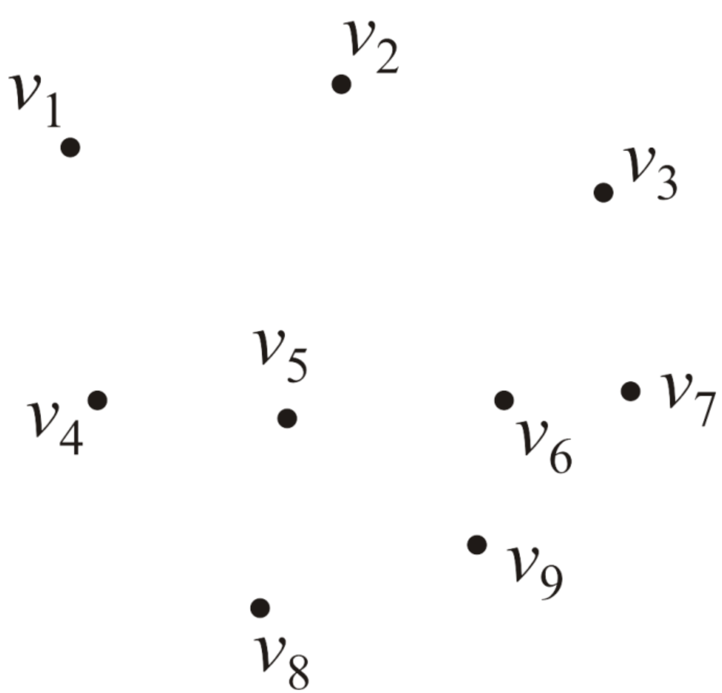
Data Structures
Instructor: Haifz Tayyeb Javed
Week-14-Lecture-01

---

20. Graphs

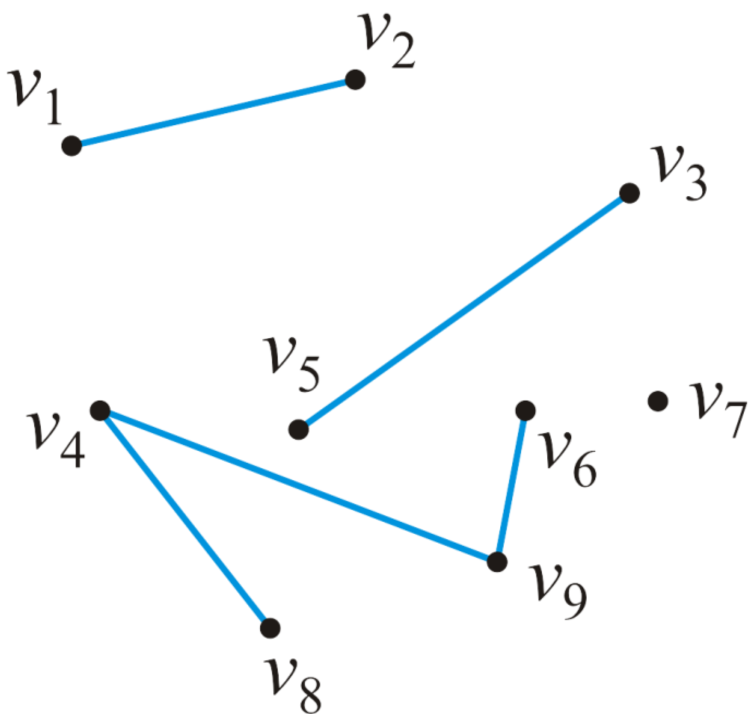
Graphs

- Consider this collection of vertices
  - `V = {v`$_1$`, v`$_2$`, ..., v`$_9$`}`
  - Where `|V| = n`



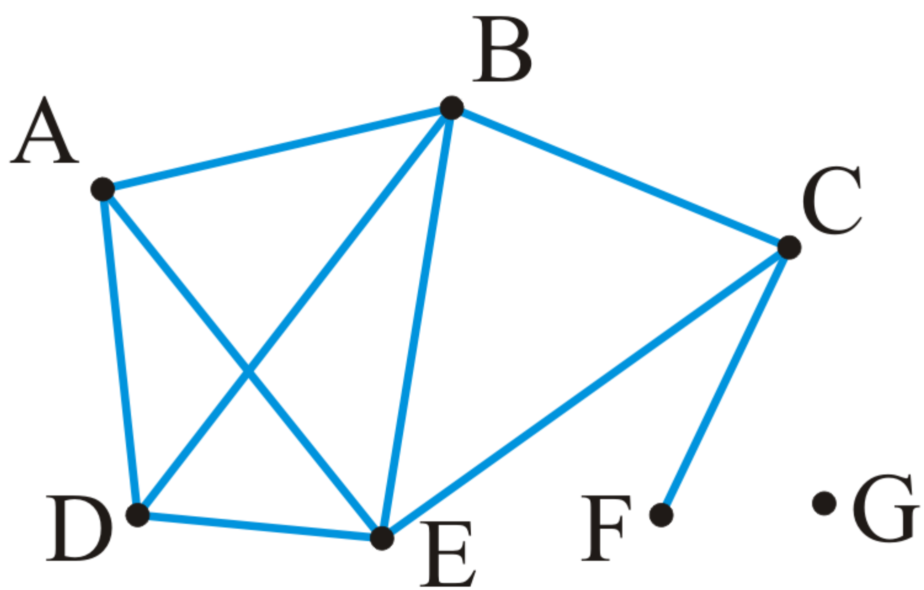
Graphs

- Associated with these vertices are $|E| = 5$ edges
  - $E = \{\{v_1, v_2\}, \{v_3, v_5\}, \{v_4, v_8\}, \{v_4, v_9\}, \{v_6, v_9\}\}$
- Pair $\{v_j, v_k\}$ indicates following relations
  - Vertex v_j is adjacent to vertex v_k
  - Vertex v_k is adjacent to vertex v_j



Graphs – Example

- Given `|V| = 7` vertices and `|E| = 9` edges
  - `V = {A, B, C, D, E, F, G}`
  - `E = {{A, B}, {A, D}, {A, E}, {B, C}, {B, D}, {B, E},`
    `{C, E}, {C, F}, {D, E}}`



Applications Of Graphs

- Driving Map
  - Vertex = Intersection, destinations
  - Edge = Road

- Airline Traffic
  - Vertex = Cities serviced by the airline
  - Edge = Flight exists between two cities

- Computer networks
  - Vertex = Server nodes, end devices, routers
  - Edge = Data link

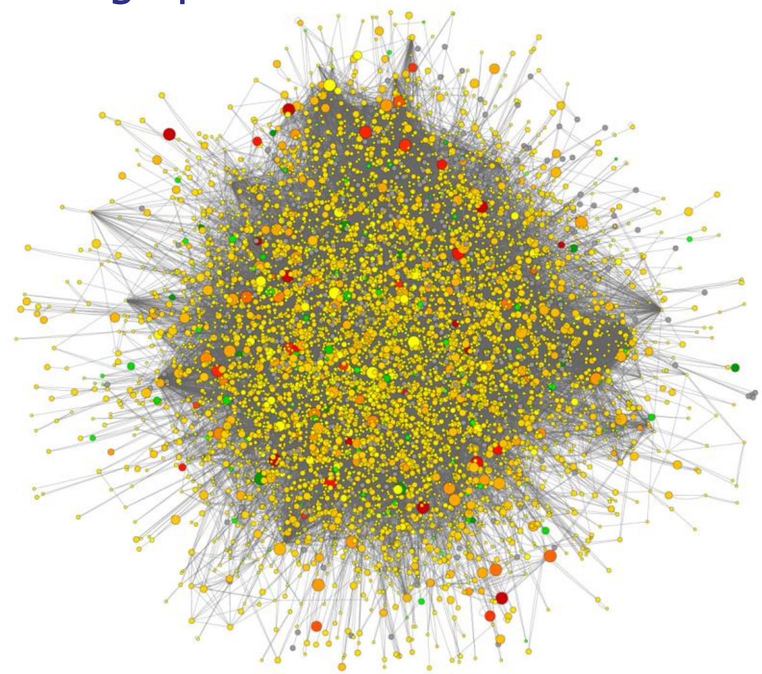
Applications Of Graphs

- Many real-world applications concern large graphs

- Web document graph - 1 trillion webpages
  - Vertex = Webpage
  - Edge = Hyperlink

- Social networks - 2.23 billion users
  - Vertex = Users
  - Edge = Friendship relation

A graph of web links on the internet



Undirected Graphs – Definition

- An undirected Graph is defined as $G=(V,E)$ consisting of
  - Set V of vertices: $V = \{v_1, v_2, ..., v_n\}$
    - Number of vertices is denoted by $|V| = n$
  - Set E of unordered pairs $\{v_i, v_j\}$ termed edges
    - Edges connect the vertices

- Maximum number of edges in an undirected graph is $O(|V|^2)$

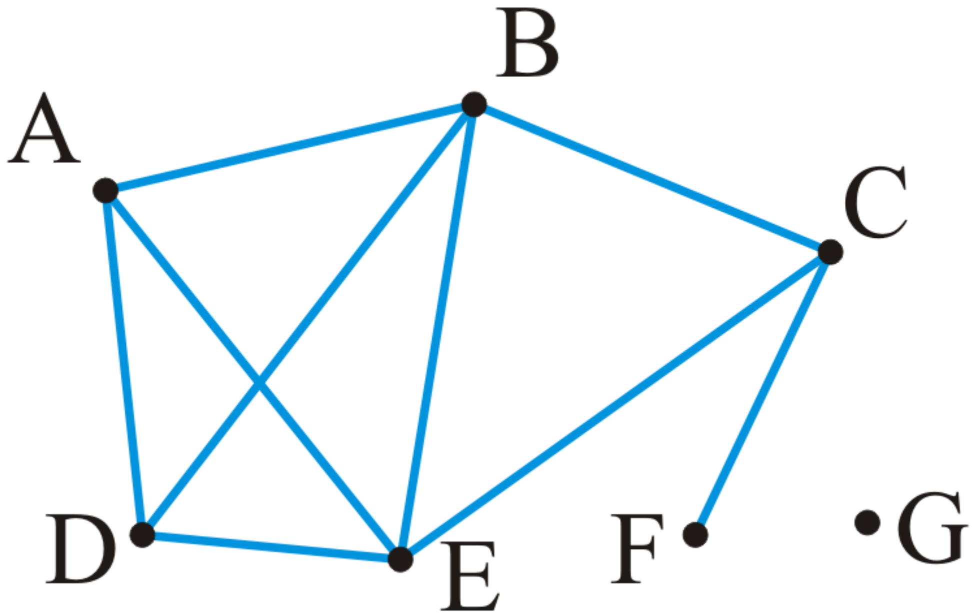
$$|E| \le \binom{|V|}{2} = \frac{|V|\left(|V|-1\right)}{2} = O\left(|V|^2\right)$$

  - Assumption: A vertex is never adjacent to itself
  - For example, $\{v_1, v_1\}$ will not define an edge

- Many data structures can implement abstract undirected graphs
  - Adjacency matrices, Adjacency lists (as discussed later in the lecture)

Degree

- Degree of a vertex is defined as the number of adjacent vertices
  - degree(A) = degree(D) = degree(C) = 3
  - degree(B) = degree(E) = 4
  - degree(F) = 1
  - degree(G) = 0



- Vertices adjacent to a given vertex are its neighbors

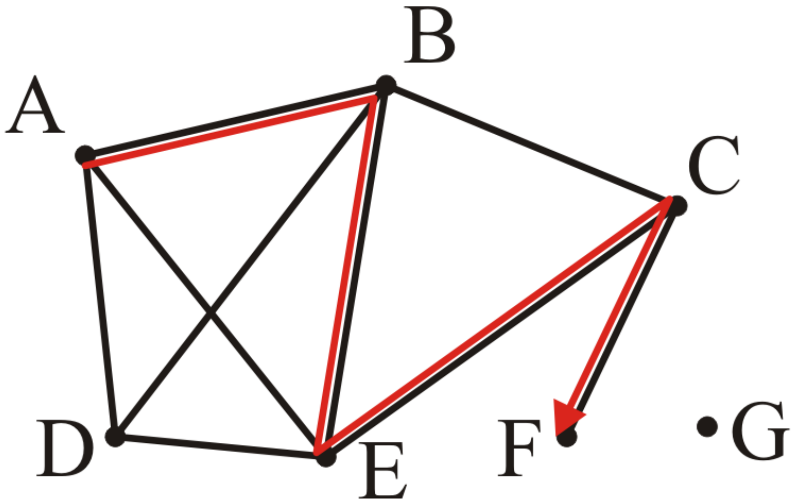
Path

- Path in an undirected graph is an ordered sequence of vertices
  - Consecutive vertices are connected through edges

- Path from vertex `0` to vertex `k` is $(v_0, v_1, v_2, ..., v_k)$
  - where $\{v_j - 1, v_j\}$ is an edge for $j = 1, ..., k$

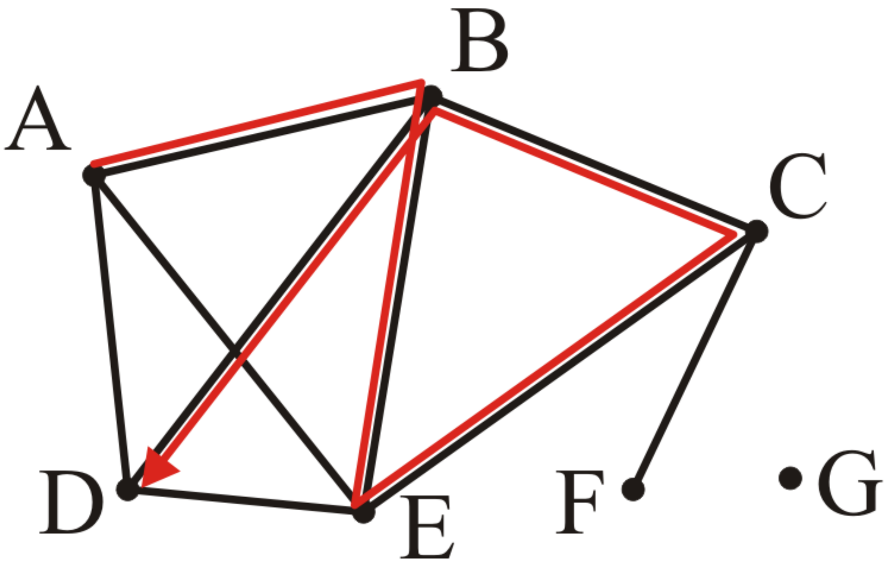
- Length of a path is equal to the number of edges

- Example: Path from A to F
  - Path: `(A, B, E, C, F)`
  - Length of the path is 4



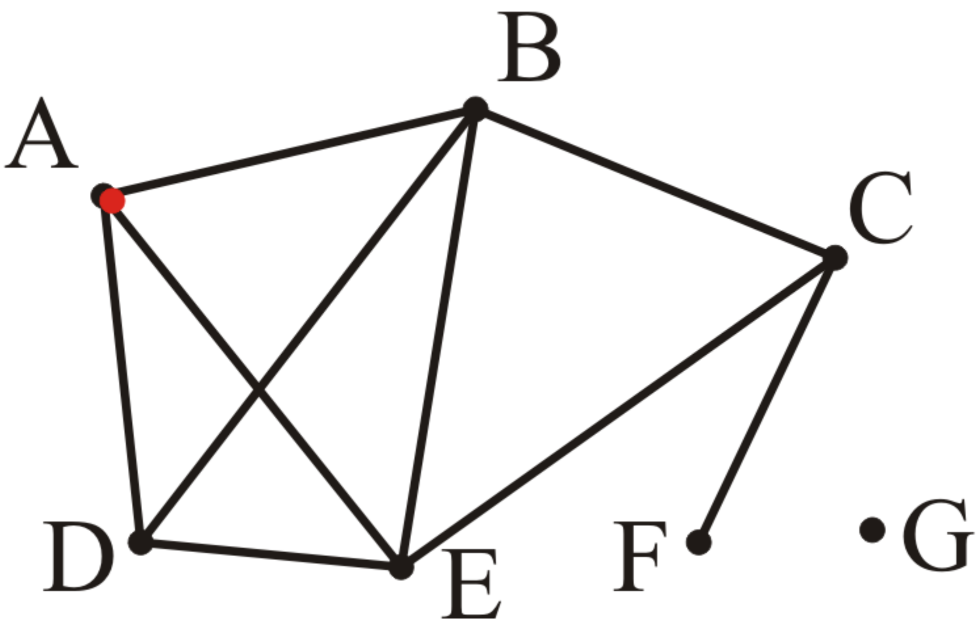
Path – Example

- Path of length 5: `(A, B, E, C, B, D)`
  - Repitition of vertex B



Path – Example

- A trivial path of length 0: `(A)`

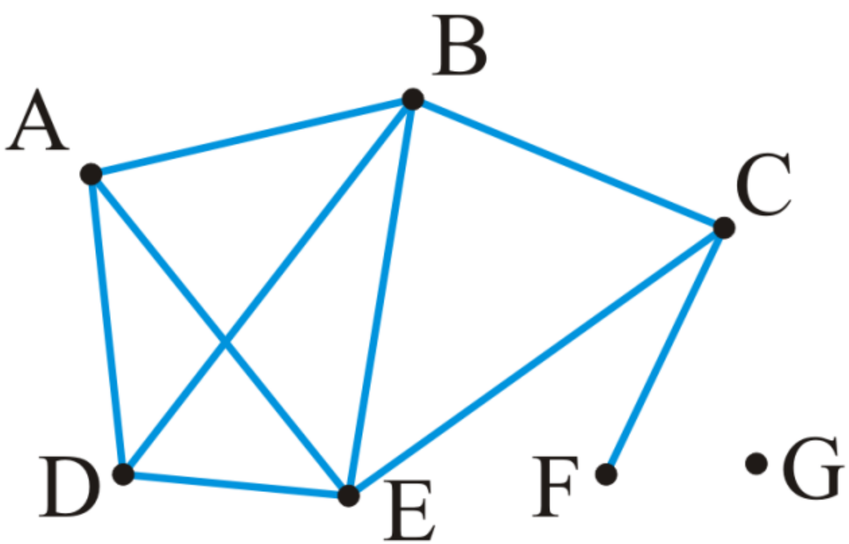


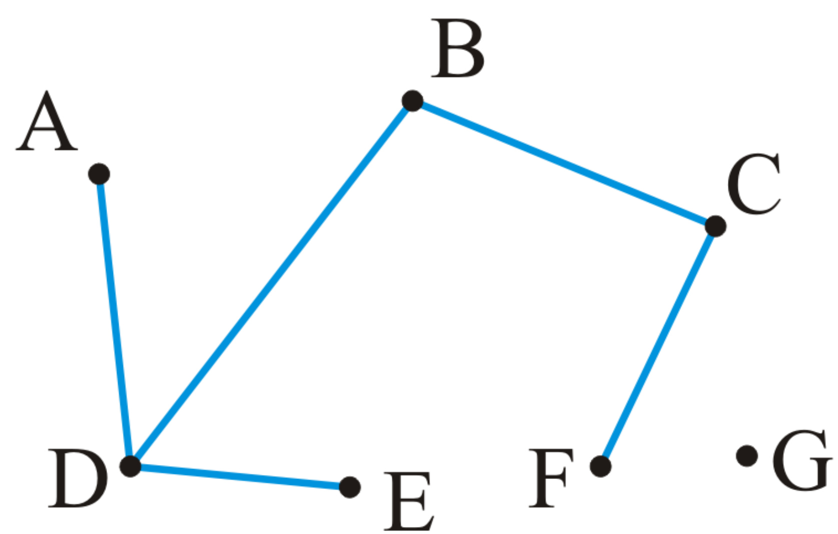
Path – Types

- A **simple path** has no repetitions other than perhaps the first and last vertices
- A **cycle** is a simple path of at least two vertices with the first and last vertices equal
  - Note: these definitions are not universal
- A **loop** is an edge from a vertex onto itself

Subgraph

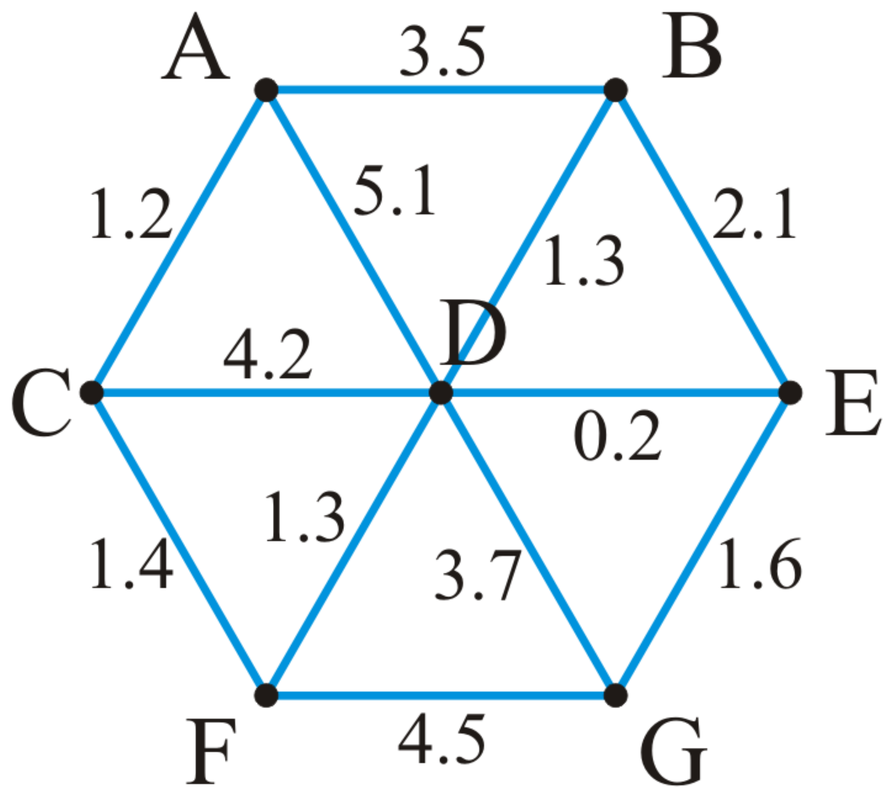
- A sub-graph of a graph G is defined by
  - Subset of the vertices
  - Subset of the edges that connected the subset of vertices in the original graph
- Every graph is a subgraph of itself





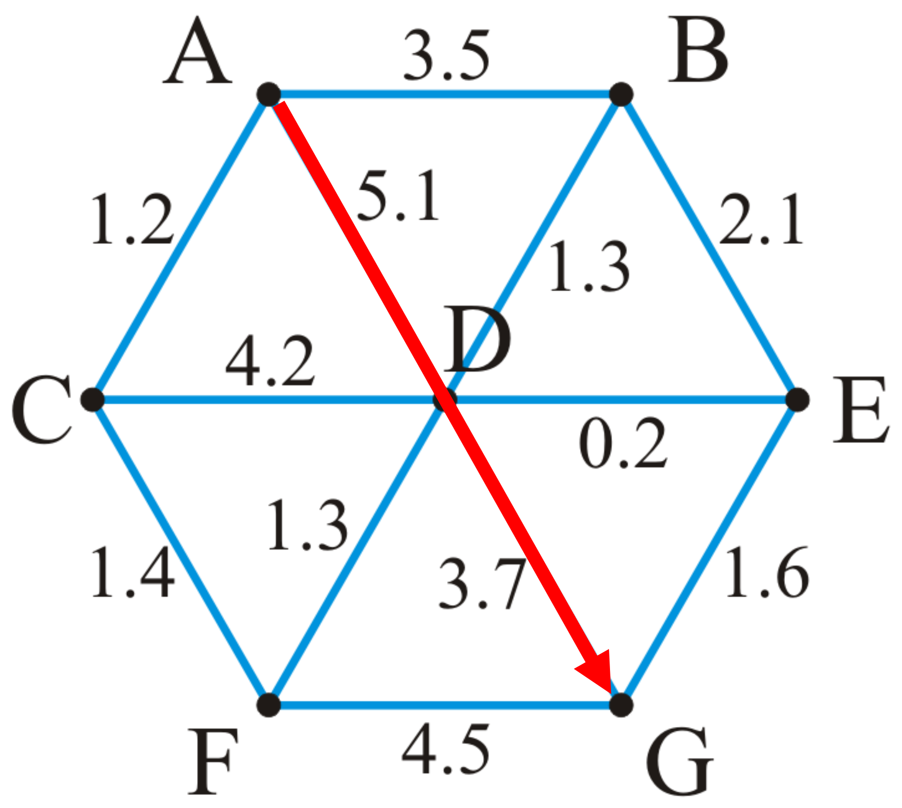
Weighted Graphs

- A weight may be associated with each edge in a graph
  - This could represent distance, energy consumption, cost, etc.
  - Such a graph is called a weighted graph
- Pictorially, we will represent weights by numbers next to the edges



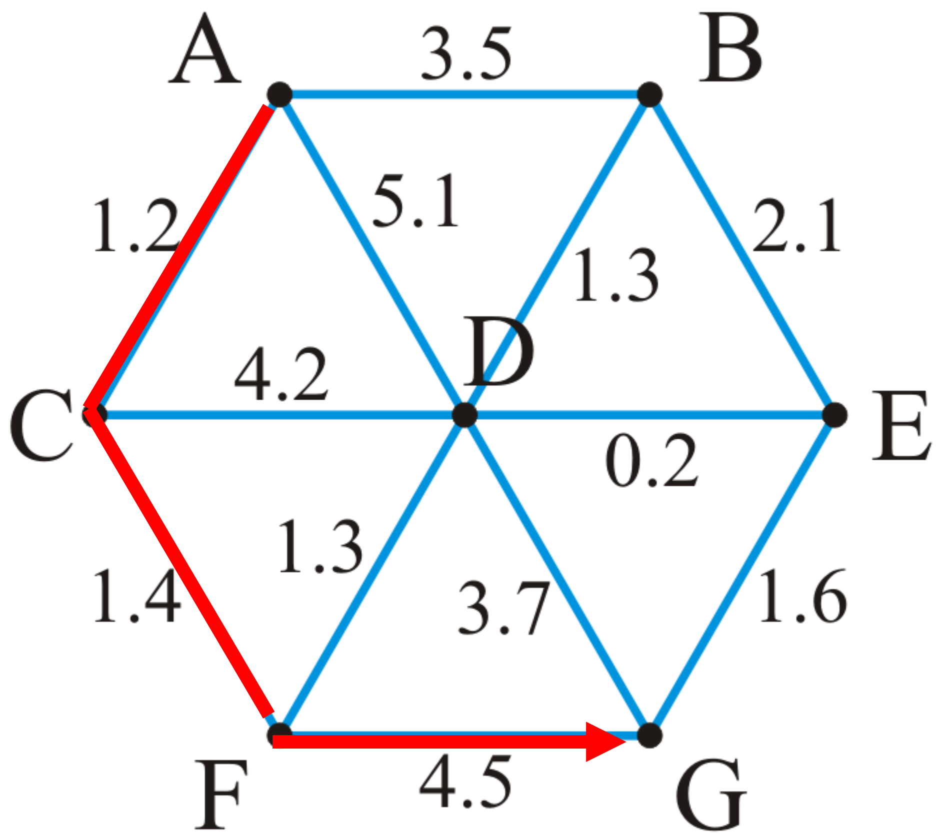
Weighted Graphs

- Length of a path within a weighted graph is the sum of all of the edges which make up the path
- The length of the path `(A, D, G)` in the following graph is `8.8`



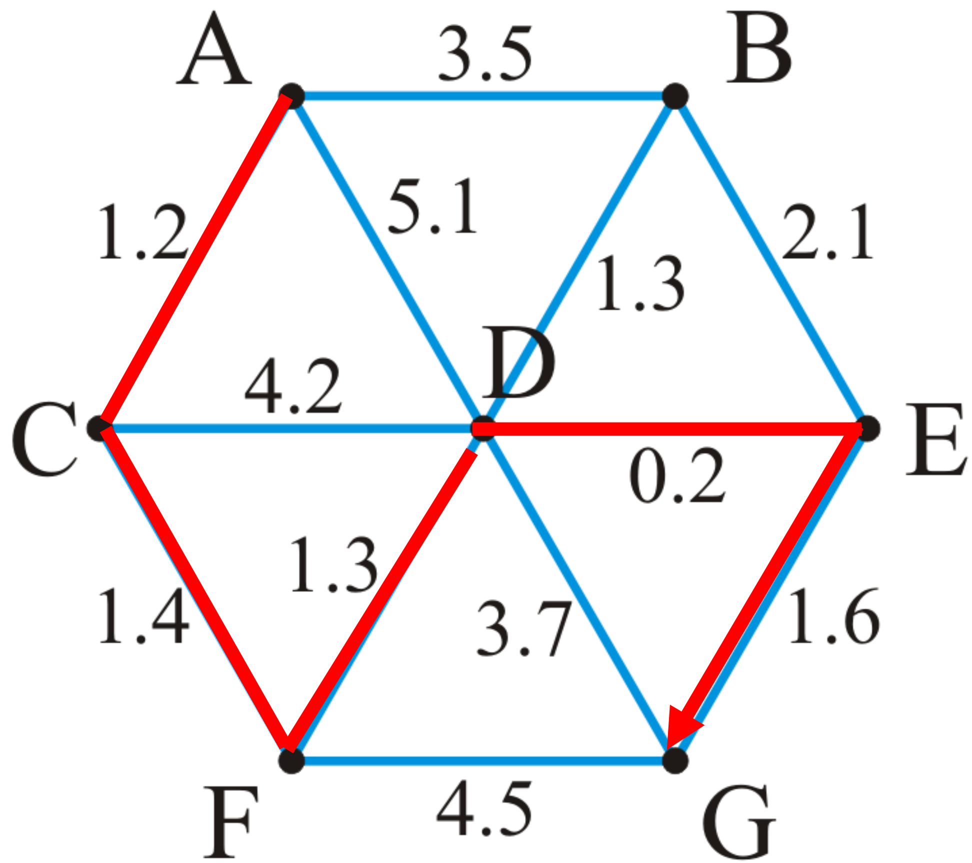
Weighted Graphs – Example

- Different paths may have different weights
  - Another path is `(A, C, F, G)` with length `1.2 + 1.4 + 4.5 = 7.1`



