# RECITATION 2
# ANALYSIS OF ALGORITHMS II

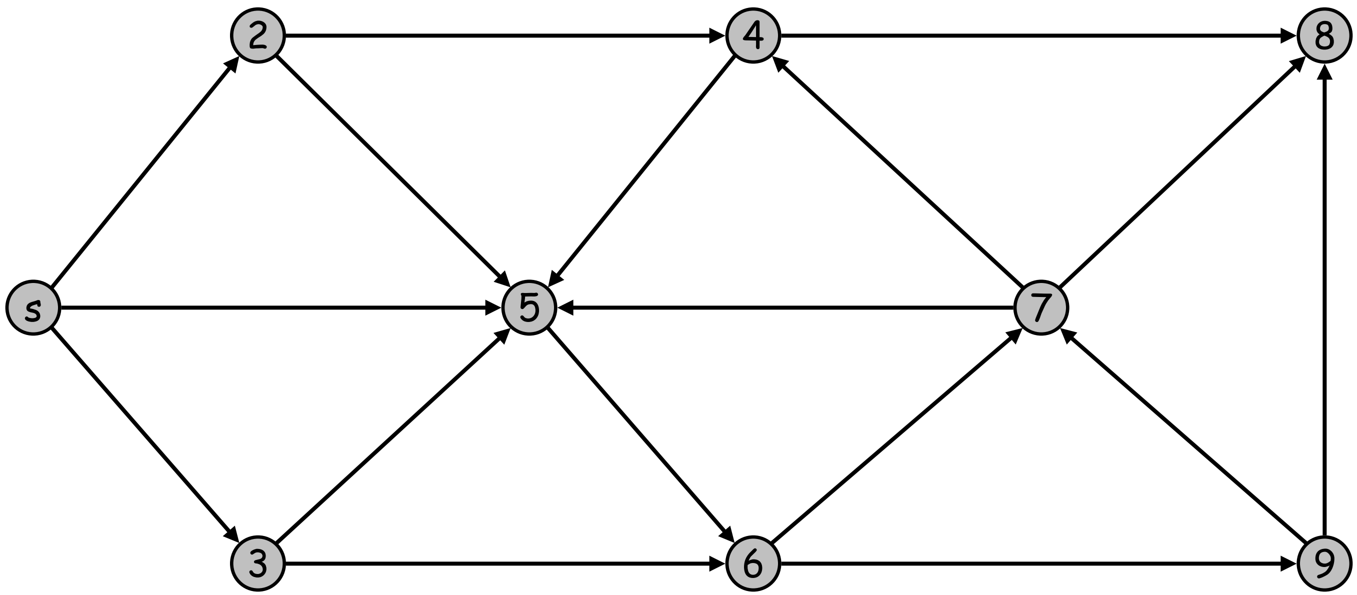
2017 SPRING
Gönül Uludağ

# Graph Search

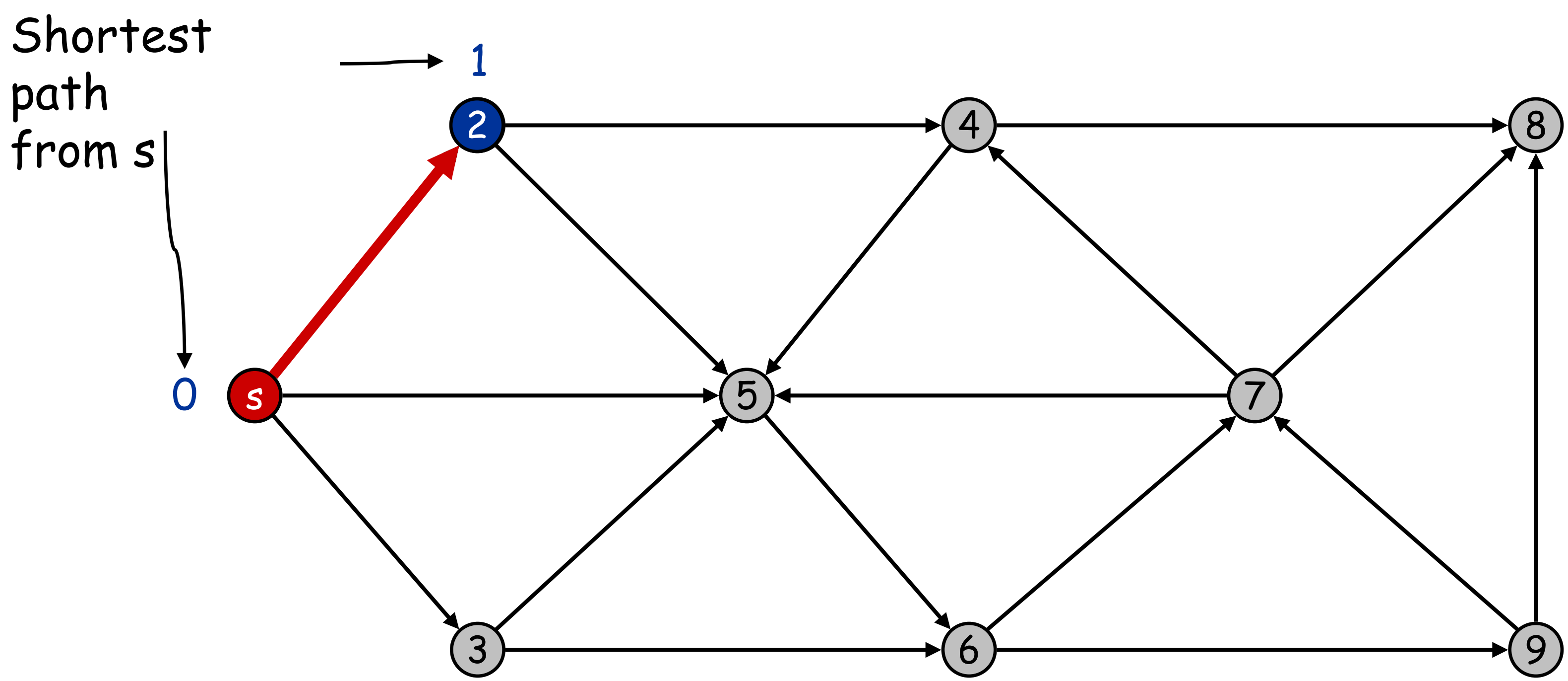
Given an arbitrary graph $G = (V, E)$ and a starting node $s \in V$, Breadth-First Search (**BFS**) and Depth-First Search (**DFS**), find shortest paths from s to each reachable node v.

- **BFS** algorithm explores outward in all directions uniformly.
- **DFS** algorithm explores out in one direction, backing up when necessary.
  - BFS always finds the shortest path from the source node to each other node, while DFS might not.

# Question 1: Directed Breadth First Search



# Breadth First Search




Undiscovered

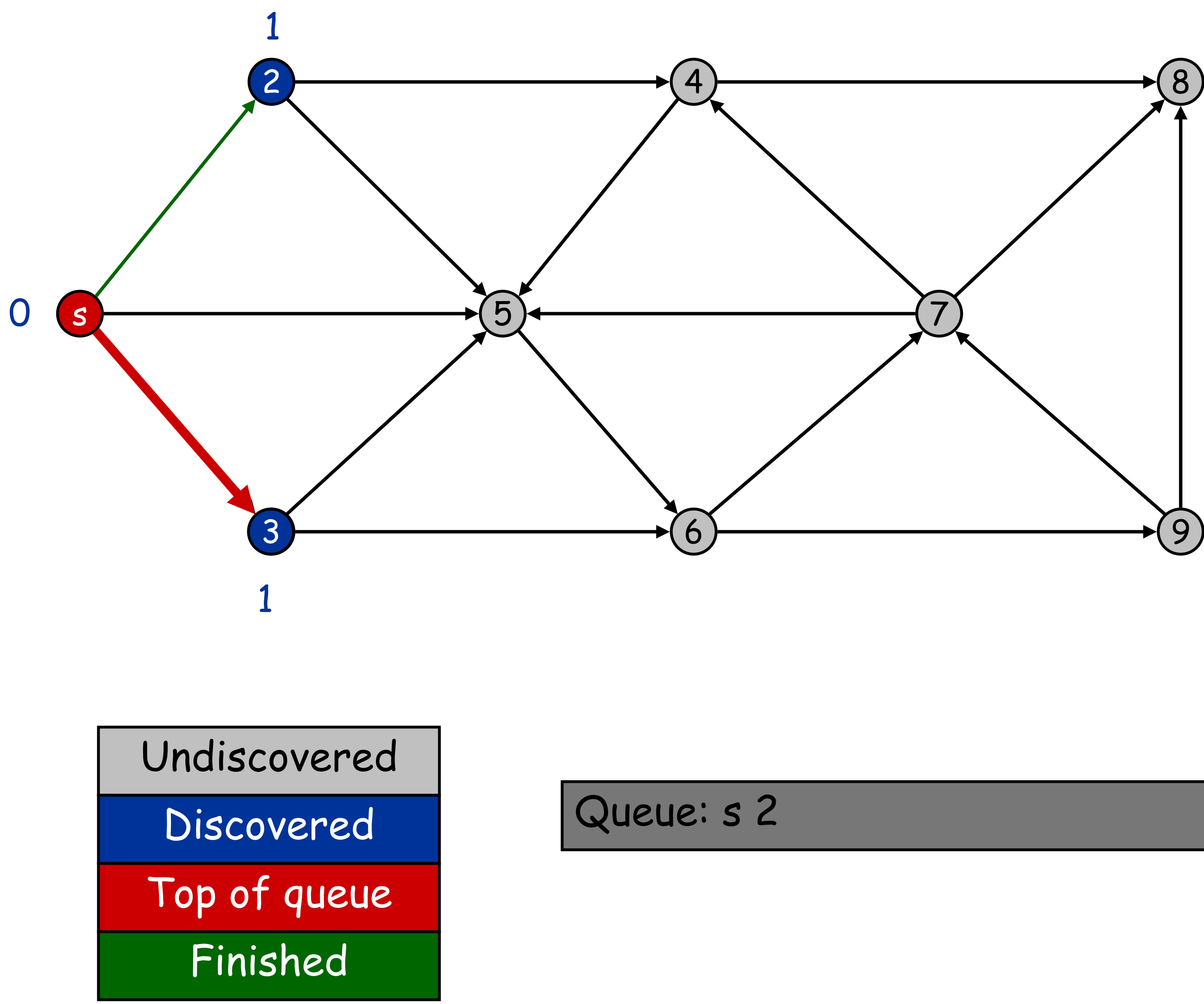
Discovered

Top of queue

Finished

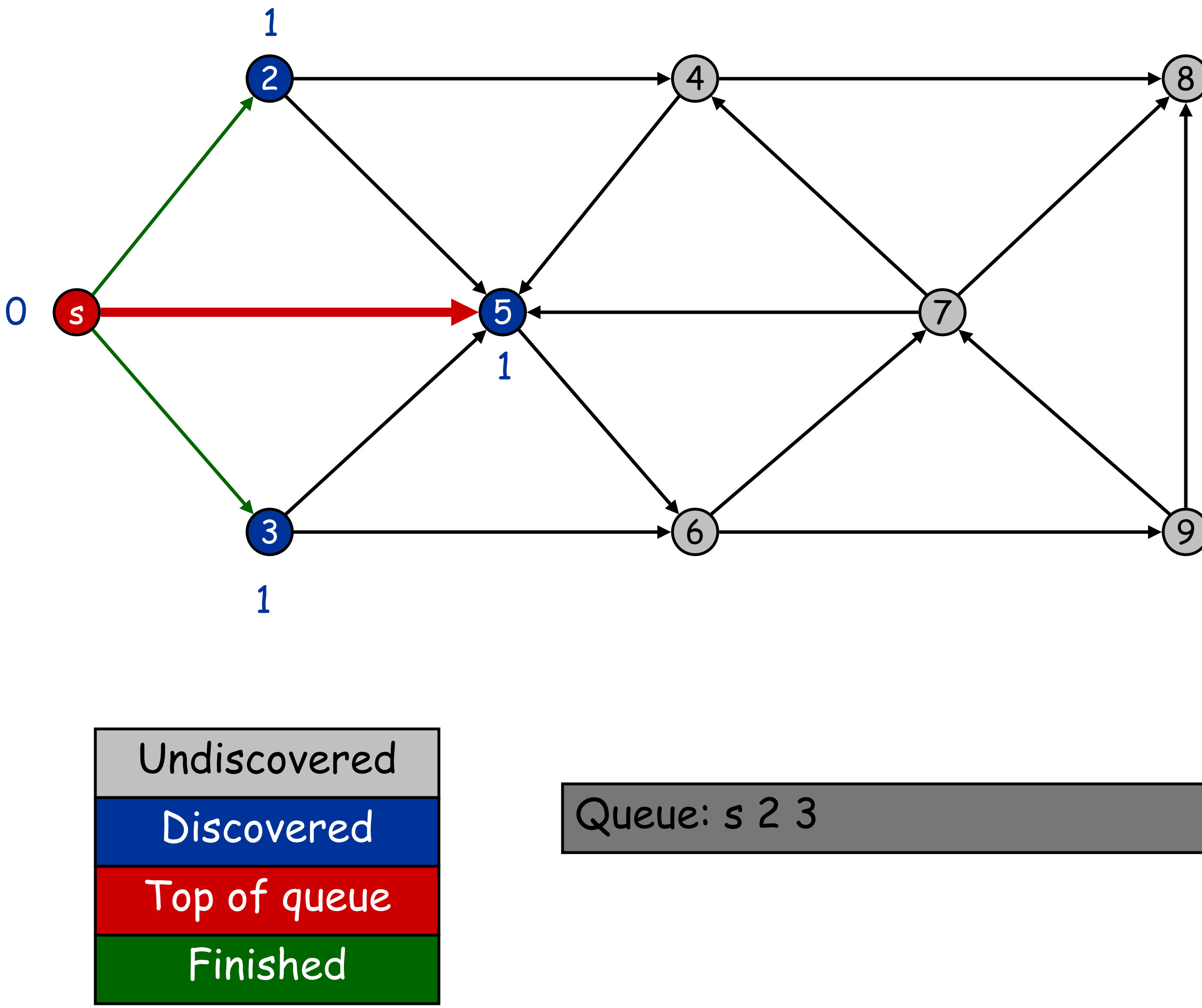
Queue: s

# Breadth First Search



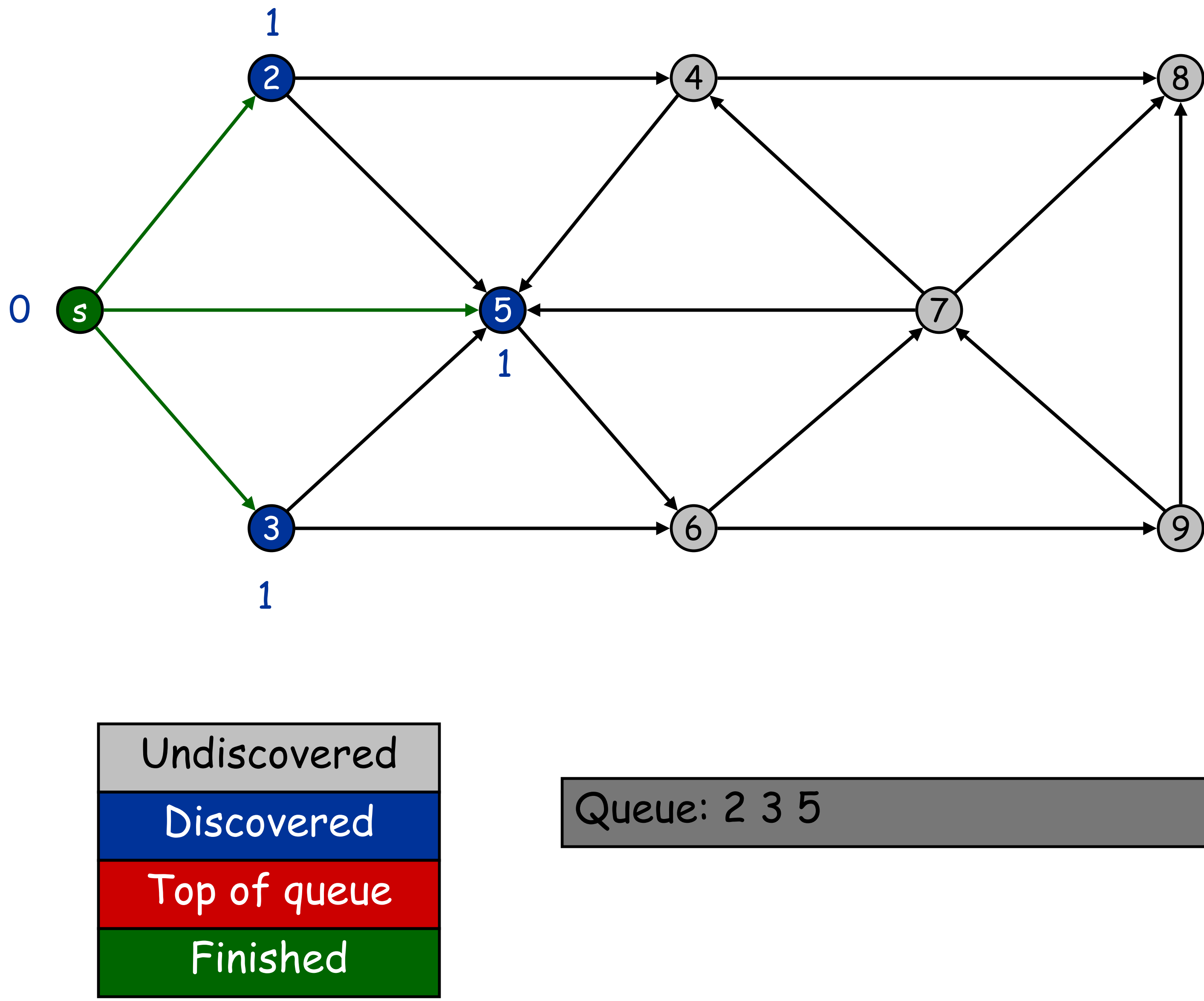

Queue: s 2

# Breadth First Search



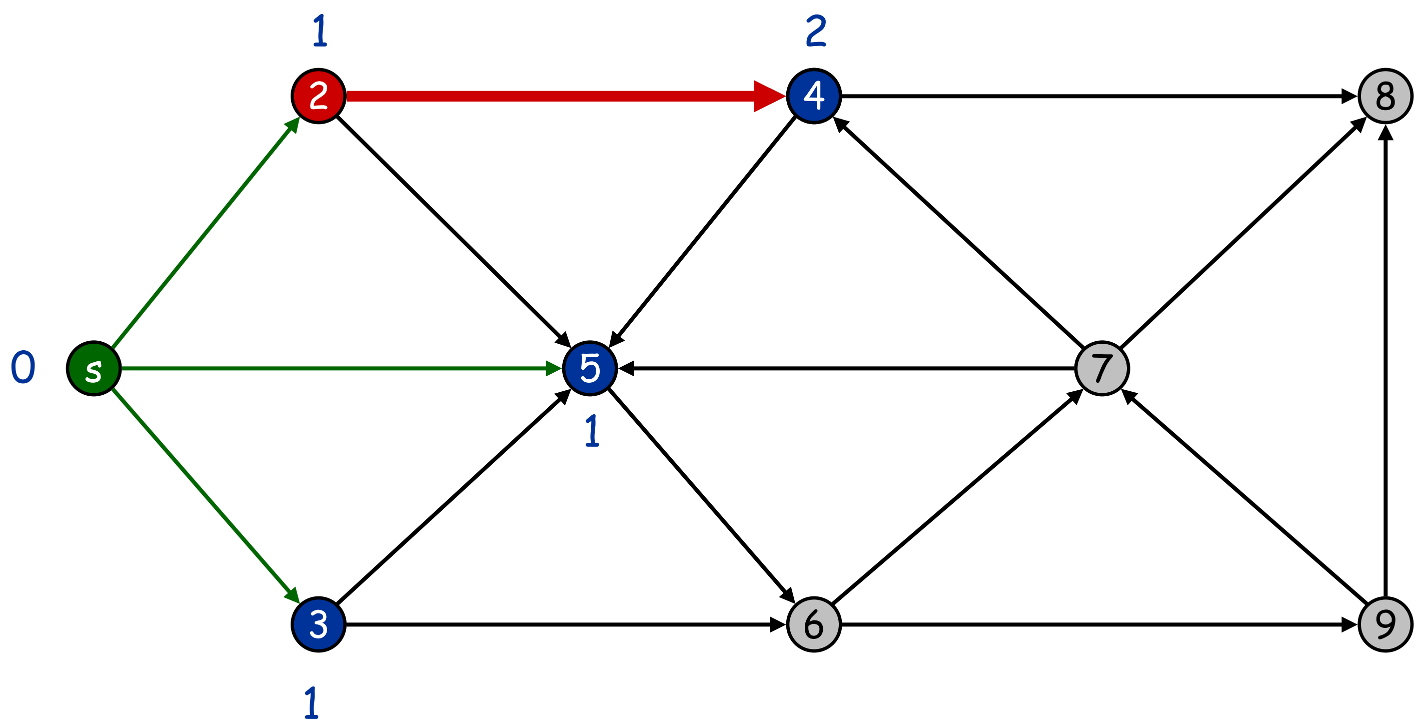

Queue: s 2 3

# Breadth First Search




Queue: 2 3 5

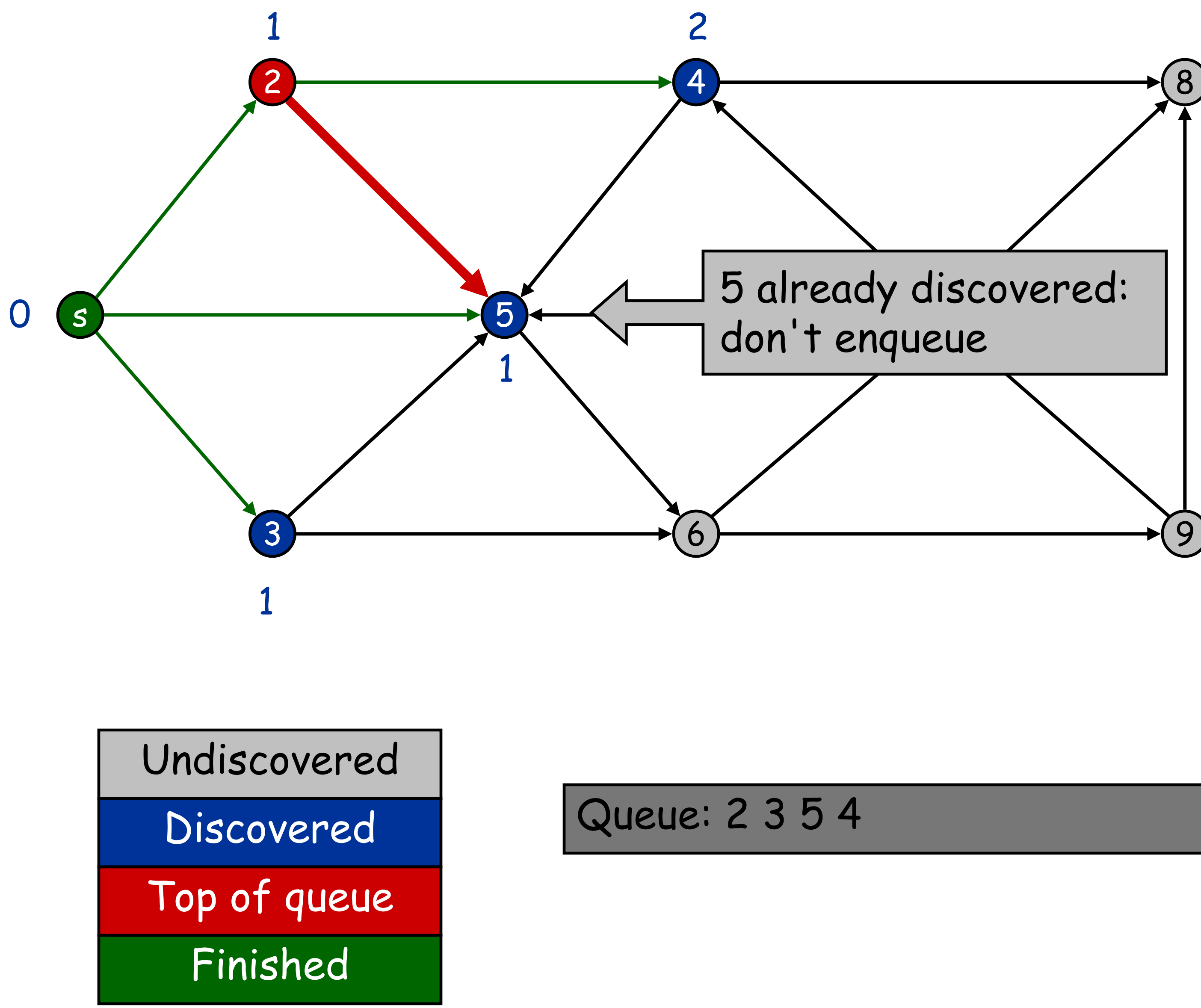
# Breadth First Search





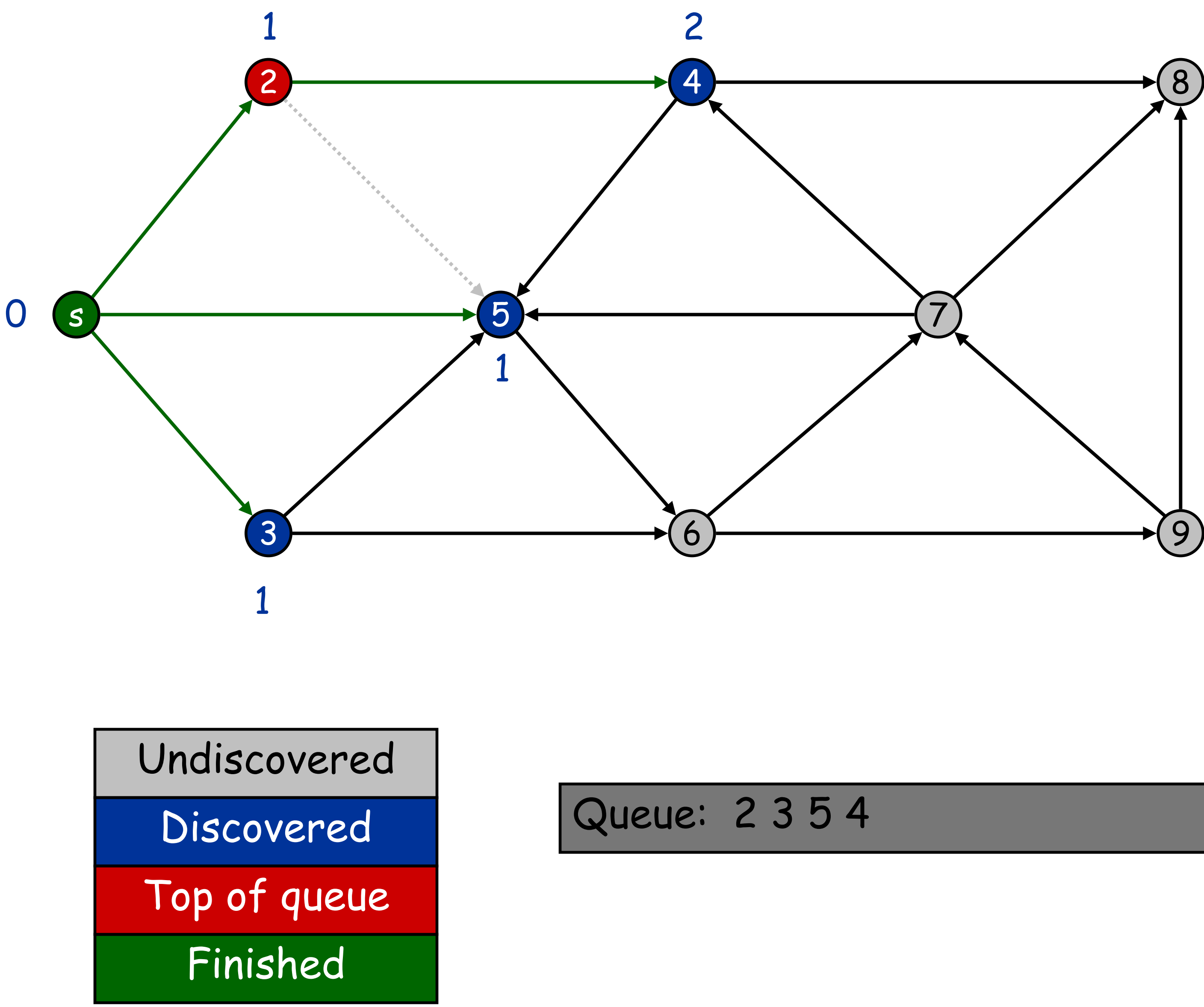
Queue: 2 3 5

# Breadth First Search



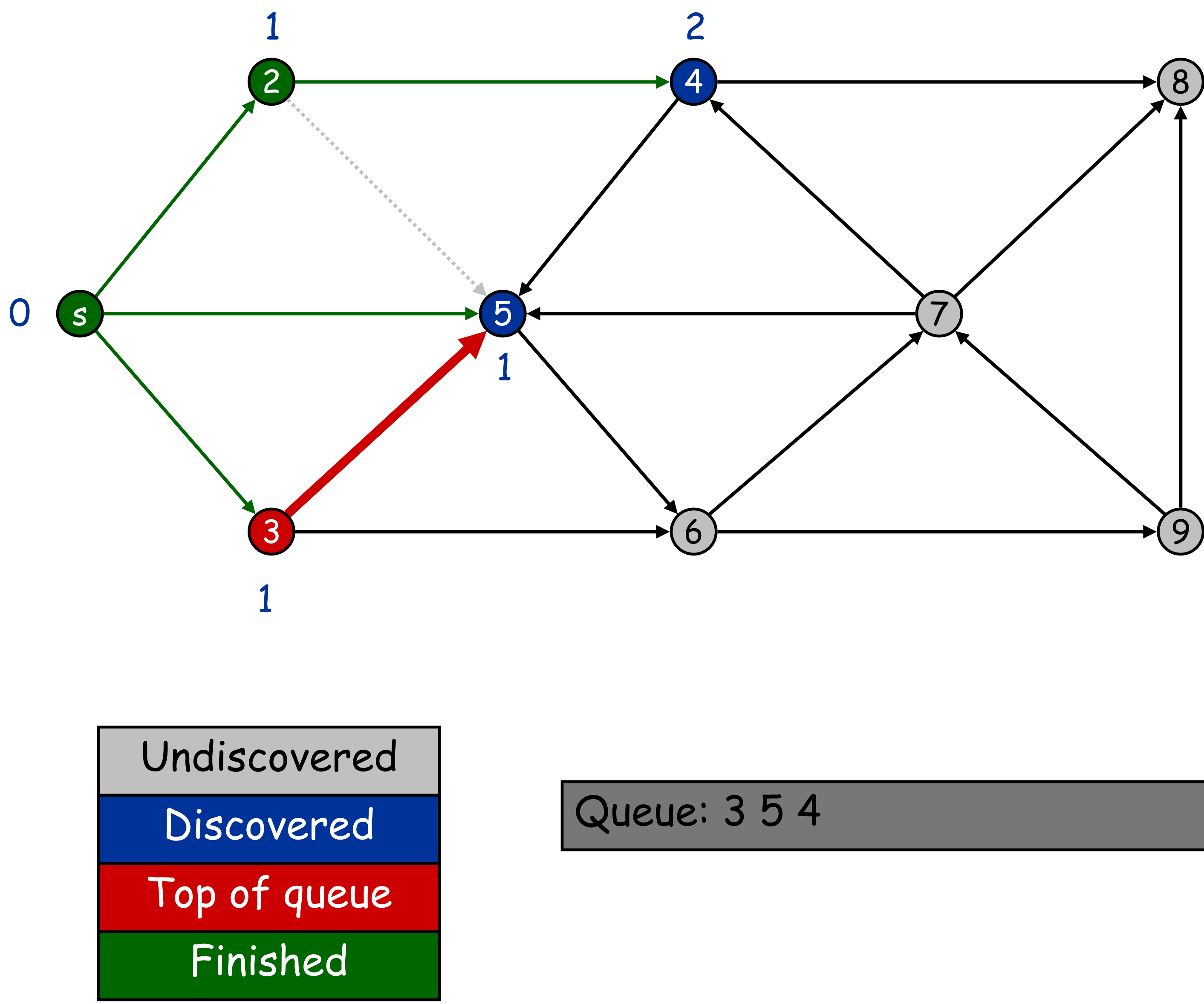

Queue: 2 3 5 4

# Breadth First Search



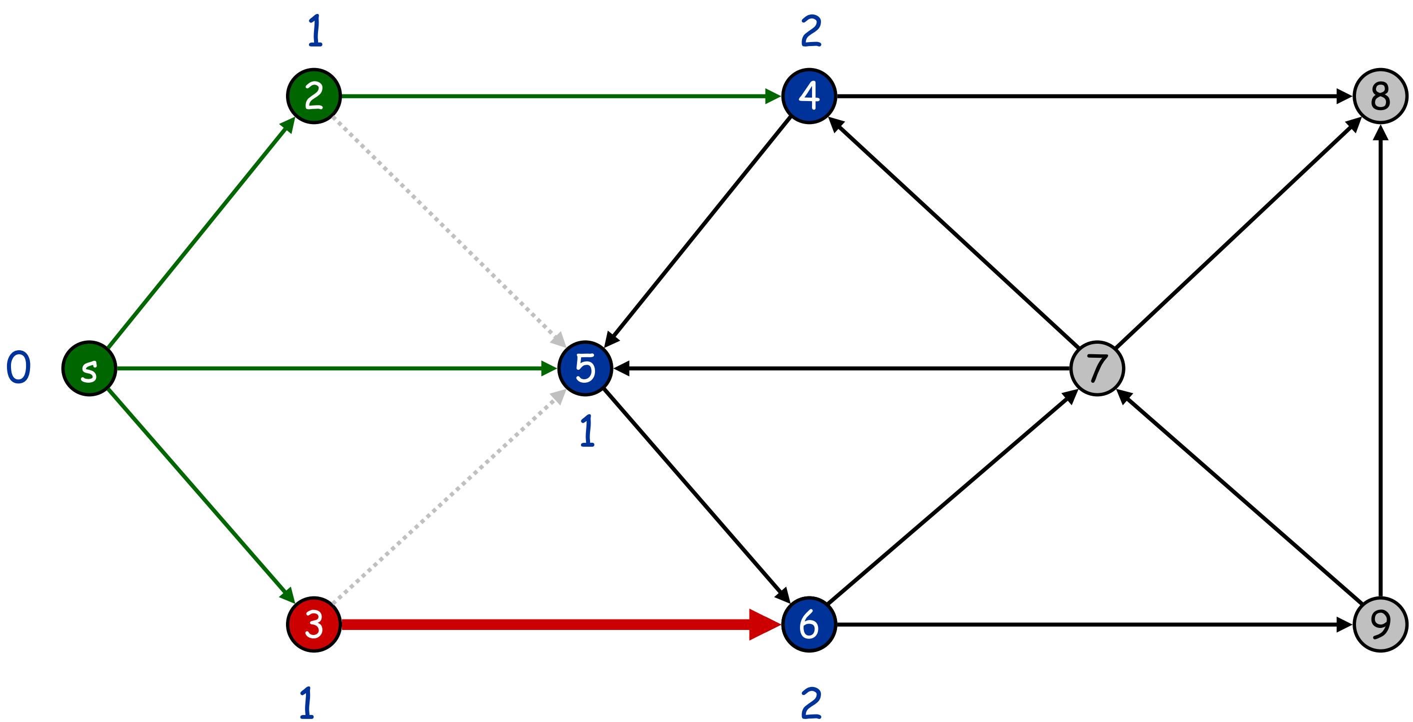

Queue: 2 3 5 4

# Breadth First Search




Queue: 3 5 4

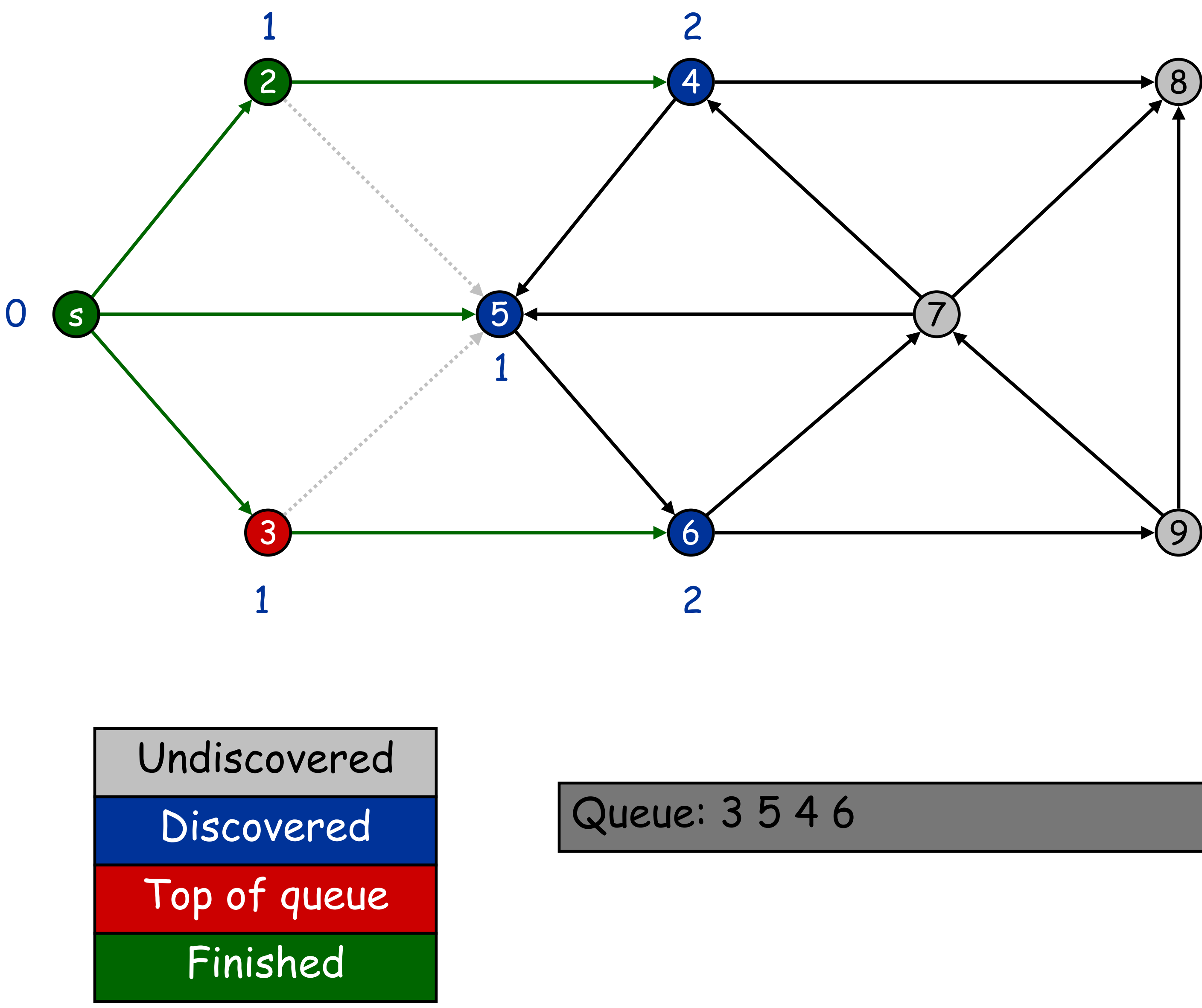
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

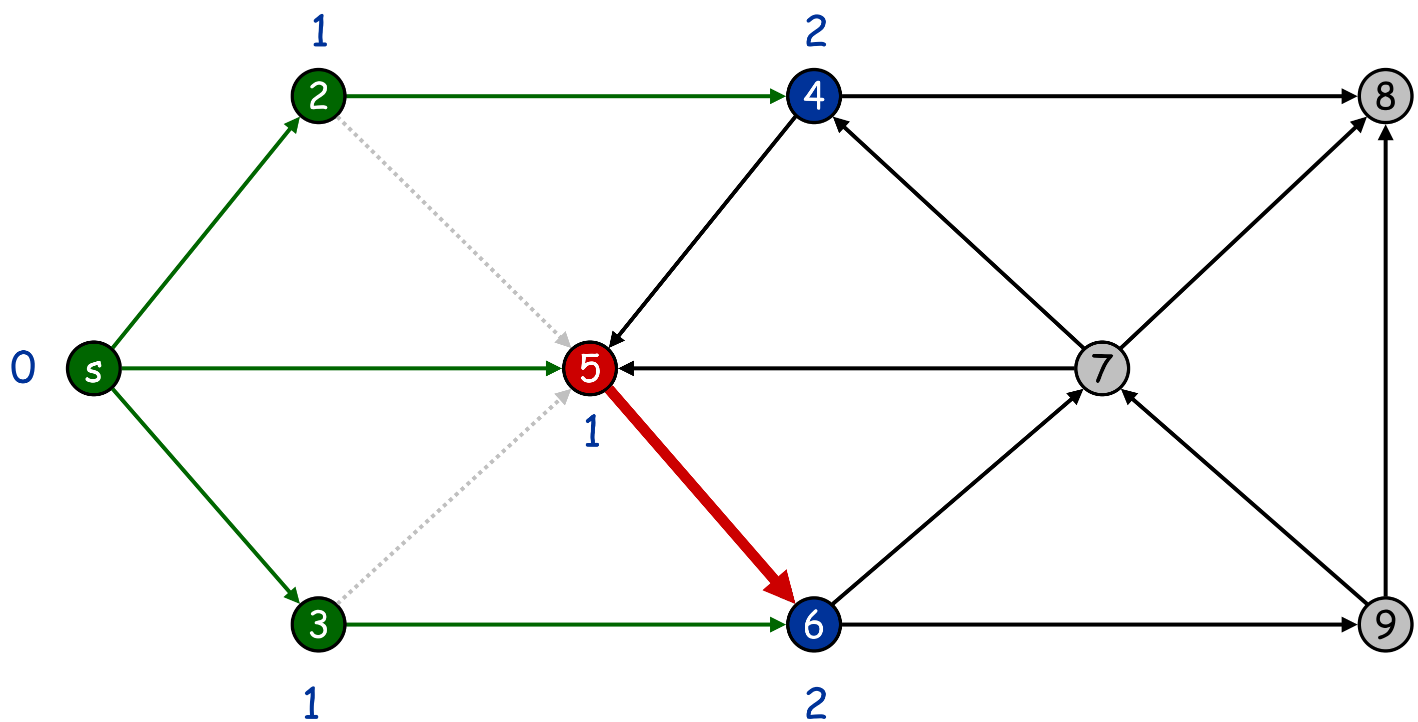
Queue: 3 5 4

# Breadth First Search




Queue: 3 5 4 6

# Breadth First Search




Undiscovered

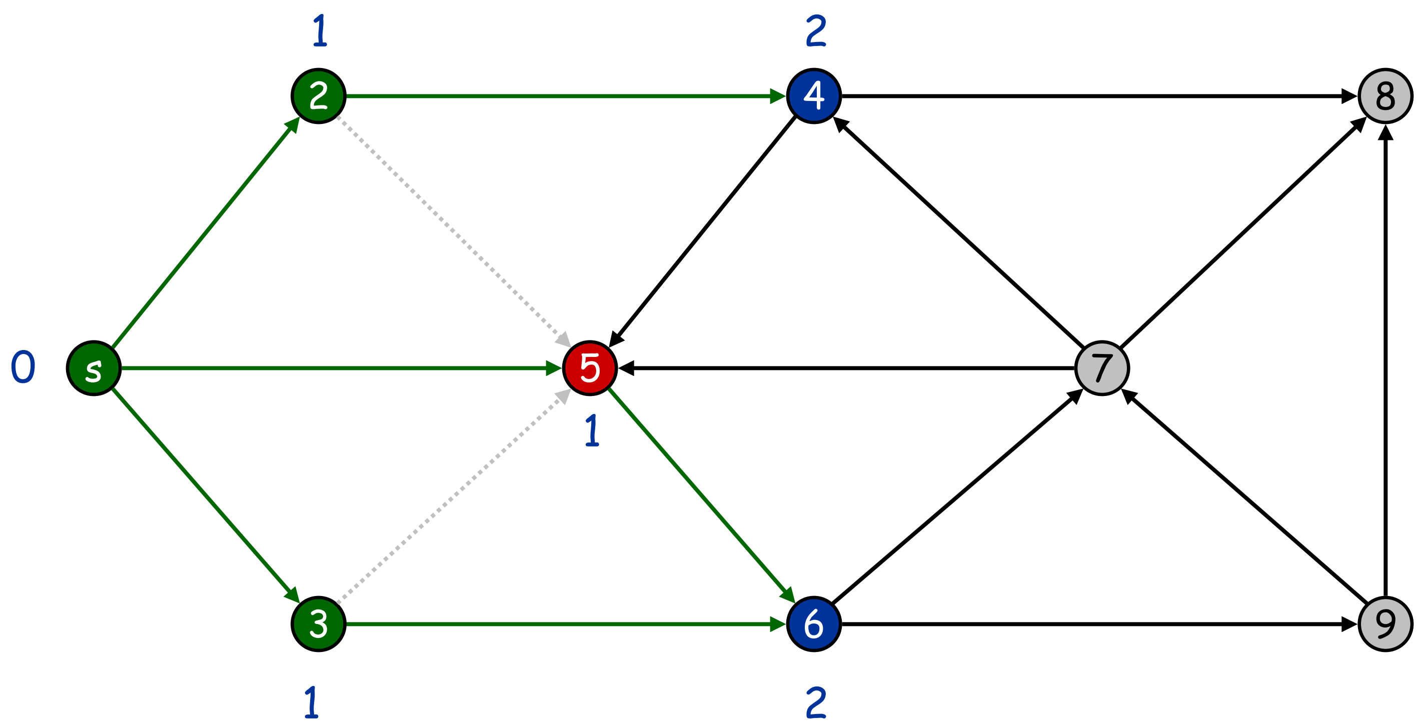
Discovered

Top of queue

Finished

Queue: 5 4 6

# Breadth First Search




Undiscovered

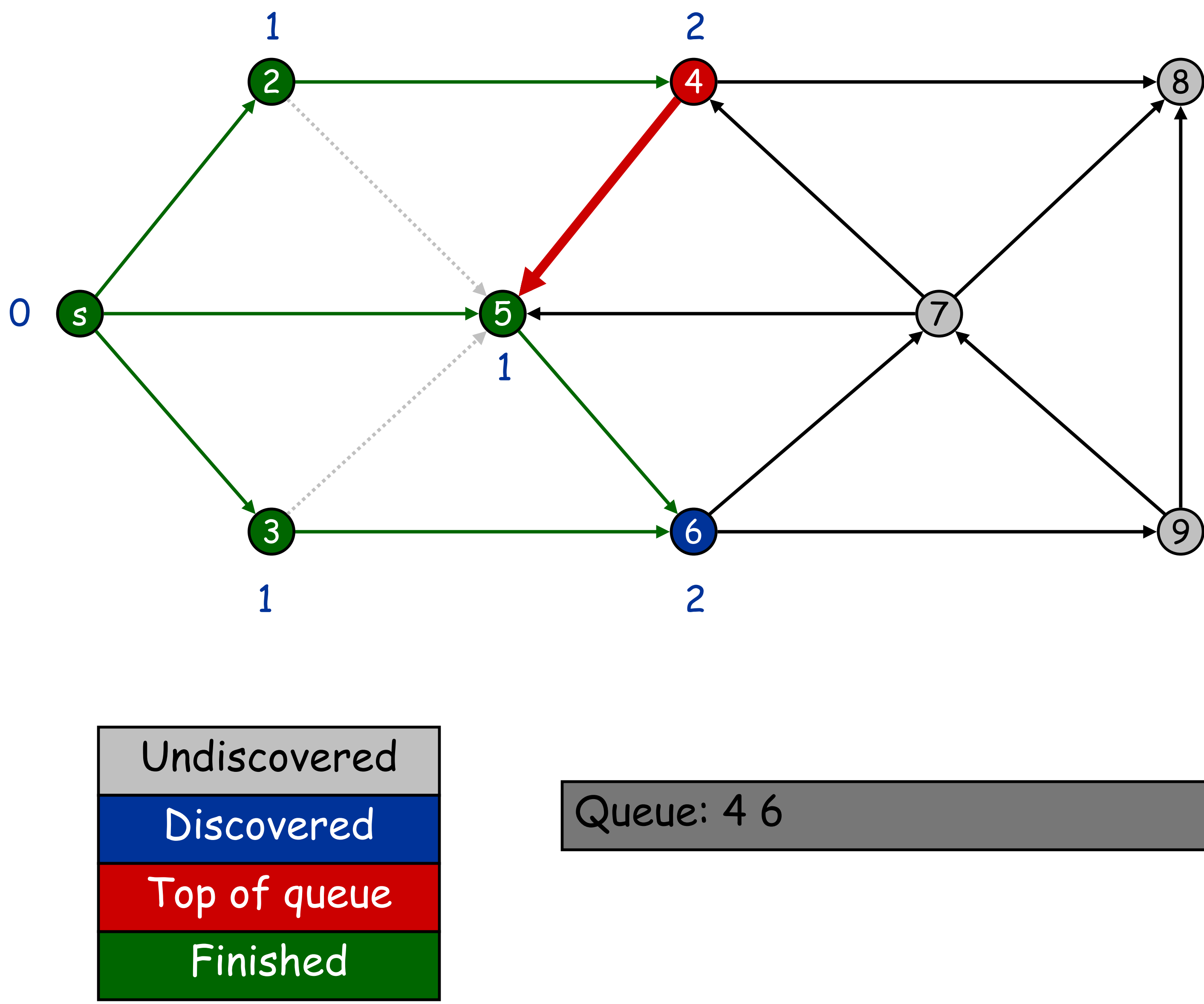
Discovered

Top of queue

Finished

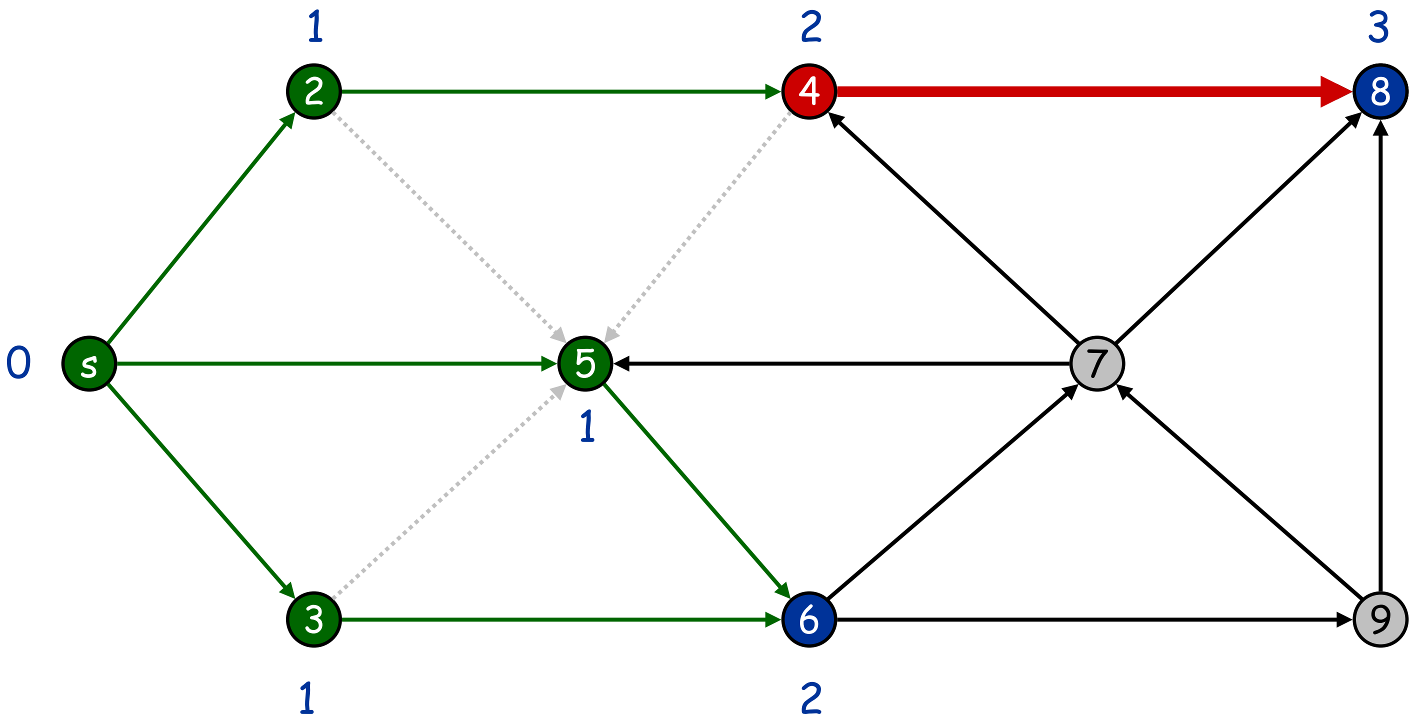
Queue: 5 4 6

# Breadth First Search




Queue: 4 6

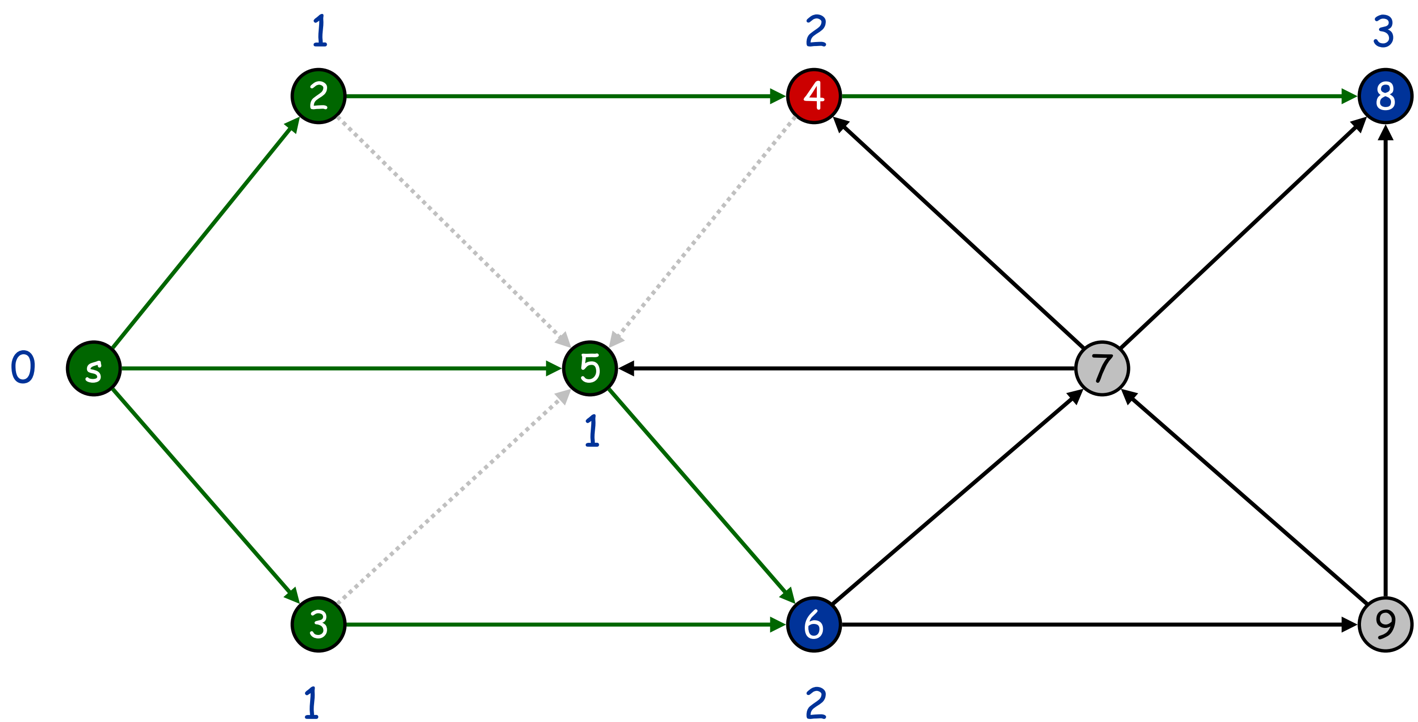
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

