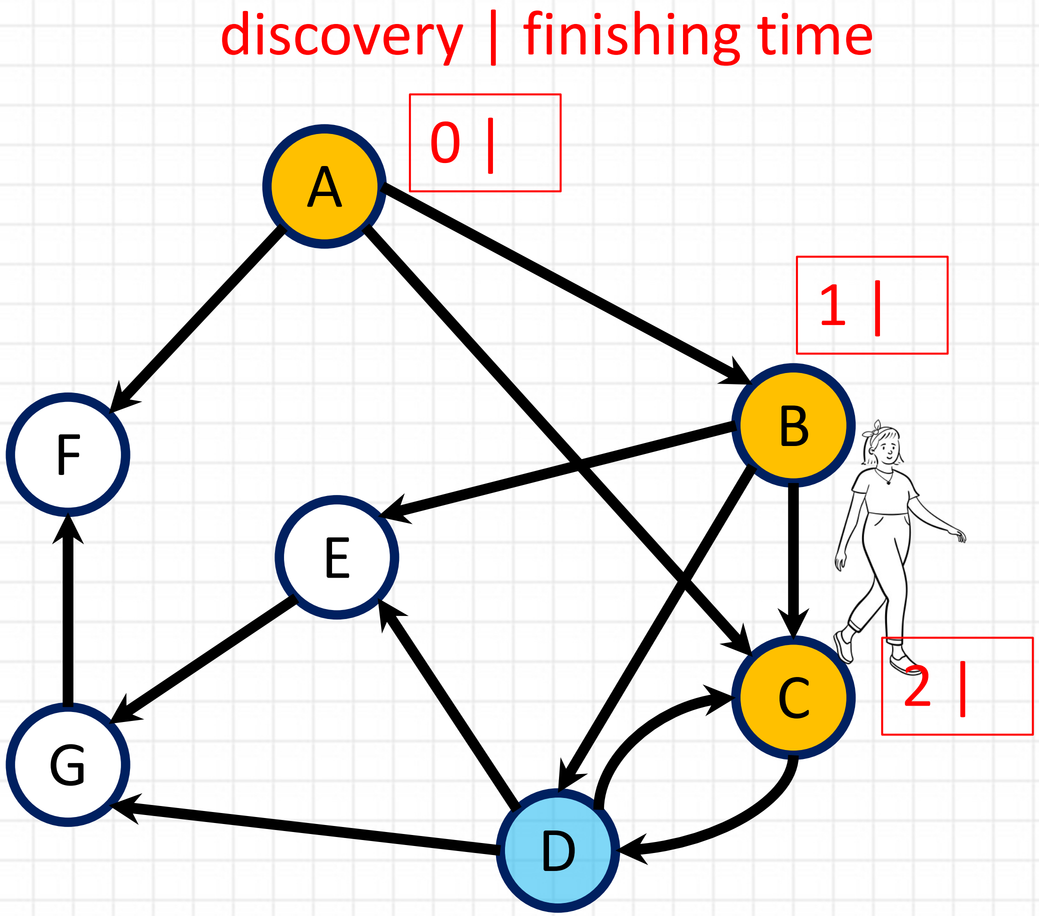


- Stack = {A, B, C}
- Visited = {A, B}

Graph Traversal: DFS

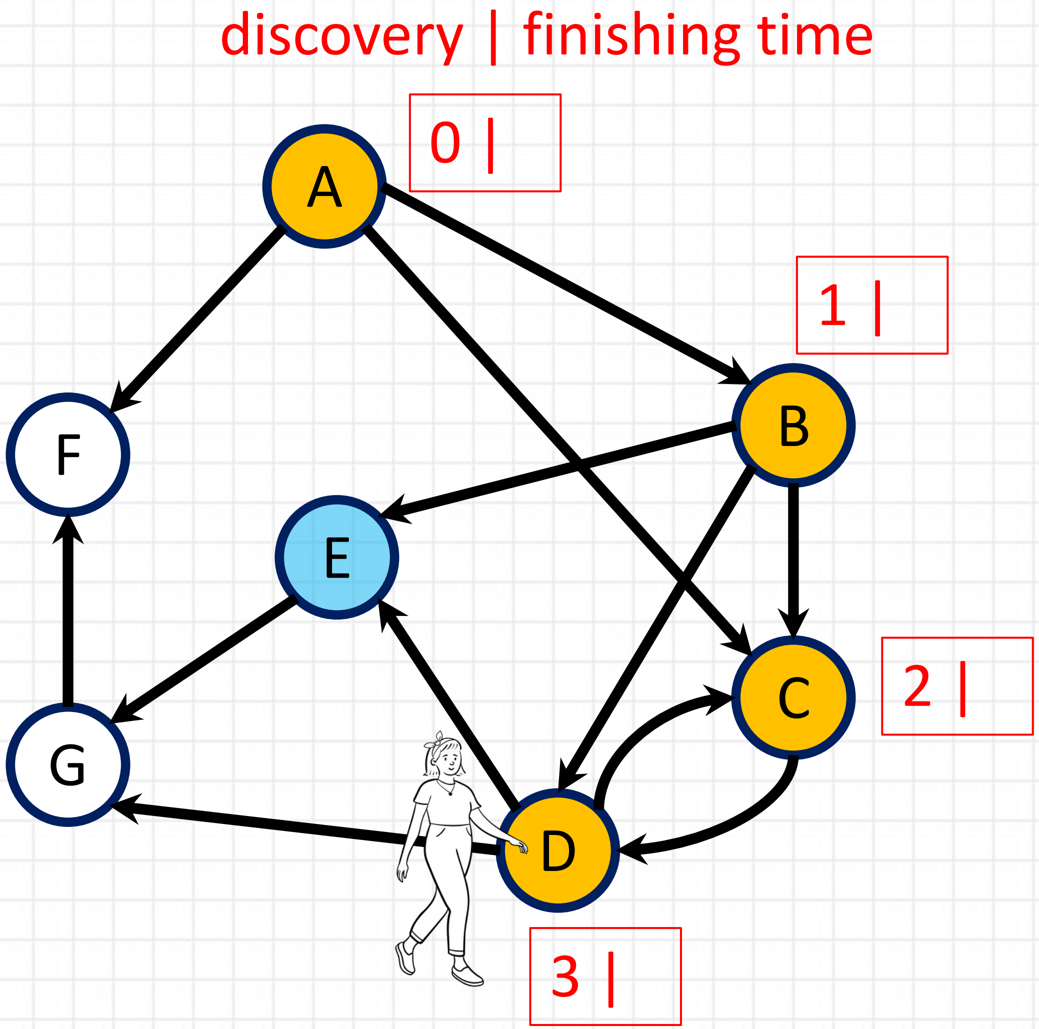
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

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



Graph Traversal: DFS

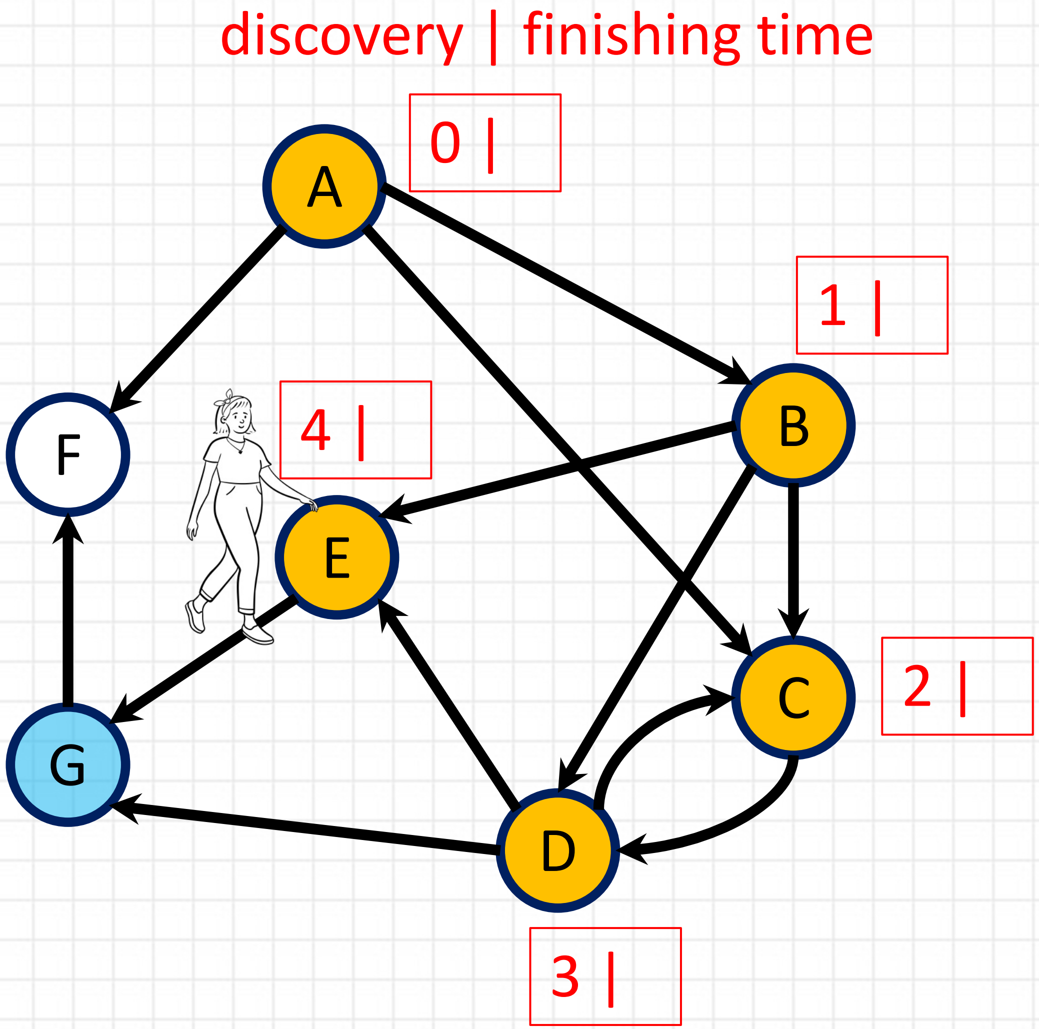
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path



- 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

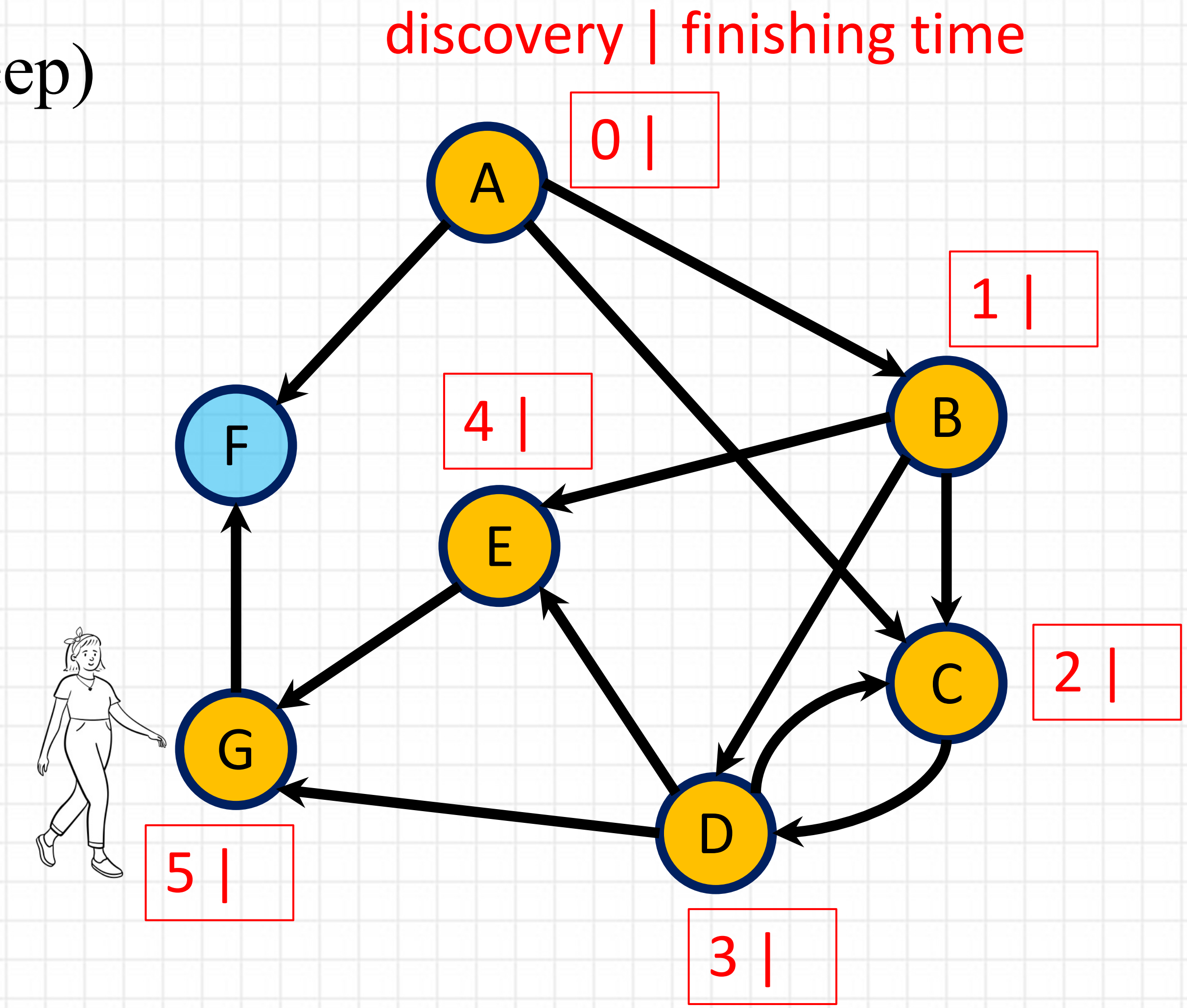


- Stack = {A, B, C, D, E, G}
- Visited = {A, B, C, D, E}

Graph Traversal: DFS

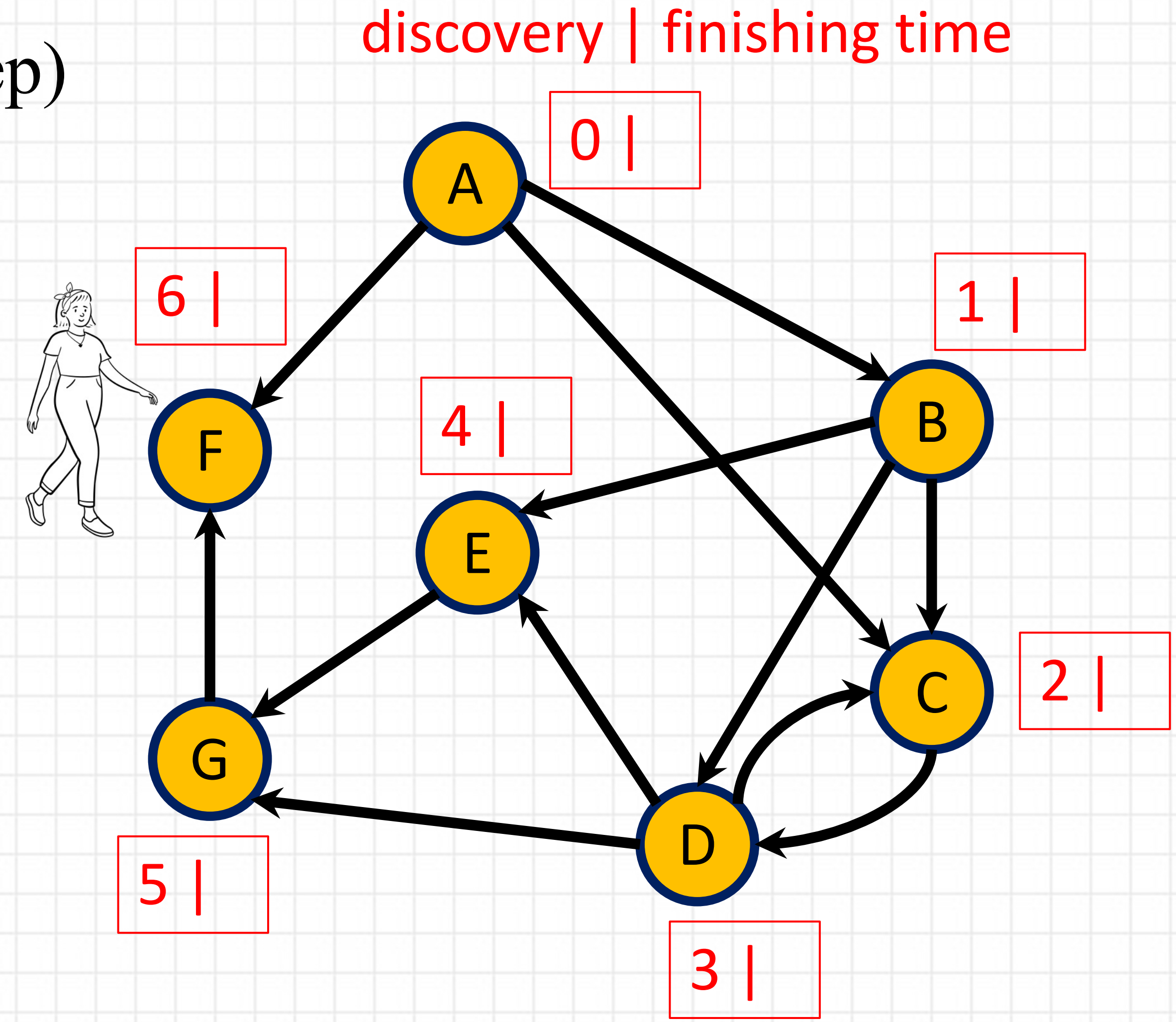
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