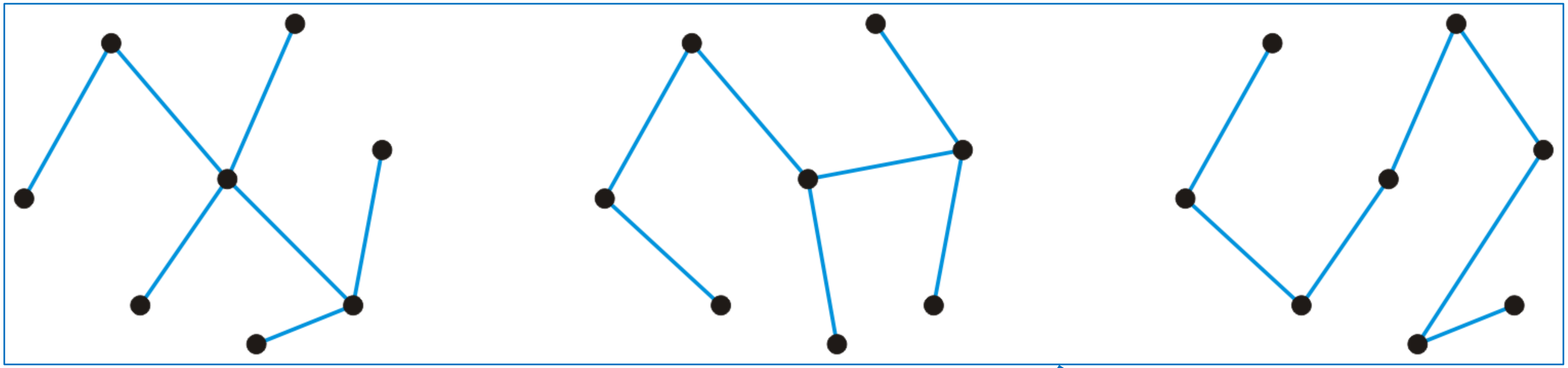
Weighted Graphs – Example

- Find the shortest path between two vertices A and G
- Shortest path is `(A, C, F, D, E, G)` with length `5.7`



Trees

- A graph is a tree if it satisfies the following two conditions
  - Graph is connected
  - There is a unique path between any two vertices



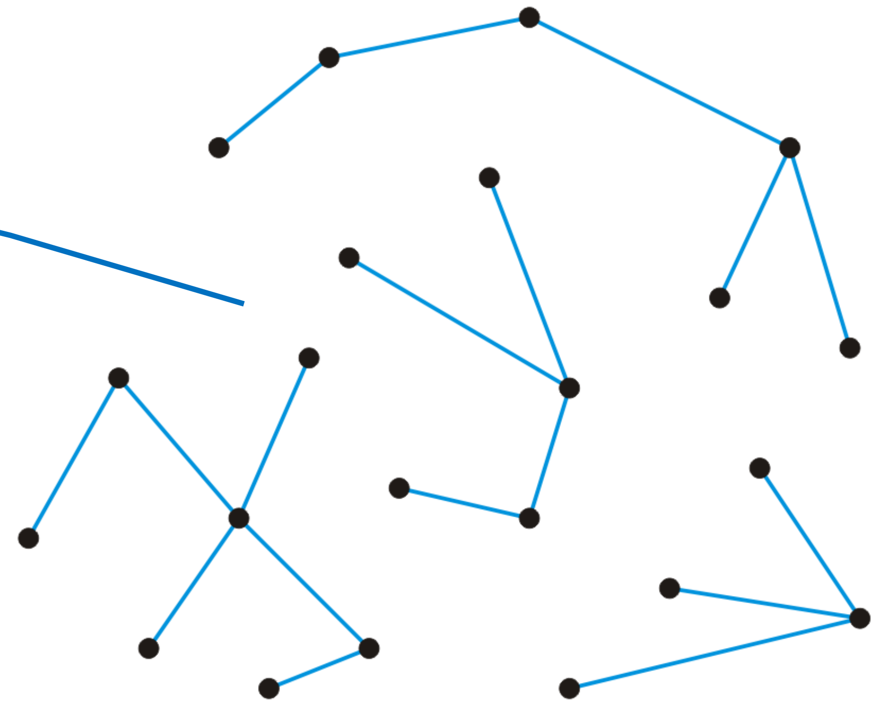
Three trees on same 8 vertices

- Consequences
  - The number of edges is `|E| = |V| - 1`
  - The graph is acyclic, that is, it does not contain any cycles
  - Adding one more edge must create a cycle
  - Removing any one edge creates two disjoint non-empty sub-graphs

Forest

- A forest is any graph that has no cycles
- Consequences
  - The number of edges is $|E| < |V|$
  - The number of trees is $|V| - |E|$
  - Removing any one edge adds one more tree to the forest

- Forest with 22 vertices and 18 edges
- Four trees

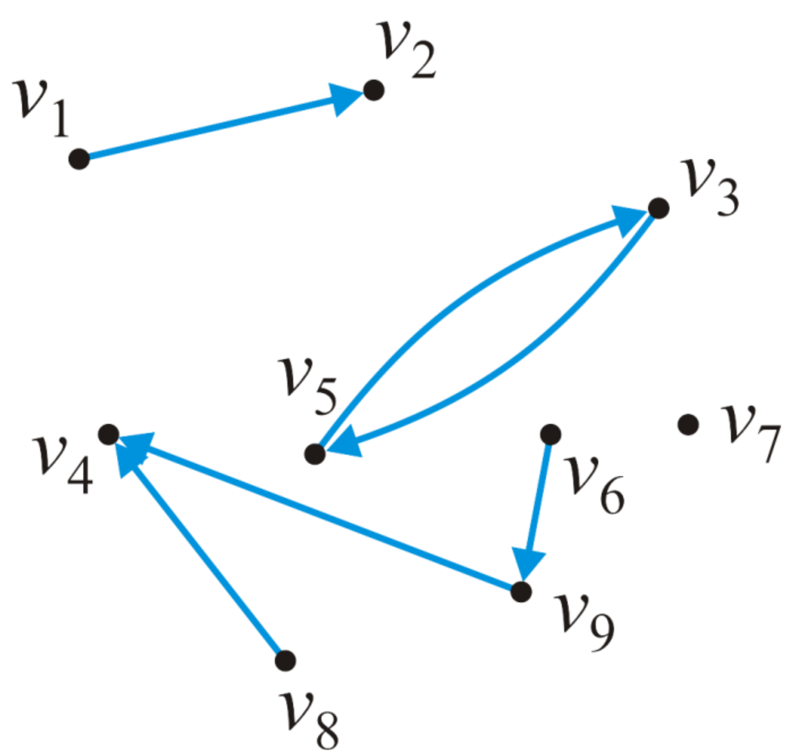


Directed Graphs

- In a directed graph, the edges on a graph are associated with a direction
  - Edges are ordered pairs (v_j, v_k) denoting a connection from v_j to v_k
  - The edge (v_j, v_k) is different from the edge (v_k, v_j)
- Streets are undirected graphs
  - In most cases, you can go two ways unless it is a one-way street

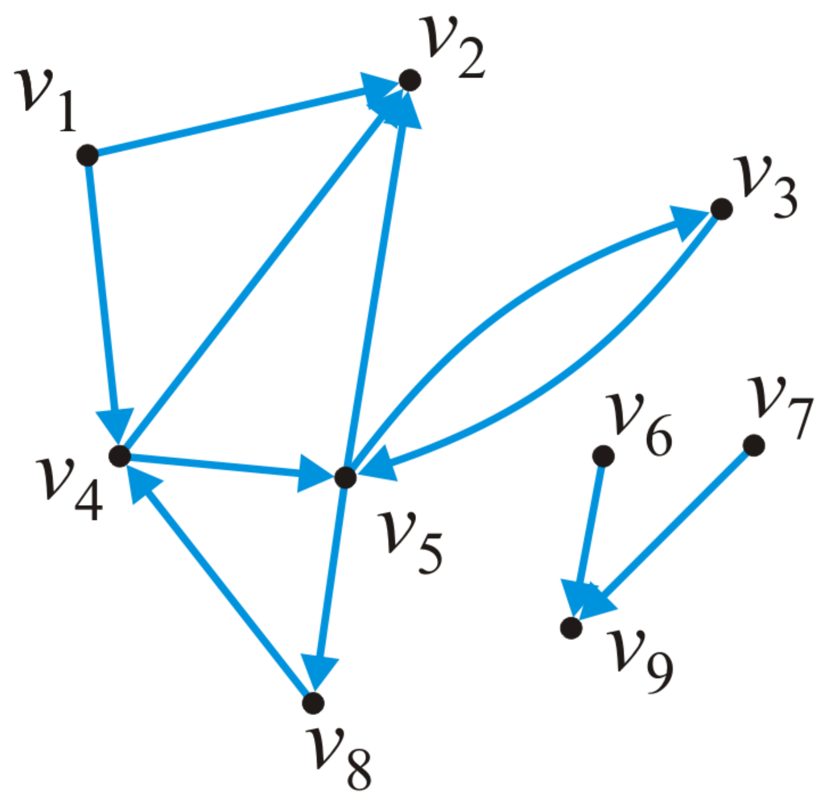
Directed Graphs

- Given our graph of nine vertices $V = \{v_1, v_2, \ldots v_9\}$
  - These six pairs (v_j, v_k) are directed edges
  - $E = \{(v_1, v_2), (v_3, v_5), (v_5, v_3), (v_6, v_9), (v_8, v_4), (v_9, v_4)\}$



In and Out Degree

- Degree of a vertex must be modified to consider both cases:
  - Out-degree of a vertex is the number of vertices which are adjacent to the given vertex
    - ➢ Number of outgoing edges
  - In-degree of a vertex is the number of vertices which this vertex is adjacent to
    - ➢ Number of incoming edges



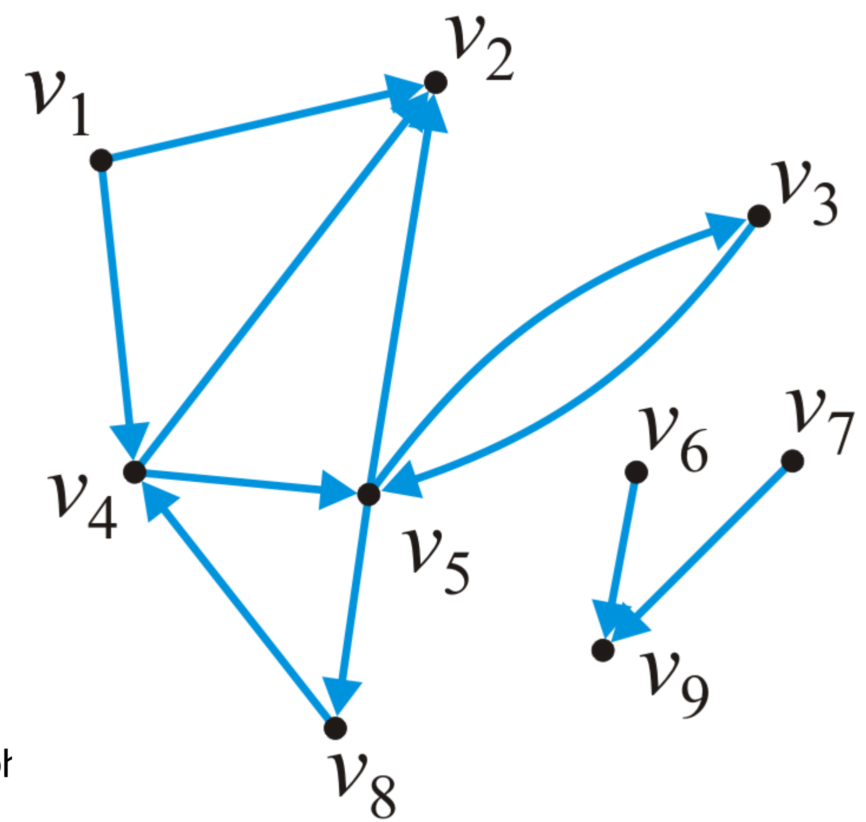
- In this graph:
  - In-degree(v_1) = 0 out-degree(v_1) = 2
  - In-degree(v_5) = 2 out-degree(v_5) = 3

Path

- A path in a directed graph is an ordered sequence of vertices
  - $(v_0, v_1, v_2, \ldots, v_k)$
  - where $(v_j - 1, v_j)$ is an edge for $j = 1, \ldots, k$

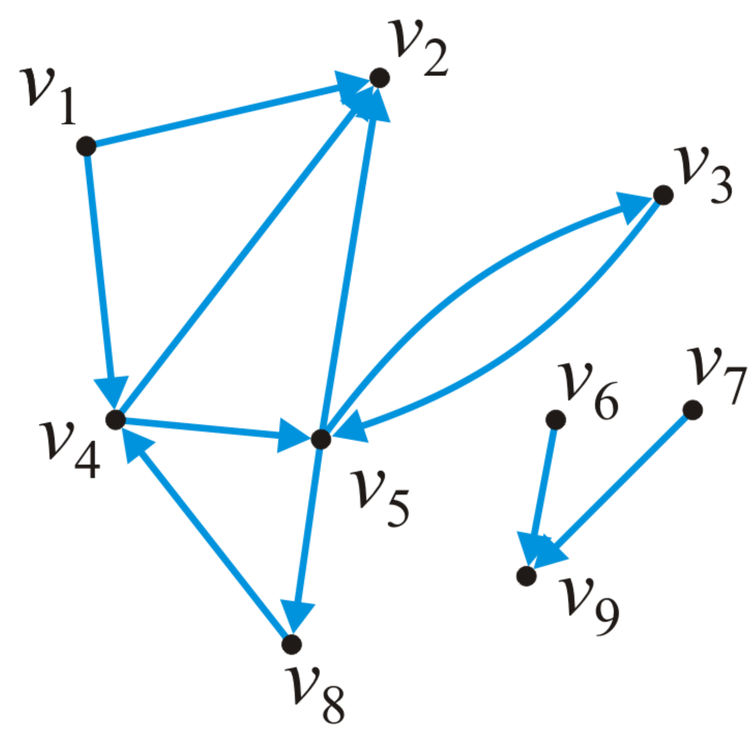
- A path of length 5 in this graph is
  - $(v_1, v_4, v_5, v_3, v_5, v_2)$

- A simple cycle of length 3 is
  - (v_8, v_4, v_5, v_8)



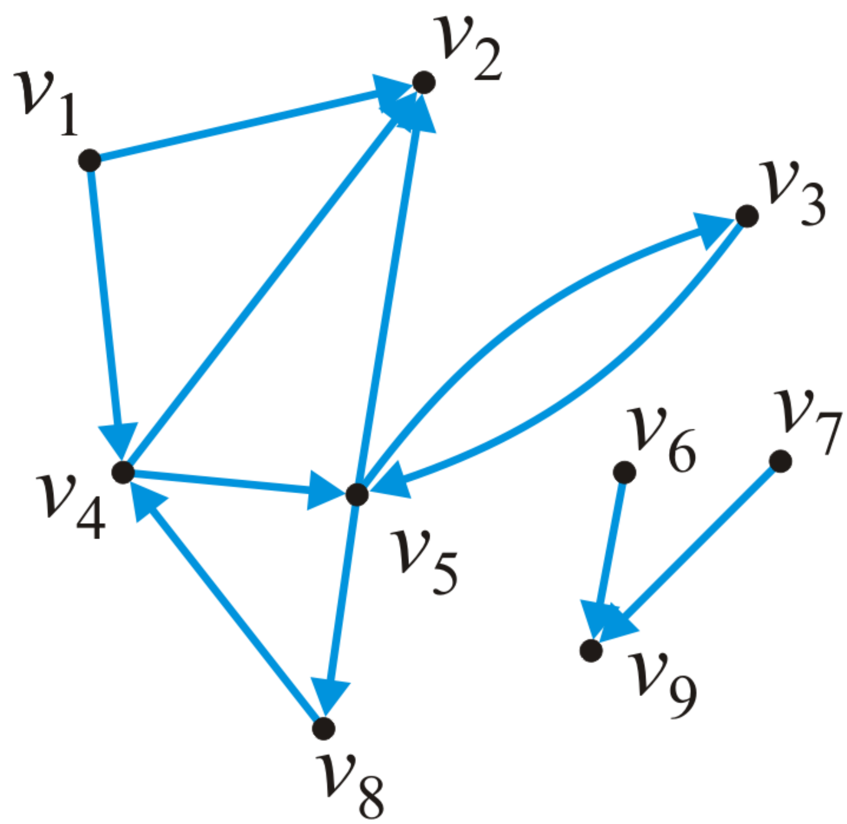
Connectedness

- Two vertices v_j, v_k are said to be connected if there exists a path from v_j to v_k
  - A graph is strongly connected if there exists a directed path between any two vertices
  - A graph is weakly connected if there exists a path between any two vertices that ignores the direction



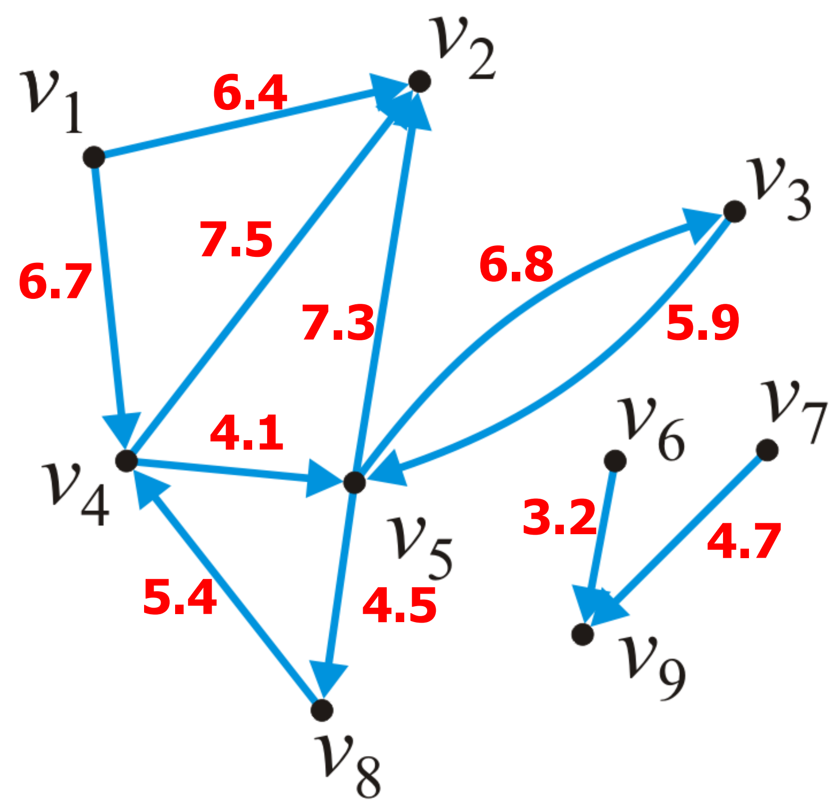
Connectedness – Example

- The sub-graph $\{v_3, v_4, v_5, v_8\}$ is strongly connected
- The sub-graph $\{v_1, v_2, v_3, v_4, v_5, v_8\}$ is weakly connected



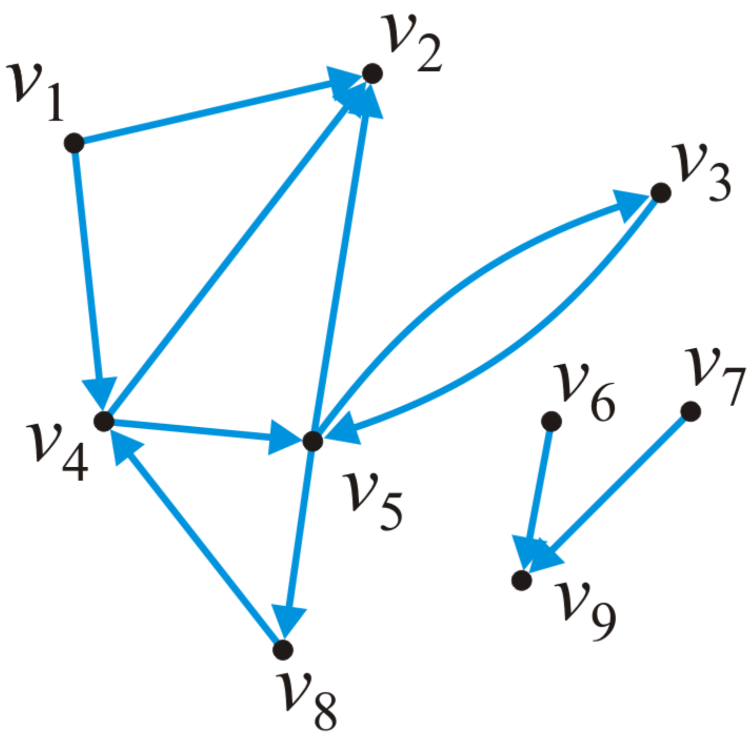
Weighted Directed Graphs

- Each edge is associated with a value
- If both (v_j, v_k) and (v_j, v_k) are edges
  - It is not required that they have the same weight



Representation

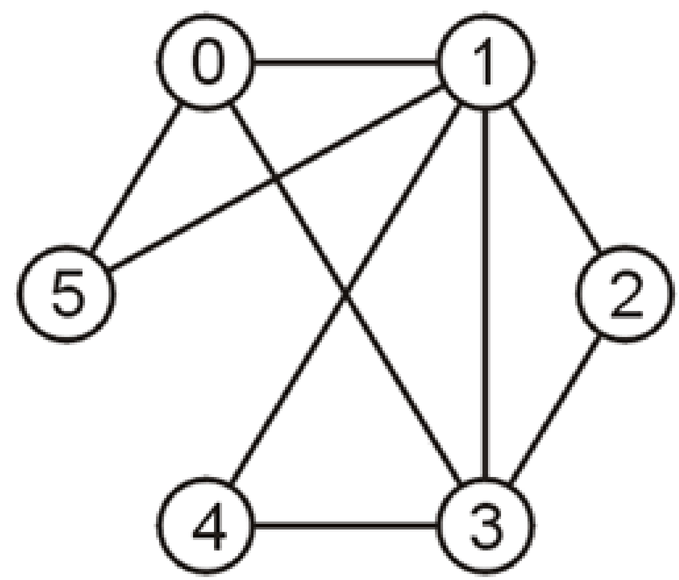
- How do we store the adjacency relations?
  - Adjacency matrix
  - Adjacency list



Adjacency Matrix

- Two dimensional matrix of size `n x n` where `n = |V|`
- `a[i, j] = 0 (F)` if there is no edge between vertices v_i and v_j
- `a[i, j] = 1 (T)` if there is an edge between vertices v_i and v_j
- Adjacency matrix of undirected graphs is symmetric
  - `a[i, j] = a[j, i]`

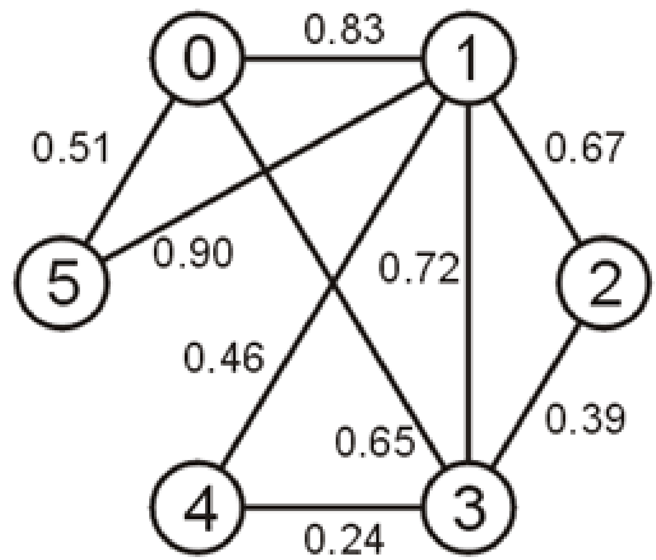
	0	1	2	3	4	5
0	F	T	F	T	F	T
1	T	F	T	T	T	T
2	F	T	F	T	F	F
3	T	T	T	F	T	F
4	F	T	F	T	F	F
5	T	T	F	F	F	F



Adjacency Matrix – Weighted Graph

- The matrix entry `[j, k]` is set to the weight of the edge (v_j, v_k)
- How to indicate absence of an edge in the graph?

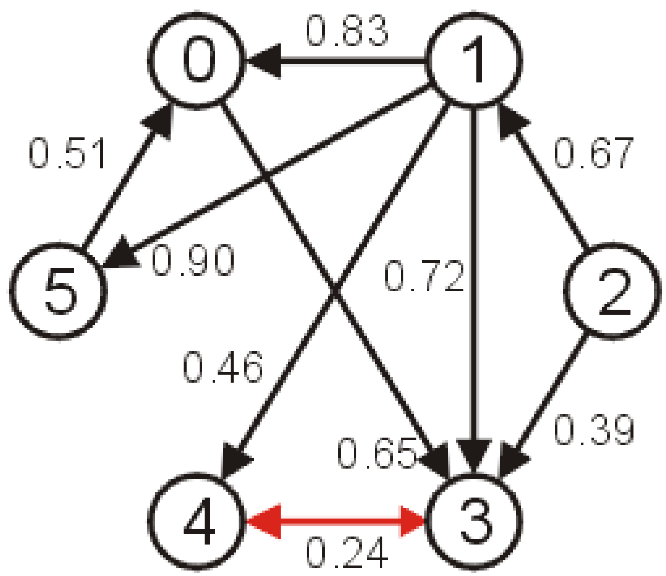
	0	1	2	3	4	5
0		0.83		0.65		0.51
1	0.83		0.67	0.72	0.46	0.90
2		0.67		0.39		
3	0.65	0.72	0.39		0.24	
4		0.46		0.24		
5	0.51	0.90				



Adjacency Matrix – Directed Graph

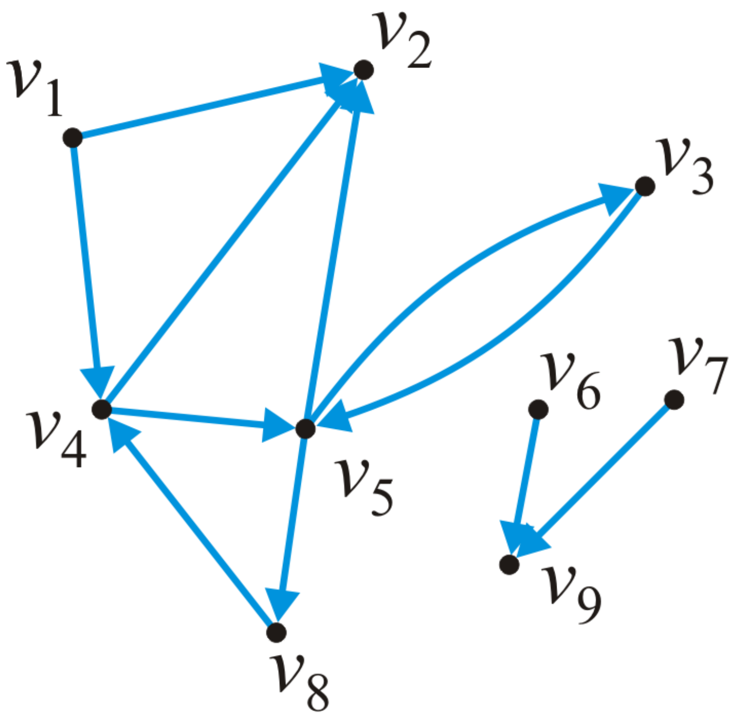
- For directed graph the matrix would not necessarily be symmetric

	0	1	2	3	4	5
0				0.65		
1	0.83			0.72	0.46	0.90
2		0.67		0.39		
3					0.24	
4				0.24		
5	0.51					



Adjacency Matrix – Analysis

	1	2	3	4	5	6	7	8	9
1		1		1					
2									
3					1				
4		1			1				
5		1	1					1	
6									1
7									1
8				1					
9									

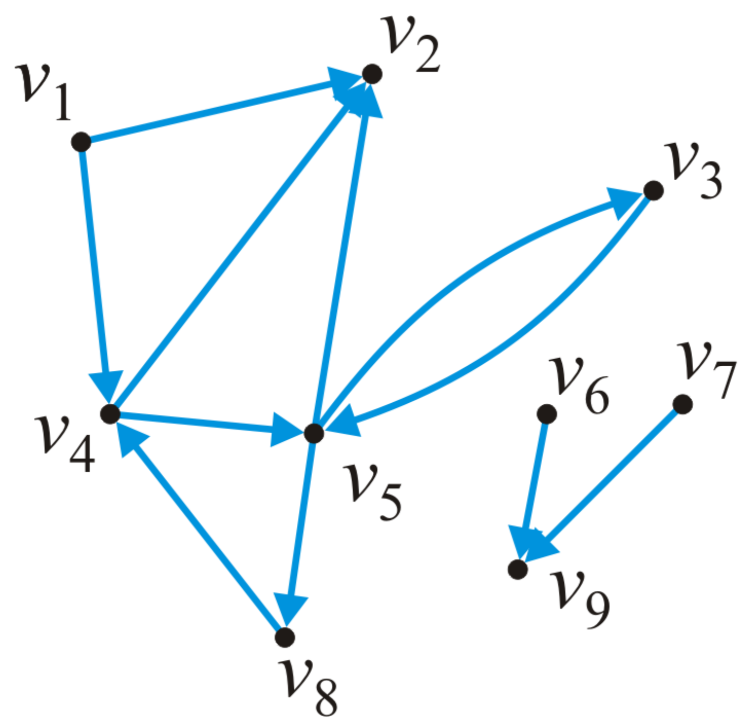


- Requires memory : $O(|V|^2)$
- Determining if v_j is adjacent to v_k : $O(1)$
- Finding all neighbors of v_j : $O(|V|)$

Adjacency Matrix – Problem

- Very sparsely populated
  - Out of 81 cells only 11 are 1 (or T)

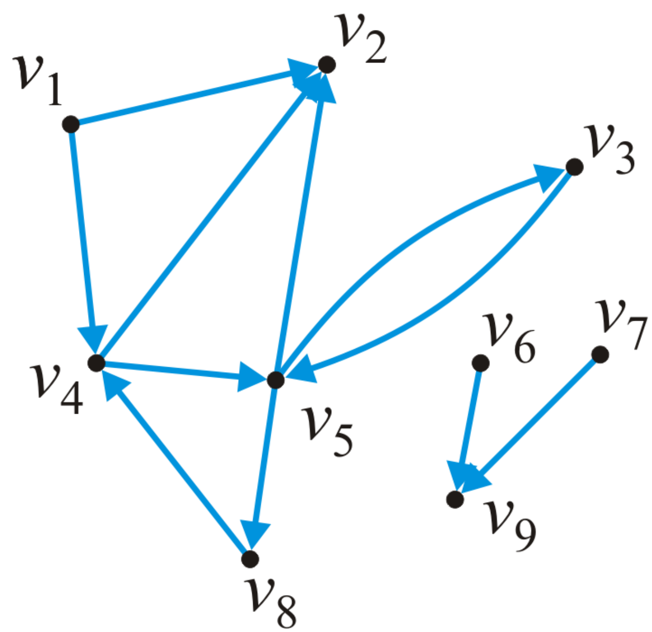
	1	2	3	4	5	6	7	8	9
1		1		1					
2									
3					1				
4		1			1				
5		1	1					1	
6									1
7									1
8				1					
9									



Adjacency List

- Each vertex is associated with a list of its neighbors
  - A vertex `w` is inserted in the list for vertex `v` if edge `(v, w)` exists