Queue: 4 6

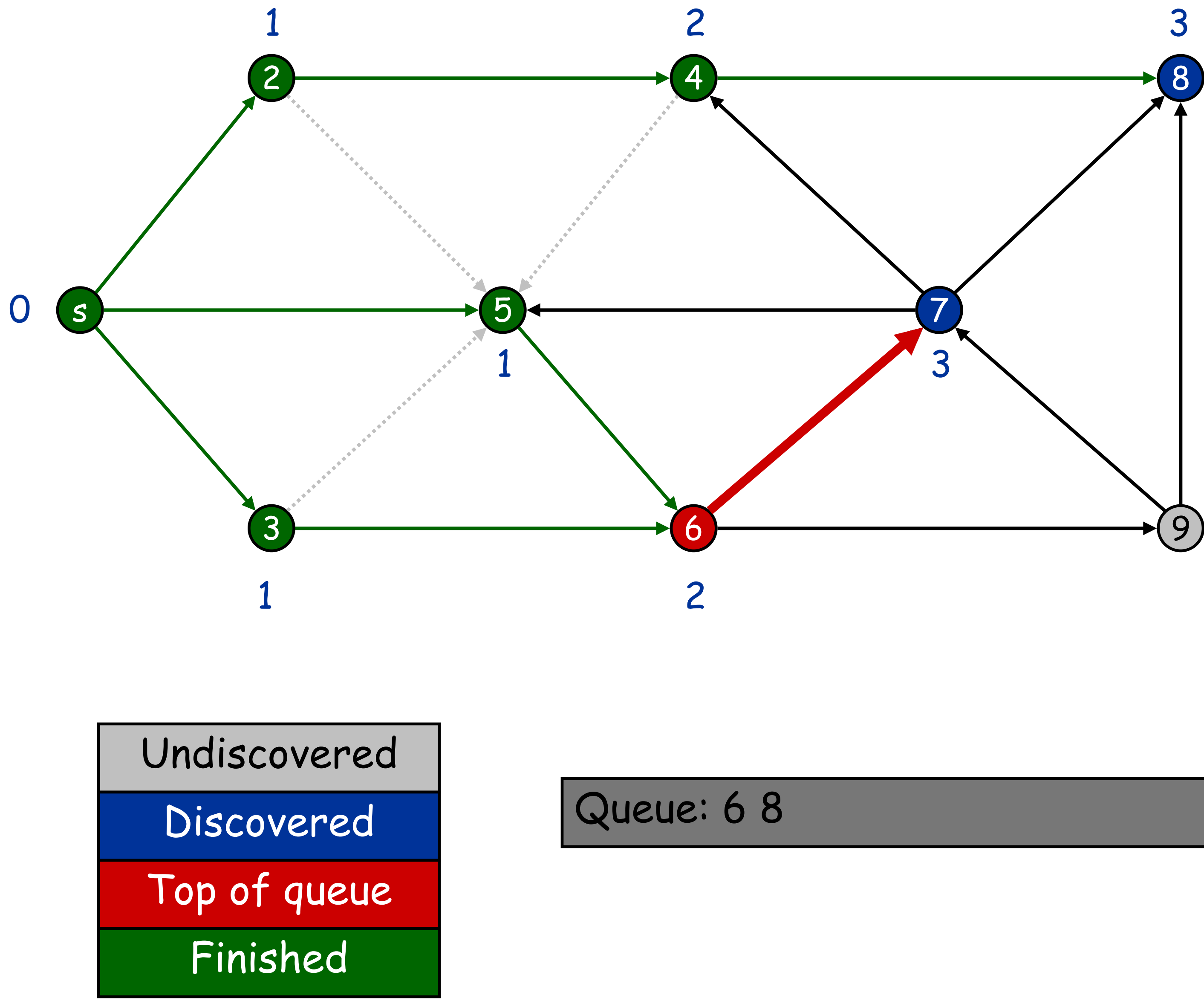
# Breadth First Search





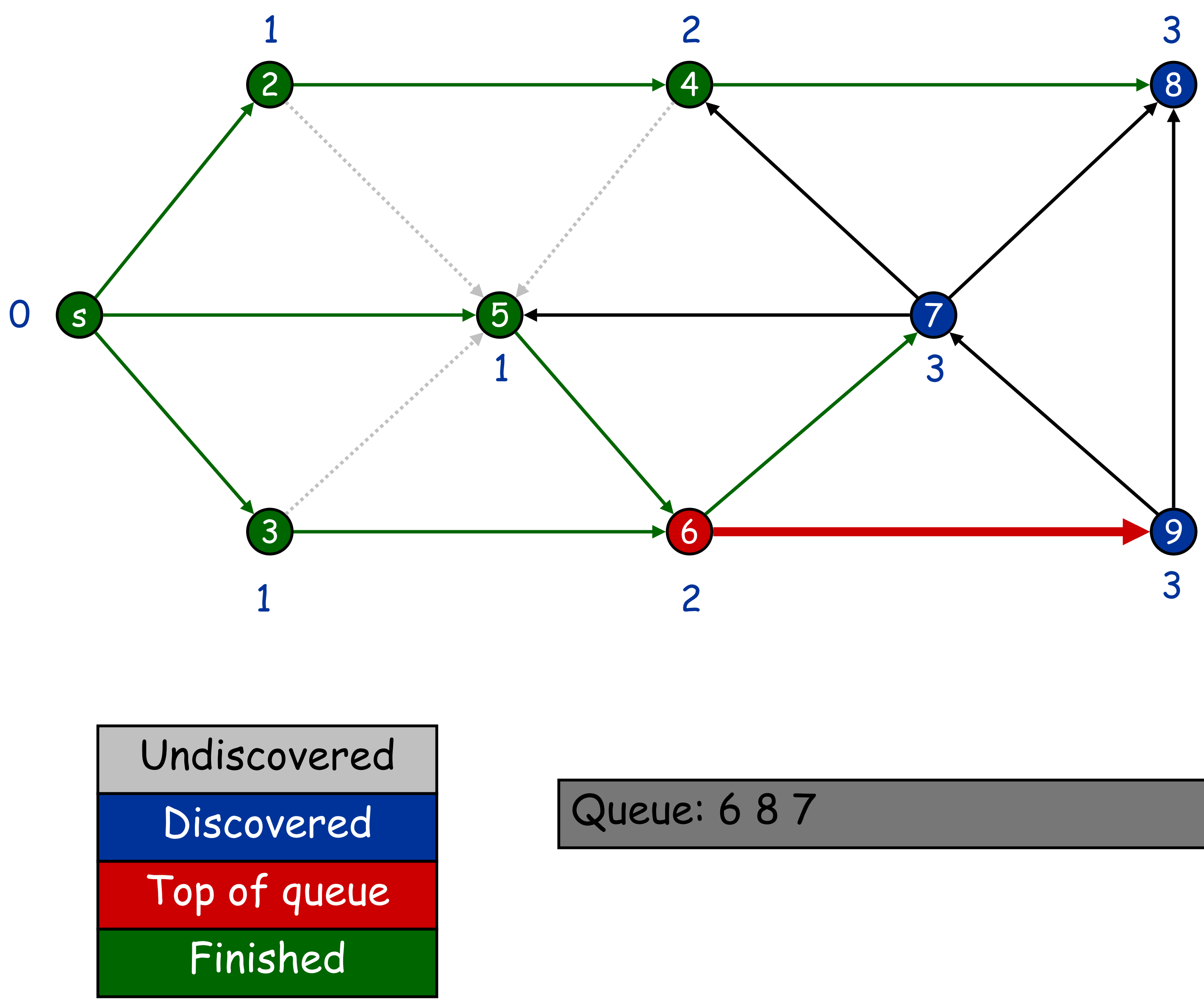
Queue: 4 6 8

# Breadth First Search



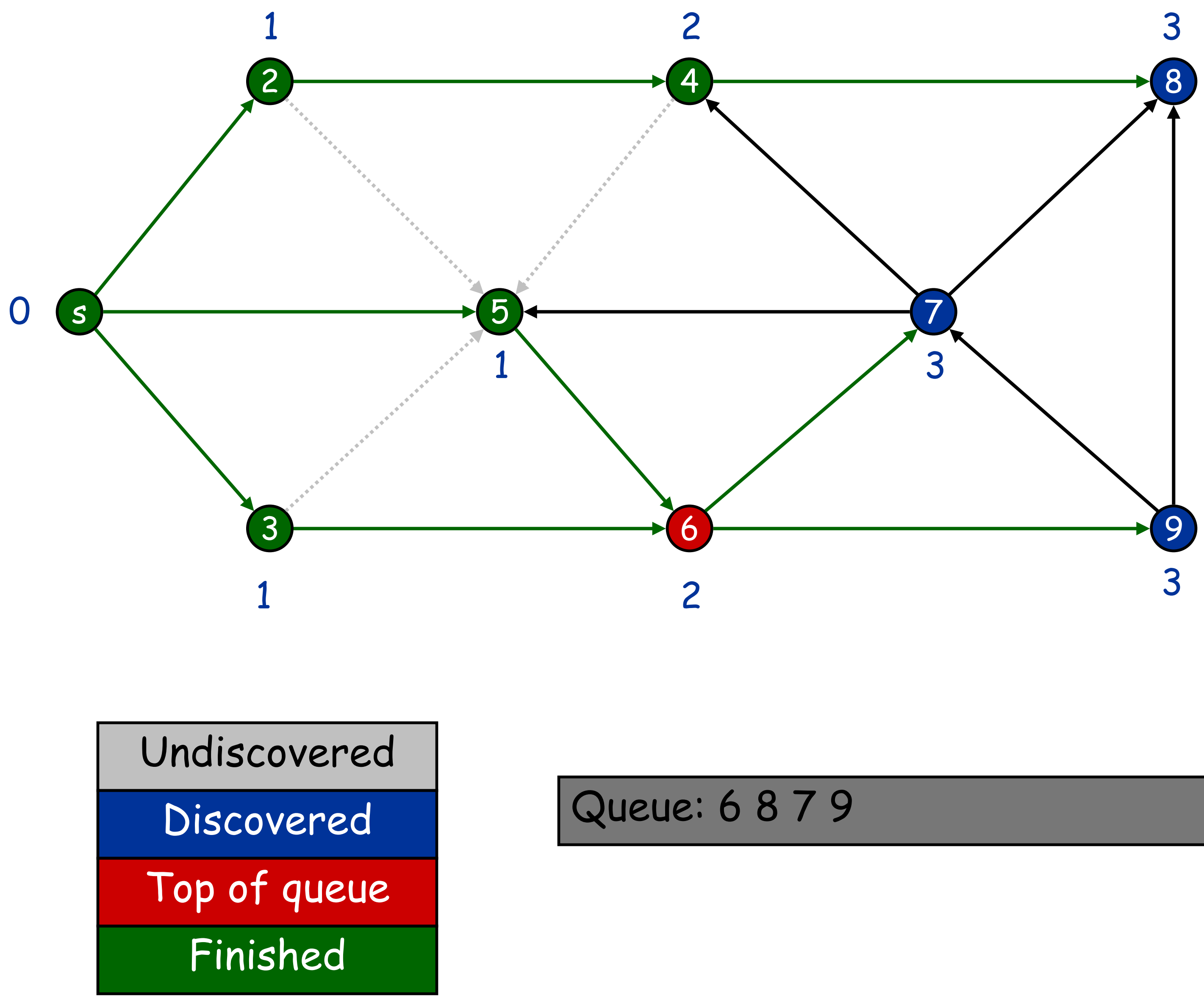

Queue: 6 8

# Breadth First Search



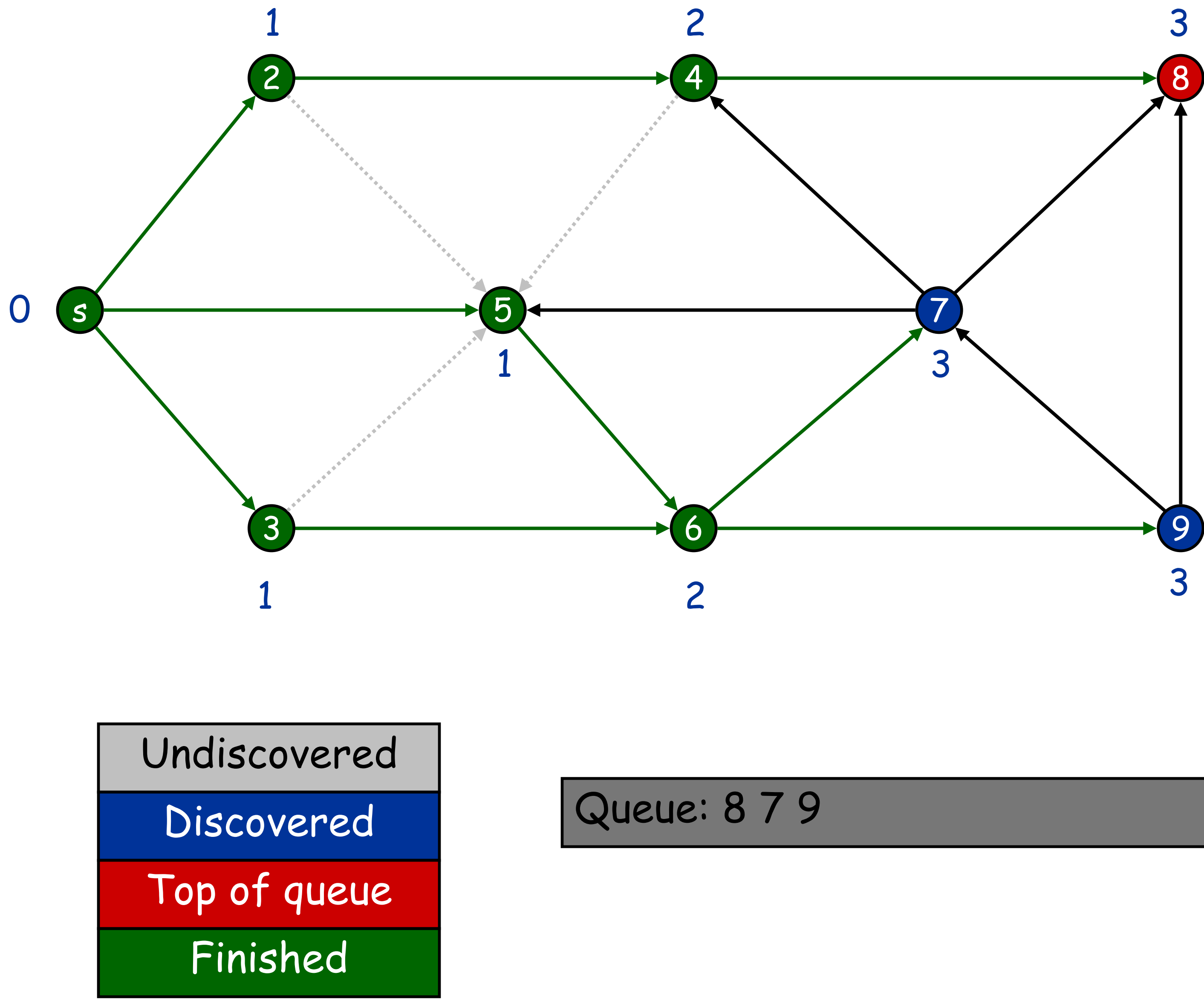

Queue: 6 8 7

# Breadth First Search



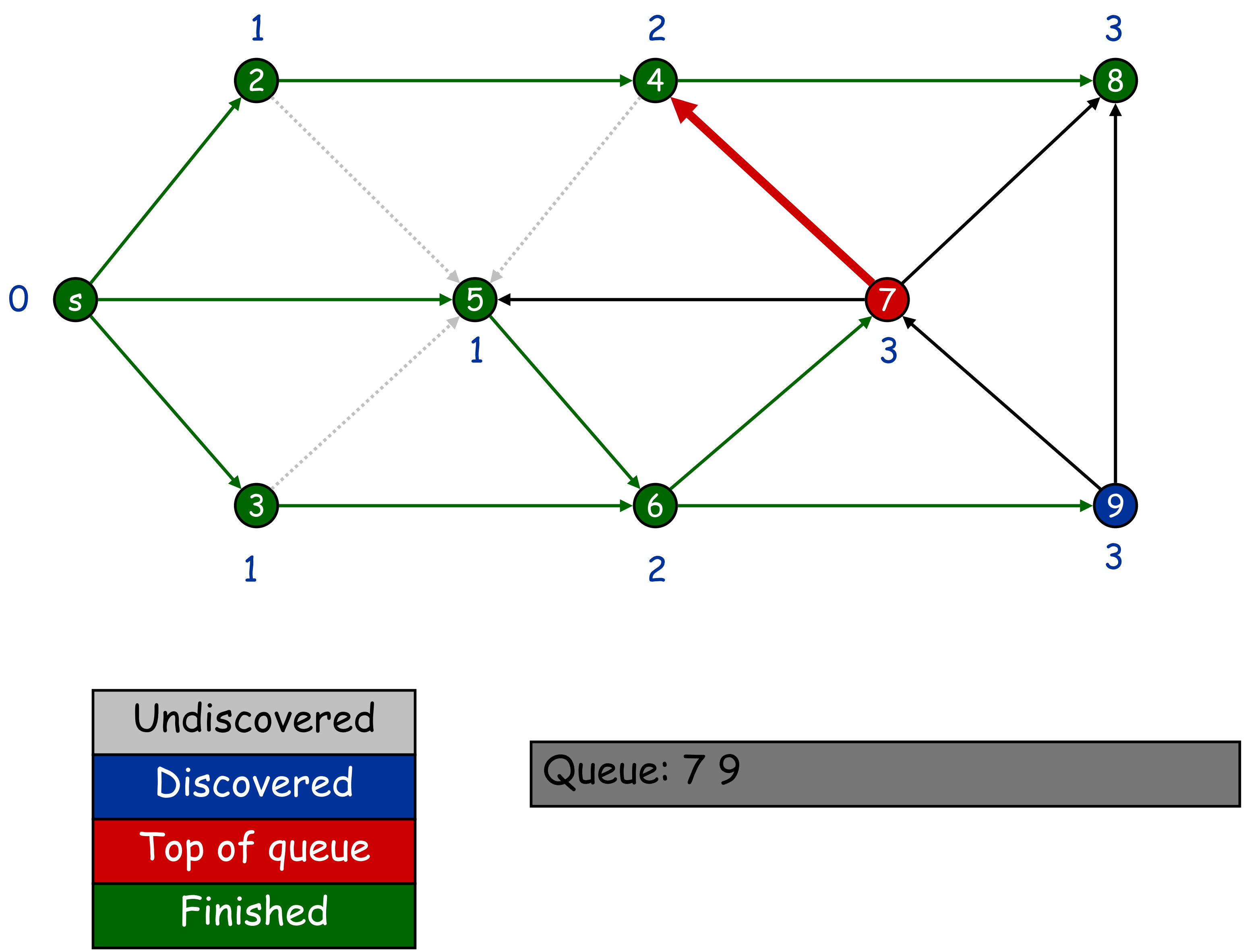

Queue: 6 8 7 9

# Breadth First Search

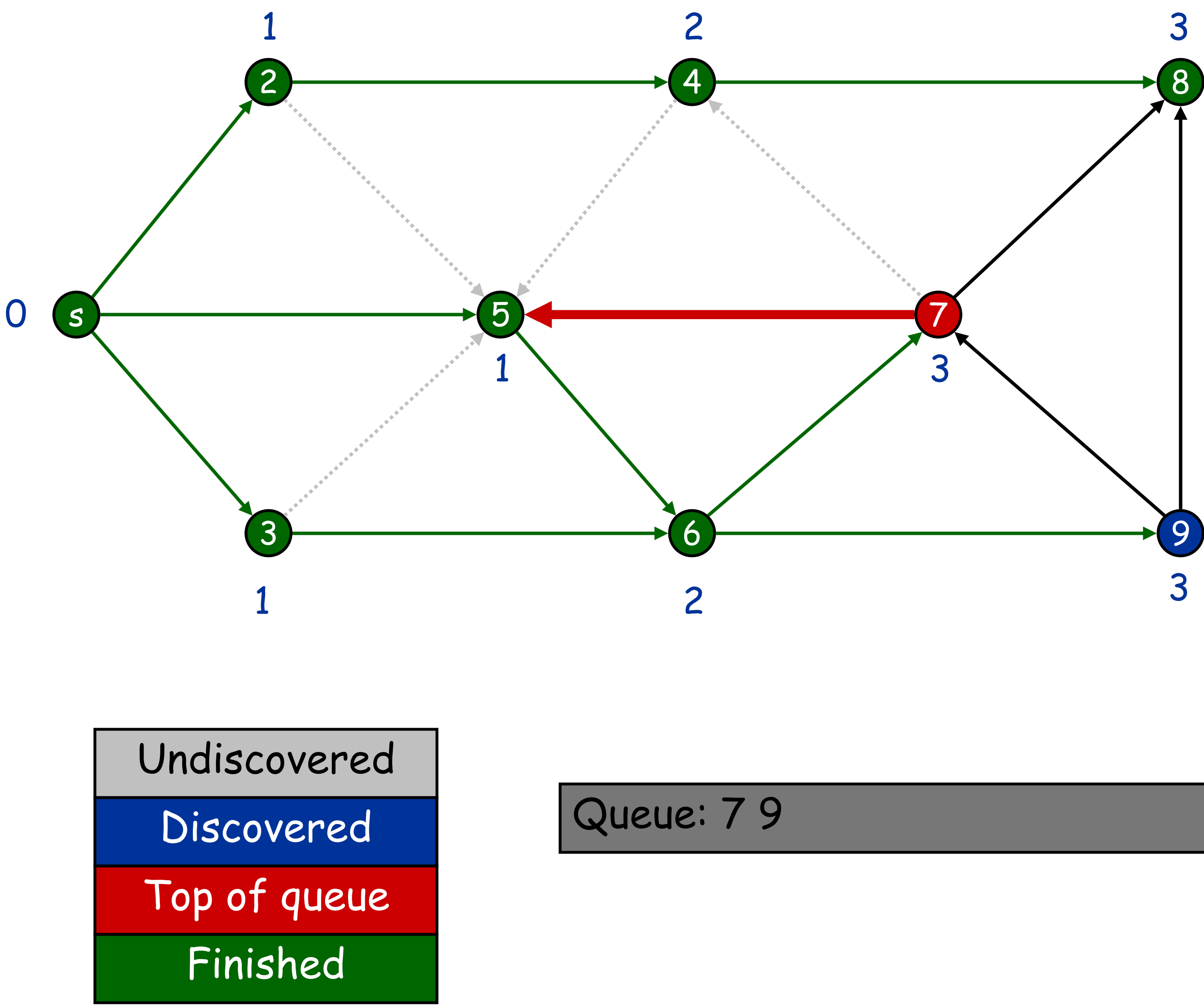



Queue: 8 7 9

# Breadth First Search

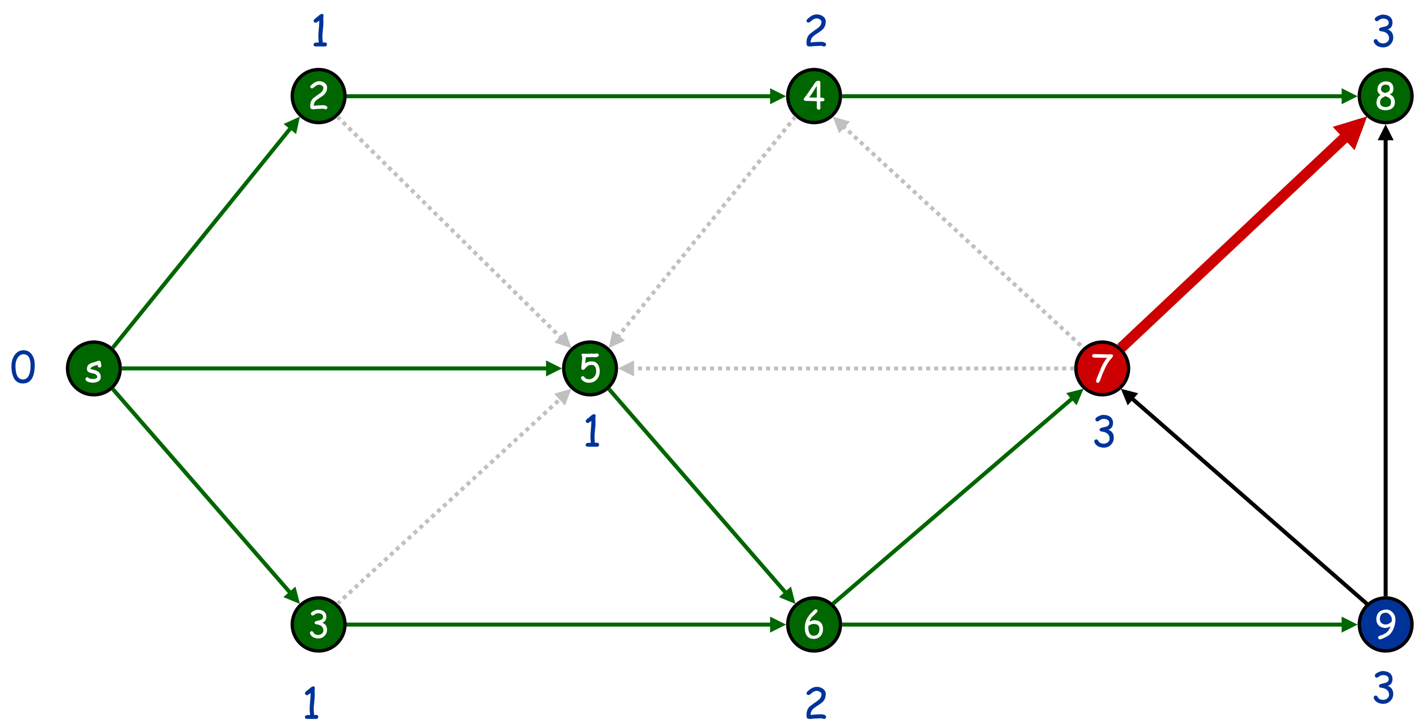


# Breadth First Search




Queue: 7 9

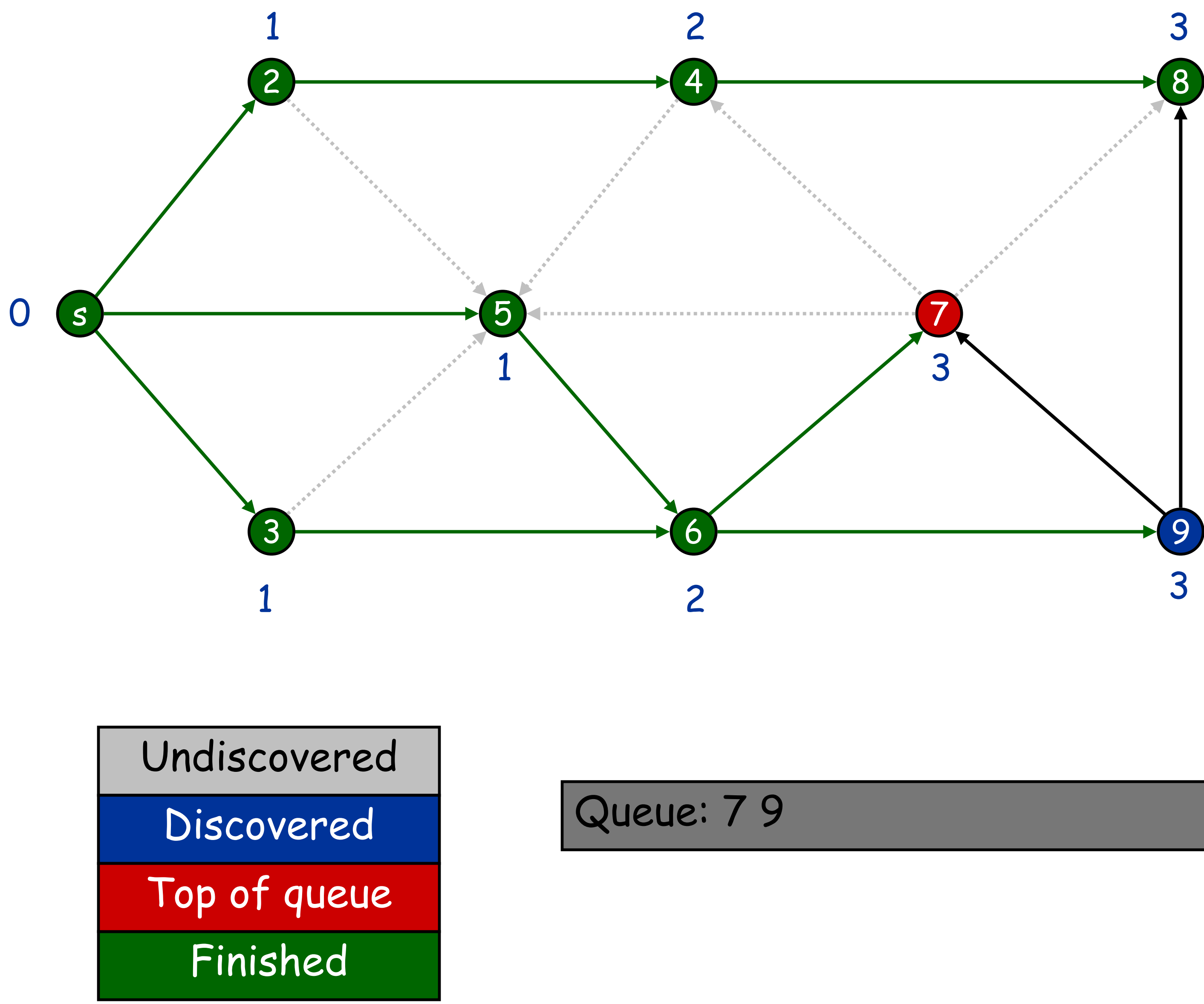
# Breadth First Search




| |
|---|
| Undiscovered |
| Discovered |
| Top of queue |
| Finished |

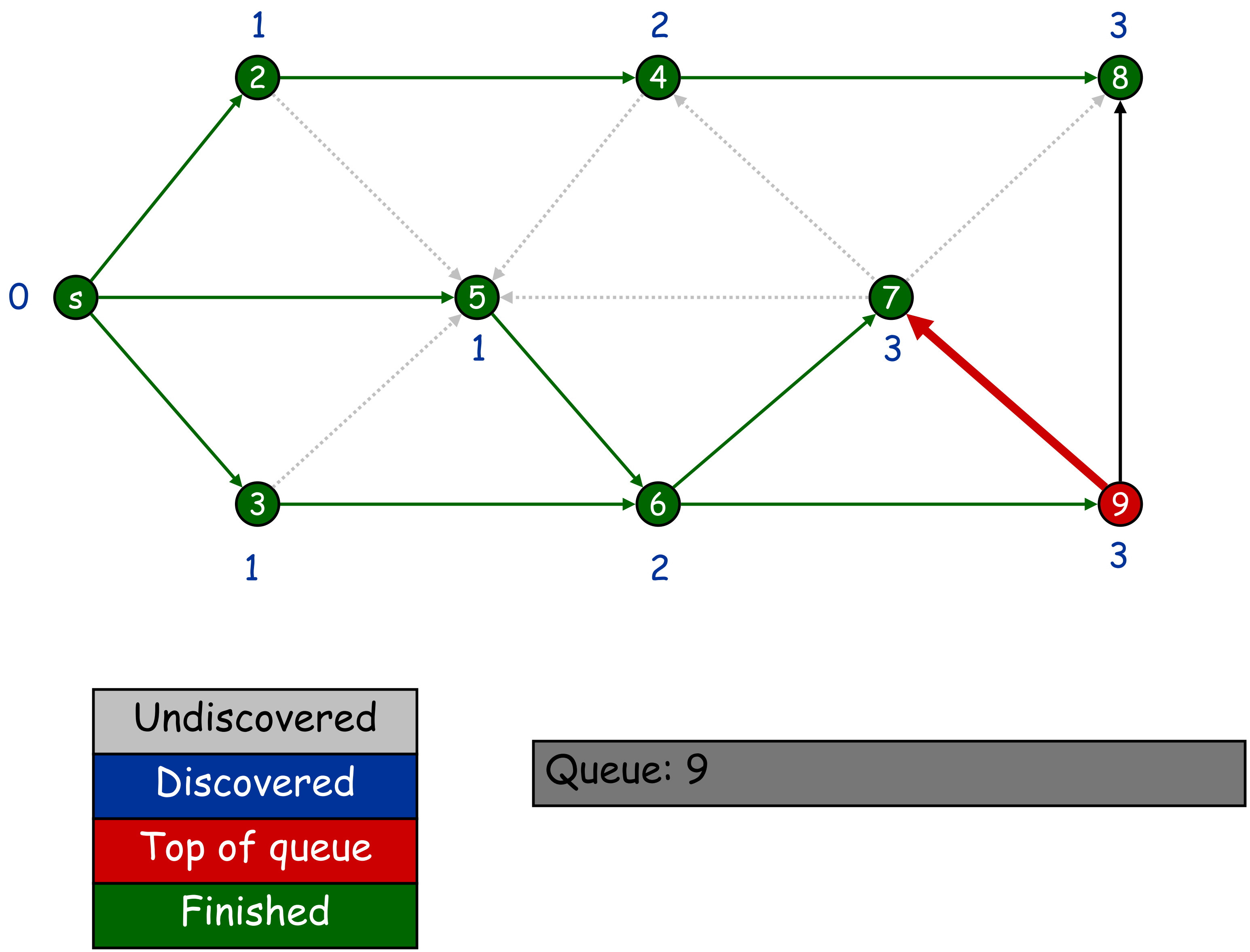
Queue: 7 9

# Breadth First Search

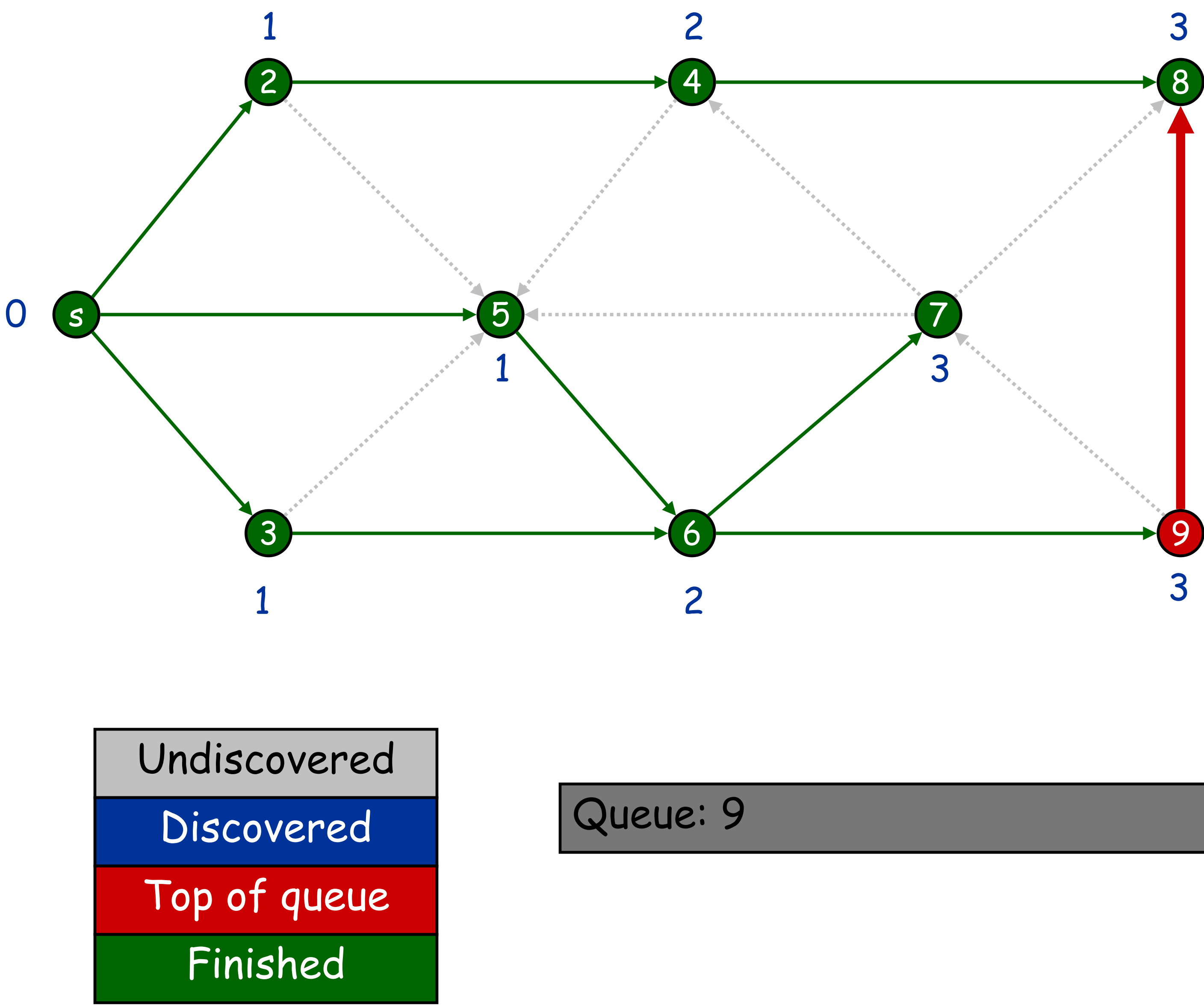



Queue: 7 9

# Breadth First Search

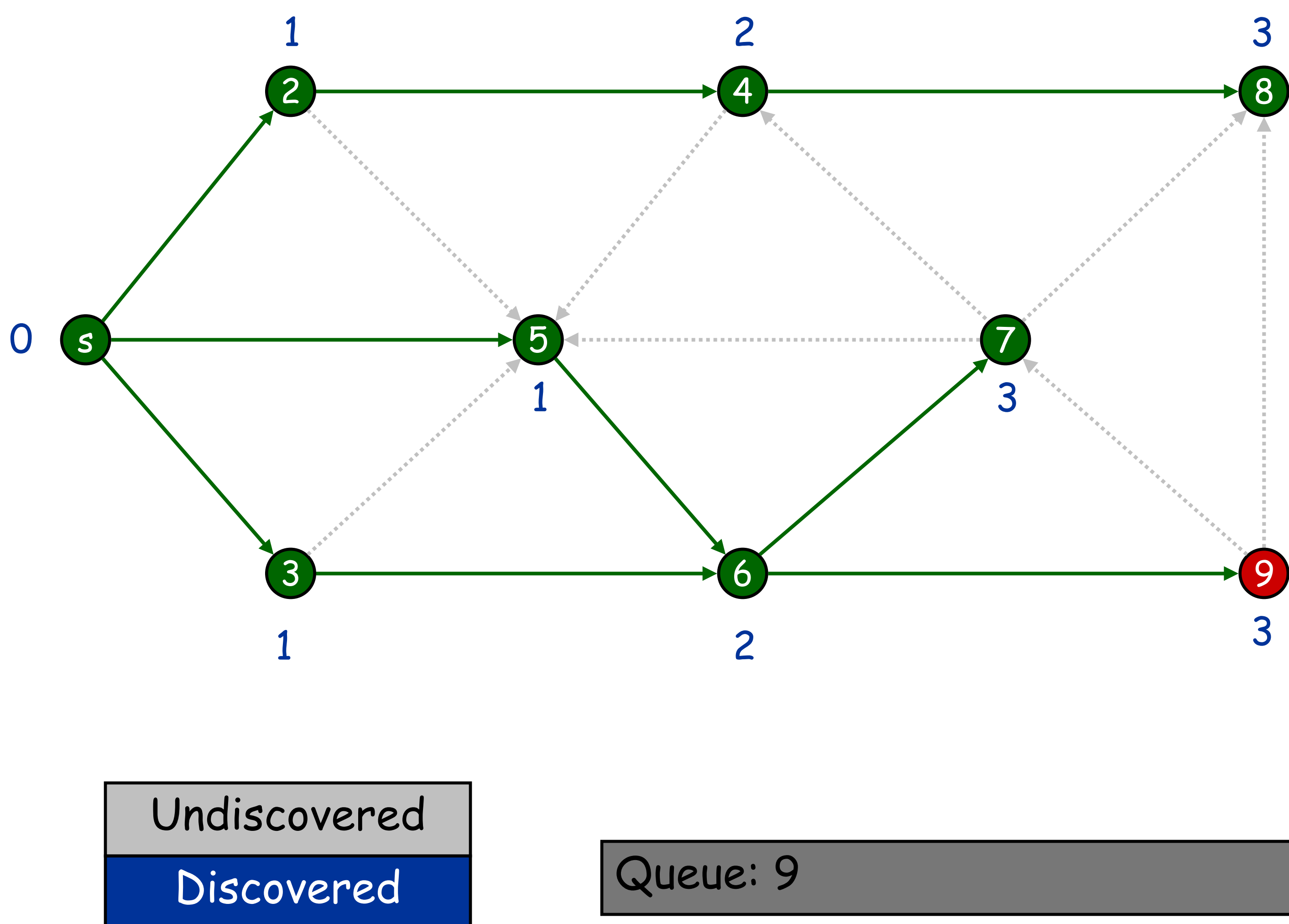


# Breadth First Search

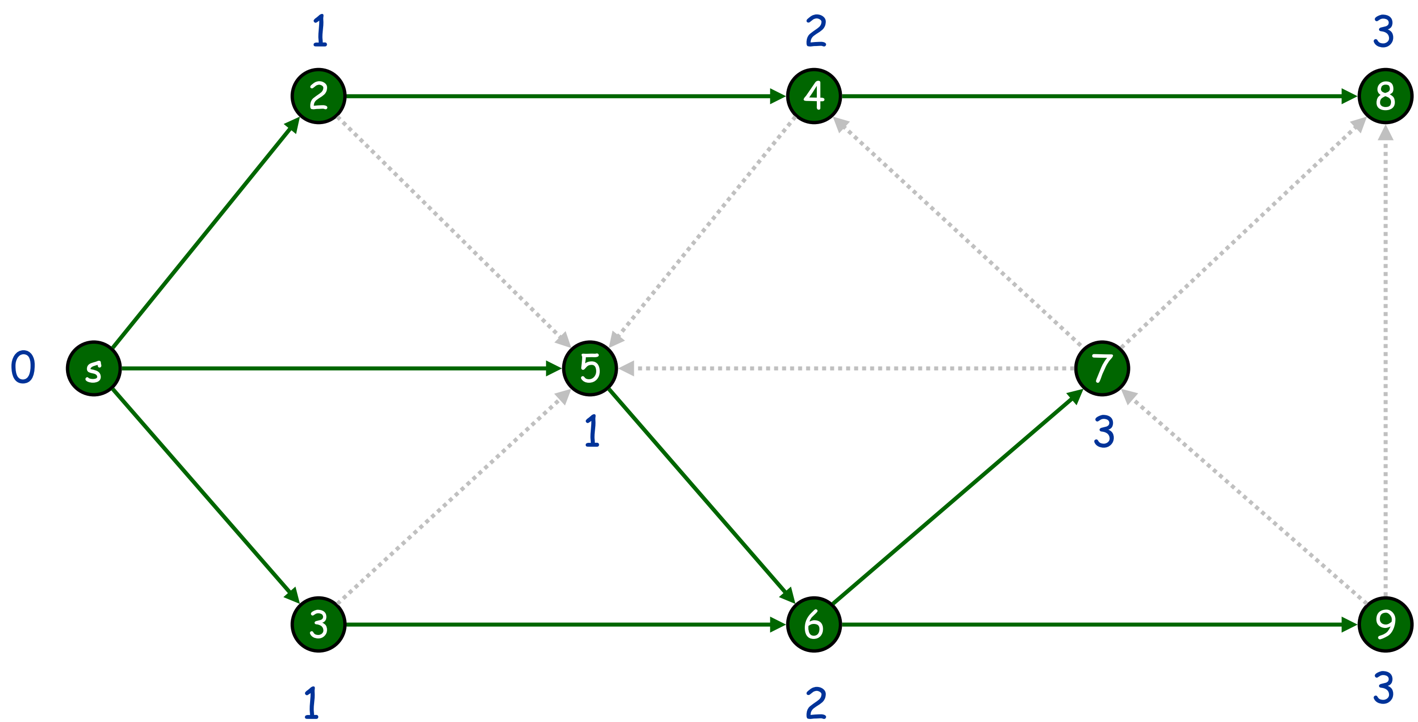



Queue: 9

# Breadth First Search



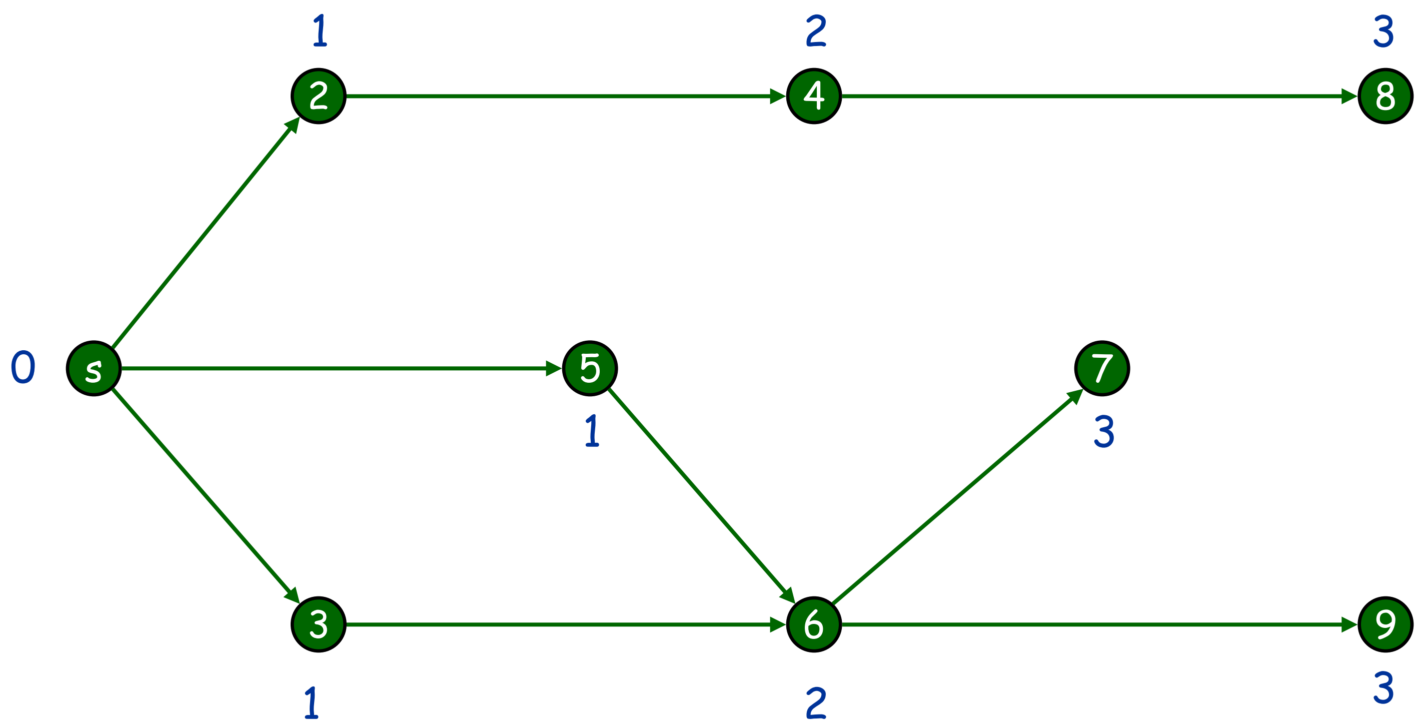
# Breadth First Search





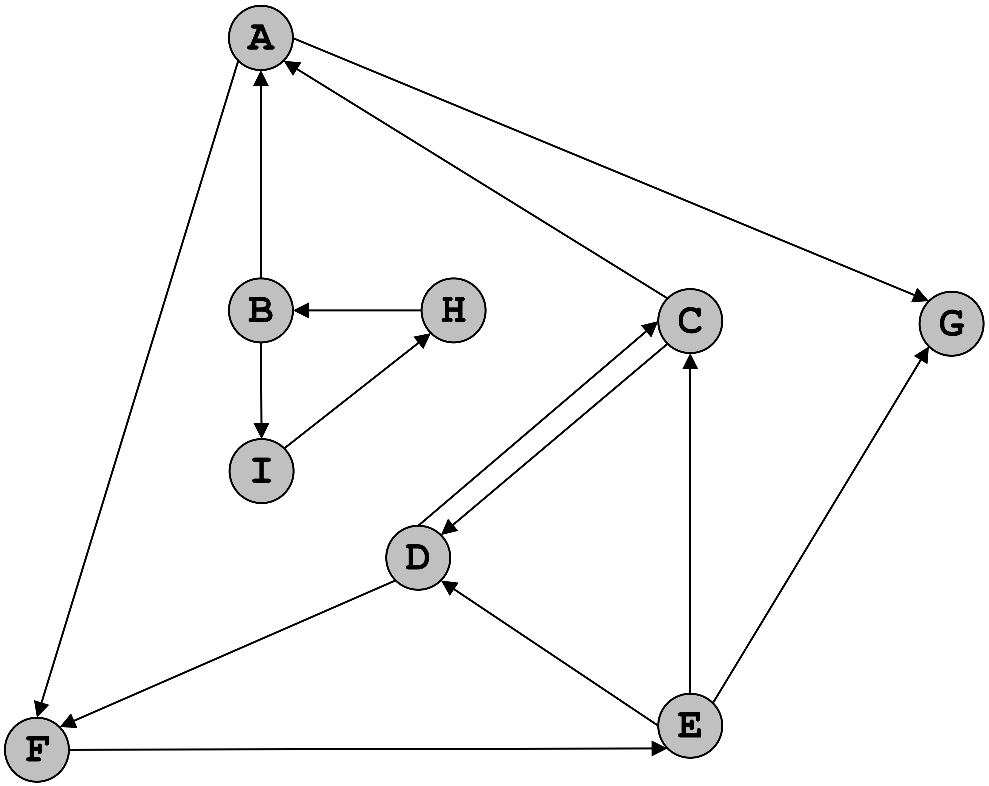
Queue:

# Breadth First Search




Level Graph

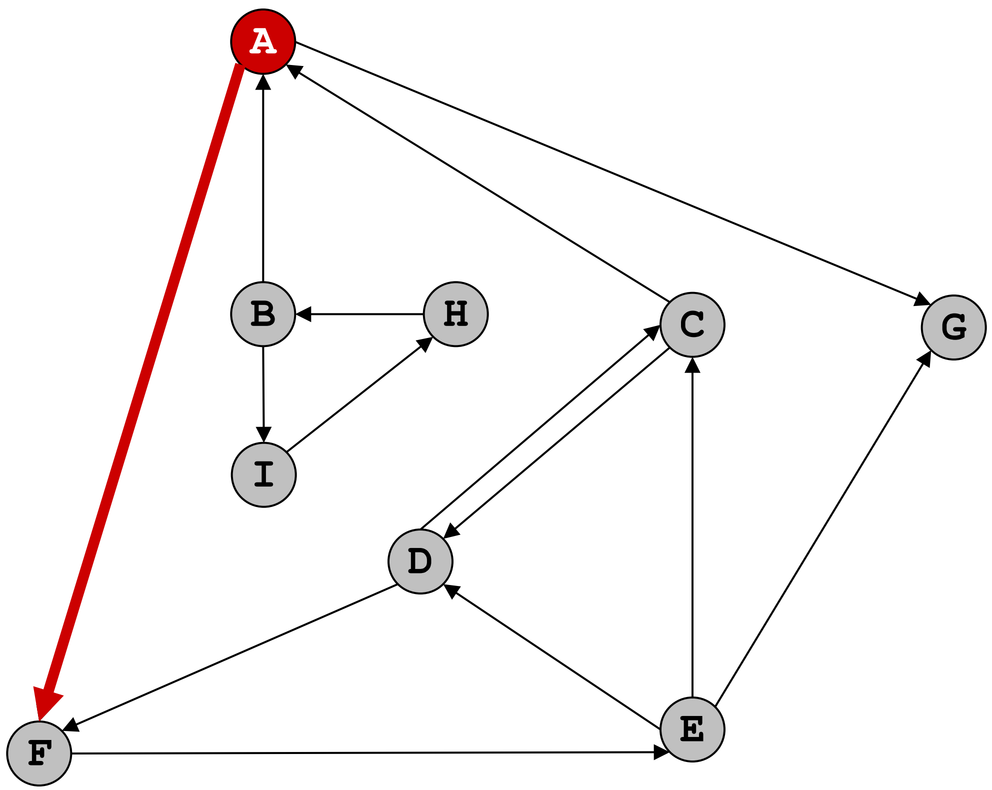
# Question 2: Directed Depth First Search