- 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}

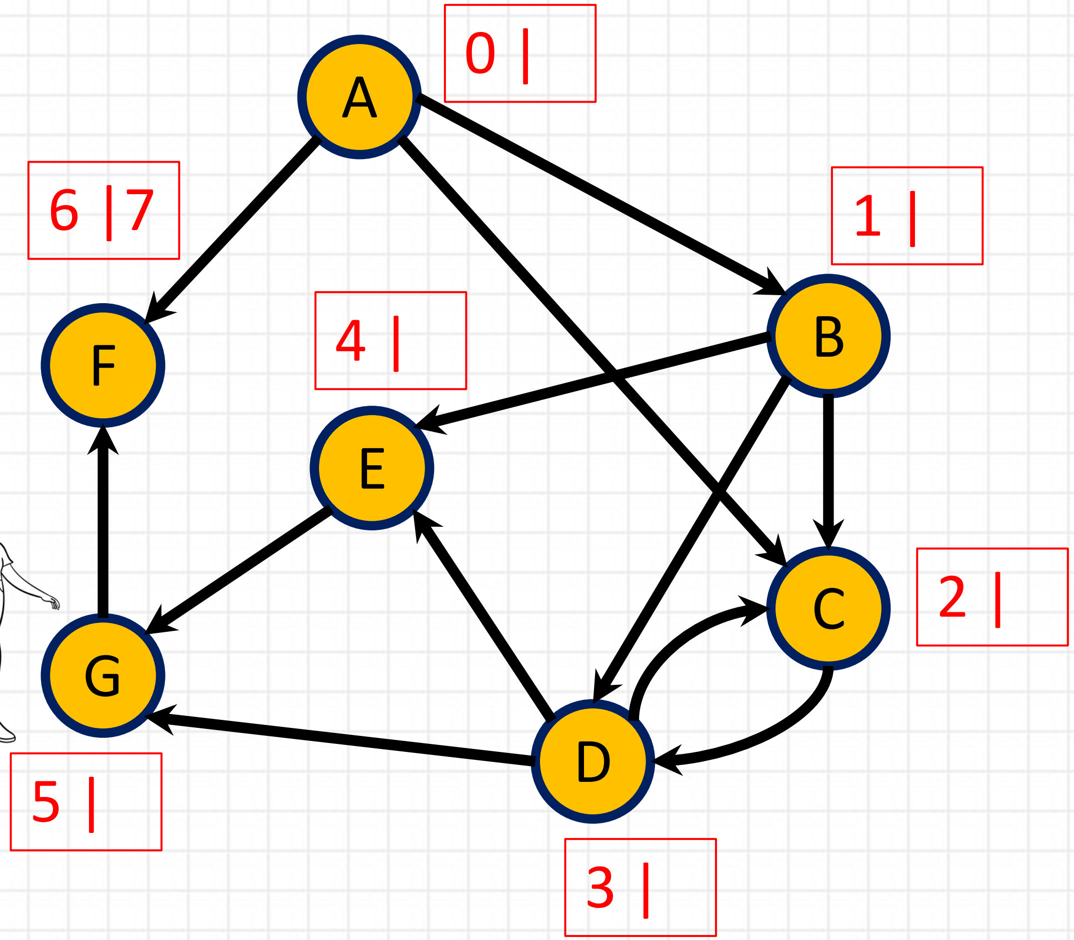
Graph Traversal: DFS

- 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

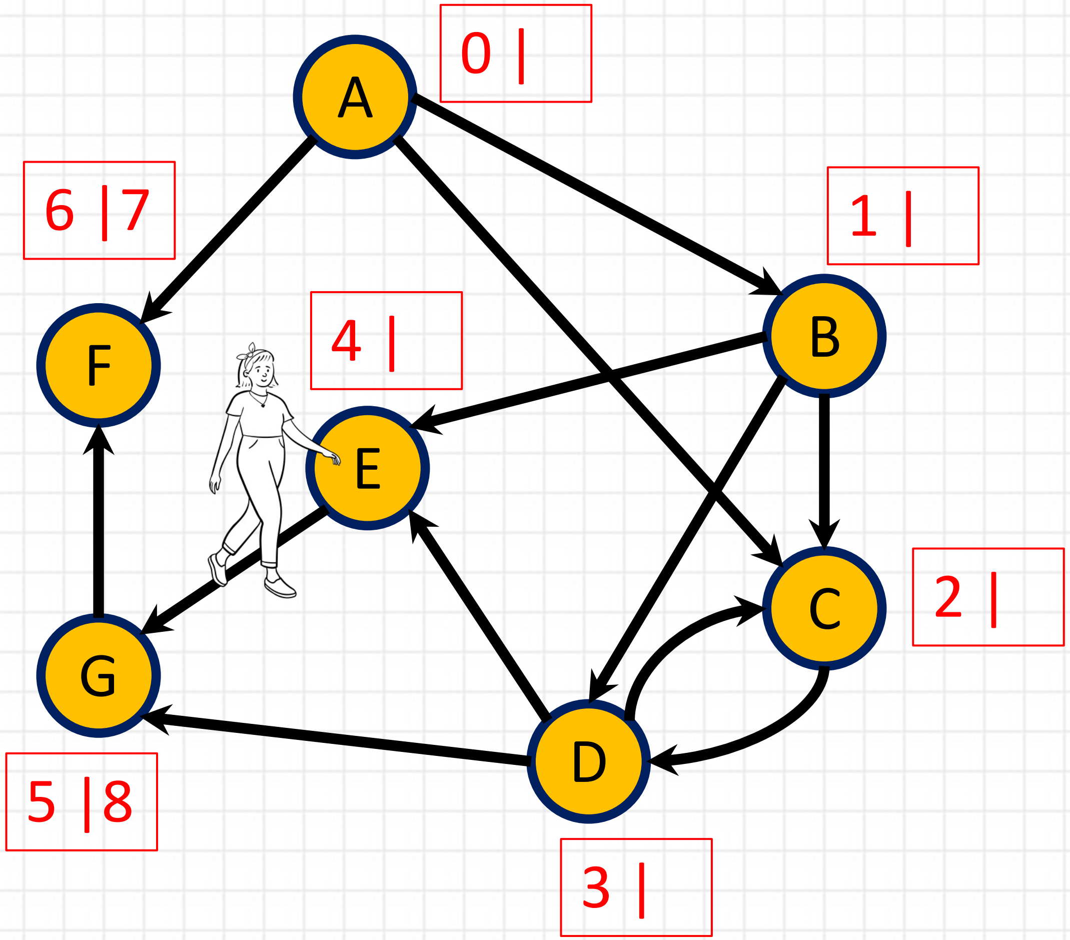
discovery | finishing time



Graph Traversal: DFS

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

discovery | finishing time



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

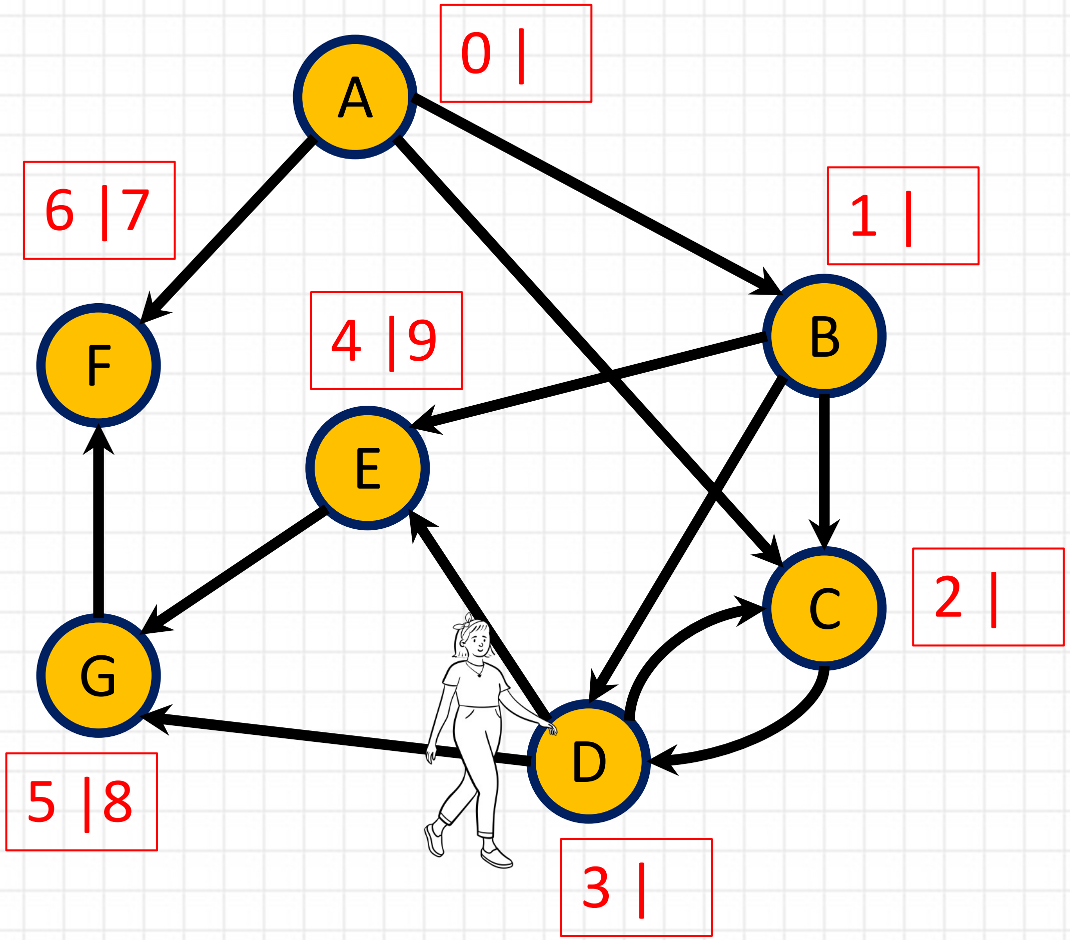
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path

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



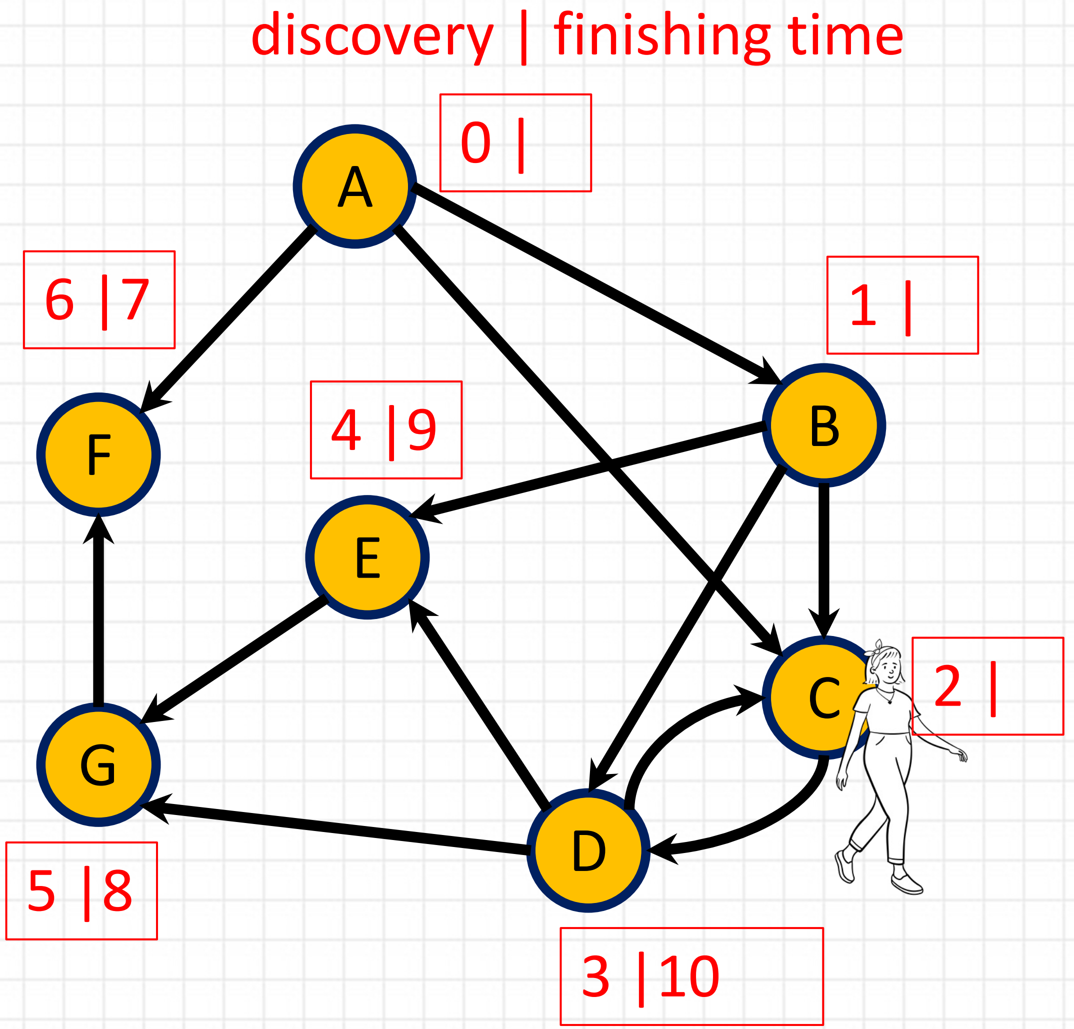
Graph Traversal: DFS

- 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}

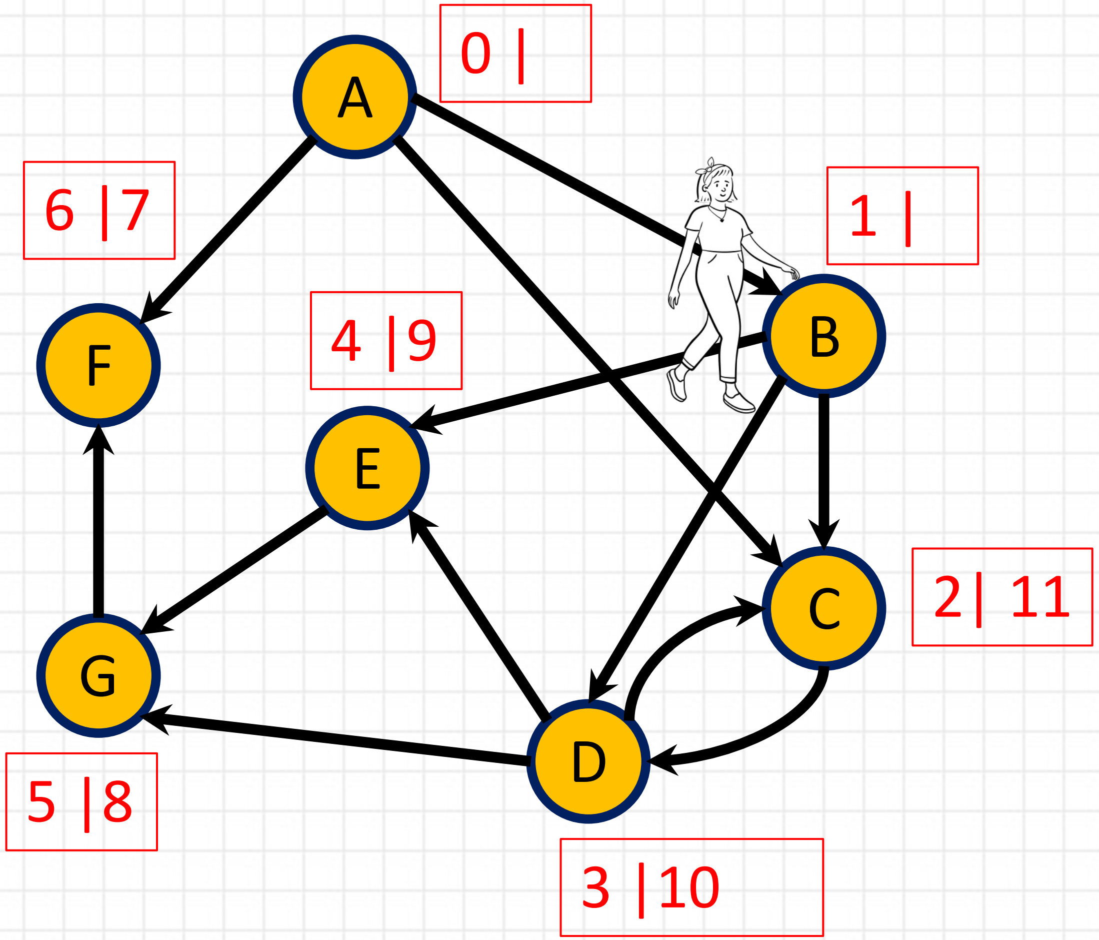
- No more path to explore→ backtrack



Graph Traversal: DFS

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

discovery | finishing time



Pop

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

- No more path to explore→ backtrack

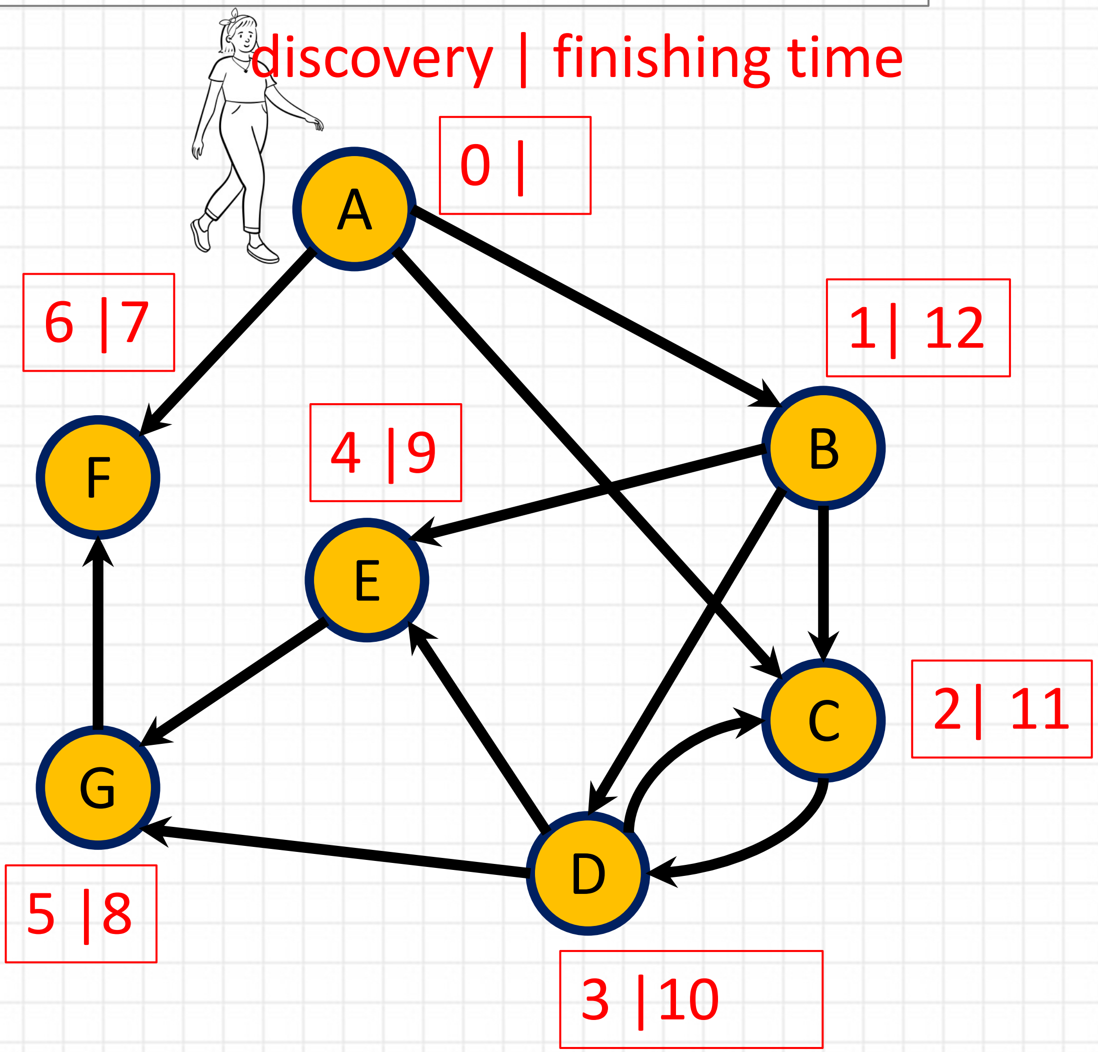
Graph Traversal: DFS

- 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

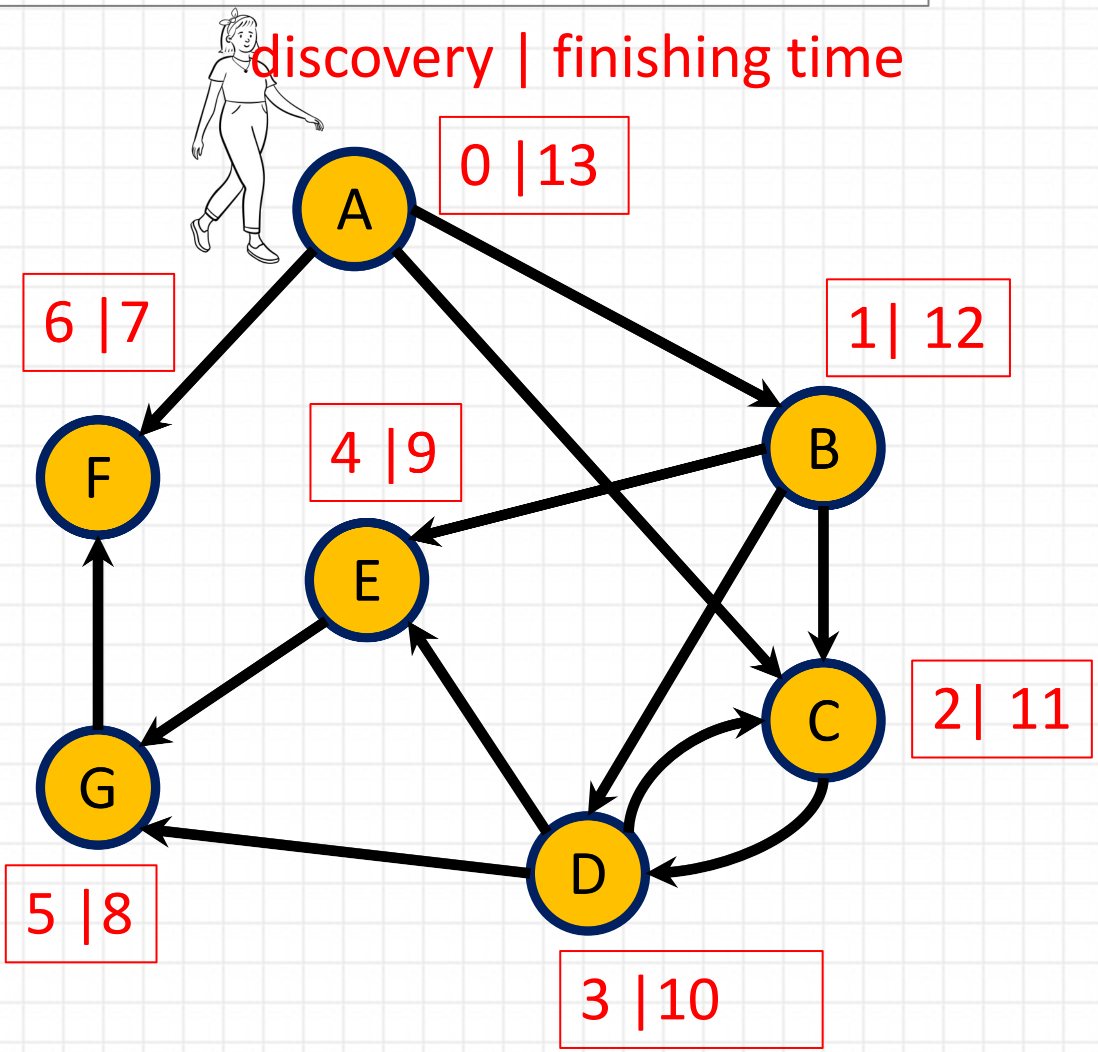


Graph Traversal: DFS

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

Pop

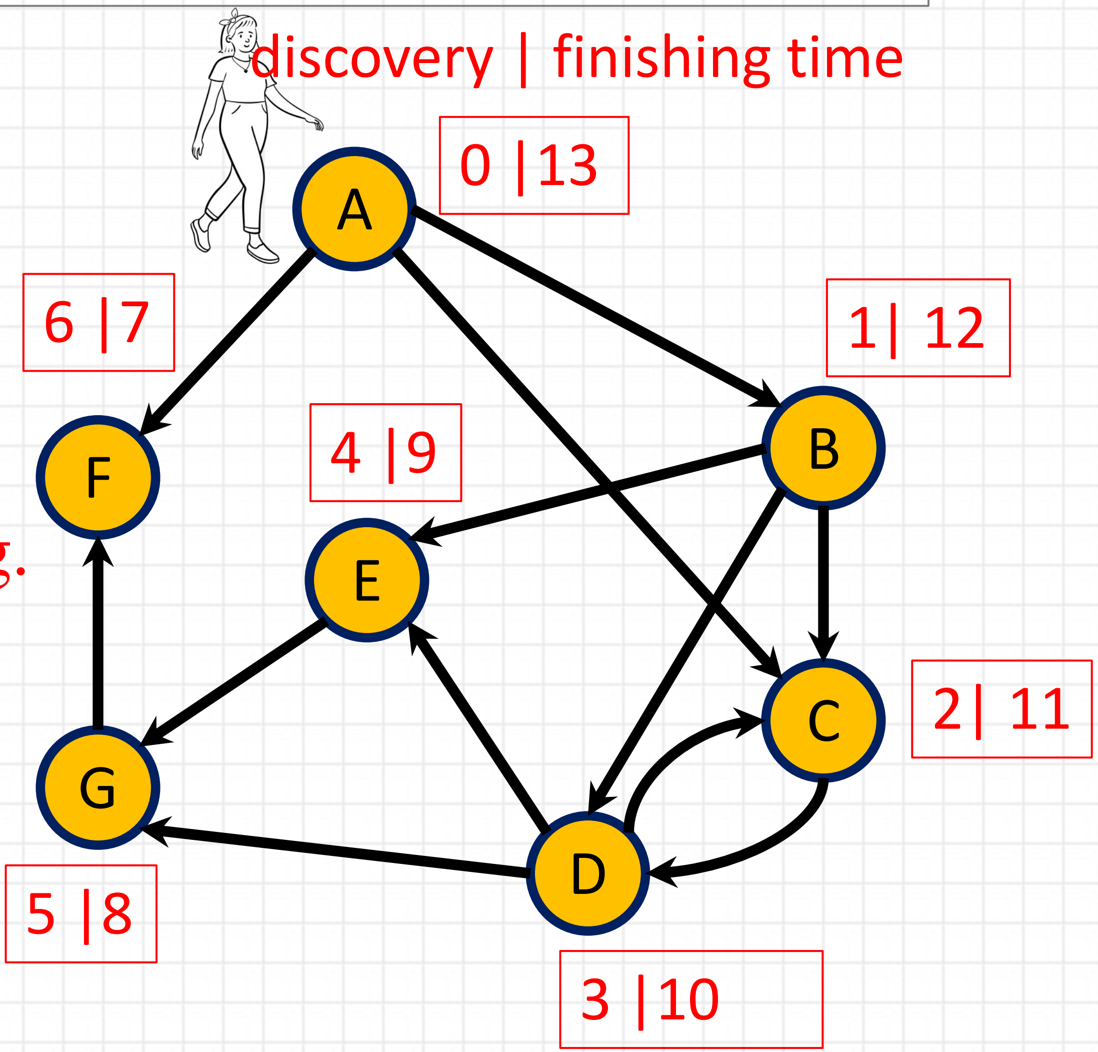
- Stack = {~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~E~~, ~~G~~, ~~F~~}
- Visited = {A, B, C, D, E, G, F}
- No more path to explore→ backtrack
- No more element in the stack → Halt



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!