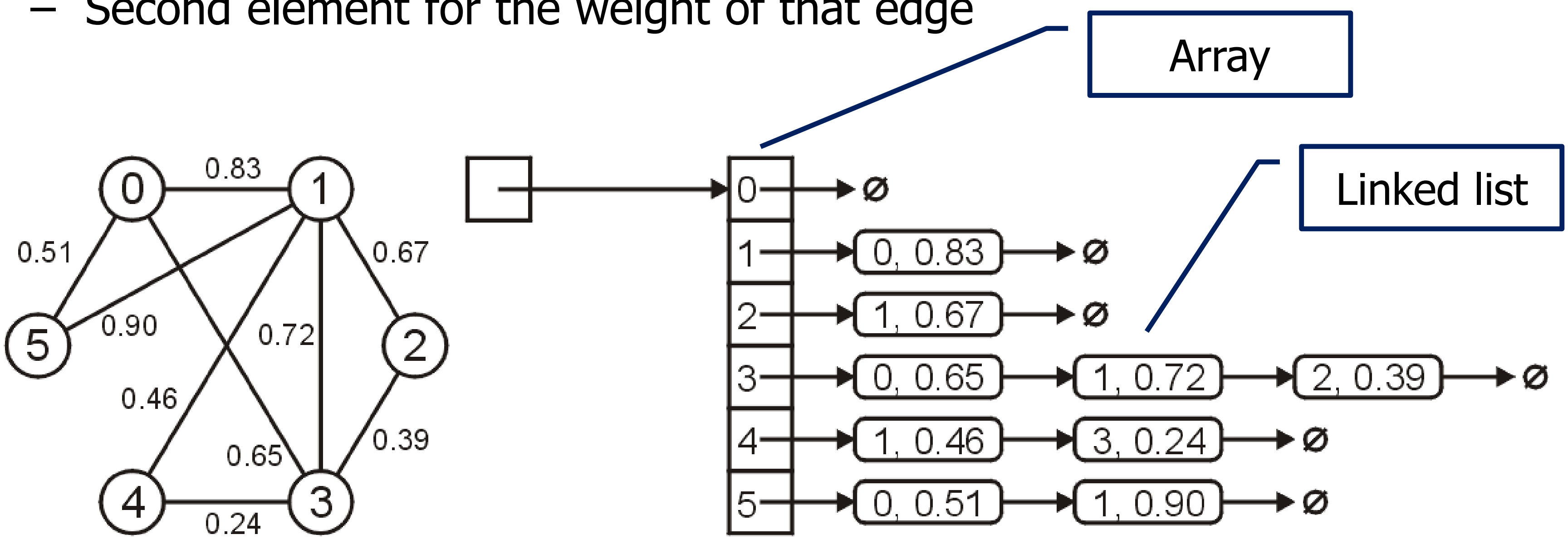
1	•	→ 2 → 4
2	•	
3	•	→ 5
4	•	→ 2 → 5
5	•	→ 2 → 3 → 8
6	•	→ 9
7	•	→ 9
8	•	→ 4
9	•	



- Requires memory : `O(|V| + |E|)`

Adjacency List – Weighted Graphs

- An adjacency list for a weighted graph contains two elements
  - First element for the vertex
  - Second element for the weight of that edge

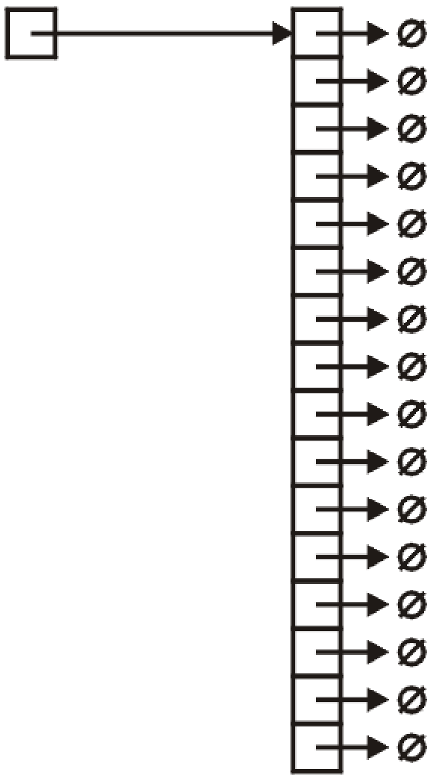


- When the vertices are identified by a name (i.e., string)
  - Hash-table of lists is used to implement the adjacency list

Adjacency List – Implementation

- Node to store adjacent vertex and weight of the edge

```
class SingleNode {
   private:
      int adjacent_vertex;
      double edge_weight;
      SingleNode * next_node;
};
```



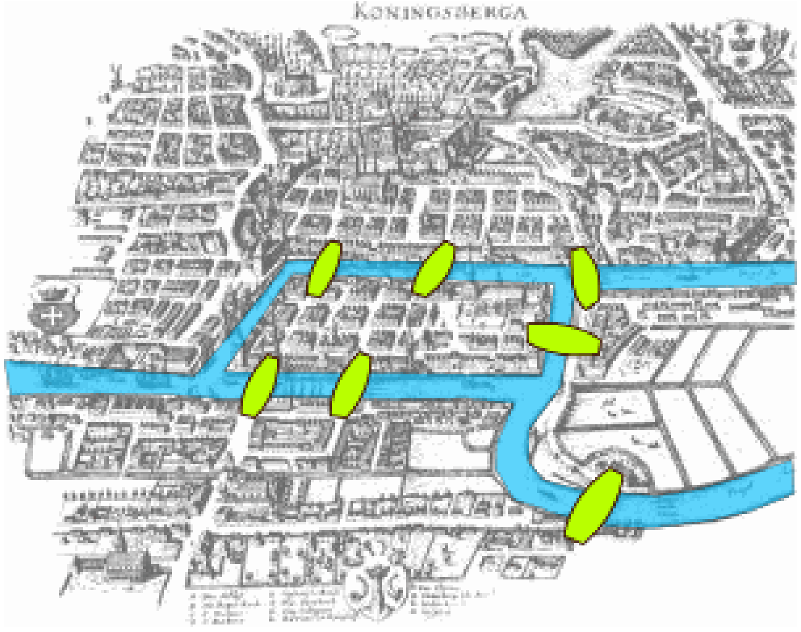
- Define and create Array

```
SingleNode * array;
array = new SingleNode[16];
```

Graph Problems – Euler Tour

- A sequence of vertices that traverse all edges in the graph exactly once
  - Leonhard Euler in 1736
  - Laid the foundations of graph theory

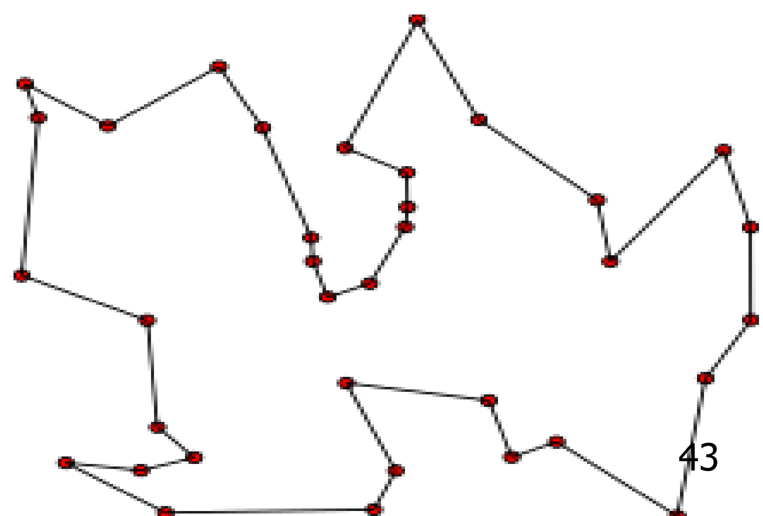
- Problem: To devise a walk through the city that would cross each of the seven bridges of Königsberg once and only once
  - Euler proved problem has no solution



Graph Problems – Traveling Salesman

- A salesman wishes to
  - Visit a number of towns, and then
  - Return to his starting town
- Given the travelling times between towns, how should the travel be planned, so that:
  - He visits each town exactly once, and
  - He travels in as short time as possible

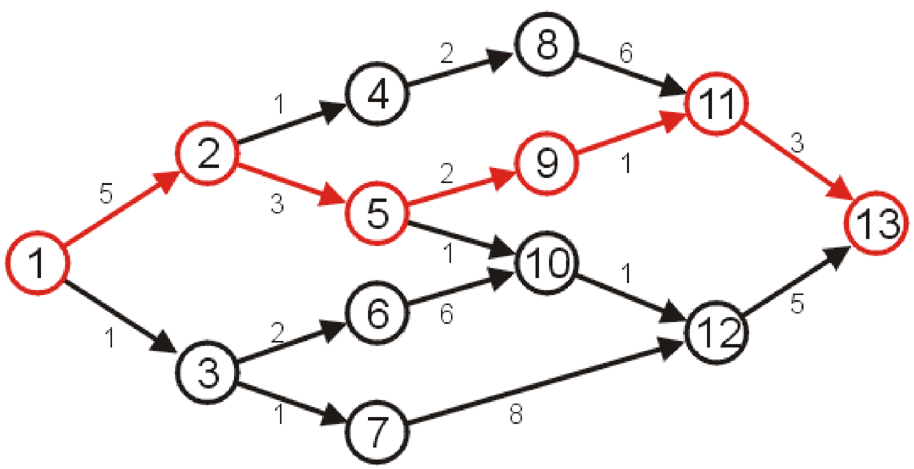
- Problem: Given a weighted graph G, provide shortest cycle that contains all vertices in G
  - NP-Hard problem

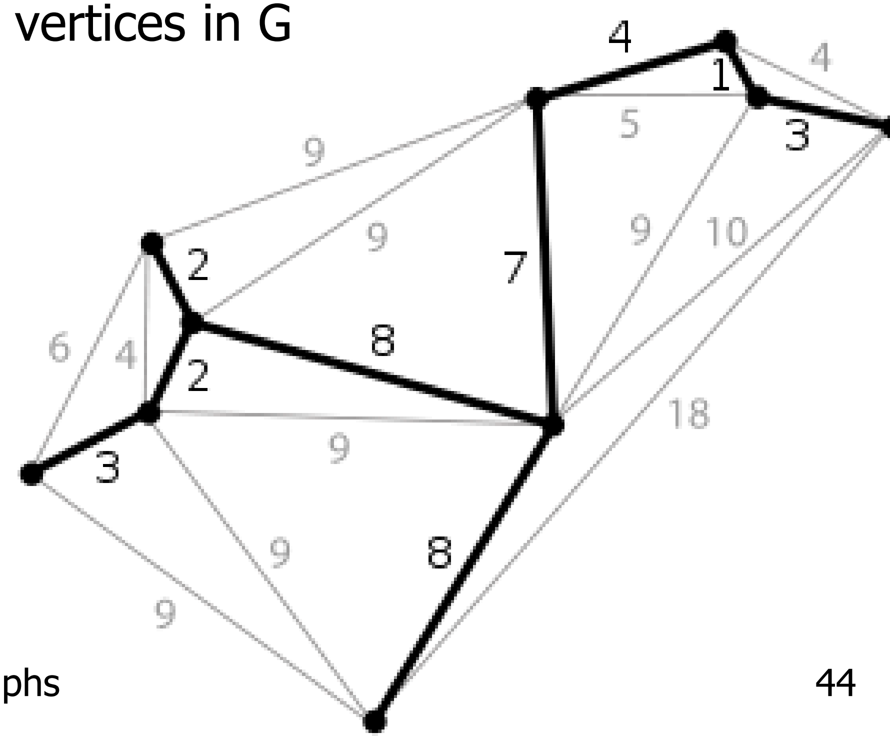


Graph Problems – Others

- Minimum-cost spanning tree
  - Given a weighted graph G, determine a spanning tree with minimum total edge cost

- Single-source shortest path
  - Given a weighted graph G and a source vertex v in G, determine the shortest paths from v to all other vertices in G





Graph Traversal

- Given a graph G = (V, E), directed or undirected
  - Goal is to methodically explore every vertex and every edge
- Traversals of graphs are also called searches
- We can use either breadth-first or depth-first traversals
  - Breadth-first requires a queue
  - Depth-first requires a stack

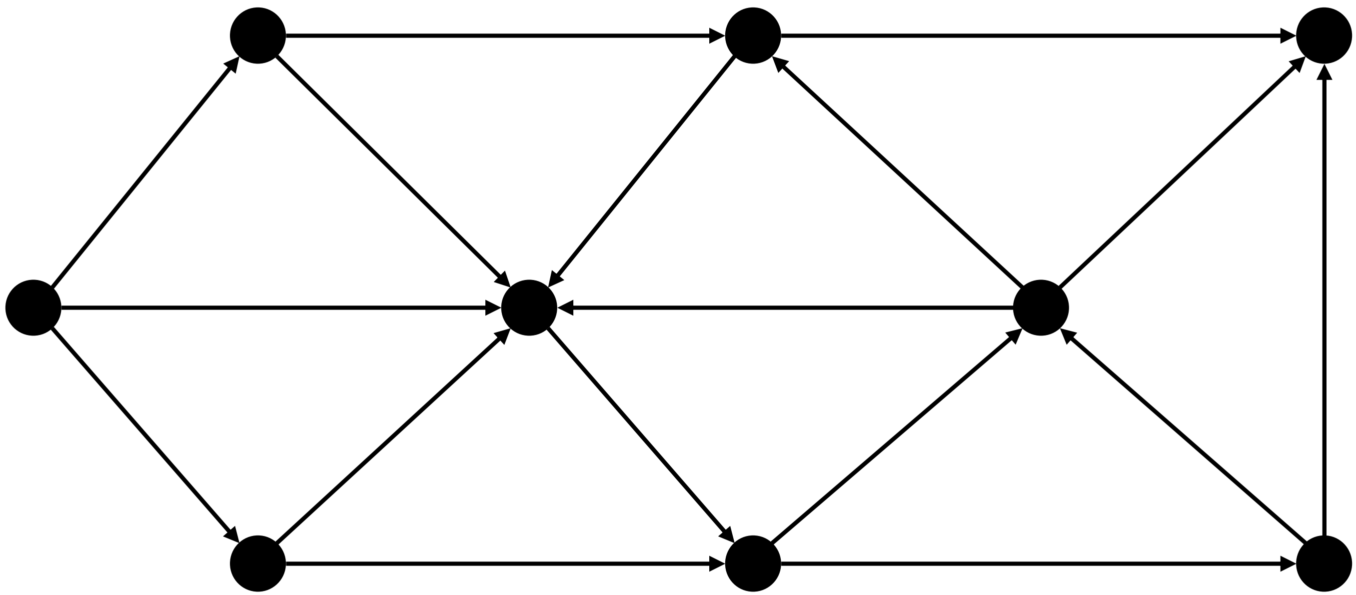
Breadth-First Search

- Choose any vertex, mark it as visited and enqueue it onto queue
- While the queue is not empty
  - Dequeue top vertex v from the queue
  - For each vertex adjacent to v that has not been visited
    - Mark it visited, and
    - Enqueue it onto the queue

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
    remove the head u of Q (Dequeue)
    mark and enqueue all (unvisited) neighbors of u
```

- The above algorithm continues until the queue is empty!
  - If there are no unvisited vertices, the graph is connected

Breadth-First Search – Example

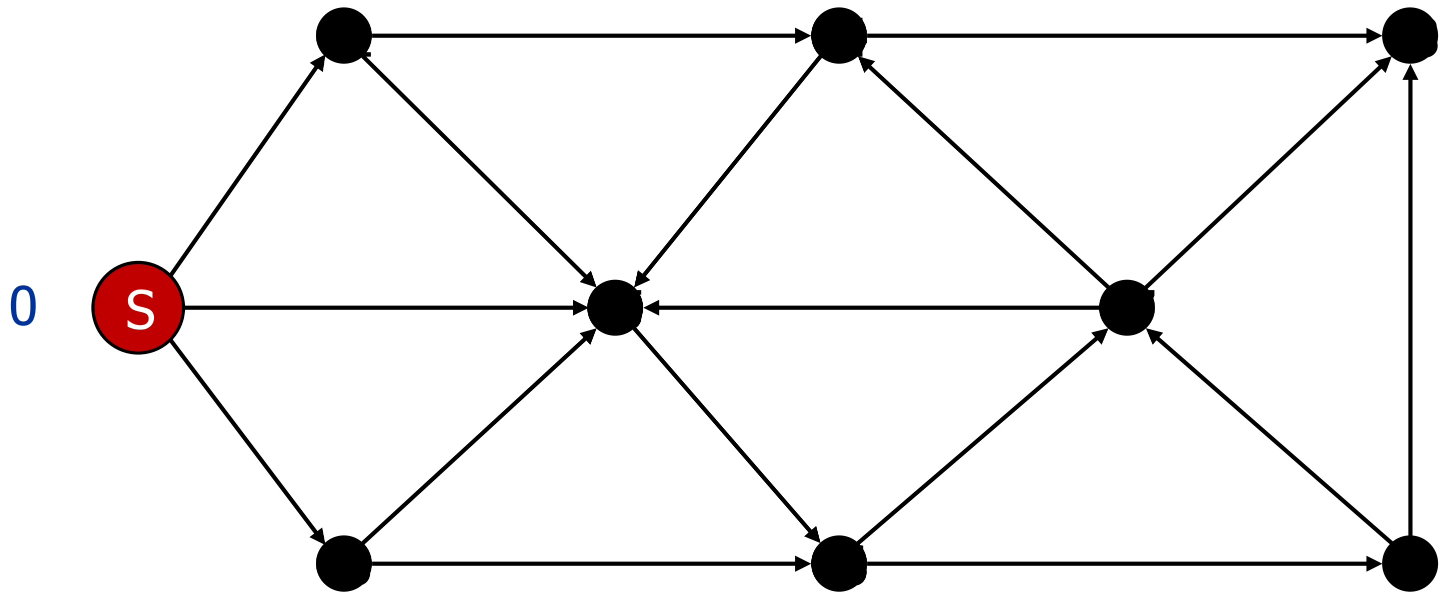


Undiscovered
Discovered
Top of queue
Finished

Queue (Q):

```
1: Create a Queue Q
```

Breadth-First Search – Example

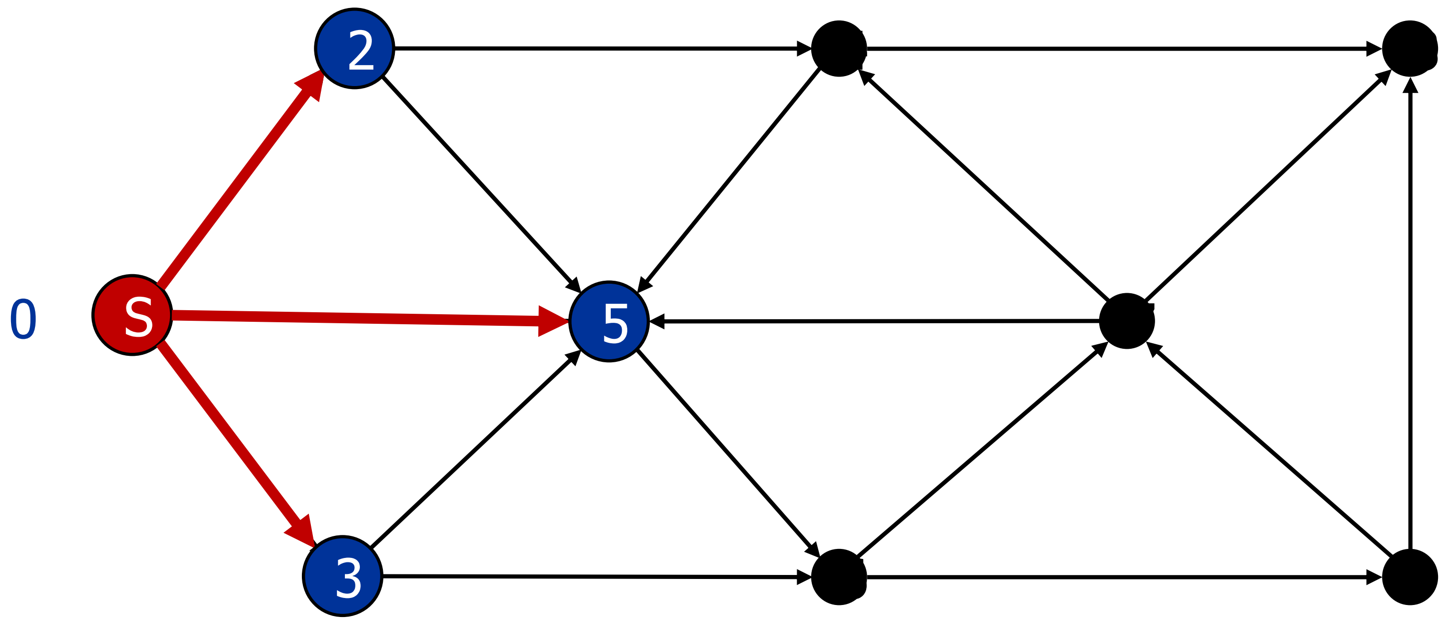


Queue (Q):

S

`2: Mark S as visited and put S into Q`

Breadth-First Search – Example

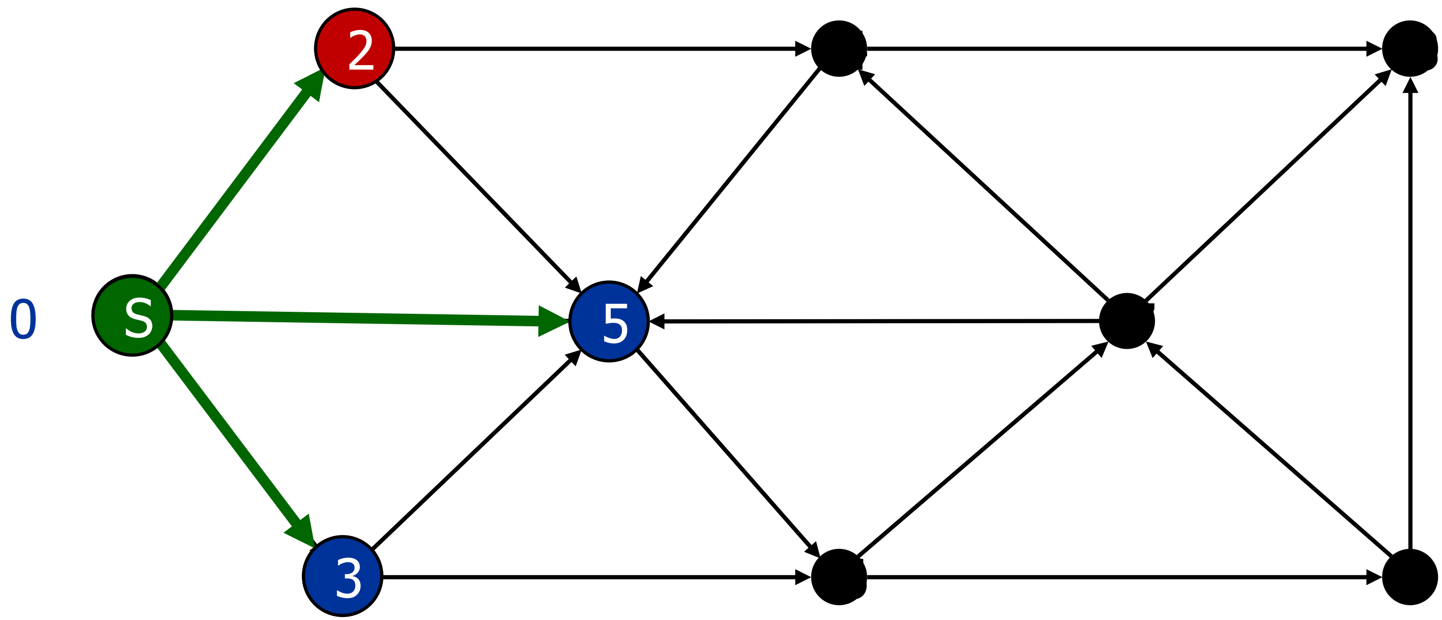


Queue (Q):

S

```
3: While Q not empty
4:     v = dequeue Q  (i.e., S)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



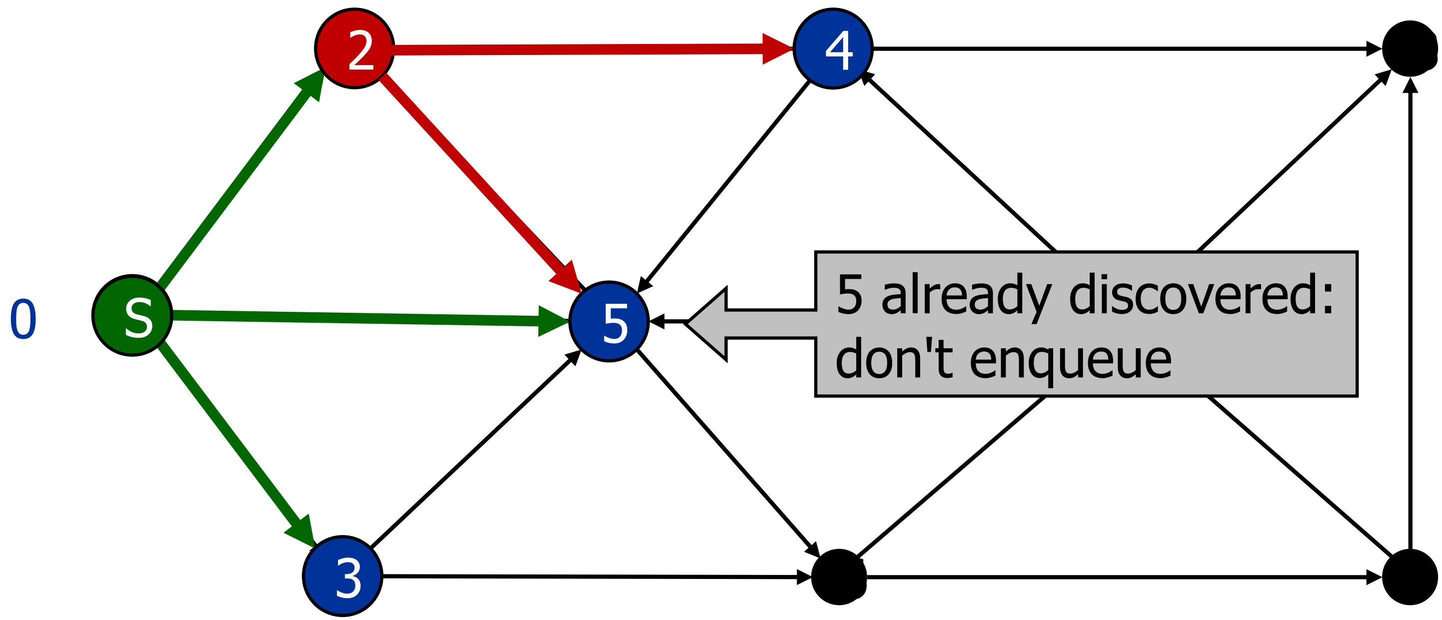
Undiscovered
Discovered
Top of queue
Finished

Queue (Q):

2 3 5

```
3: While Q not empty
4:     v = dequeue Q (i.e., 2)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



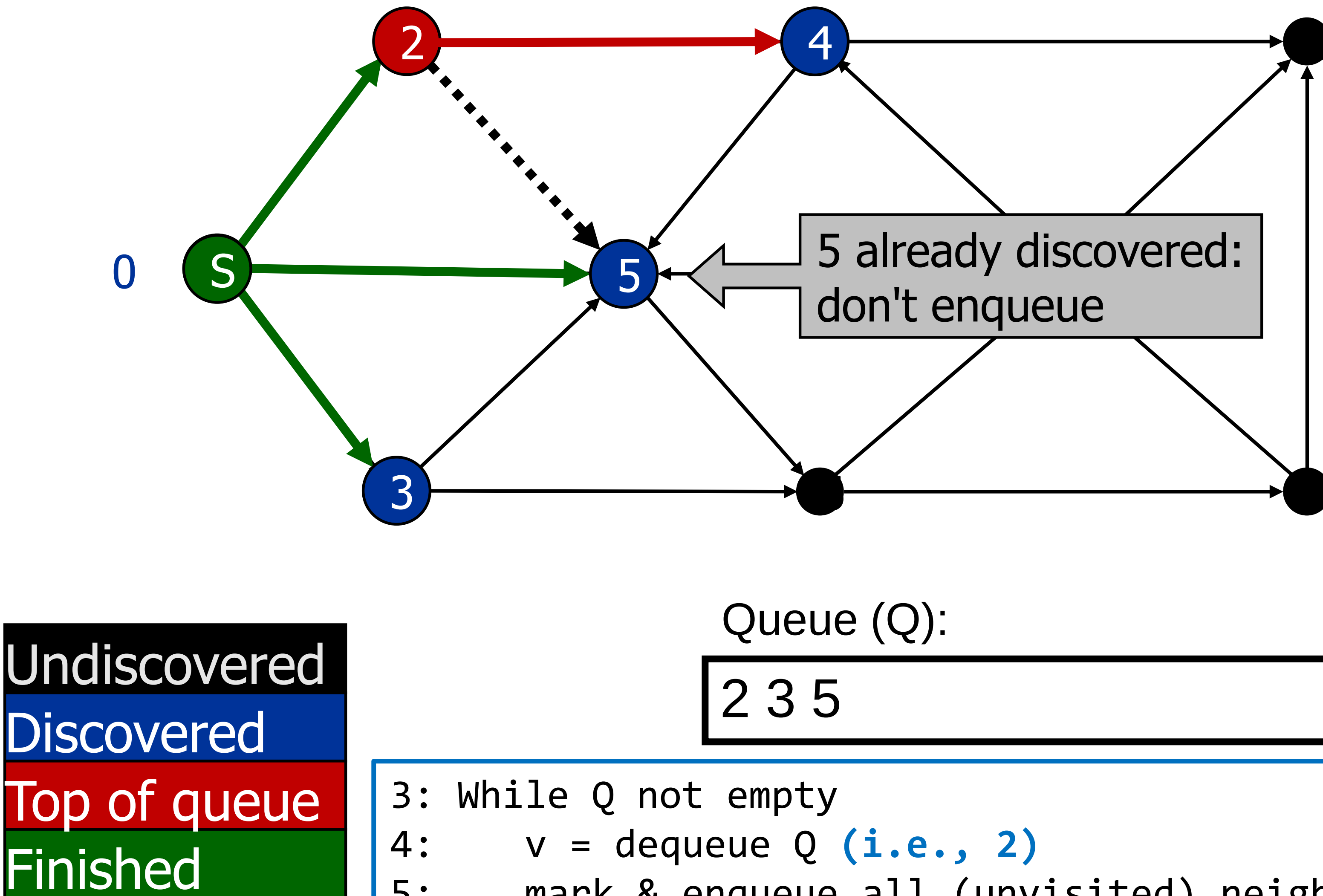
Undiscovered
Discovered
Top of queue
Finished

Queue (Q):

2 3 5

```
3: While Q not empty
4:     v = dequeue Q (i.e., 2)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example