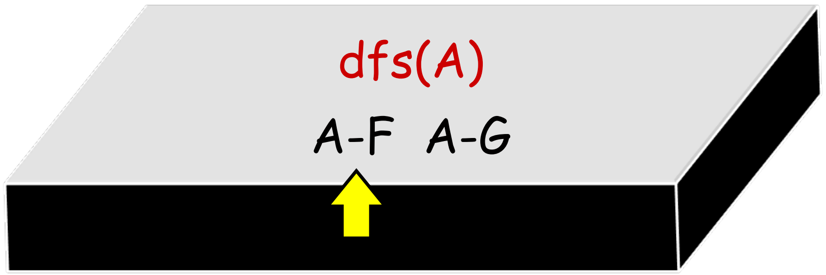

Adjacency Lists

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

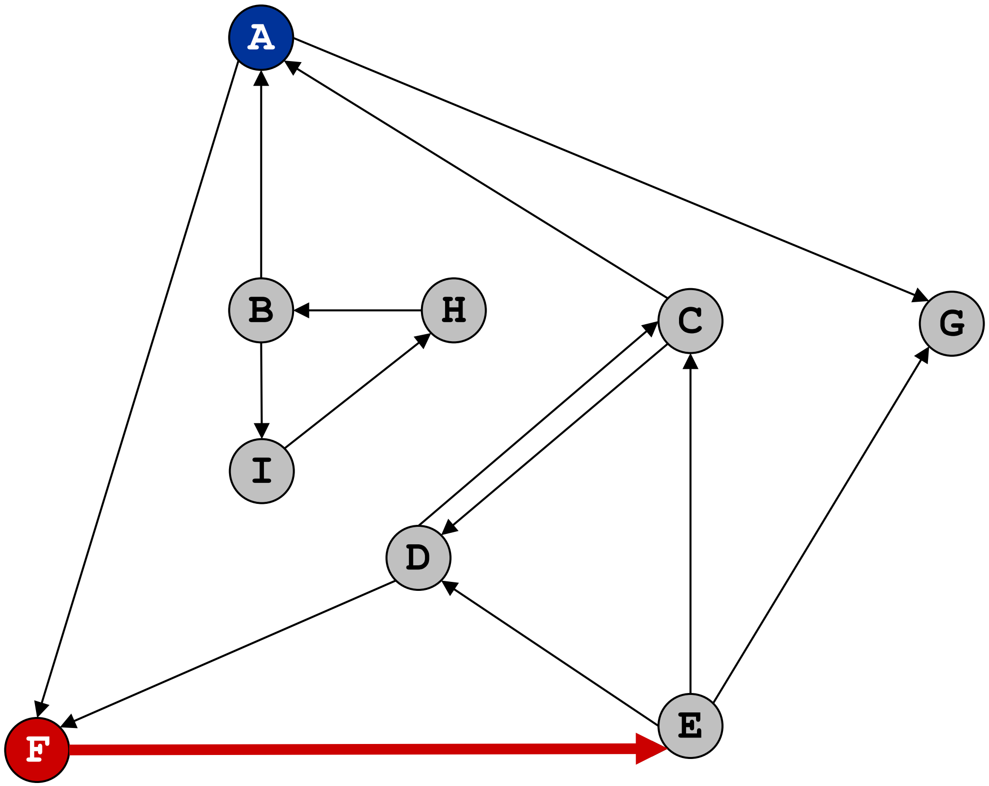
# Directed Depth First Search



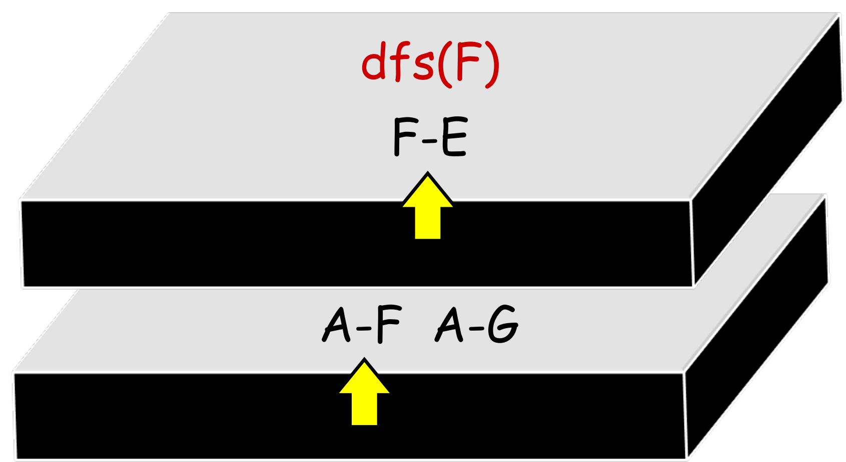

Function call stack:



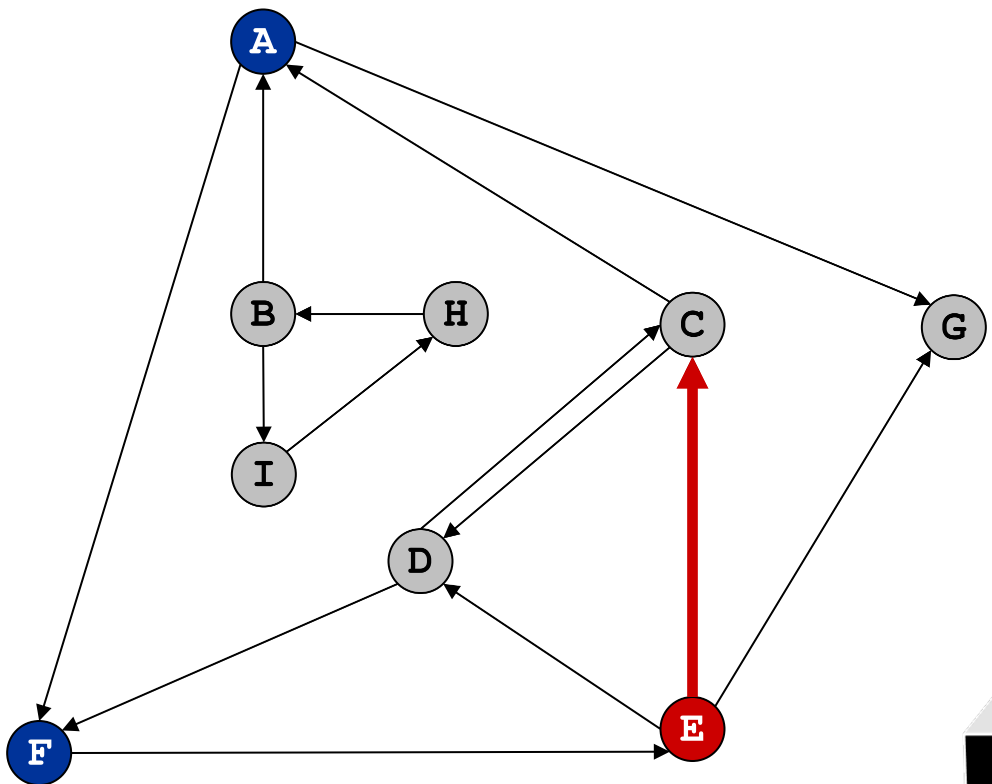
# Directed Depth First Search



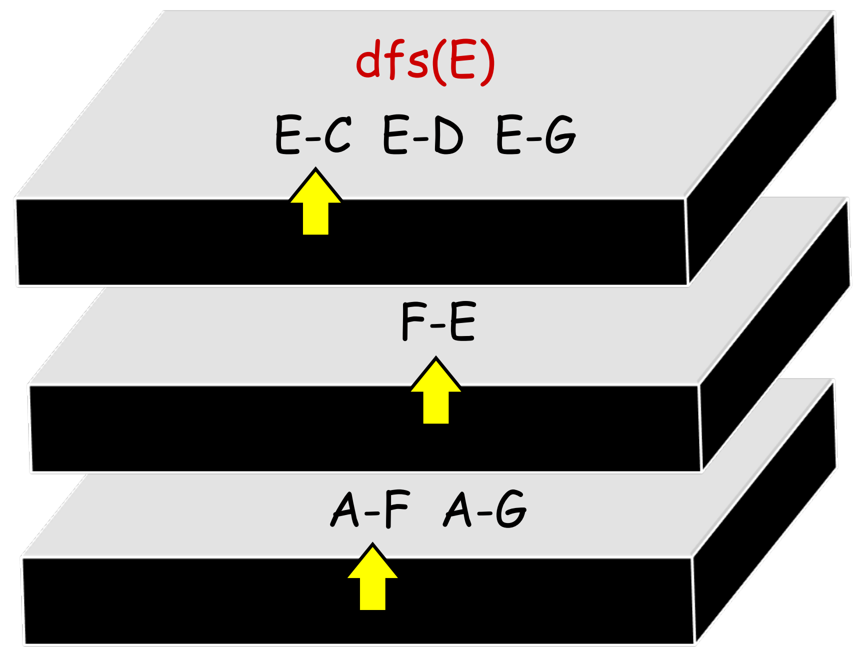

Function call stack:



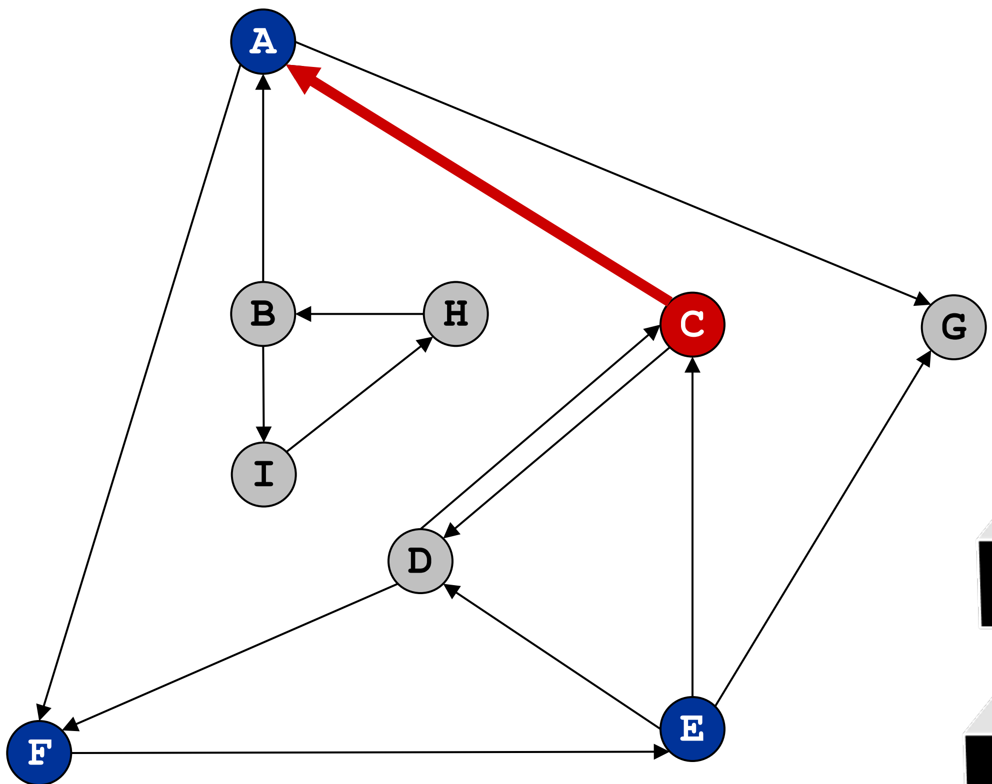
# Directed Depth First Search



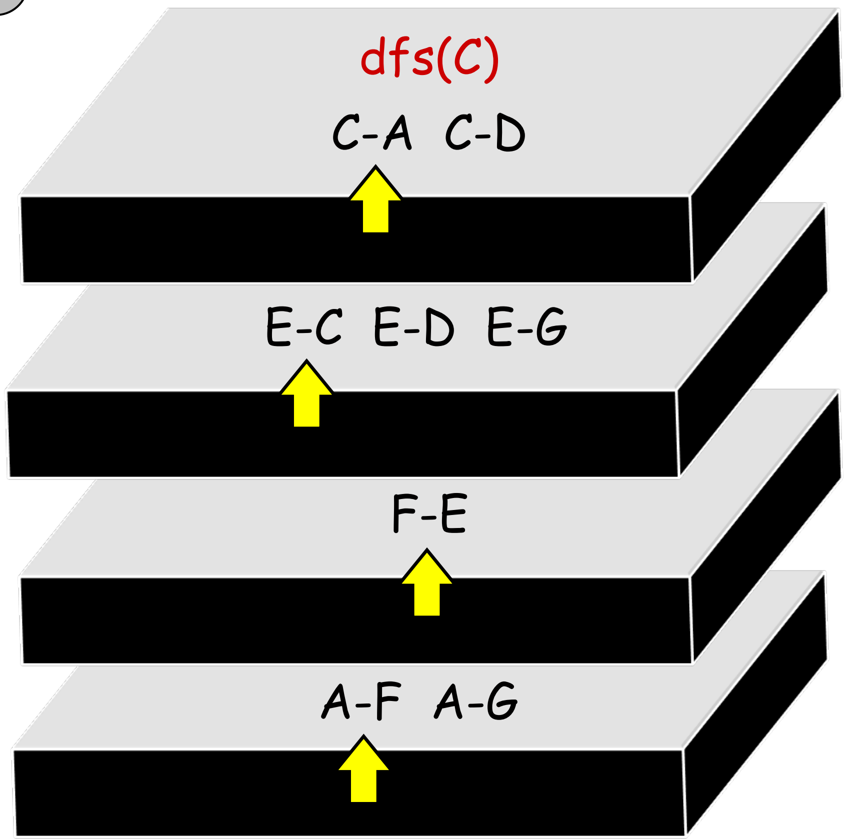

Function call stack:



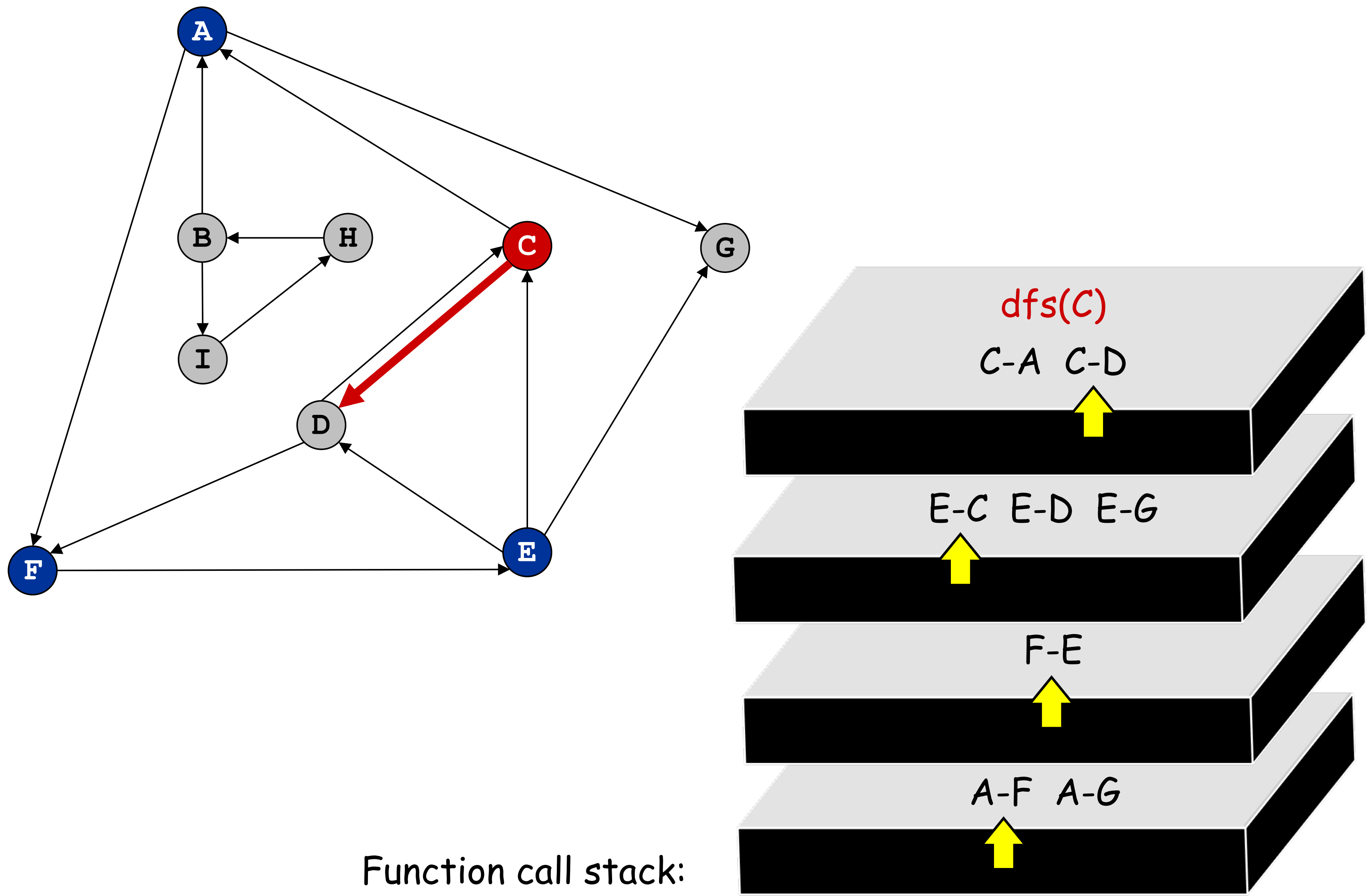
# Directed Depth First Search



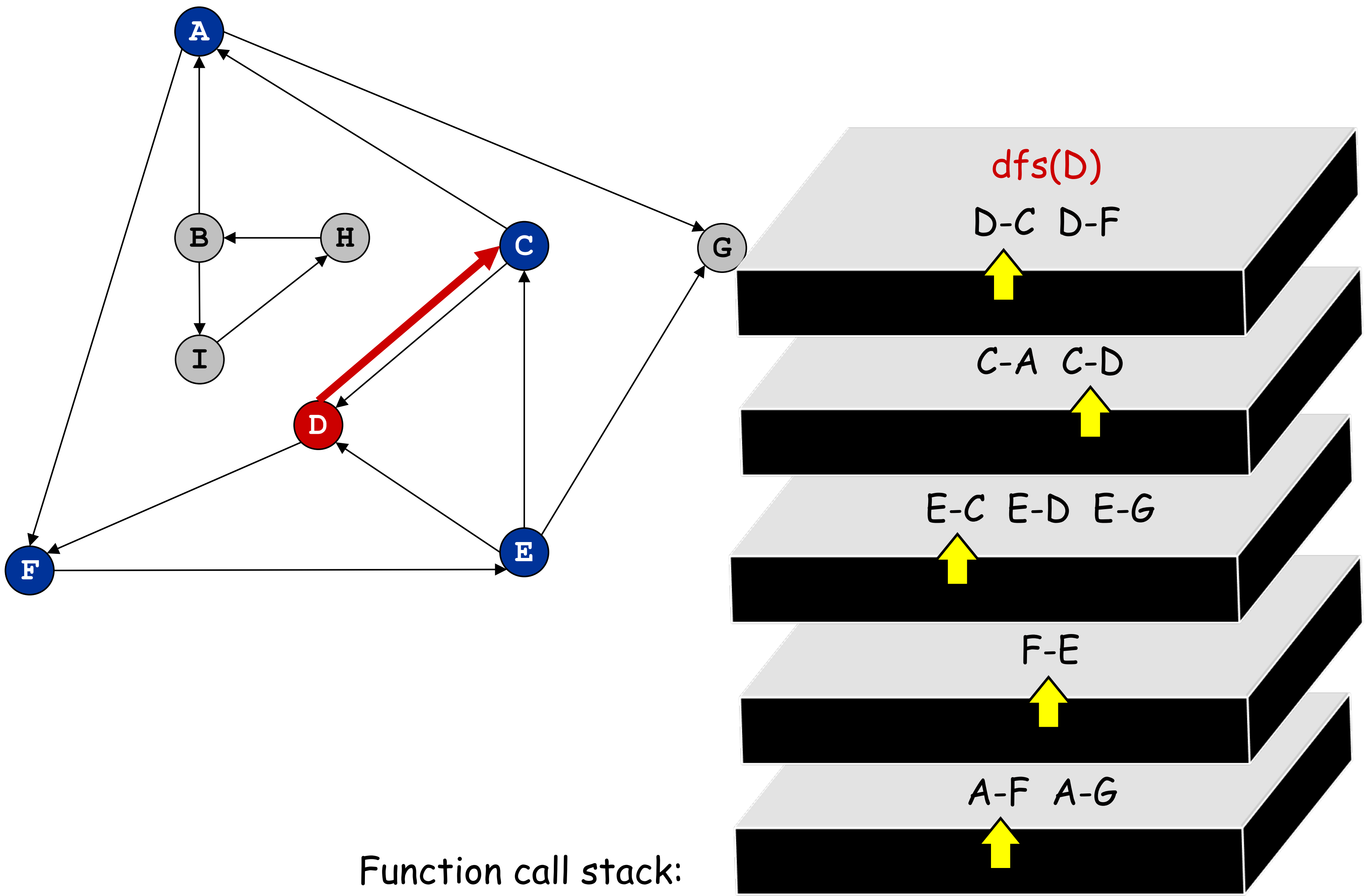

Function call stack:



# Directed Depth First Search

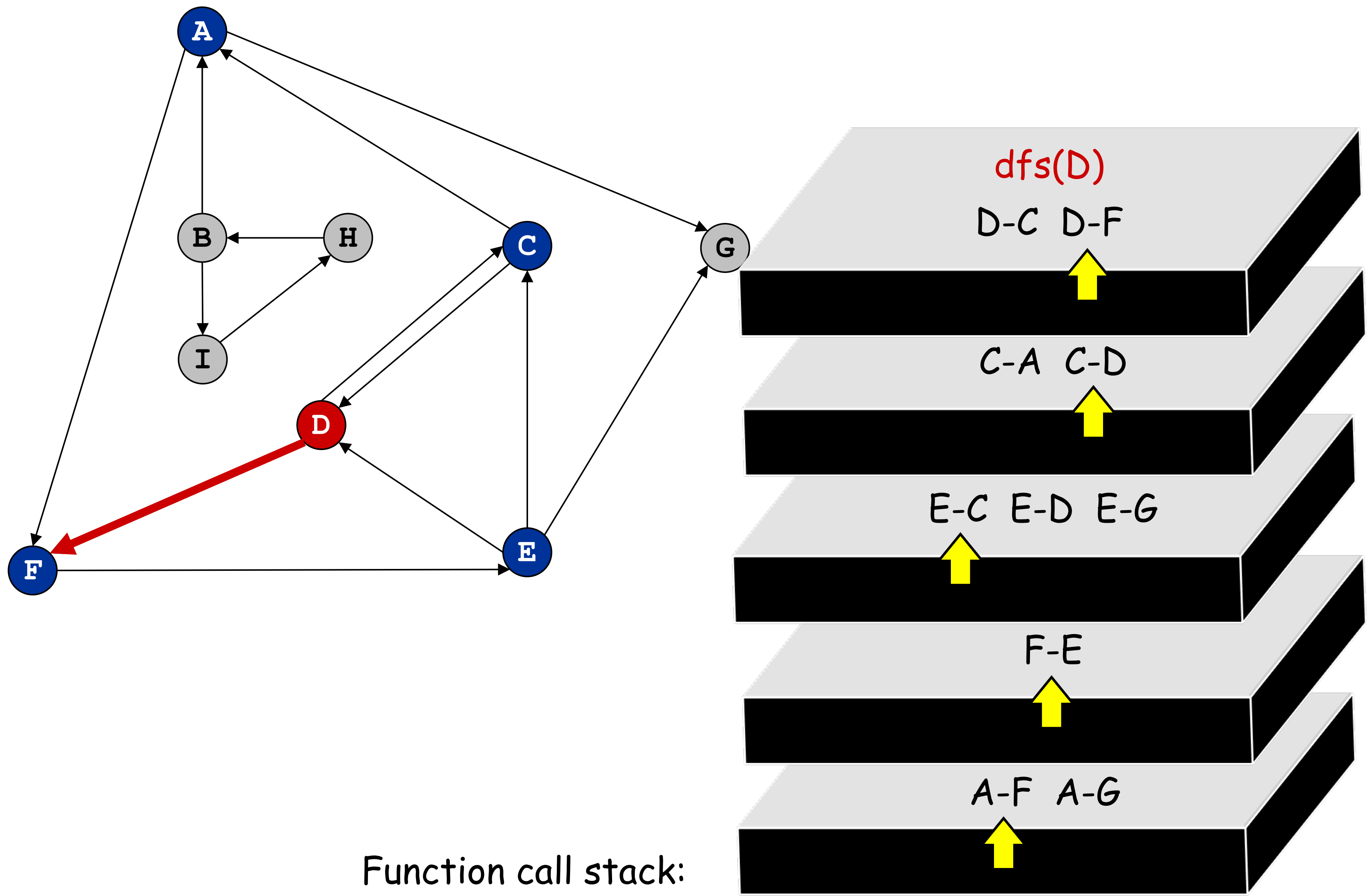


# Directed Depth First Search

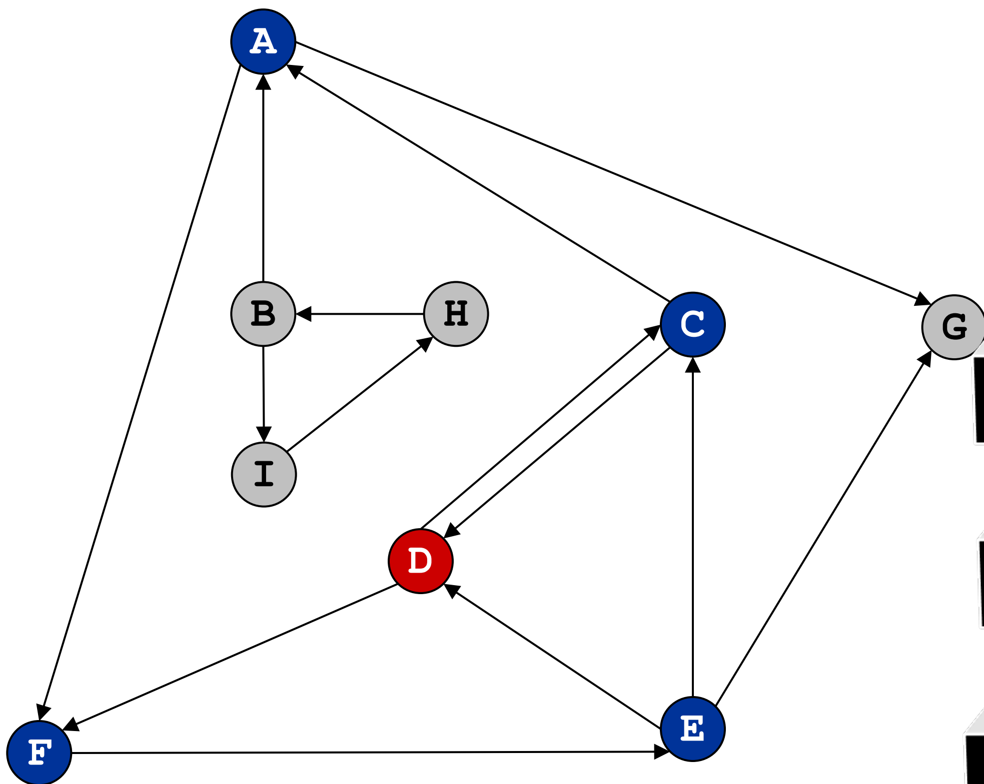


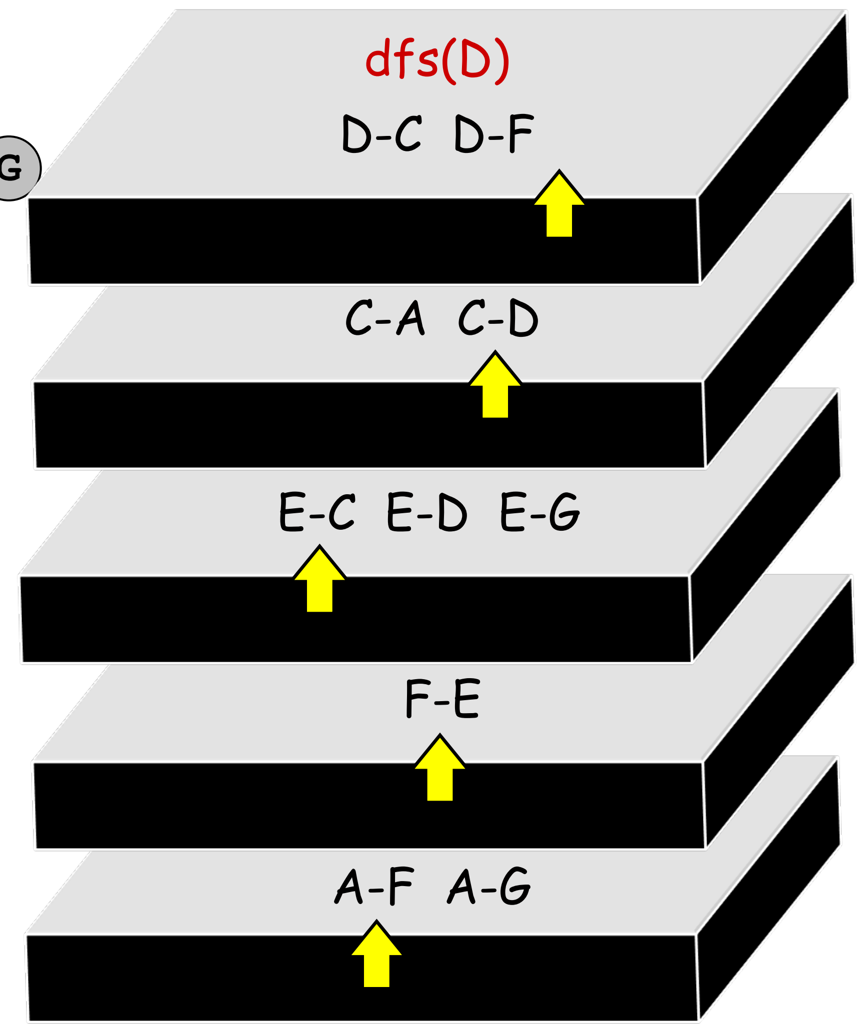

Function call stack:

# Directed Depth First Search



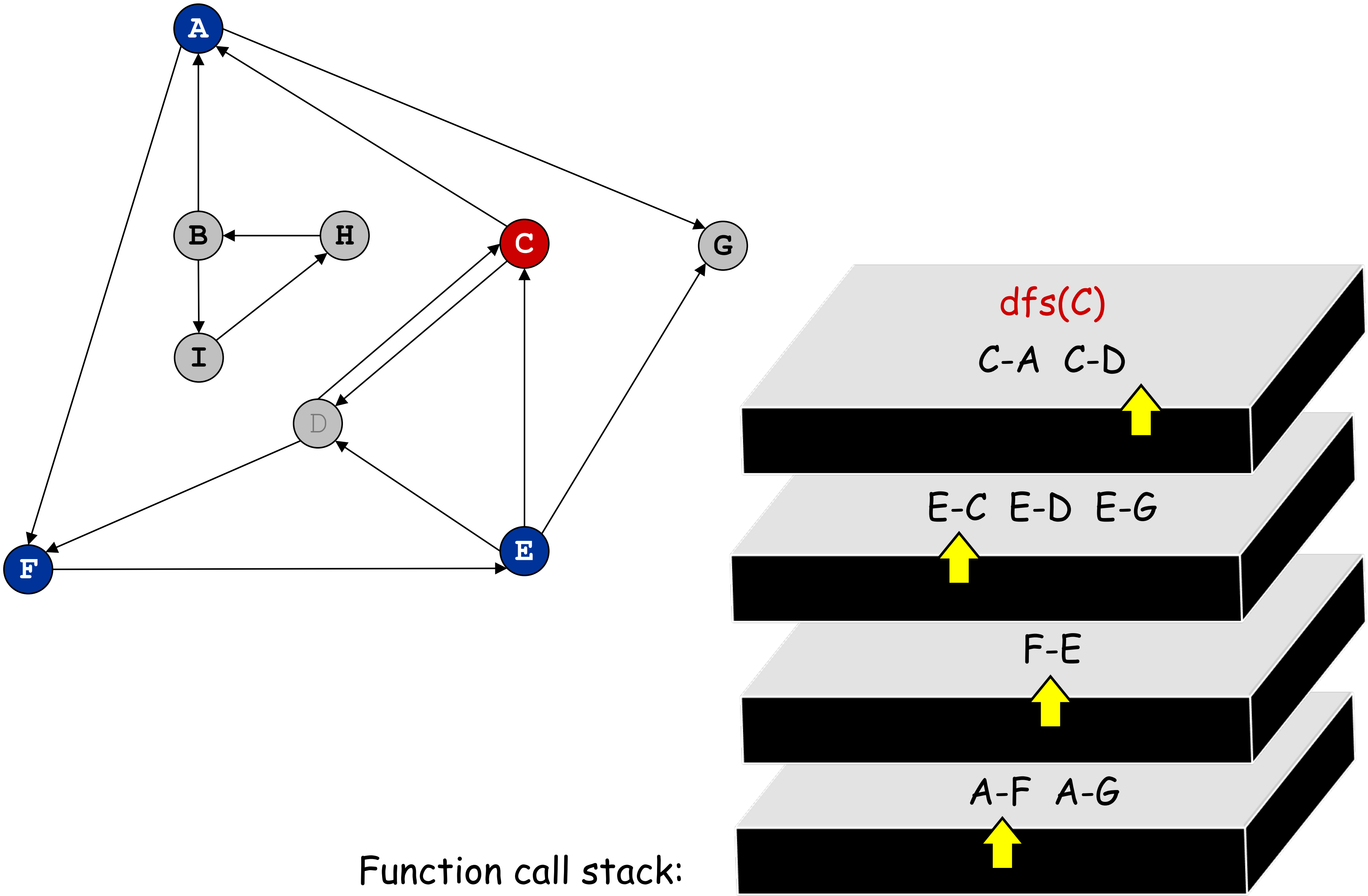
# Directed Depth First Search



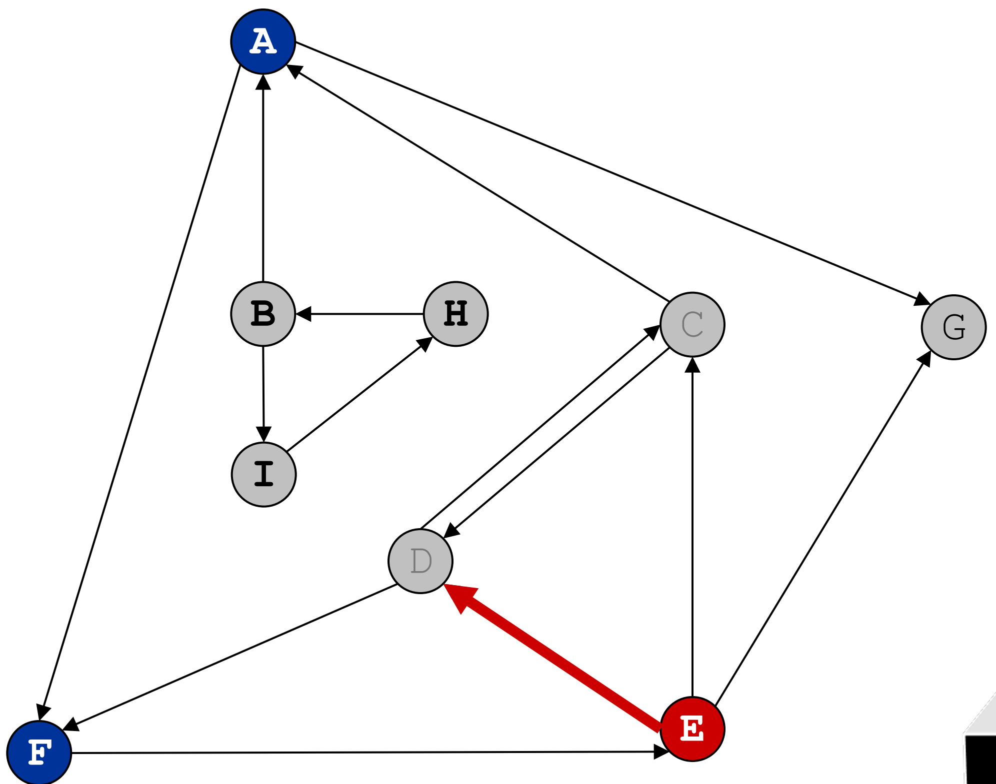




Function call stack:

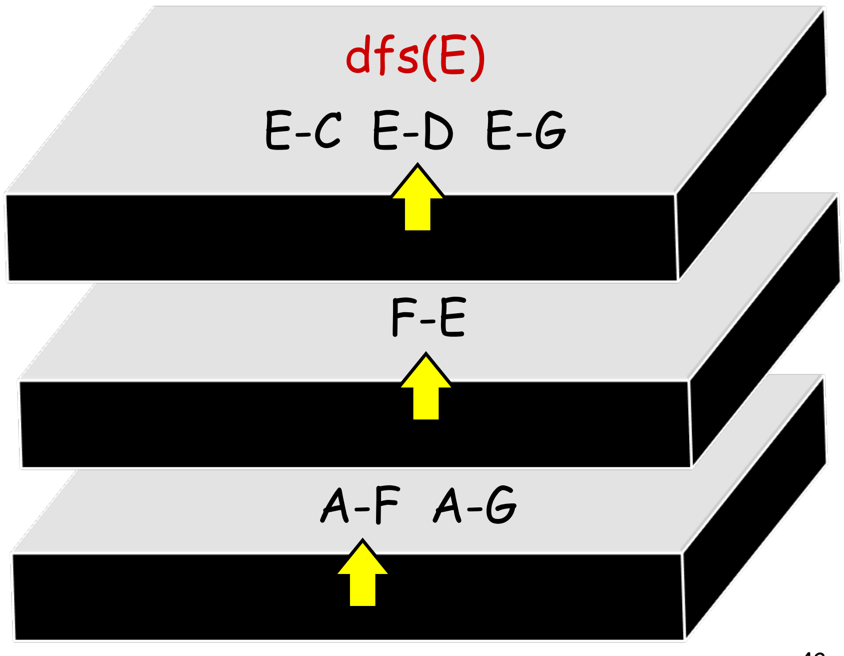
# Directed Depth First Search



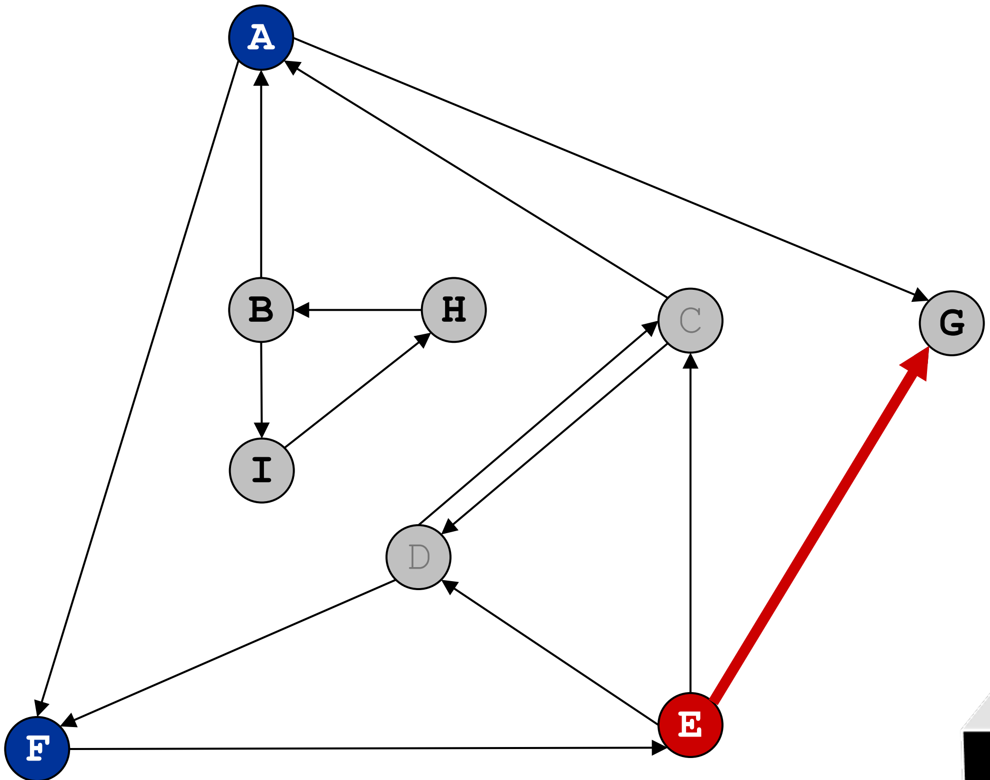
# Directed Depth First Search



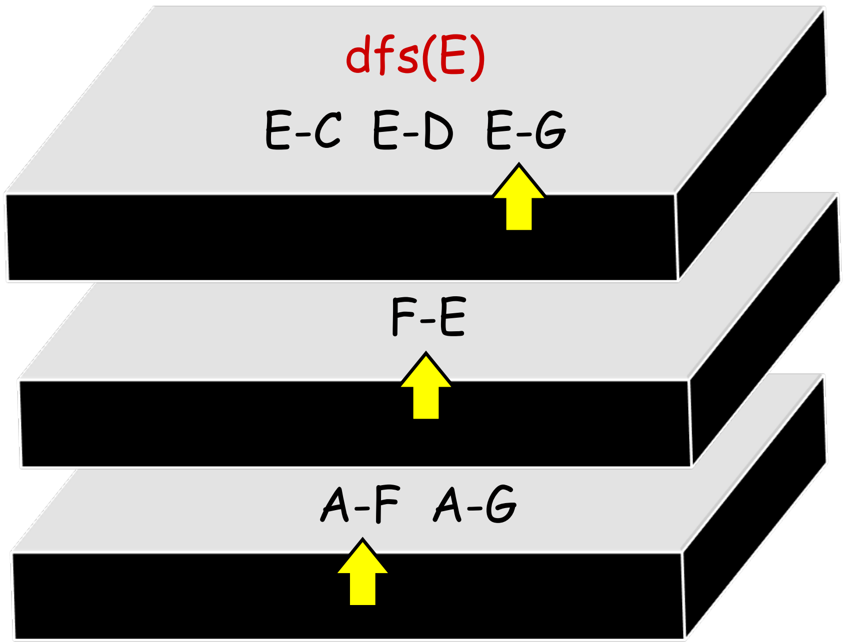

Function call stack:



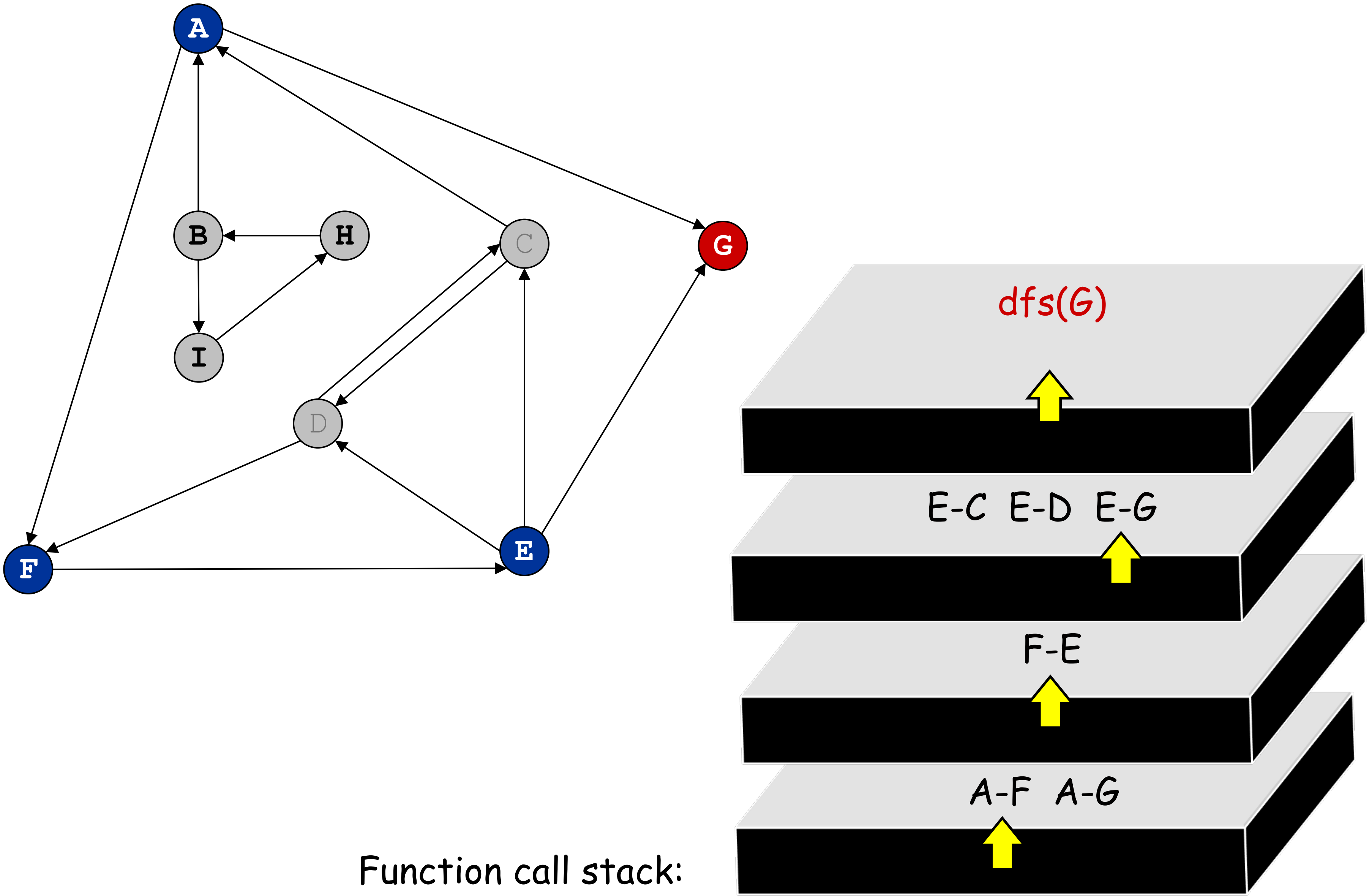
# Directed Depth First Search




Function call stack:

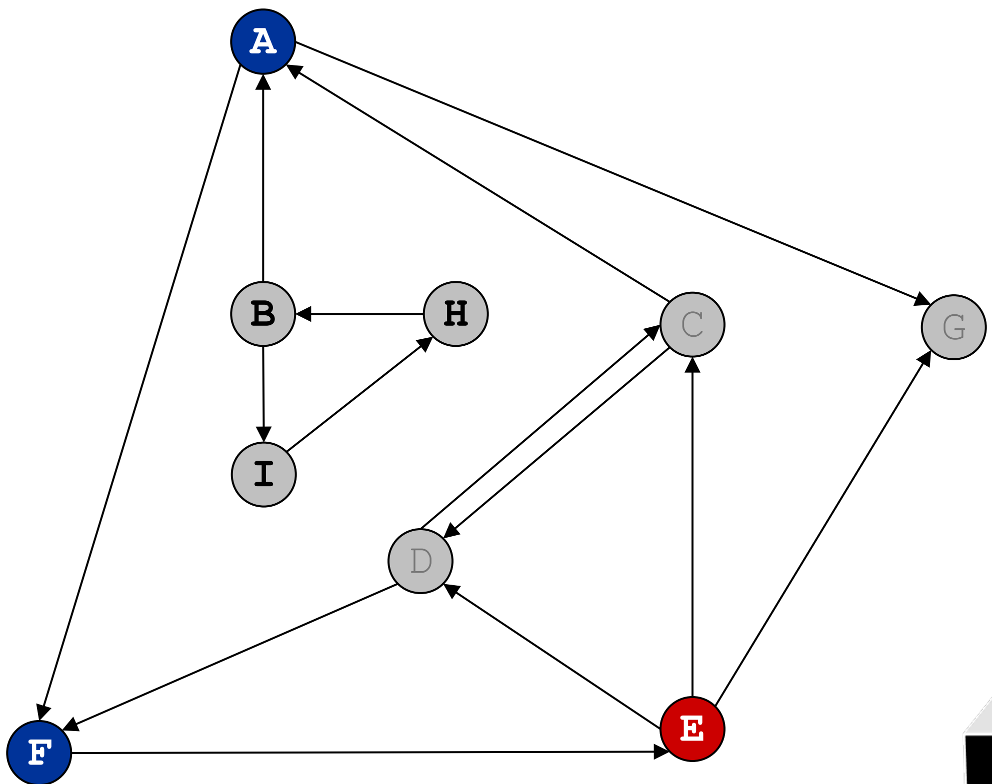


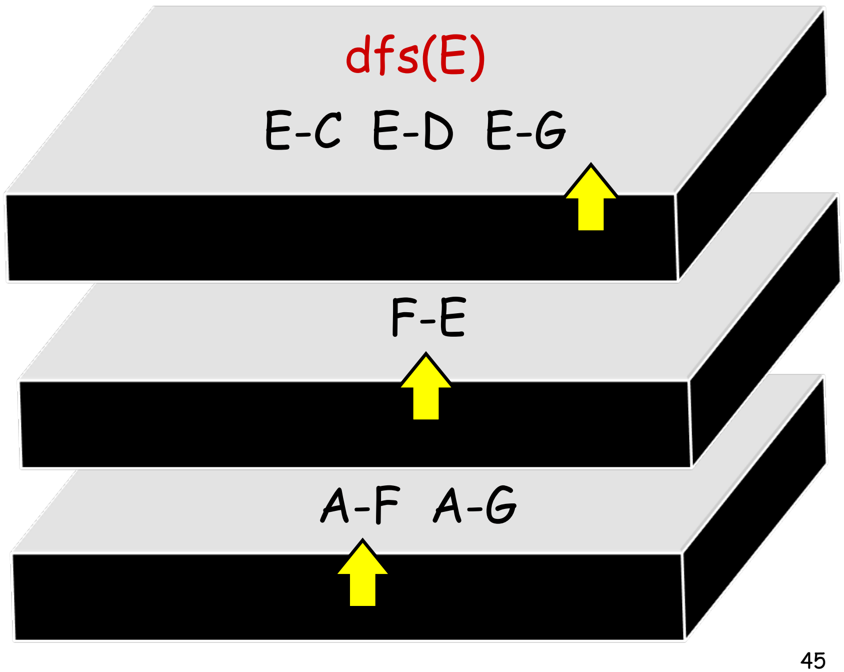
# Directed Depth First Search




Function call stack:

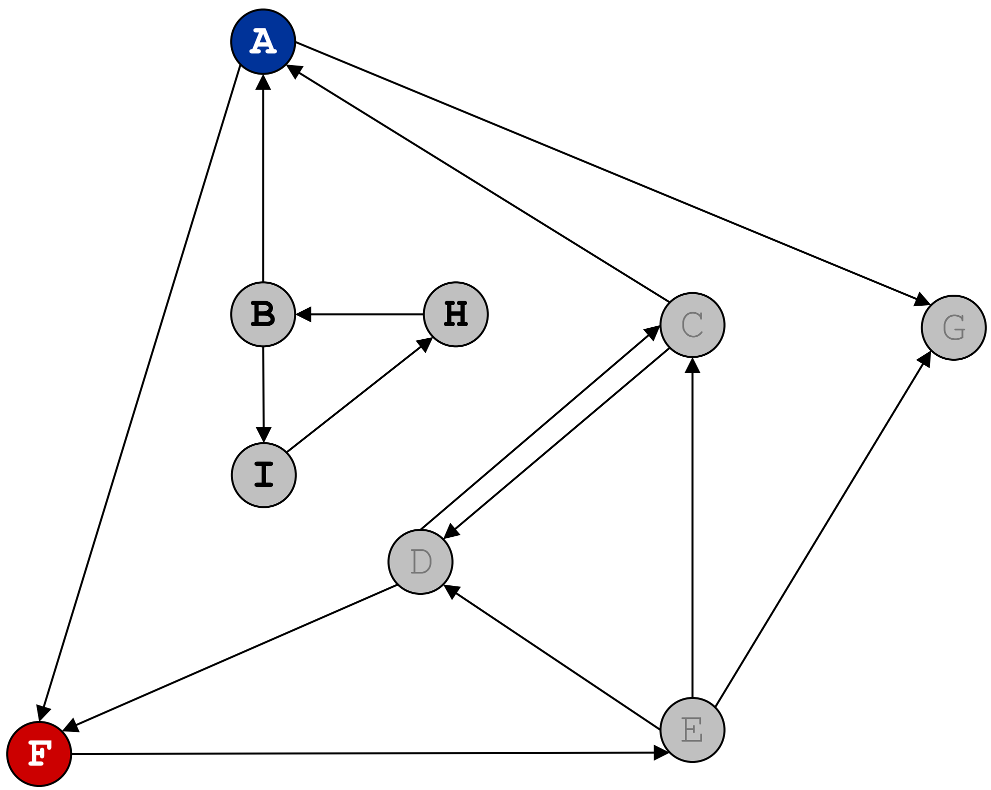
# Directed Depth First Search



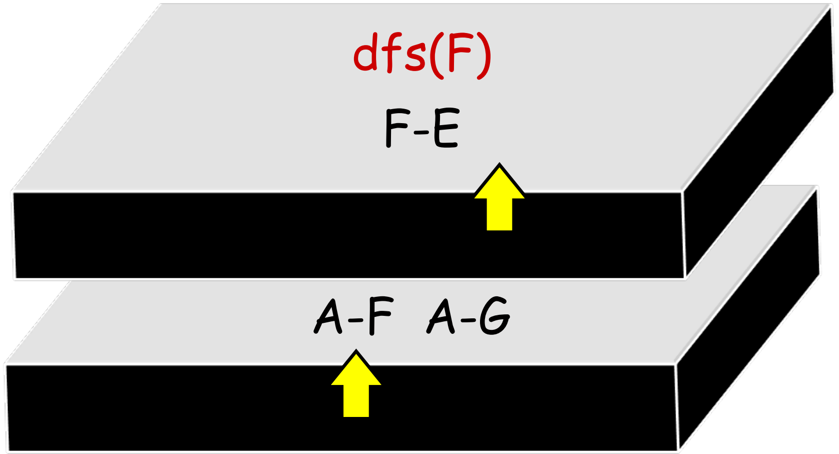




Function call stack:

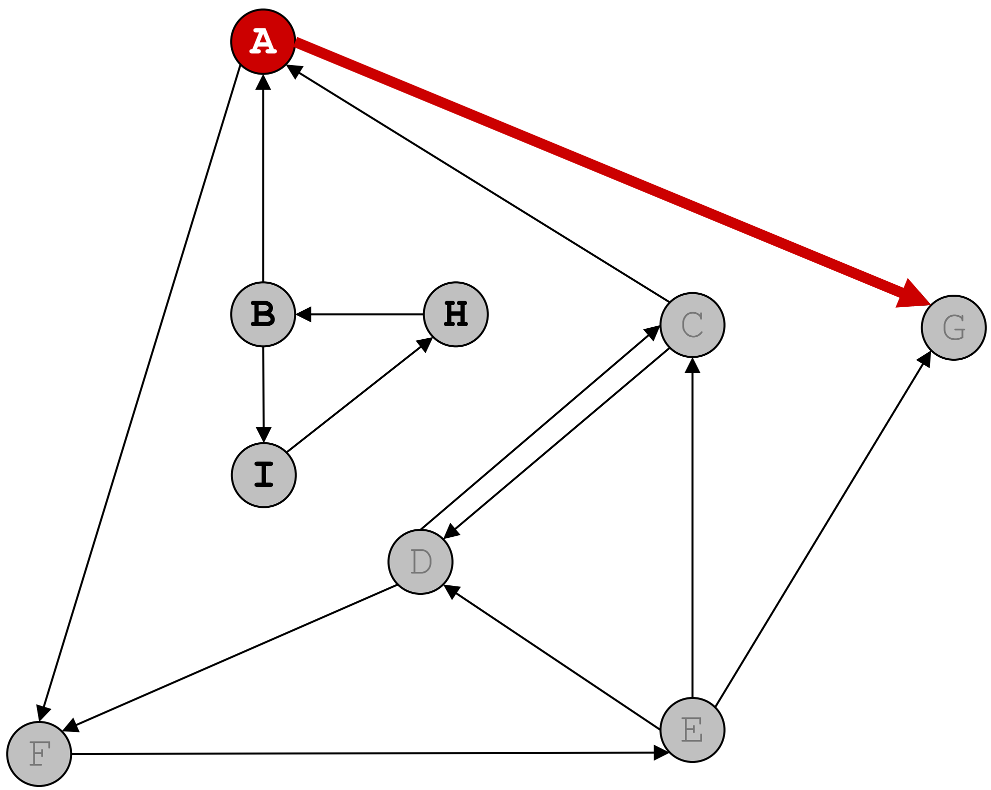
# Directed Depth First Search



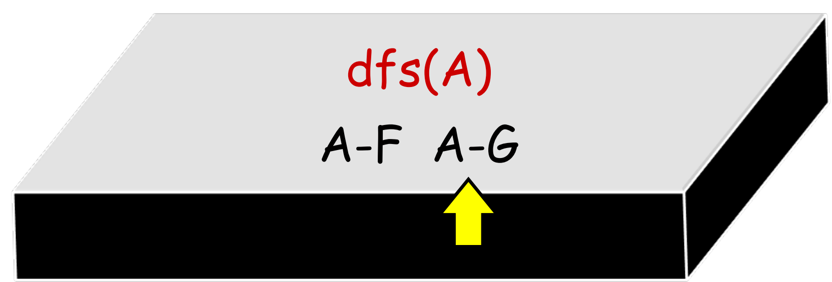

Function call stack:



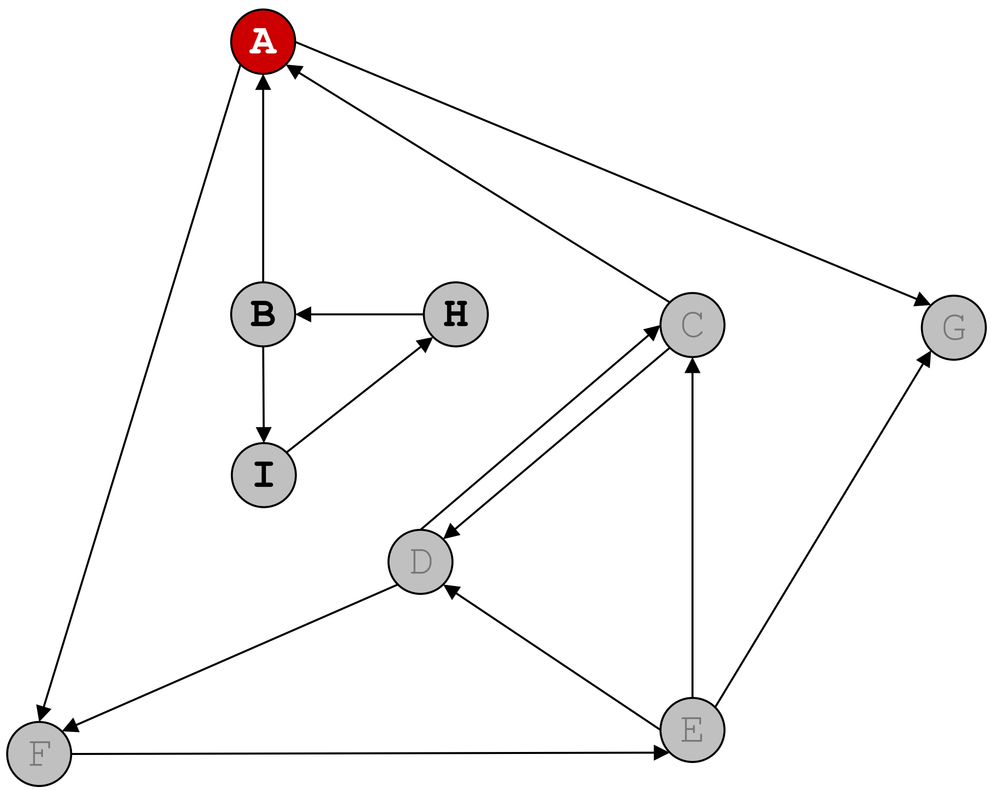
# Directed Depth First Search



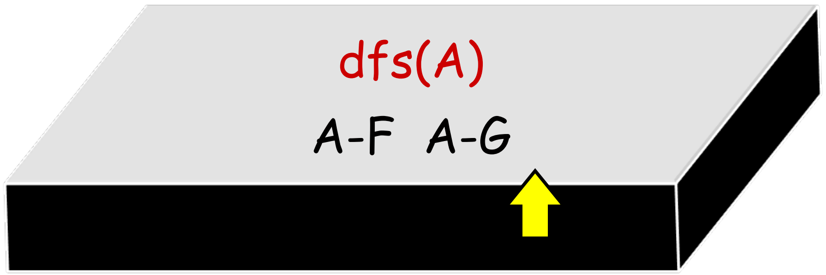

Function call stack:



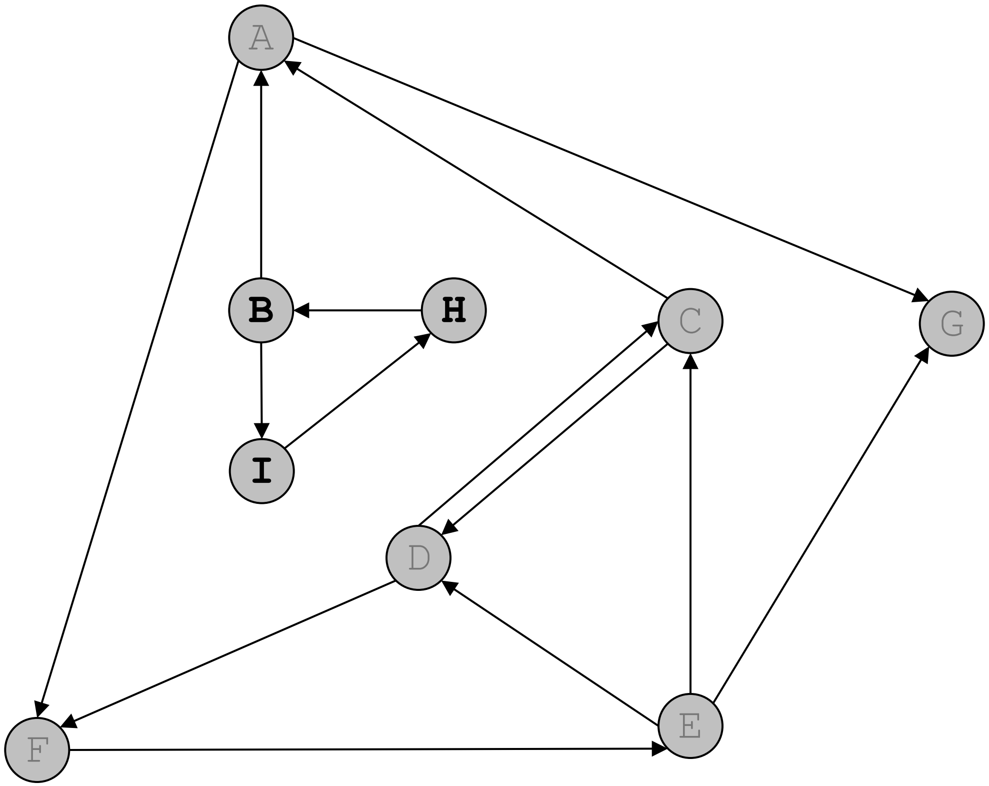
# Directed Depth First Search




Function call stack:



# Directed Depth First Search




Nodes reachable from A: A, C, D, E, F, G

# Question 3: Topological Ordering Algorithm

**Q)** The algorithm described in Section 3.6 for computing a topological ordering of a Directed Acyclic Graph (DAG).

DAG repeatedly finds a node with no incoming edges and deletes it.
At the end, this will produce a topological ordering, provided that the input graph really is a DAG.