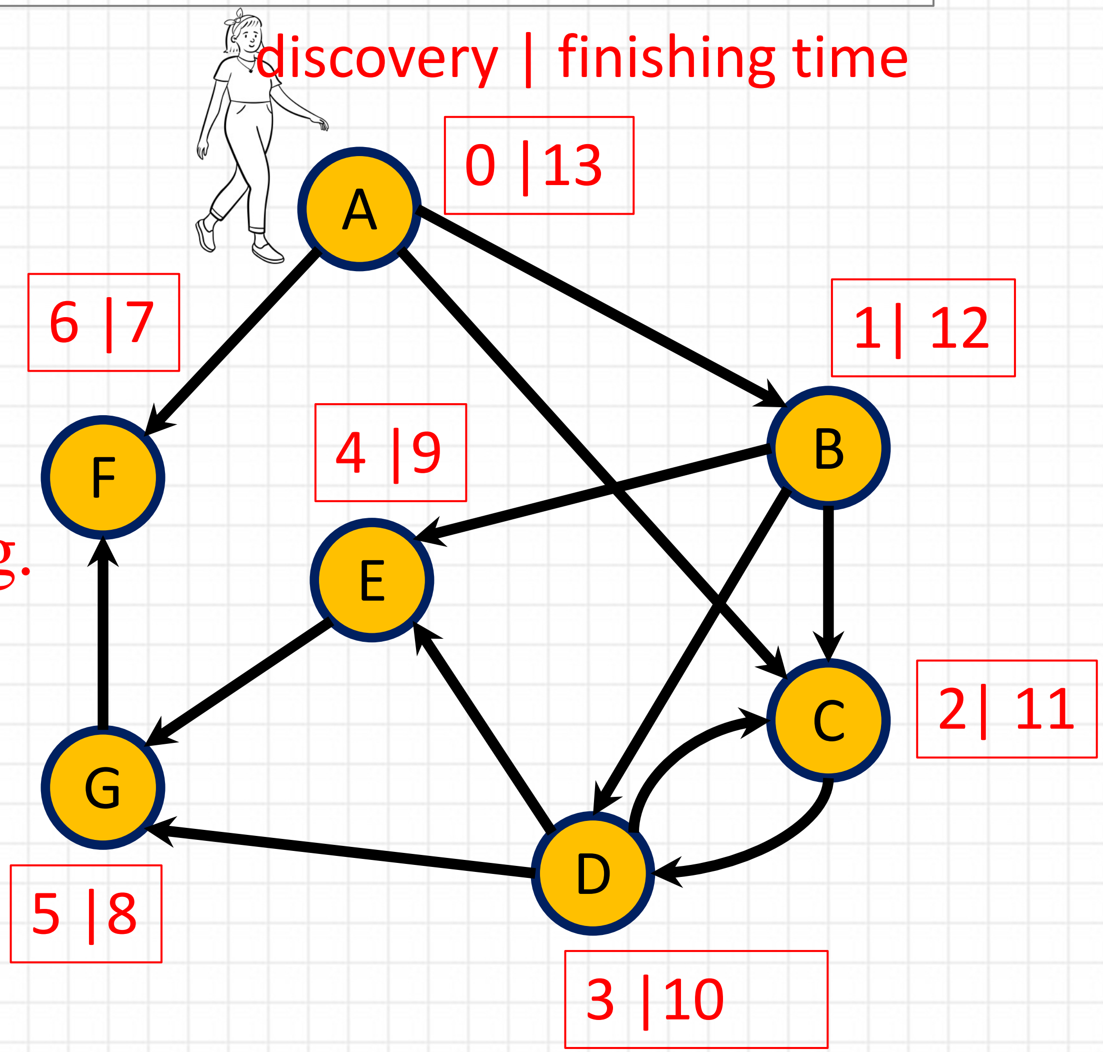


Graph Traversal: DFS

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

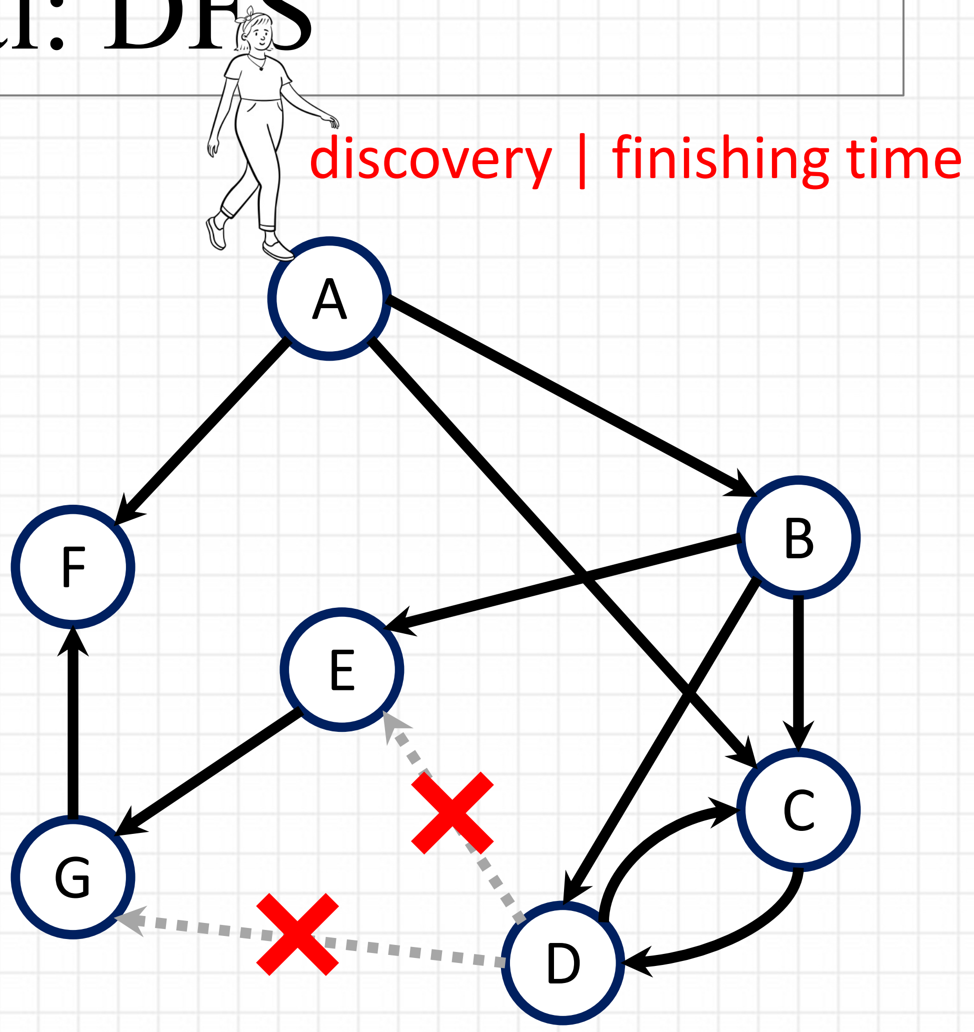
- Note in this example we were able to reach all nodes without any backtracking. But this is not usually the case in many examples!

- → Consider the same example, with minor difference:



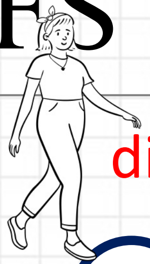
Graph Traversal: DFS

- 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 →



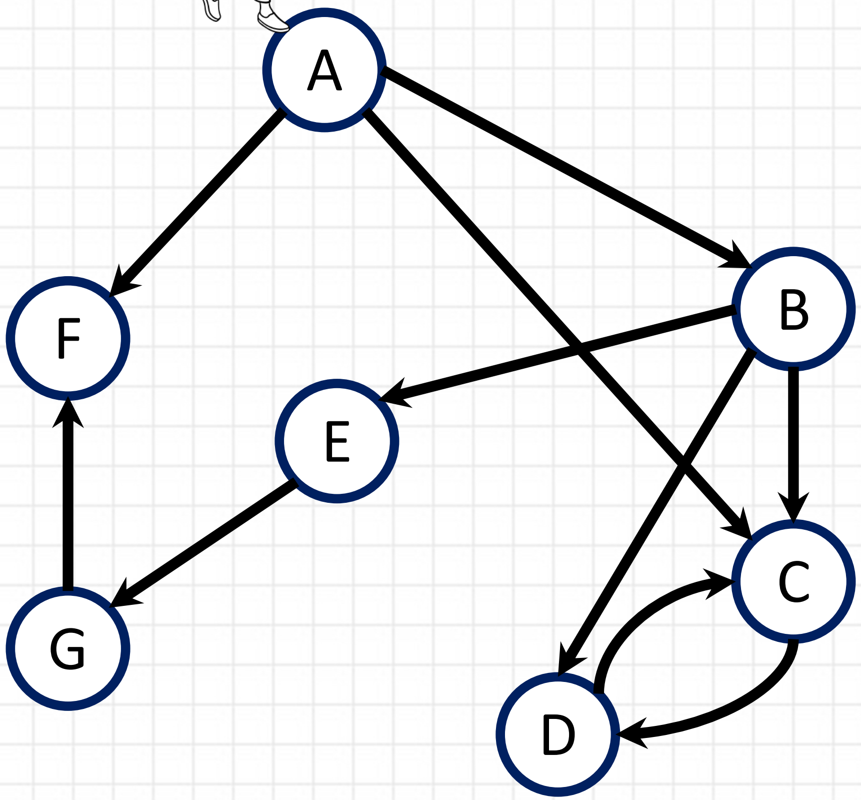
- Stack = {A}
- Visited = {A}

Graph Traversal: DFS



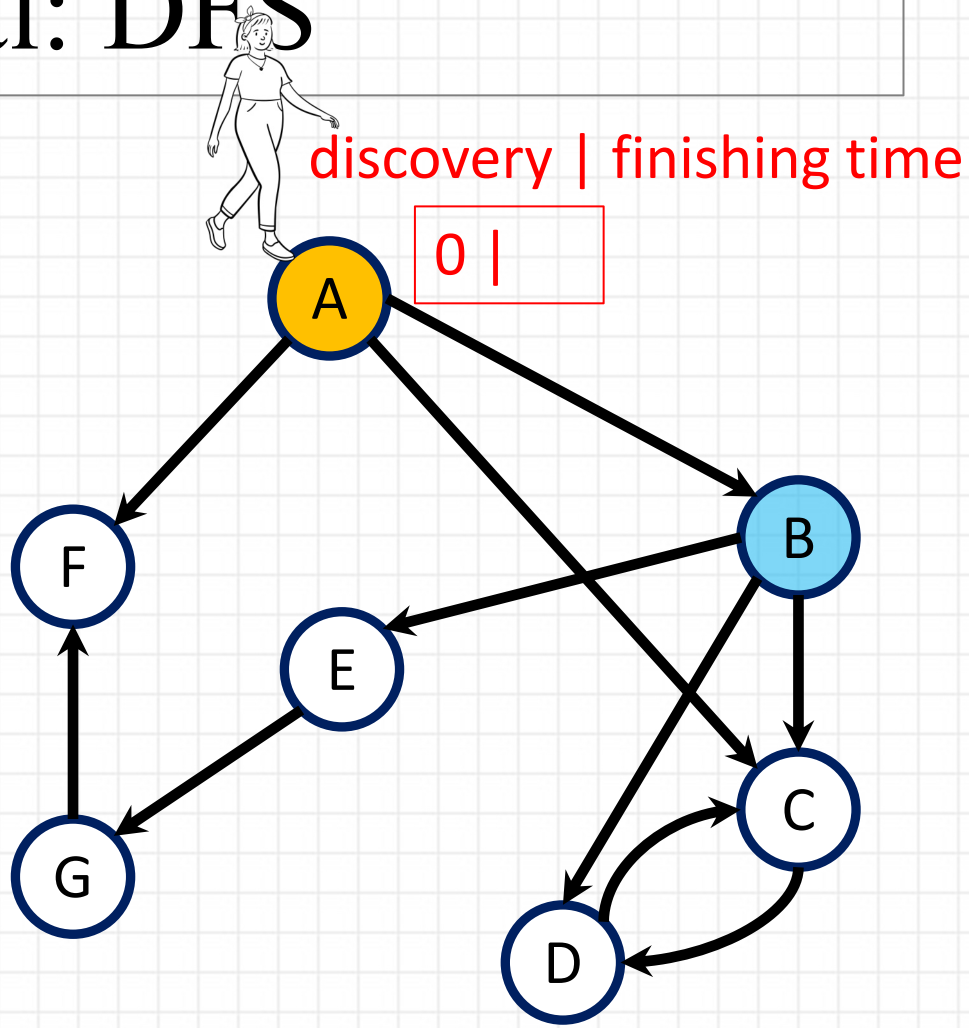
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →
- Stack = {A}
- Visited = {A}

discovery | finishing time



Graph Traversal: DFS

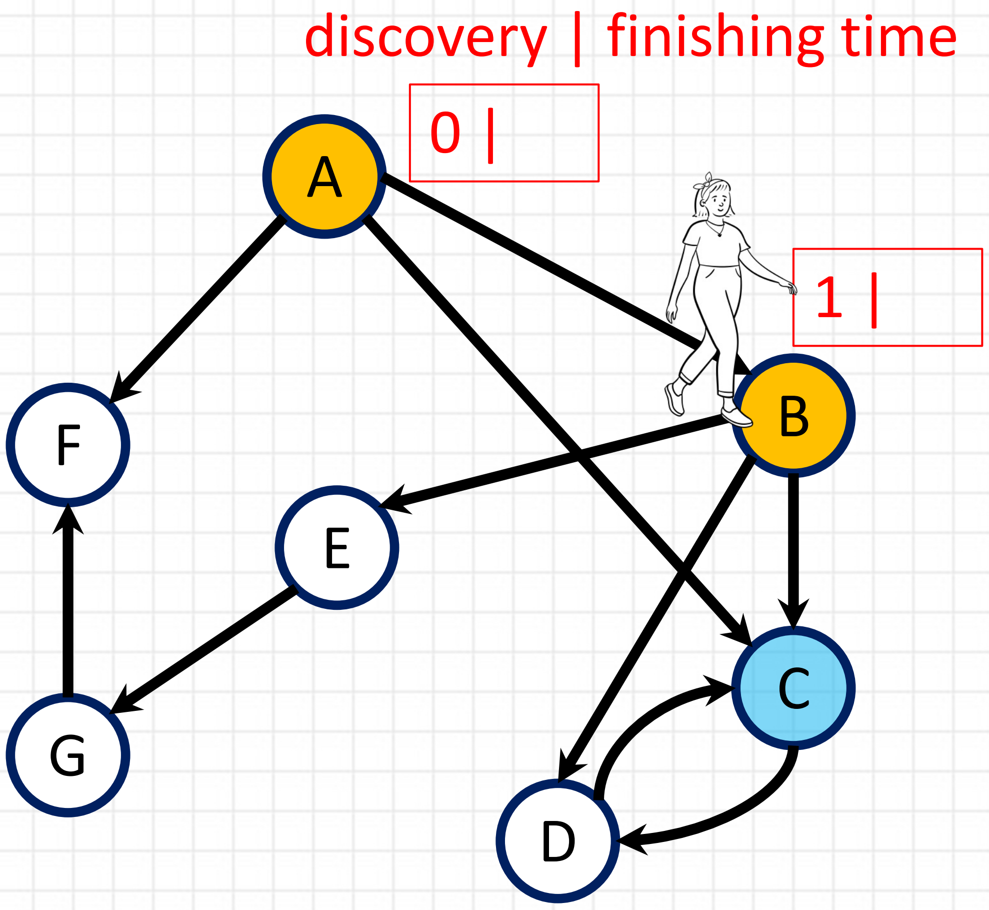
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →
- Stack = {A, B}
- Visited = {A}



Graph Traversal: DFS

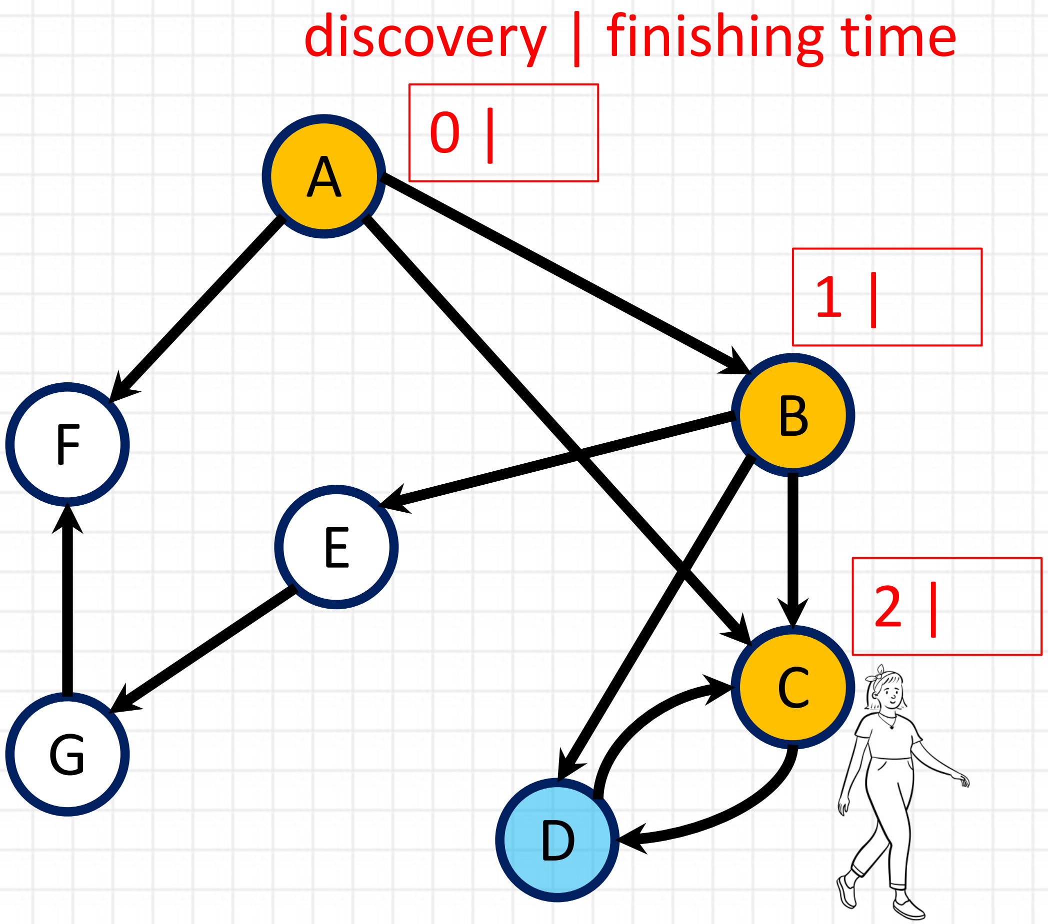
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