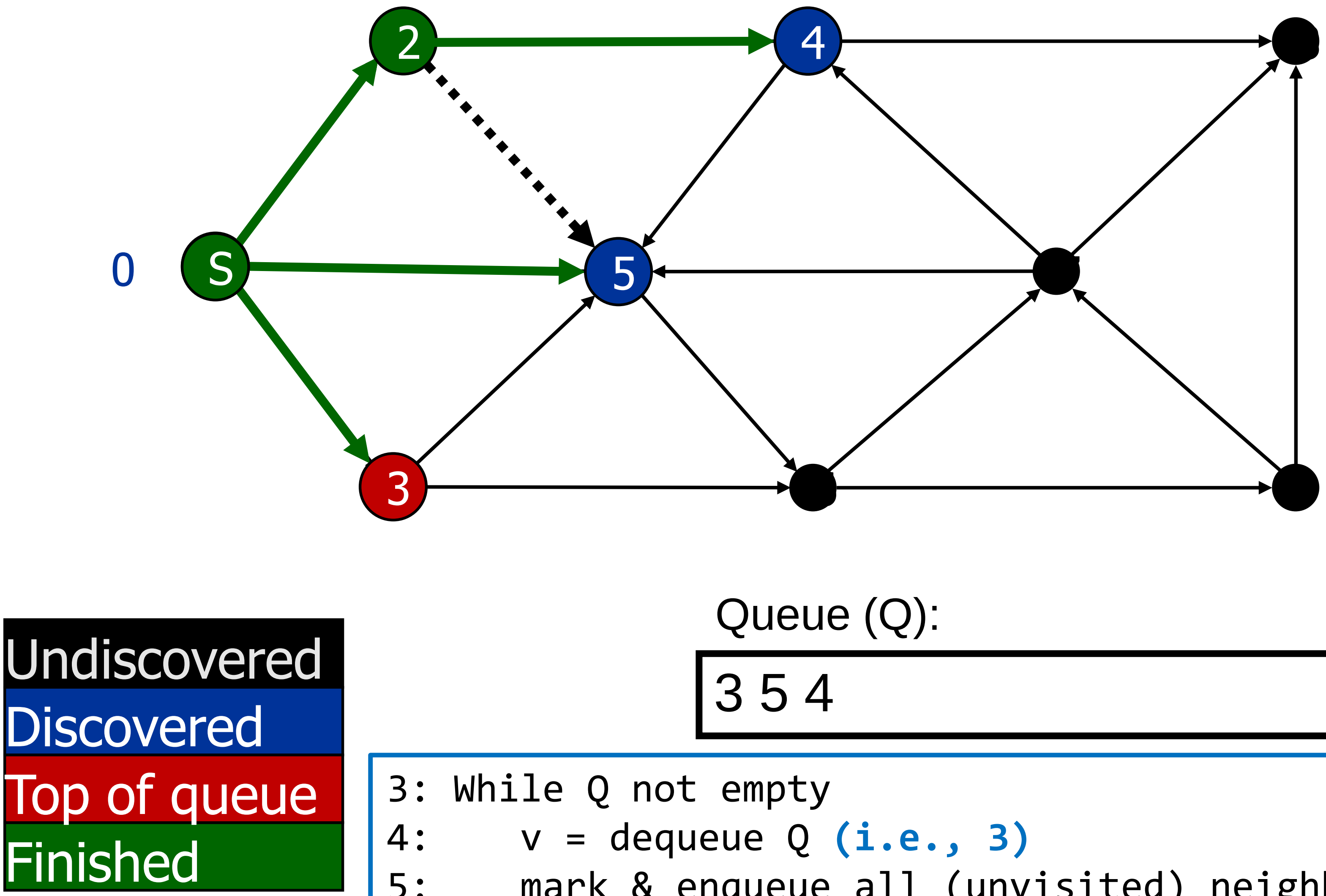


Queue (Q):

2 3 5

```
3: While Q not empty
4:    v = dequeue Q (i.e., 2)
5:    mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example

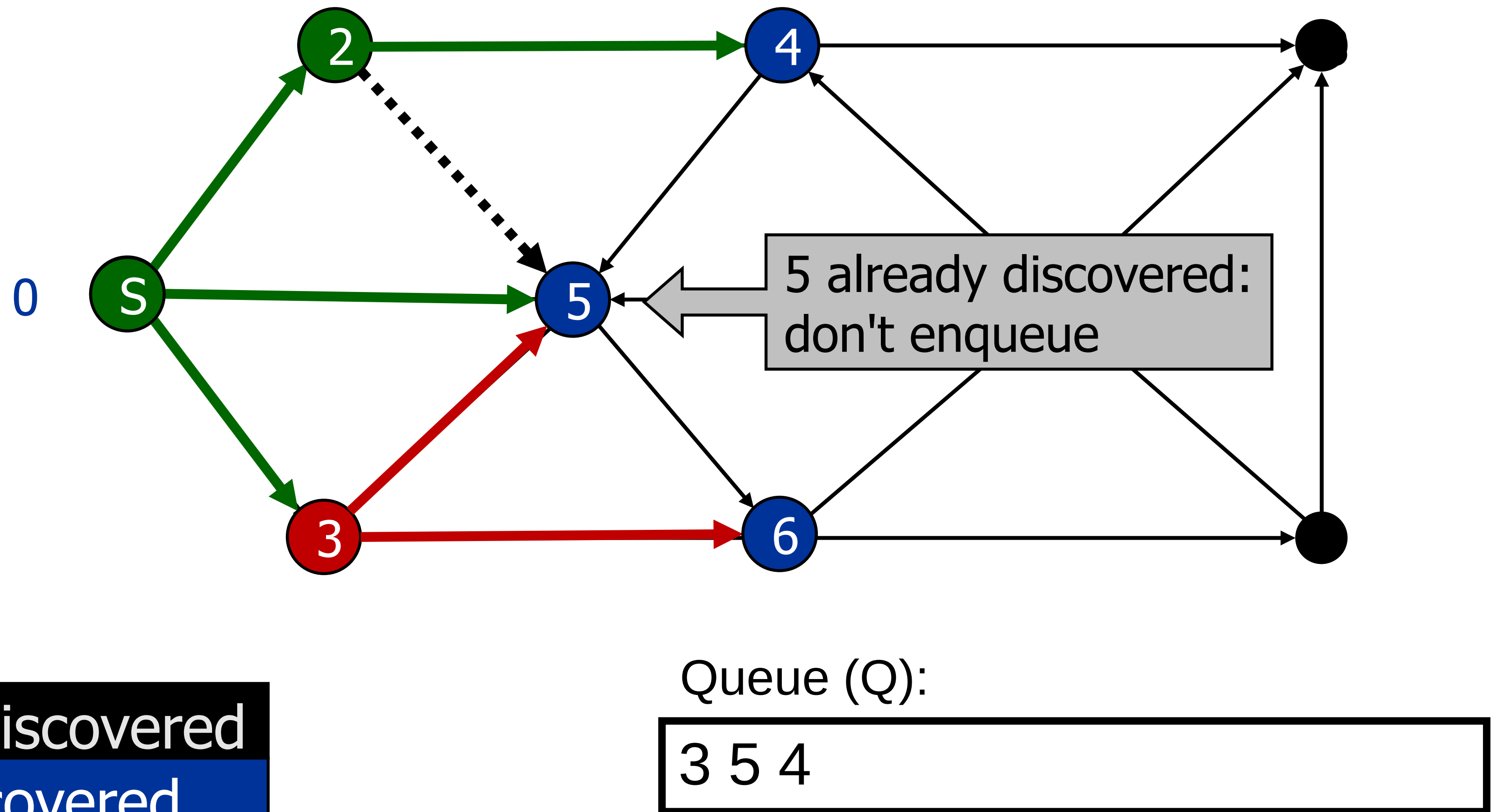


Queue (Q):

3 5 4

```
3: While Q not empty
4:     v = dequeue Q (i.e., 3)
5:     mark & enqueue all (unvisited) neighbors of v
```

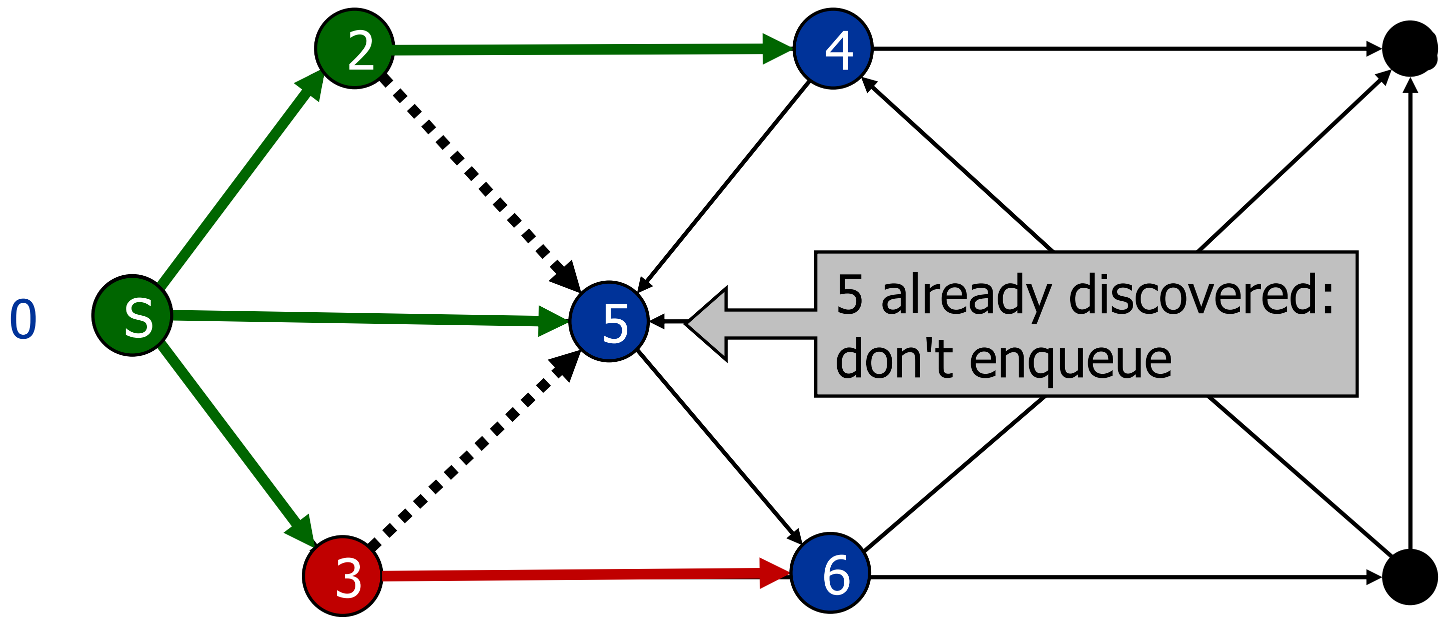
Breadth-First Search – Example



Undiscovered
Discovered
Top of queue
Finished

```
3: While Q not empty
4:     v = dequeue Q (i.e., 3)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



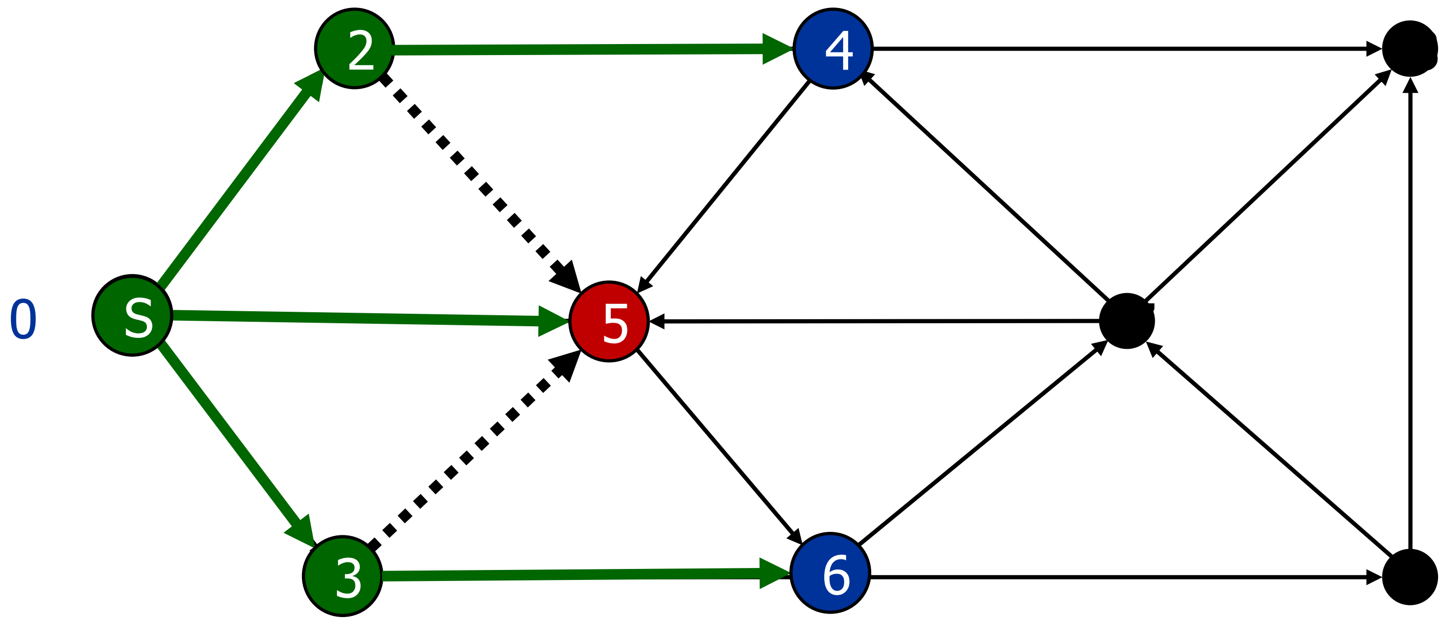
Undiscovered
Discovered
Top of queue
Finished

Queue (Q):

3 5 4

```
3: While Q not empty
4:     v = dequeue Q (i.e., 3)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example

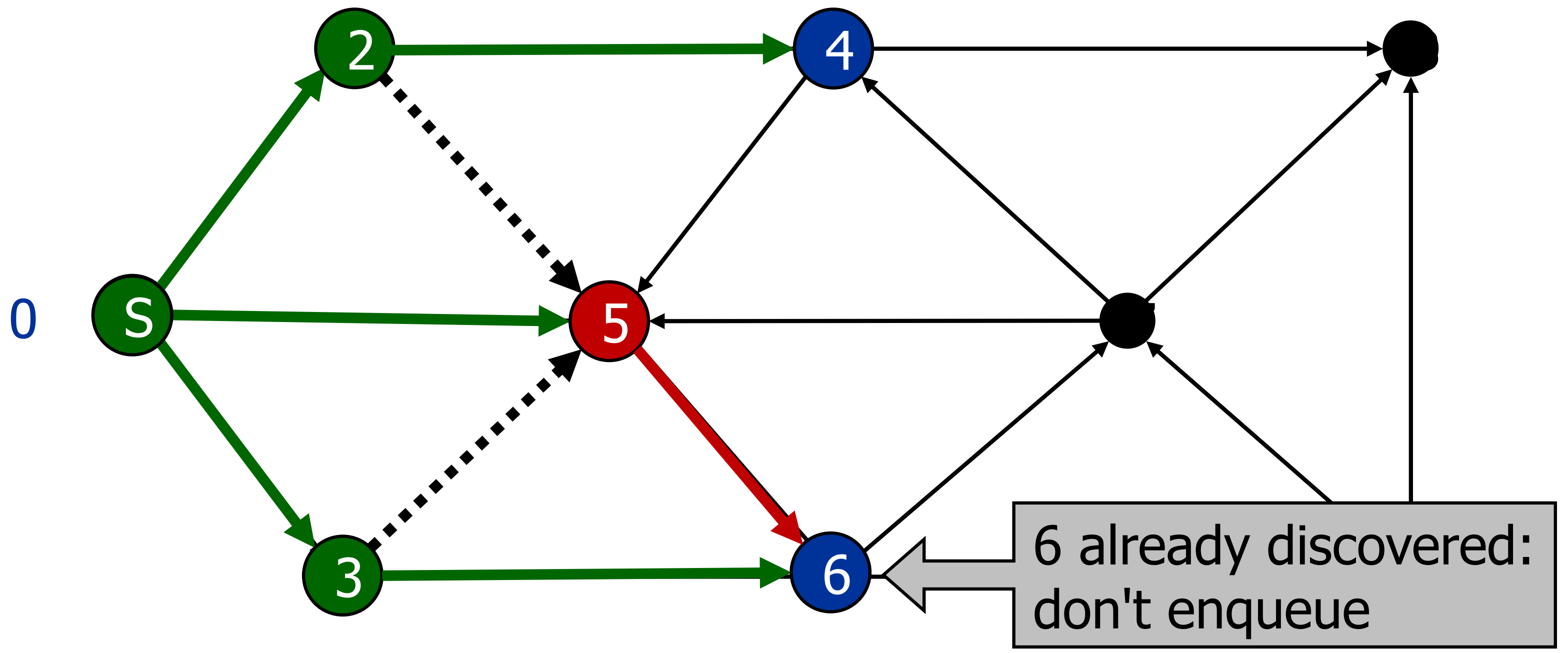


Queue (Q):

5 4 6

```
3: While Q not empty
4:     v = dequeue Q (i.e., 5)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



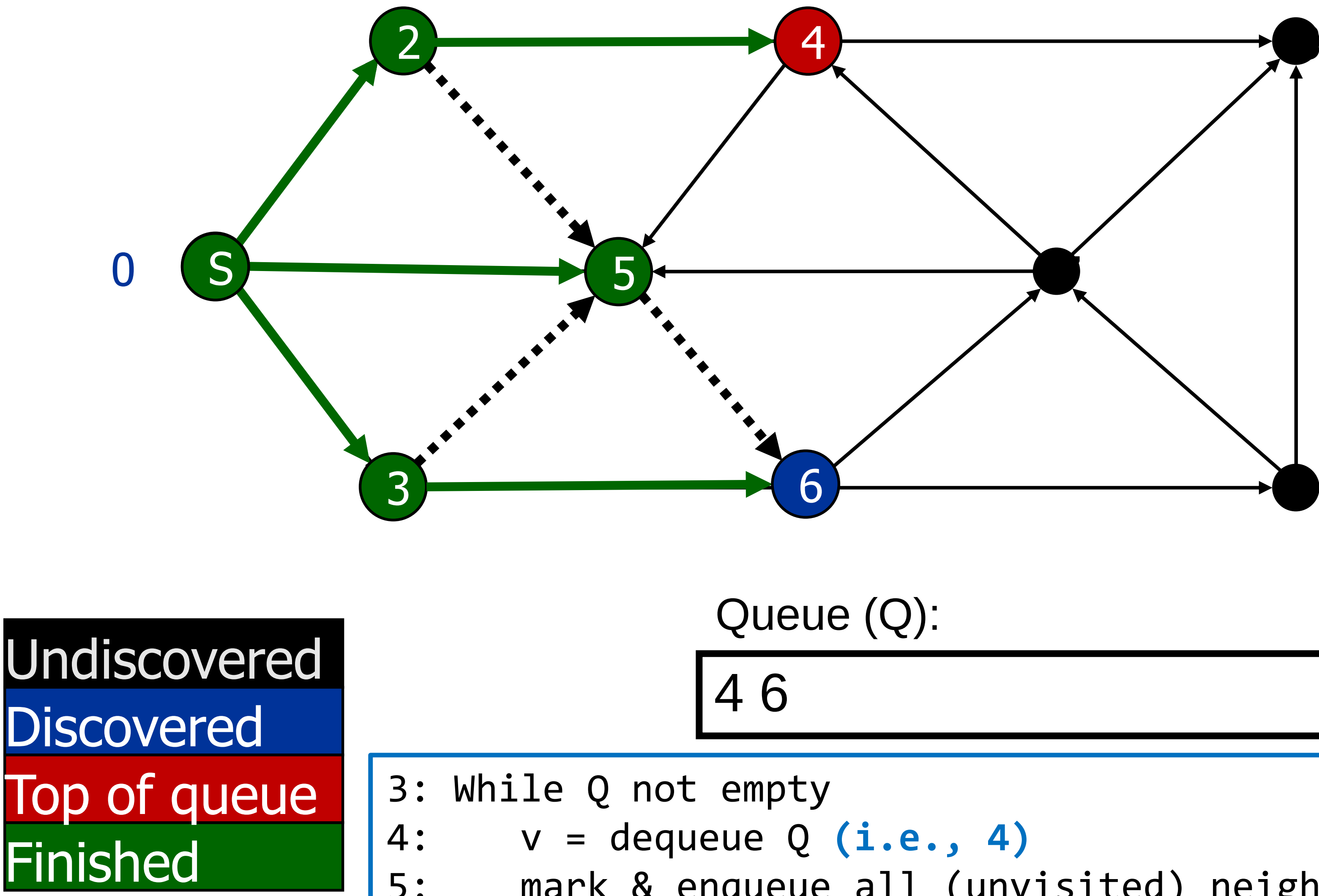
Undiscovered
Discovered
Top of queue
Finished

Queue (Q):

5 4 6

```
3: While Q not empty
4:     v = dequeue Q (i.e., 5)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example

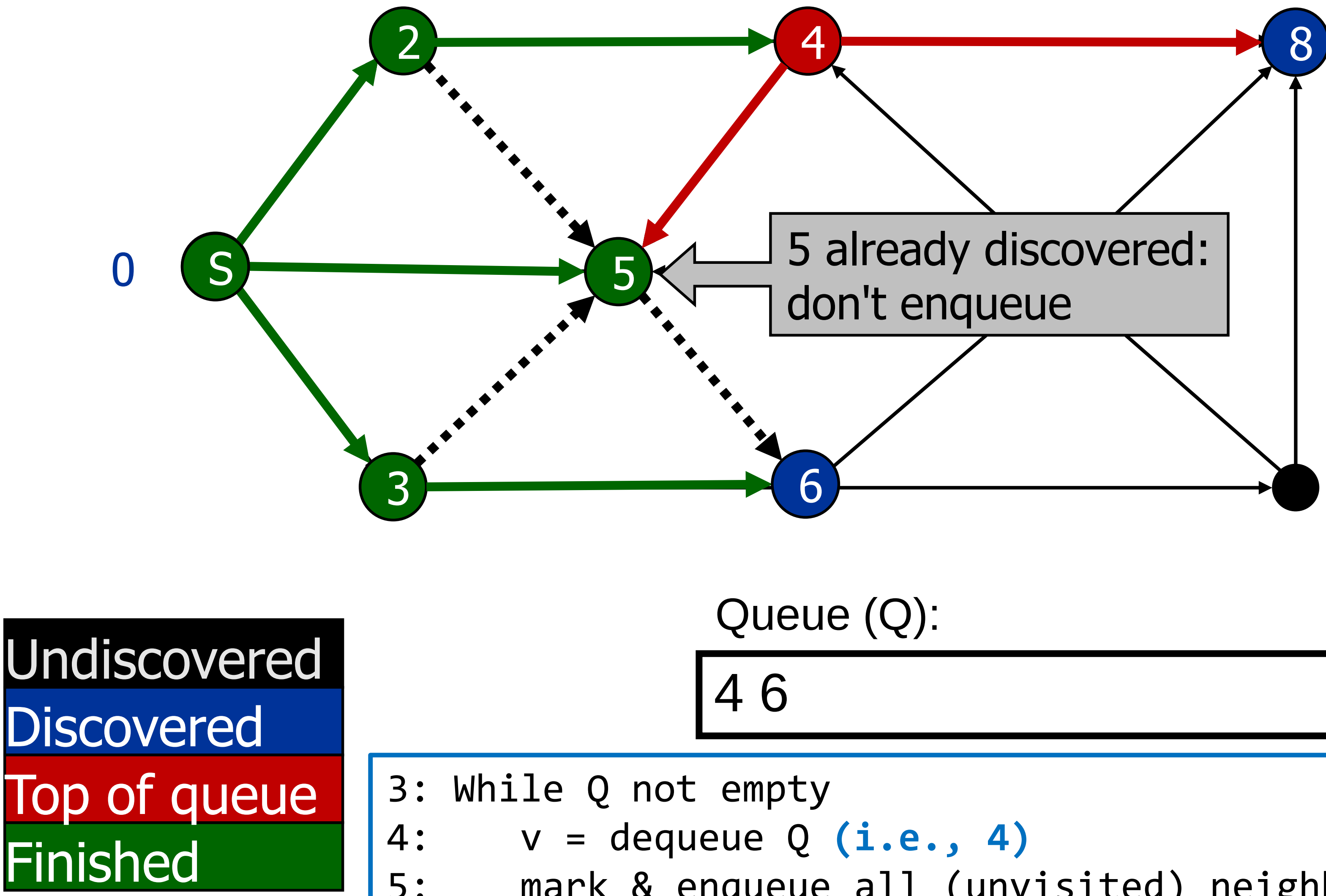


Queue (Q):

4 6

```
3: While Q not empty
4:     v = dequeue Q (i.e., 4)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example

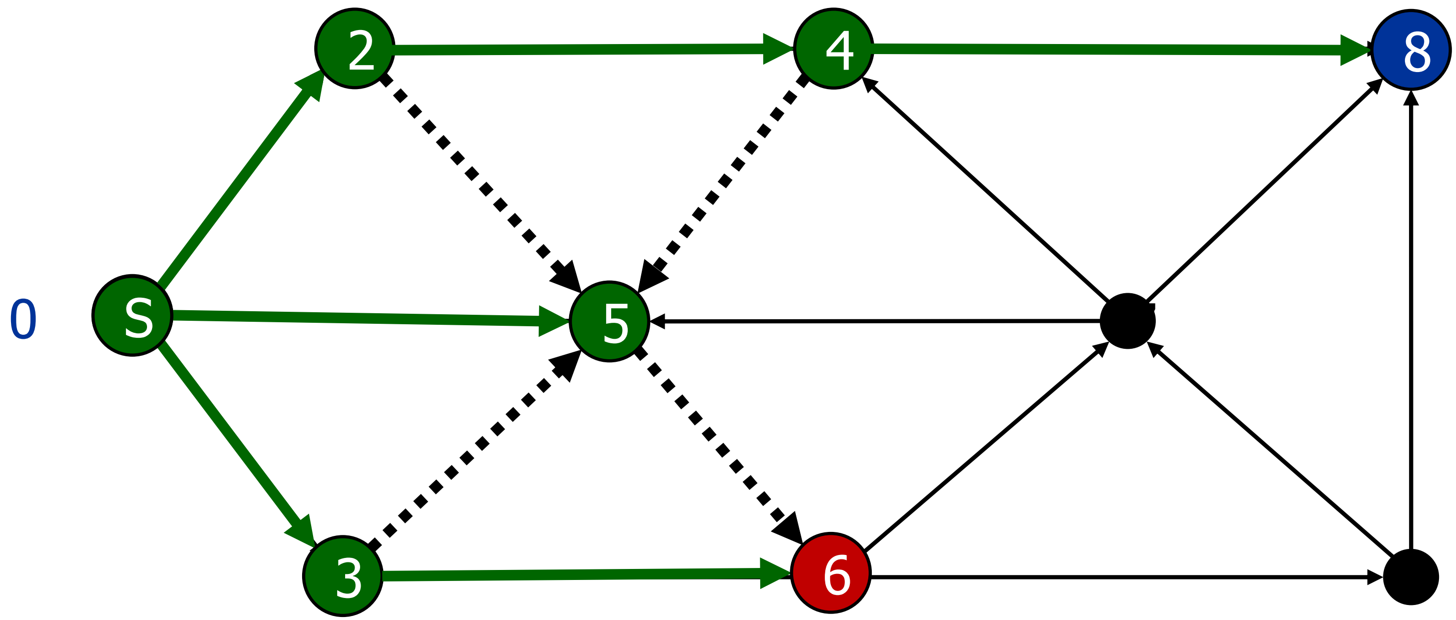


Queue (Q):

4 6

```
3: While Q not empty
4:     v = dequeue Q (i.e., 4)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example

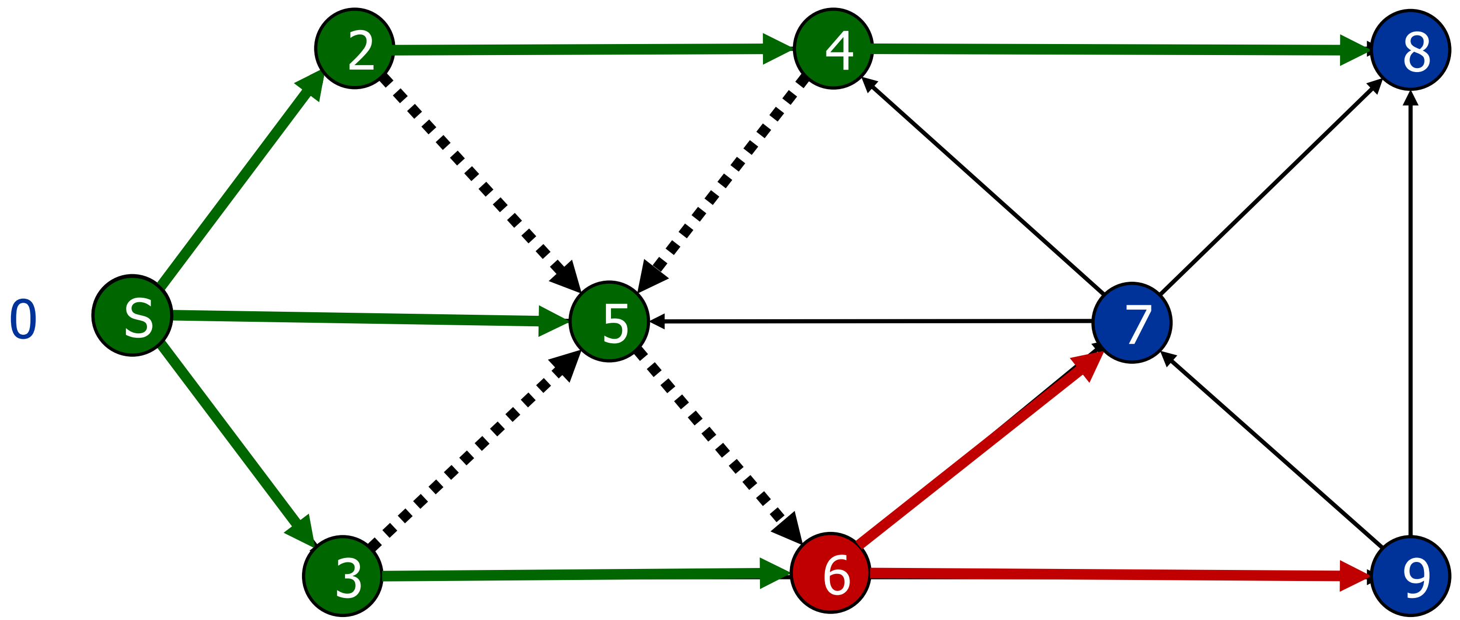


Queue (Q):

6 8

```
3: While Q not empty
4:     v = dequeue Q (i.e., 6)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