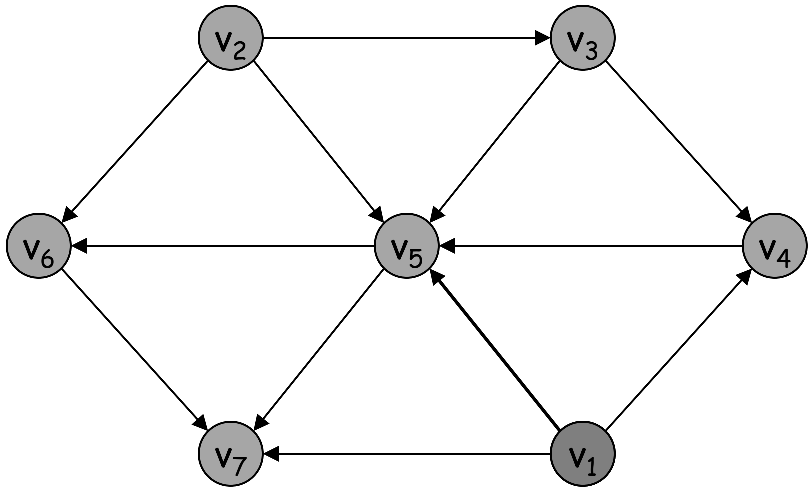
But suppose that we are given an arbitrary graph **that may or may not be a DAG.**

- Extend the topological ordering algorithm so that, given an input directed graph G, it outputs one of the two things:

  a) a topological ordering, thus establishing that G is a DAG;
  b) a cycle in G, thus establishing that G is not a DAG.

The running time of your algorithm should be O(m+n) for a directed graph with **n** nodes and **m** edges

# Topological Ordering Algorithm



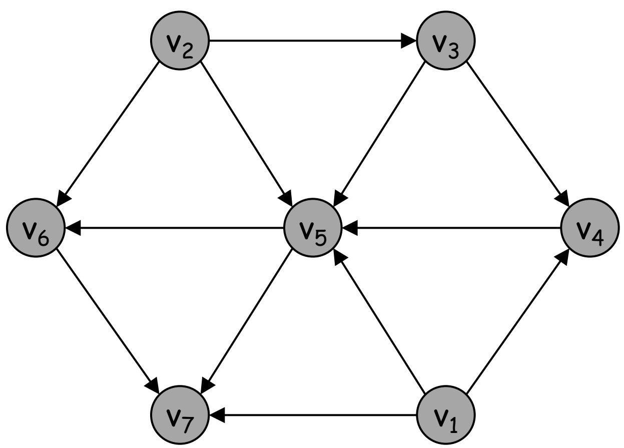

Topological order: ???

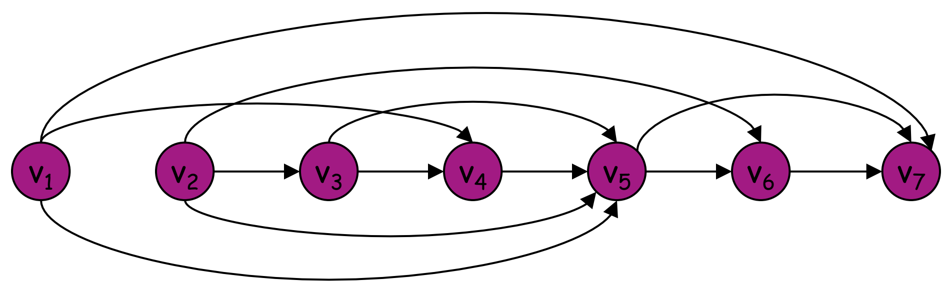
# Topological Ordering Algorithm: Example

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

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

- then **j** appears after **i**




Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

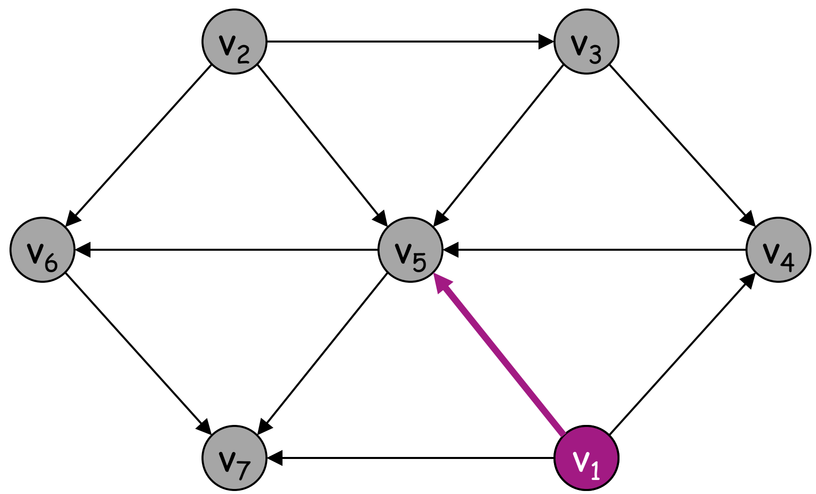
# Computing Topological Ordering

- In every DAG G, there is at least one node with no incoming edges

```
procedure topologicalSort(DAG G)
    let result be an empty list.
    while G is not empty
        let v be a node in G with indegree 0
        add v to result
        remove v from G
    return result
```

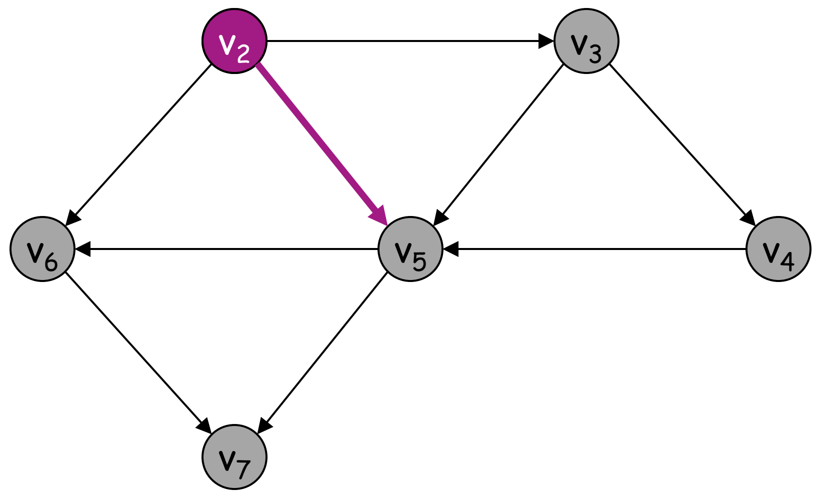
# Computing Topological Ordering

- Look for nodes with no incoming edges



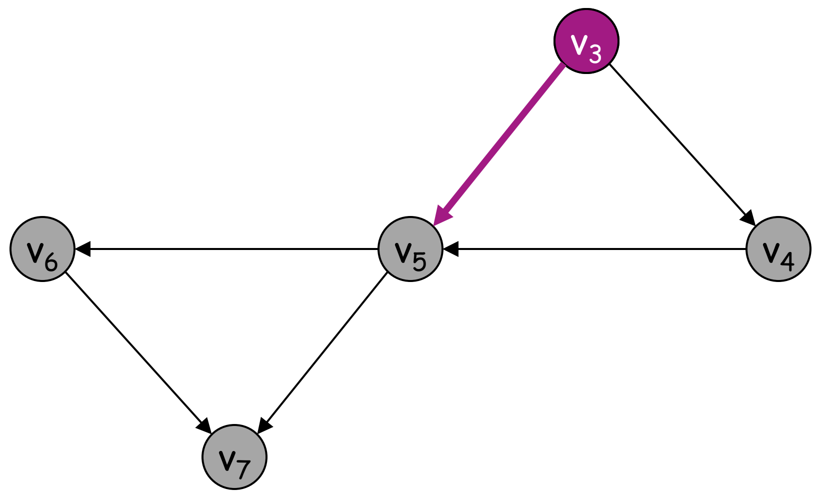

Topological order:

# Topological Ordering



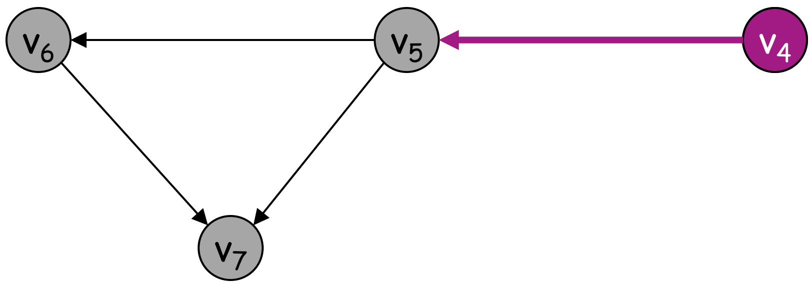

Topological order: $v_1$

# Topological Ordering Algorithm: Example



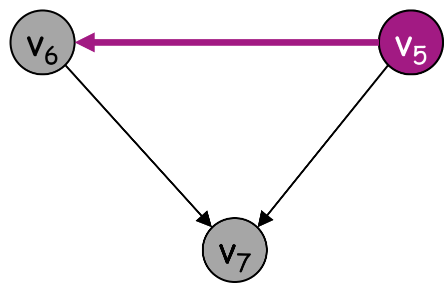

Topological order: $v_1$, $v_2$

# Topological Ordering Algorithm: Example



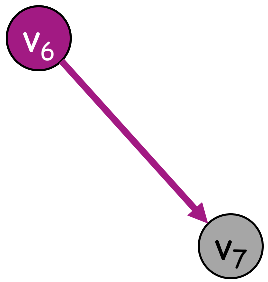

Topological order: $v_1$, $v_2$, $v_3$

# Topological Ordering Algorithm: Example




Topological order: $v_1, v_2, v_3, v_4$

# Topological Ordering Algorithm: Example

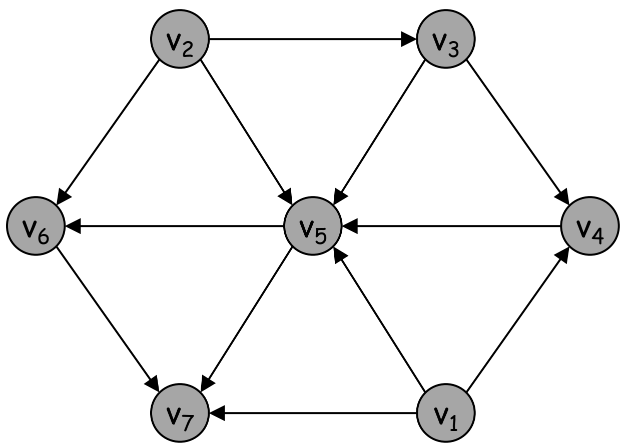


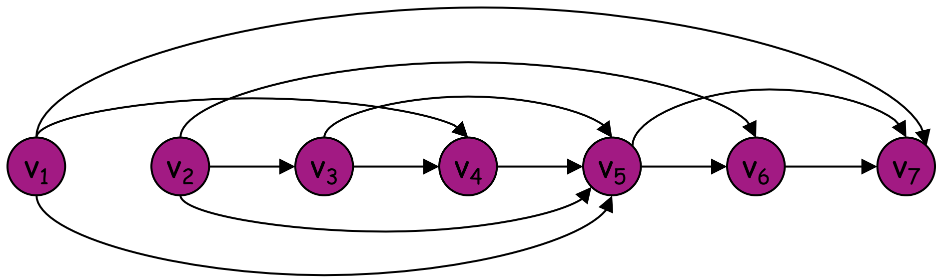

Topological order: $v_1, v_2, v_3, v_4, v_5$

# Topological Ordering Algorithm: Example


Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$

# Topological Ordering




Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

# Find Cycle: Topological Ordering

- If in each iteration, we find a node with no incoming edges then we will keep the algorithm

- **If in some iteration, every node has at least one incoming edge, then this means G must contain a cycle**

We need to modify the above graph to include this condition:

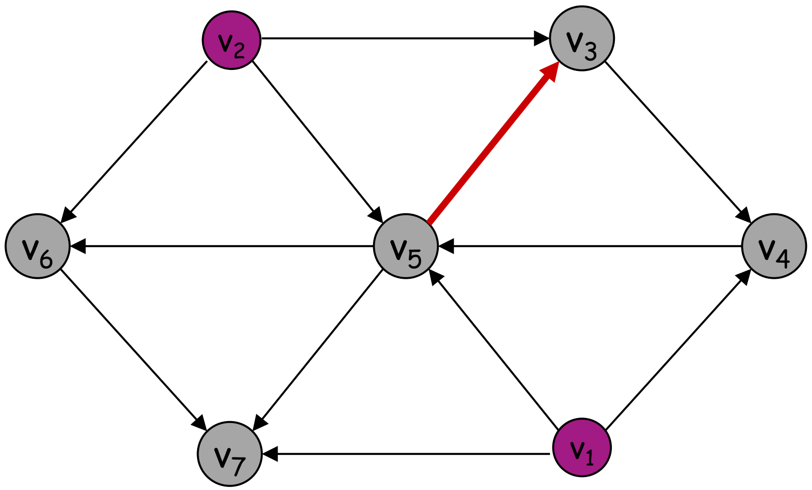
  - We backtrack the edges,
    - i.e. we follow the first edge back until we reach a node that was visited before taking the first edge on the adjacency list of incoming edges assures O(n),

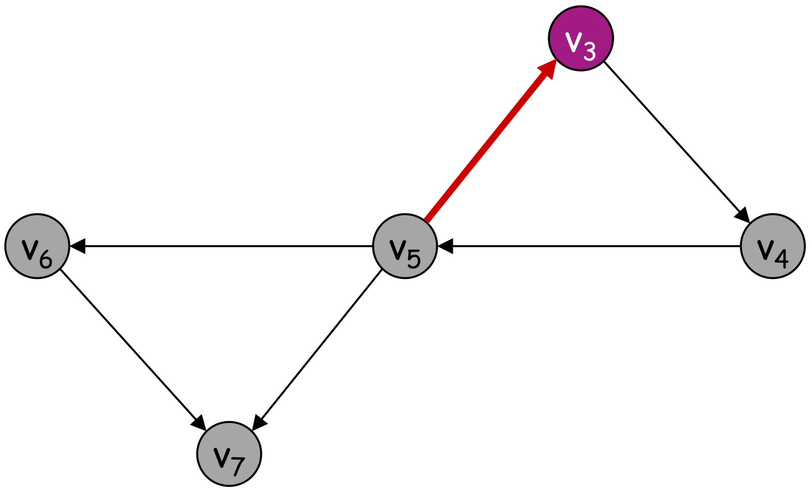
  - We don't make a search since every node has an incoming edge, we will revisit a node v the nodes between two consecutive visits will be the nodes in the cycle C

# Find Cycle: Topological Ordering

In the third iteration, all the nodes have at least one incoming edge

- Choose a node
- Choose the first edge on the adjacency list of incoming edges so that this can run in constant time per node
- Repeat until you find a cycle



# Question 4

**Q)** A problem on graph is to assign integers to the vertices of **G** such that if there is a directed edge from vertex **i** to vertex **j**, then **i** < **j**.

One solution to the problem is search for a vertex with no incoming edge, assign this vertex the lowest number, and delete it from the graph, along with all outgoing edges. Repeat this process on the resulting graph, assign the next lowest number, and so on.

- a) Write the data structures and algorithm necessary to solve this problem.
- b) What is the complexity of your algorithm in terms of the number of vertices **n** and/or the number of edge **m**?
- c) Draw an example graph and show the relationship between your data structures and the graph.

# Solution: Topological Ordering

a) Write the data structures and algorithm necessary to solve this problem.

A two dimensional integer array is necessary to represent the topology of graph and another single dimensional array can be used to keep values assigned to vertices:

**Topology:Array[1…n+1, 1…n+1]** of integer {0/1}; *// n; #of vertices*
**Values:Array[1…n+1]** of integer

# Topological Ordering

## a)

Topology:Array[1…n, 1…n] of integer {0/1};

Values:Array[1…n] of integer

---

- Function AssignNumbers (graph: Topology):Numbers:Values;
- TempGraph: Topology {TempGraph is used not to lose the original graph structure - optional}
- k: integer
- TempGraph <- Graph
- k <- 1
- While k < (n+1) begin
-   find a column having no '1's (say column **j**)
-   if no such column then **return error** *// what is that means?*
-   else
-     Number[j] <- k
-     k++
-     assign '0' to all entries in row **j** **(**say row **i)**
- end
- return Numbers

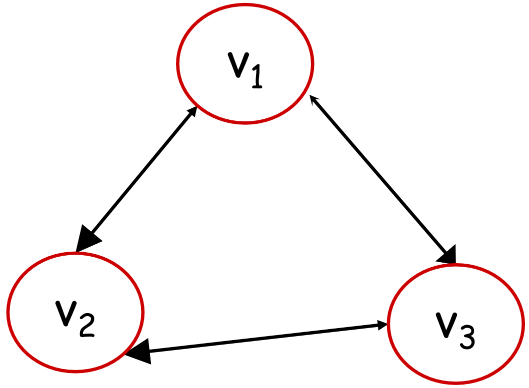
b) What is the complexity of your algorithm in terms of the number of vertices **n** and/or the number of edge **m**?

Θ(n3); outer while loop is repeated n times. In the loop in order to locate a column having no '1' entries one should check whole two dimensional array (worst case)

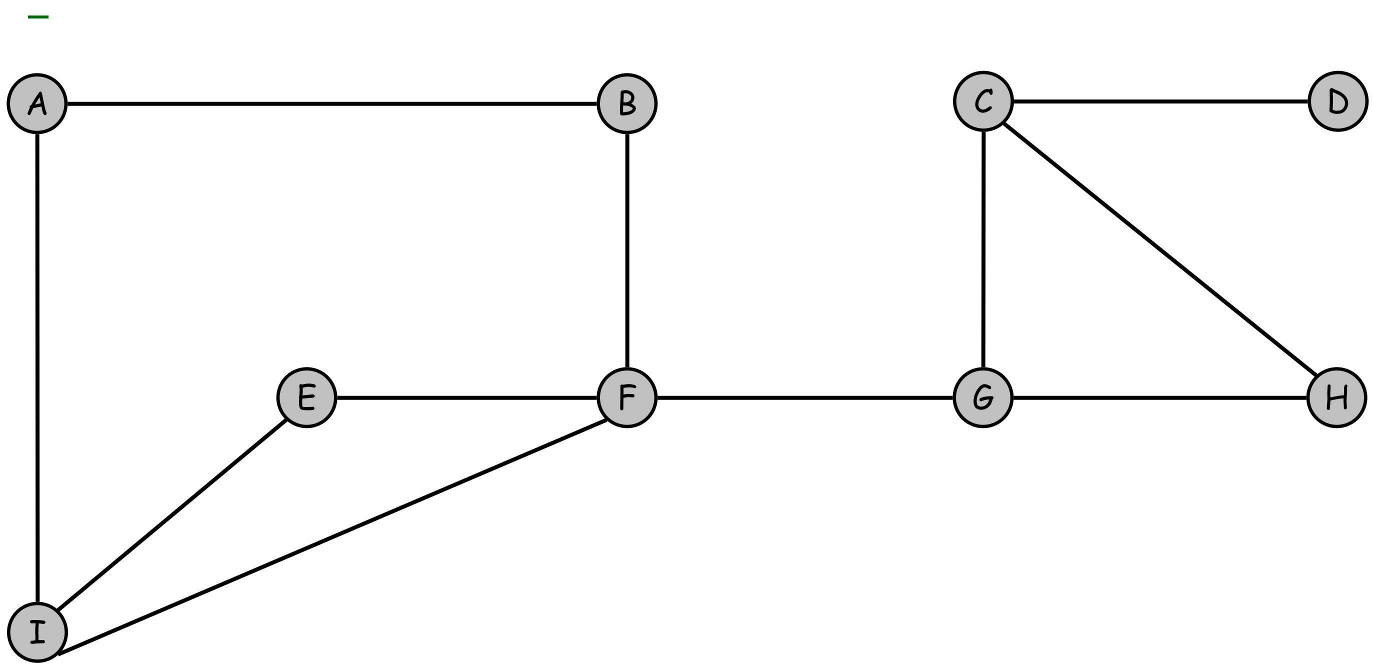
c) Draw an example graph and show the relationship between your data structures and the graph.

Assume Numbers; 3, 5, 6

| | [1] | [2] | [3] |
|---|---|---|---|
| [1] | 0 | 1 | 1 |
| [2] | 0 | 0 | 0 |
| [3] | 0 | 1 | 0 |



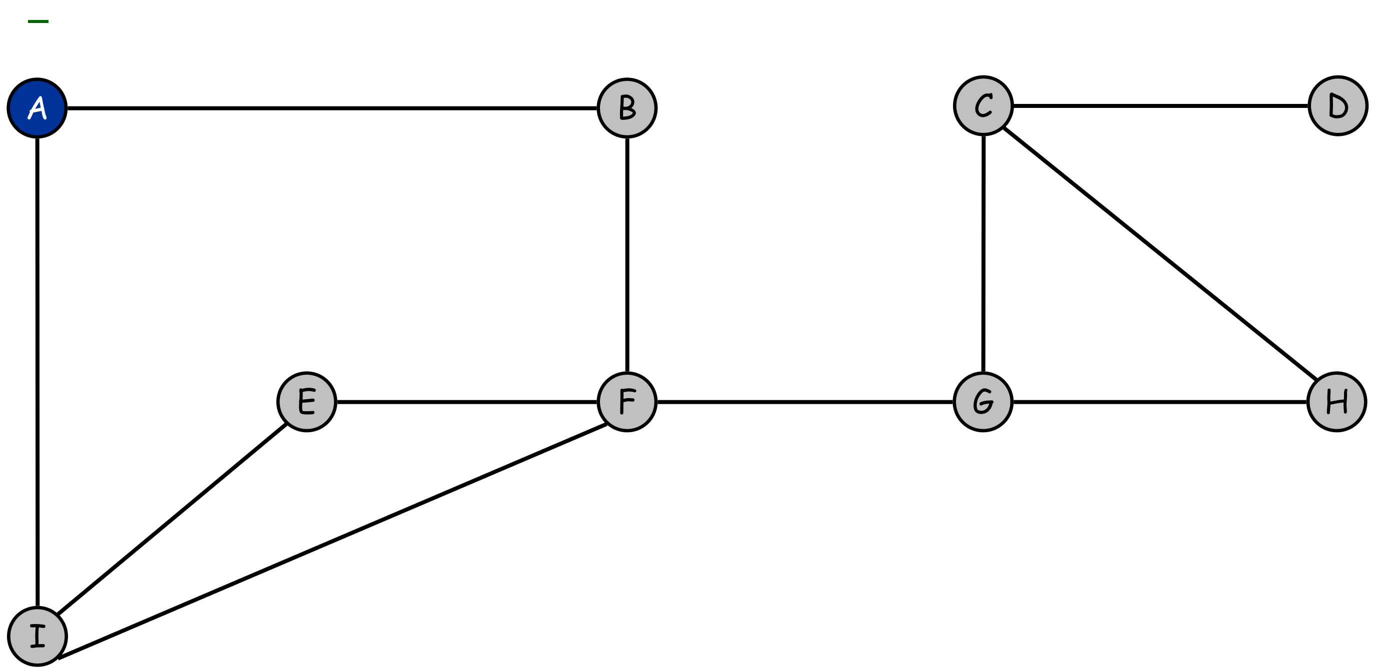
# Question 5: Undirected Breadth First Search