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



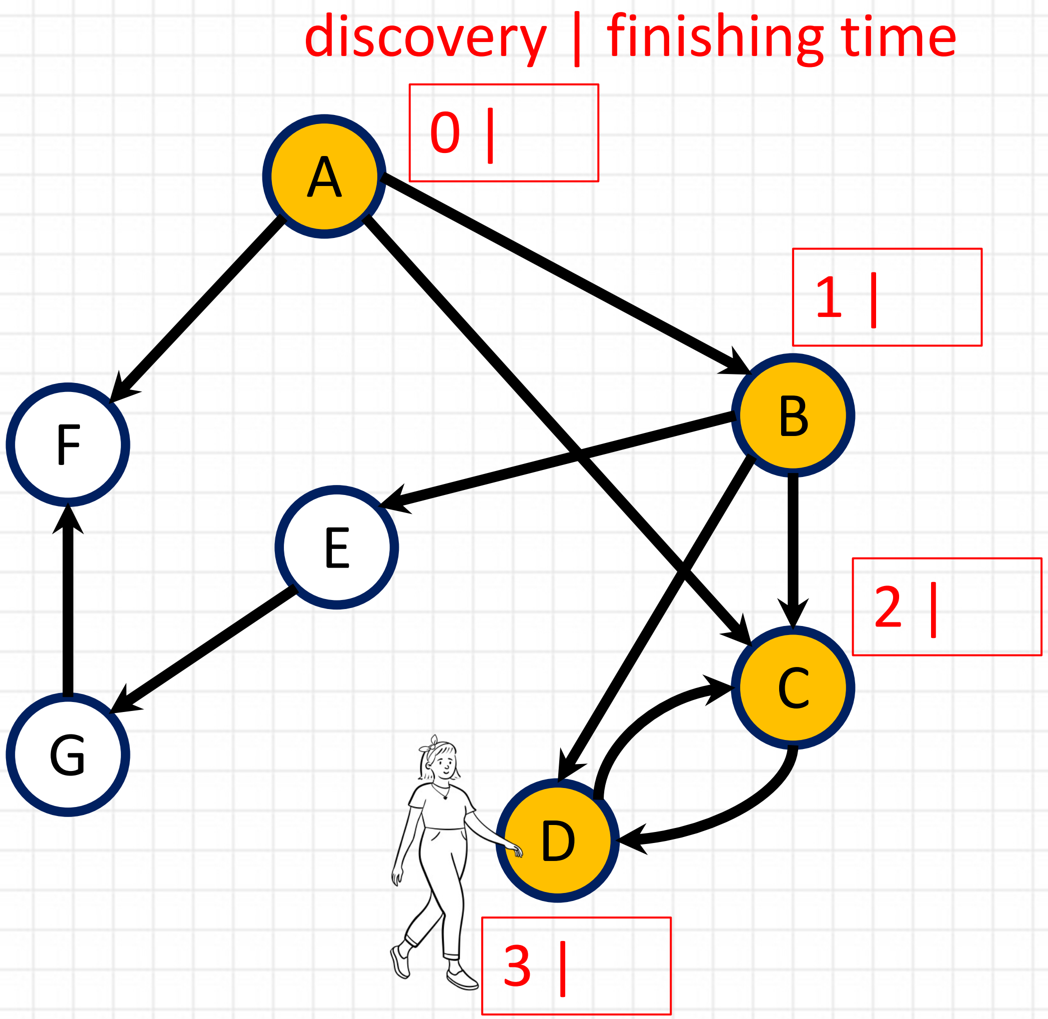
Graph Traversal: DFS

- 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 →

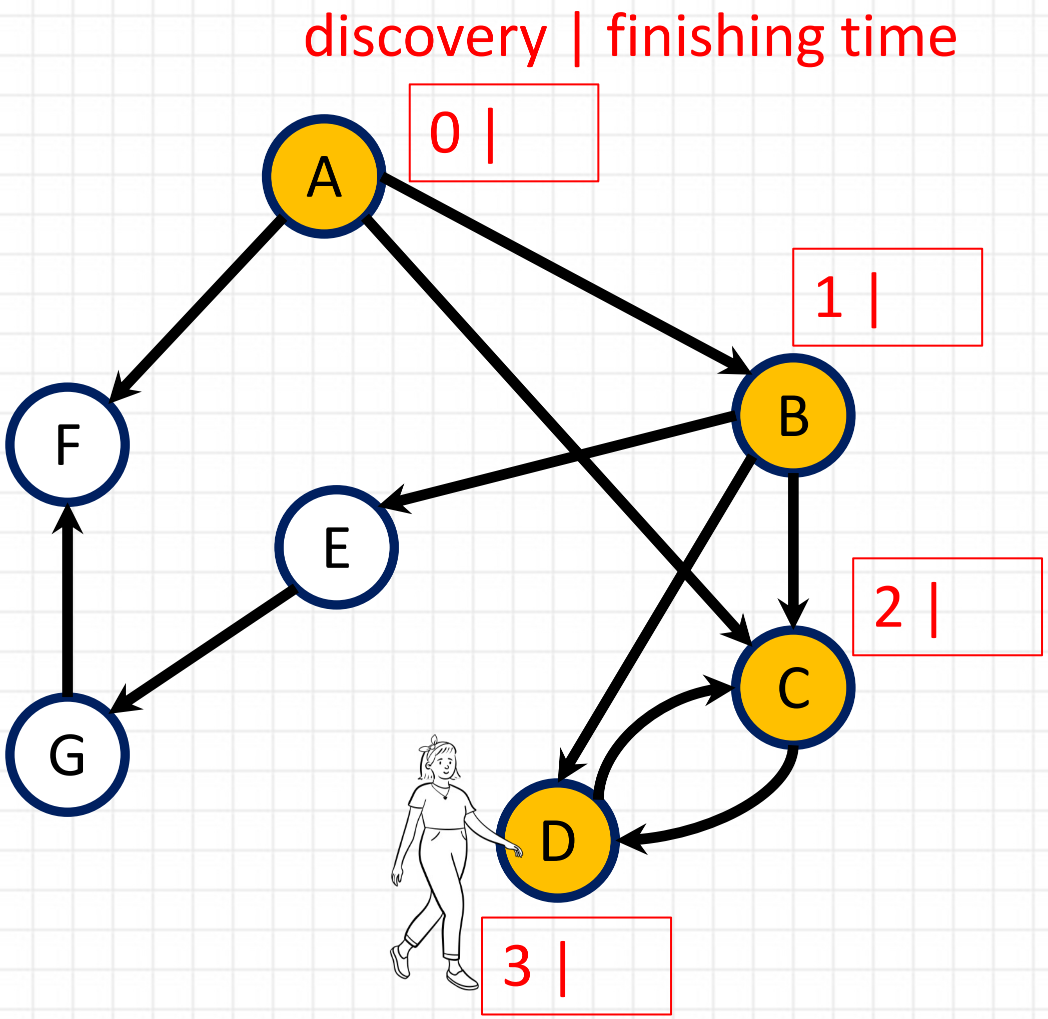


- Stack = {A, B, C, D}
- Visited = {A, B, C, D}

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

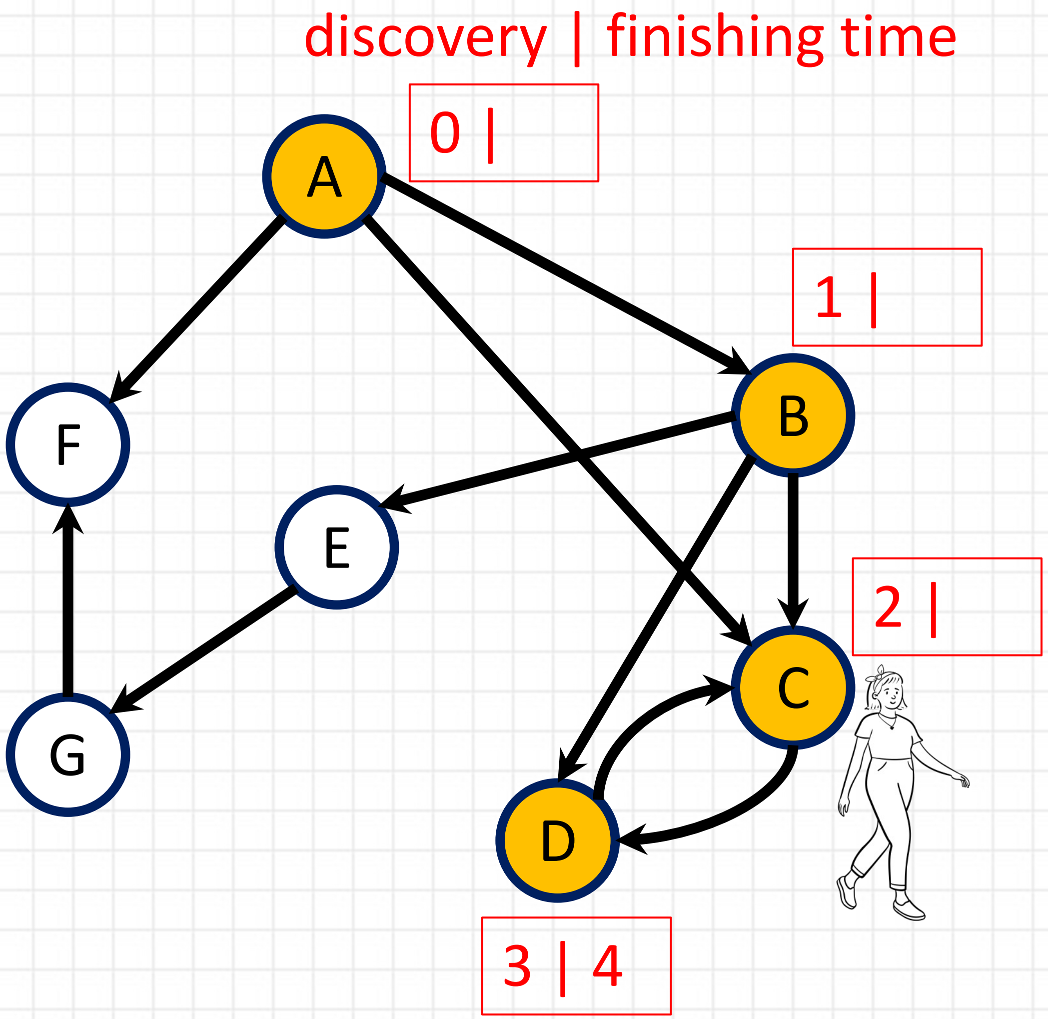


Graph Traversal: DFS

- 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

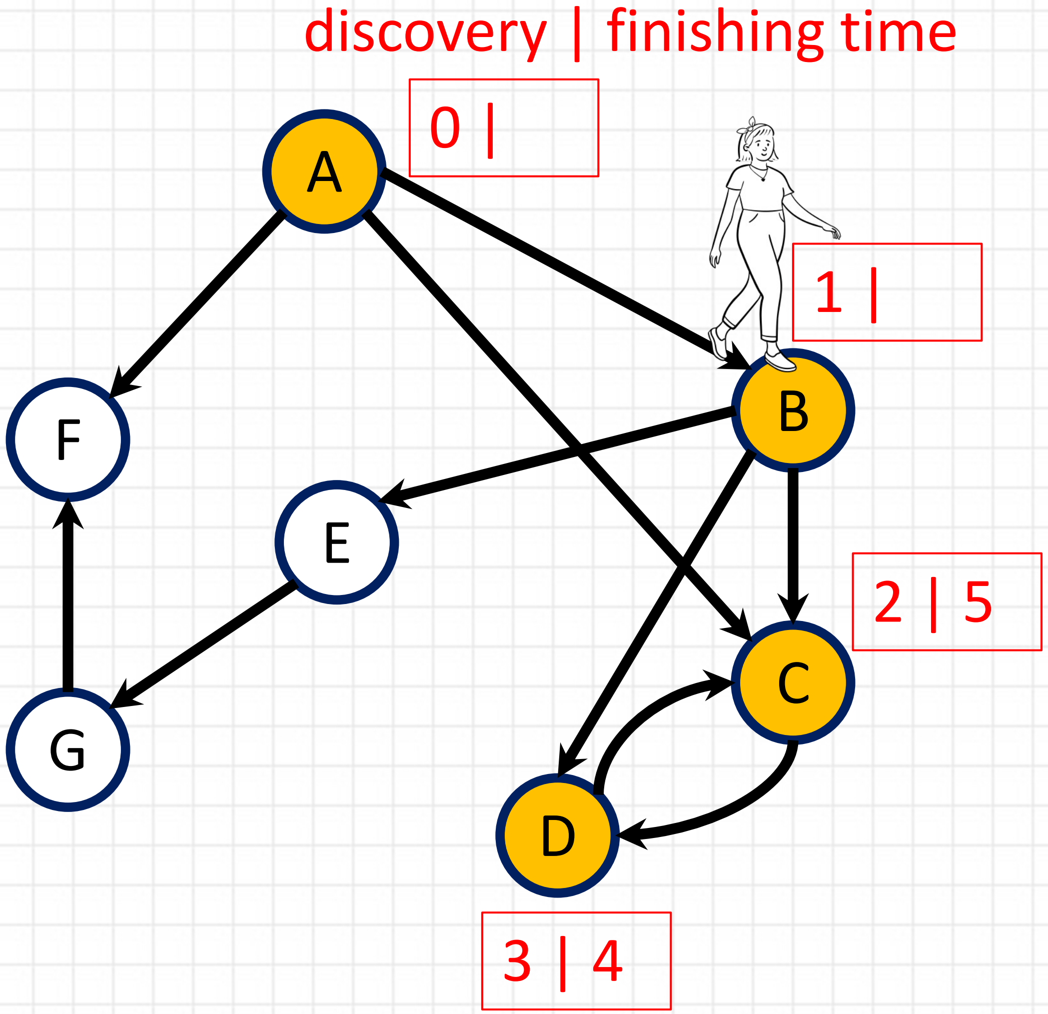


Graph Traversal: DFS

- 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

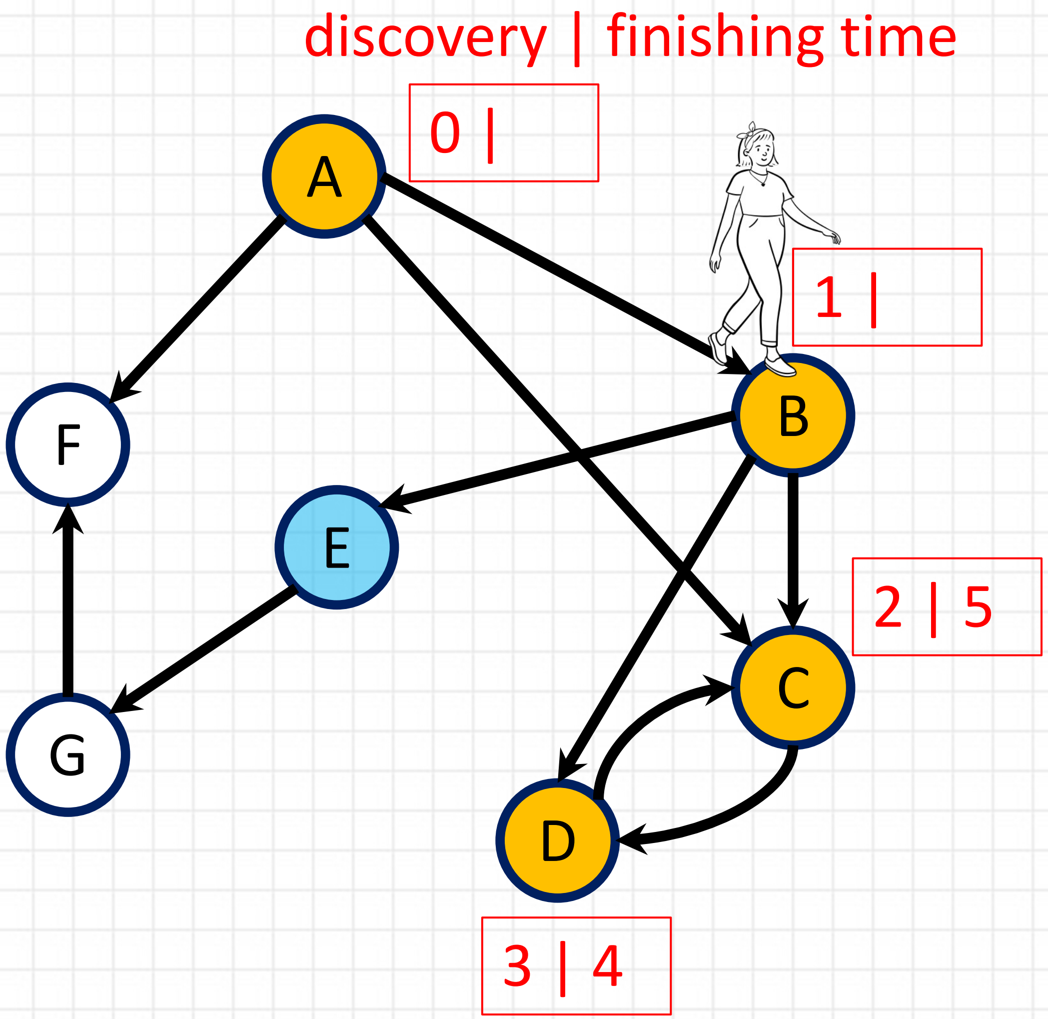
- Stack = {A, B, ~~C~~, ~~D~~}
- Visited = {A, B, C, D}
- No more path to explore → backtrack



Graph Traversal: DFS

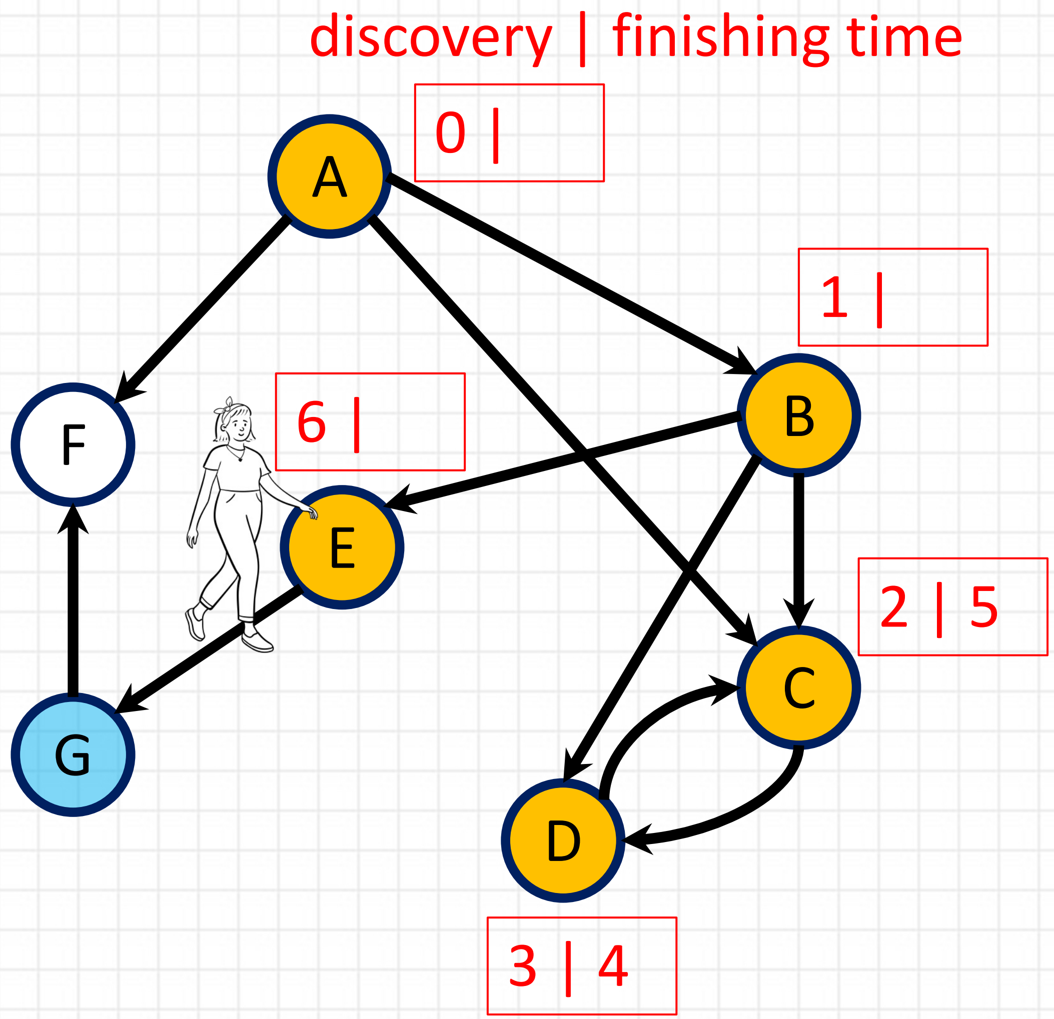
- DFS follows a single path as far (deep) as possible and then backtracks to the last alternative path
- Consider the same example, with minor difference →