Undiscovered

Discovered

Top of queue

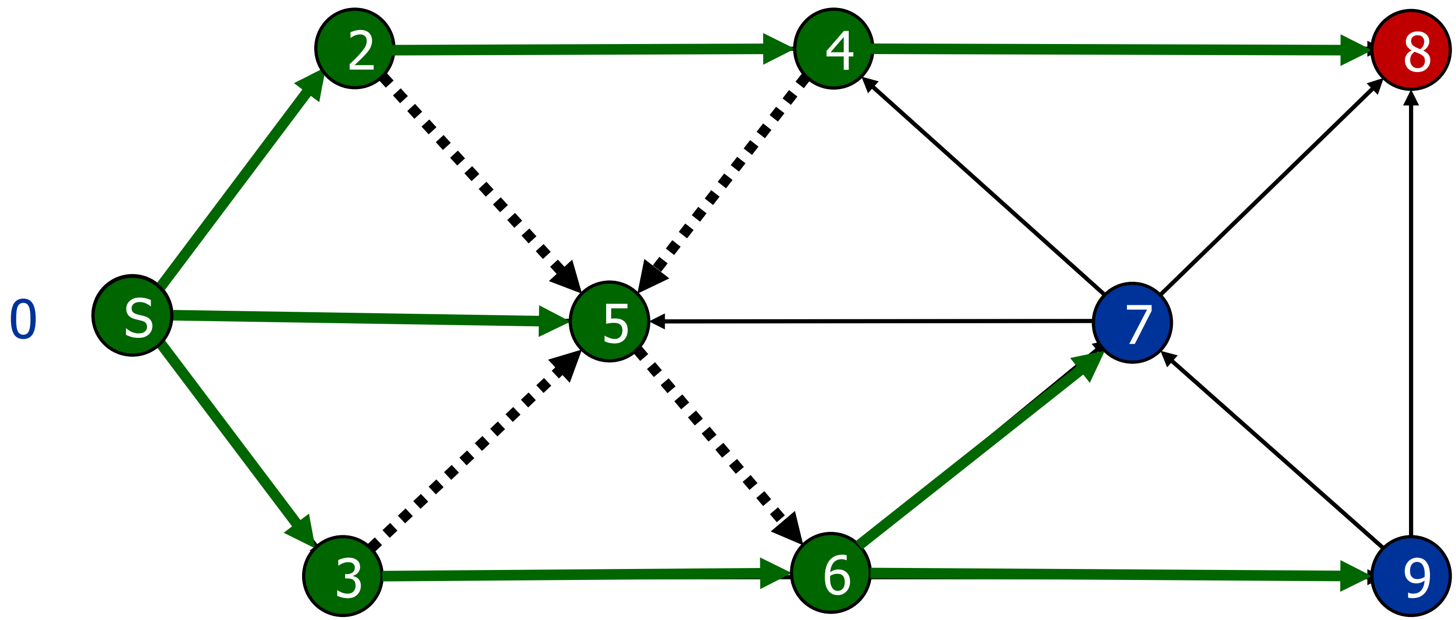
Finished

Queue (Q):

6 8

```
3: While Q not empty
4:     v = dequeue Q (i.e., 6)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



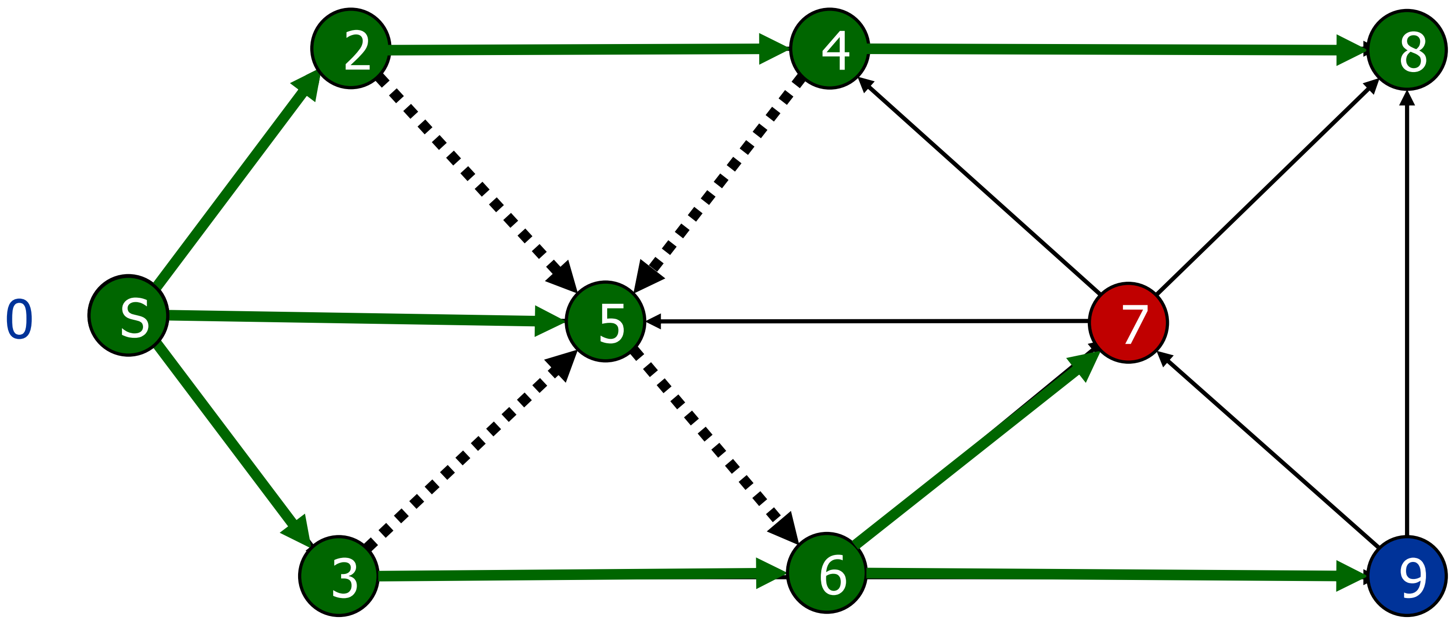
Undiscovered
Discovered
Top of queue
Finished

Queue (Q):

8 7 9

```
3: While Q not empty
4:     v = dequeue Q (i.e., 8)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



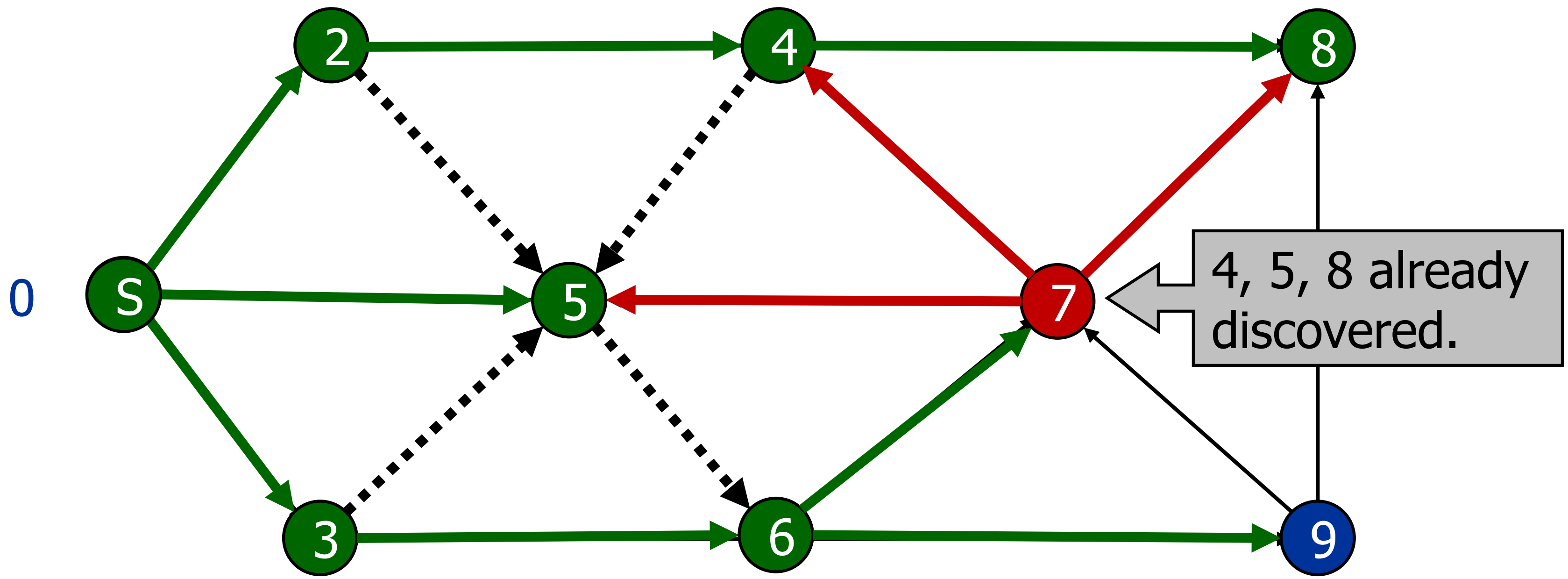
Undiscovered
Discovered
Top of queue
Finished

Queue (Q):

7 9

```
While Q not empty
    v = dequeue Q (i.e., 7)
    mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



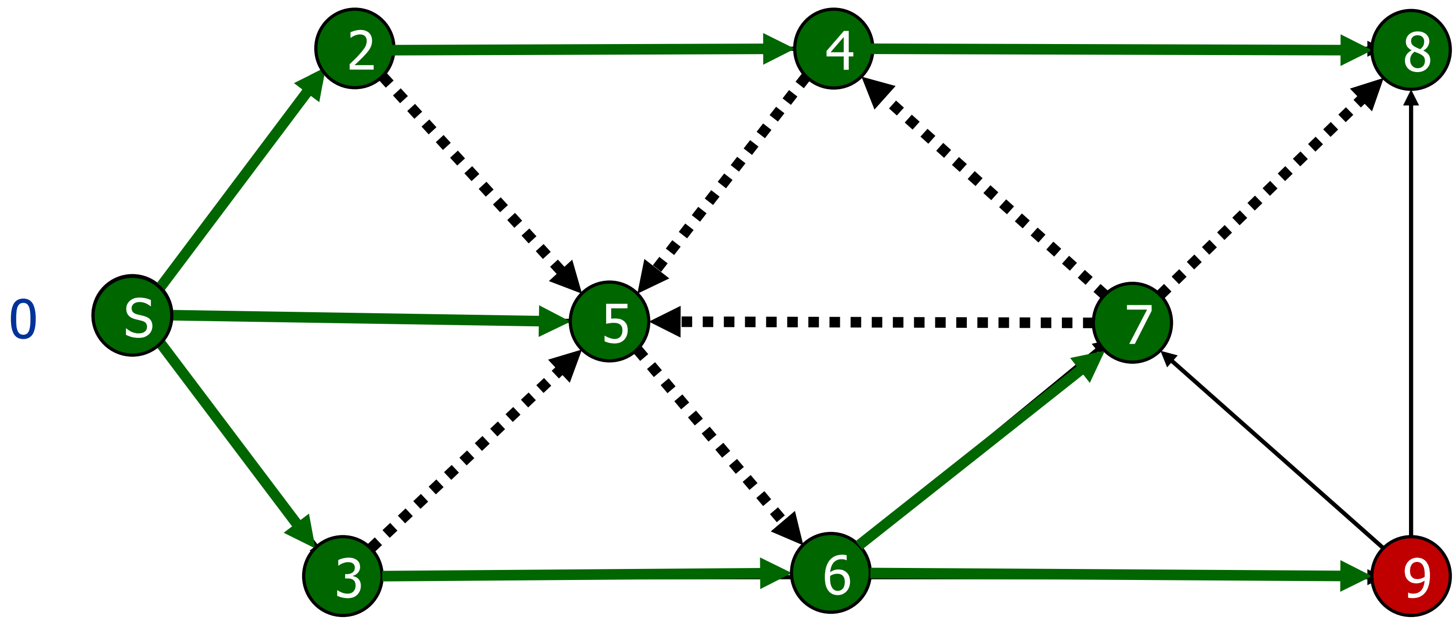
Undiscovered
Discovered
Top of queue
Finished

Queue (Q):

7 9

```
3: While Q not empty
4:     v = dequeue Q (i.e., 7)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



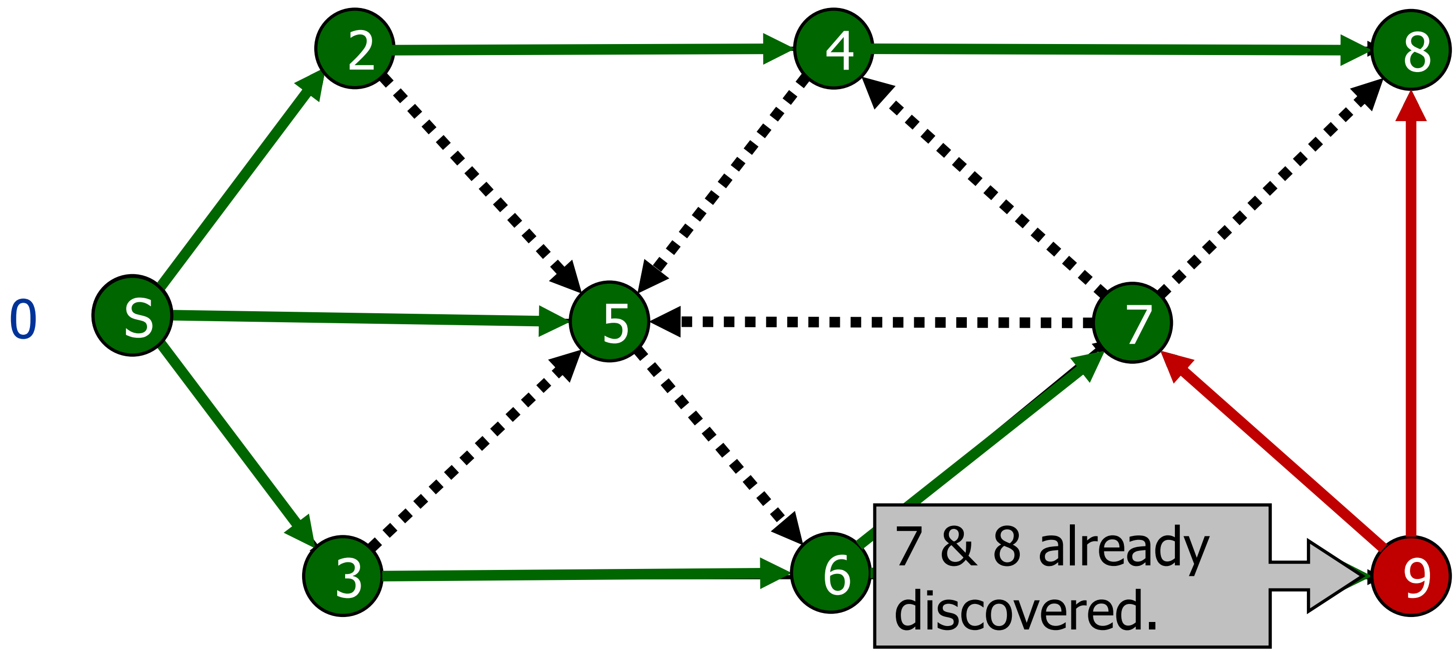
Undiscovered
Discovered
Top of queue
Finished

Queue (Q):

9

```
3: While Q not empty
4:     v = dequeue Q (i.e., 9)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



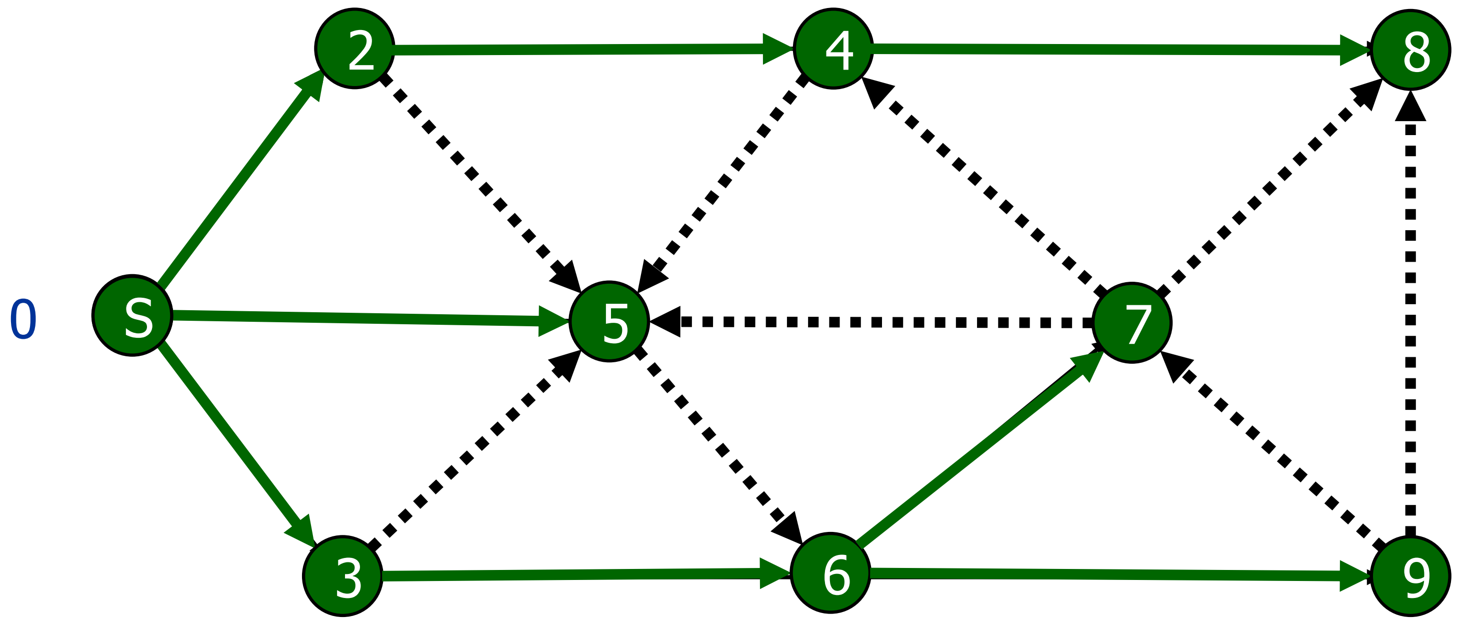
Undiscovered
Discovered
Top of queue
Finished

Queue (Q):

9

```
3: While Q not empty
4:     v = dequeue Q (i.e., 9)
5:     mark & enqueue all (unvisited) neighbors of v
```

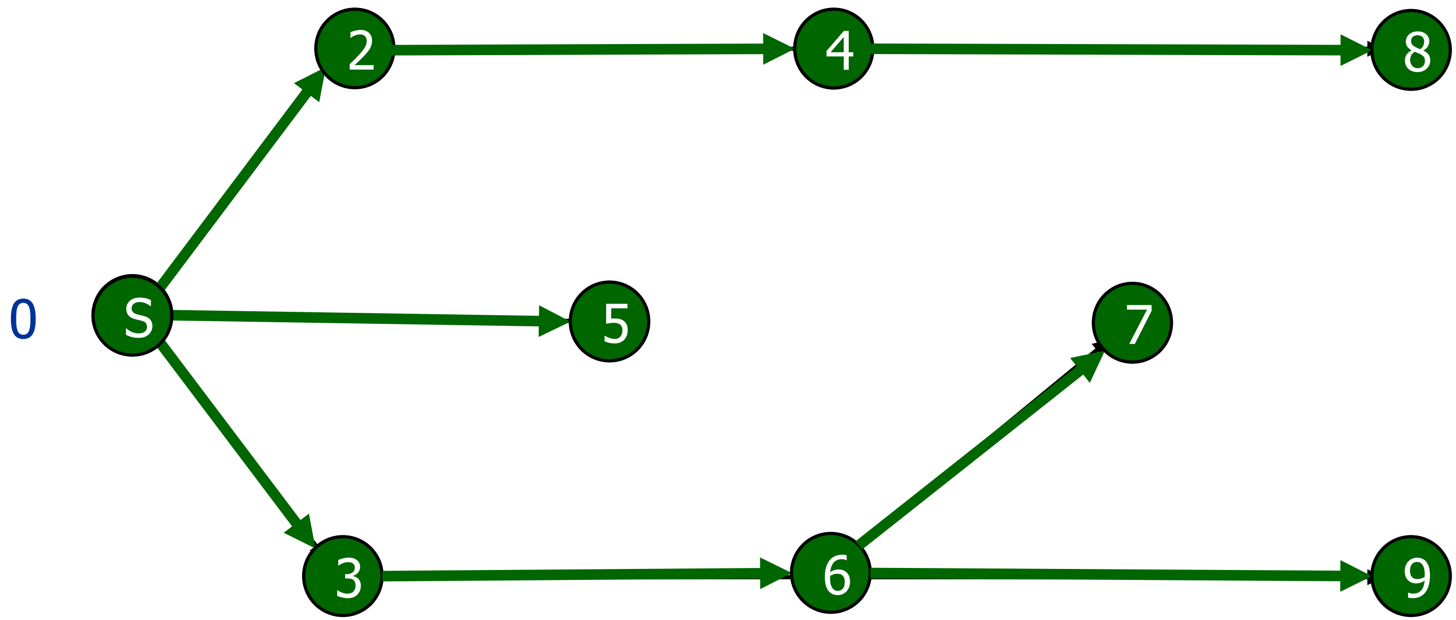
Breadth-First Search – Example



Queue (Q):


```
3: While Q not empty
4:     v = dequeue Q (i.e., NULL)
5:     mark & enqueue all (unvisited) neighbors of v
```

Breadth-First Search – Example



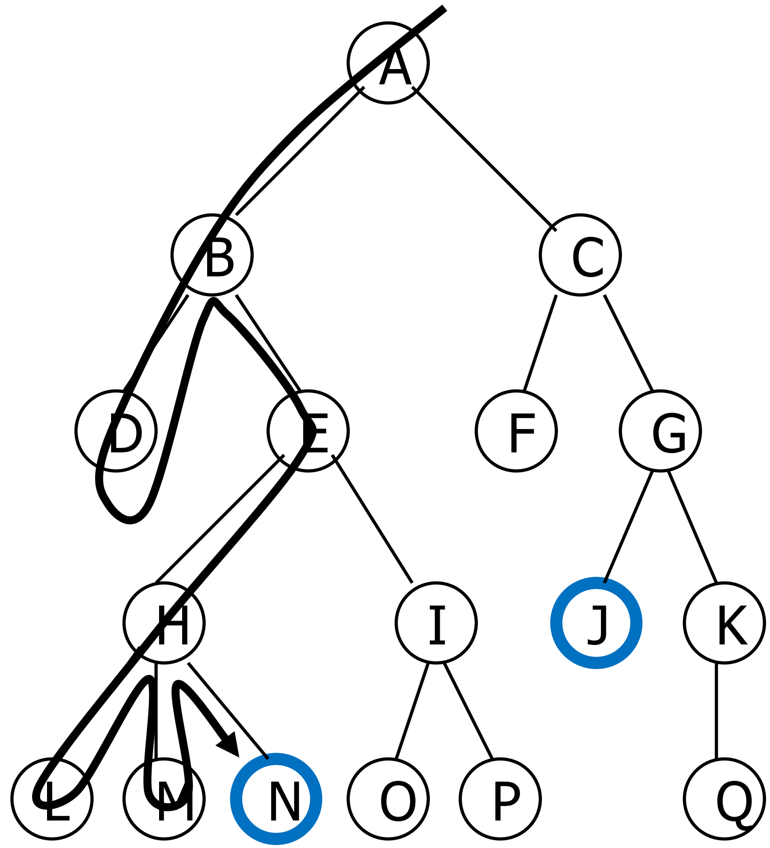
Breadth-First Search (BFS) tree rooted at S containing all nodes of the graph

Breadth-First Search – Properties

- Given a graph `G=(V,E)` and source vertex `s`, the following holds for the BFS algorithm
  - Systematically explores the edges of `G` to “discover” every vertex reachable from `s`
  - Creates a BFS tree rooted at `s` that contains all such vertices
  - Discovers all vertices at distance `k` from `s` before discovering any vertices at distance `k+1`

Depth-First Search – Trees

- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- N will be found before J
- Node are explored in the order A B D E H L M N I O P C F G J K Q

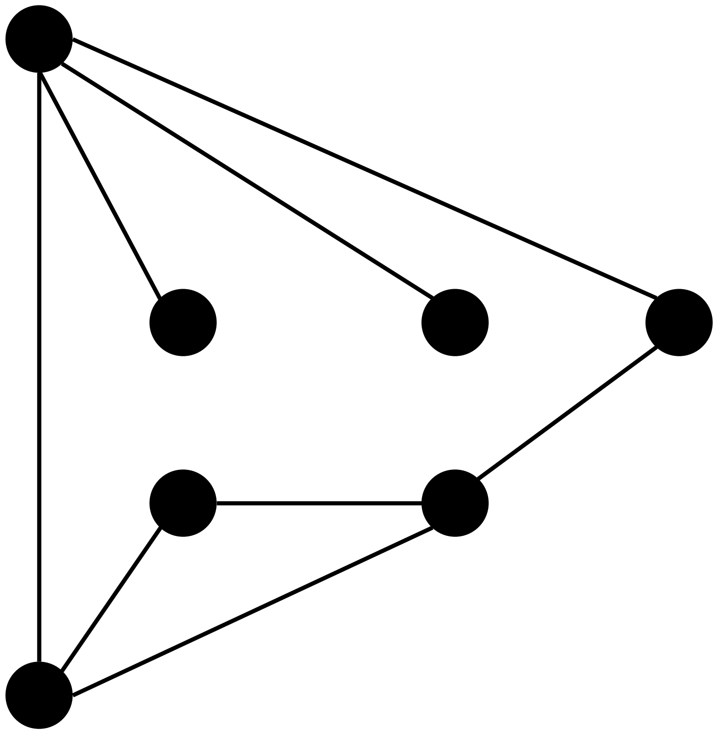


Depth-First Search

- Choose any vertex, mark it as visited
- From that vertex:
  - If there is another adjacent vertex not yet visited, go to it
  - Otherwise, go back to the most previous vertex that has not yet had all of its adjacent vertices visited and continue from there
- Continue until no visited vertices have unvisited adjacent vertices