front

FIFO Queue

# Breadth First Search

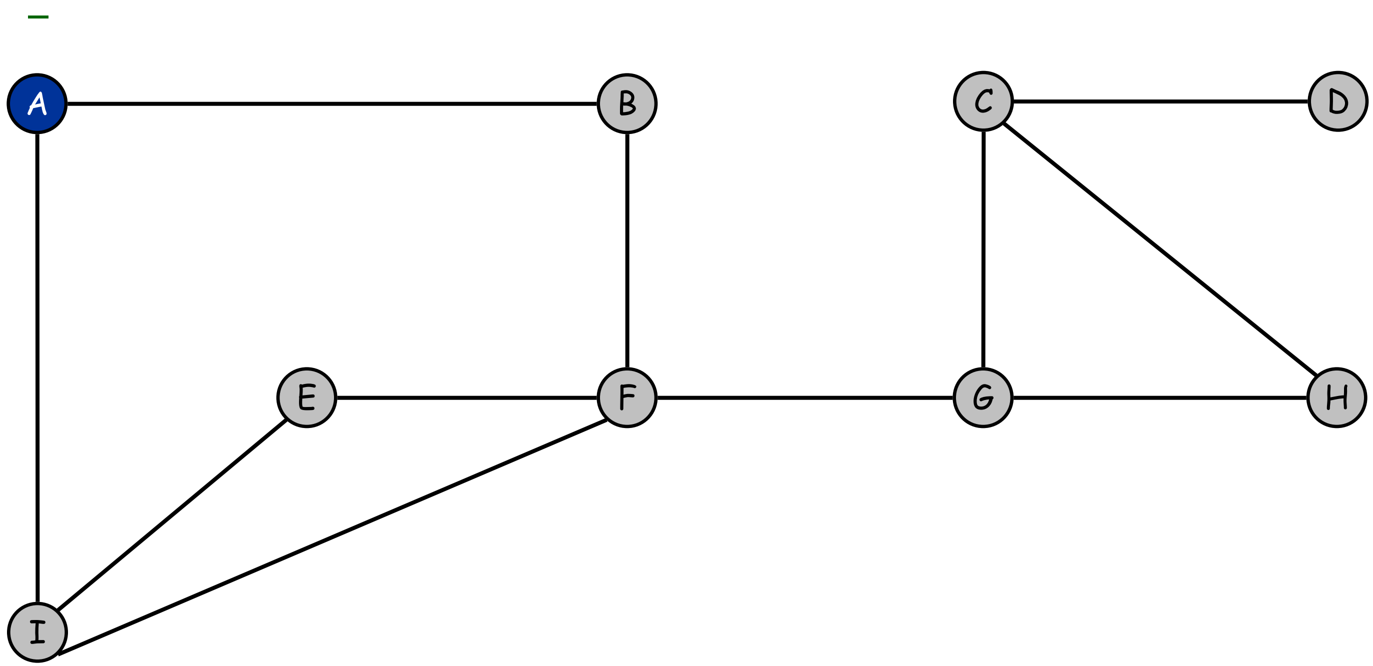



enqueue source node

front | A

FIFO Queue

# Breadth First Search



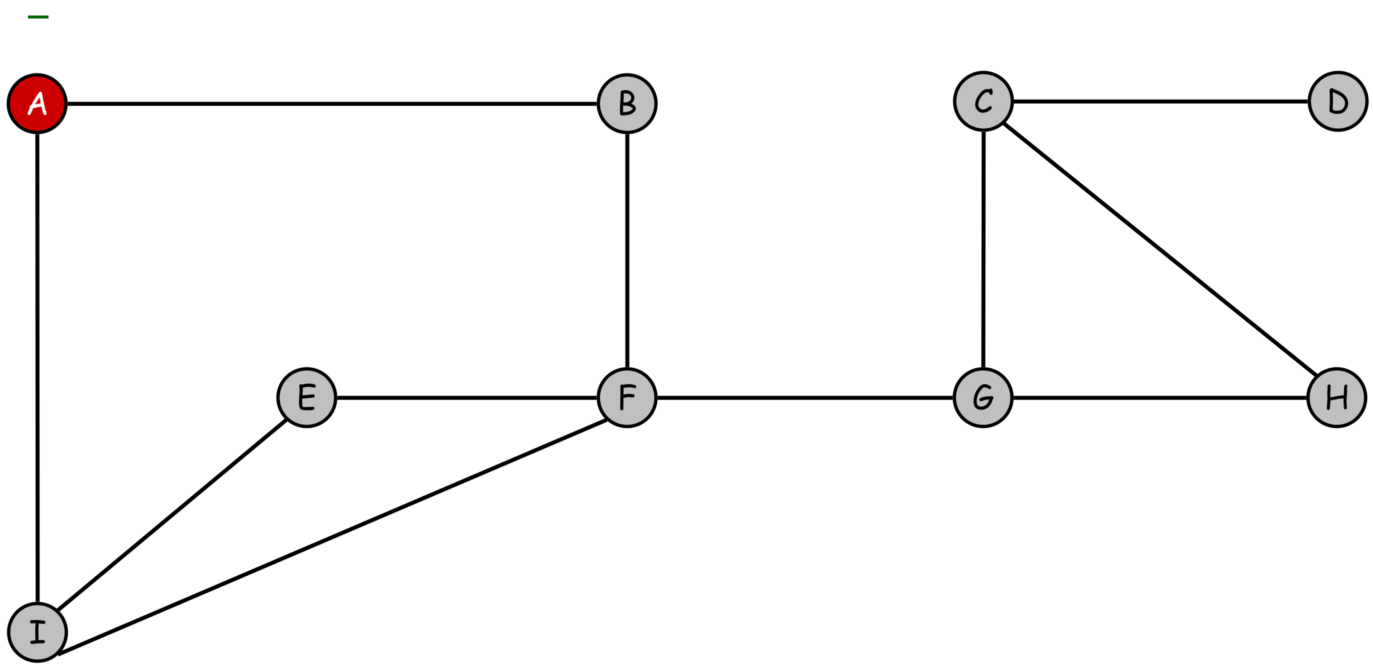

dequeue next vertex

front

A

FIFO Queue

# Breadth First Search

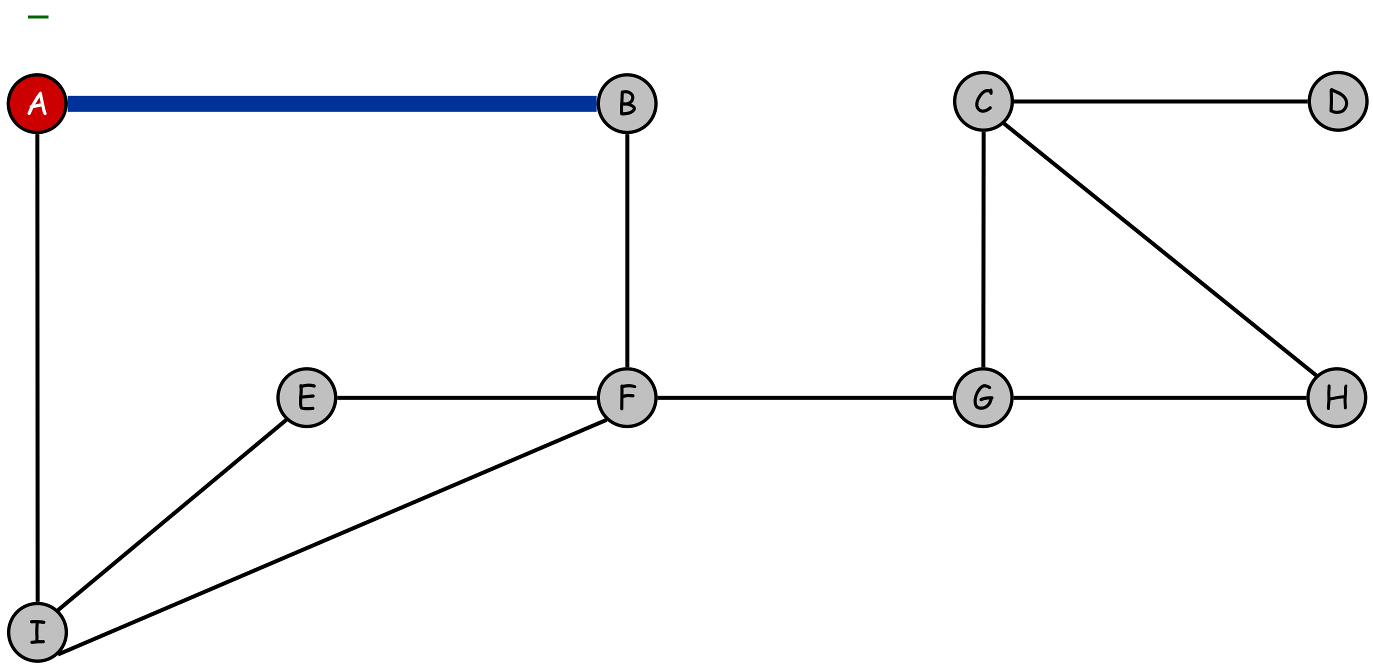



visit neighbors of A

front

FIFO Queue

# Breadth First Search

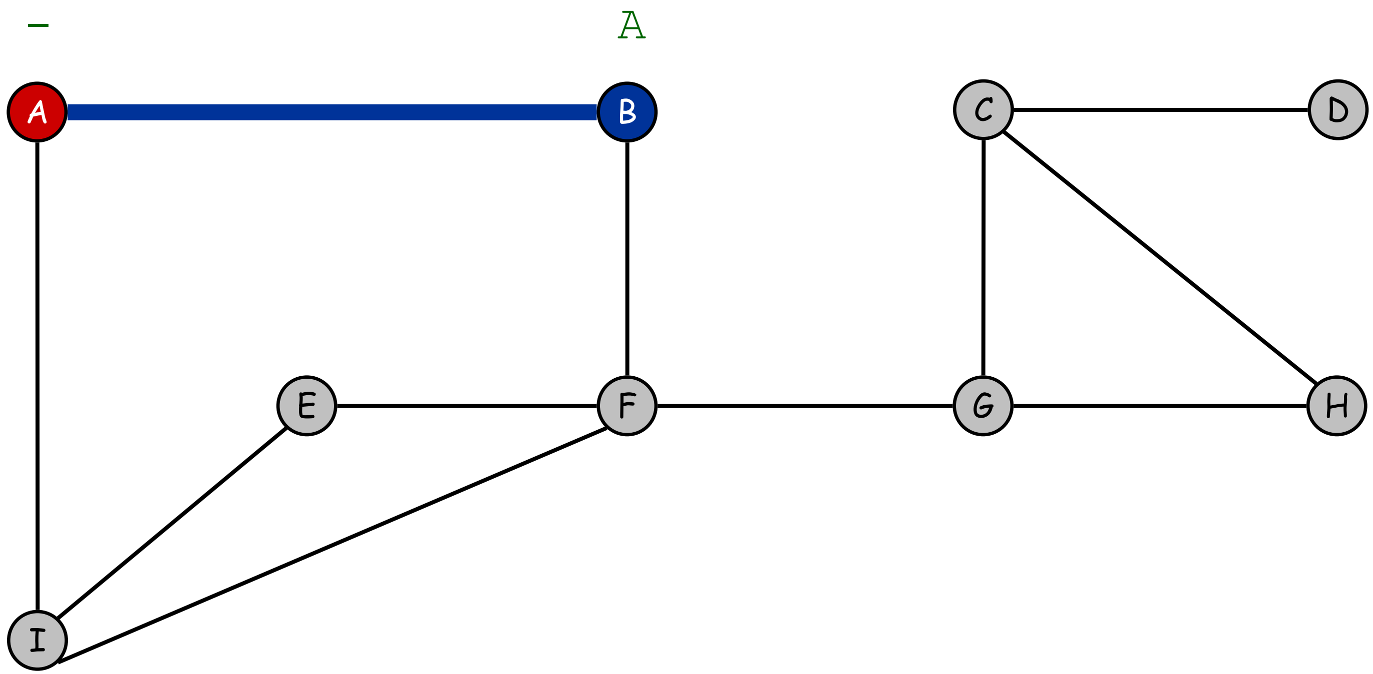



visit neighbors of A

front

FIFO Queue

# Breadth First Search

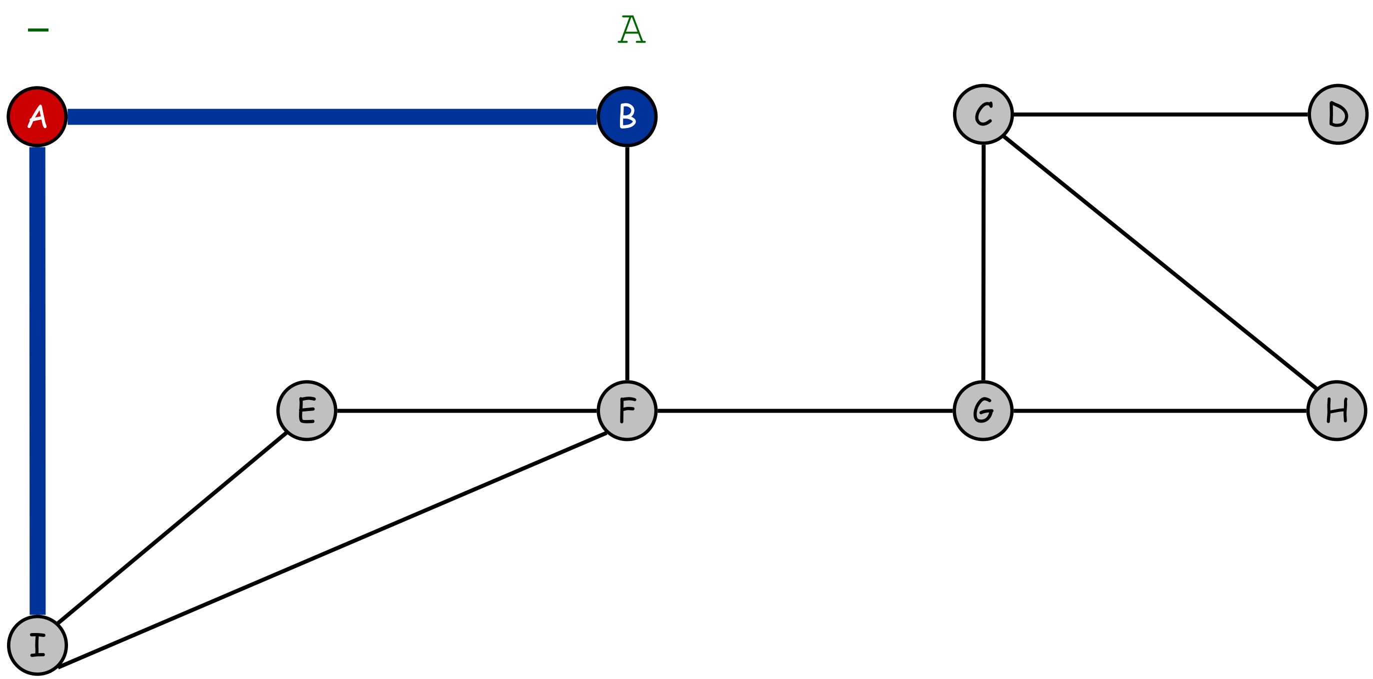



B discovered

front B

FIFO Queue

# Breadth First Search



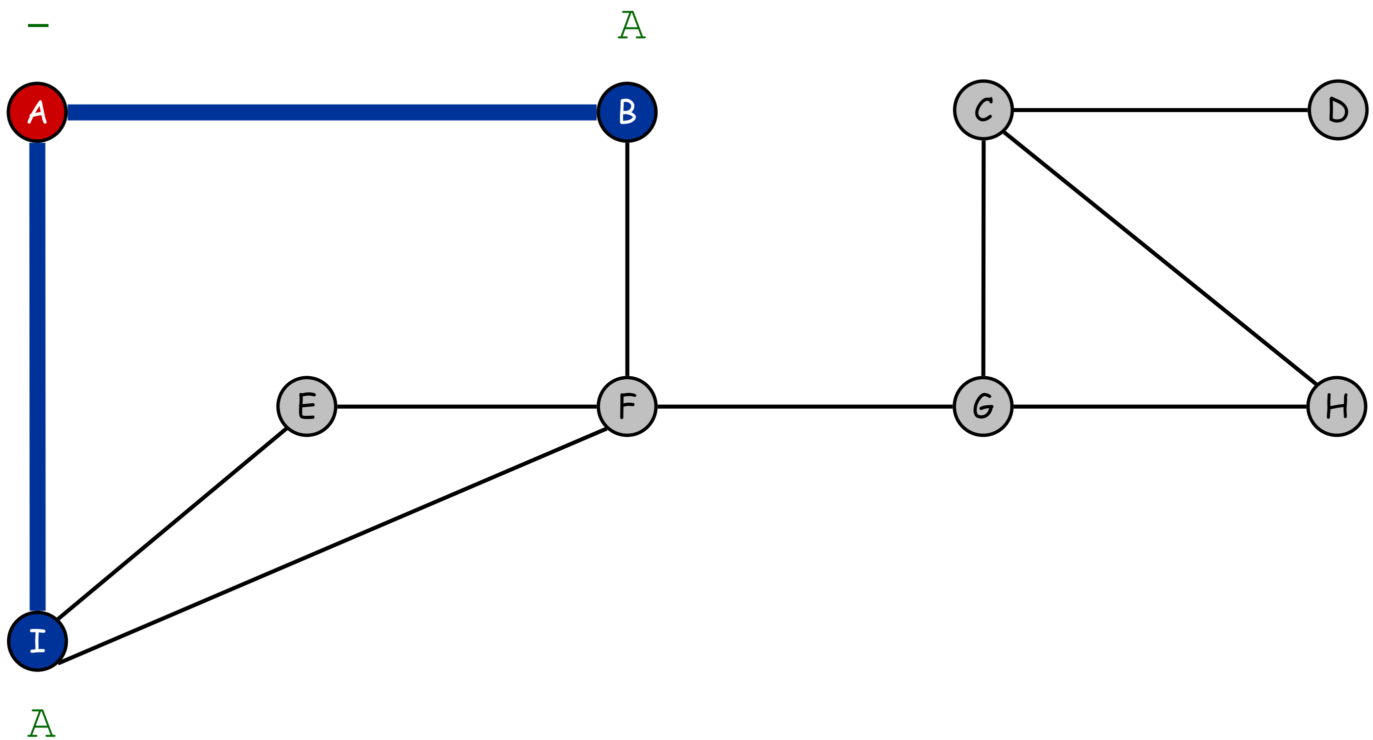

visit neighbors of A

front

B

FIFO Queue

# Breadth First Search



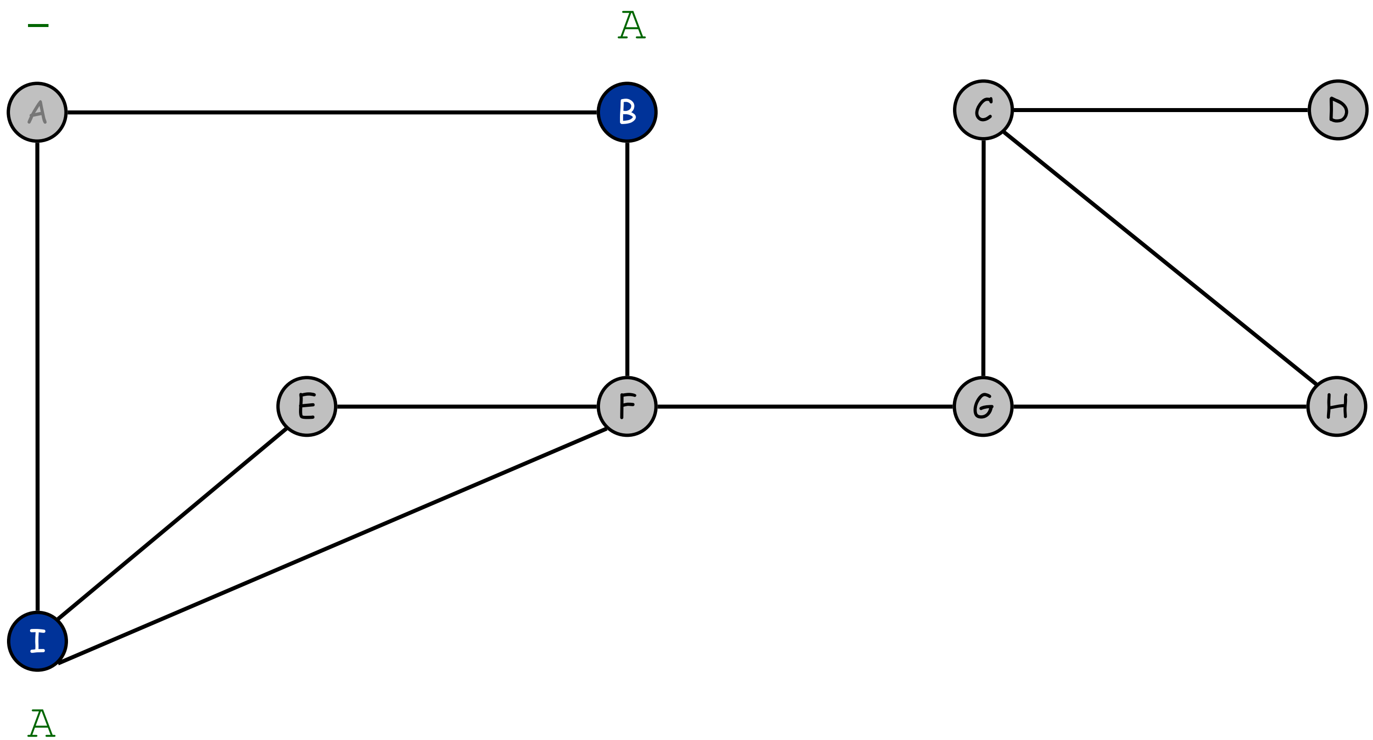

I discovered

front

B I

FIFO Queue

# Breadth First Search



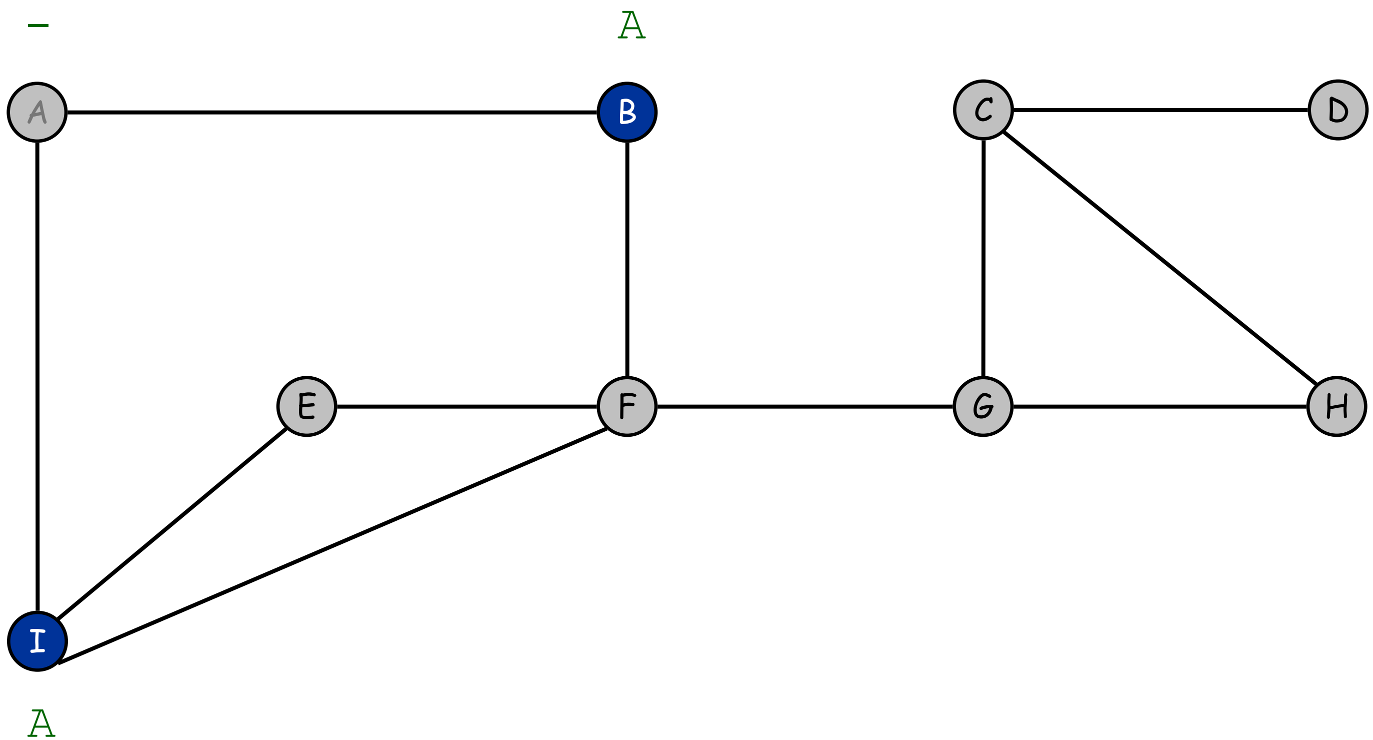

finished with A

front

B I

FIFO Queue

# Breadth First Search

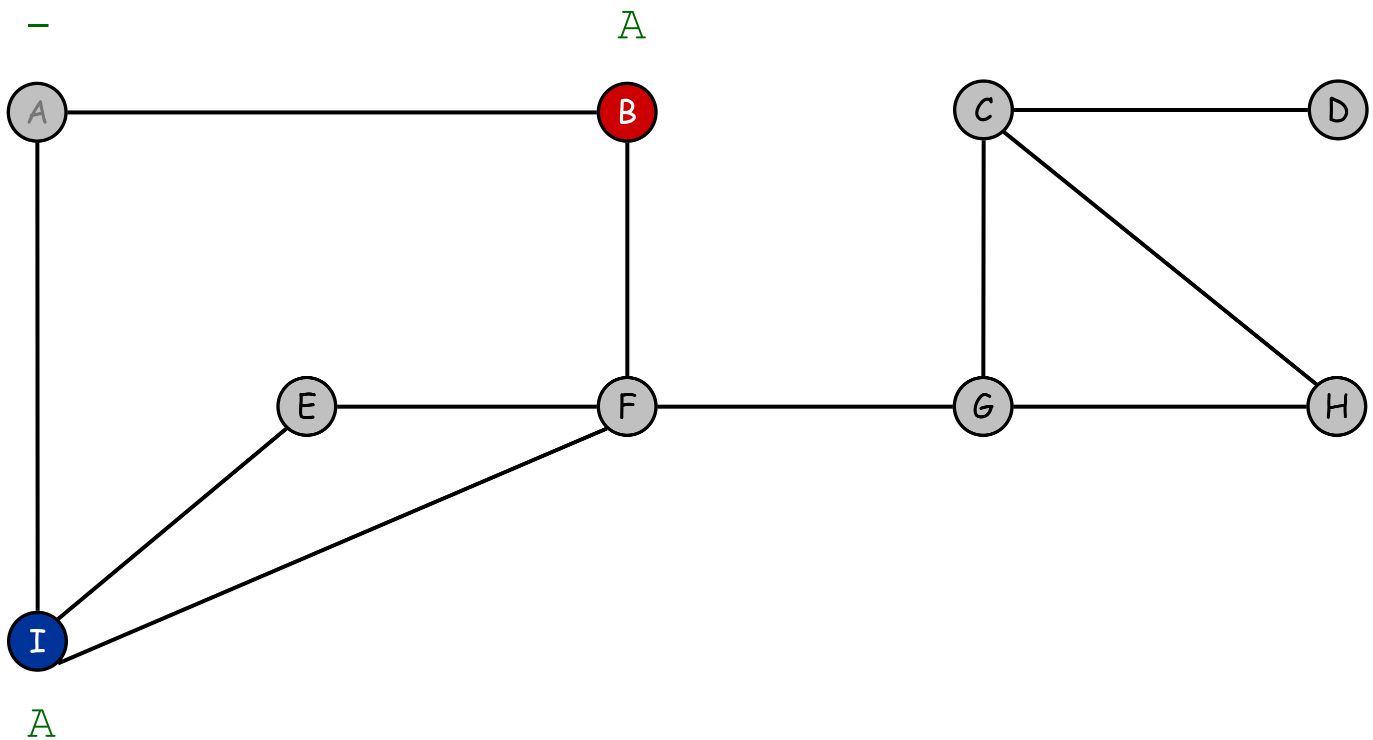



dequeue next vertex

front | B I

FIFO Queue

# Breadth First Search

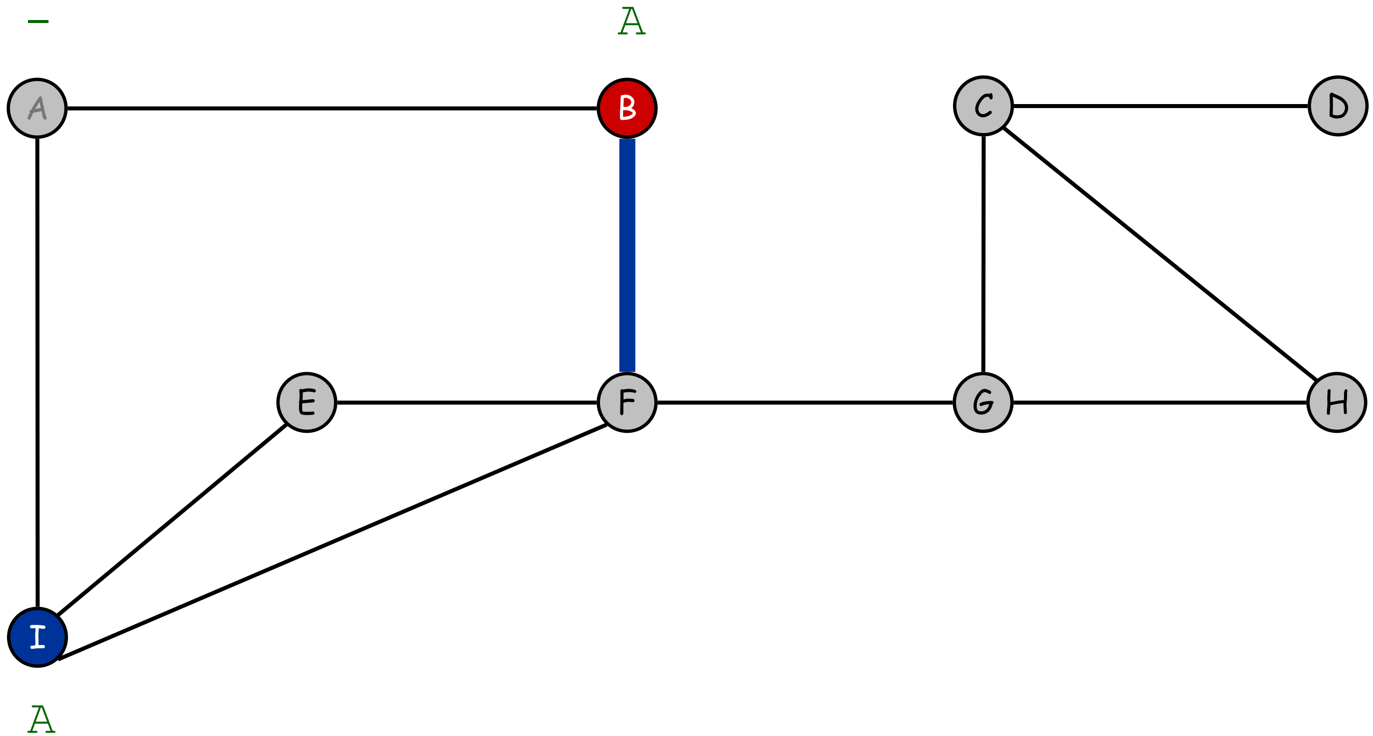



visit neighbors of B

front I

FIFO Queue

# Breadth First Search

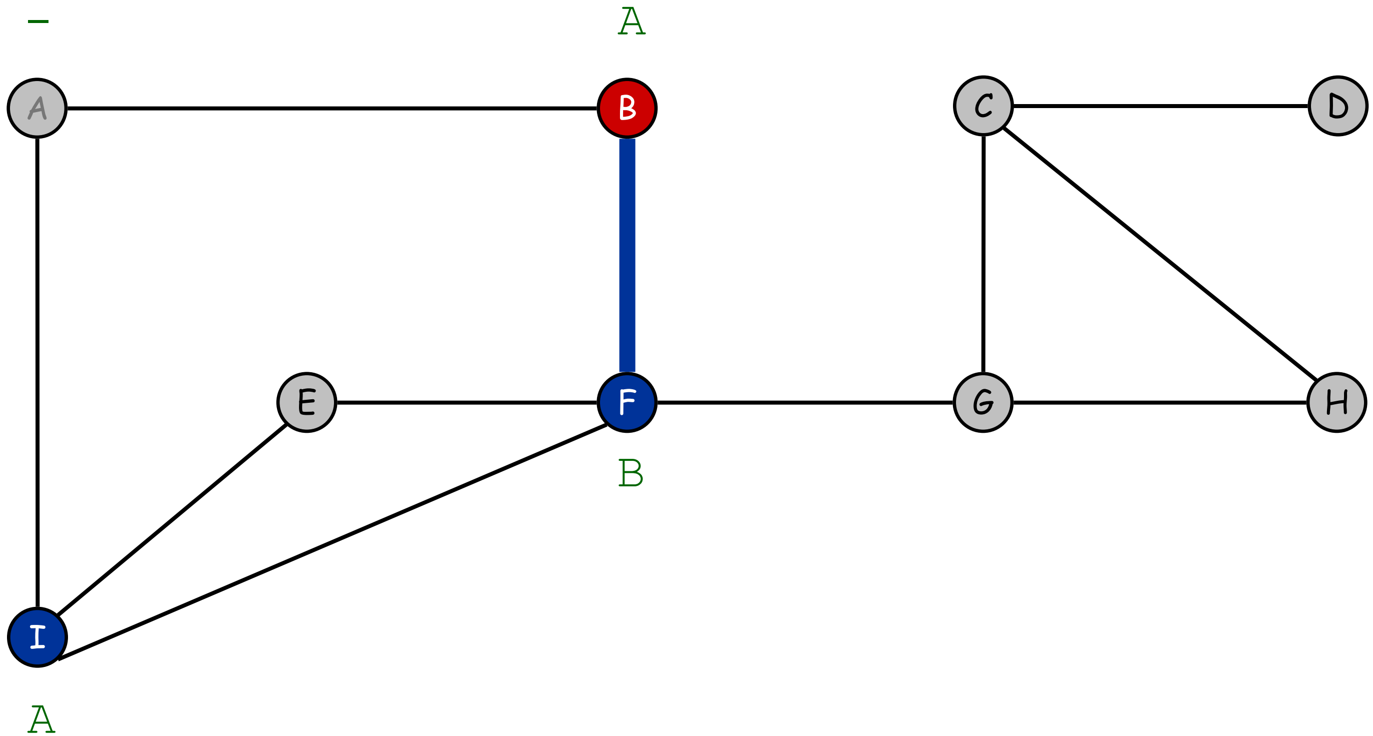



visit neighbors of B

front I

FIFO Queue

# Breadth First Search



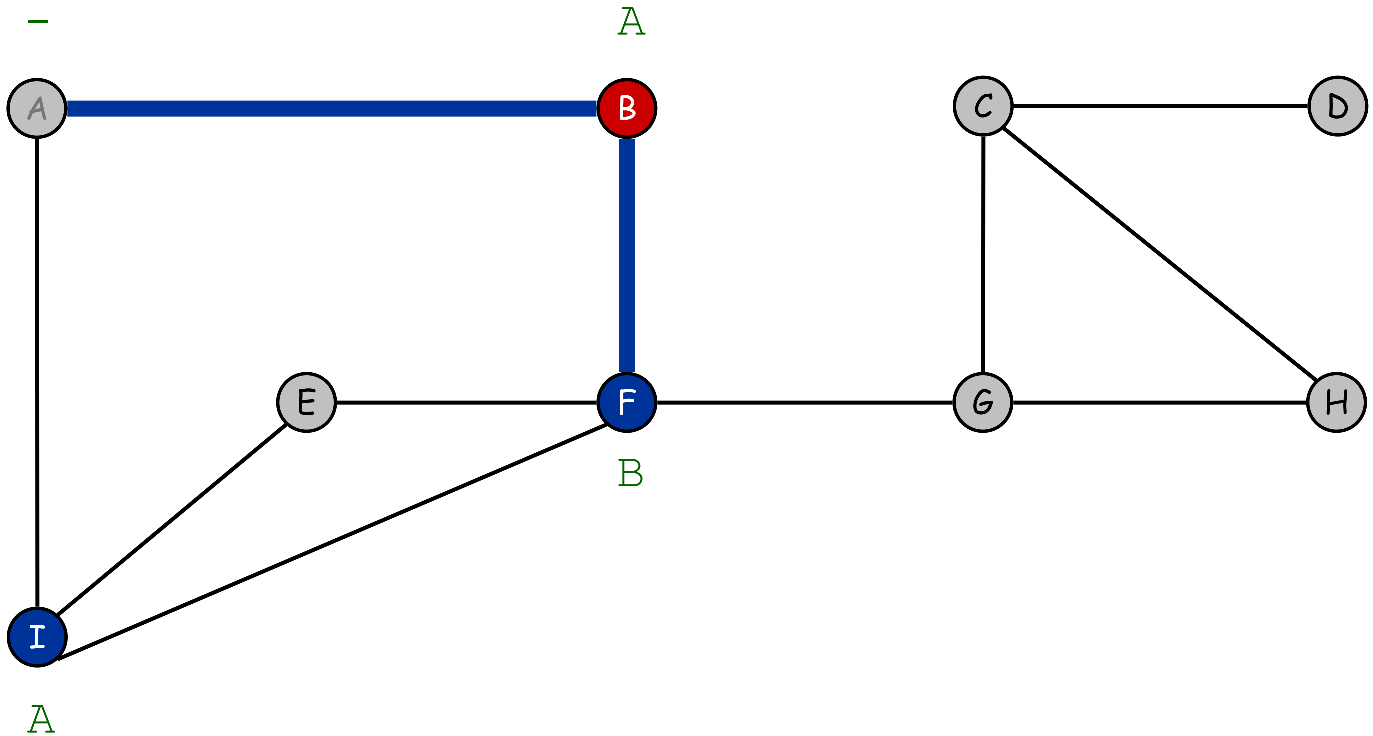

F discovered

front

I F

FIFO Queue

# Breadth First Search



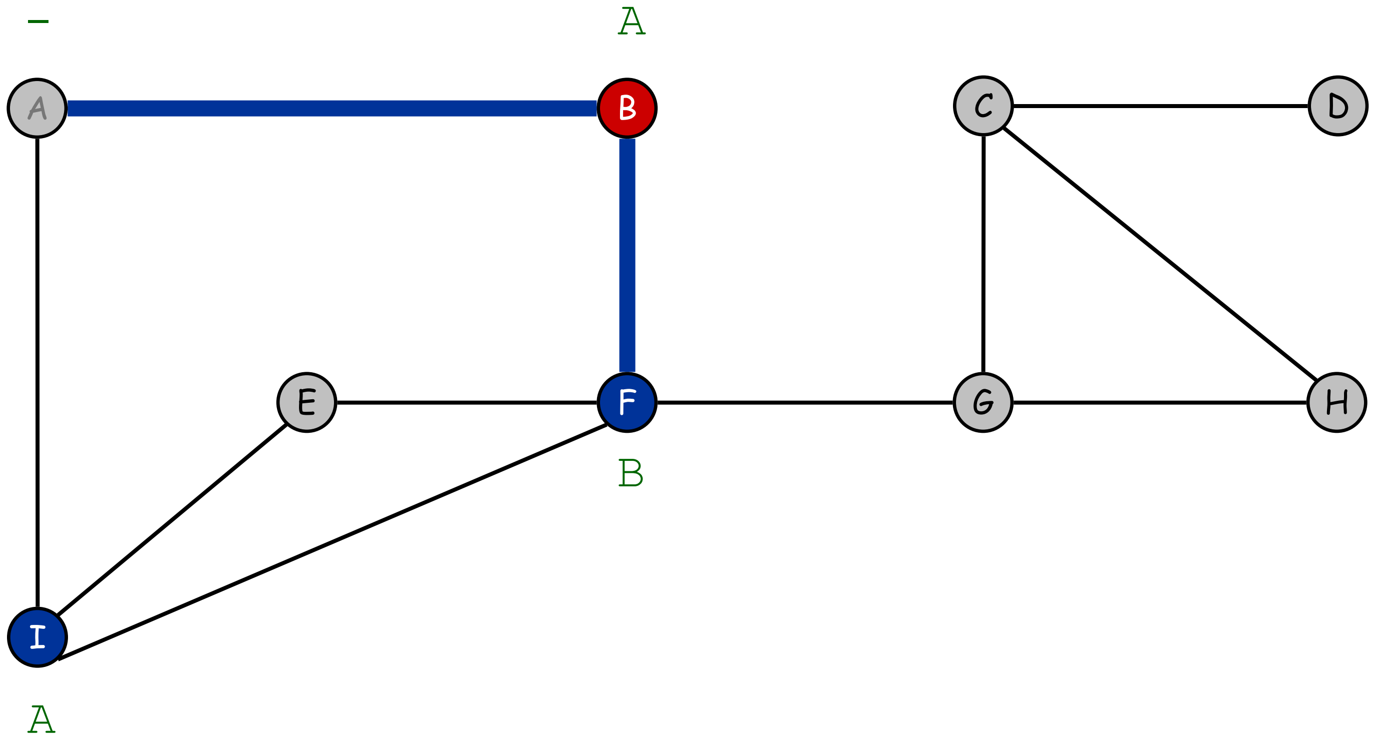

visit neighbors of B

front

I F

FIFO Queue

# Breadth First Search

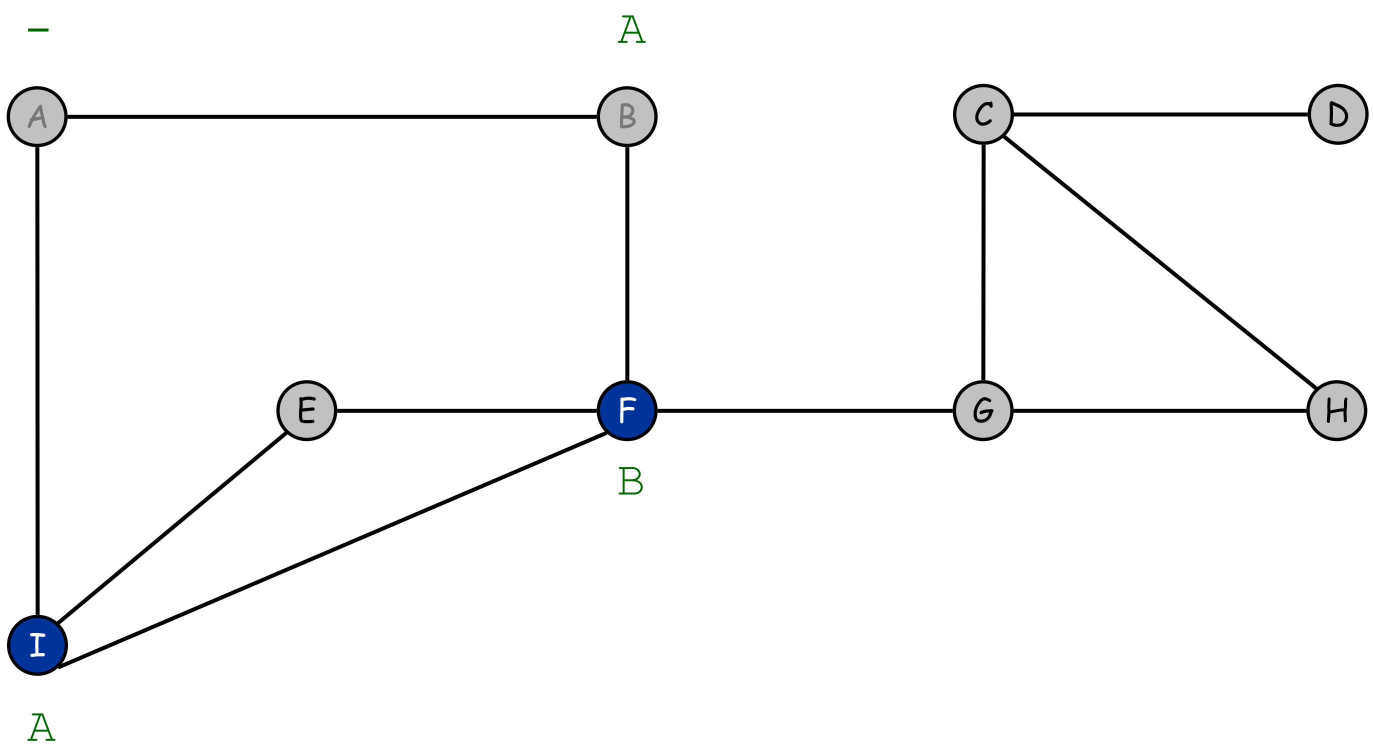



A already discovered

front I F

FIFO Queue

# Breadth First Search



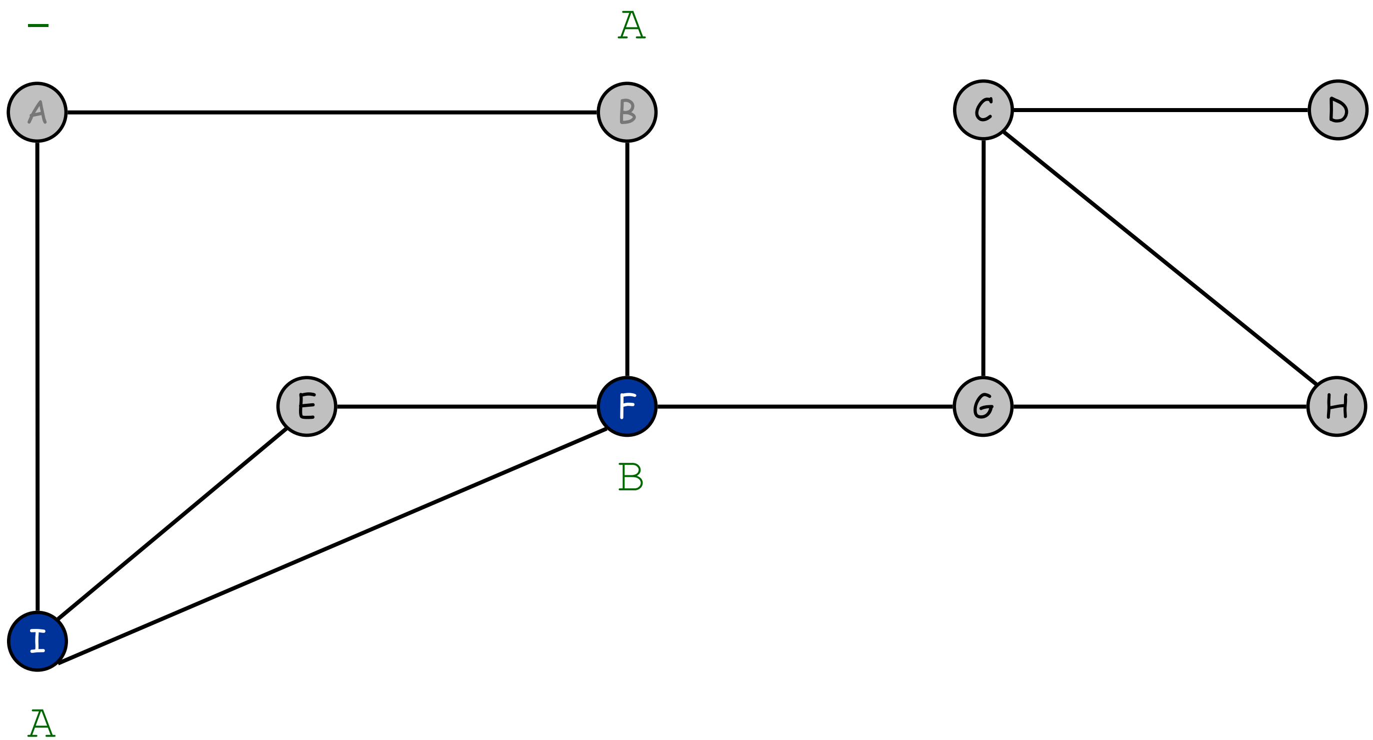

finished with B

front

I F

FIFO Queue

# Breadth First Search

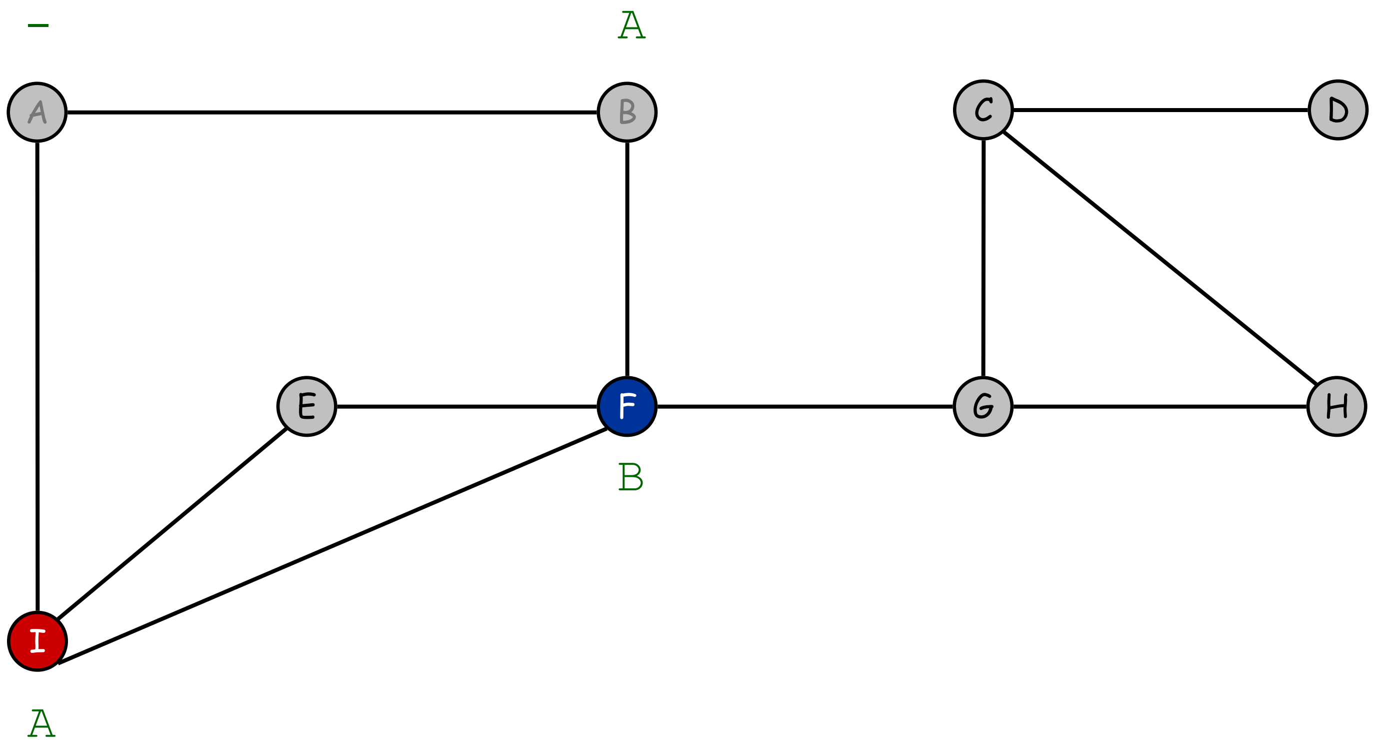



dequeue next vertex

front | I F

FIFO Queue

# Breadth First Search

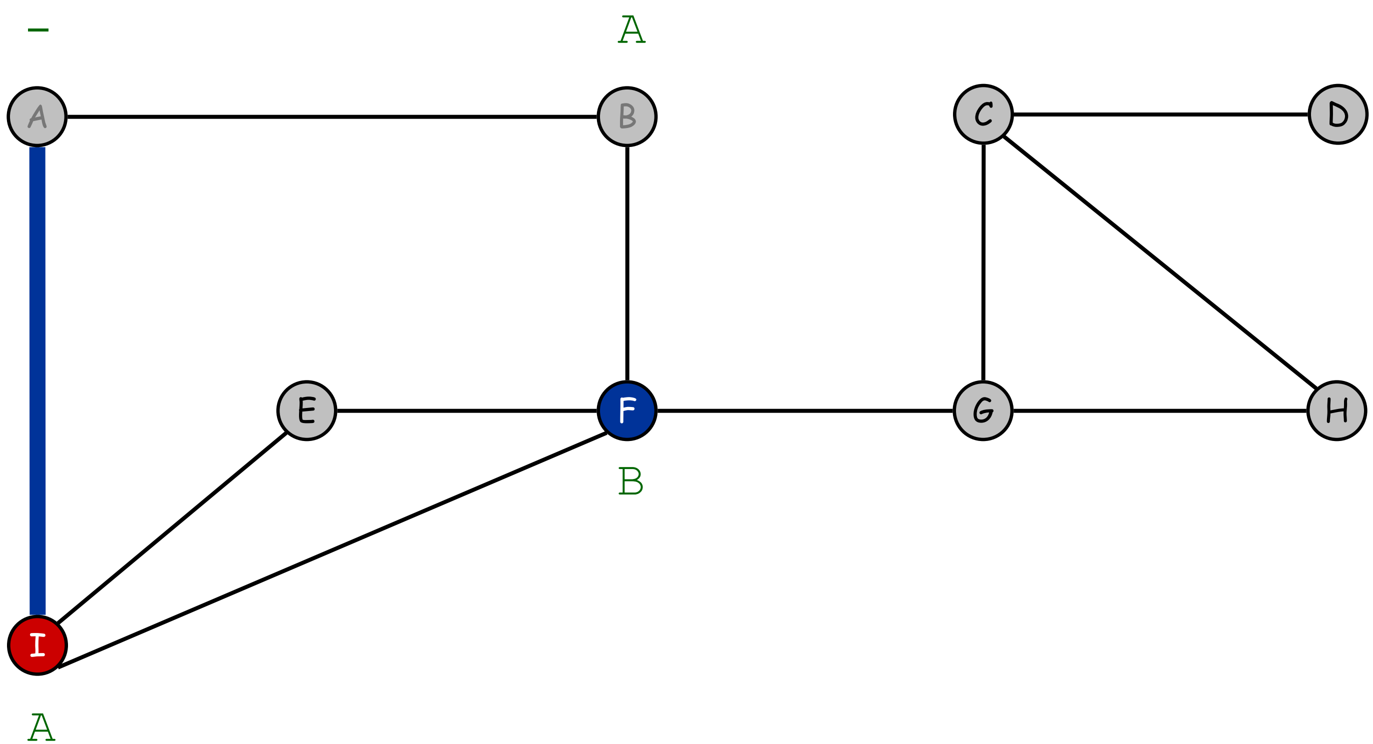