- 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 →

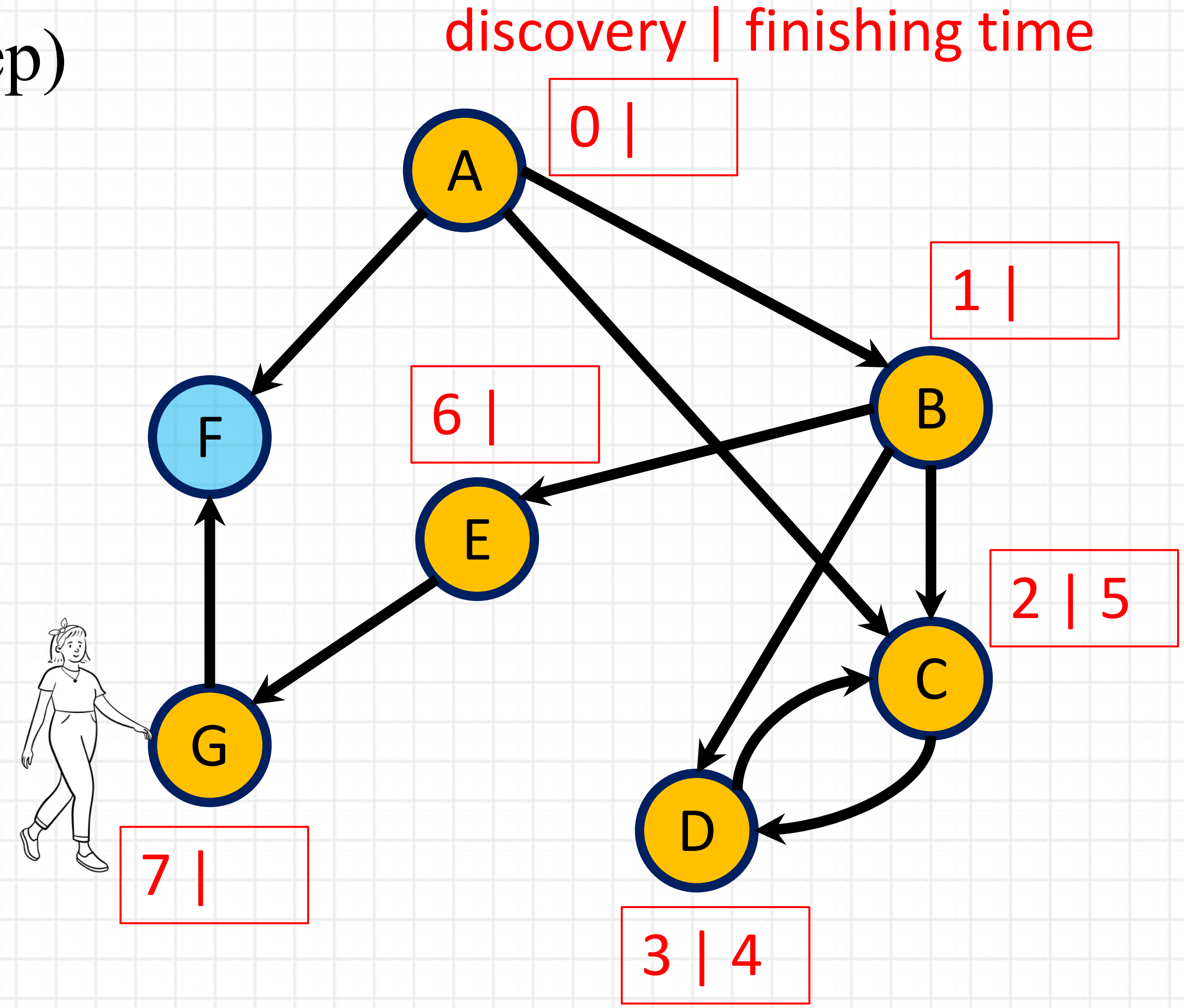


- Stack = {A, B, ~~C~~, ~~D~~, E, G}
- Visited = {A, B, C, D, E}

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 →

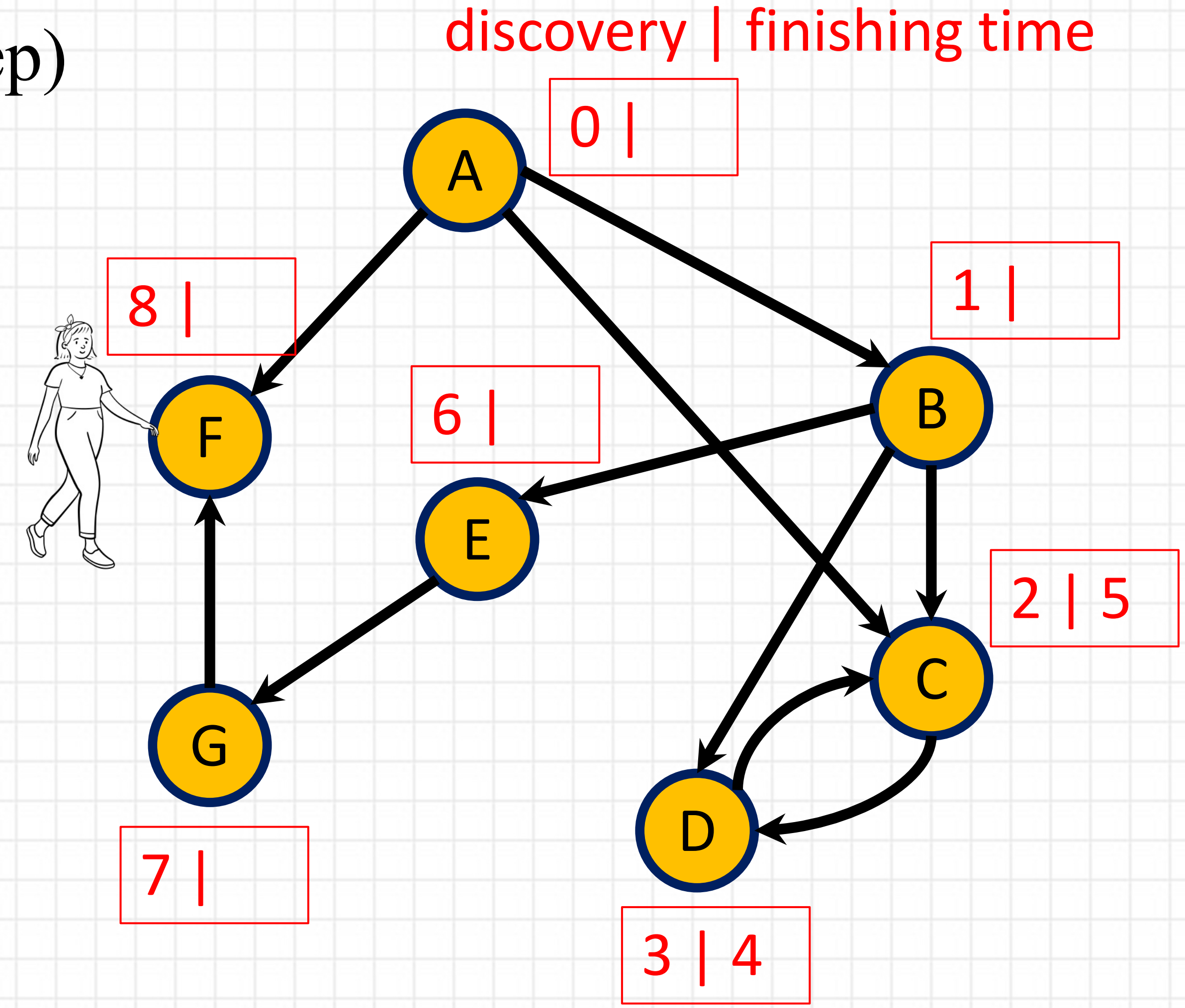
- Stack = {A, B, ~~C~~, ~~D~~, E, G, F}
- Visited = {A, B, C, D, E, G}



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 →

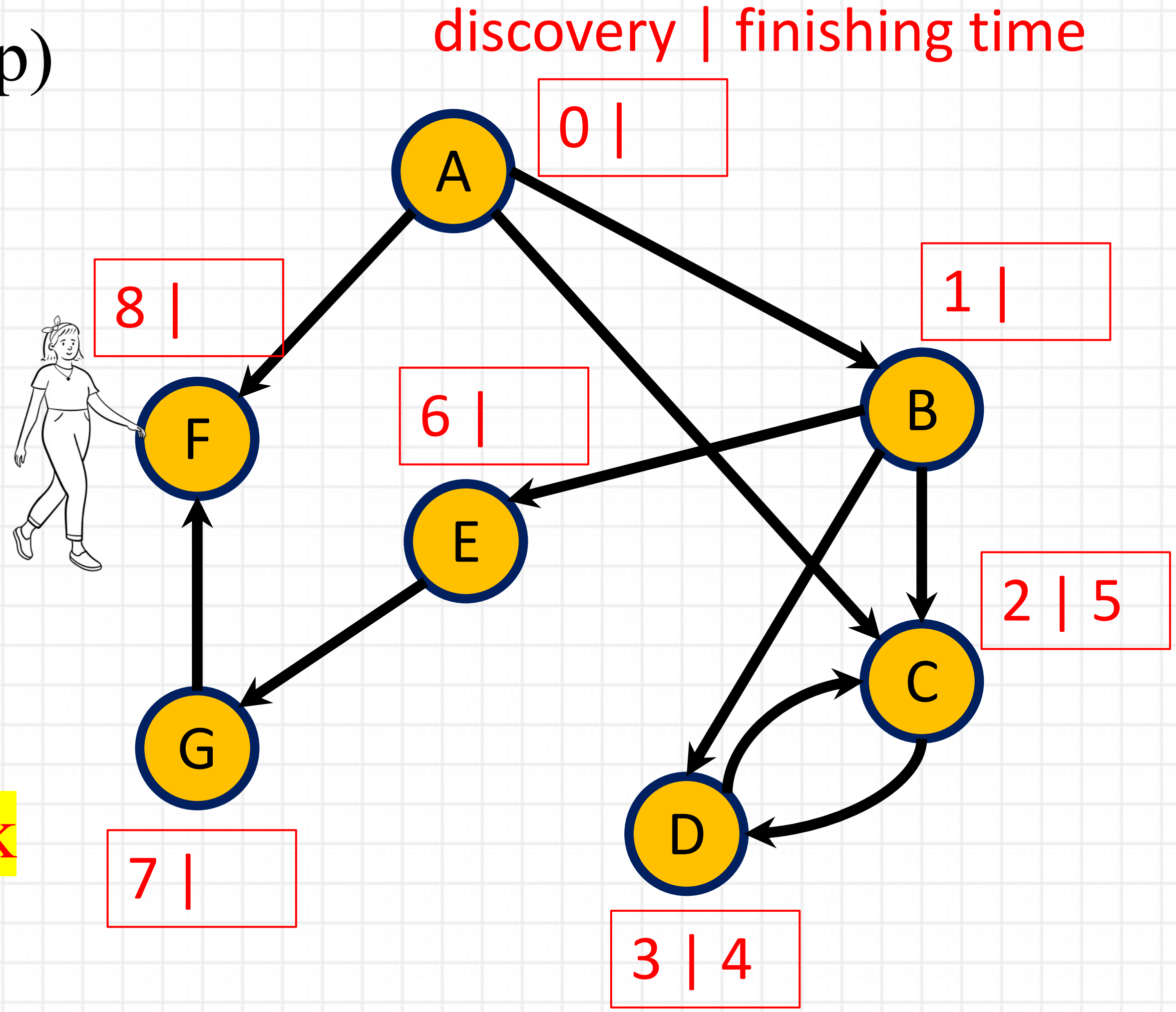
- Stack = {A, B, ~~C~~, ~~D~~, E, G, F}
- Visited = {A, B, C, D, E, G, F}