```
Create a stack S
Mark v as visited and push v onto S
while S is non-empty
    peek at the top u of S
    if u has an (unvisited) neighbor w
        mark w and push it onto S
    else
        pop S
```

Depth-First Search – Example

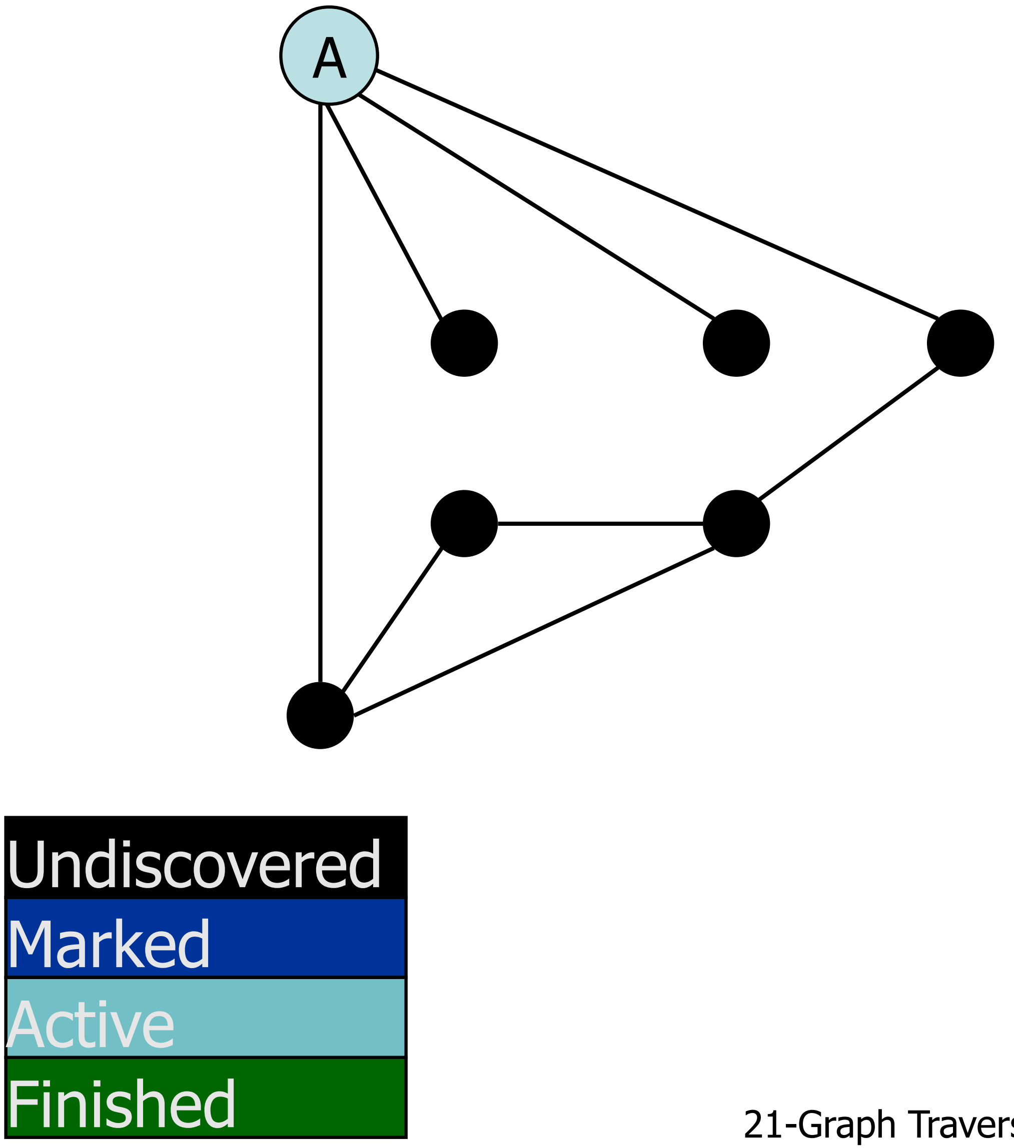


Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

Undiscovered
Marked
Active
Finished

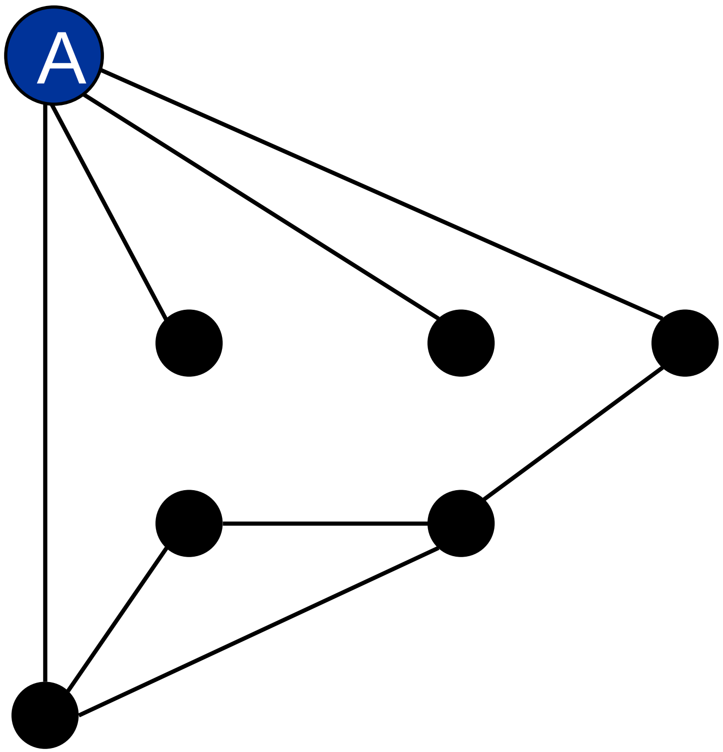
Depth-First Search – Example



Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

Depth-First Search – Example



Adjacency List

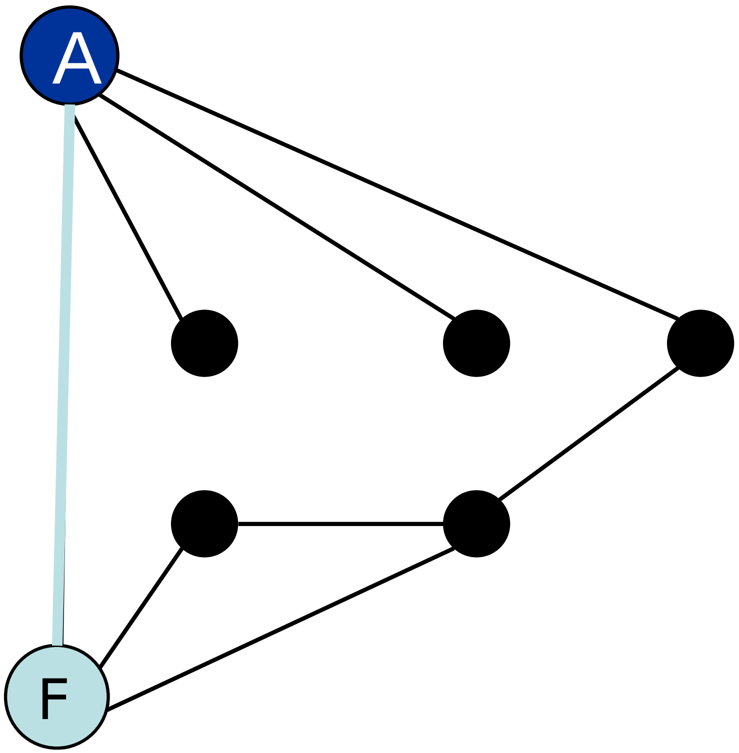
A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

Undiscovered
Marked
Active
Finished

Depth-First Search – Example

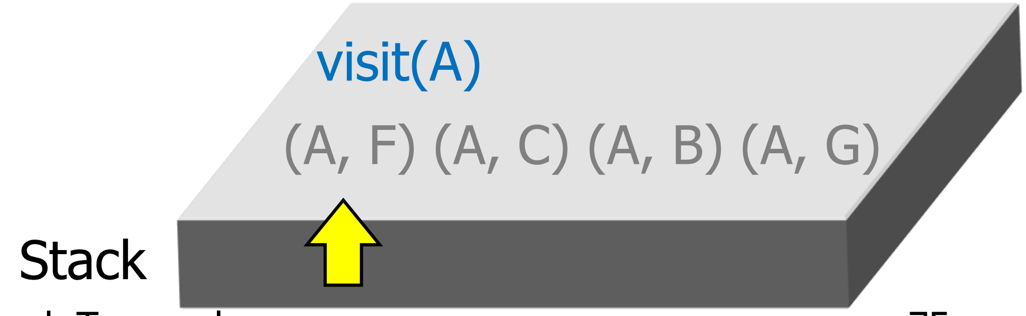
Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

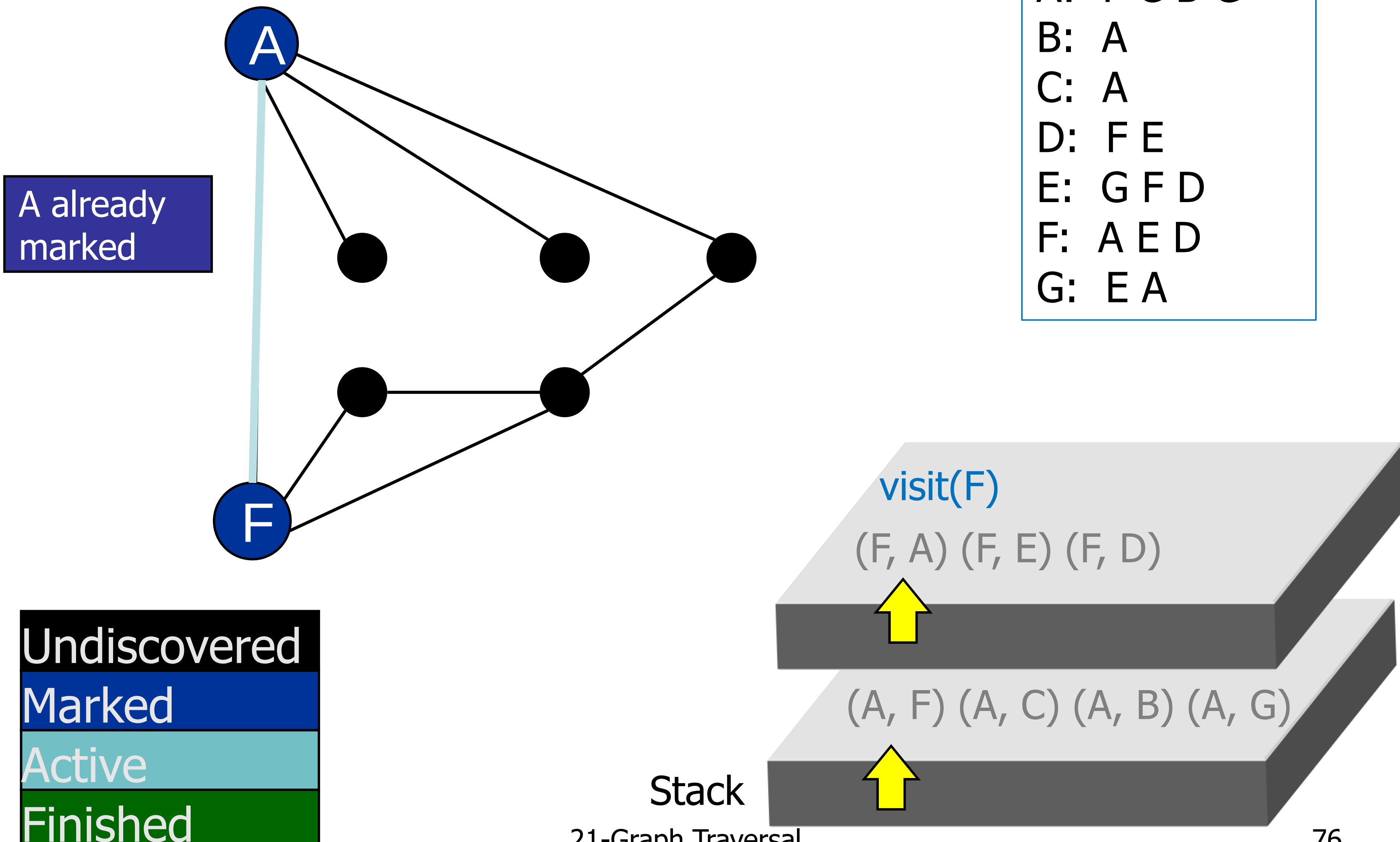


F newly discovered

Undiscovered
Marked
Active
Finished



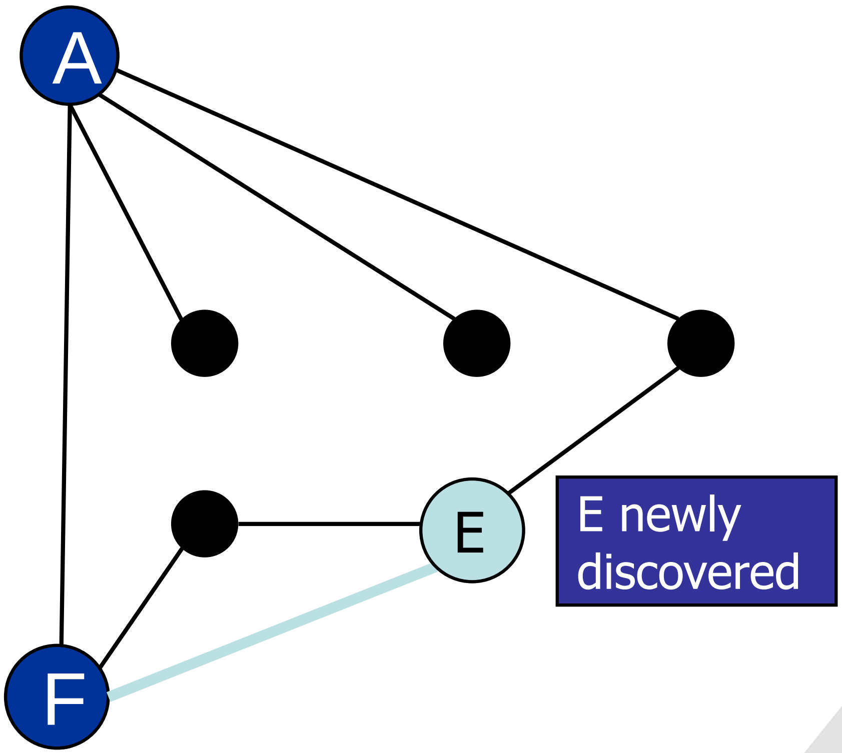
Depth-First Search – Example



Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

Depth-First Search – Example

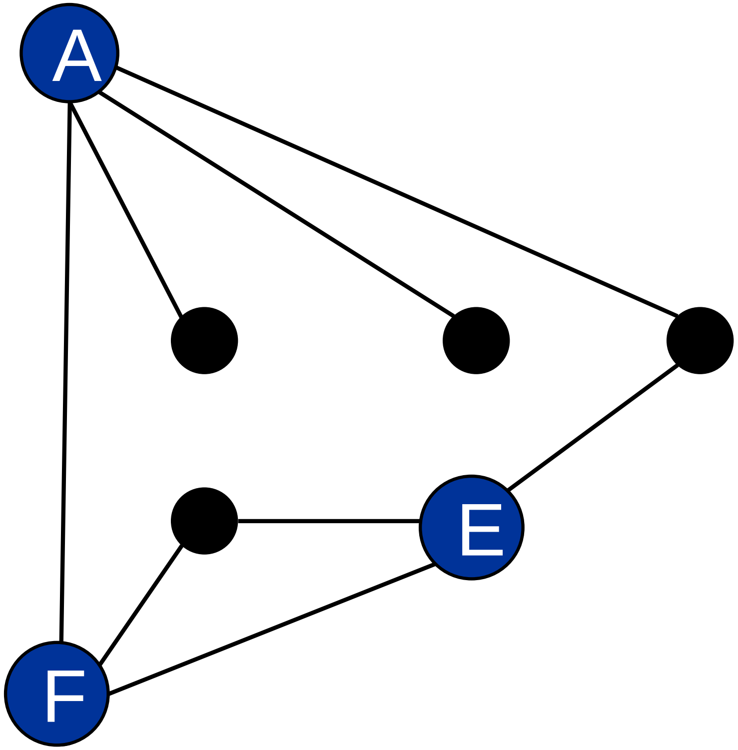


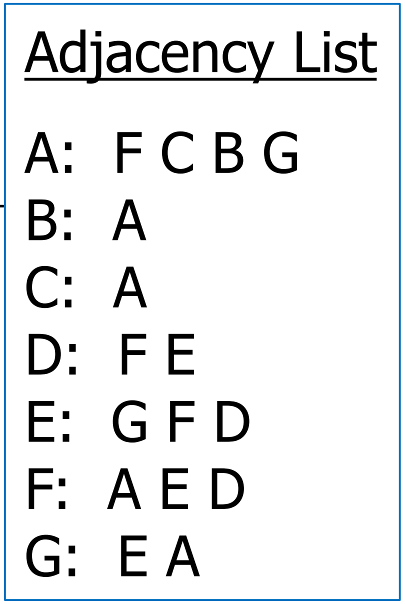
Adjacency List

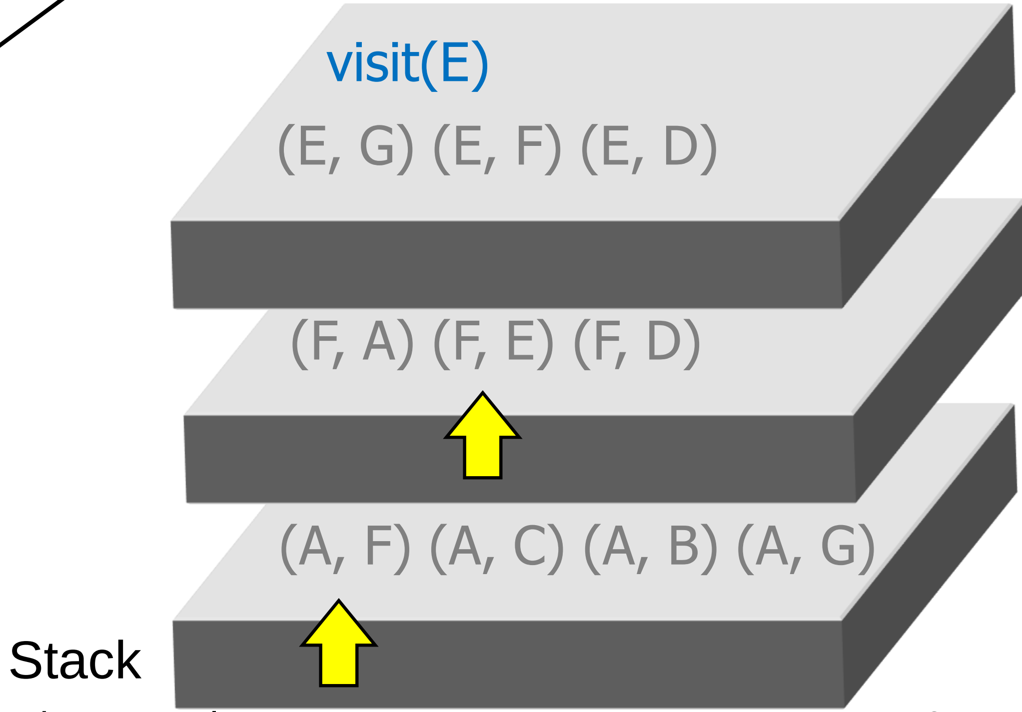
A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

Undiscovered
Marked
Active
Finished

Depth-First Search – Example



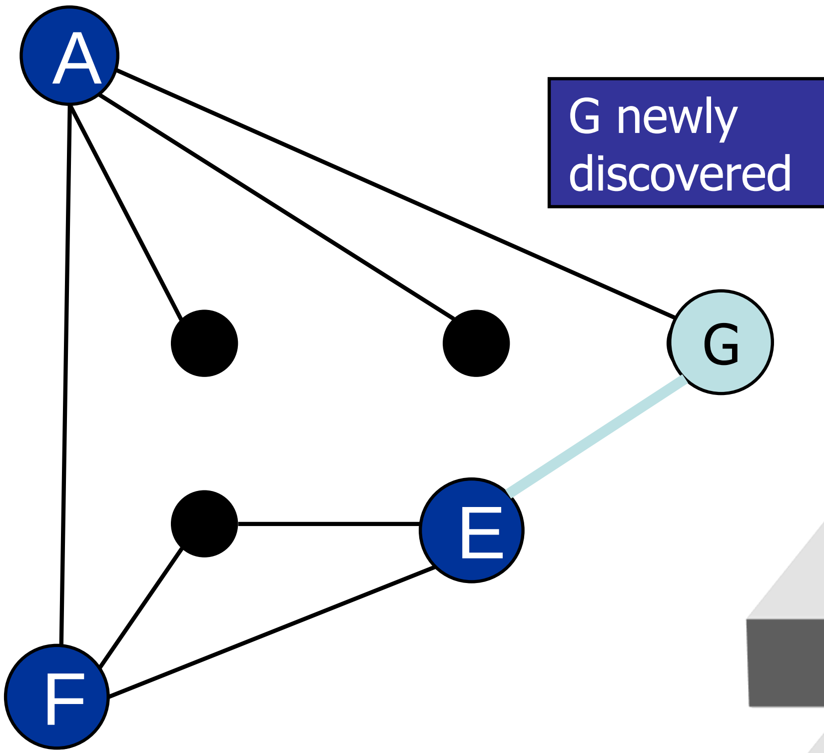


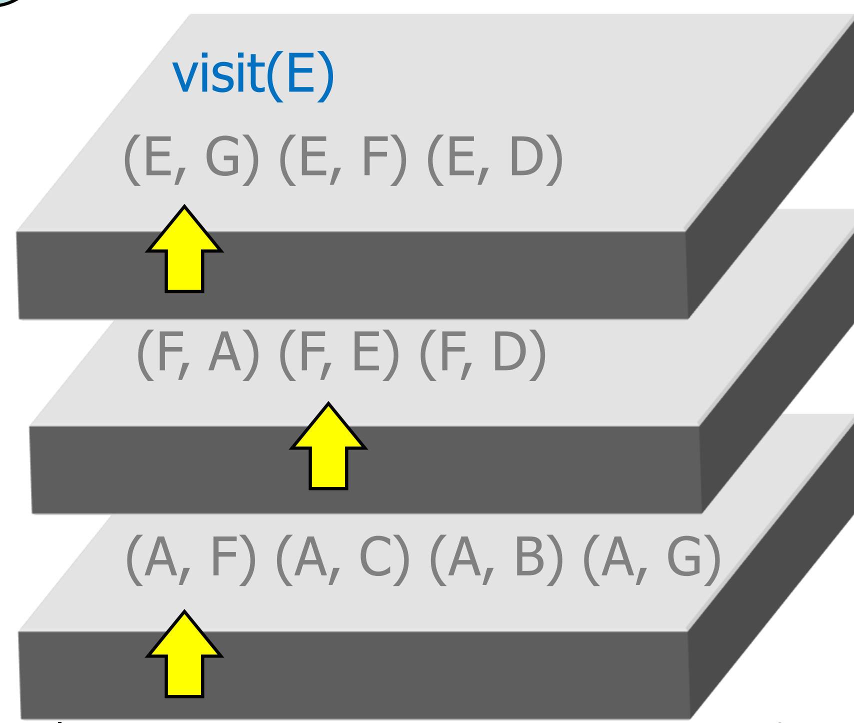


Depth-First Search – Example

<u>Adjacency List</u>

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A





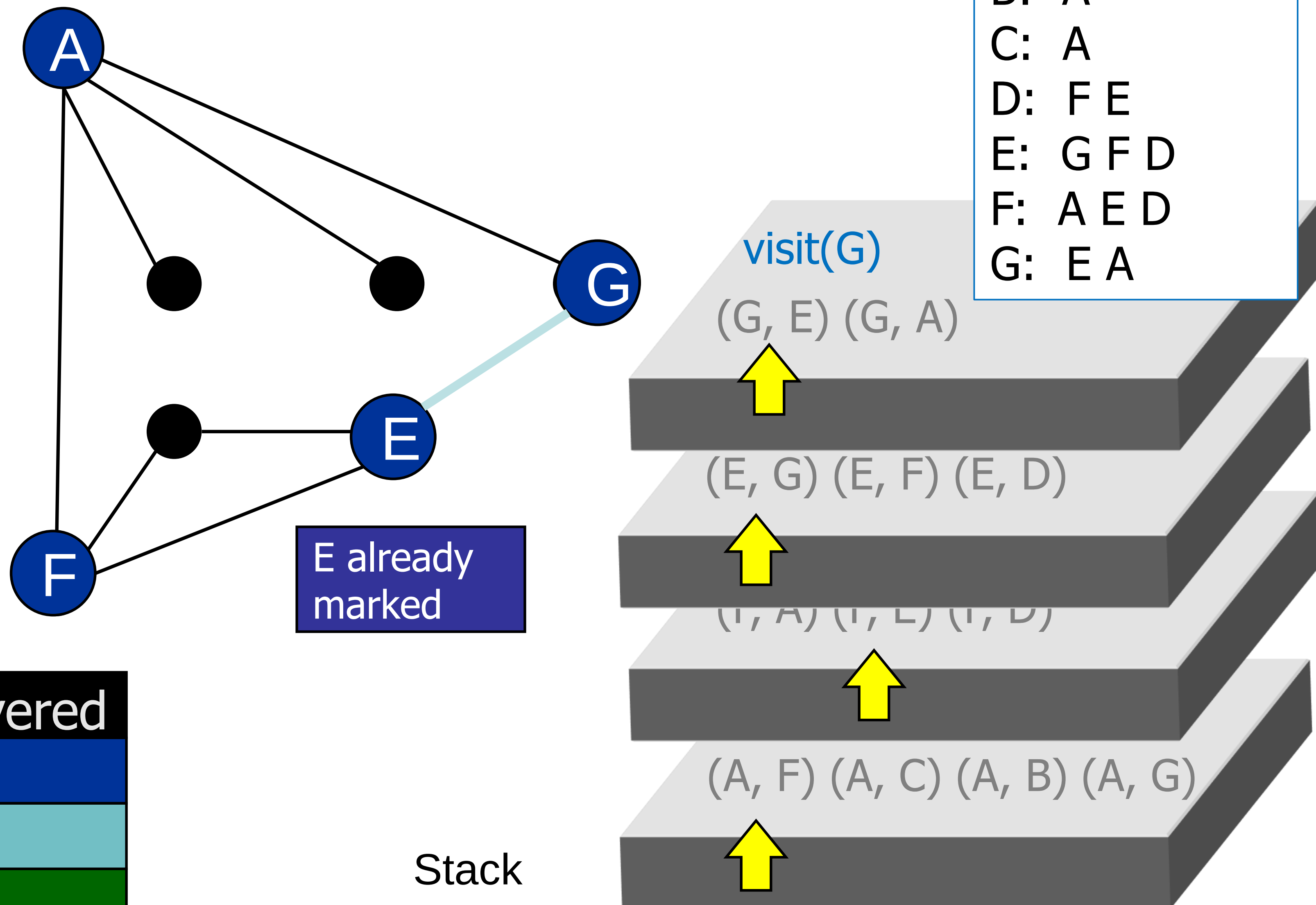
Stack

Undiscovered
Marked
Active
Finished

Depth-First Search – Example

Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

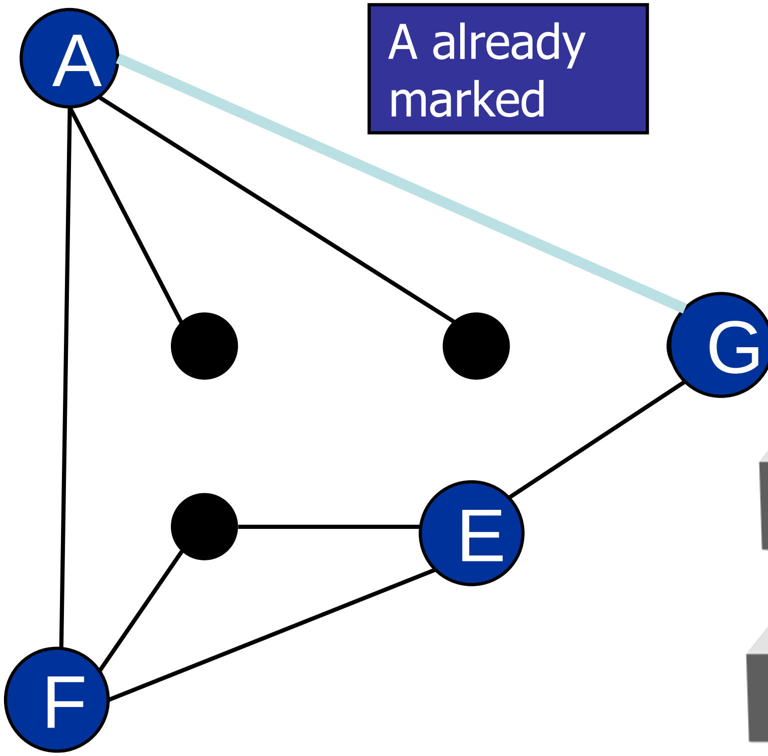


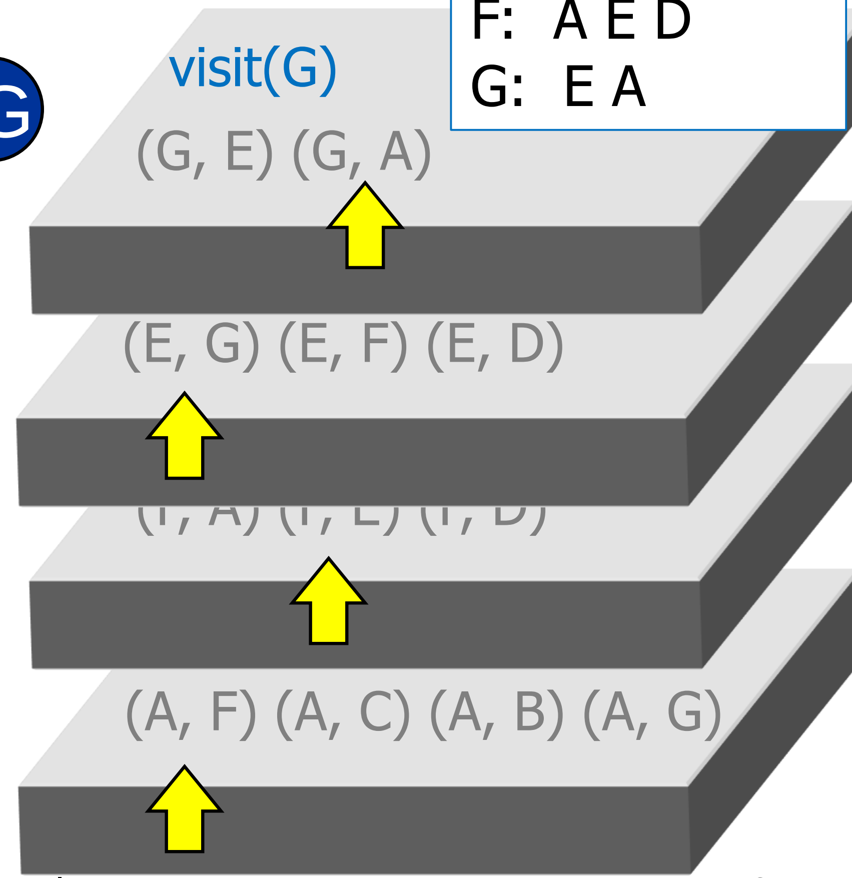
Undiscovered
Marked
Active
Finished

Depth-First Search – Example

Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A





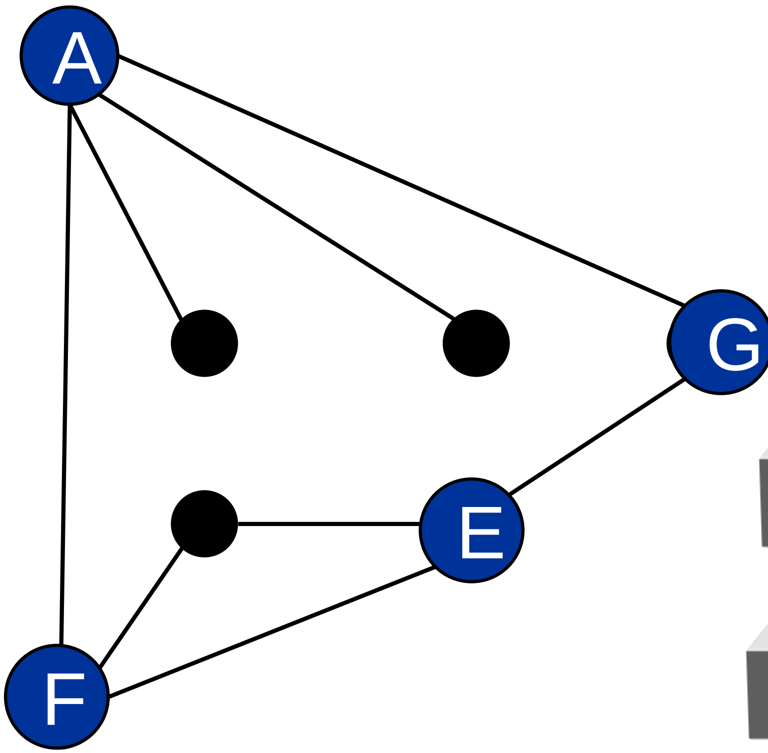
Stack

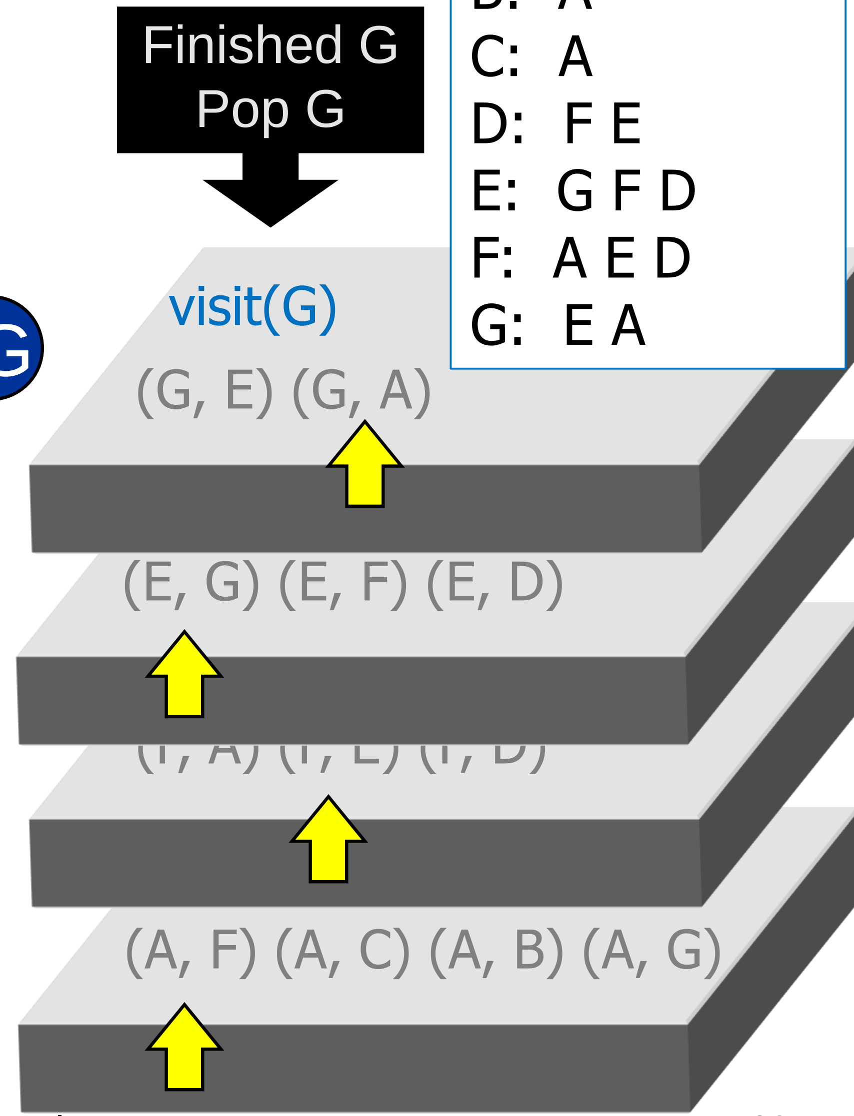
Undiscovered
Marked
Active
Finished

Depth-First Search – Example

Adjacency List