visit neighbors of I

front F

FIFO Queue

# Breadth First Search

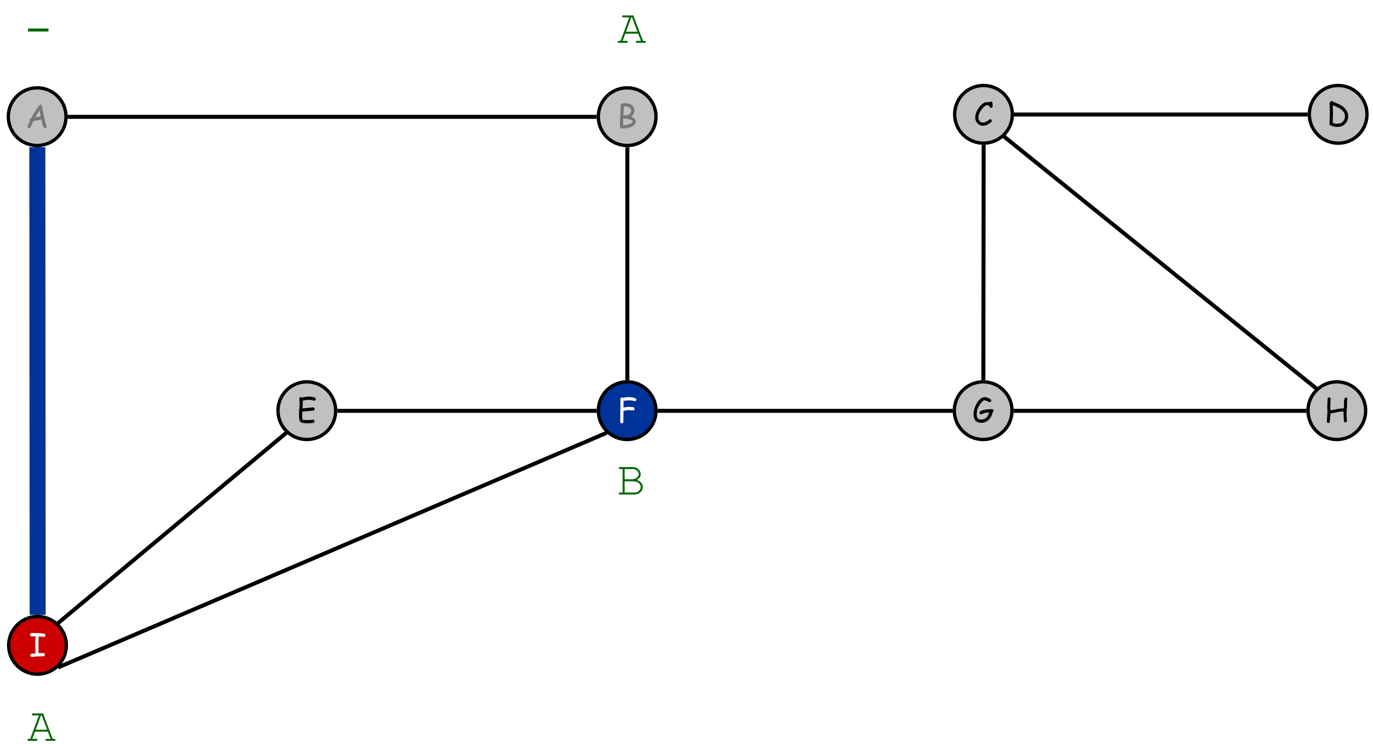



visit neighbors of I

front F

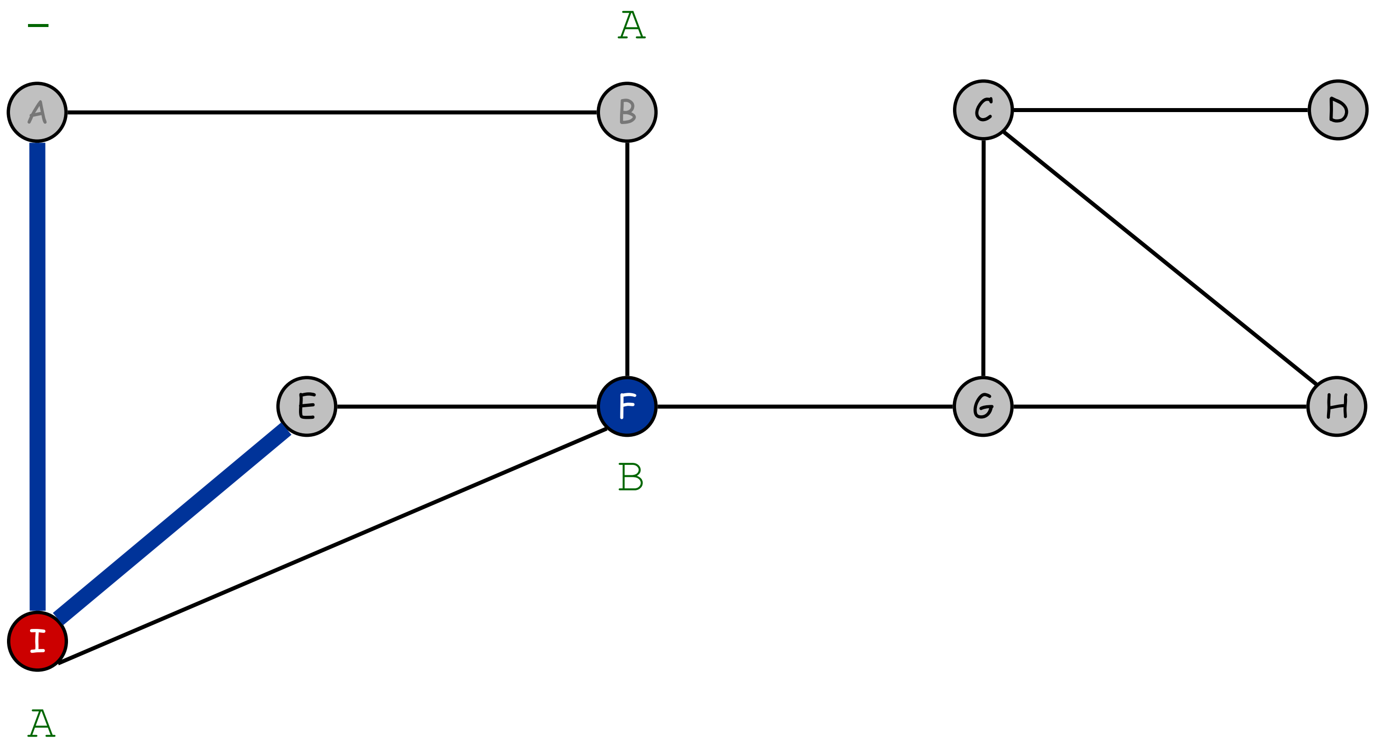
FIFO Queue

# Breadth First Search




A already discovered

# Breadth First Search

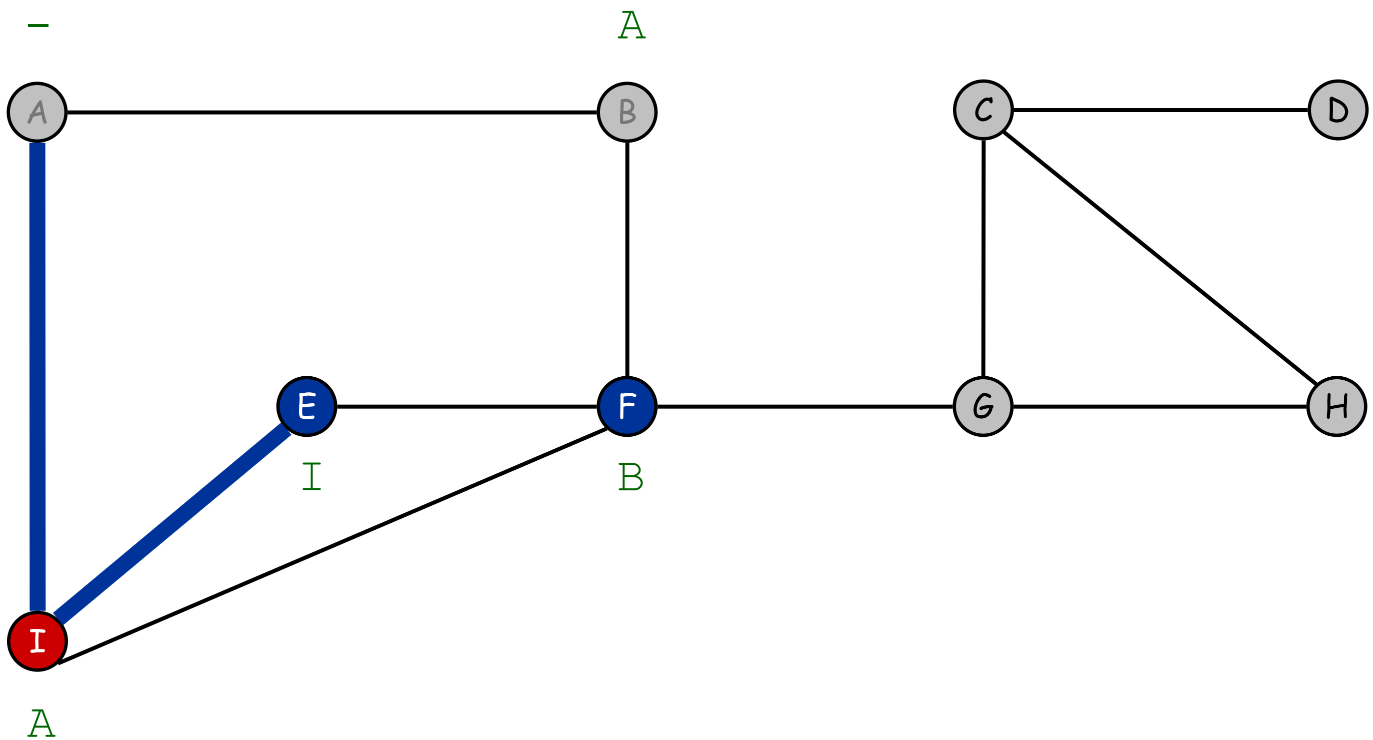



visit neighbors of I

front F

FIFO Queue

# Breadth First Search

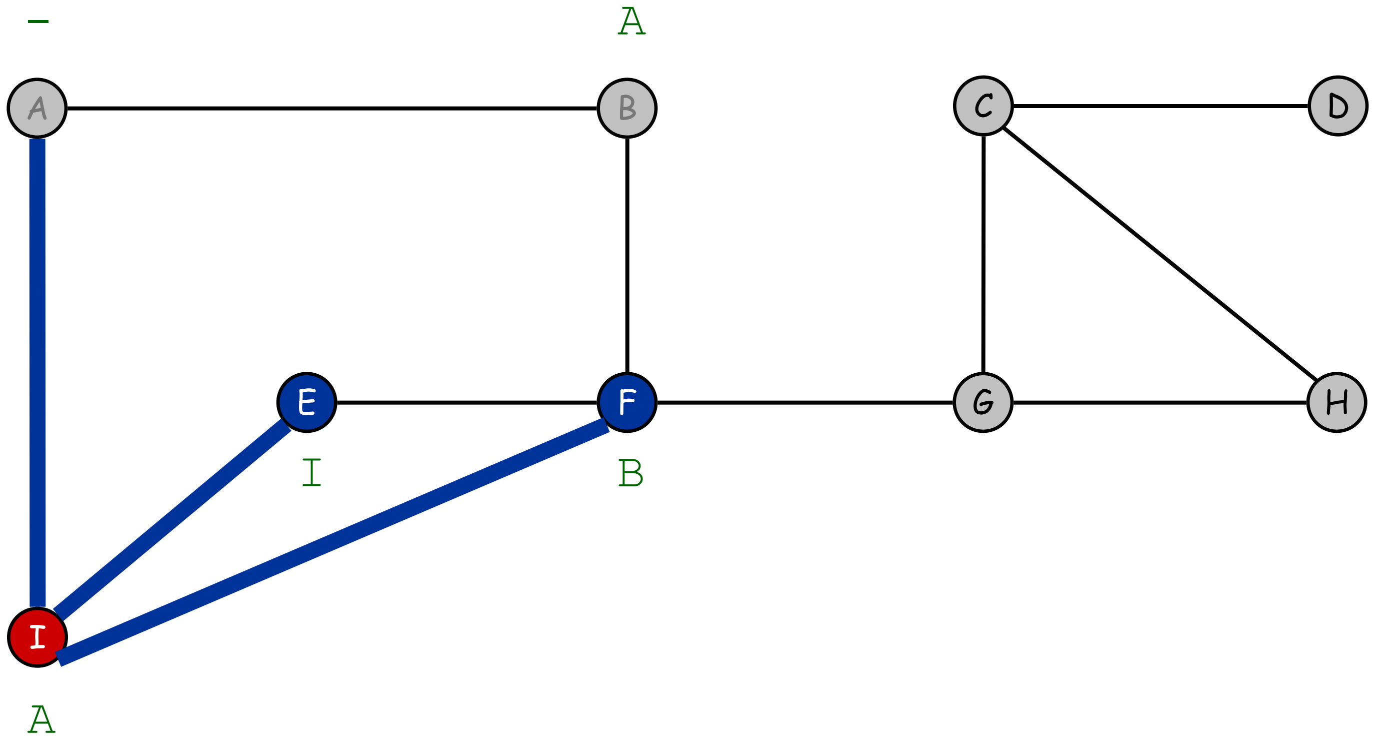



E discovered

front | F E

FIFO Queue

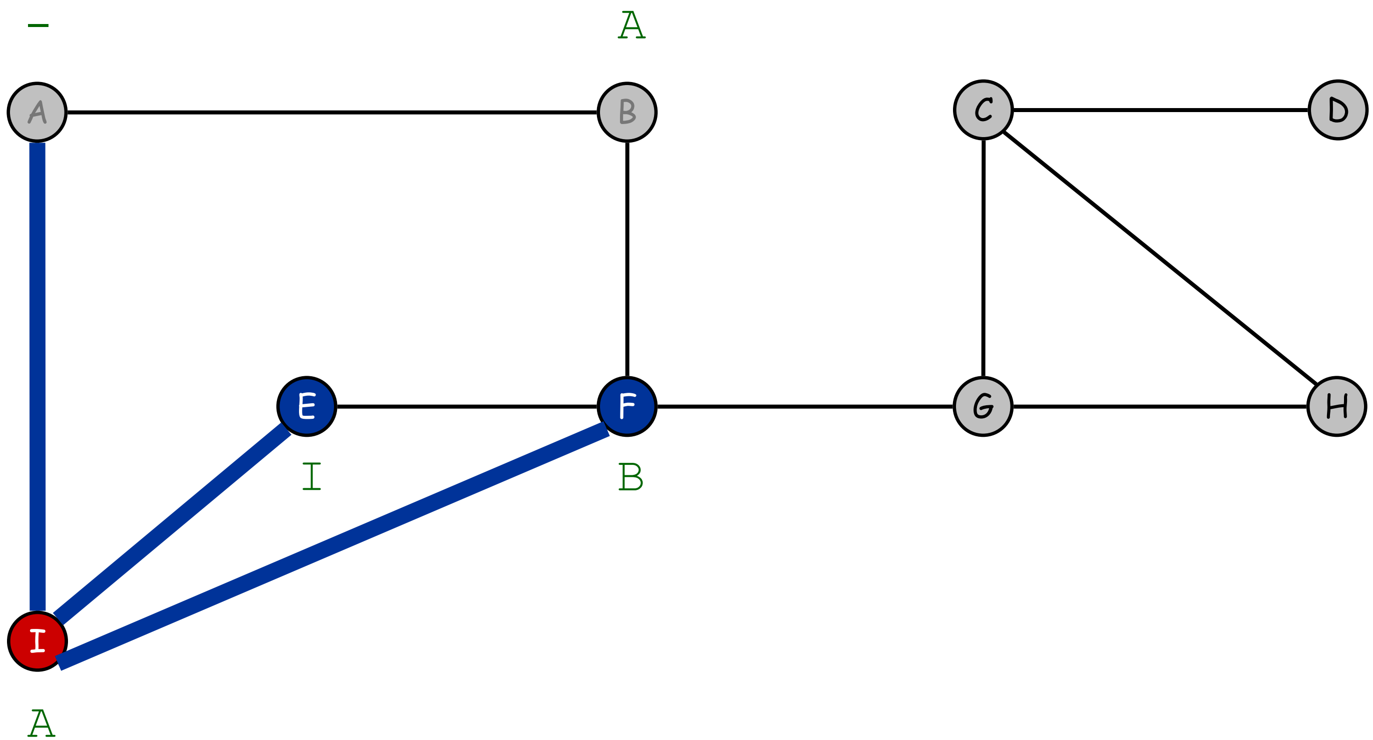
# Breadth First Search




visit neighbors of I


FIFO Queue

# Breadth First Search



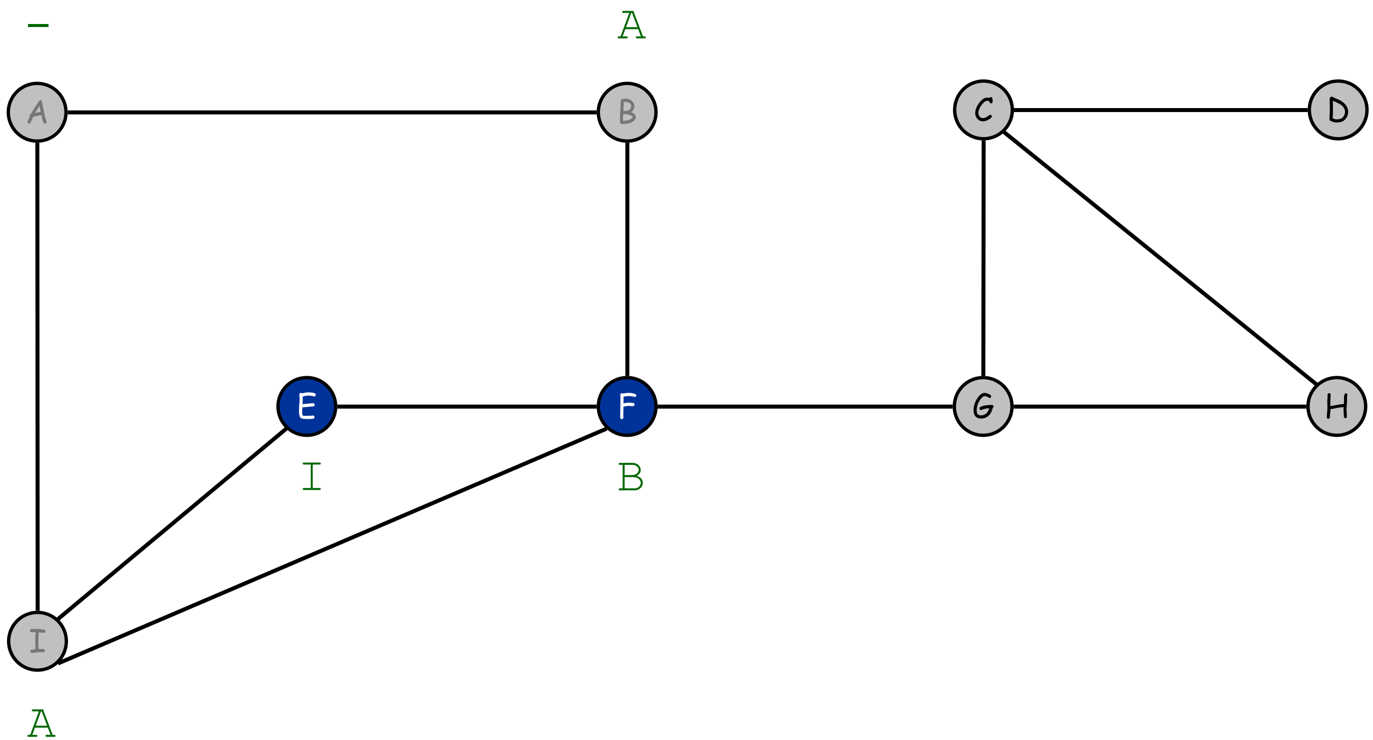

F already discovered

front

F E

FIFO Queue

# Breadth First Search



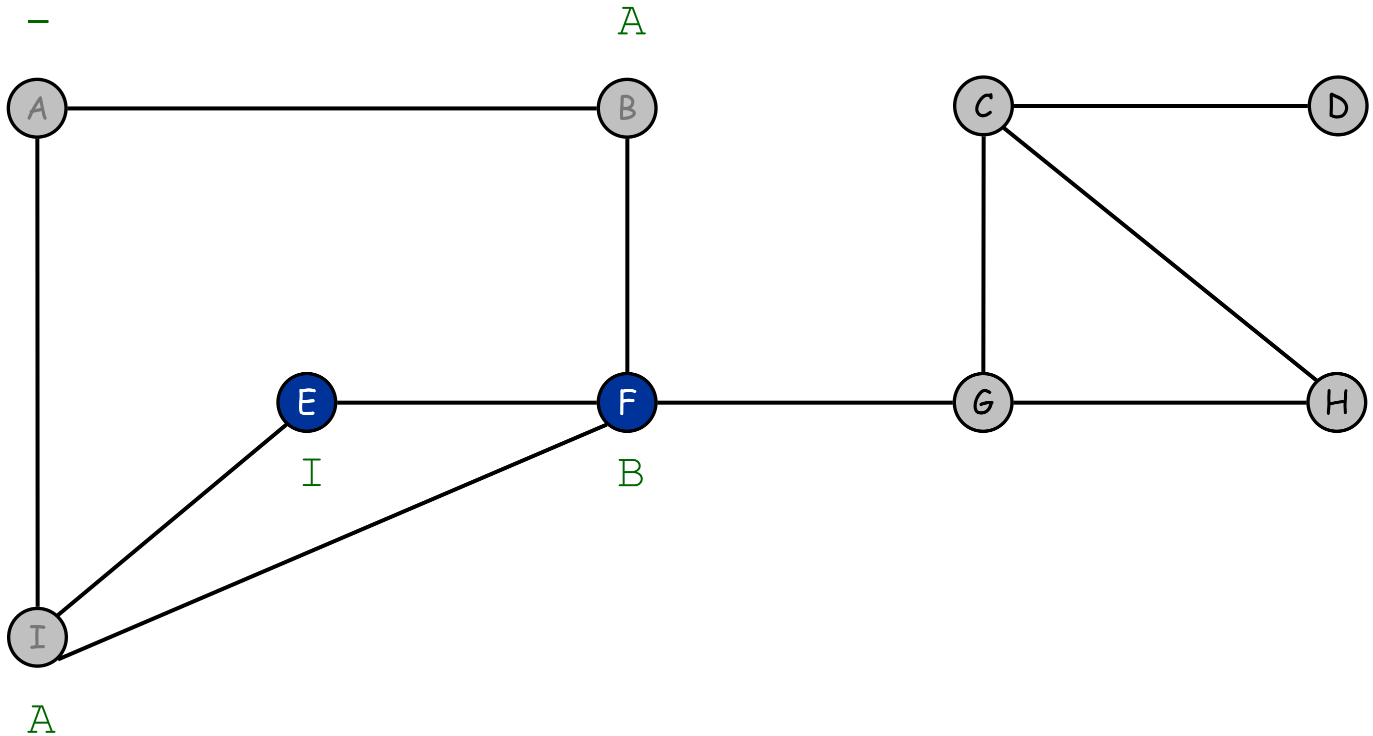

I finished

front

F E

FIFO Queue

# Breadth First Search

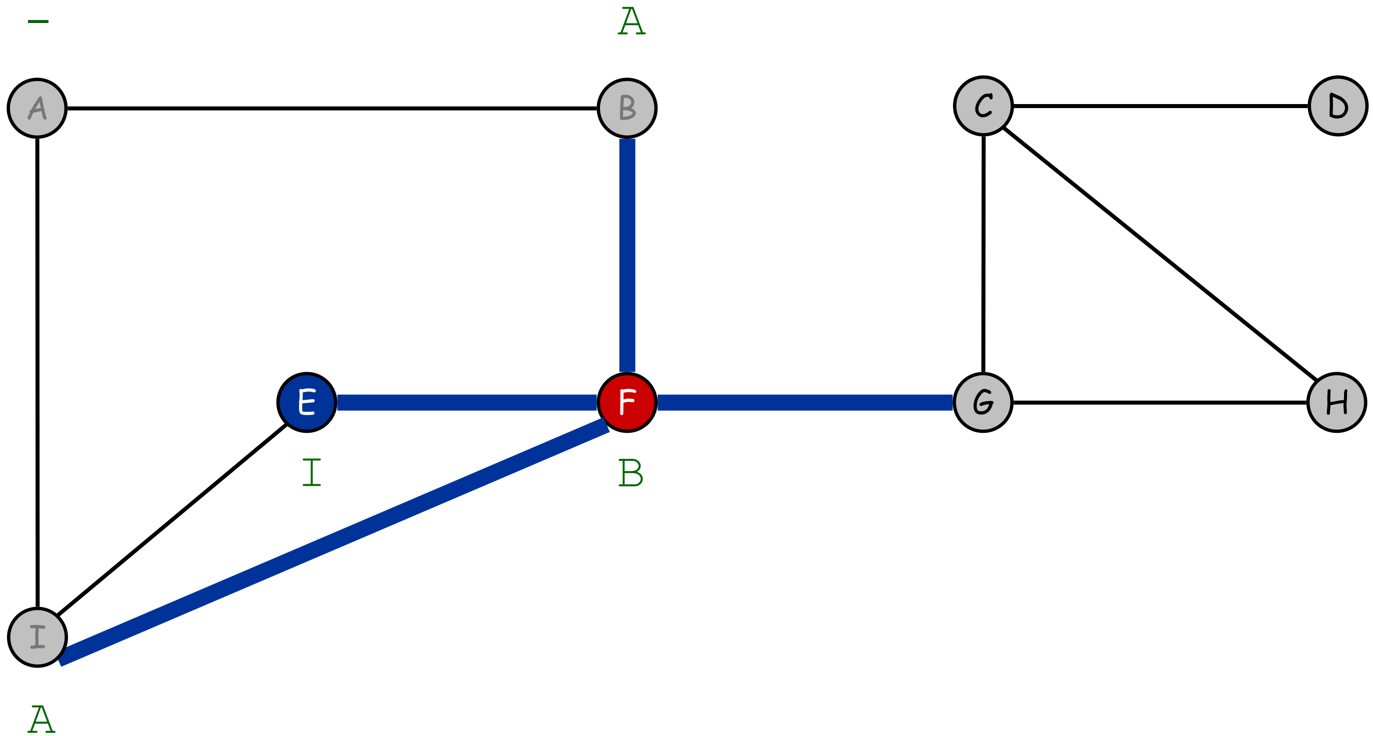



dequeue next vertex

front F E

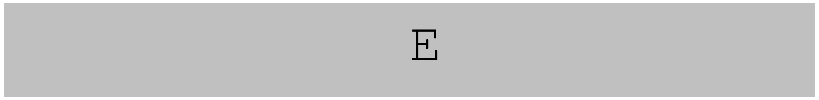
FIFO Queue

# Breadth First Search



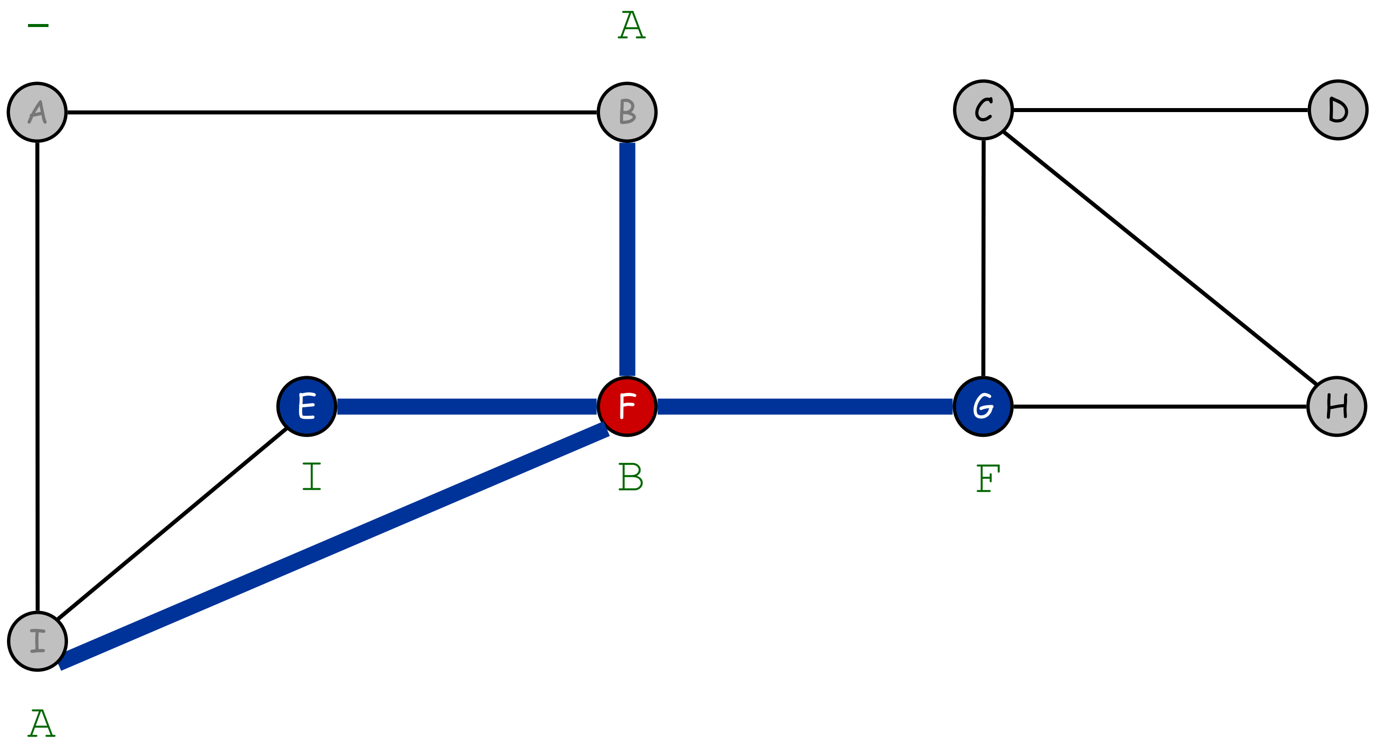

visit neighbors of F

front




FIFO Queue

# Breadth First Search



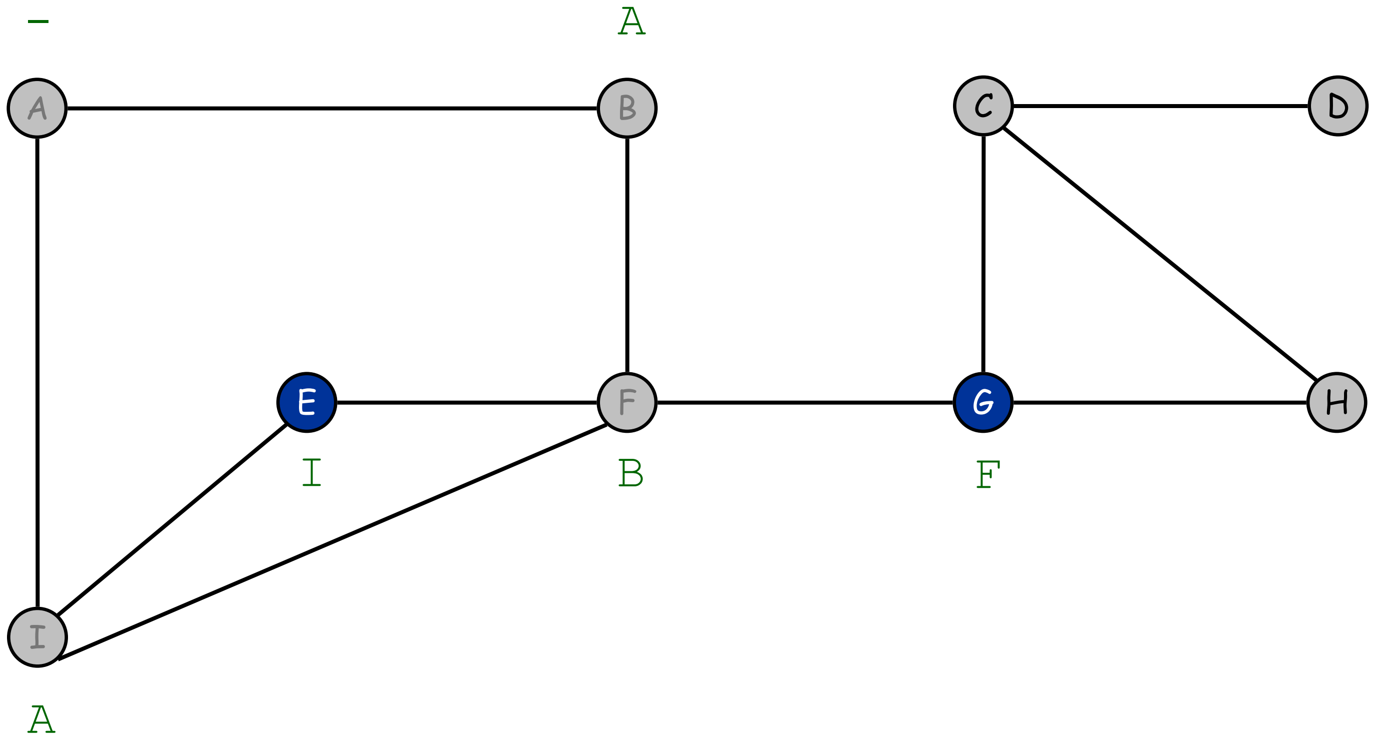

G discovered

front

E G

FIFO Queue

# Breadth First Search



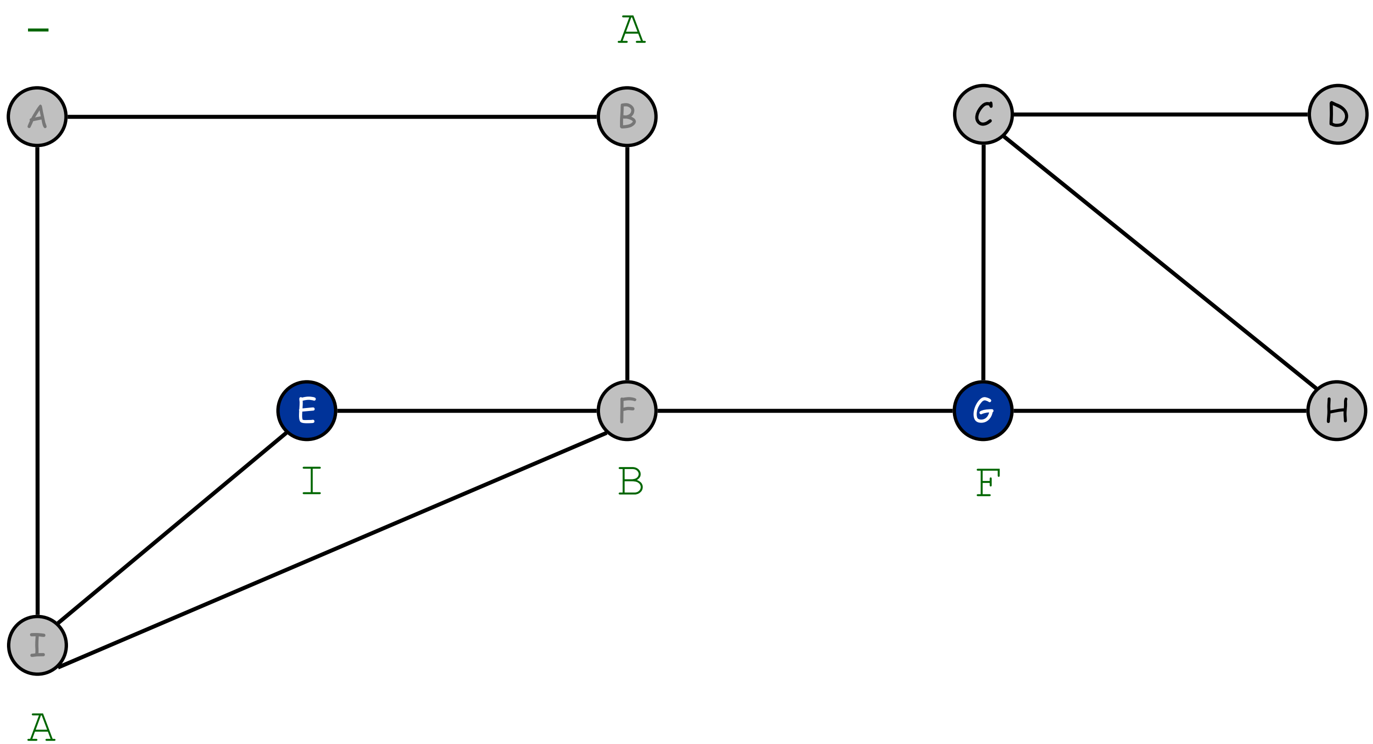

F finished

front

E G

FIFO Queue

# Breadth First Search

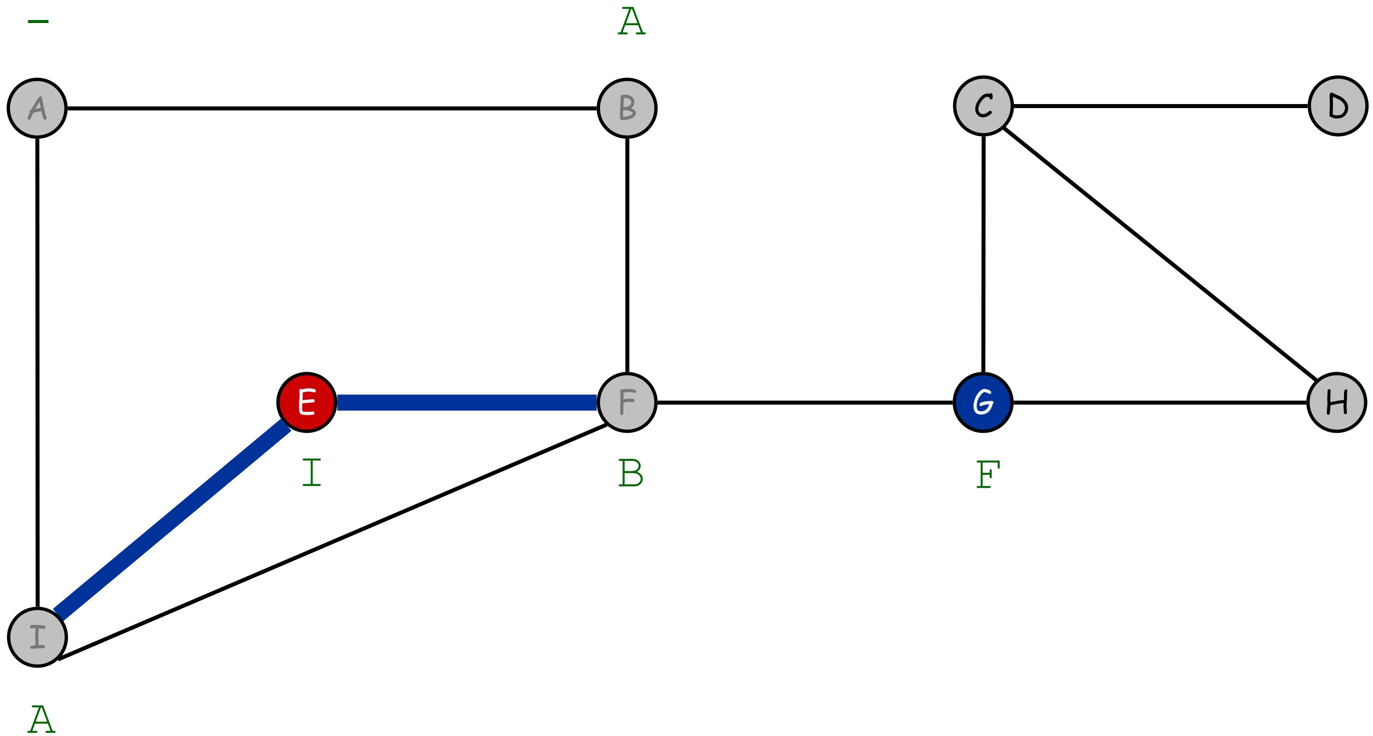



dequeue next vertex

front | E G

FIFO Queue

# Breadth First Search

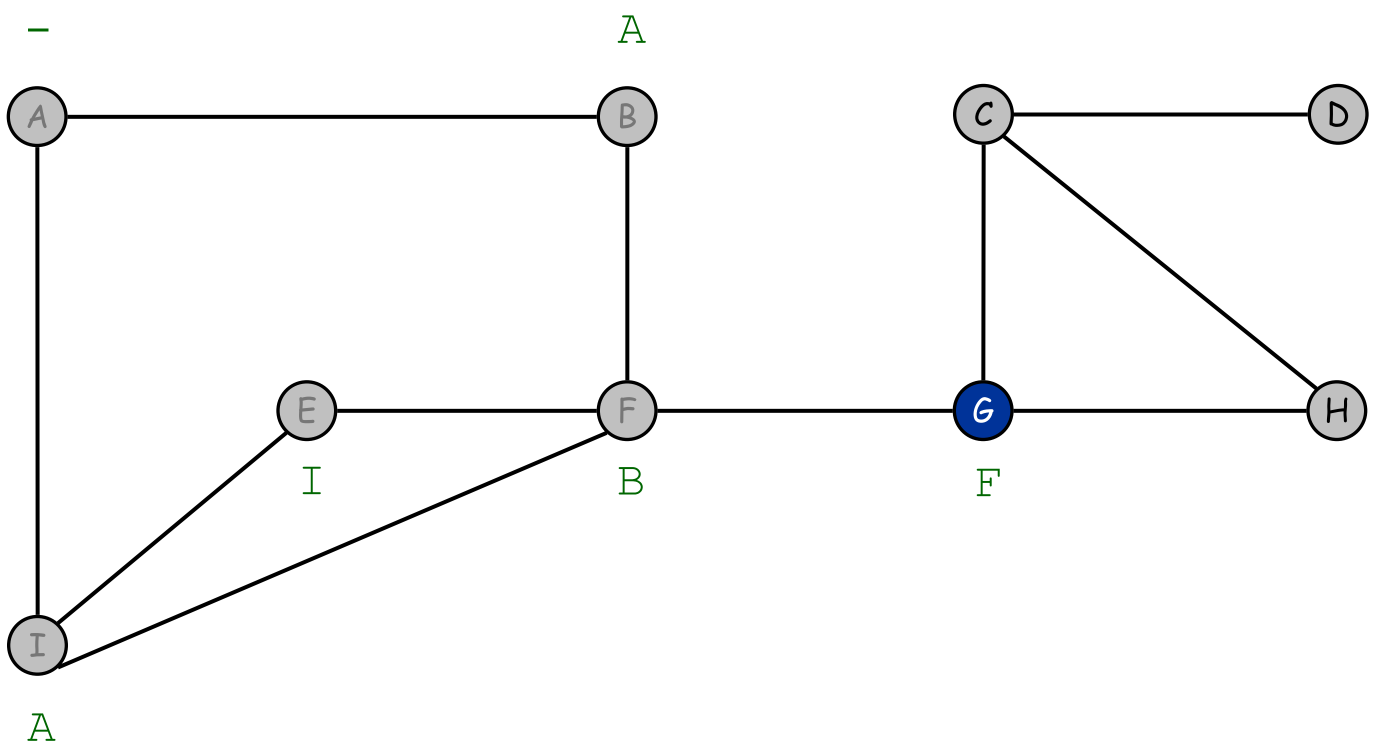



visit neighbors of E

front G

FIFO Queue

# Breadth First Search

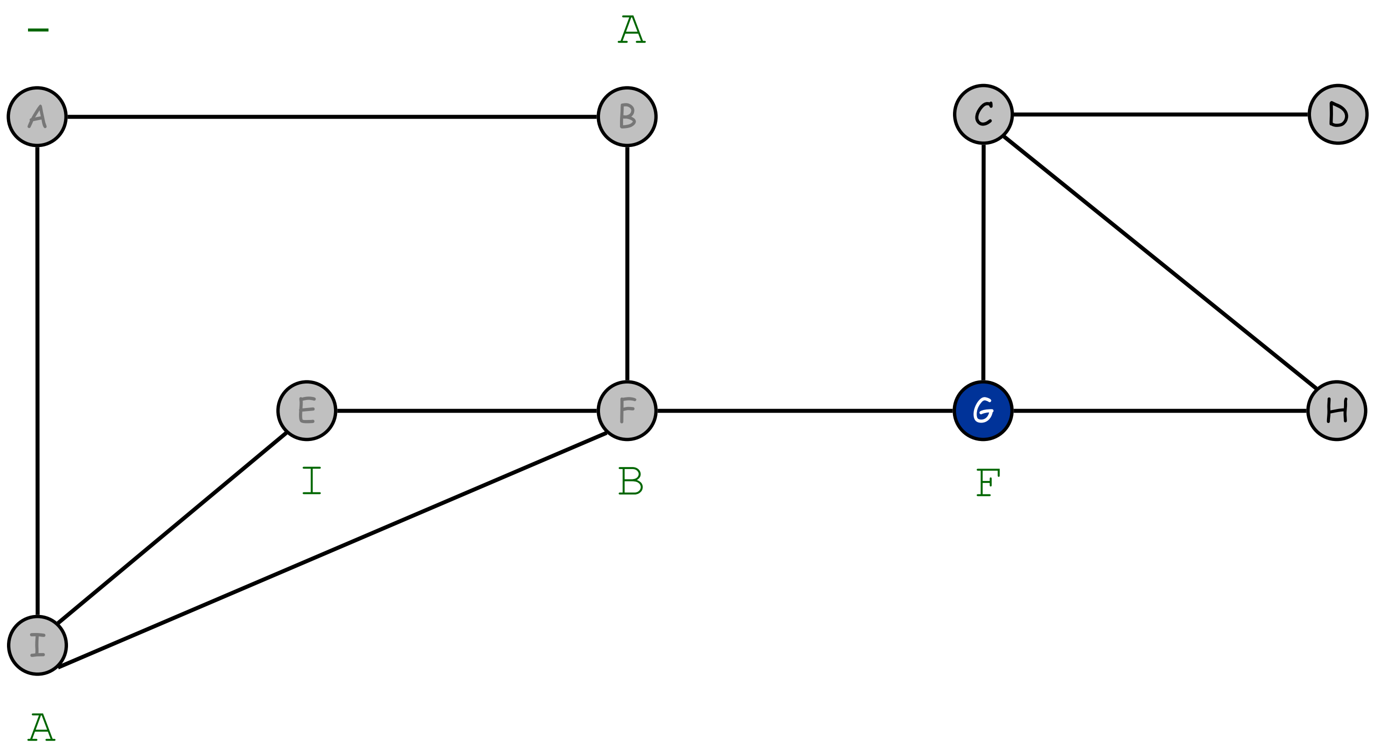



E finished

front G

FIFO Queue

# Breadth First Search



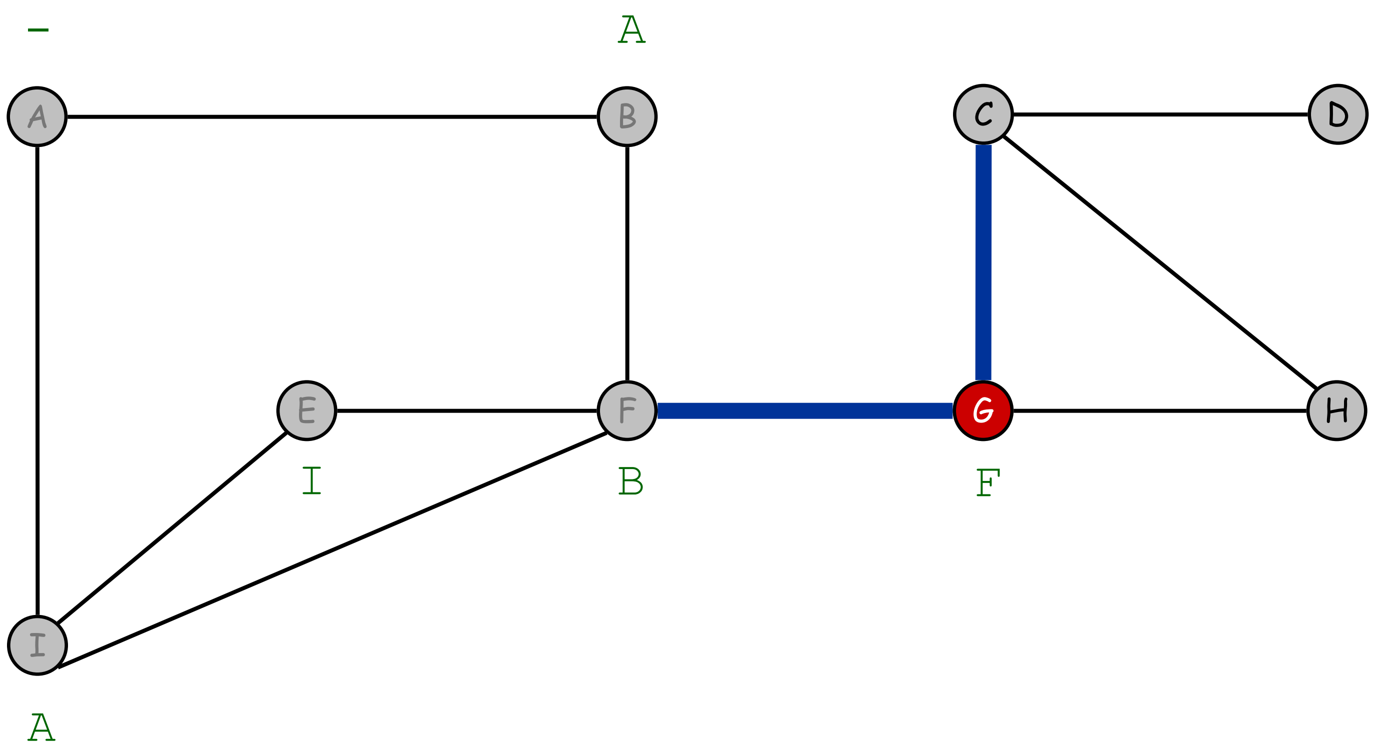

dequeue next vertex

front

G

FIFO Queue

# Breadth First Search

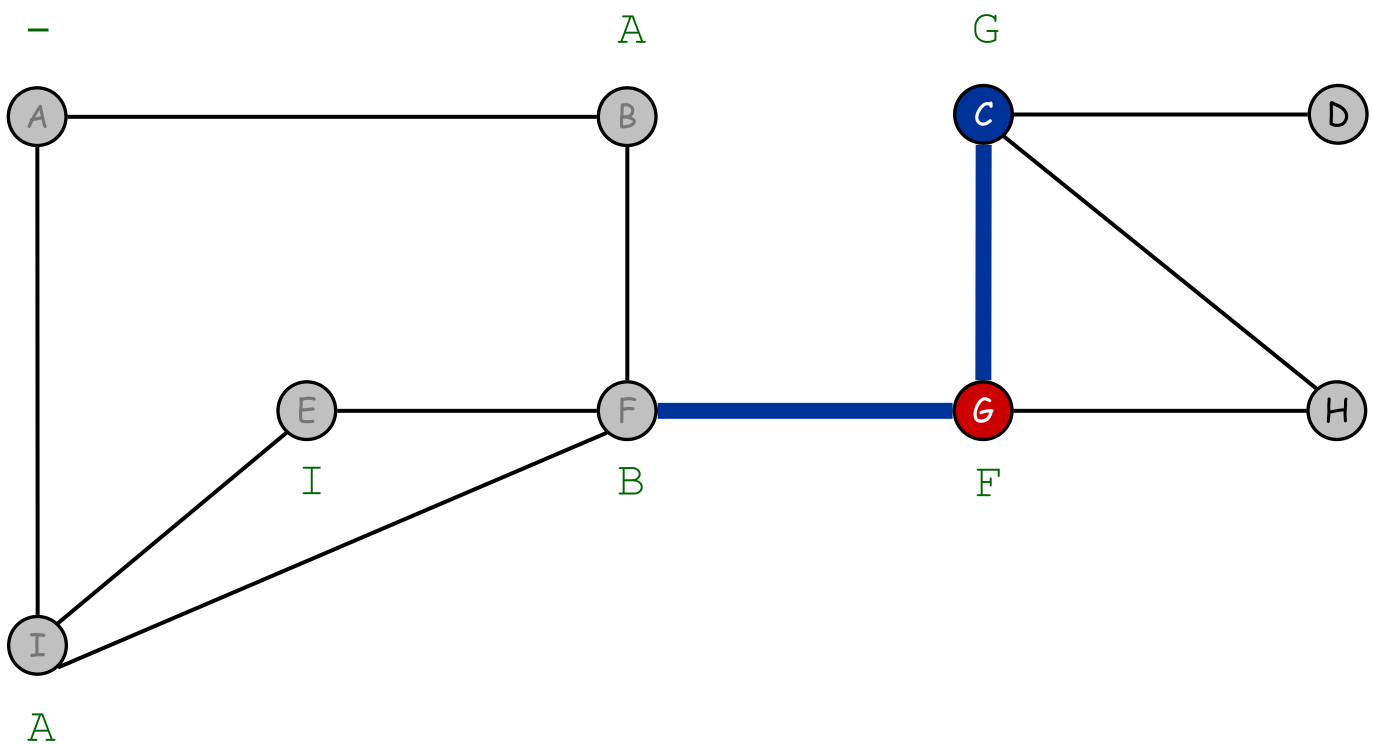



visit neighbors of G

front

FIFO Queue