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

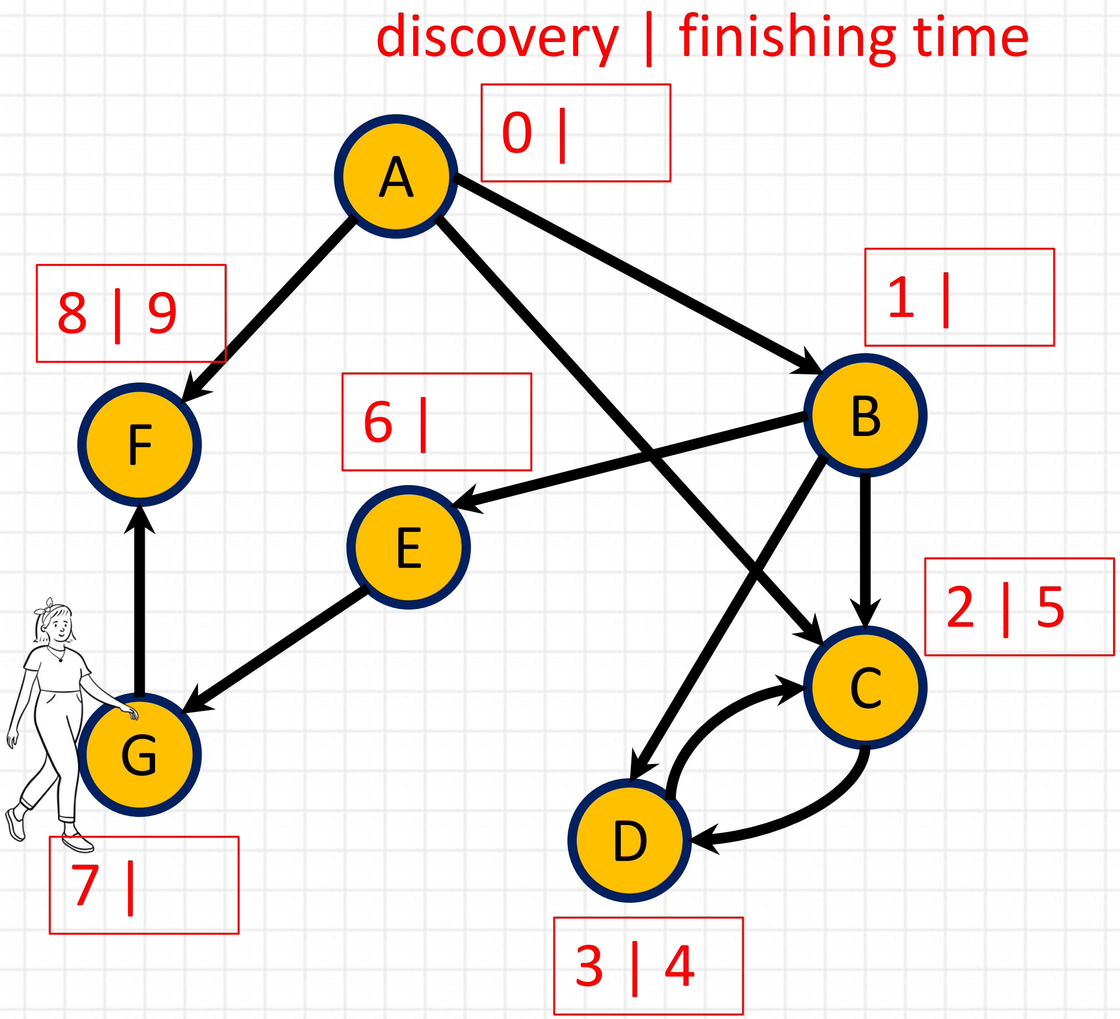


Graph Traversal: DFS

- 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

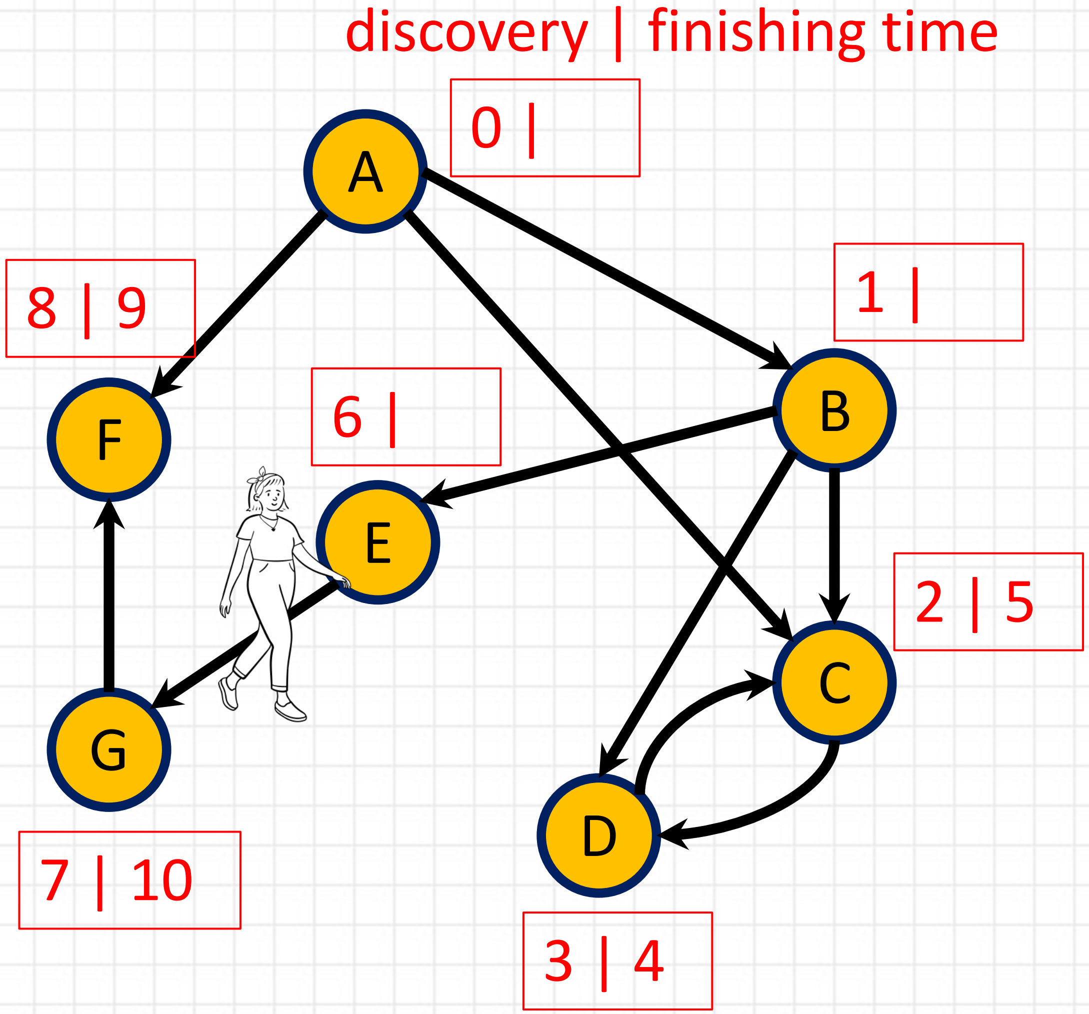


Graph Traversal: DFS

- 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

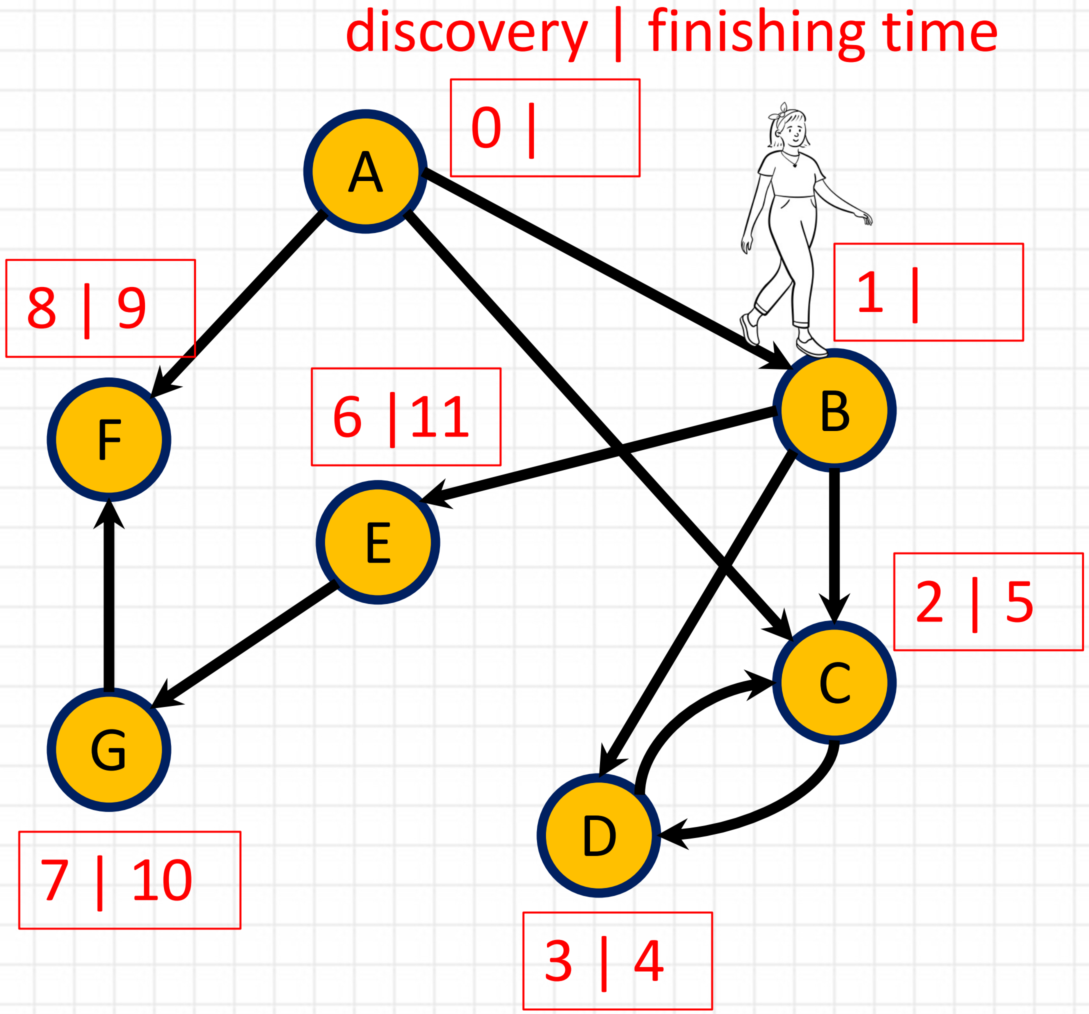


Graph Traversal: DFS

- 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

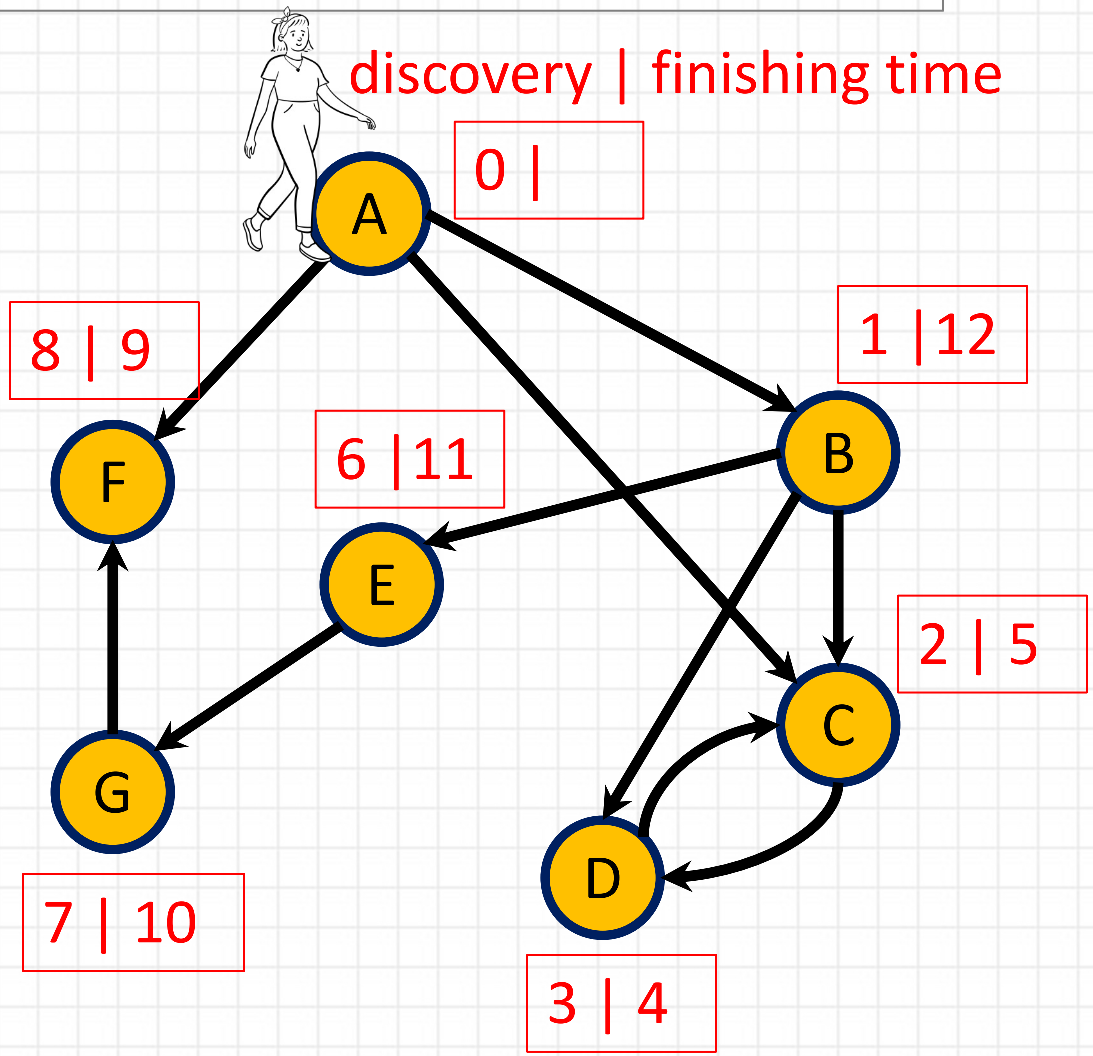


Graph Traversal: DFS

- 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



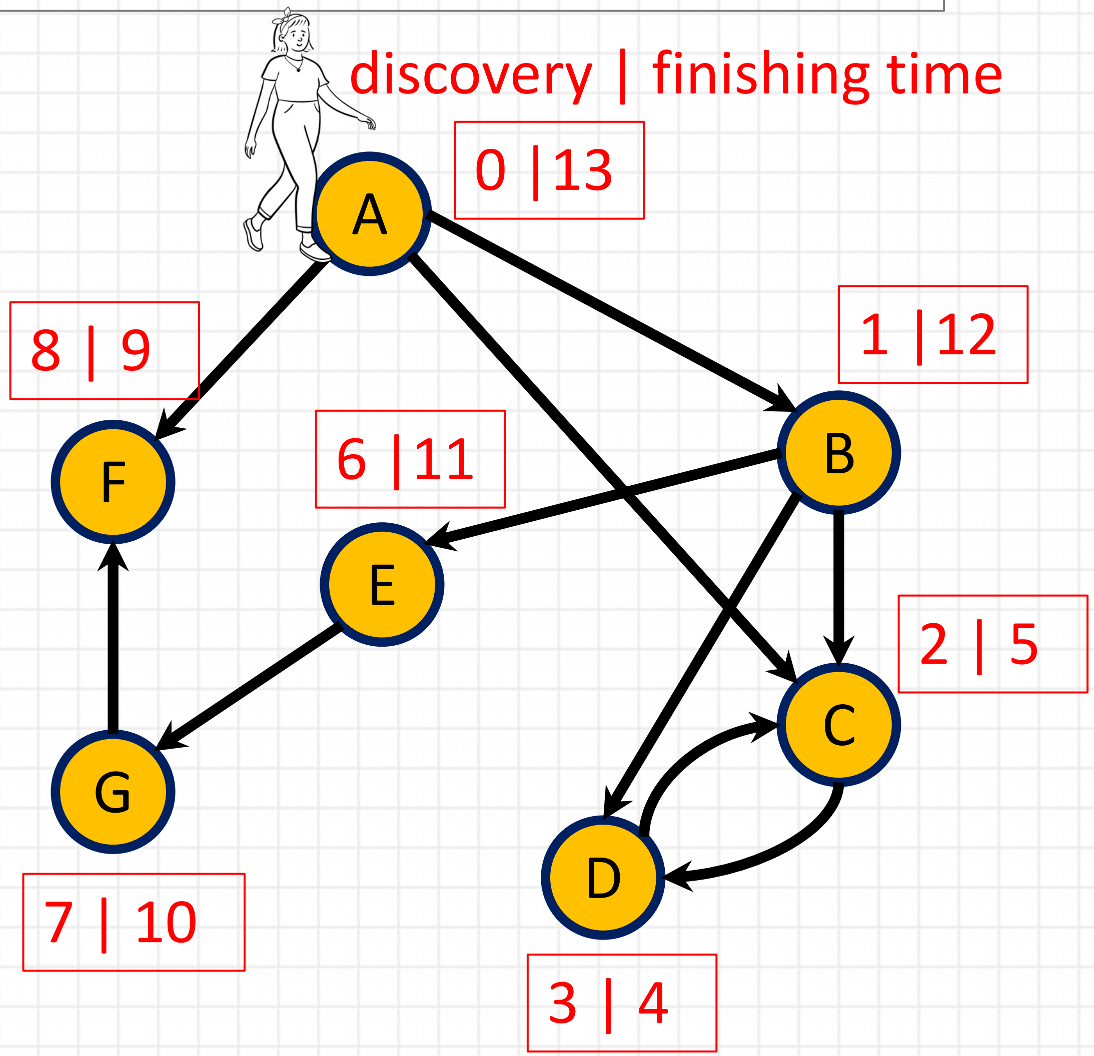
Graph Traversal: DFS

- 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

CLRS

```
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
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

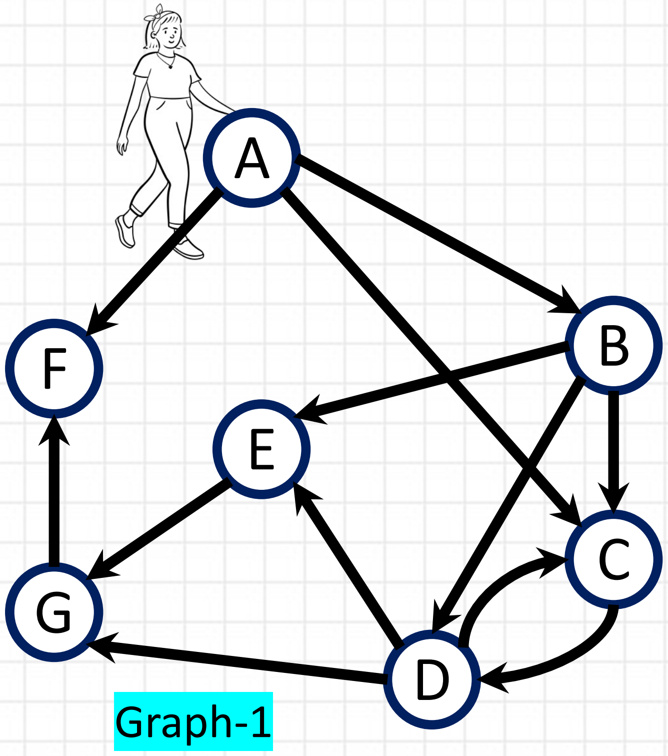
DPV

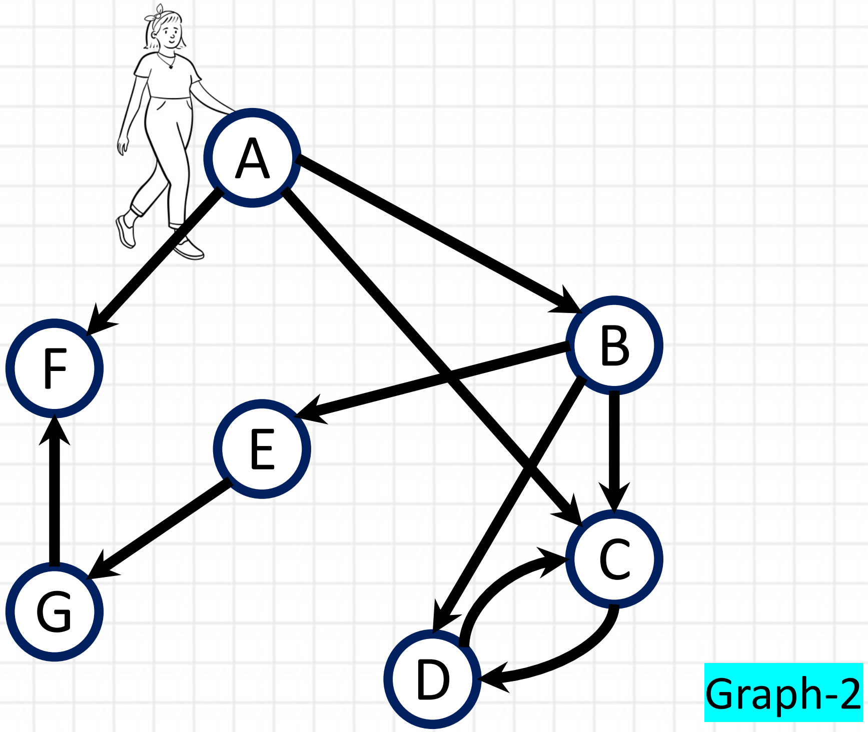
```
procedure dfs(G)