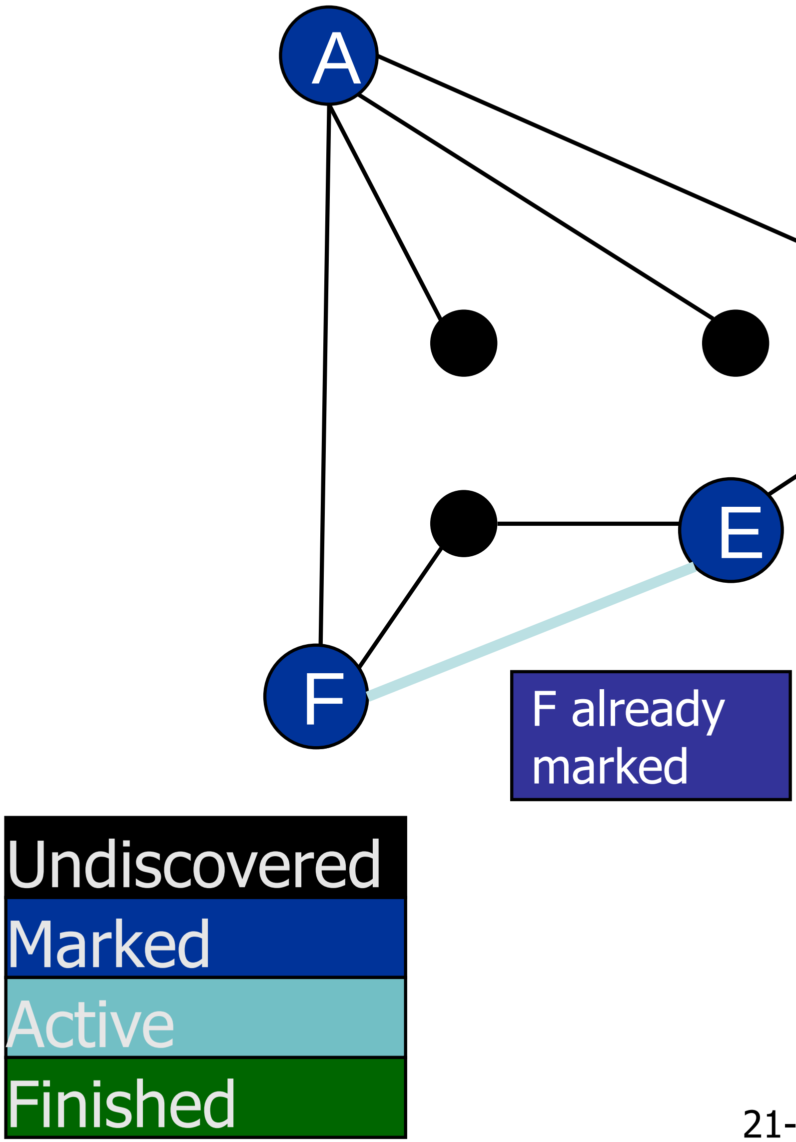
A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A





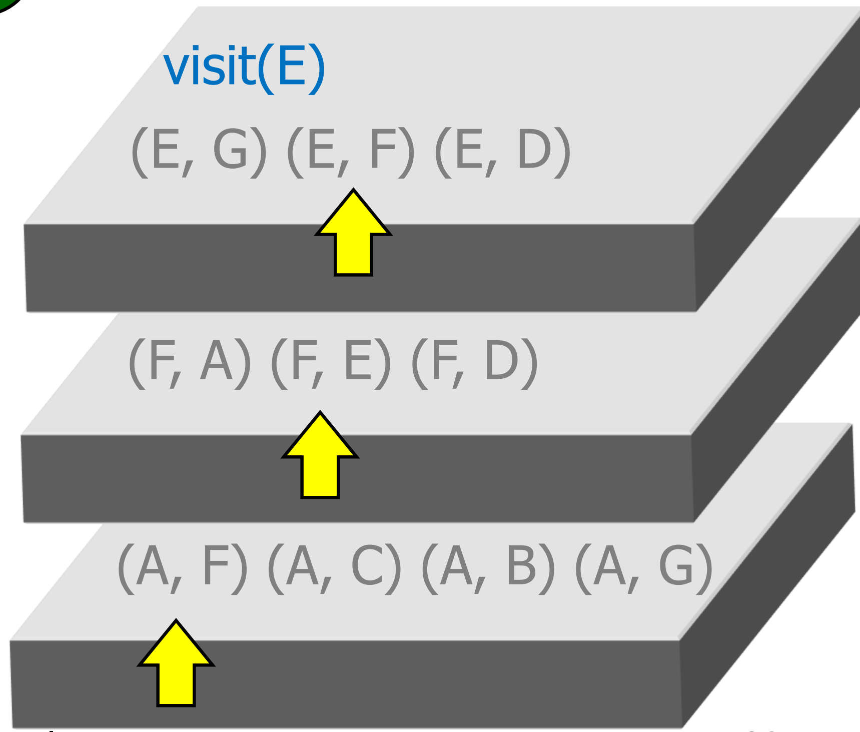
- Undiscovered
- Marked
- Active
- Finished

Depth-First Search – Example

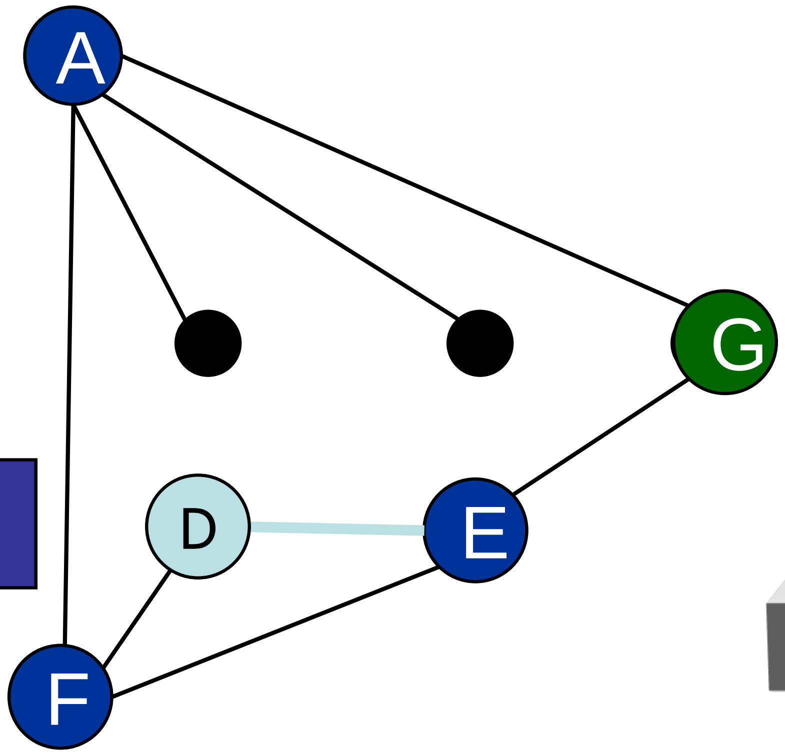


Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A



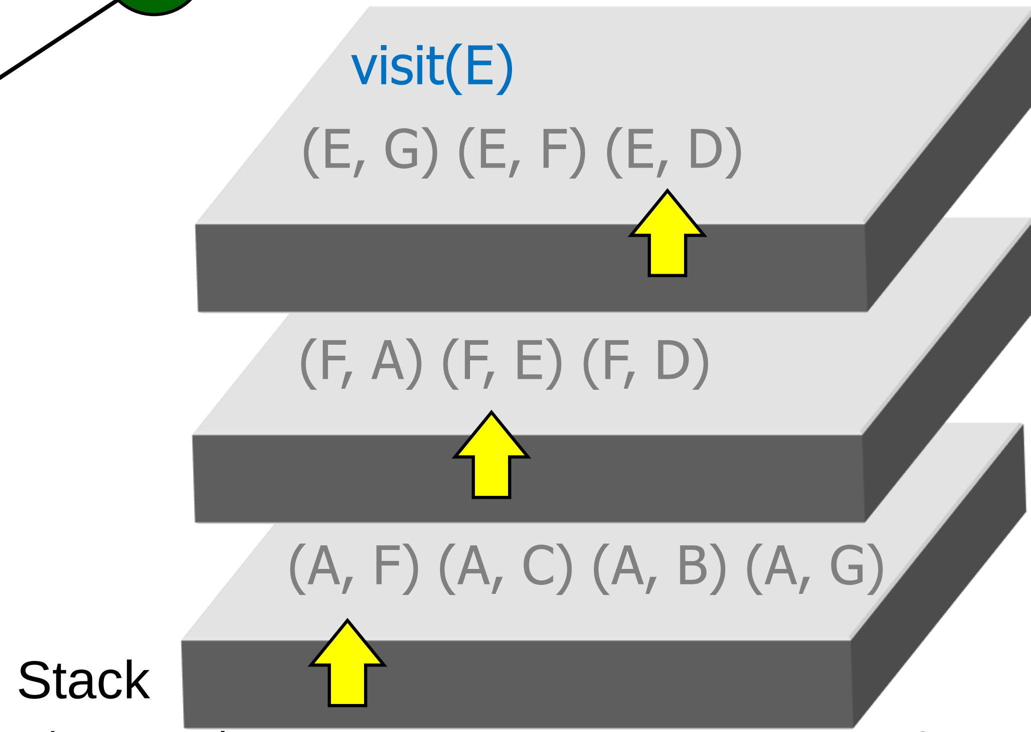
Depth-First Search – Example



D newly discovered

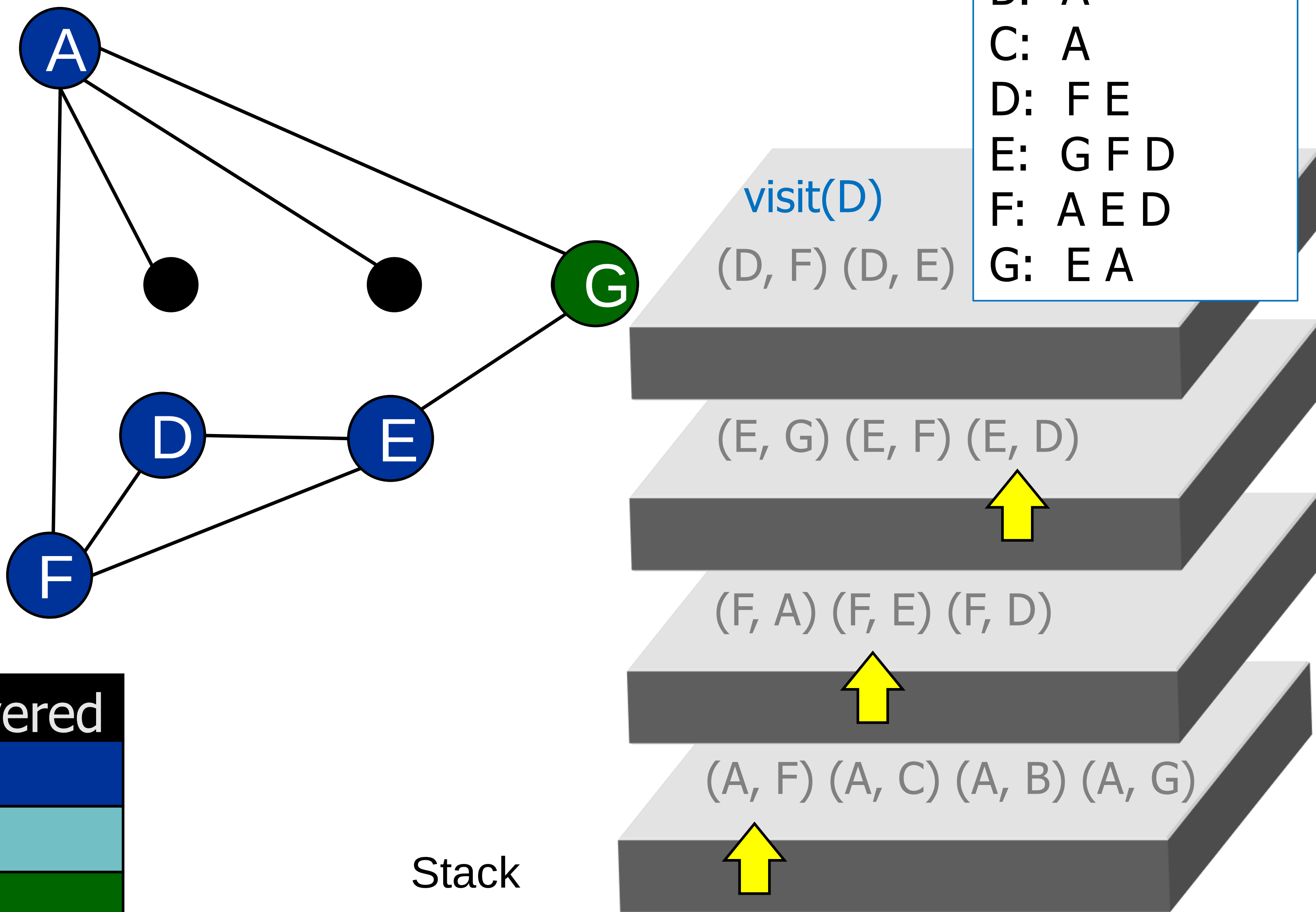
Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A



Undiscovered
Marked
Active
Finished

Depth-First Search – Example

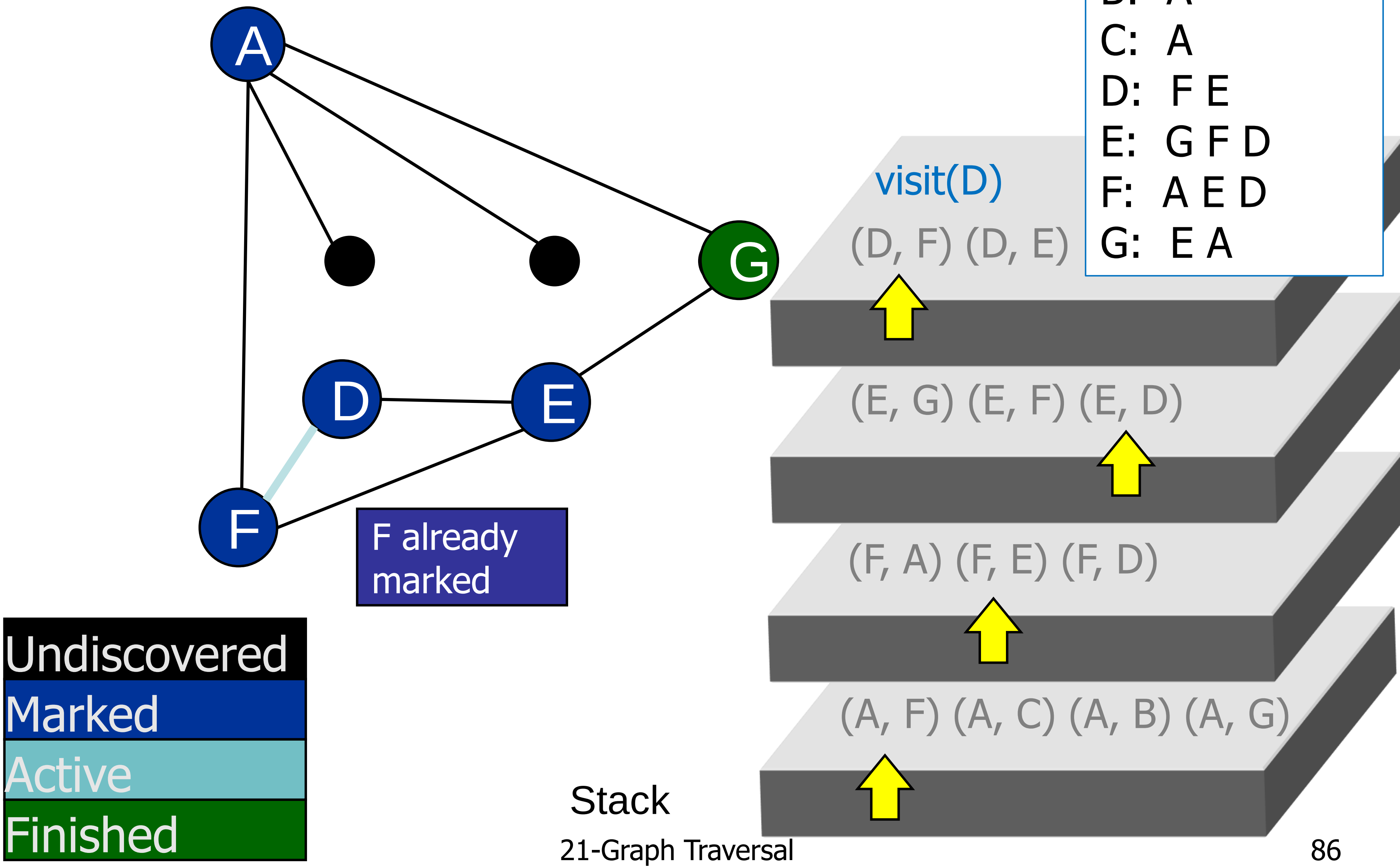


Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

Undiscovered
Marked
Active
Finished

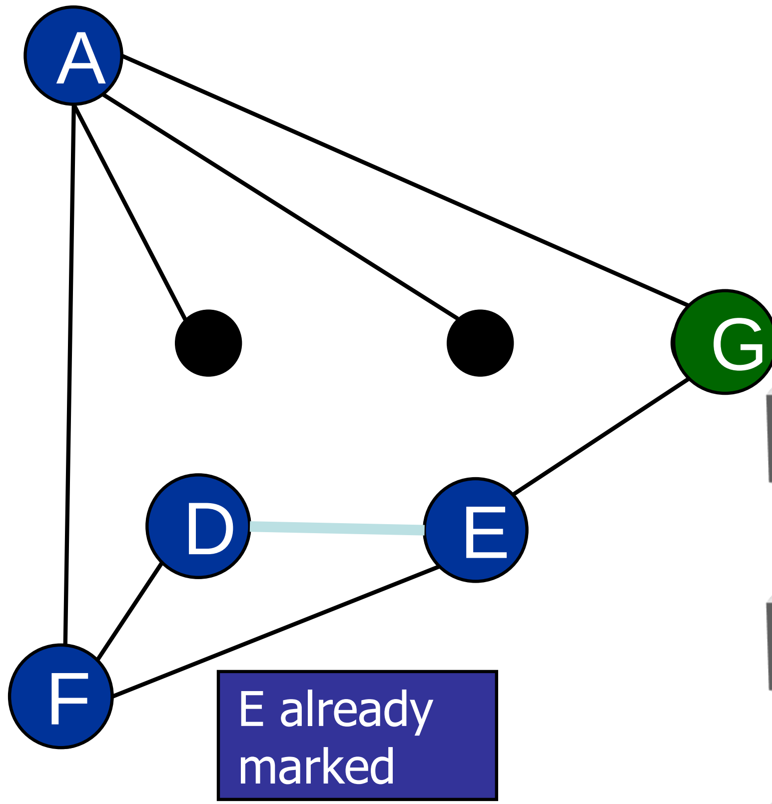
Depth-First Search – Example

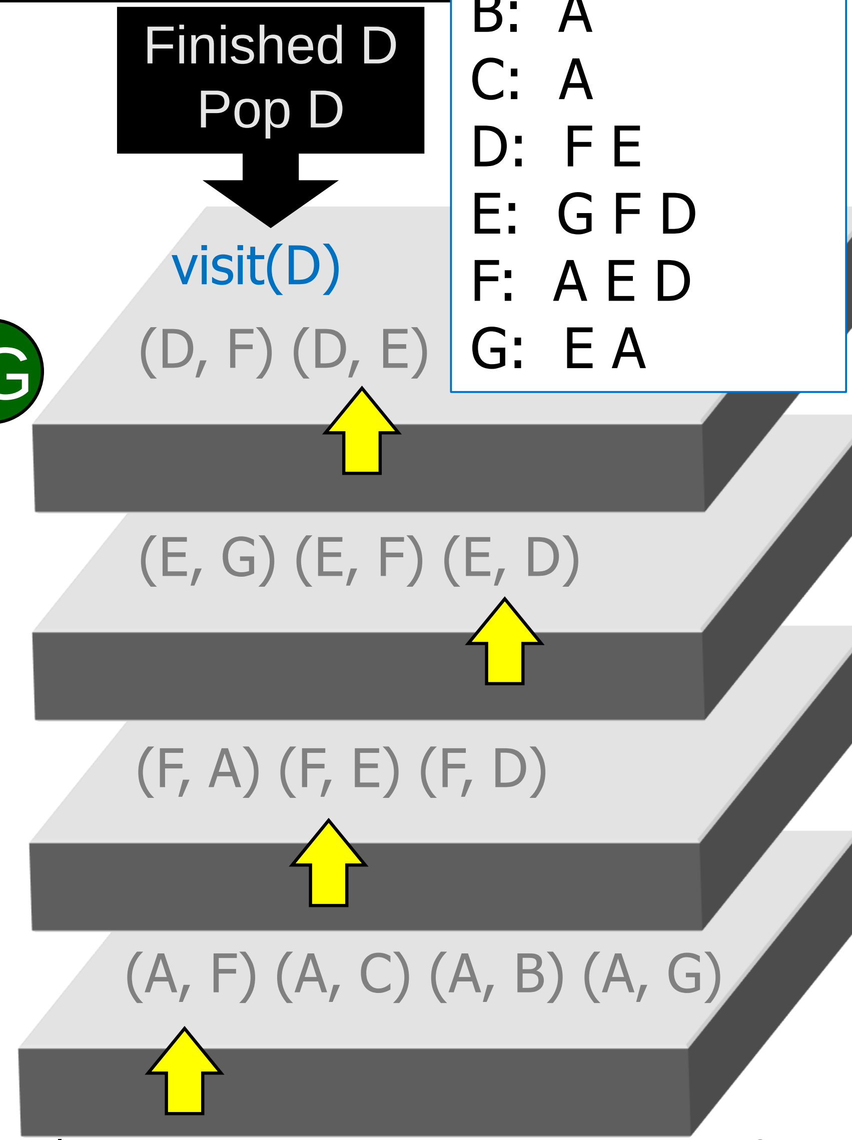


Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

Depth-First Search – Example





Stack

Adjacency List

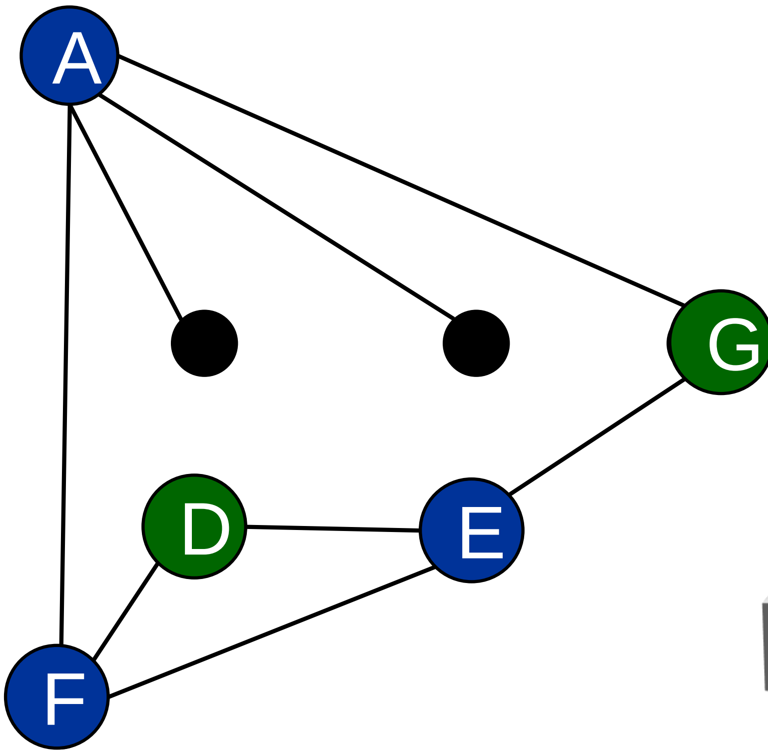
A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

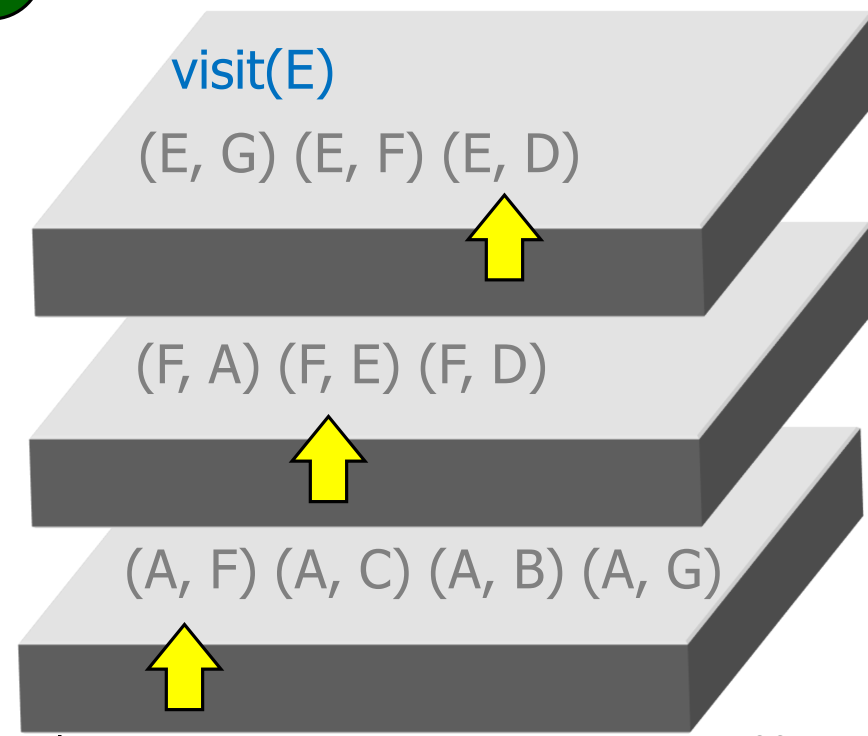
Undiscovered
Marked
Active
Finished

Depth-First Search – Example

<u>Adjacency List</u>

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A





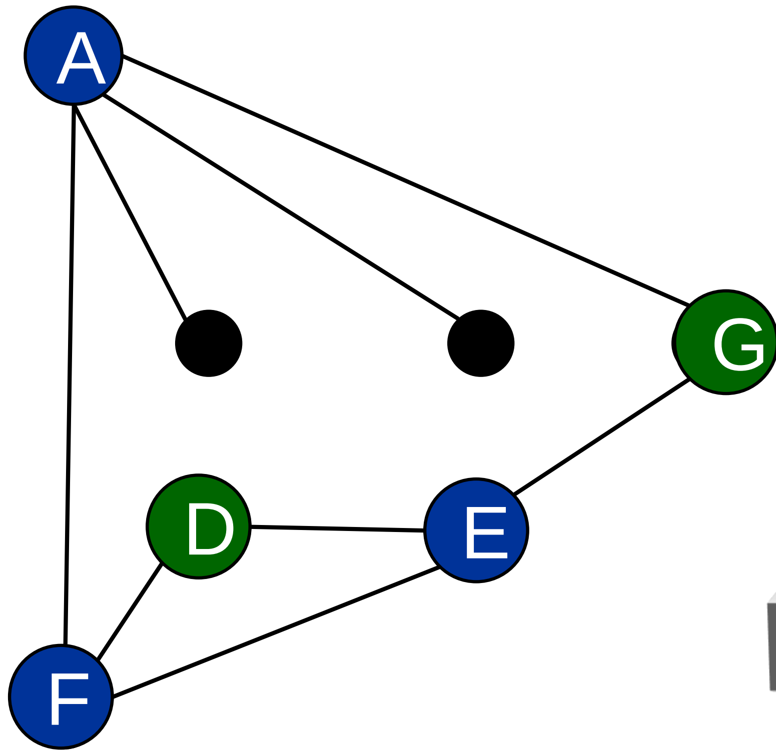
Stack

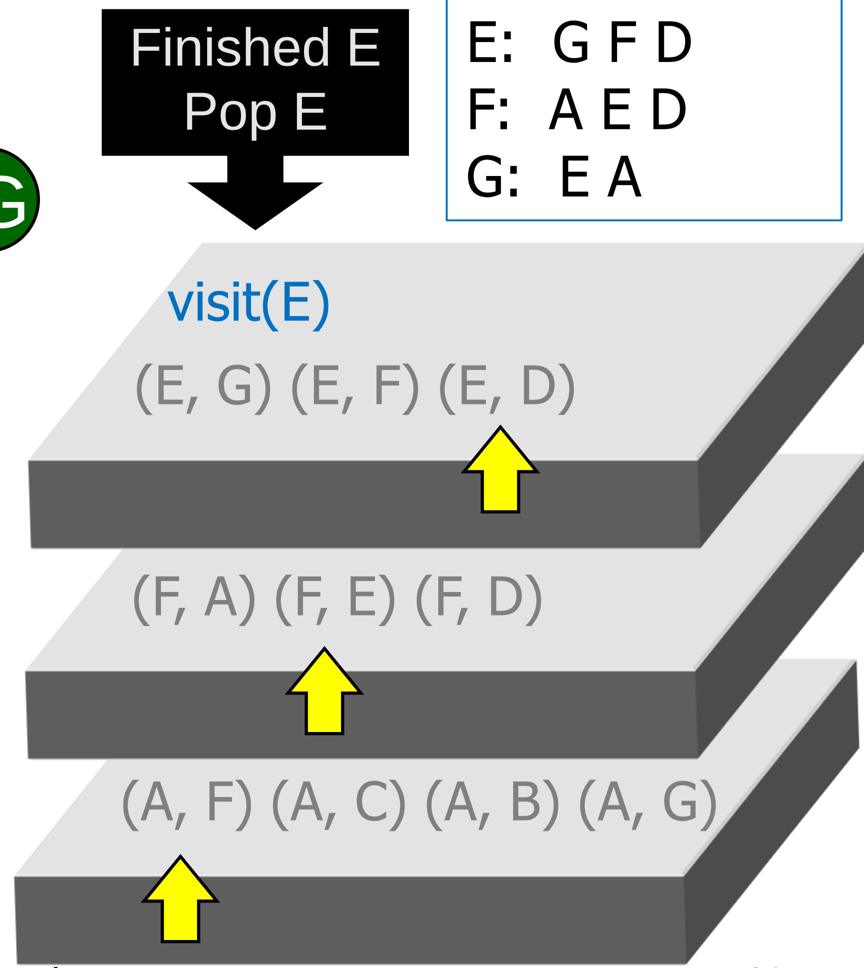
Undiscovered
Marked
Active
Finished

Depth-First Search – Example

Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A





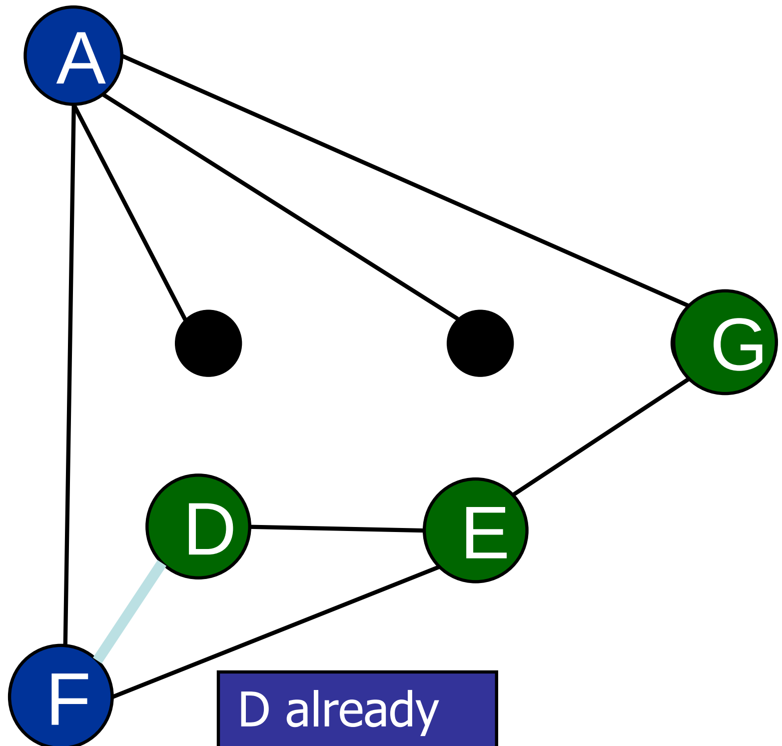
Stack

Undiscovered
Marked
Active
Finished

Depth-First Search – Example

Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A



D already marked

Finished F
Pop F

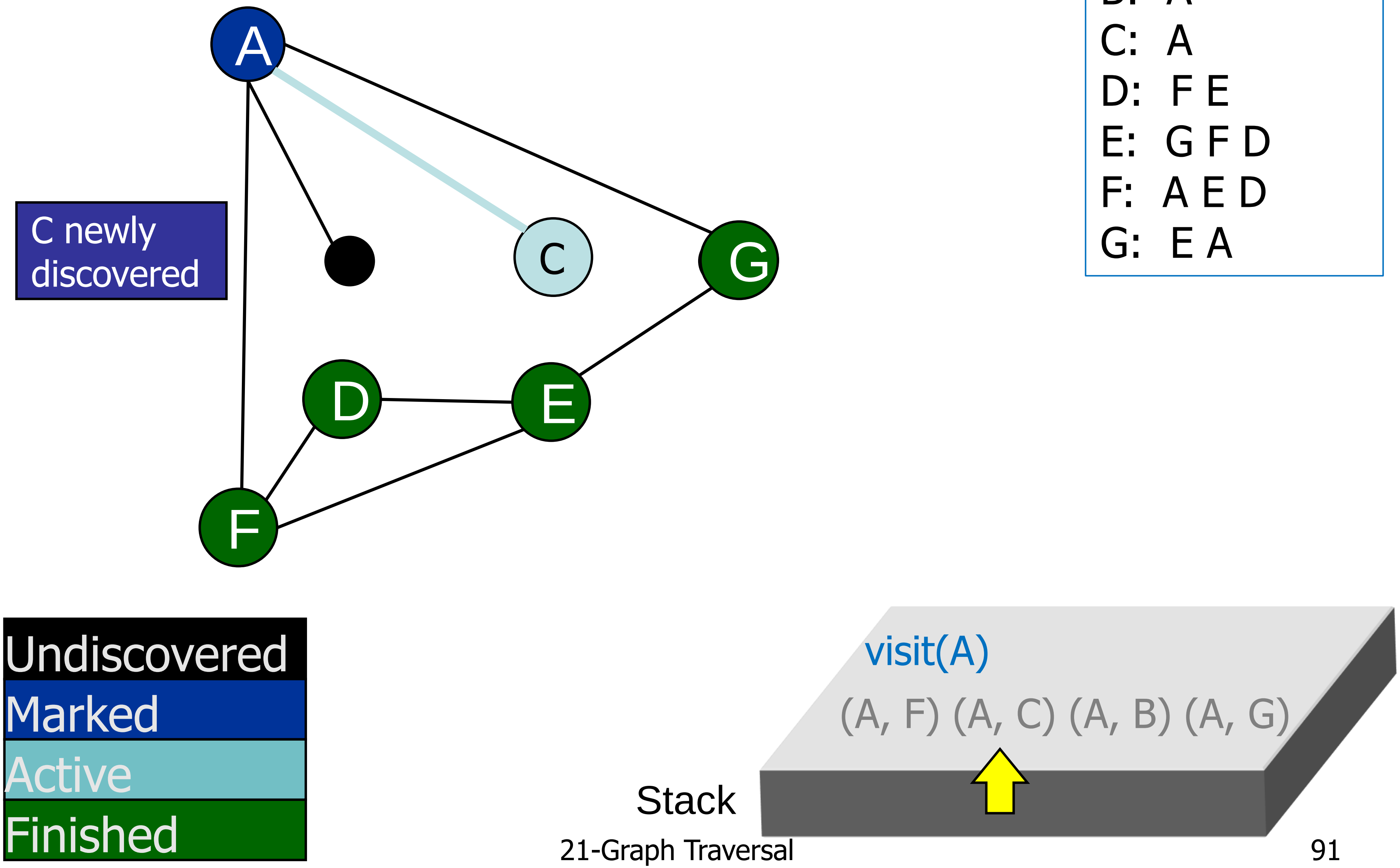
Stack

Undiscovered
Marked
Active
Finished

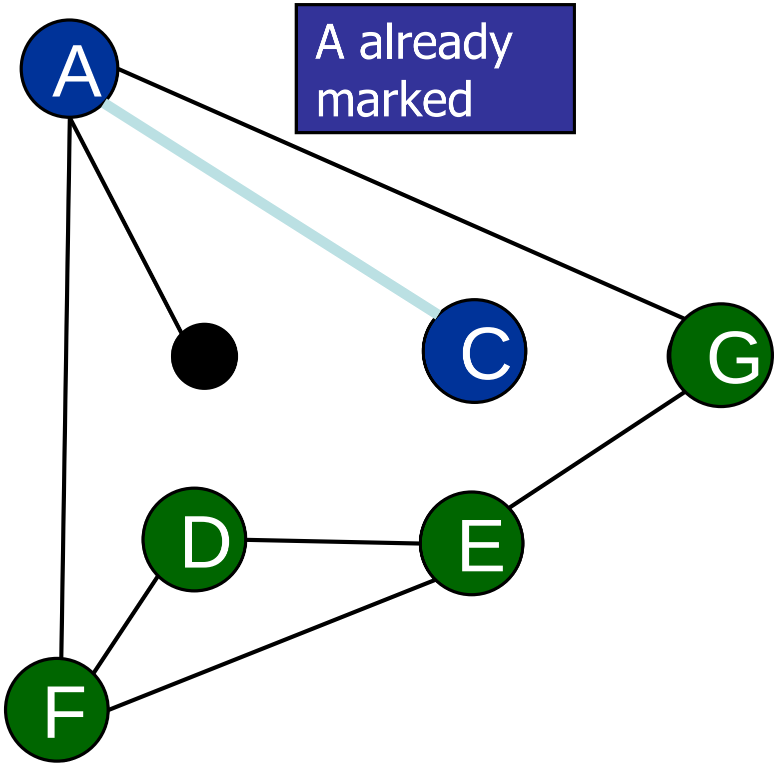
Depth-First Search – Example

Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

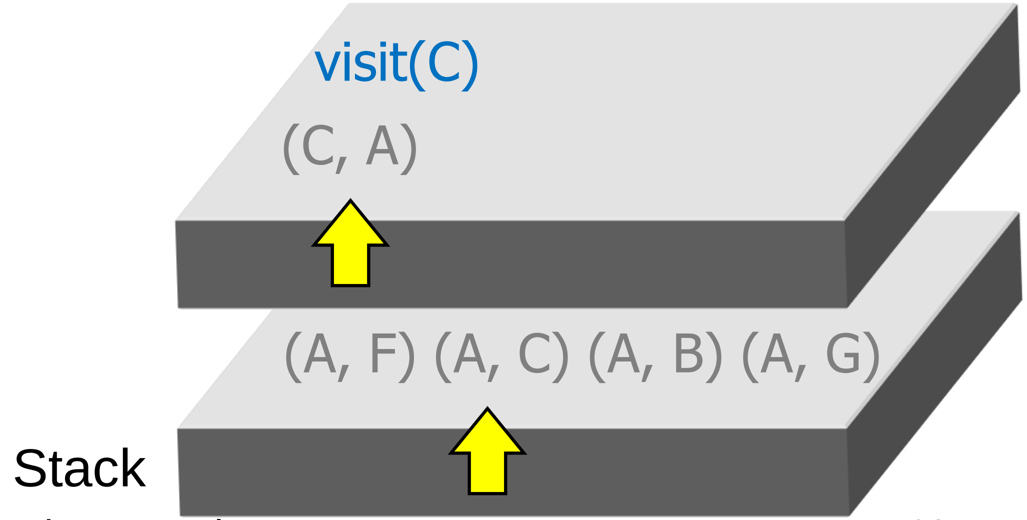


Depth-First Search – Example



Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

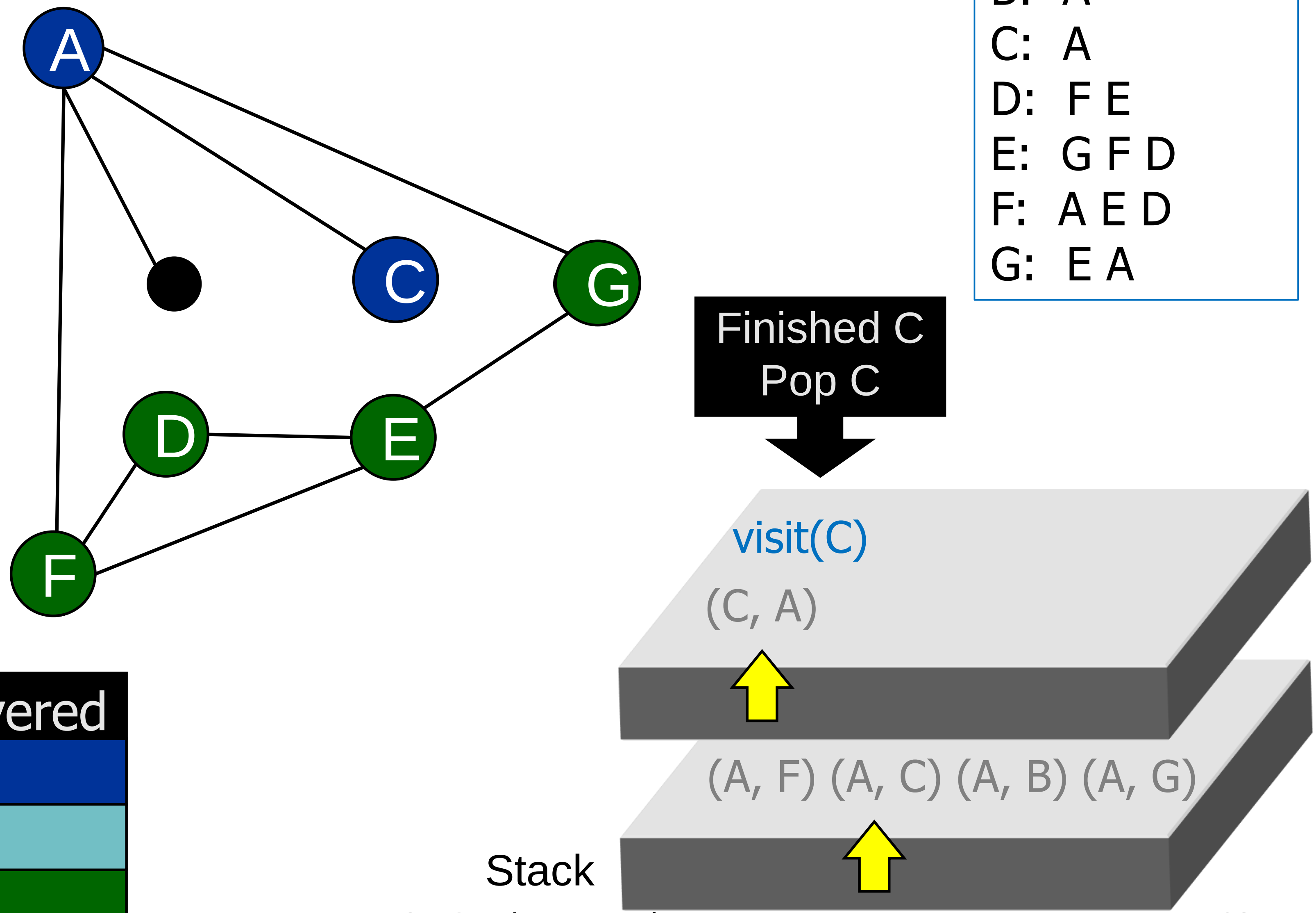


Undiscovered
Marked
Active
Finished

Depth-First Search – Example