for all v ∈ V:
    visited(v) = false

for all v ∈ V:
    if not visited(v): explore(v)
```

```
procedure explore(G, v)
Input: G = (V, E) is a graph; v ∈ V
Output: visited(u) is set to true for all nodes u reachable
    from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
    if not visited(u): explore(u)
postvisit(v)
```

Graph Traversal: DFS

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





💻 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	Space Complexity
BFS	Iterative	Queue (FIFO)	O(|V| + |E|)	O(|V|)
DFS	Recursive Iterative	(not explicitly required → execution stack) Stack (LIFO)	O(|V| + |E|)	O(|V|)

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

Now, we know how to run BFS and DFS from a given source node.

- 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

Graph traversal

Breadth first search (BFS)

Depth first search (DFS)

Connectivity problem
(Connected components)

Shortest path (unweighted graphs)
Testing bipartiteness

Topological sorting
Strongly connected components

Tree traversal

- level-order

Tree traversal

- In-order, Pre-order, post-order

Graph

Graph traversal

Breadth first search (BFS)

- Shortest path (unweighted graphs)
- Testing bipartiteness
- Tree traversal
  - level-order

Depth first search (DFS)

- Topological sorting
- Strongly connected components
- Tree traversal
  - In-order, Pre-order, post-order

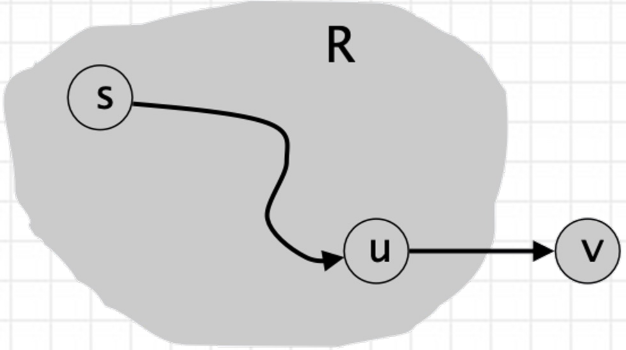
Connectivity problem (Connected components)

Connectivity problem can be solved by both BFS and DFS

Graph Traversal: Connected Component

- Connected component problem. 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

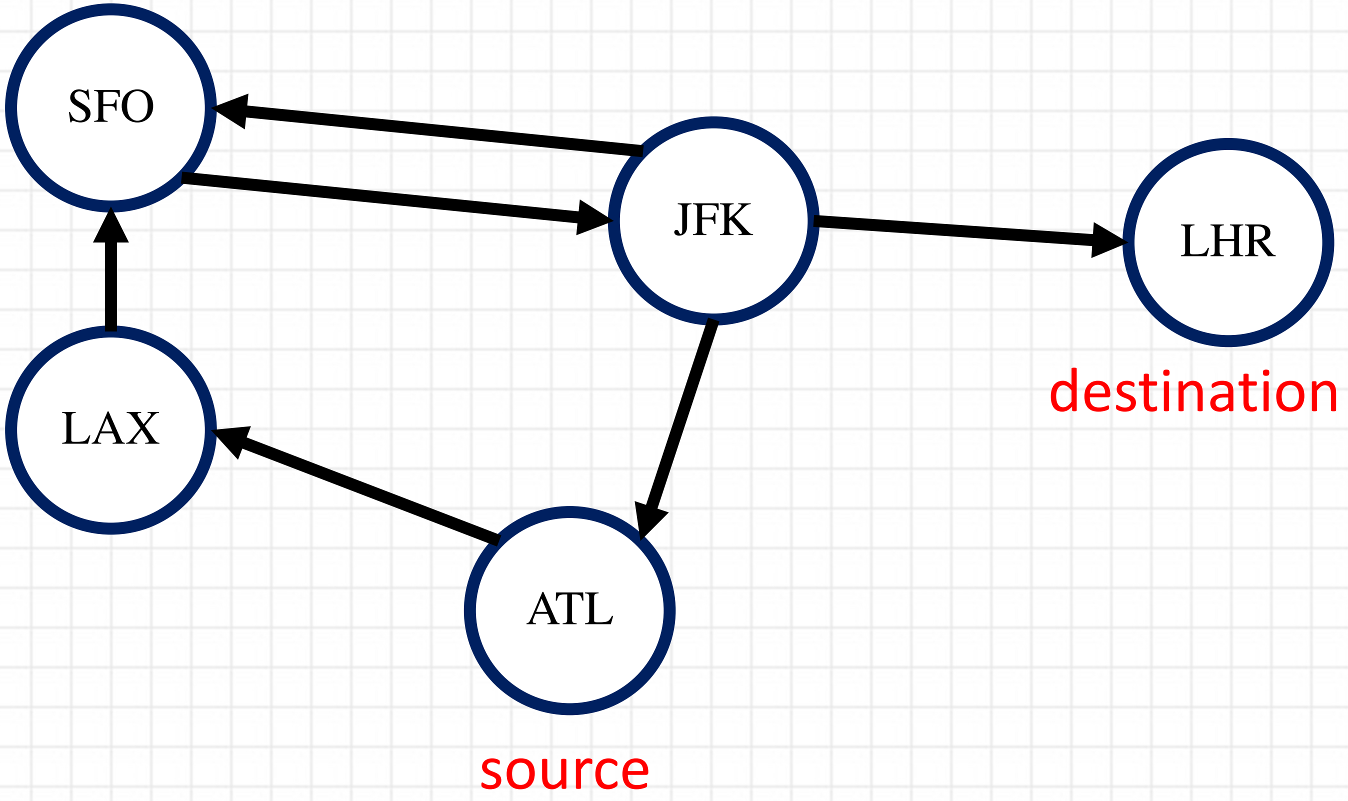
- Upon termination, R is the connected component containing s.
  - BFS
  - DFS

Graph Traversal: Connected Component

- Ex1: Given a set of flight plans, can we travel from Atlanta (ATL) to London (LHR)?
- Flights:
  - (JFK, ATL)
  - (ATL, LAX)
  - (LAX, SFO)
  - (JFK, SFO)
  - (SFO, JFK)
  - (JFK, LHR)

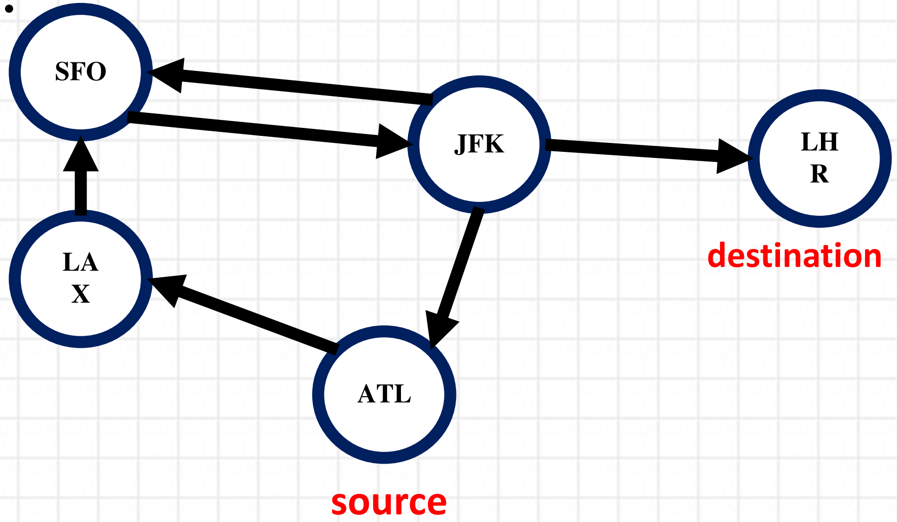
Graph Traversal: Connected Component

- Ex1: Given a set of flight plans, can we travel from Atlanta (ATL) to London (LHR)?
- Flights:
  - (JFK, ATL)
  - (ATL, LAX)
  - (LAX, SFO)
  - (JFK, SFO)
  - (SFO, JFK)
  - (JFK, LHR)



Graph Traversal: Connected Component

- Ex1: Given a set of flight plans, can we travel from Atlanta (ATL) to London (LHR)?
- Flights:
  - (JFK, ATL)
  - (ATL, LAX)
  - (LAX, SFO)
  - (JFK, SFO)
  - (SFO, JFK)
  - (JFK, LHR)



- Define the corresponding graph
- Run BFS or DFS from the source node, i.e., the node associated with ATL
- During the traversal check if the destination (LHR) is a neighbor of the current node

Demo code time!

Graph Traversal: Connected Component

- Ex2 [Grid problems]: Given an m-by-n 2D binary matrix in which 0 represent water and 1 represent land, design an algorithm computing the number islands. An island includes one or more horizontally or vertically cells surrounded by water.

1	1	0	0	0
1	1	0	0	1
1	0	0	0	1
0	0	1	0	0

1	1	0	0	0
1	1	0	0	1
1	0	0	0	1
0	0	1	0	0

Graph Traversal: Connected Component

- Ex2 [Grid problems]: Given an m-by-n 2D binary matrix in which 0 represent water and 1 represent land, design an algorithm computing the number islands. An island includes one or more horizontally or vertically cells surrounded by water.

1	1	0	0	0
1	1	0	0	1
1	0	0	0	1
0	0	1	0	0

grid =

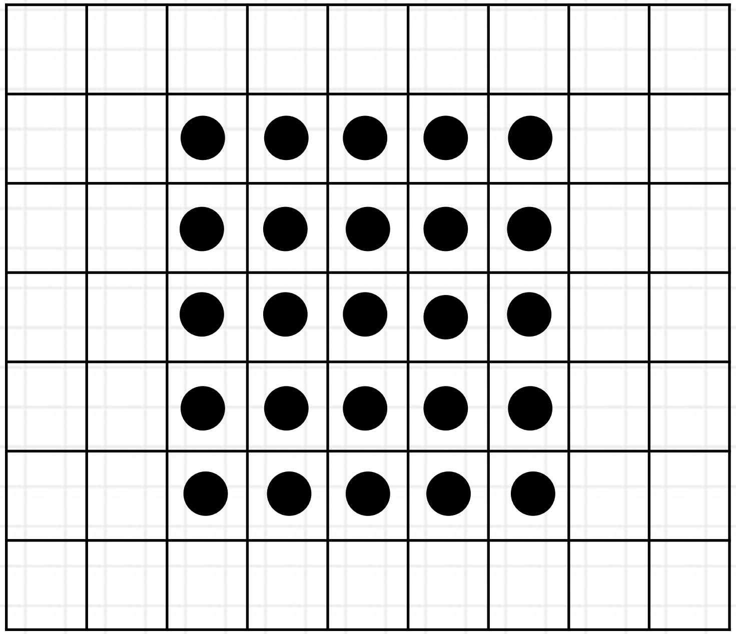
Graph Traversal: Connected Component

- Ex2 [Grid problems]: Given an m-by-n 2D binary matrix in which 0 represent water and 1 represent land, design an algorithm computing the number islands. An island includes one or more horizontally or vertically cells surrounded by water.

1	1	0	0	0
1	1	0	0	1
1	0	0	0	1
0	0	1	0	0

Each cell = graph node

grid =



Graph Traversal: Connected Component

- Ex2 [Grid problems]: Given an m-by-n 2D binary matrix in which 0 represent water and 1 represent land, design an algorithm computing the number islands. An island includes one or more horizontally or vertically cells surrounded by water.

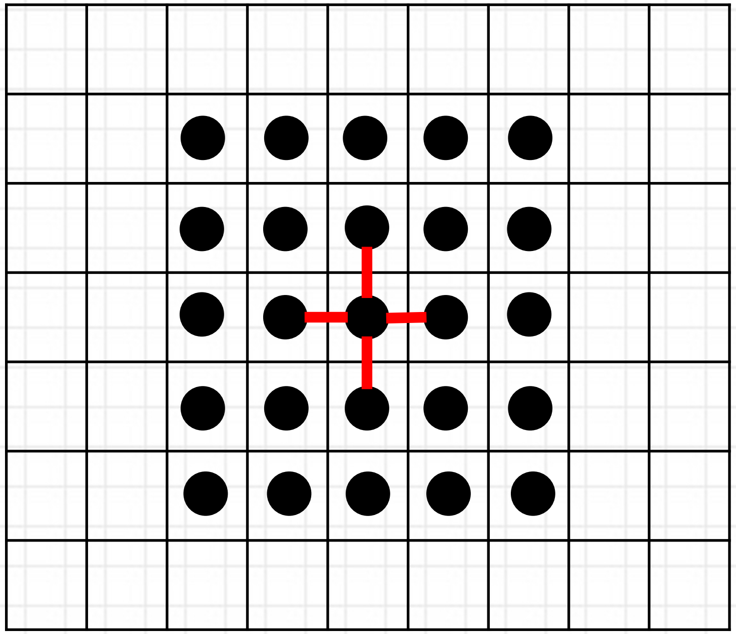
1	1	0	0	0
1	1	0	0	1
1	0	0	0	1
0	0	1	0	0

Each cell = graph node

Neighbors of grid[i][j]:

- grid[i-1][j]
- grid[i+1][j]
- grid[i][j-1]
- grid[i][j+1]

grid =



Graph Traversal: Connected Component

- Ex2 [Grid problems]: Given an m-by-n 2D binary matrix in which 0 represent water and 1 represent land, design an algorithm computing the number islands. An island includes one or more horizontally or vertically cells surrounded by water.

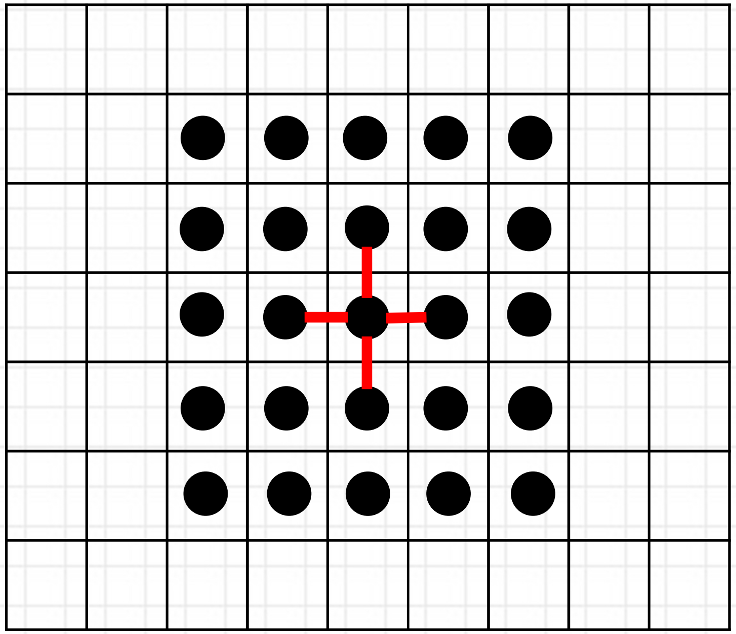
So, we can define the corresponding graph, then run BFS or DFS from each grid cell with the value of 1, i.e., grid[i][j]=1 and mark visited nodes.
Number of islands is equal to the number of times we need to start a search

Each cell = graph node

Neighbors of grid[i][j]:

- grid[i-1][j]
- grid[i+1][j]
- grid[i][j-1]
- grid[i][j+1]

grid =



Graph Traversal: Connected Component

- Ex2 [Grid problems]: Given an m-by-n 2D binary matrix in which 0 represent water and 1 represent land, design an algorithm computing the number islands. An island includes one or more horizontally or vertically cells surrounded by water.

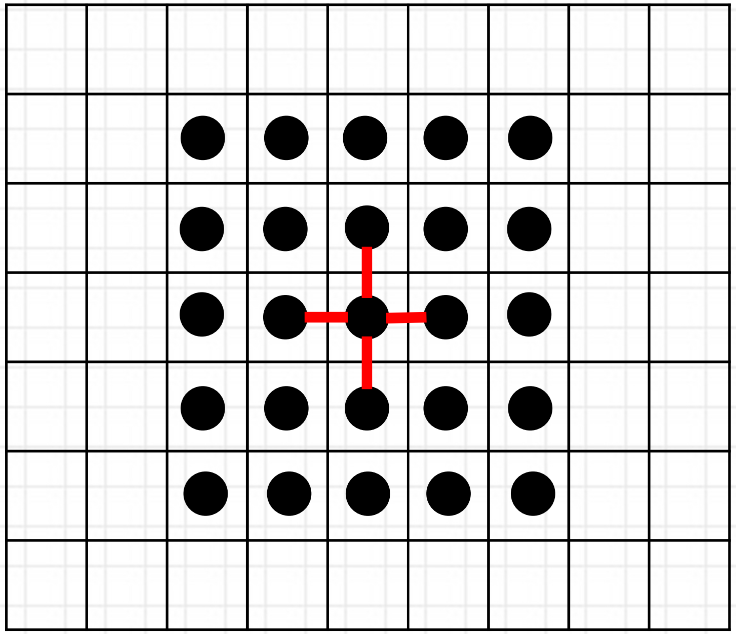
But do we need to define the corresponding graph explicitly? (by defining the adjacency matrix or adjacency list?)

Each cell = graph node

Neighbors of grid[i][j]:

- grid[i-1][j]
- grid[i+1][j]
- grid[i][j-1]
- grid[i][j+1]

grid =



Graph Traversal: Connected Component

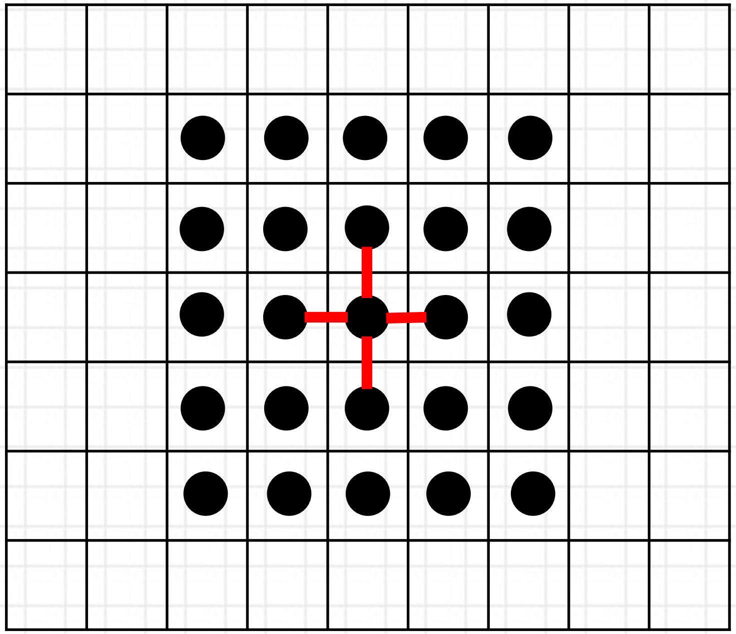
- Ex2 [Grid problems]: Given an m-by-n 2D binary matrix in which 0 represent water and 1 represent land, design an algorithm computing the number islands. An island includes one or more horizontally or vertically cells surrounded by water.

We know the nodes (= grid cells) and we know the neighbors (the relationship), so we can skip the graph definition part!

Each cell = graph node

Neighbors of grid[i][j]:
- grid[i-1][j]
- grid[i+1][j]
- grid[i][j-1]
- grid[i][j+1]

grid =



Demo code!

Graph

Graph traversal

Breadth first search (BFS)

Depth first search (DFS)

Connectivity problem
(Connected components)

Shortest path (unweighted graphs)
Testing bipartiteness

Tree traversal
- level-order

Before continuing with applications, let's revisit a few things from BFS and DFS

Topological sorting
Strongly connected components

Tree traversal
- In-order, Pre-order, post-order

BFS and DFS

- Both are graph traversal algorithms

BFS	DFS
Iterative: Queue (FIFO), Time: $O(\|V\| + \|E\|)$, Space: $O(\|V\|)$	Recursive: (execution stack), Iterative: Stack(LIFO) Time: $O(\|V\| + \|E\|)$, Space: $O(\|V\|)$