Adjacency List

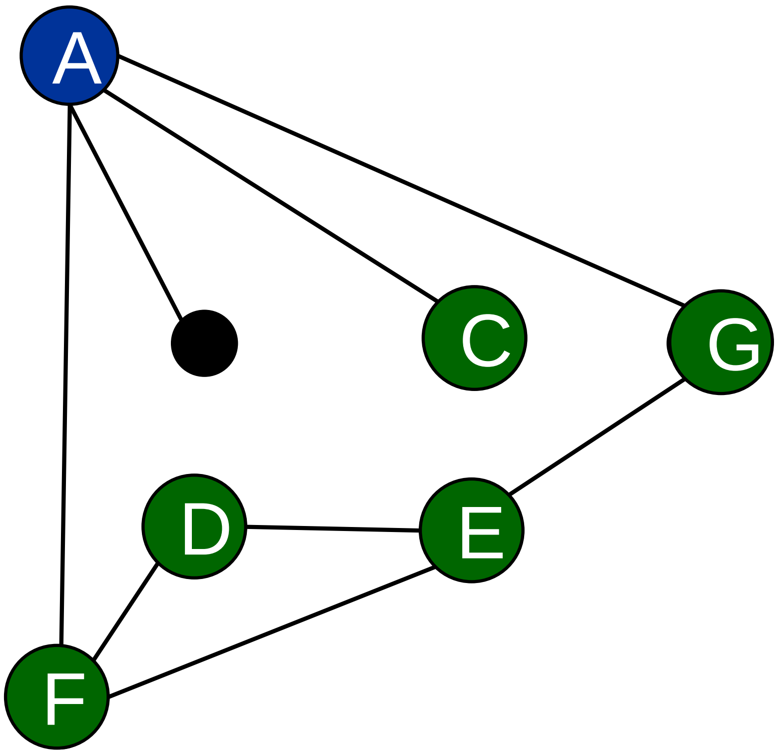
A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A



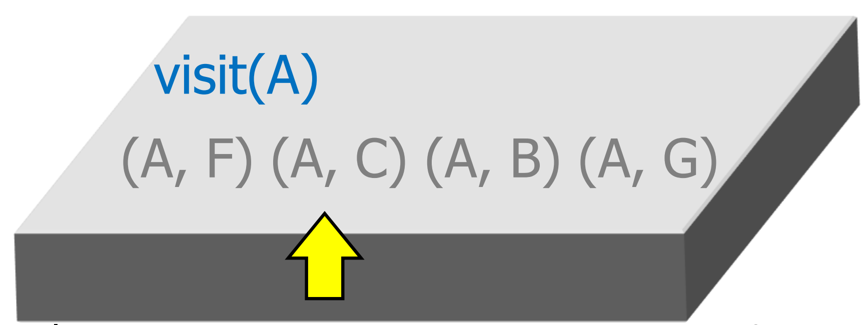
Depth-First Search – Example

Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A



Undiscovered
Marked
Active
Finished



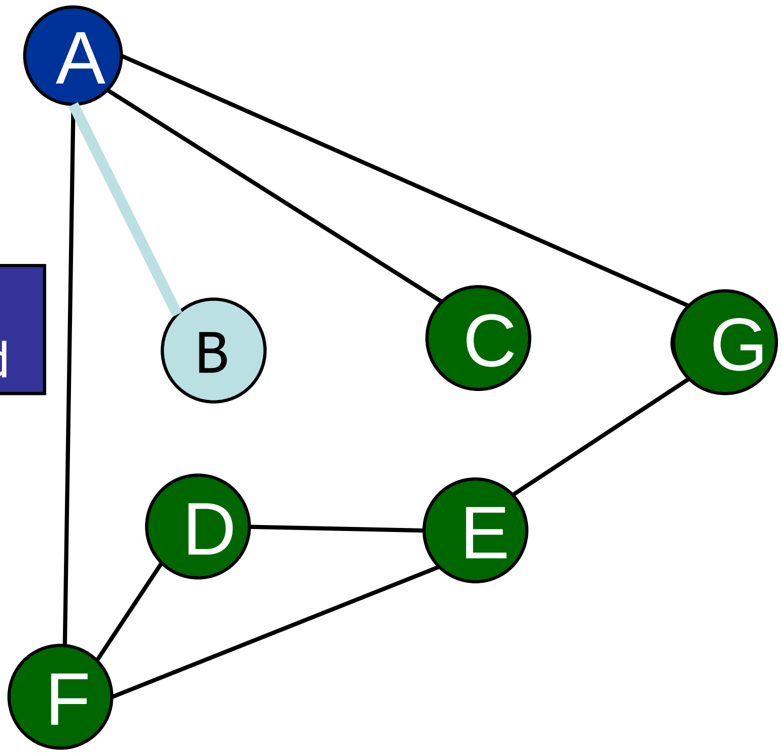
Stack

Depth-First Search – Example

Adjacency List

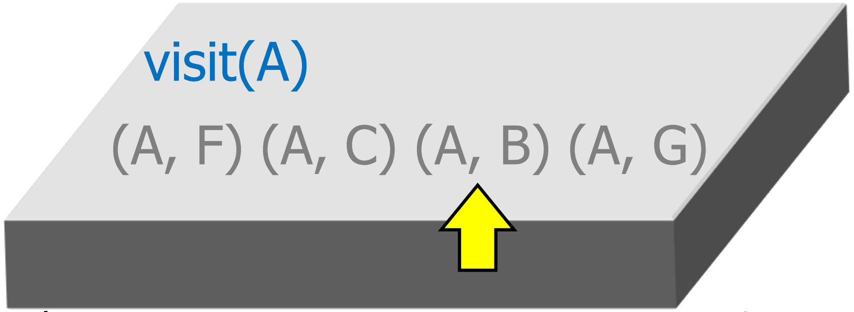
A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

B newly discovered



Undiscovered
Marked
Active
Finished

Stack

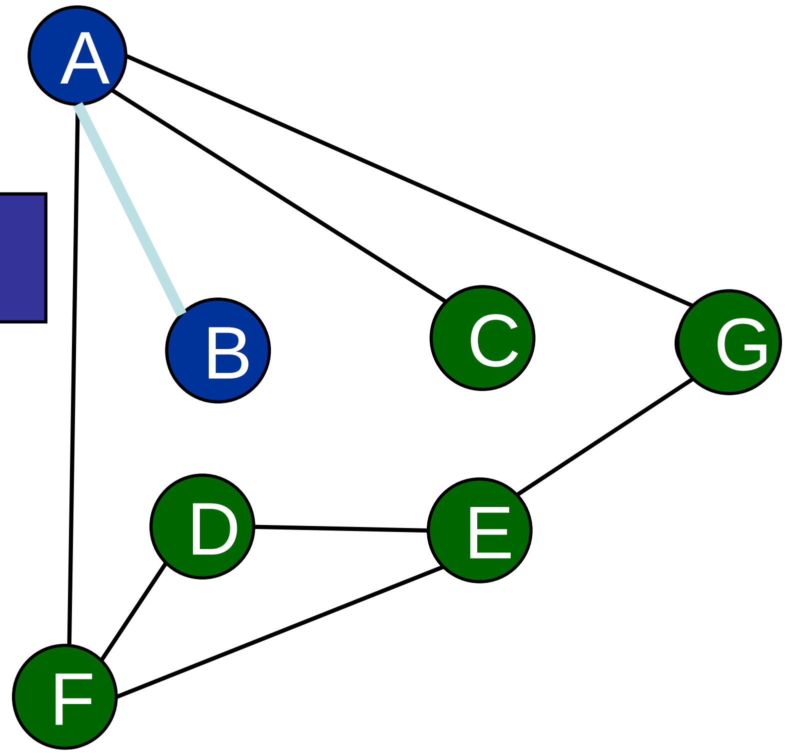


Depth-First Search – Example

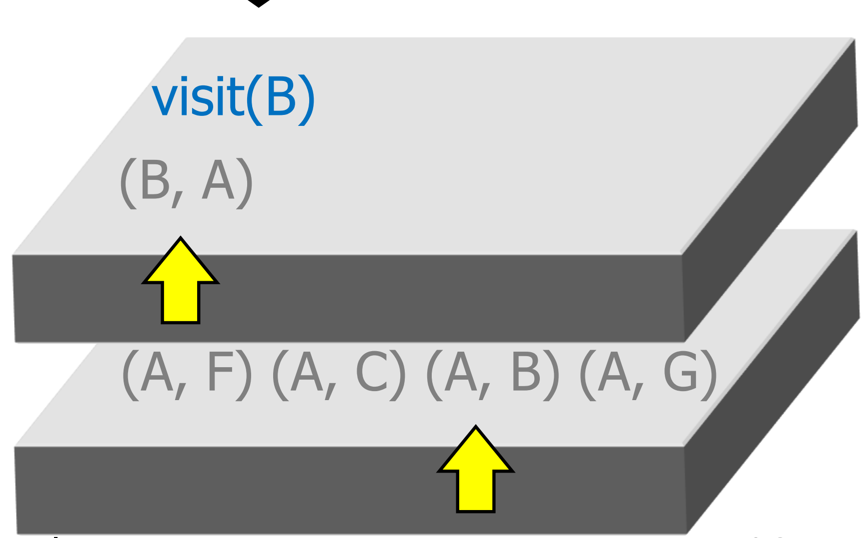
Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

A already marked

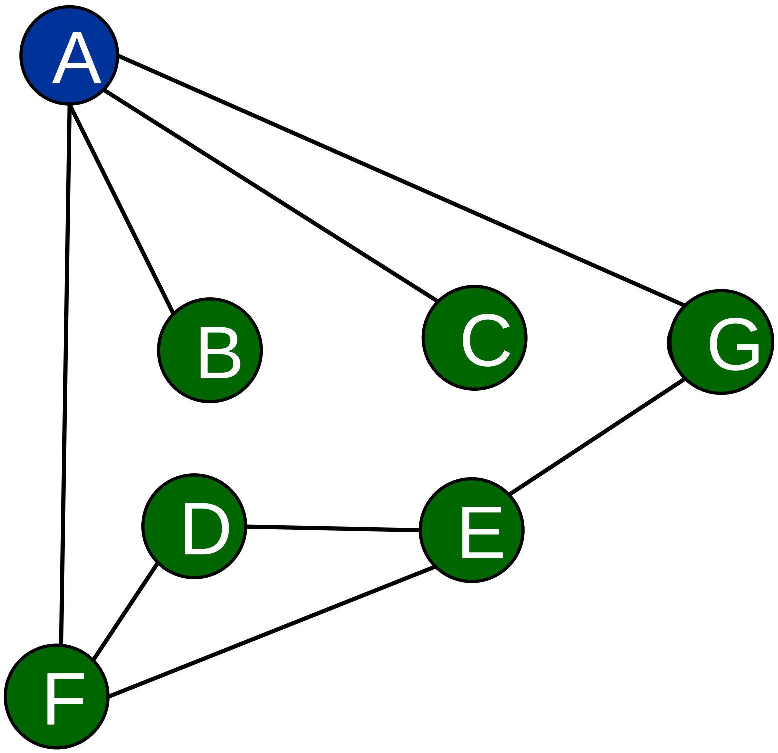


Finished B
Pop B



Undiscovered
Marked
Active
Finished

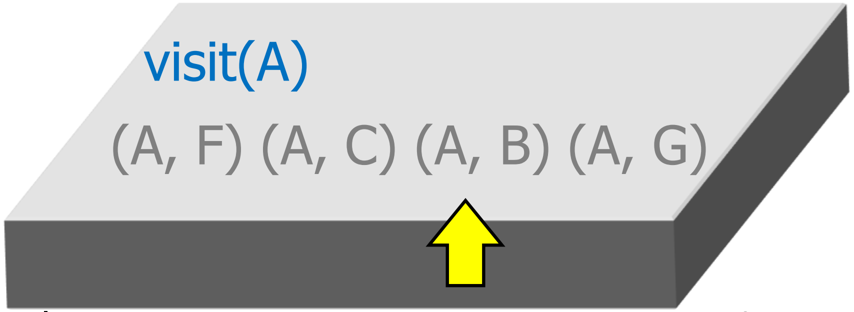
Depth-First Search – Example



Adjacency List

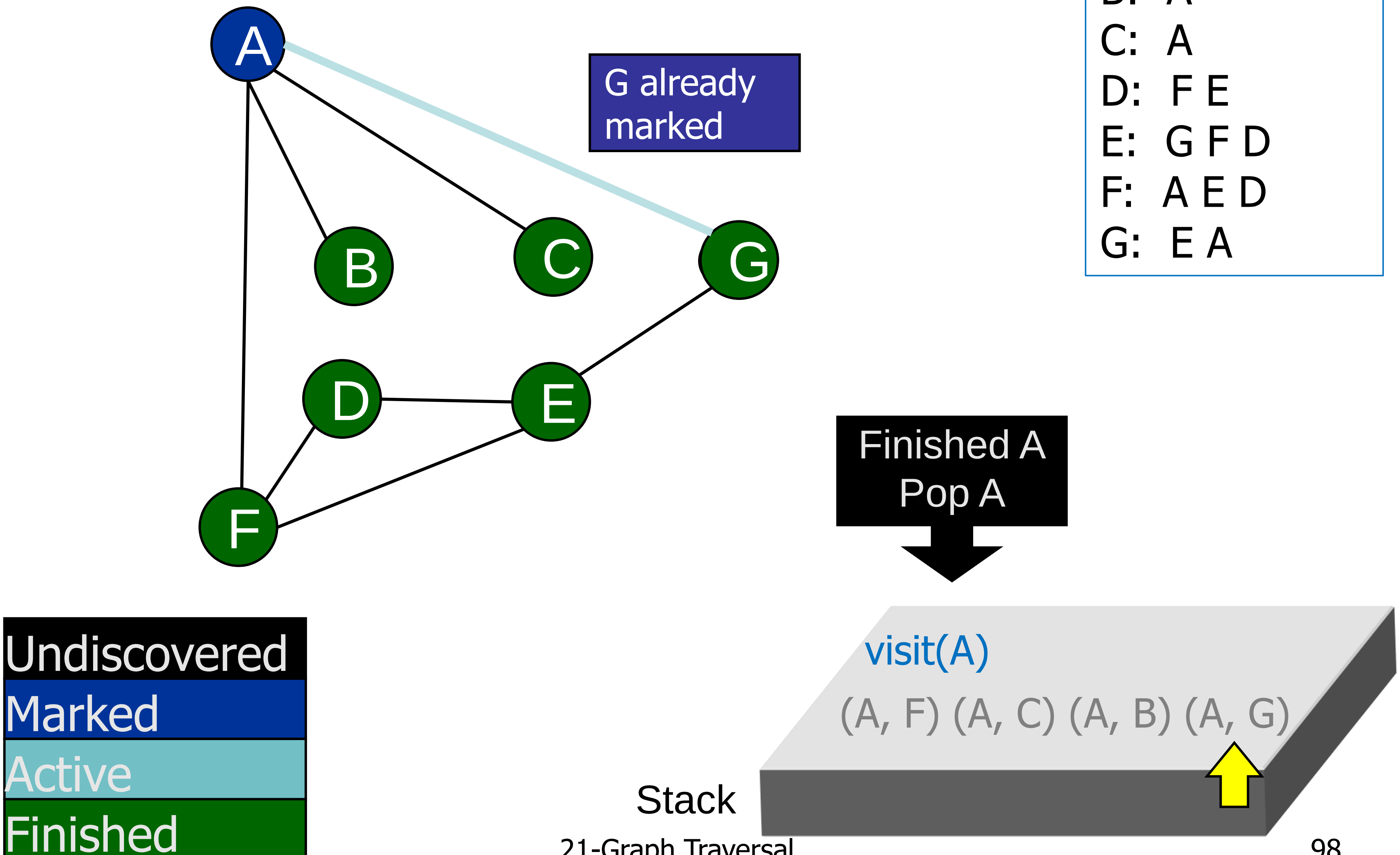
A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

Undiscovered
Marked
Active
Finished



Stack

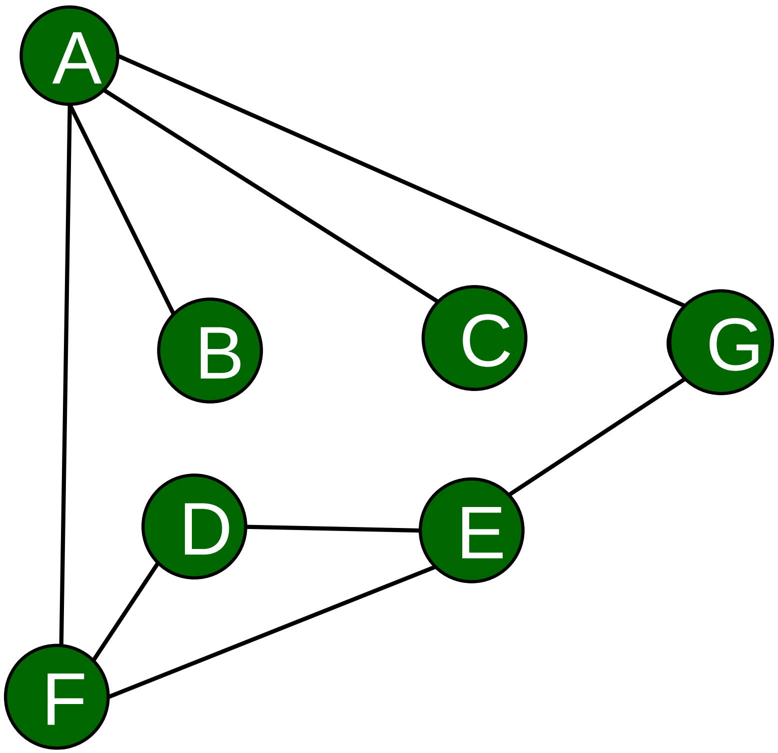
Depth-First Search – Example



<u>Adjacency List</u>

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

Depth-First Search – Example



Adjacency List

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

Undiscovered
Marked
Active
Finished

BFS vs. DFS

- Depending on the application, either DFS or BFS could be advantageous

- Example: Consider your family tree
  - If you are searching for some of your siblings cousins then it would be safe to assume that person would be on the bottom of the tree
  - Which approach is better in this case?
    - In general, both approaches have the same time complexity
    - In worst case, they need to visit all the nodes

Applications of Graph Traversal

- Determining connectedness and finding connected sub-graphs
- Construct a BFS or DFS tree/forest from a graph
- Determining the path length from one vertex to all others
  - Find the shortest path from a vertex s to a vertex v (BFS)

Any Question So Far?

Any Question So Far?