```
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
```


```
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
```

BFS and DFS

- Both are graph traversal algorithms

BFS	DFS
Iterative: Queue (FIFO), Time: $O(\|V\| + \|E\|)$, Space: $O(\|V\|)$	Recursive: (execution stack), Iterative: Stack(LIFO) Time: $O(\|V\| + \|E\|)$, Space: $O(\|V\|)$
BFS builds a breadth-first tree as it searches the graph. Formally, given graph $G = (V, E)$ with source s, we define the predecessor subgraph of G as $G_\pi = (V_\pi, E_\pi)$ where $V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$ $E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$ (tree edges) When applied to a directed or undirected graph $G = (V, E)$, BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.	

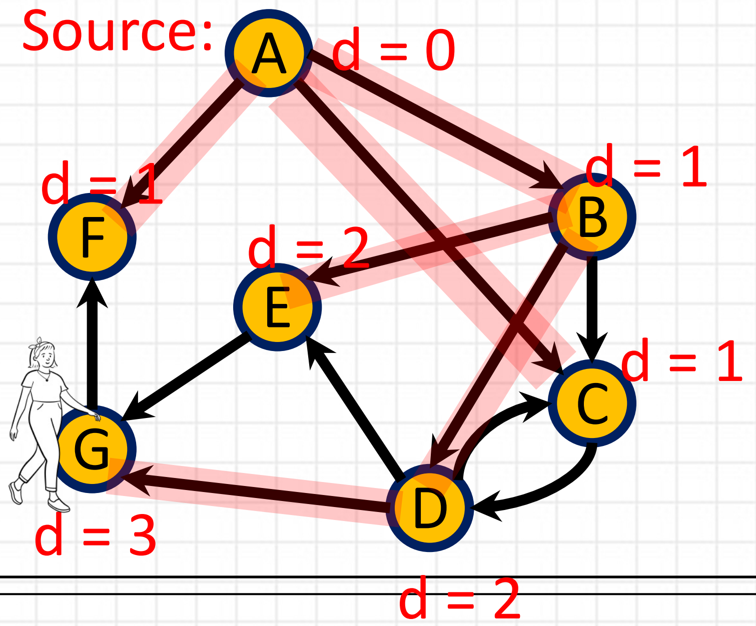
BFS and DFS

- Both are graph traversal algorithms

BFS	DFS
Iterative: Queue (FIFO), Time: O(|V| + |E|), Space: O(|V|)	Recursive: (execution stack), Iterative: Stack(LIFO) Time: O(|V| + |E|), Space: O(|V|)
BFS builds a breadth-first tree as it searches the graph. Formally, given graph $G = (V, E)$ with source s, we define the predecessor subgraph of G as $G_\pi = (V_\pi, E_\pi)$ where $V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$ $E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$ (tree edges) When applied to a directed or undirected graph $G = (V, E)$, BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.	

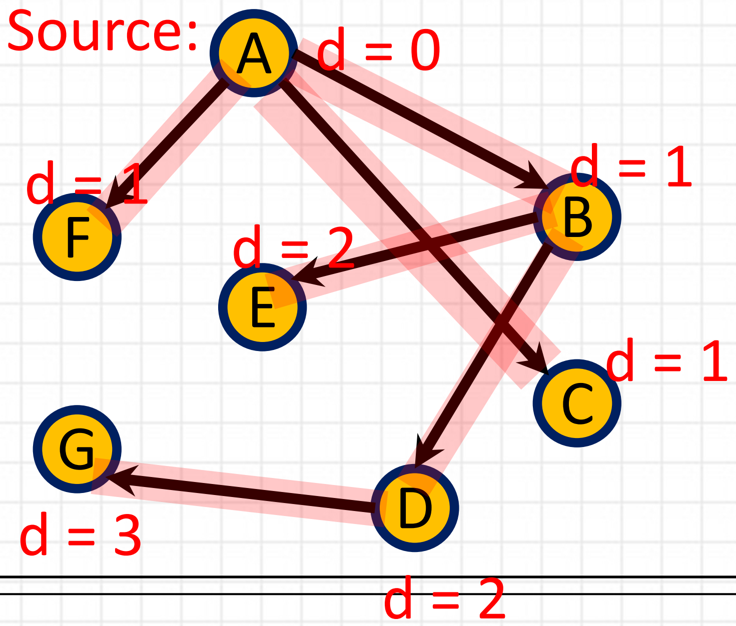
BFS and DFS

- Both are graph traversal algorithms

BFS	DFS
Iterative: Queue (FIFO), Time:O(|V| + |E|), Space: O(|V|)	Recursive: (execution stack), Iterative: Stack(LIFO) Time:O(|V| + |E|), Space: O(|V|)
BFS builds a breadth-first tree as it searches the graph. 	

BFS and DFS

- Both are graph traversal algorithms

BFS	DFS
Iterative: Queue (FIFO), Time:O(|V| + |E|), Space: O(|V|)	Recursive: (execution stack), Iterative: Stack(LIFO) Time:O(|V| + |E|), Space: O(|V|)
BFS builds a breadth-first tree as it searches the graph.  	

BFS and DFS

- Both are graph traversal algorithms

BFS	DFS
Iterative: Queue (FIFO), Time:O(|V| + |E|), Space: O(|V|)	Recursive: (execution stack), Iterative: Stack(LIFO) Time:O(|V| + |E|), Space: O(|V|)

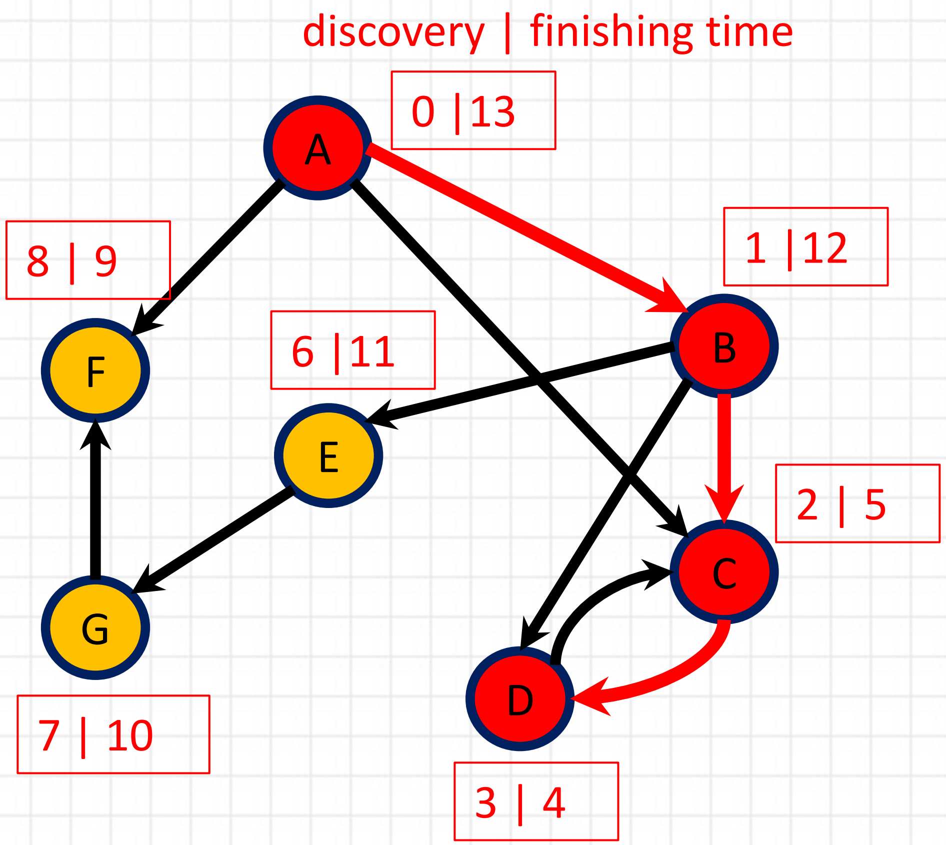
BFS builds a breadth-first tree as it searches the graph. Printing out the vertices on a shortest path from s to v, using the BFS tree
 	
PRINT-PATH(G, s, v)
if $v == s$
    print s
elseif $v.\pi ==$ NIL
    print "no path from" s "to" v "exists"
else PRINT-PATH$(G, s, v.\pi)$
    print v	

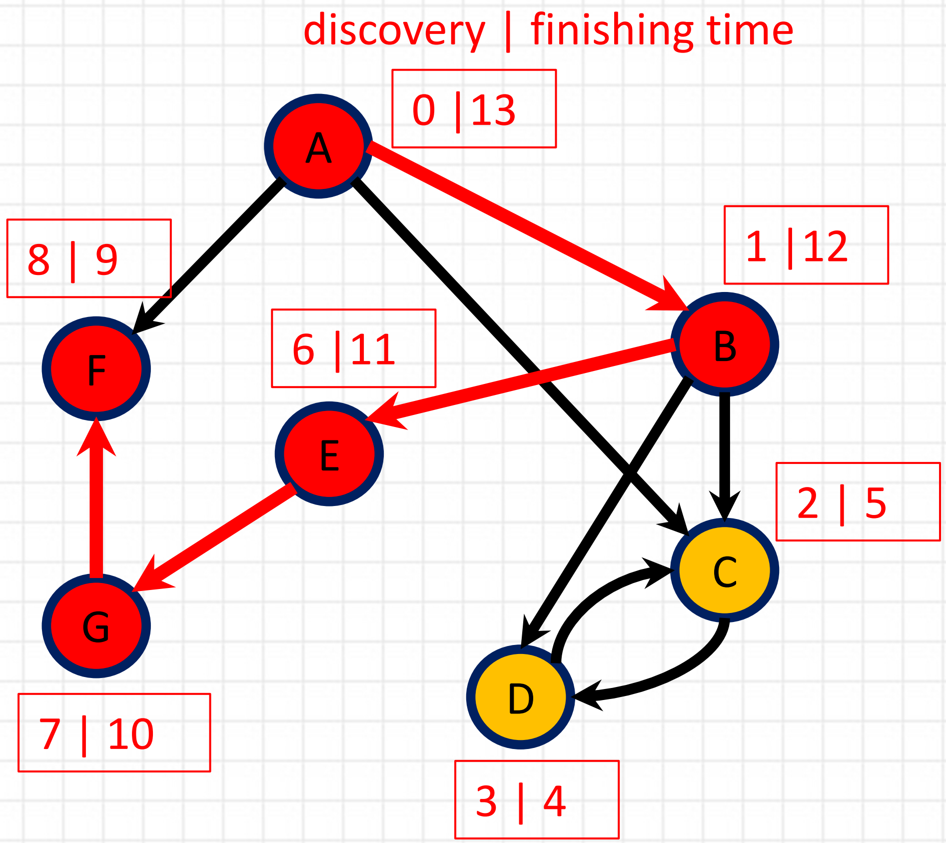
BFS and DFS

- Both are graph traversal algorithms

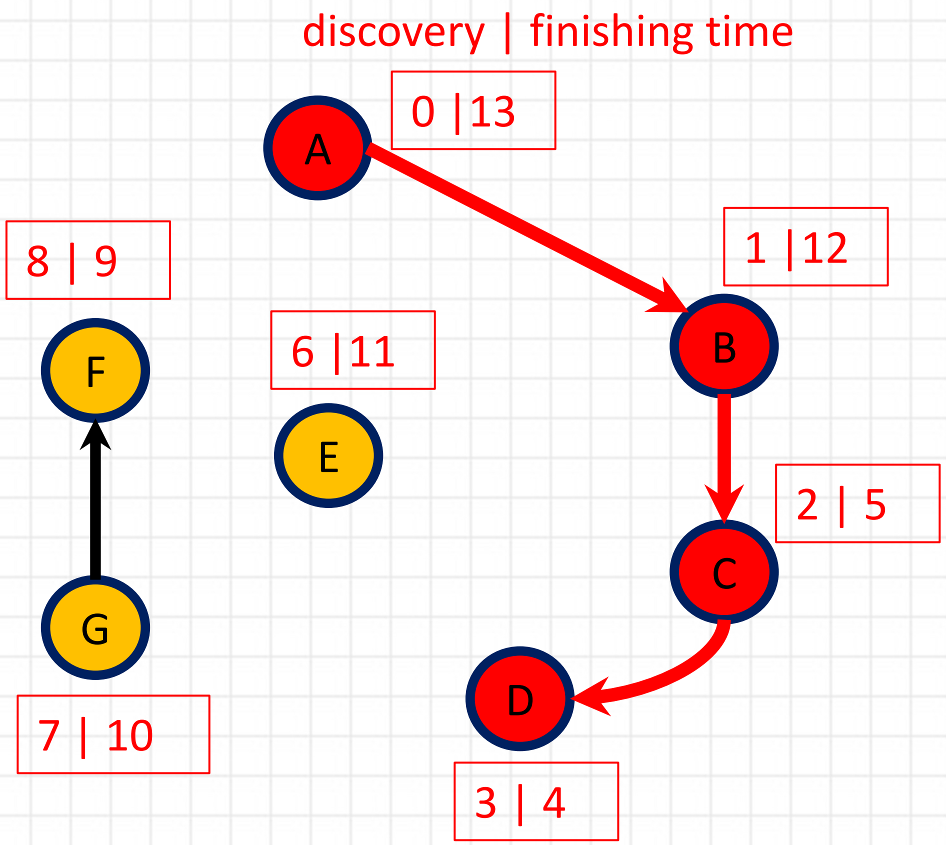
BFS	DFS
Iterative: Queue (FIFO), Time:O(|V| + |E|), Space: O(|V|)	Recursive: (execution stack), Iterative: Stack(LIFO) Time:O(|V| + |E|), Space: O(|V|)
➢ **BFS builds a breadth-first tree as it searches the graph.** ➢ **We can print out the vertices on a shortest path from s to v, using the BFS tree** ➢ **We only have one distance measure (timestamp), denoted by d, assigned to each node, i.e., the time that a node visited for the first (and last) time.**	The predecessor subgraph produced by a DFS may be composed of several trees, because the search may repeat from multiple sources. The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees. DFS timestamps each node with two numbers; d (discovery time): the first time we visit the node and f (finishing time): the last time we visit the node (after exploring all possible paths followed the node)

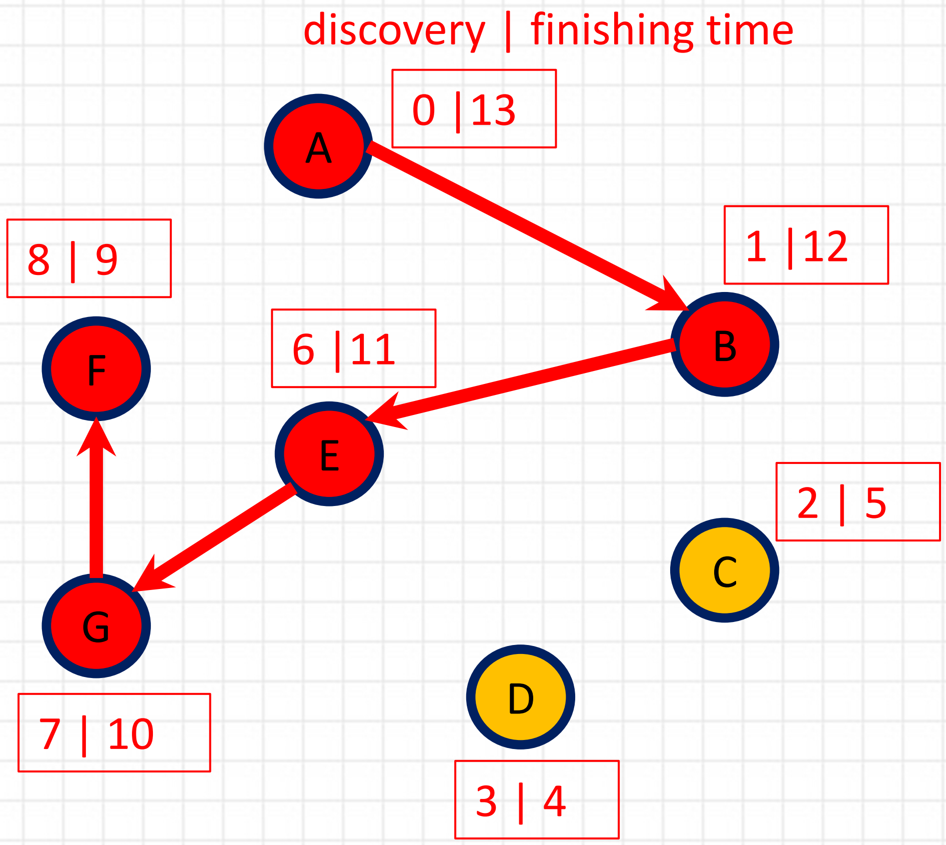
DFS Forest Example





DFS Forest Example

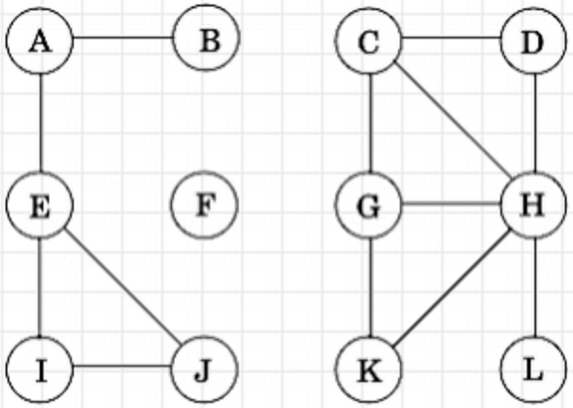




DFS Forest Example

DFS(G)

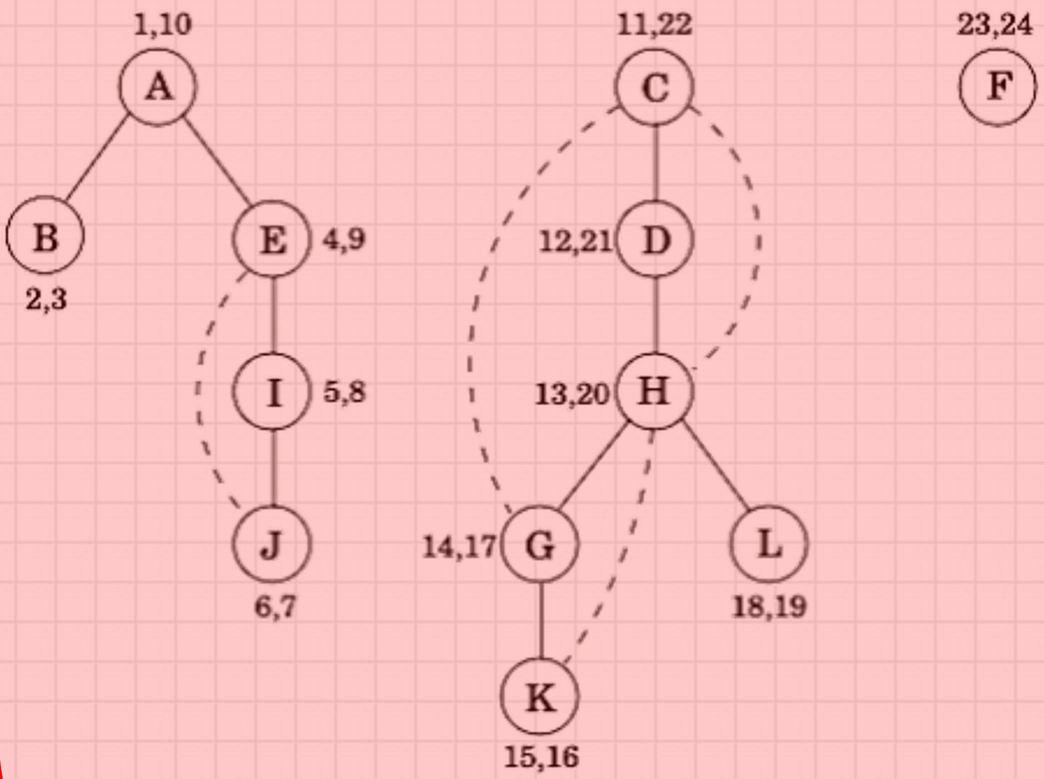
for each vertex $u \in G.V$
  $u.color$ = WHITE
  $u.\pi$ = NIL
$time = 0$
for each vertex $u \in G.V$
  if $u.color$ == WHITE
    DFS-VISIT(G, u)



DFS-VISIT(G, u)

$time = time + 1$
$u.d = time$
$u.color$ = GRAY
for each $v \in G.Adj[u]$
  if $v.color$ == WHITE
    $v.\pi = u$
    DFS-VISIT(G, v)
$u.color$ = BLACK
$time = time + 1$
$u.f = time$

More accepted definition of DFS Forest



Number of times DFS-Visit(G, u) is called = Number of trees

BFS and DFS

- Both are graph traversal algorithms

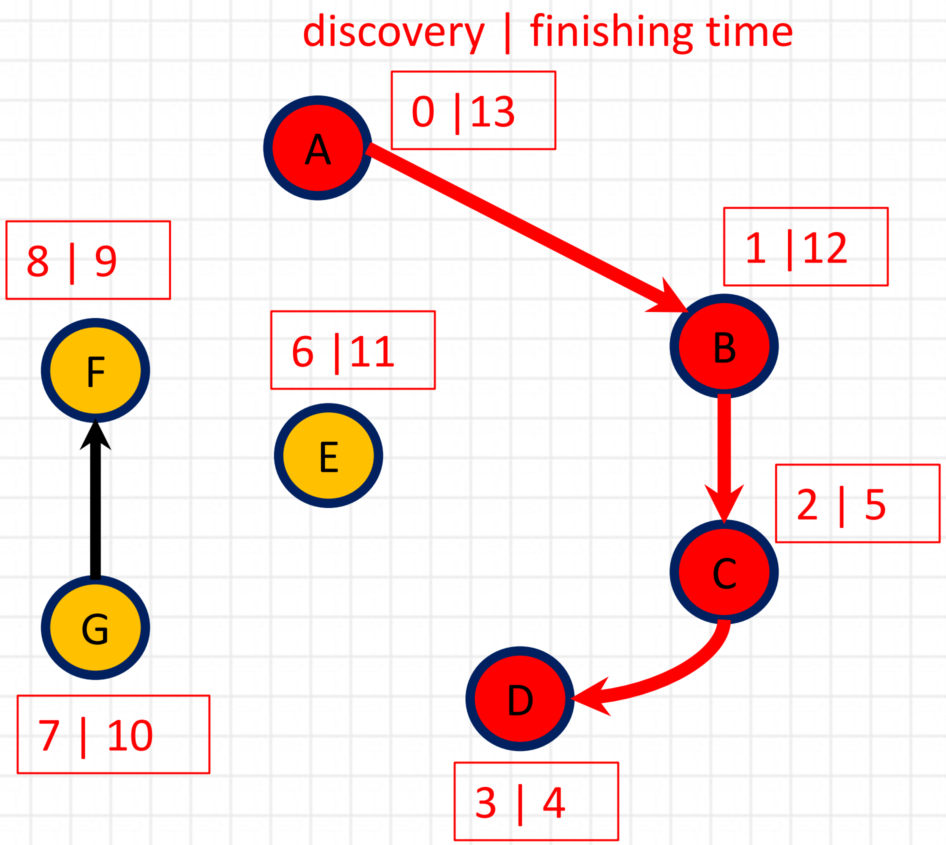
BFS	DFS
Iterative: Queue (FIFO), Time:O(|V| + |E|), Space: O(|V|)	Recursive: (execution stack), Iterative: Stack(LIFO) Time:O(|V| + |E|), Space: O(|V|)
➢ **BFS builds a breadth-first tree as it searches the graph.** ➢ **We can print out the vertices on a shortest path from s to v, using the BFS tree** ➢ **We only have one distance measure (timestamp), denoted by d, assigned to each node, i.e., the time that a node visited for the first (and last) time.**	The timestamps have parenthesis structure: In any DFS on graph $G = (V, E)$, for any two vertices u and v, exactly one of the following can happen: 1. $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint 2. The interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$ 3. The interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$

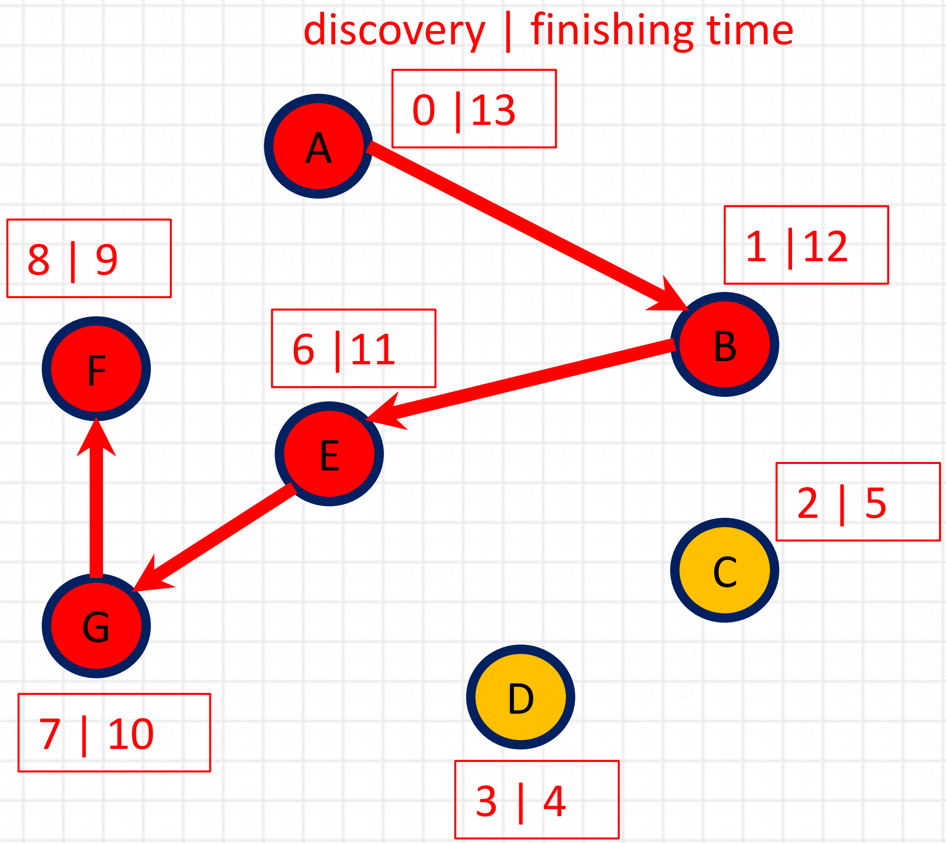
DFS: Timestamps Parenthesis Structure

- The timestamps have parenthesis structure:
- In any DFS on graph $G = (V, E)$, for any two vertices u and v, exactly one of the following can happen:

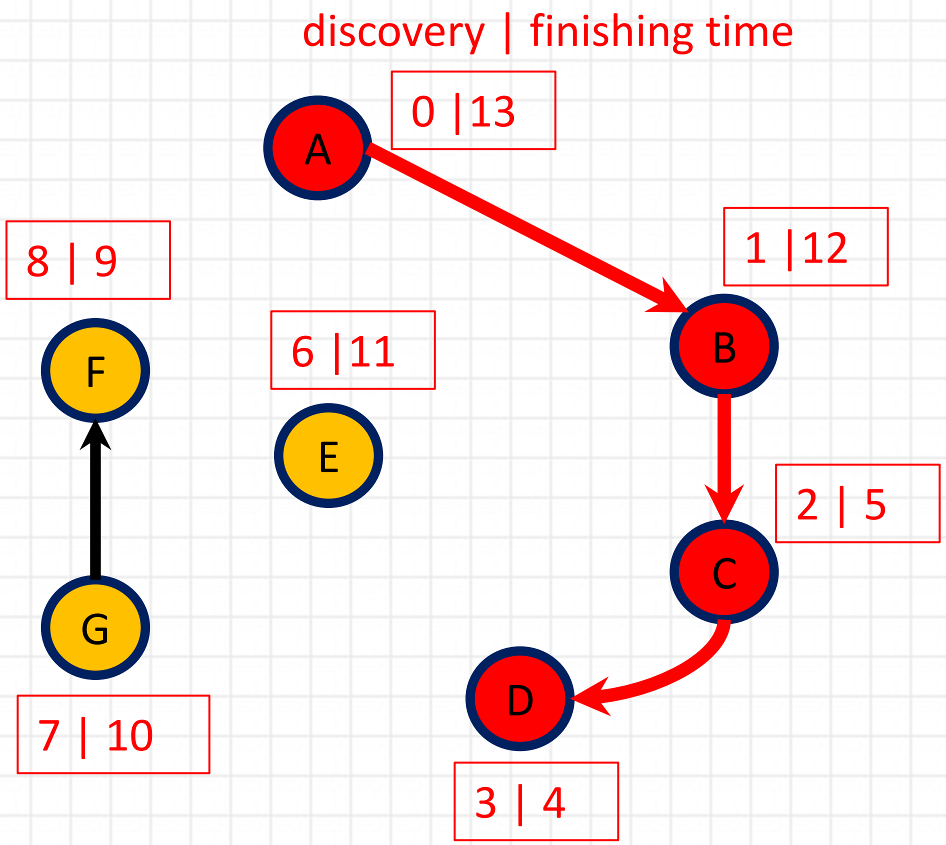
1. The intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest.
2. The interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree.
3. The interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.

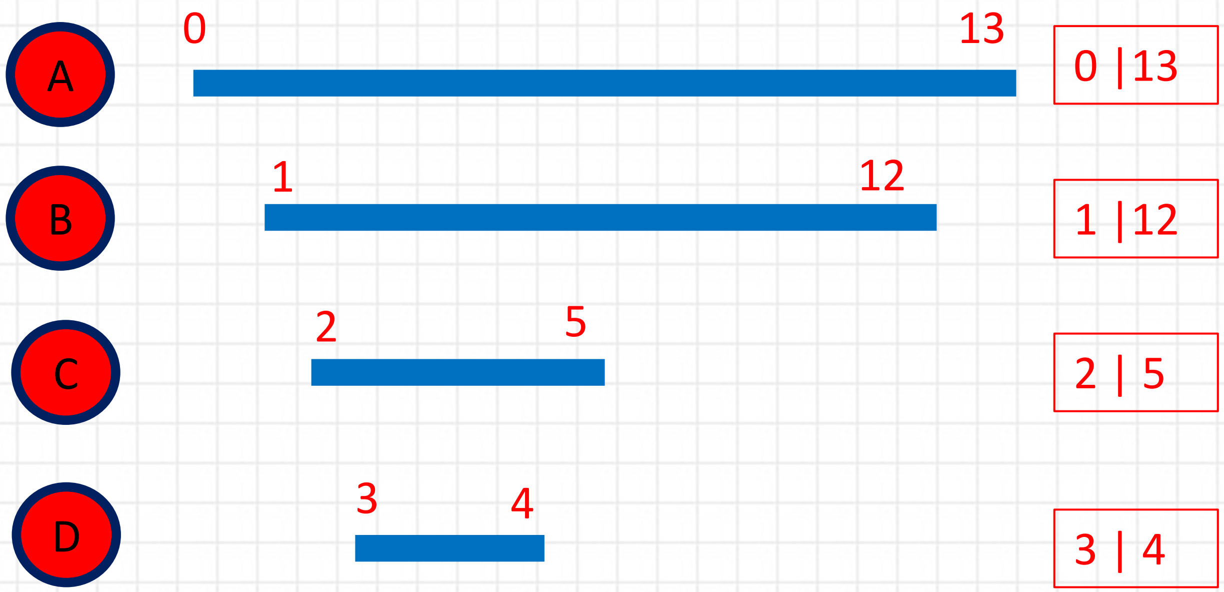
DFS Parenthesis Structure Example





DFS Parenthesis Structure Example





BFS and DFS

- Both are graph traversal algorithms

BFS	DFS
Iterative: Queue (FIFO), Time: $O(\lvert V\rvert + \lvert E\rvert)$, Space: $O(\lvert V\rvert)$	Recursive: (execution stack), Iterative: Stack(LIFO) Time: $O(\lvert V\rvert + \lvert E\rvert)$, Space: $O(\lvert V\rvert)$
➢ **BFS builds a breadth-first tree as it searches the graph.** ➢ **We can print out the vertices on a shortest path from s to v, using the BFS tree** ➢ **We only have one distance measure (timestamp), denoted by d, assigned to each node, i.e., the time that a node visited for the first (and last) time.**	➢ **The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees.** ➢ **DFS timestamps each node with two numbers;** **d (discovery time) and f (finishing time).** ➢ **The timestamps have parenthesis structure.**

Graph

Graph traversal

Breadth first search (BFS)

Depth first search (DFS)

Connectivity problem
(Connected components)

Now, we can continue with the applications

Shortest path (unweighted graphs)
Testing bipartiteness

Topological sorting
Strongly connected components

Tree traversal
- level-order

Tree traversal
- In-order, Pre-order, post-order

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.