# Breadth First Search

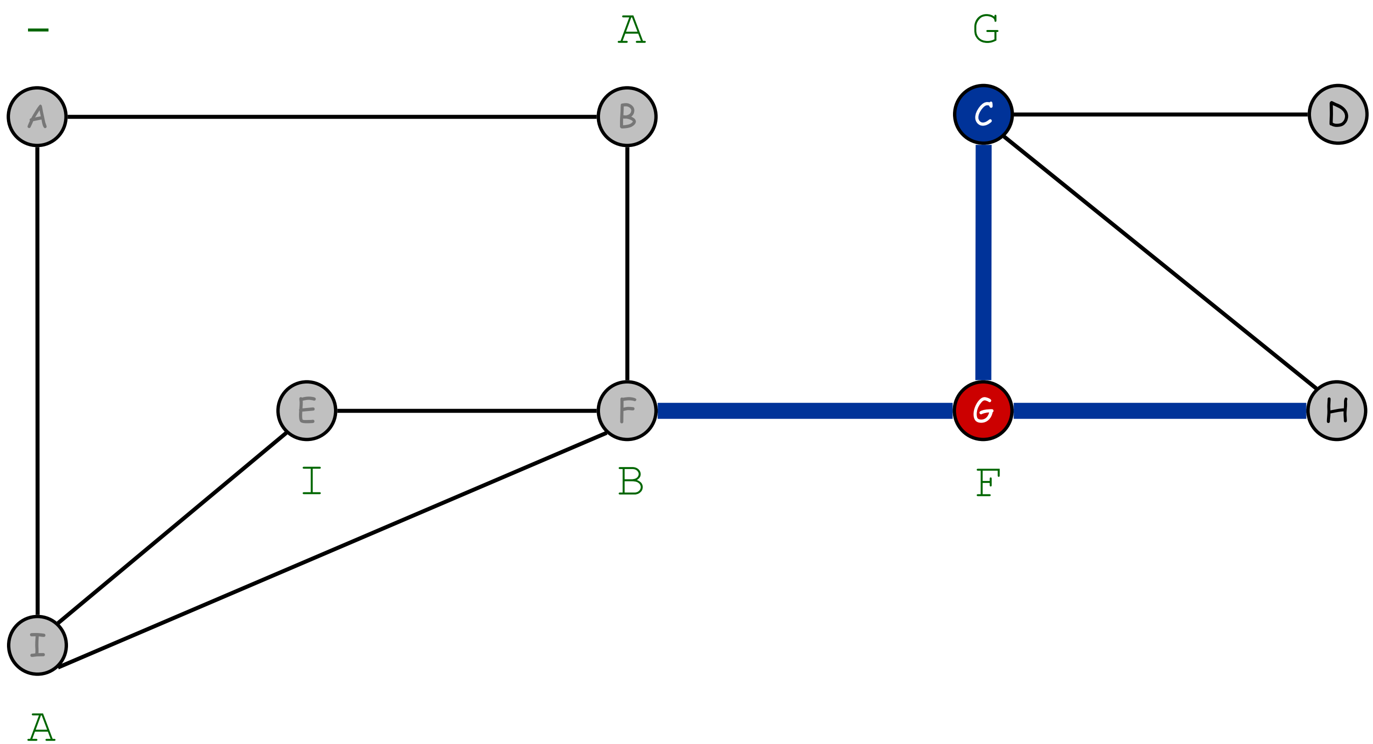



C discovered

front C

FIFO Queue

# Breadth First Search

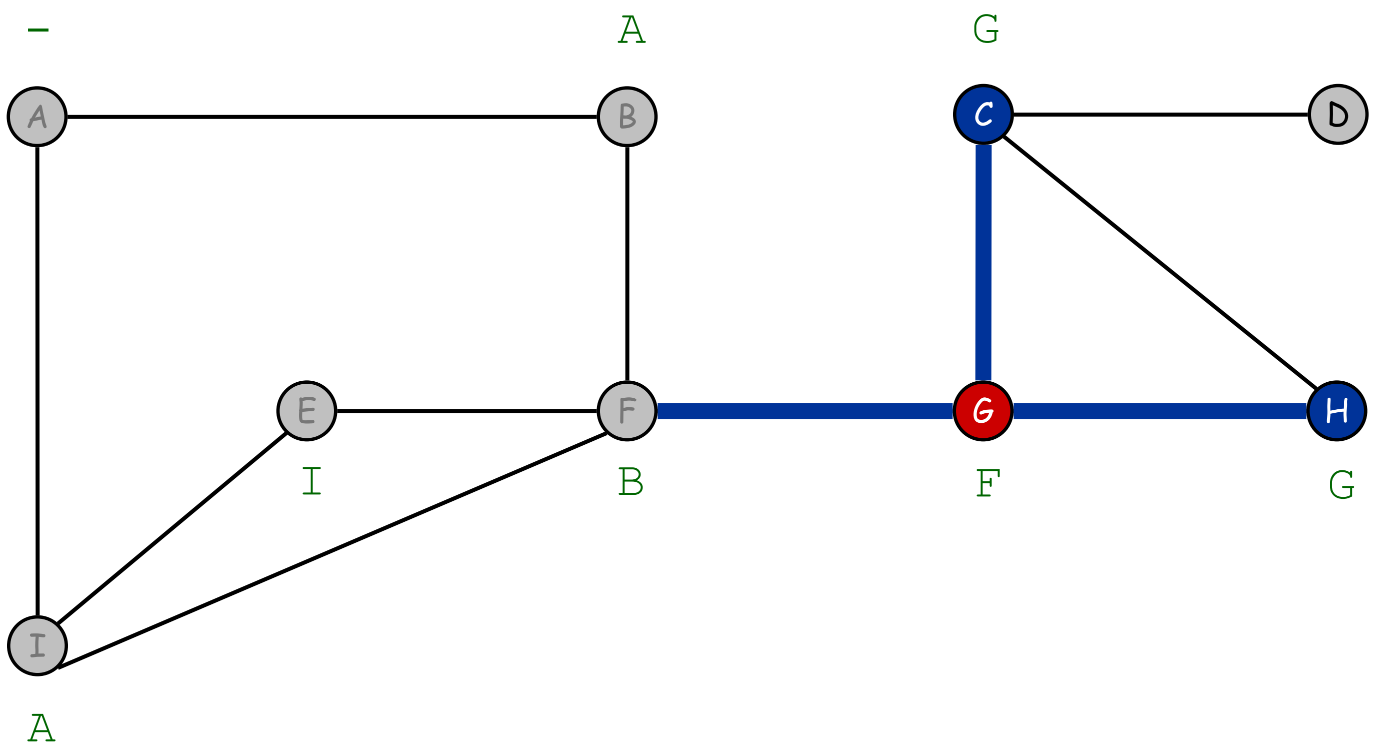



visit neighbors of G

front | C

FIFO Queue

# Breadth First Search



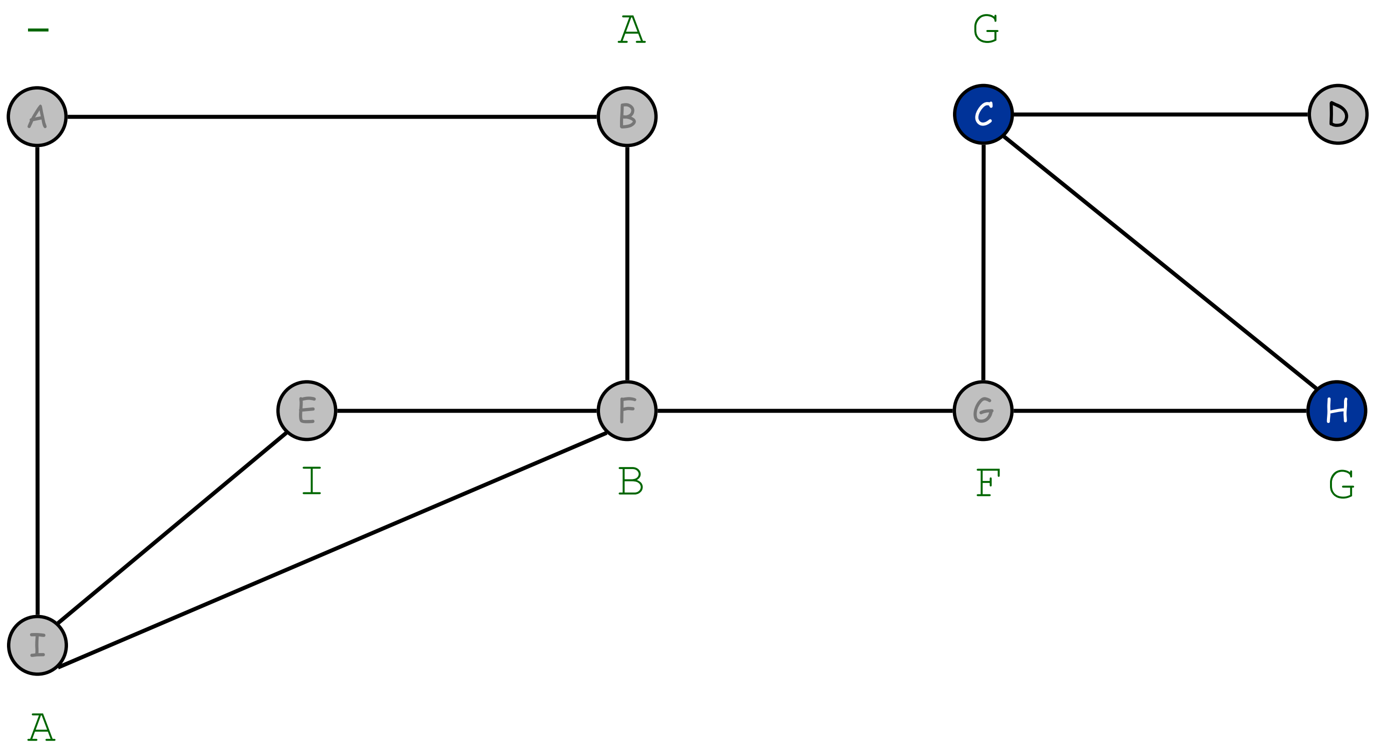

H discovered

front

C H

FIFO Queue

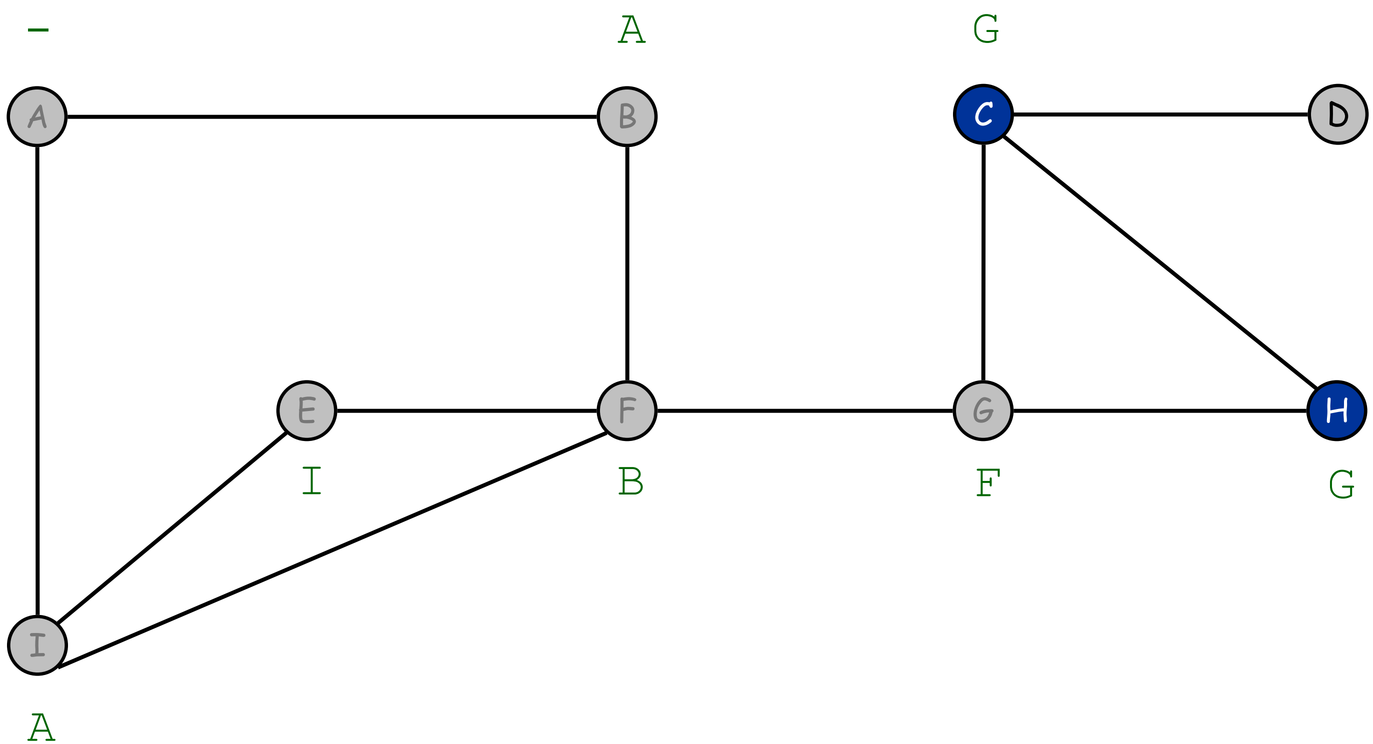
# Breadth First Search




G finished

front
C H
FIFO Queue

# Breadth First Search

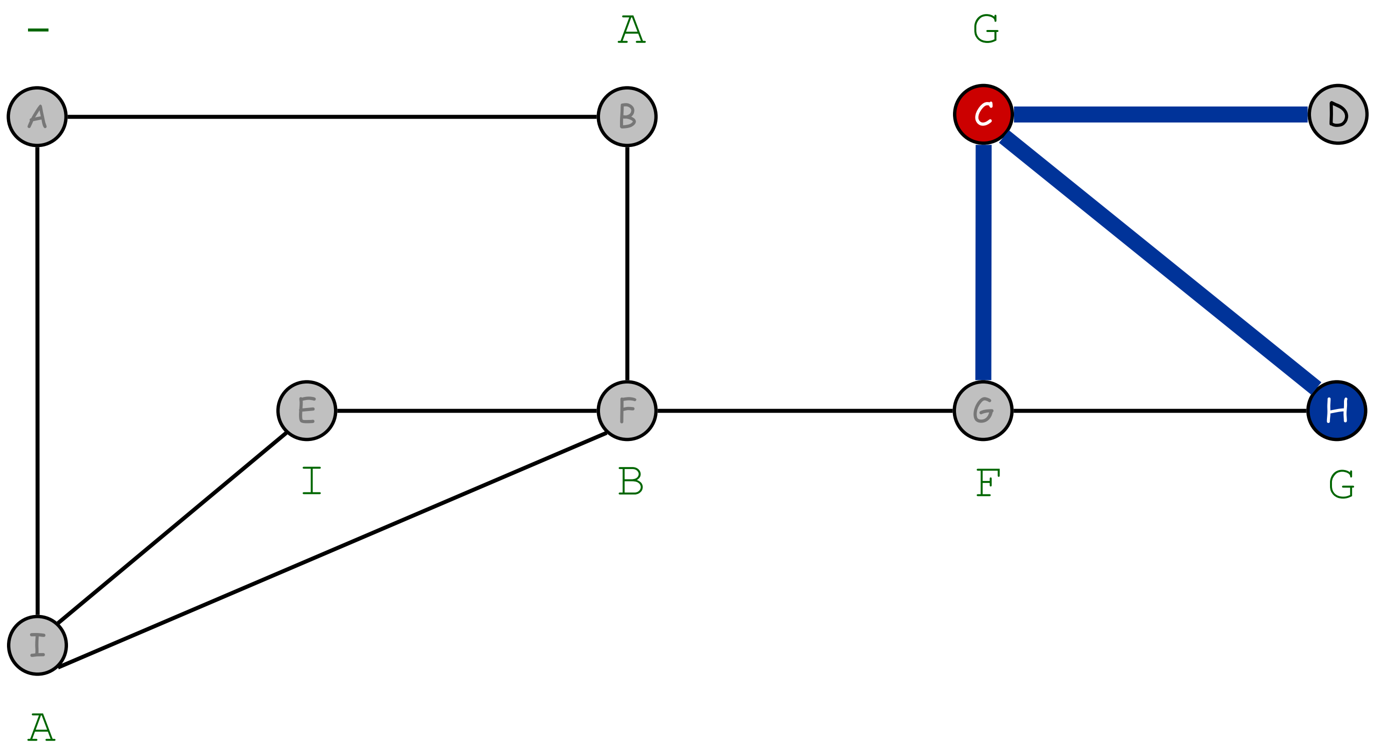



dequeue next vertex

front C H

FIFO Queue

# Breadth First Search

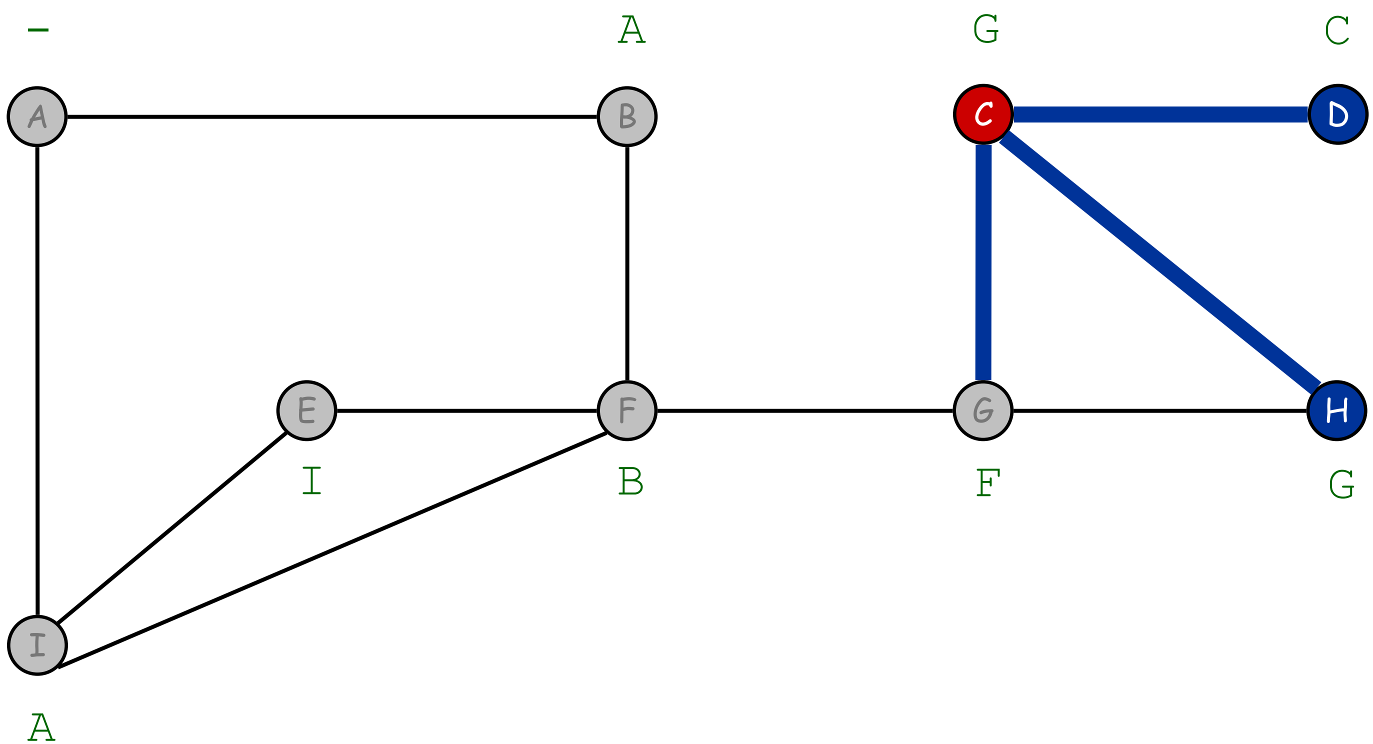



visit neighbors of C

front H

FIFO Queue

# Breadth First Search

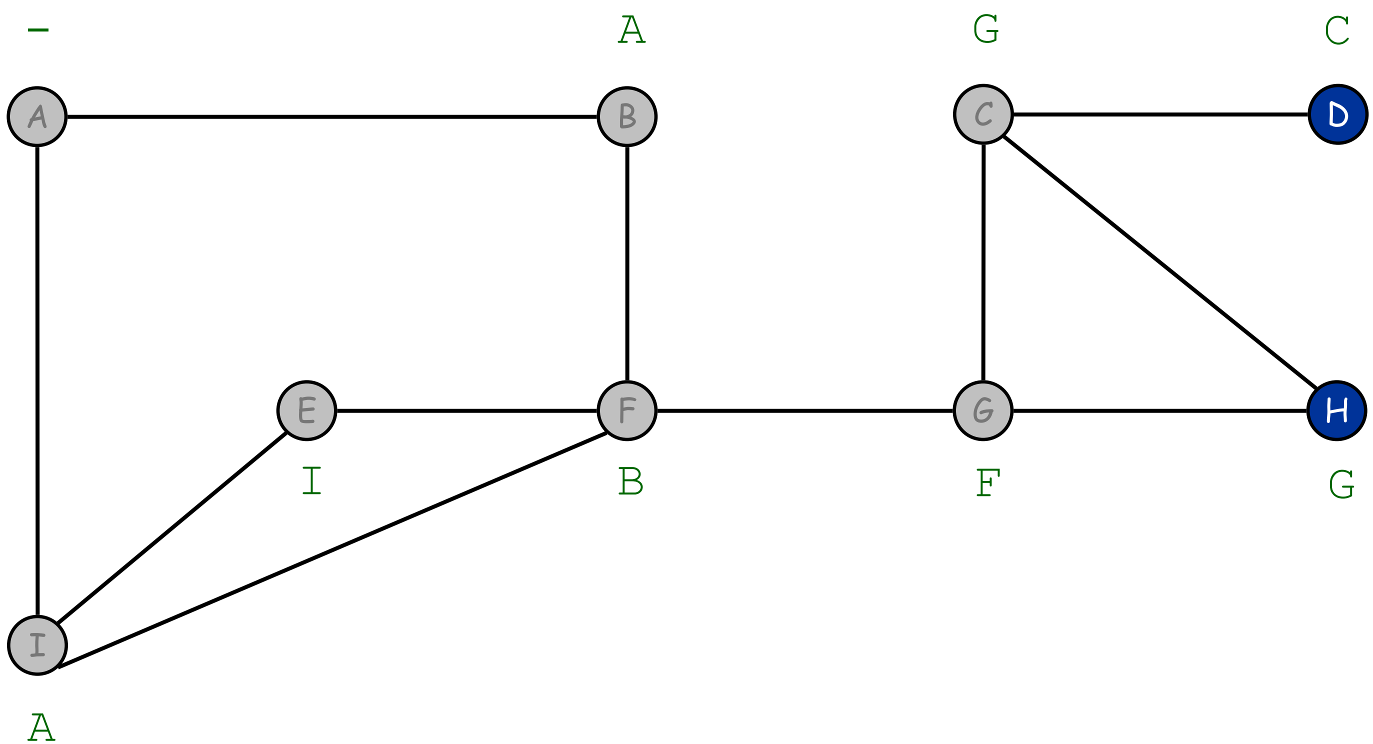



D discovered

front H D

FIFO Queue

# Breadth First Search

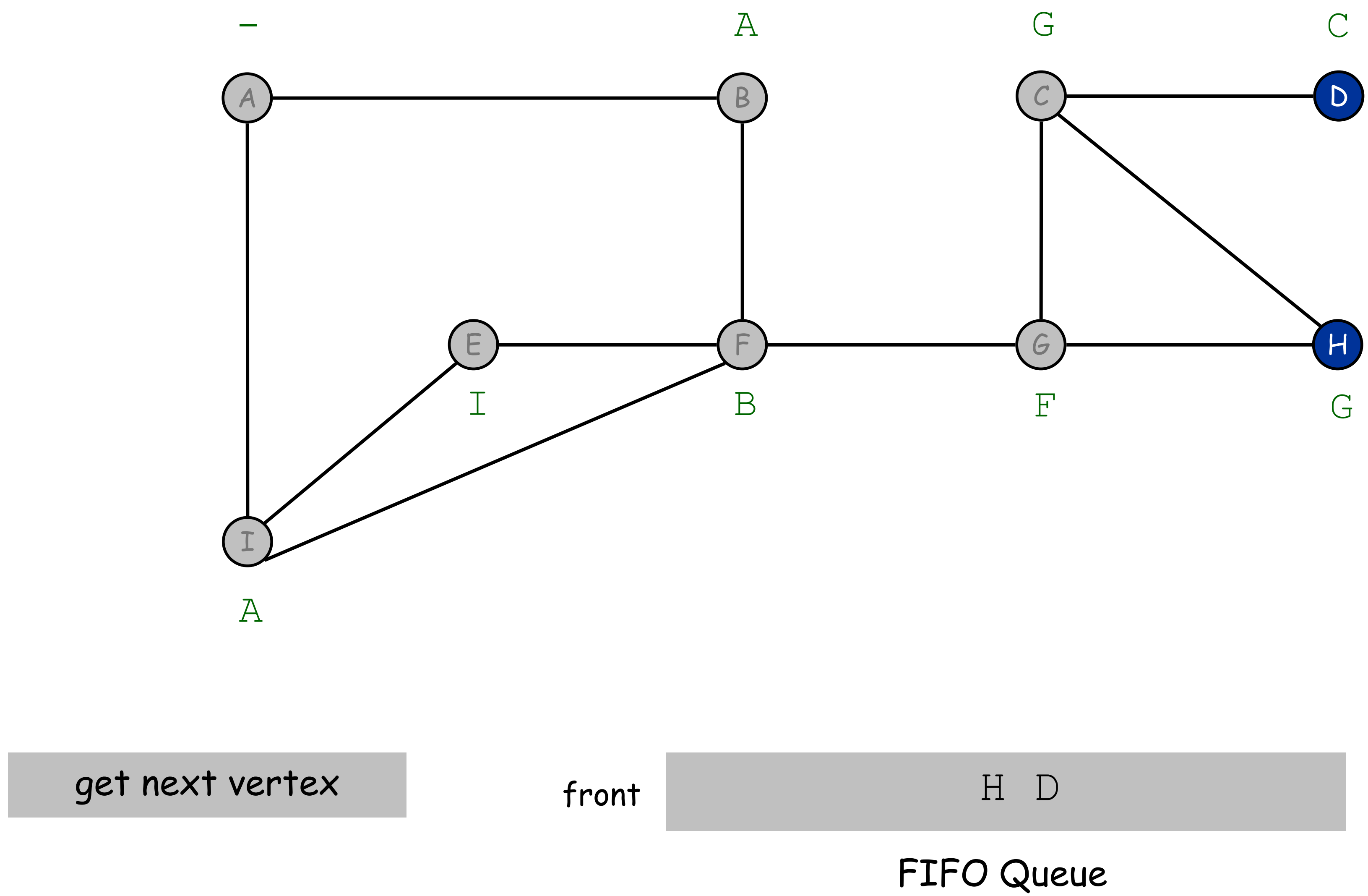



C finished

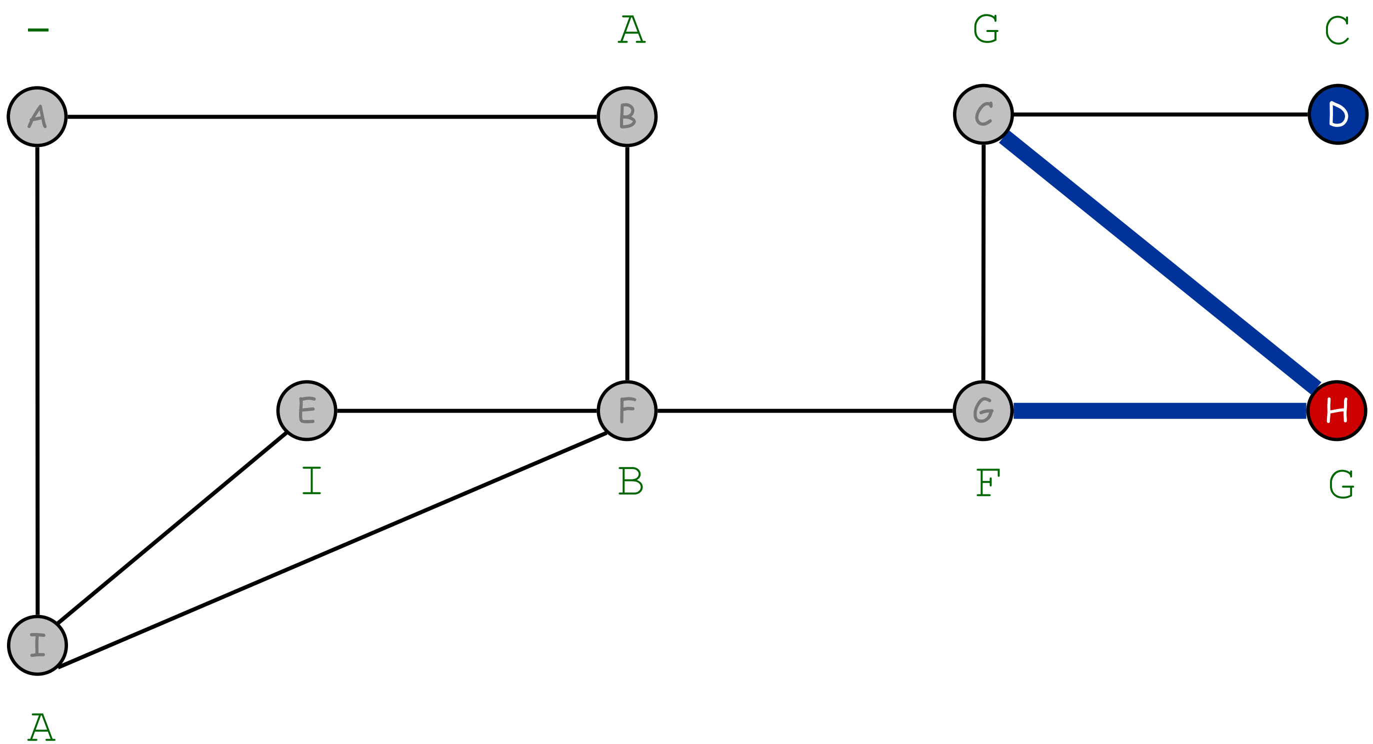
front H D

FIFO Queue

# Breadth First Search



# Breadth First Search

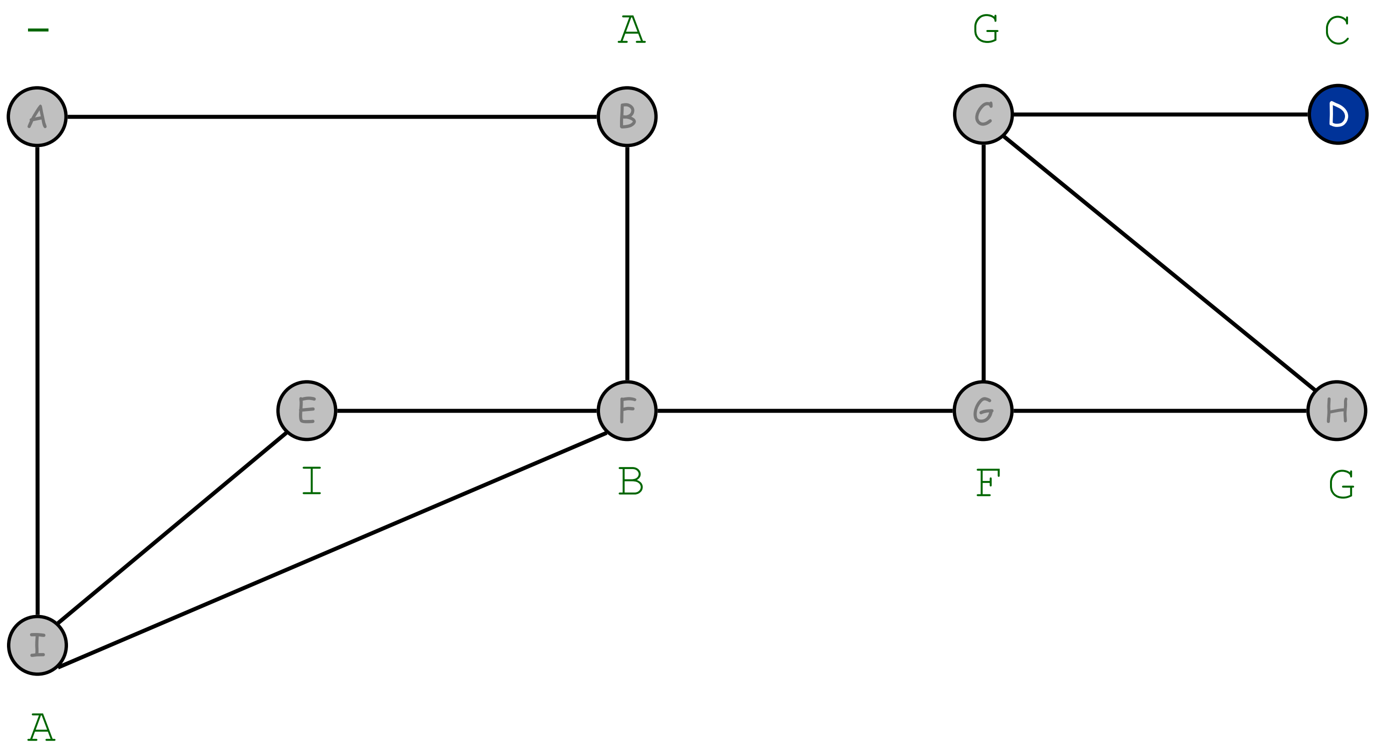



visit neighbors of H

front D

FIFO Queue

# Breadth First Search

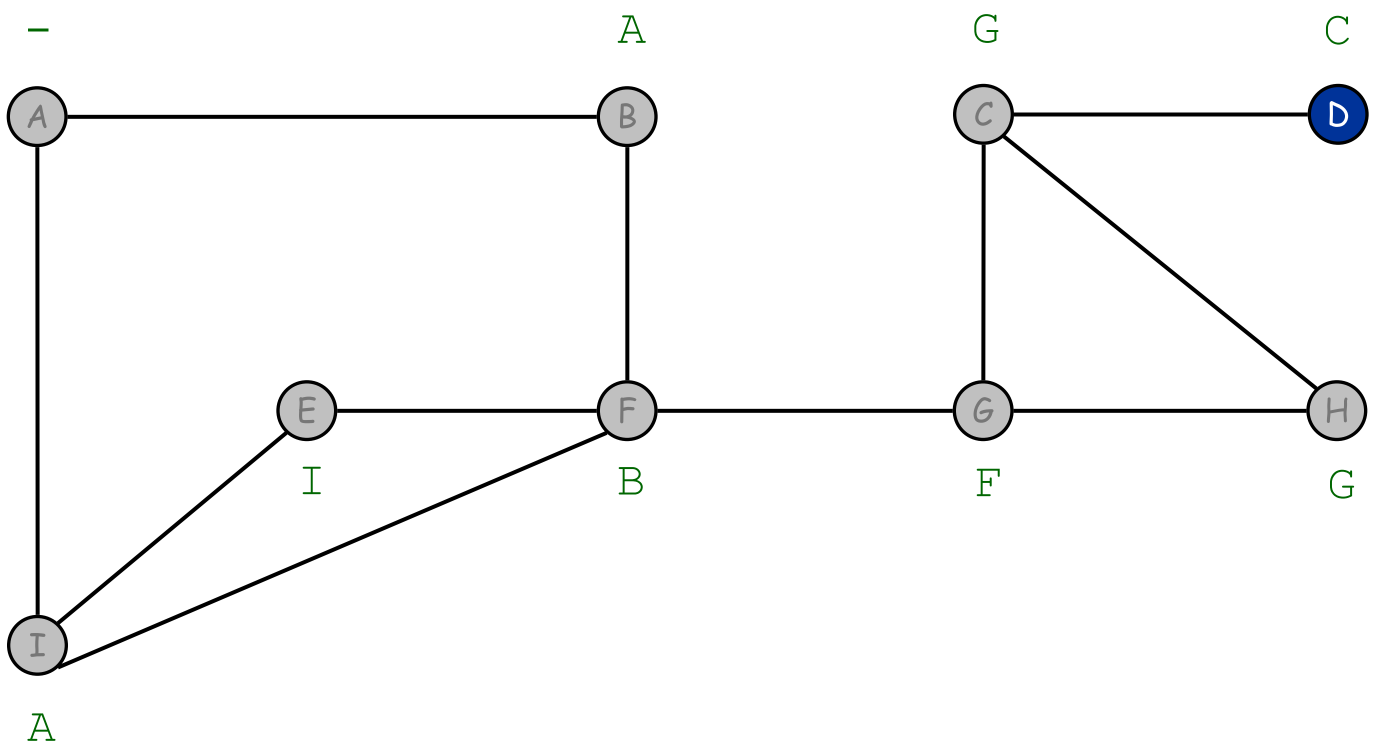



finished H

front D

FIFO Queue

# Breadth First Search

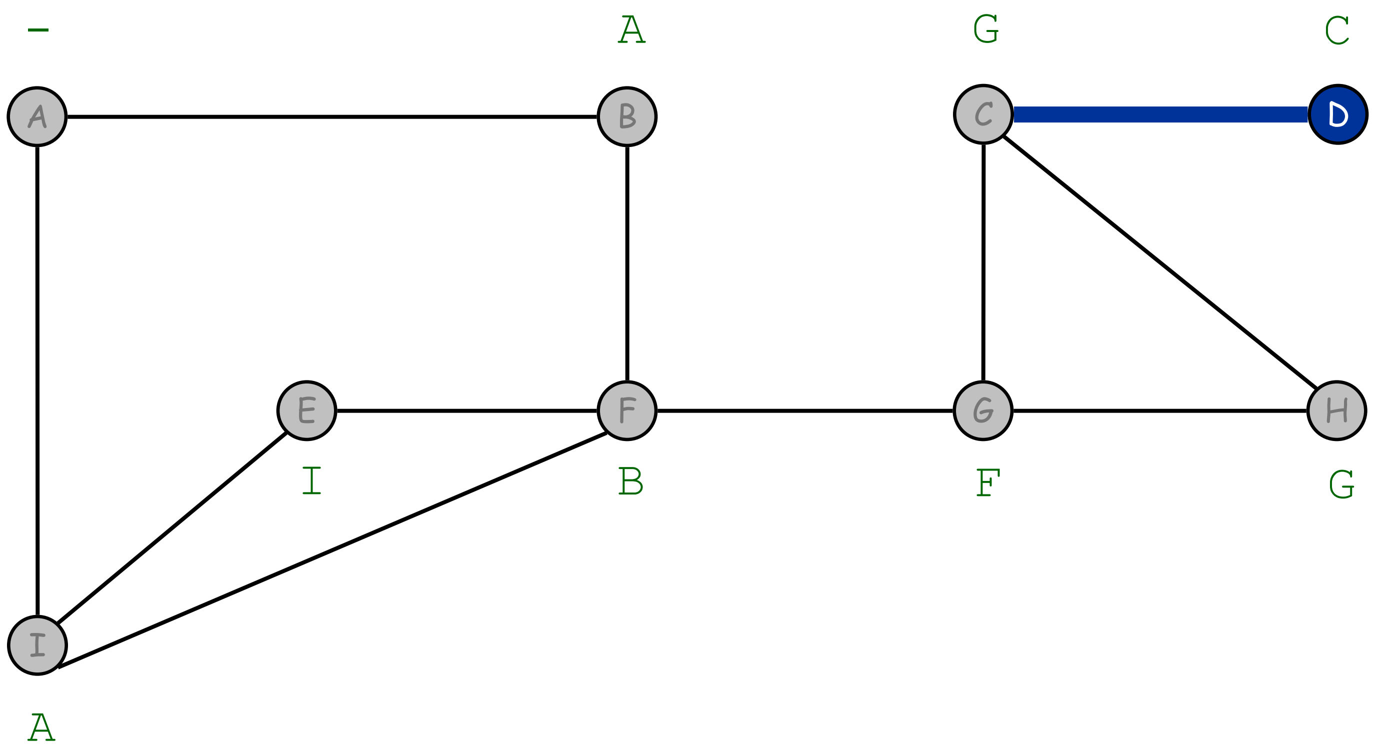



dequeue next vertex

front D

FIFO Queue

# Breadth First Search

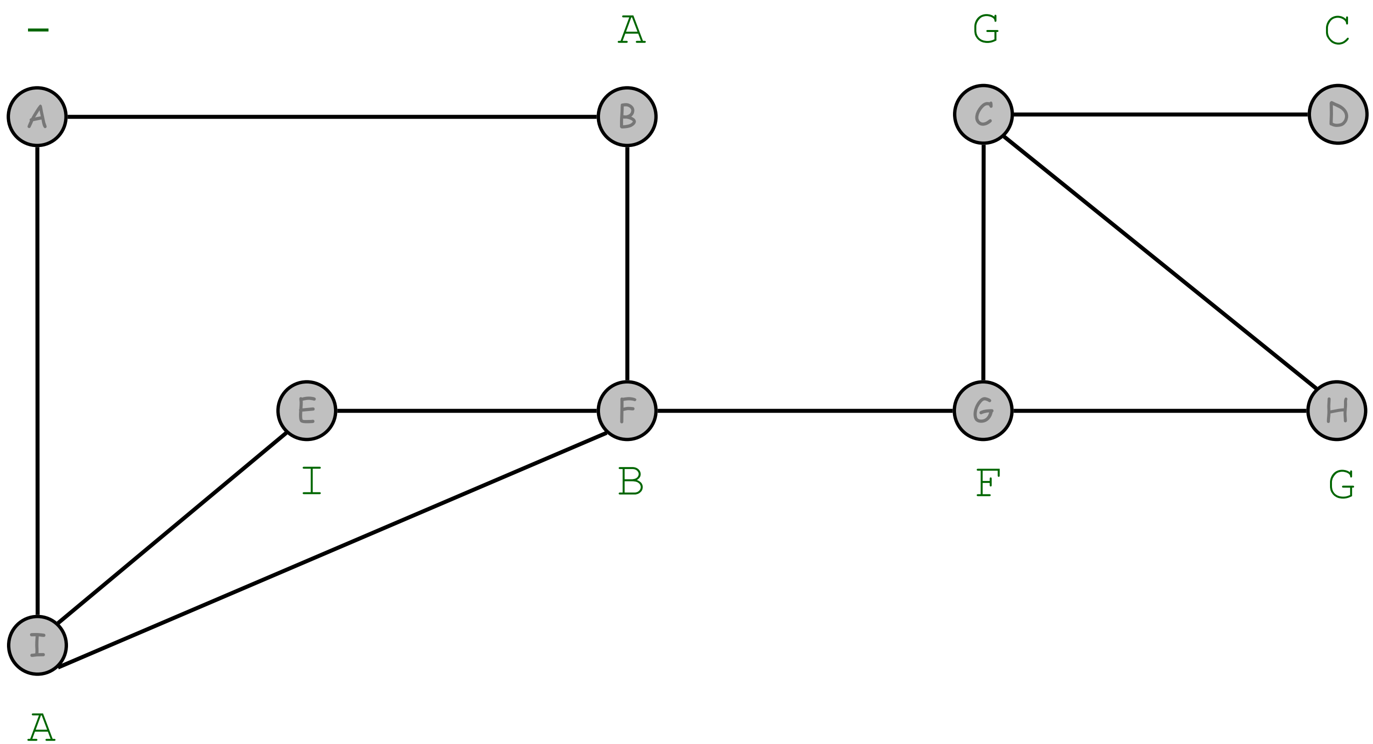



visit neighbors of D

front

FIFO Queue

# Breadth First Search

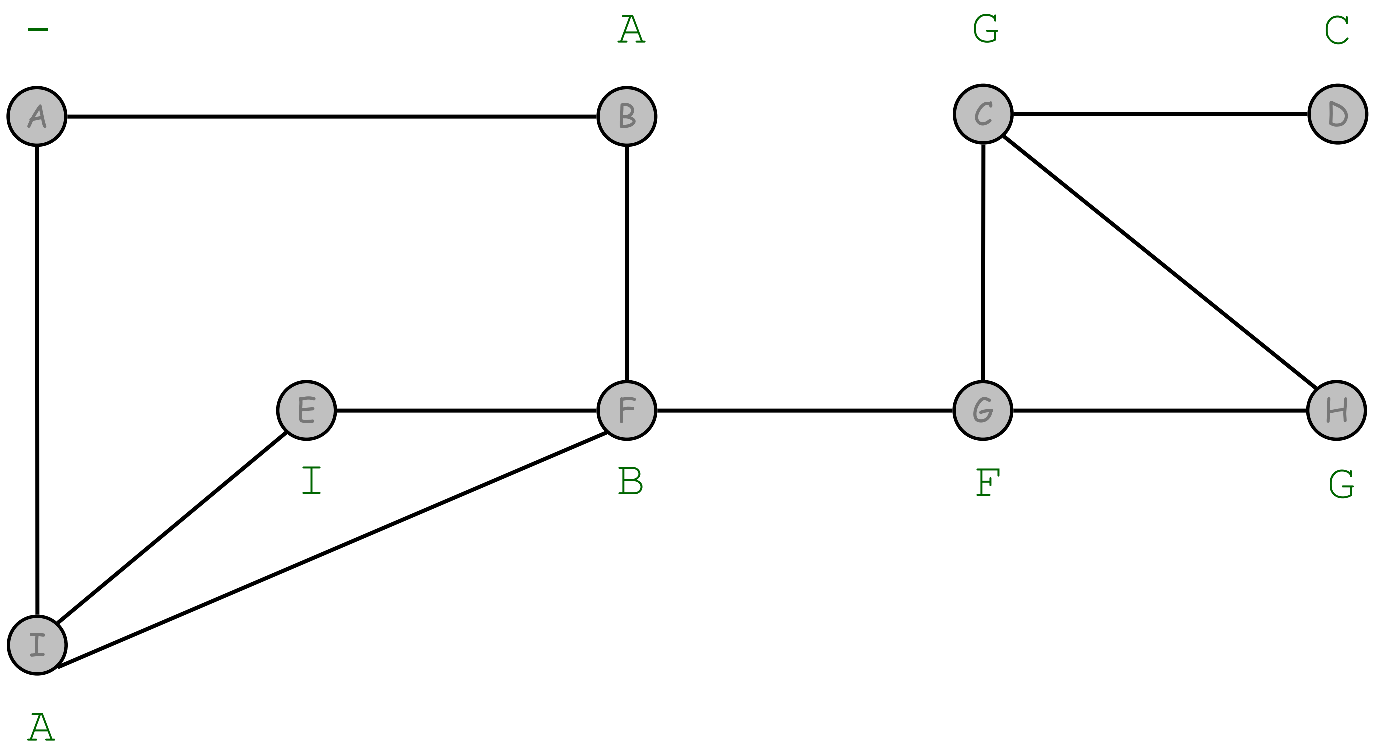



D finished

front

FIFO Queue

# Breadth First Search

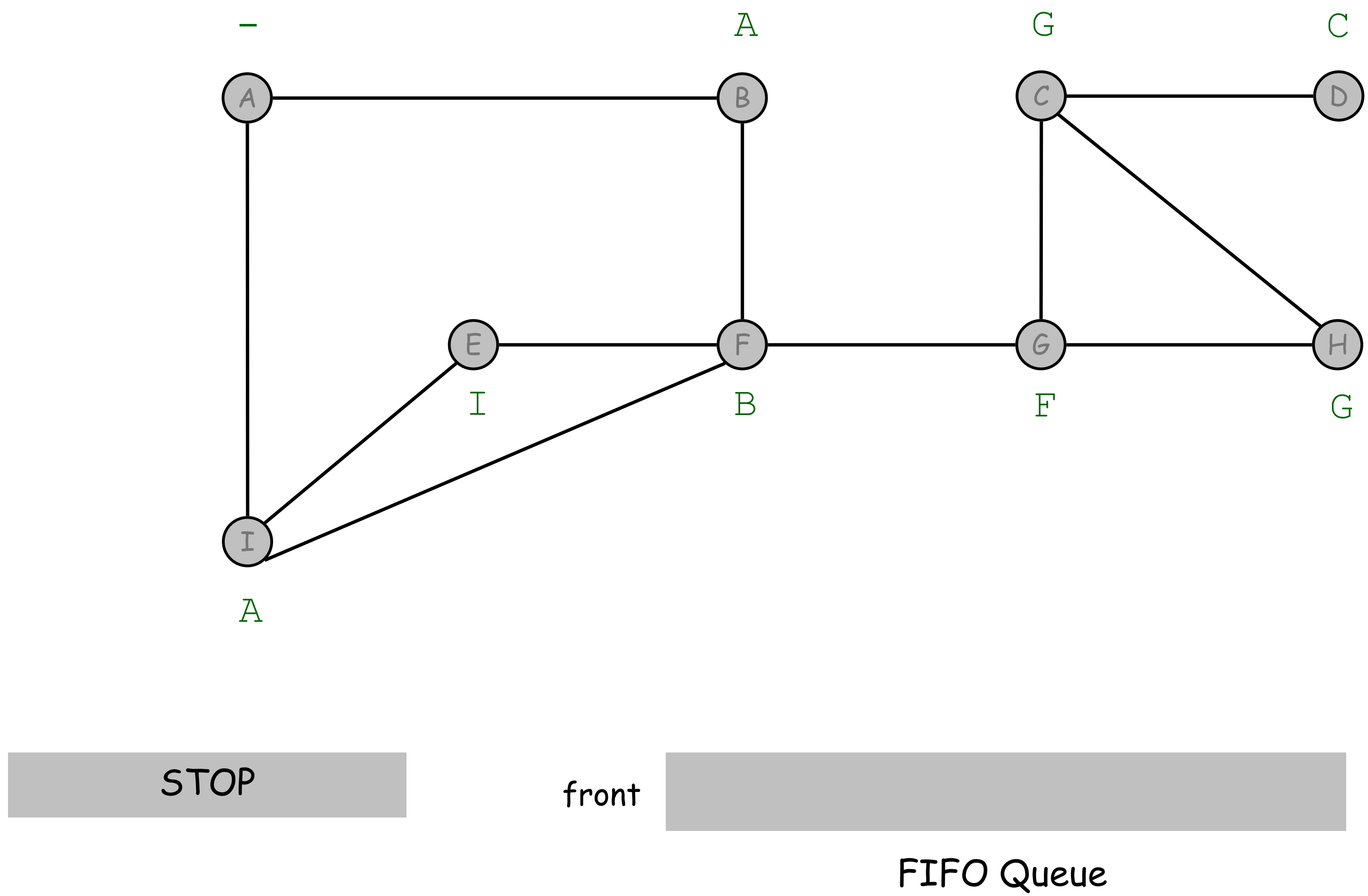



dequeue next vertex

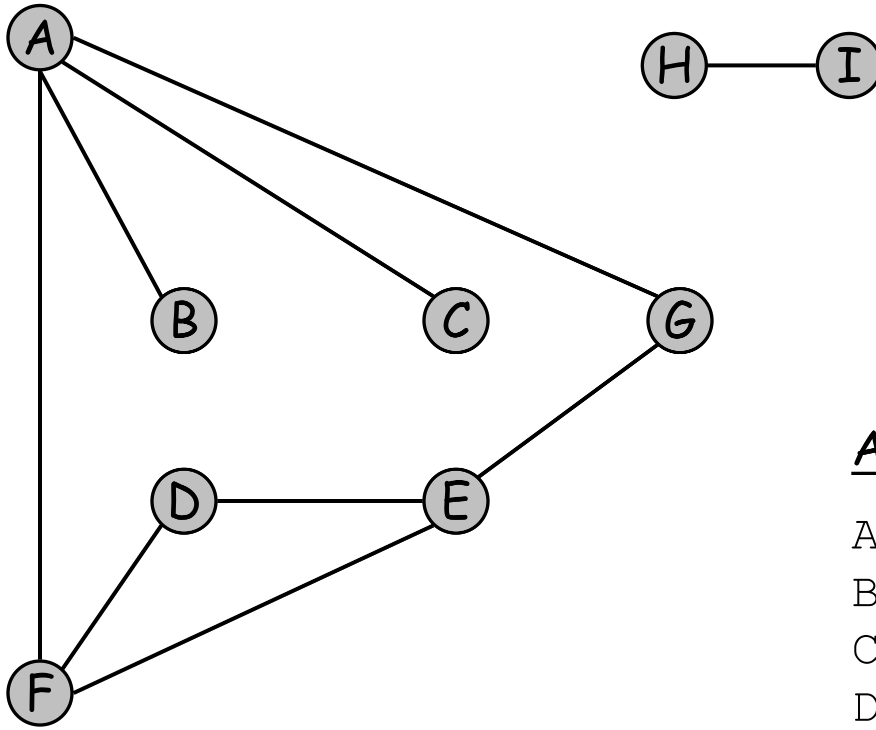
front

FIFO Queue

# Breadth First Search



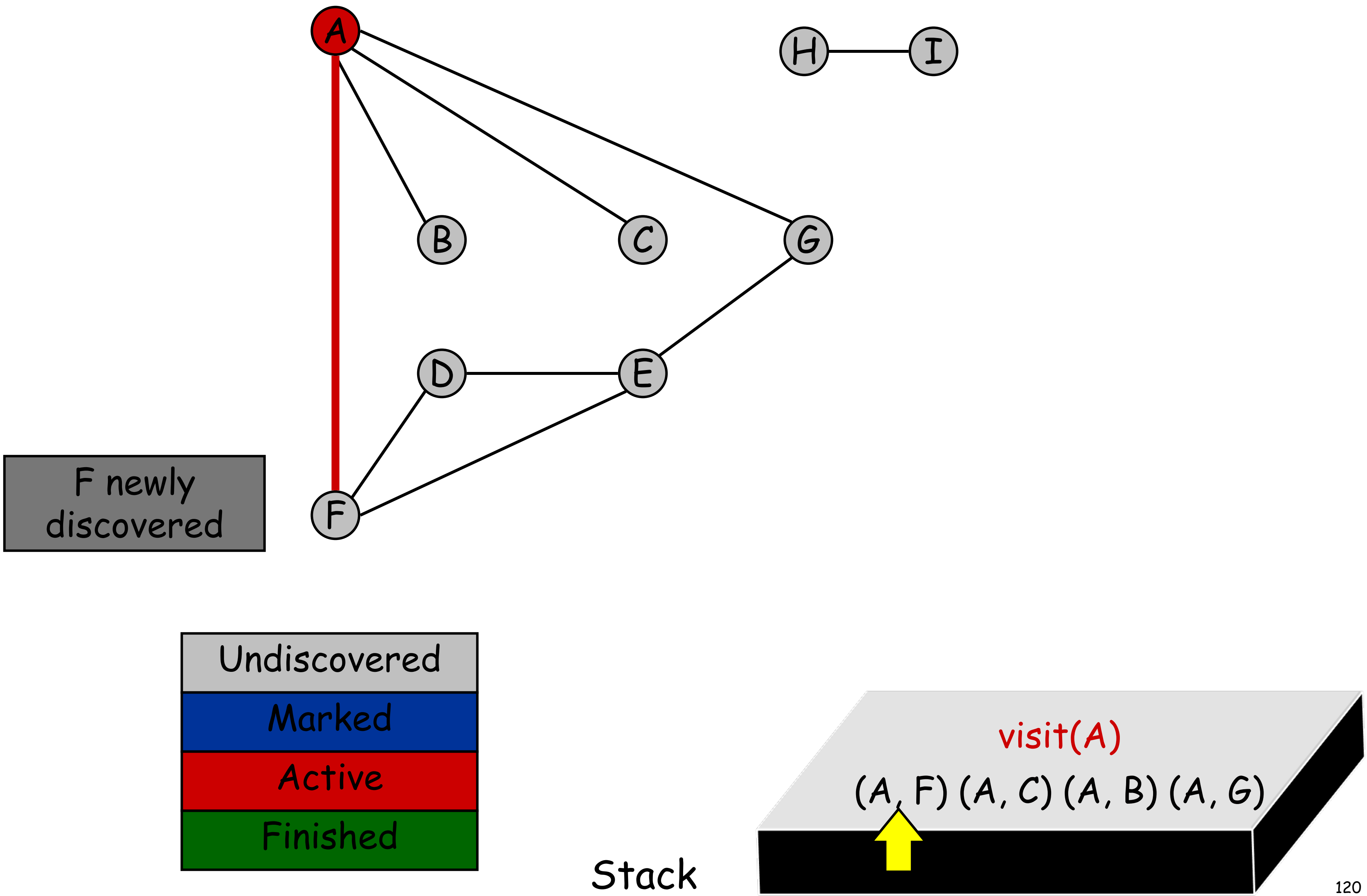
# Question 6: Undirected Depth First Search




<u>Adjacency Lists</u>

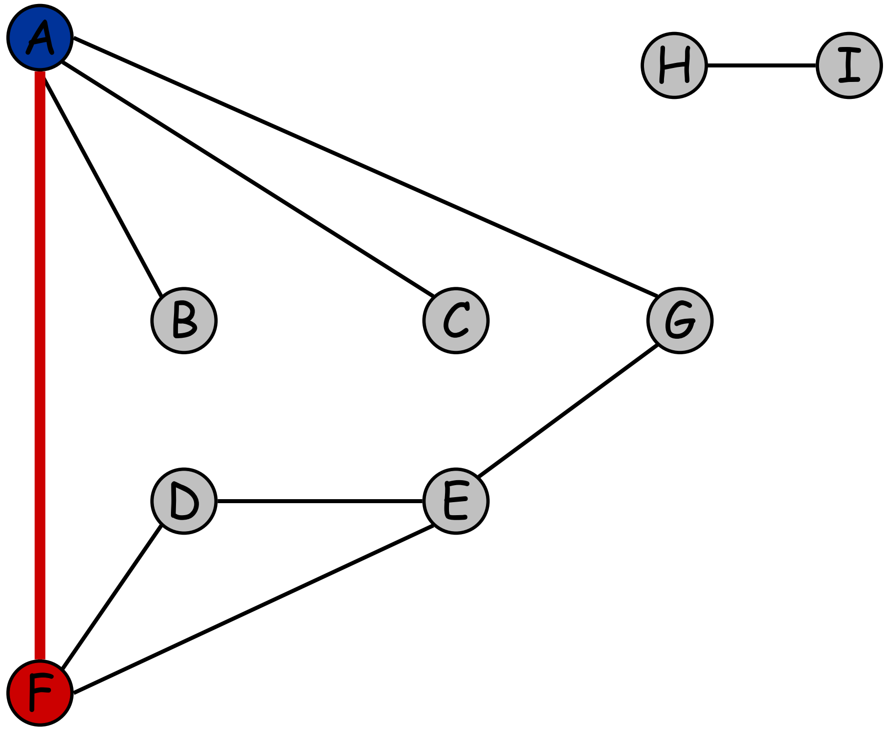
A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D:
G: E A:
H: I:
I: H:

# Undirected Depth First Search



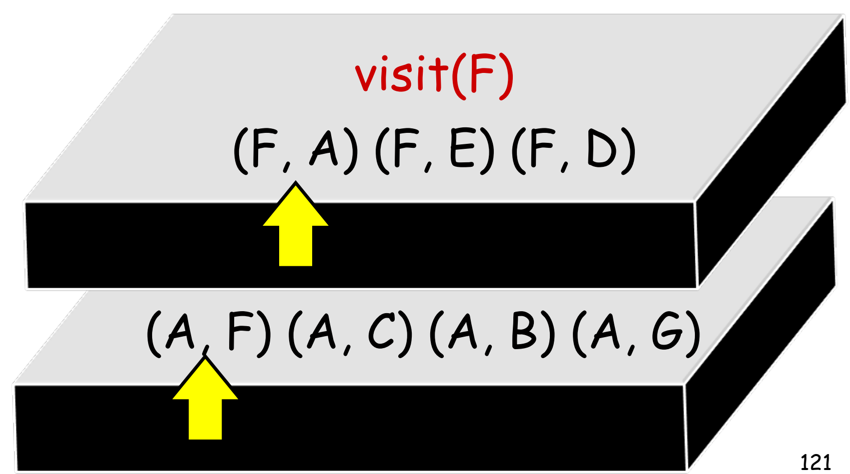
# Undirected Depth First Search

A already marked

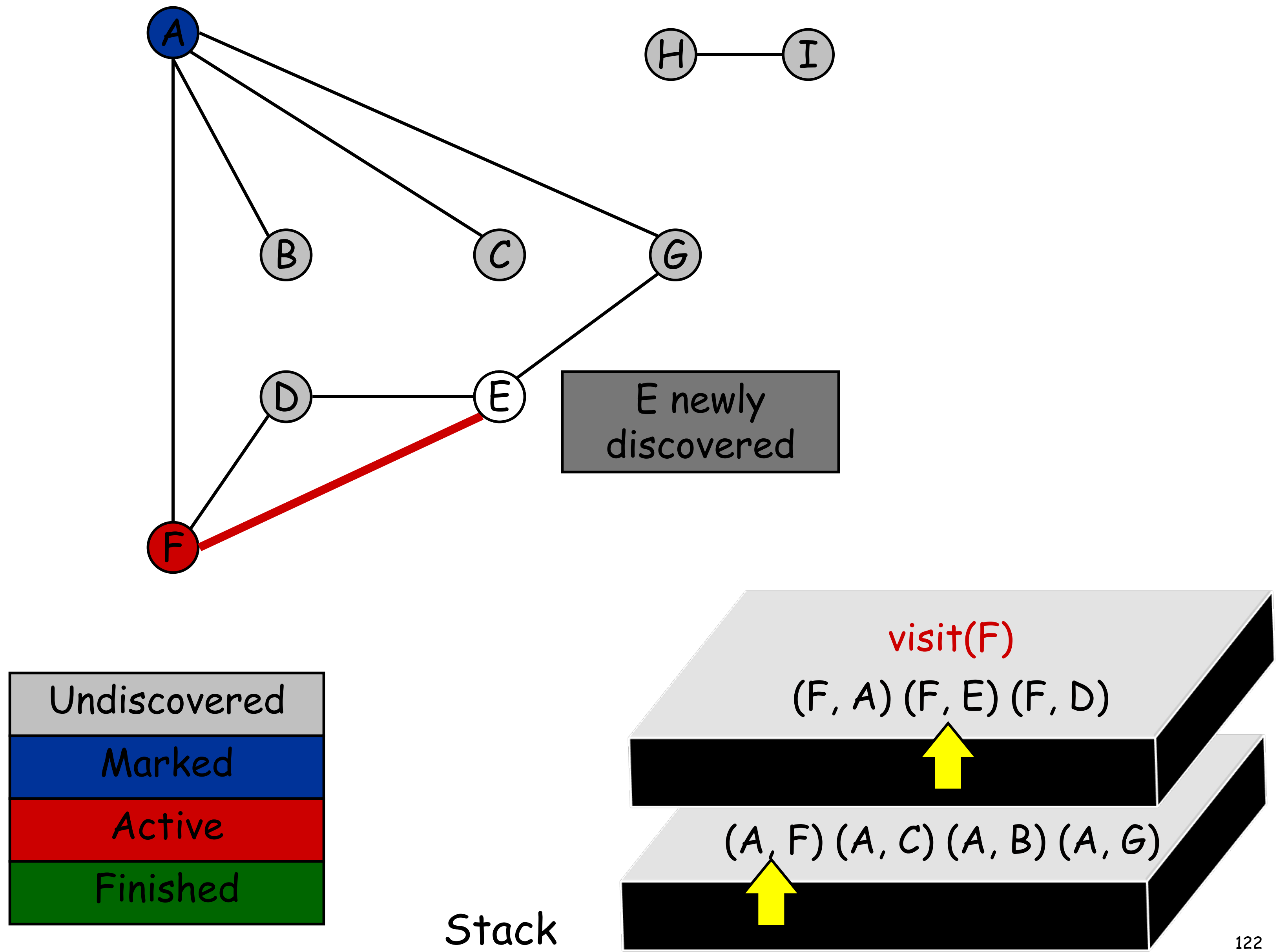



Undiscovered
Marked
Active
Finished

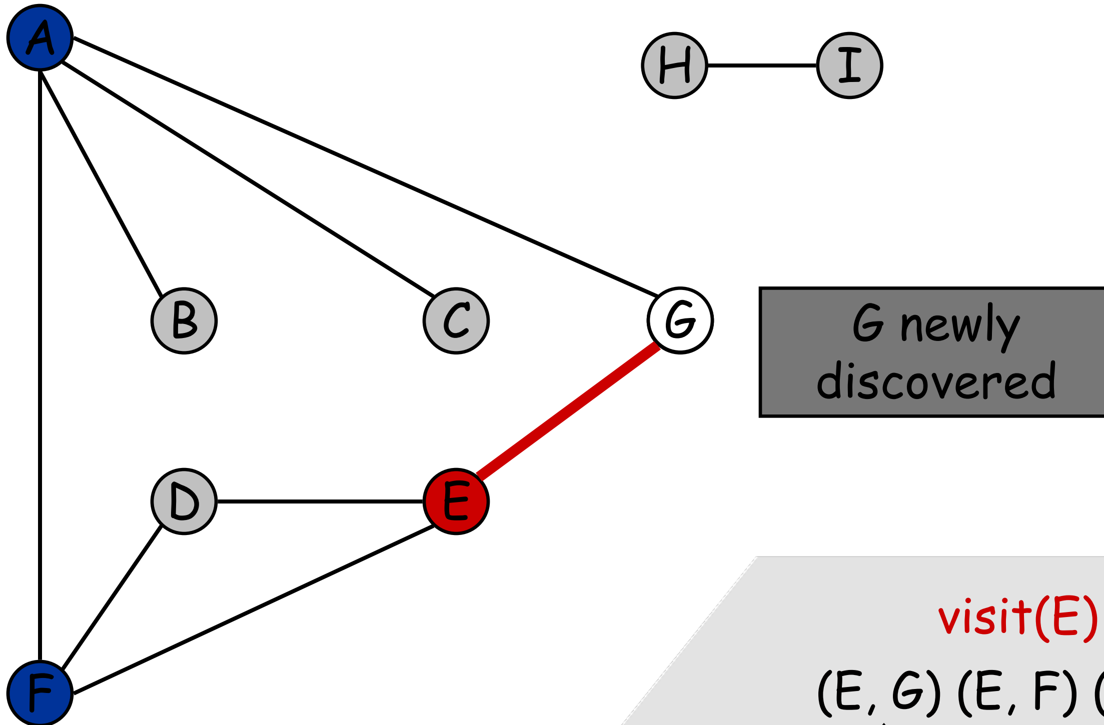
Stack



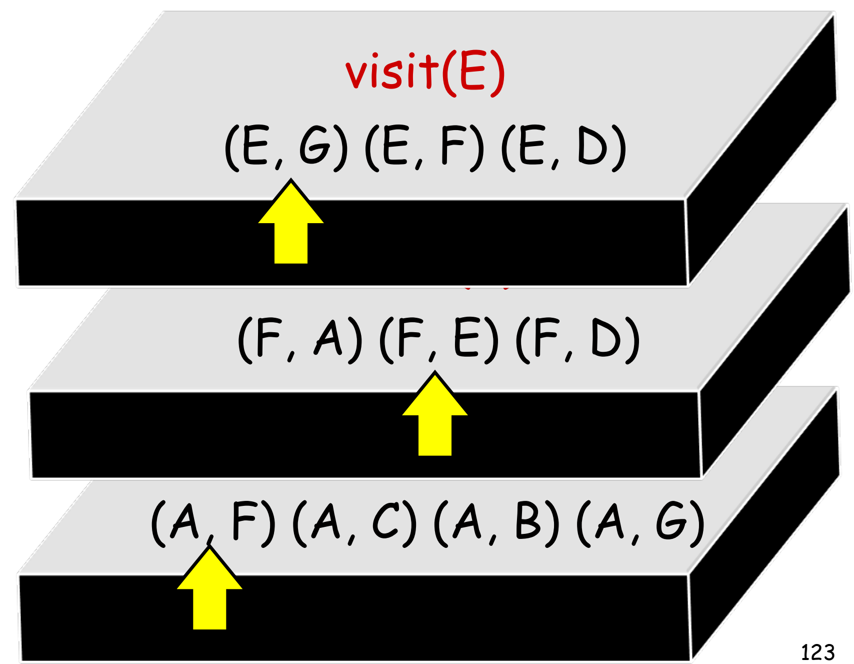
# Undirected Depth First Search



# Undirected Depth First Search

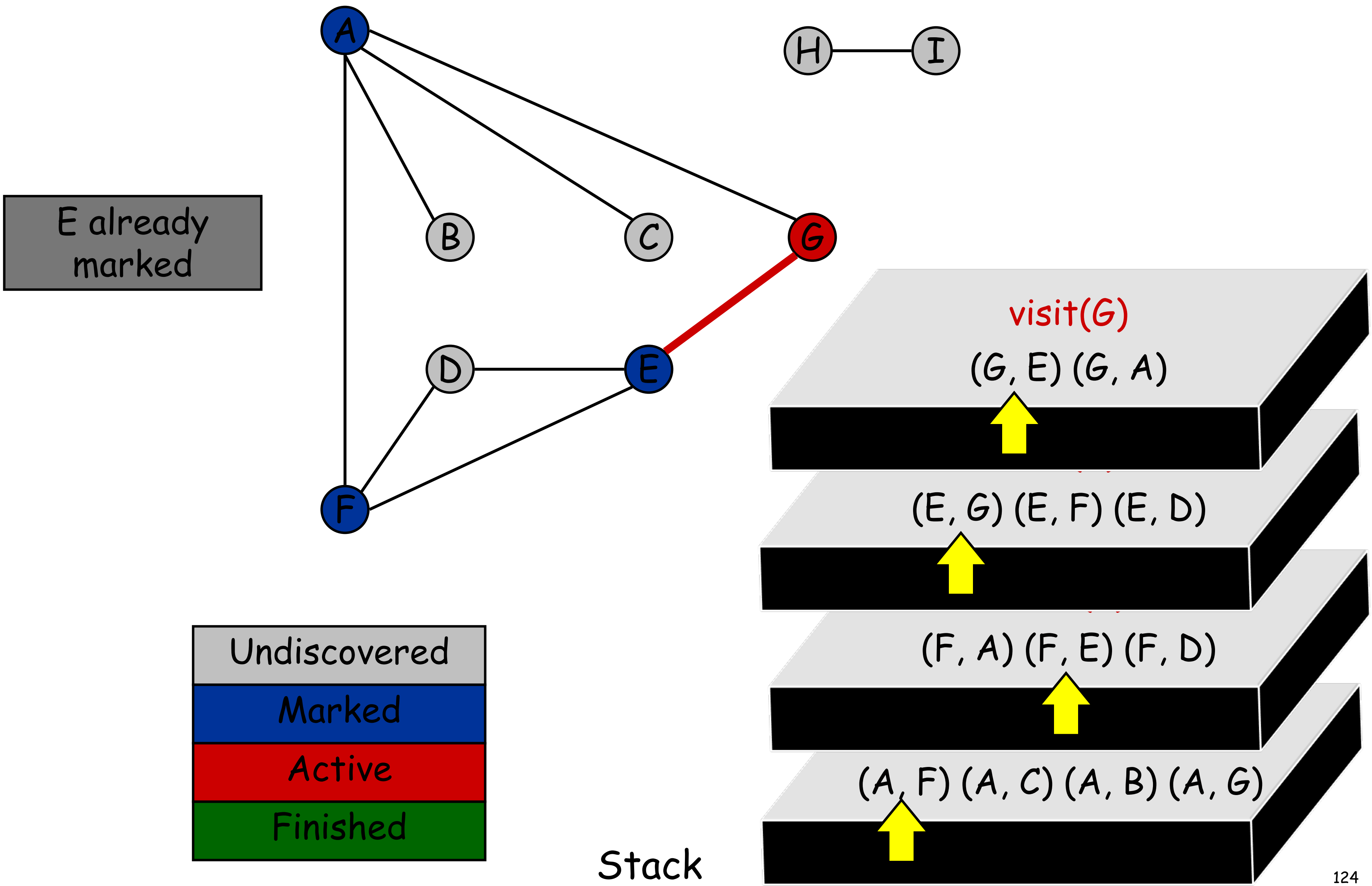



Undiscovered
Marked
Active
Finished

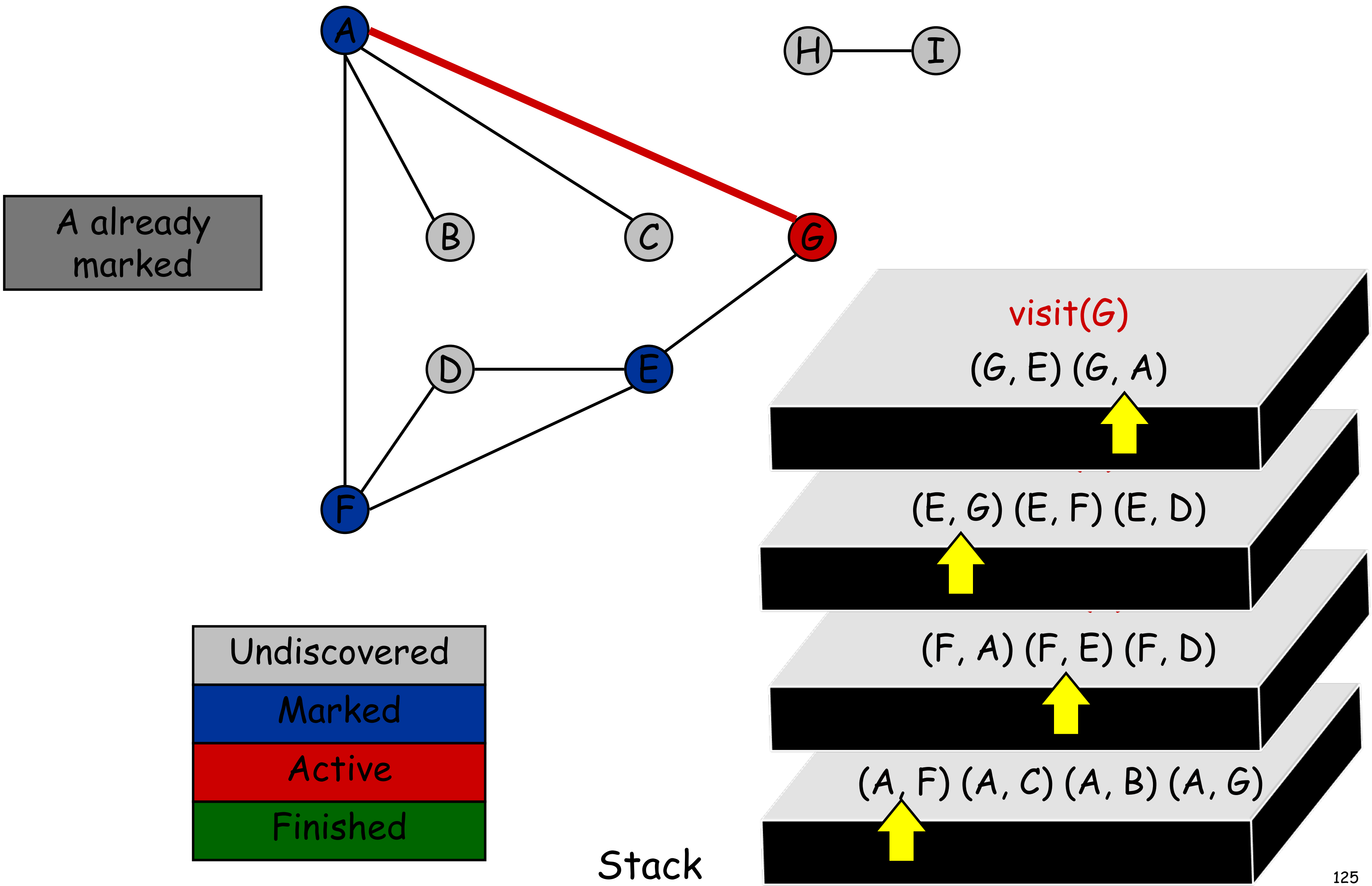



Stack

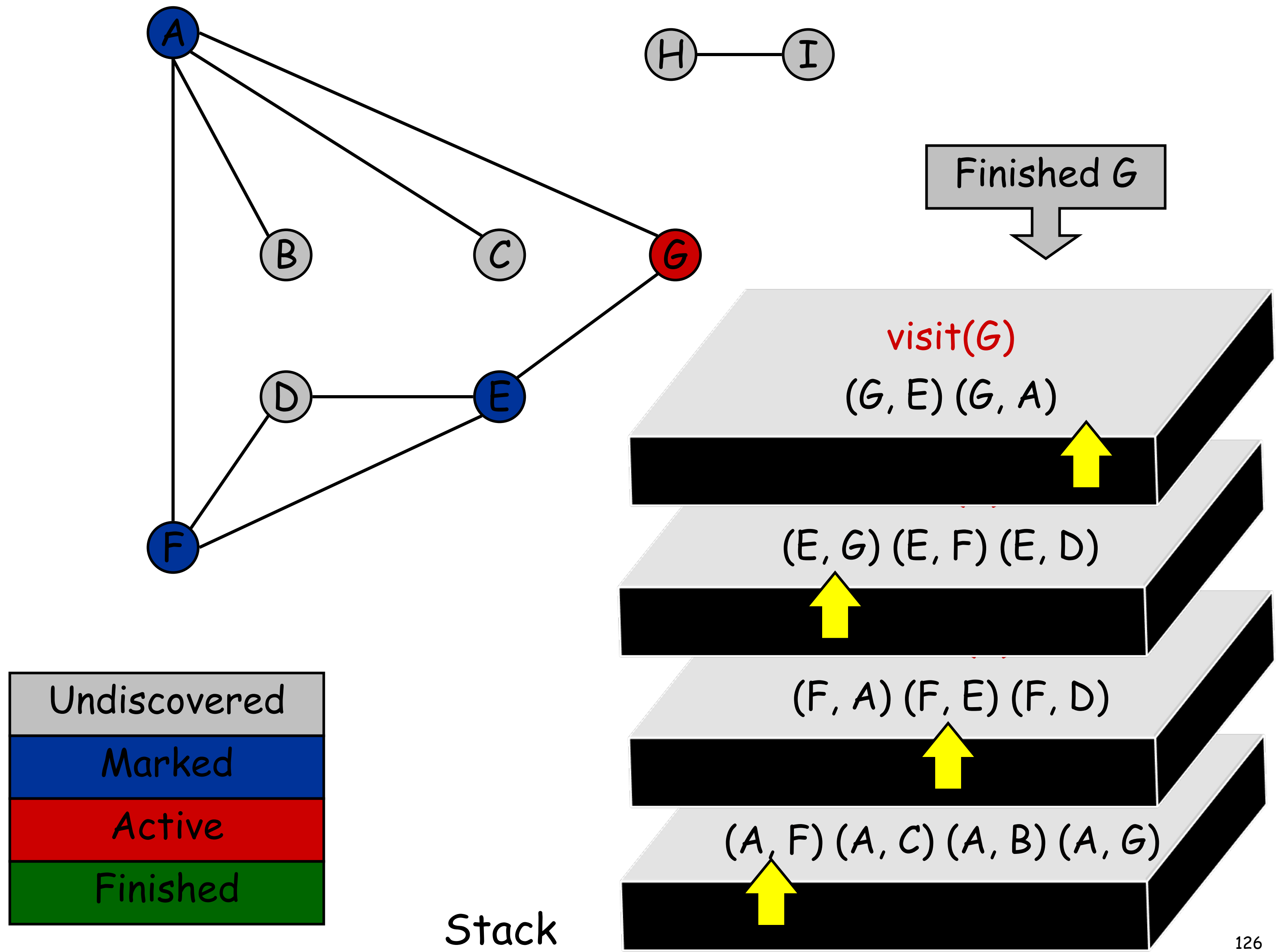
# Undirected Depth First Search



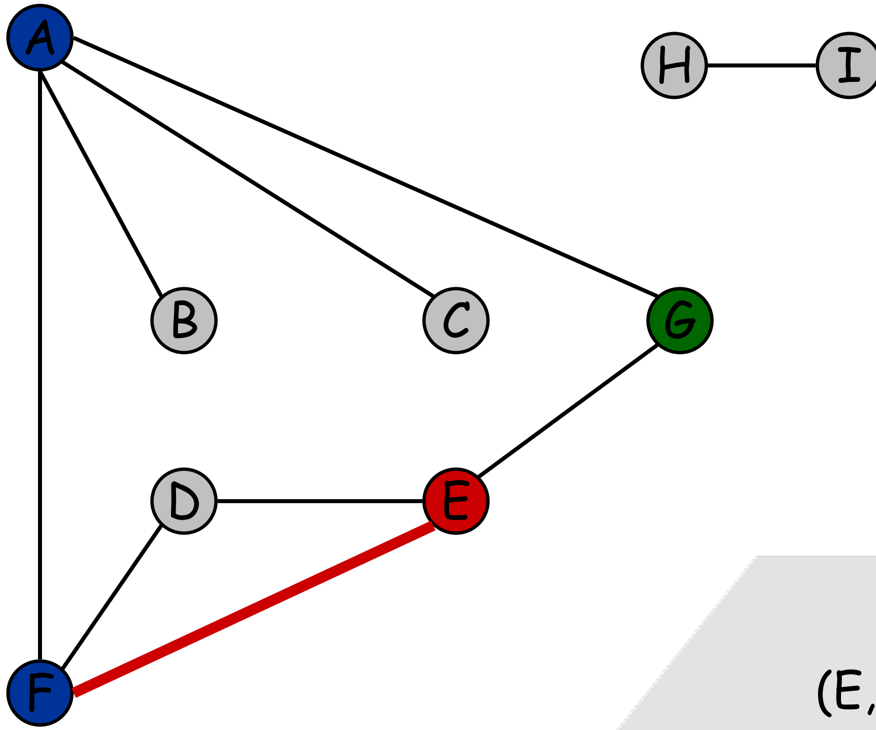
# Undirected Depth First Search



# Undirected Depth First Search

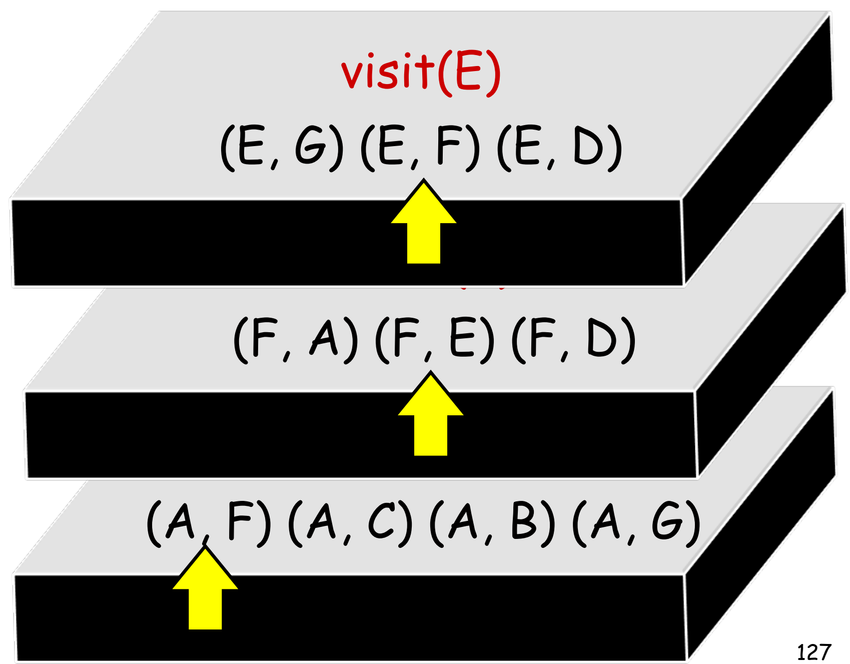


# Undirected Depth First Search



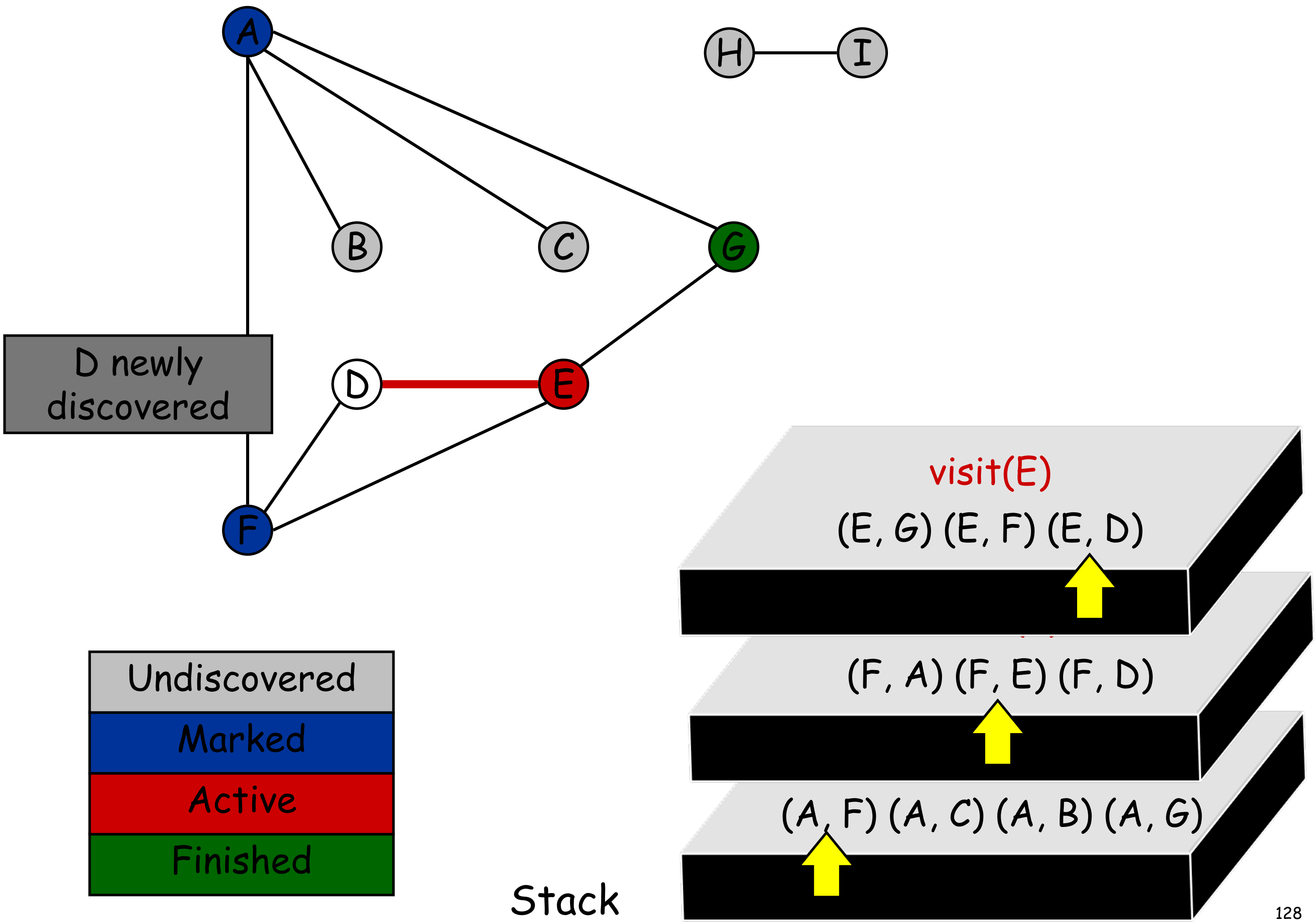

F already marked

| |
|---|
| Undiscovered |
| Marked |
| Active |
| Finished |

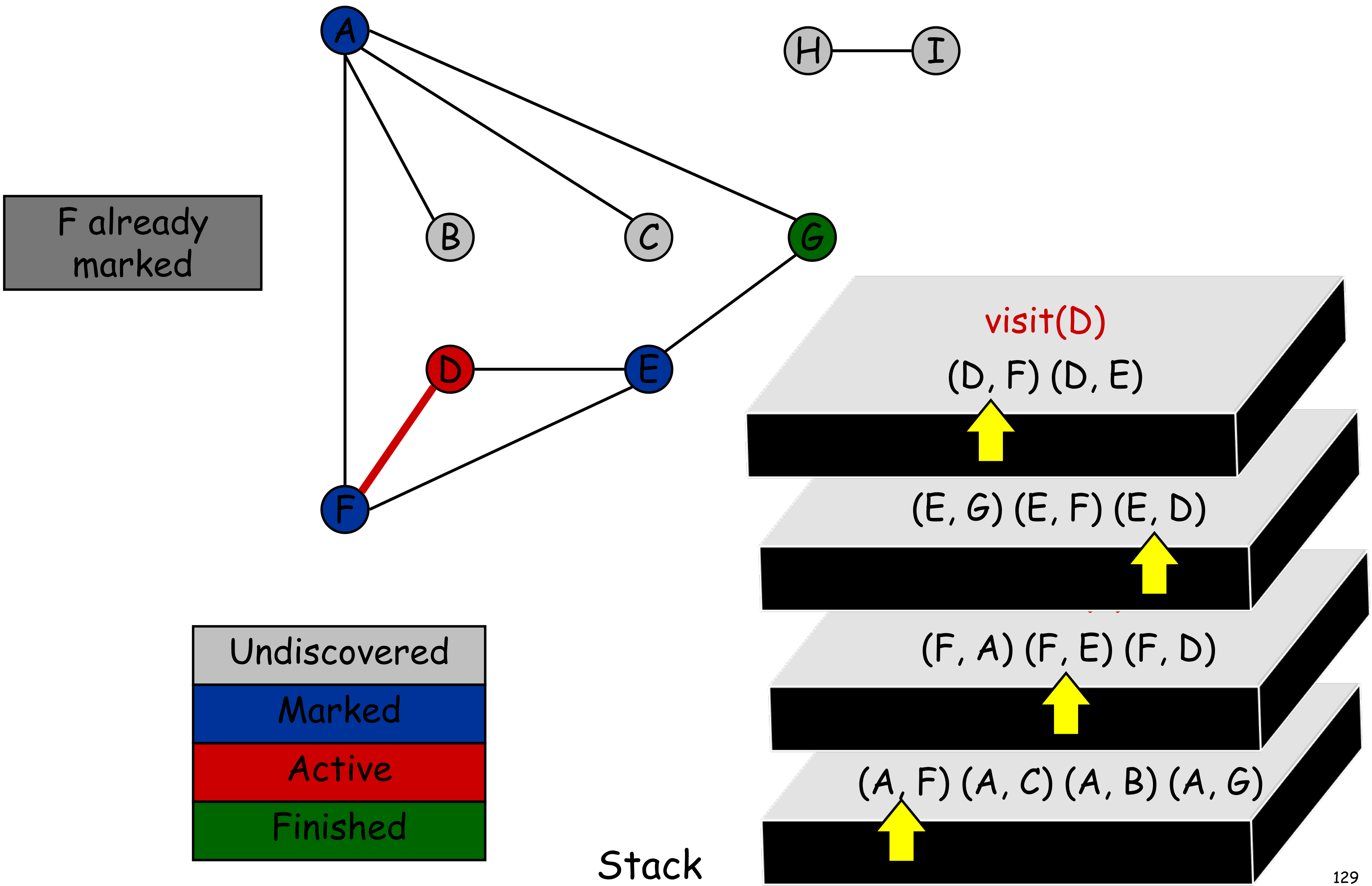



Stack

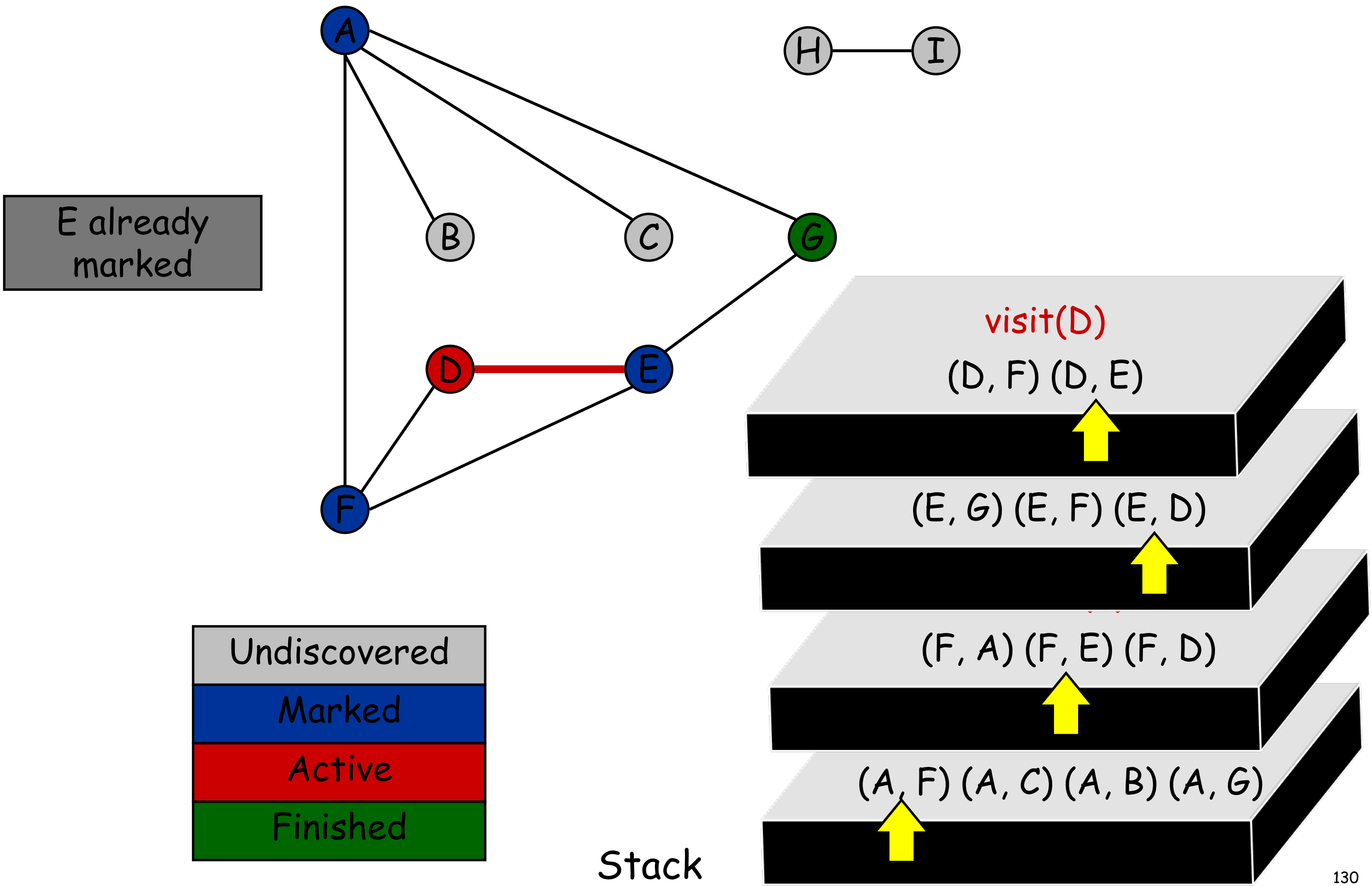
# Undirected Depth First Search



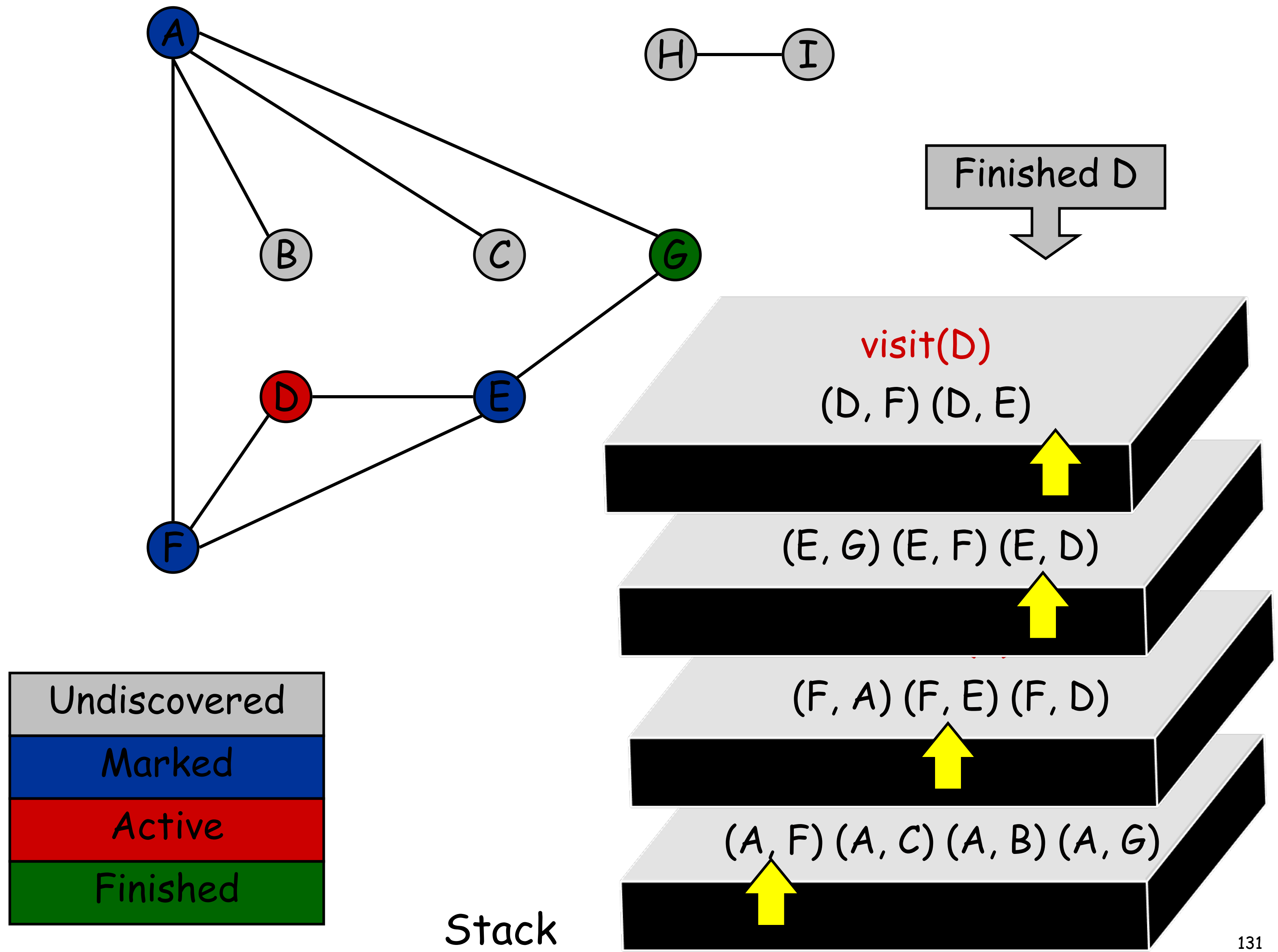
# Undirected Depth First Search



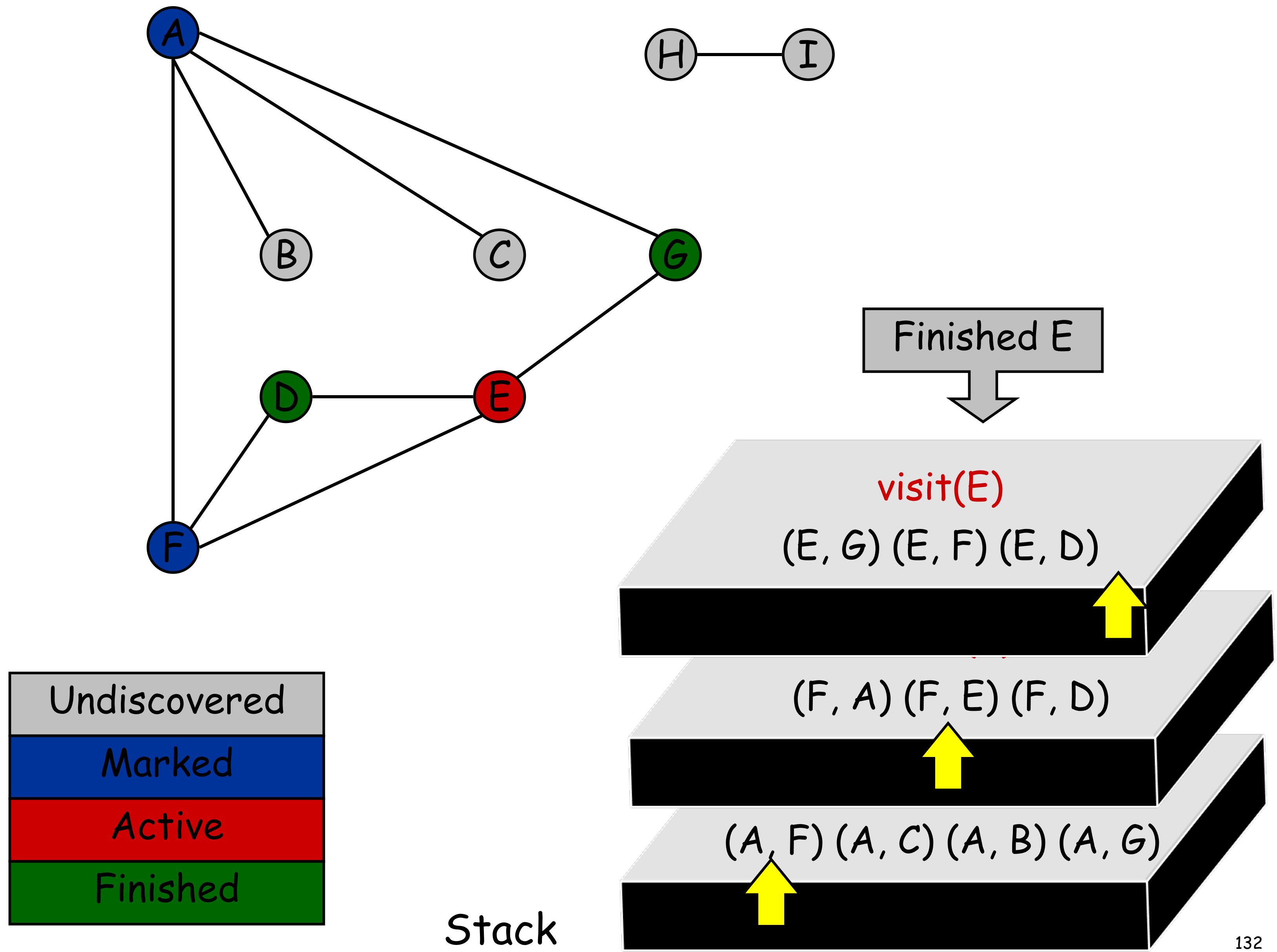
# Undirected Depth First Search



# Undirected Depth First Search

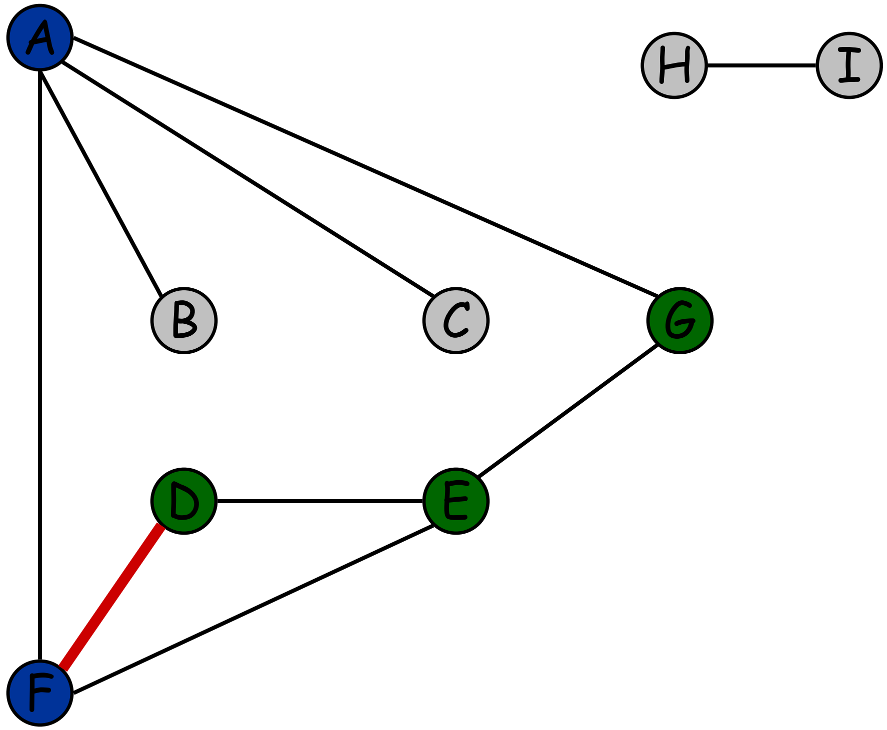


# Undirected Depth First Search

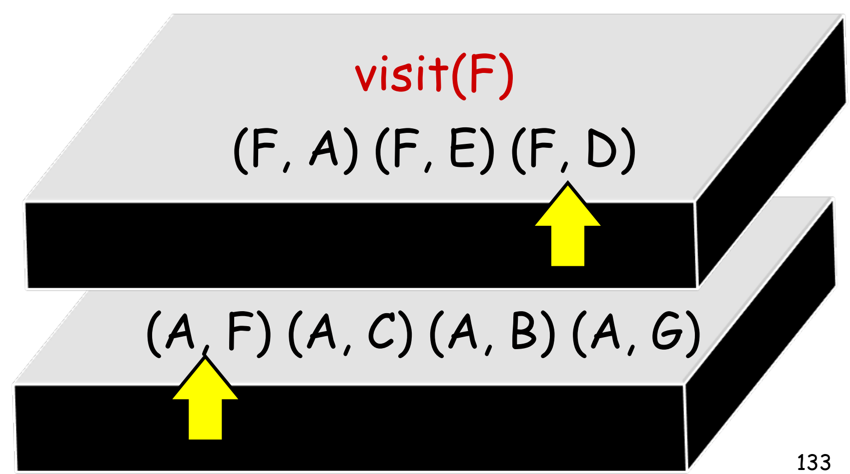


# Undirected Depth First Search

D already marked

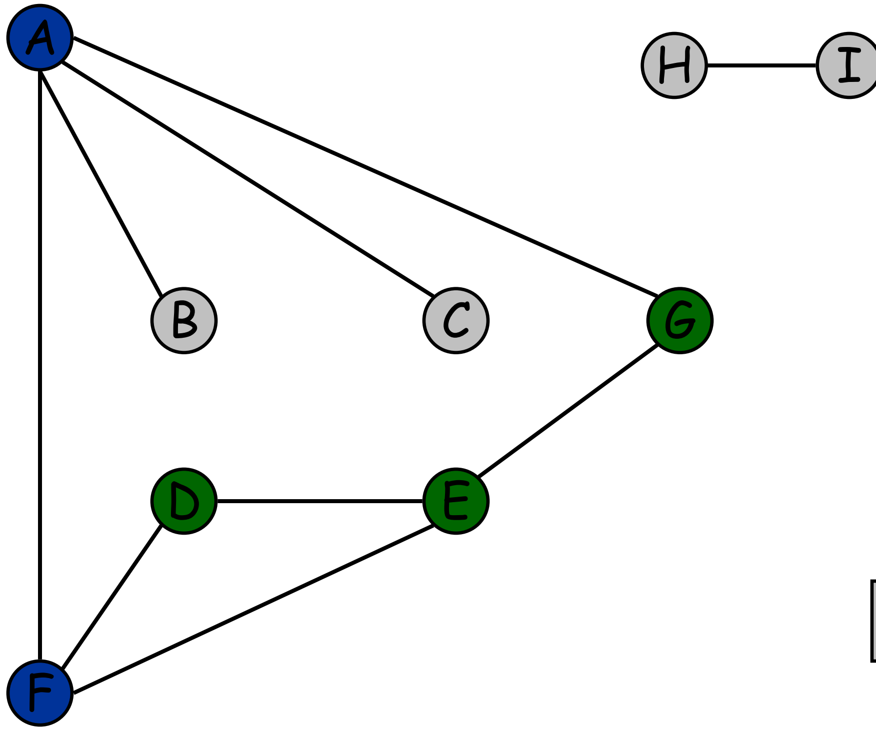


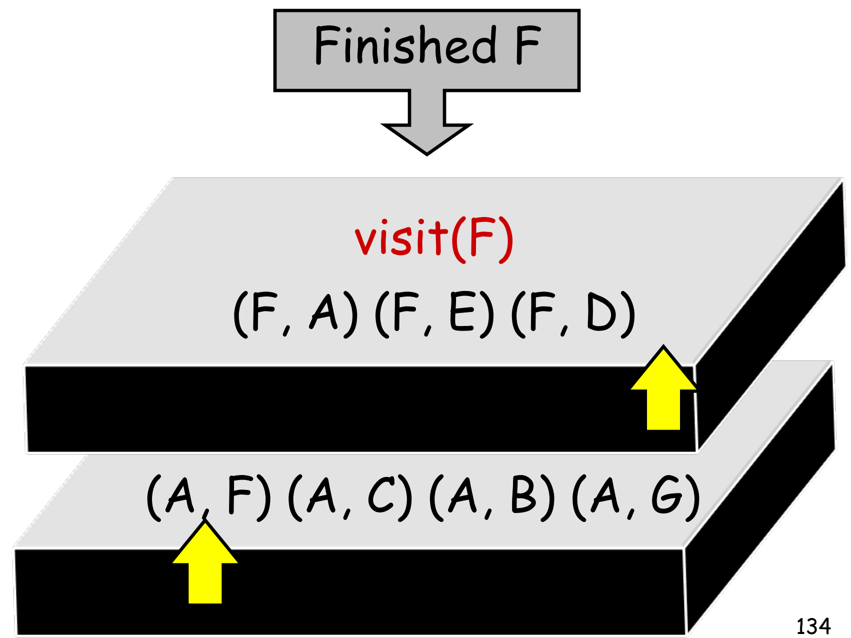

| |
|---|
| Undiscovered |
| Marked |
| Active |
| Finished |




Stack

# Undirected Depth First Search

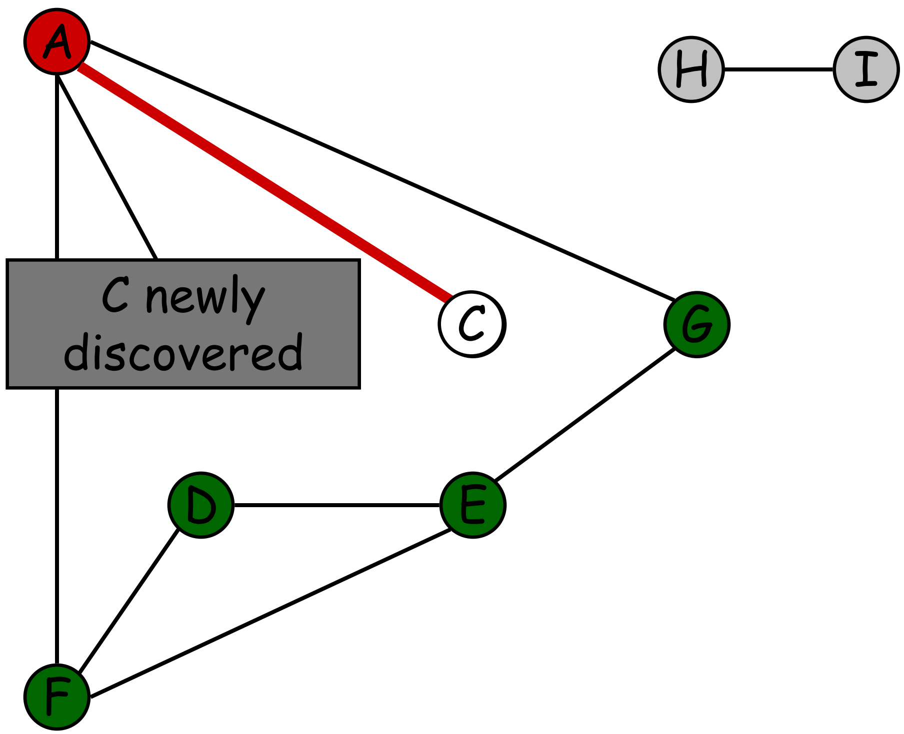


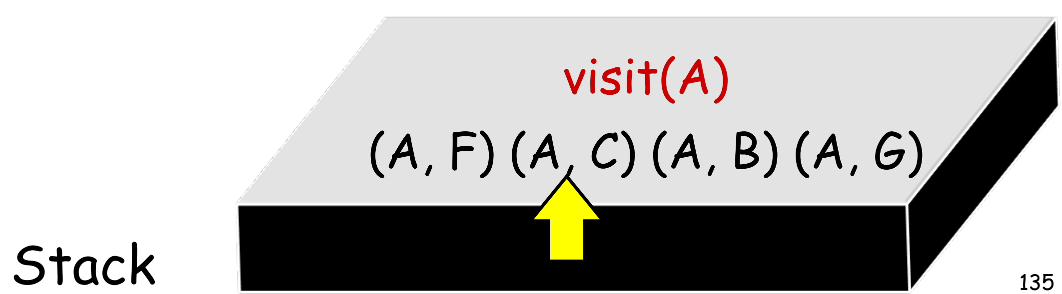

| |
|---|
| Undiscovered |
| Marked |
| Active |
| Finished |

Stack

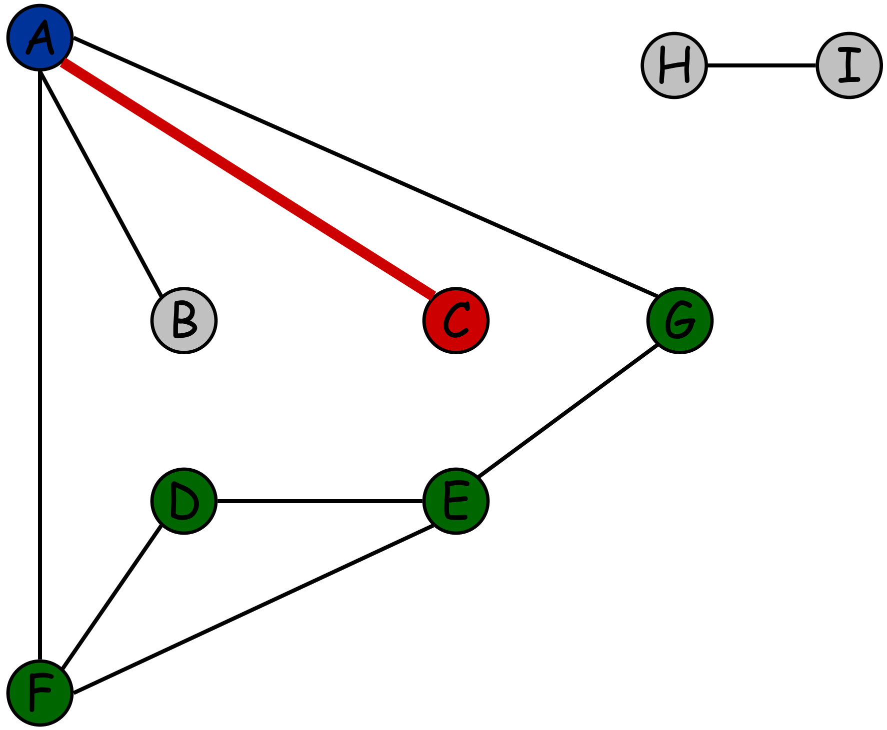
# Undirected Depth First Search





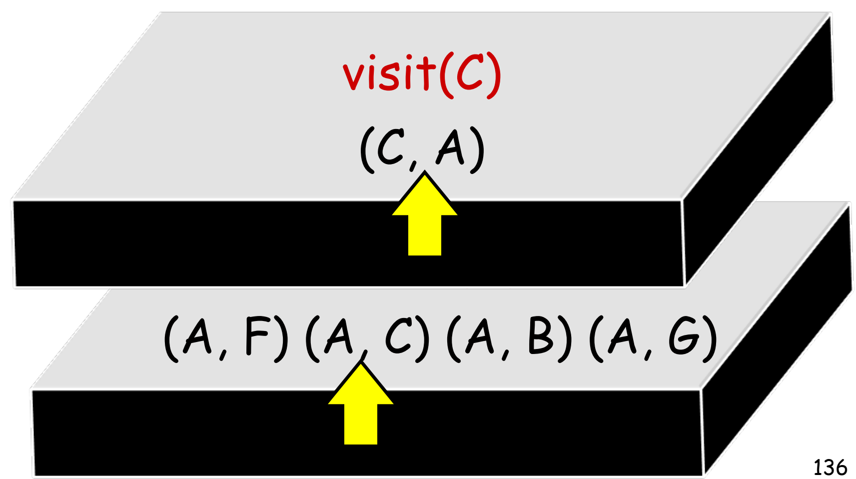


# Undirected Depth First Search



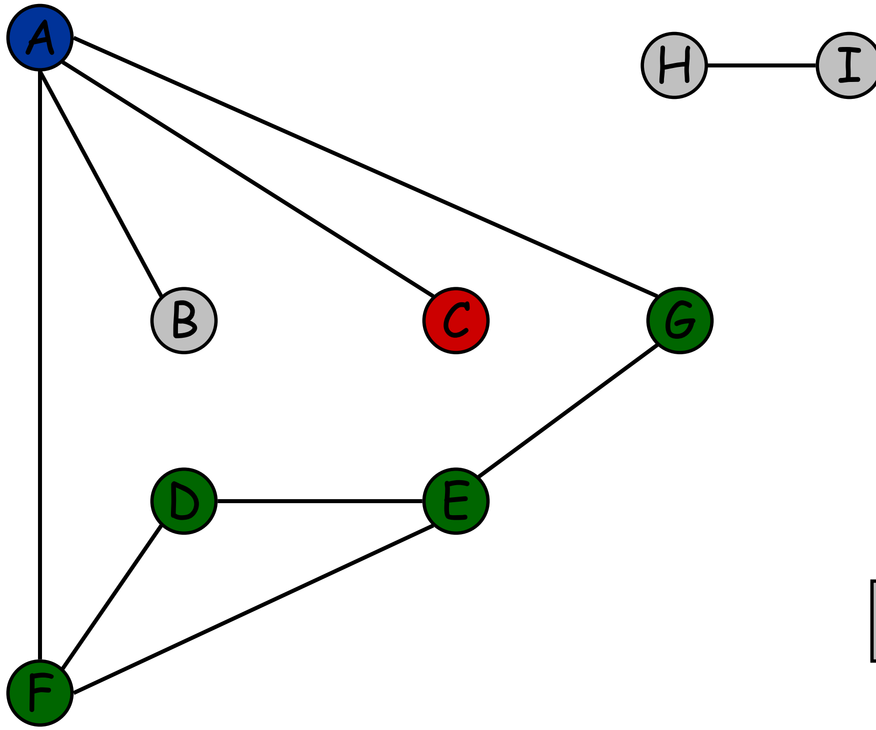

A already marked

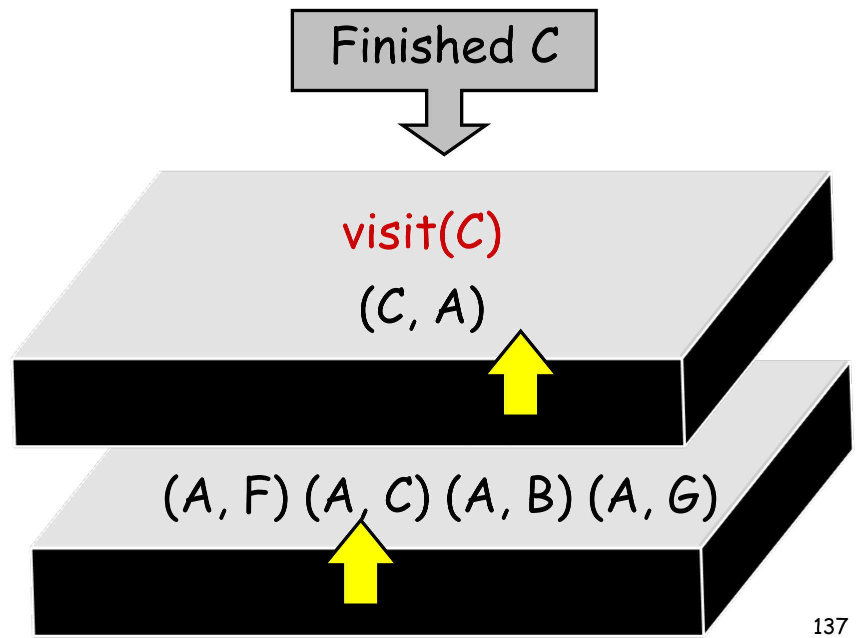
| |
|---|
| Undiscovered |
| Marked |
| Active |
| Finished |




Stack

# Undirected Depth First Search




Undiscovered

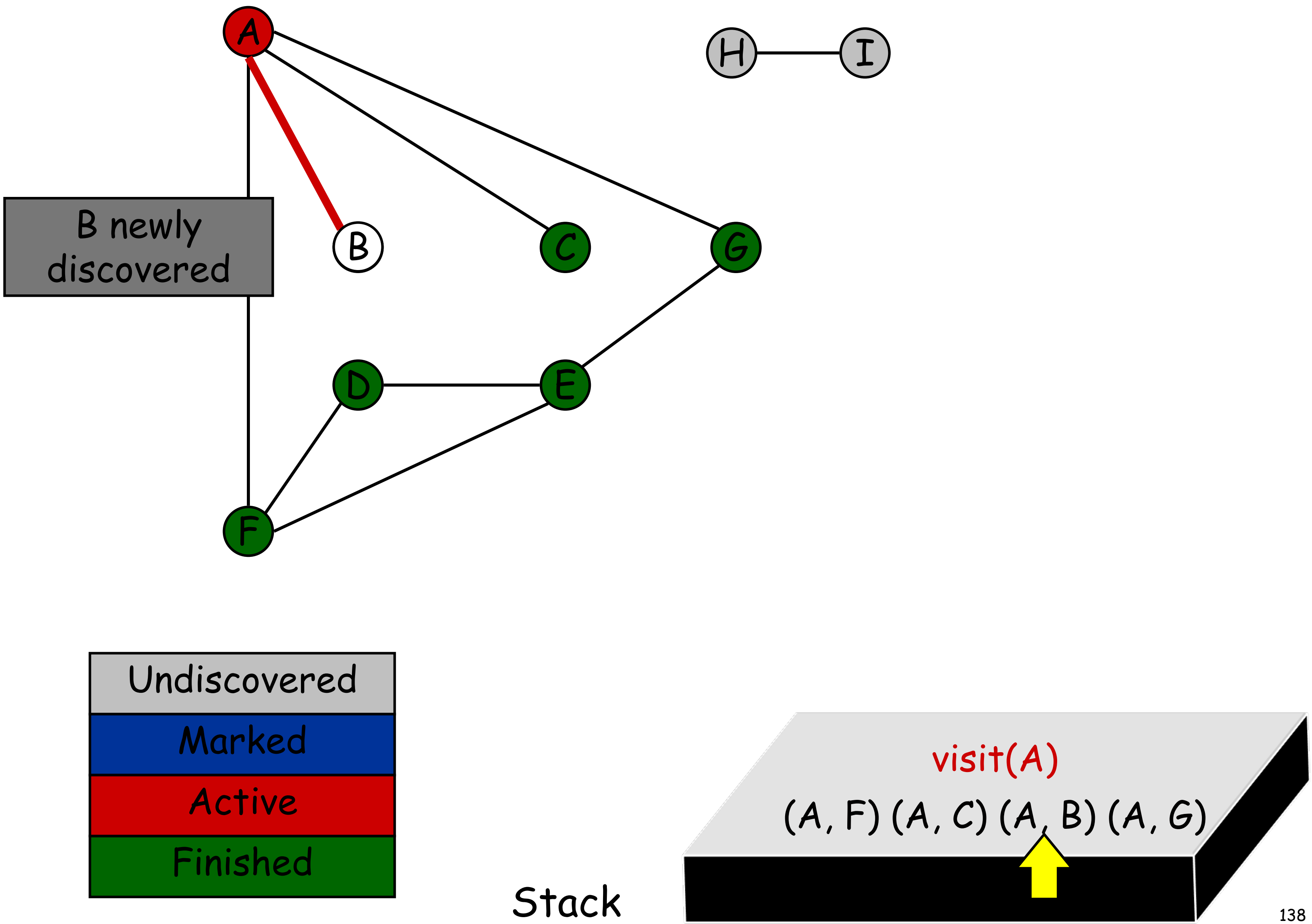
Marked

Active

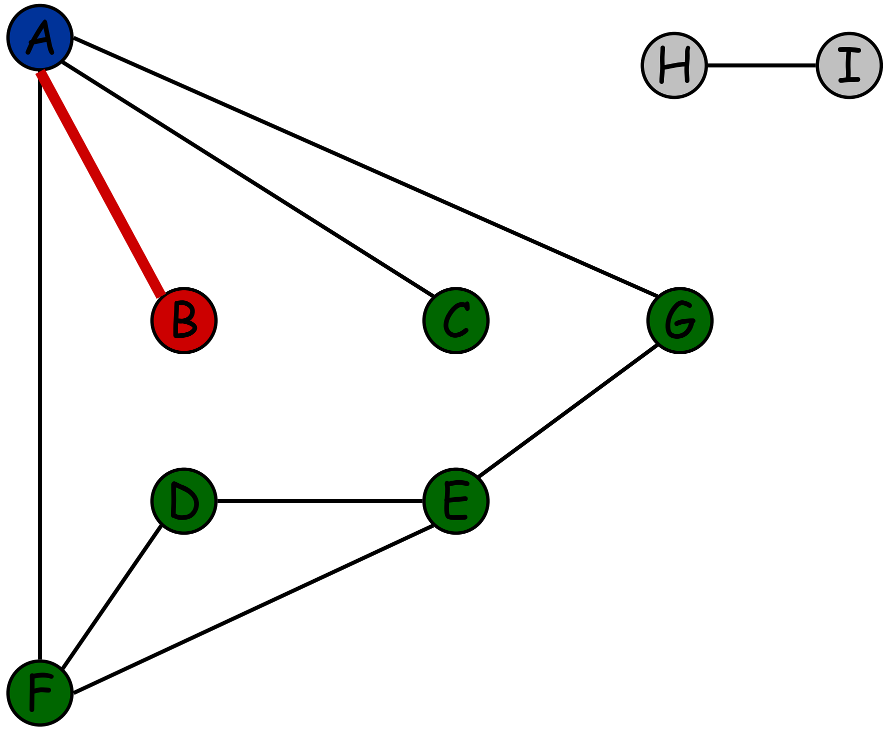
Finished

Stack

# Undirected Depth First Search

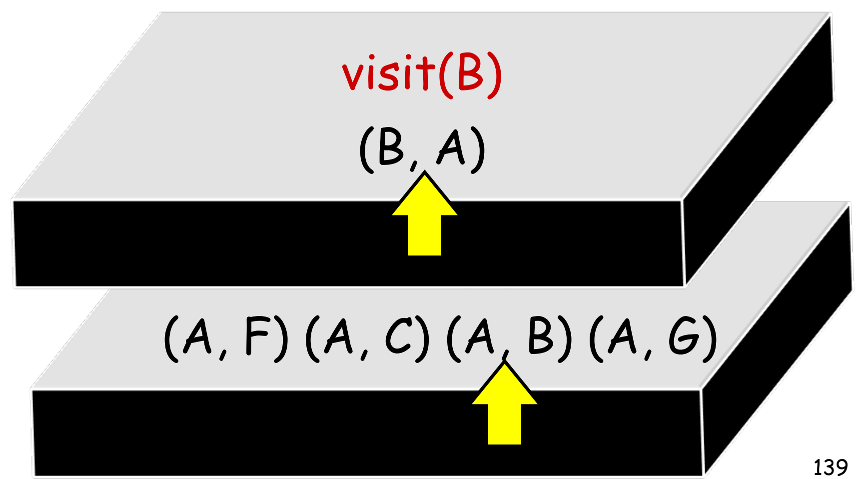


# Undirected Depth First Search



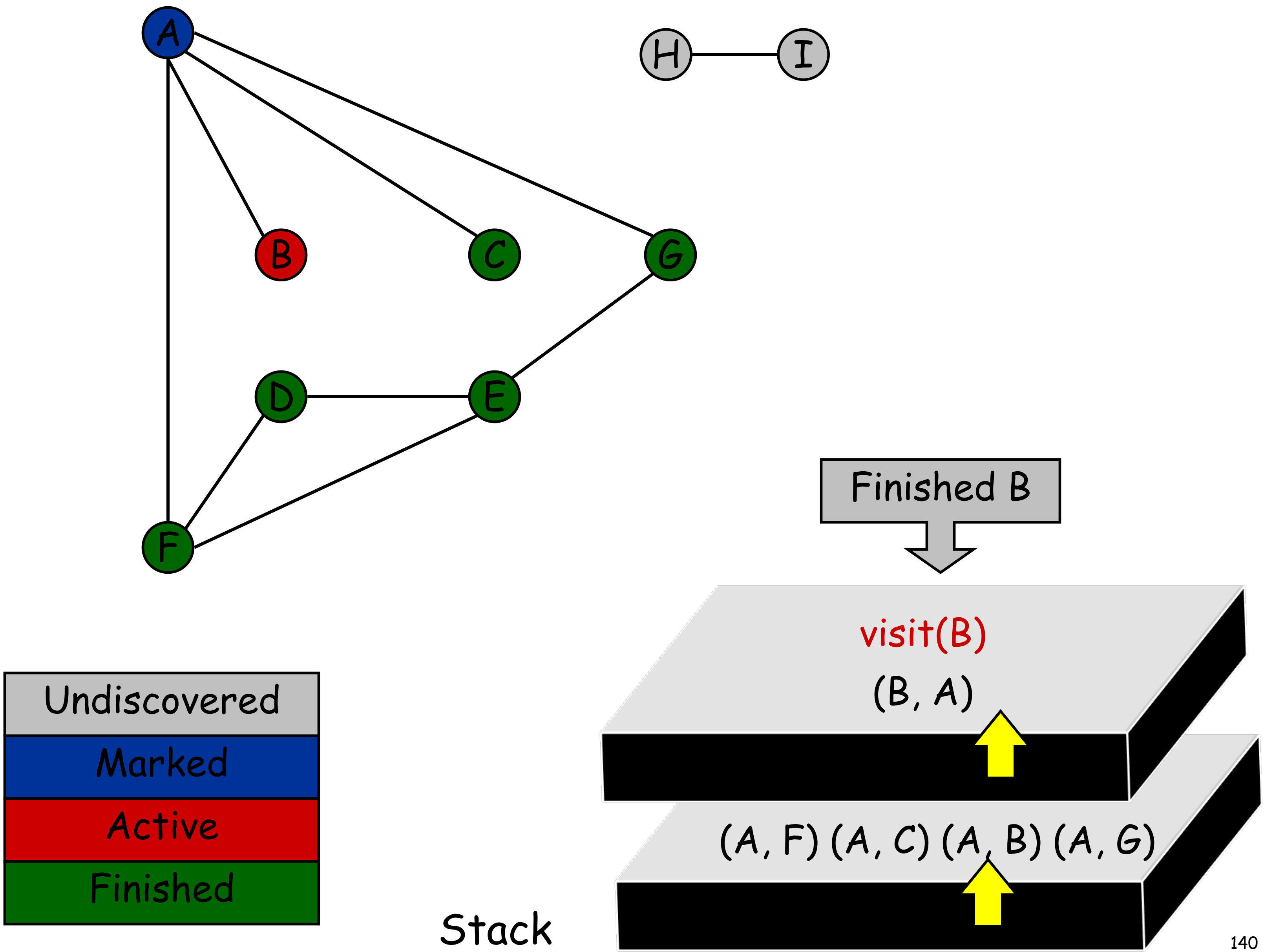

A already marked

| |
|---|
| Undiscovered |
| Marked |
| Active |
| Finished |

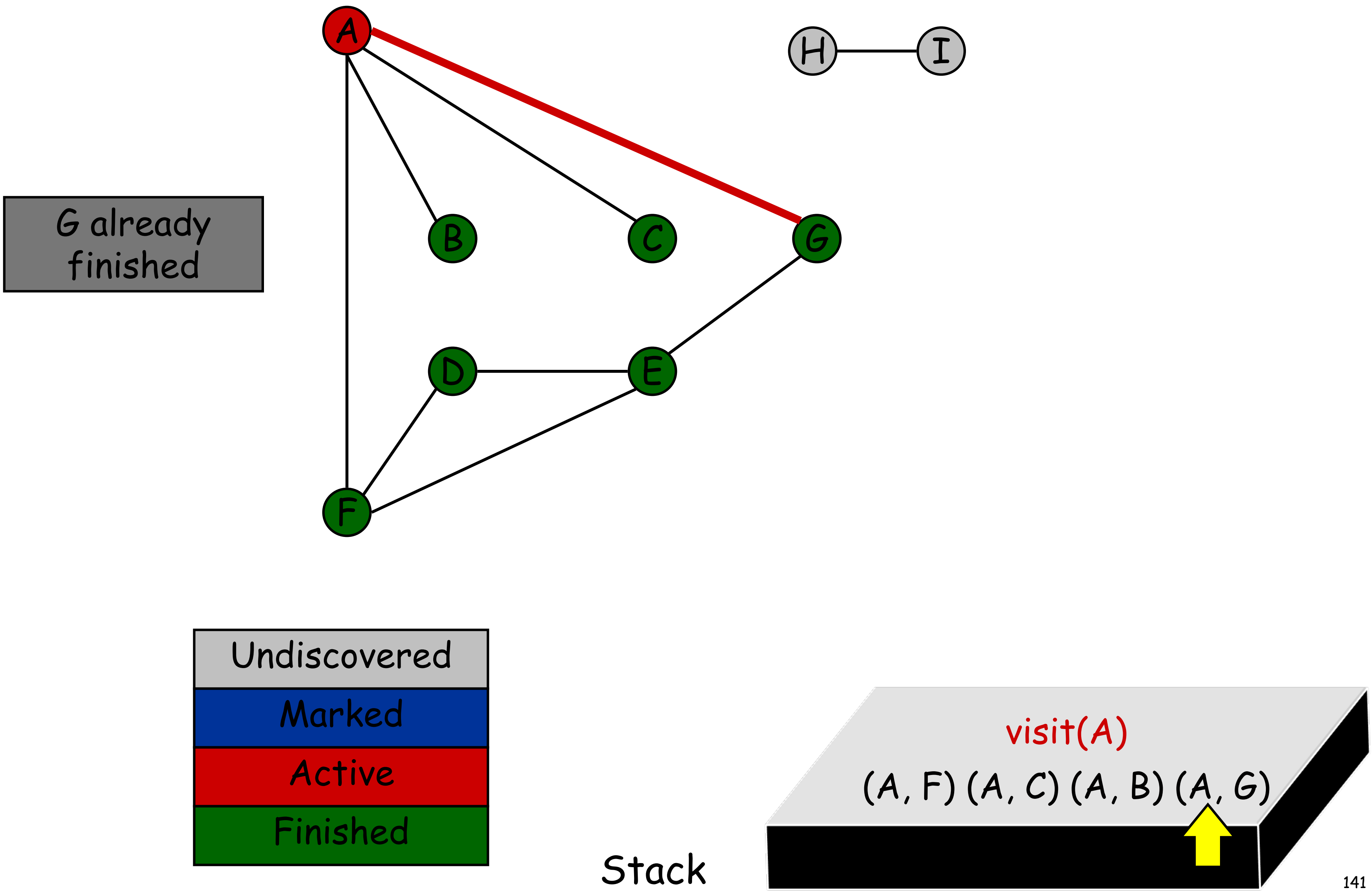



Stack

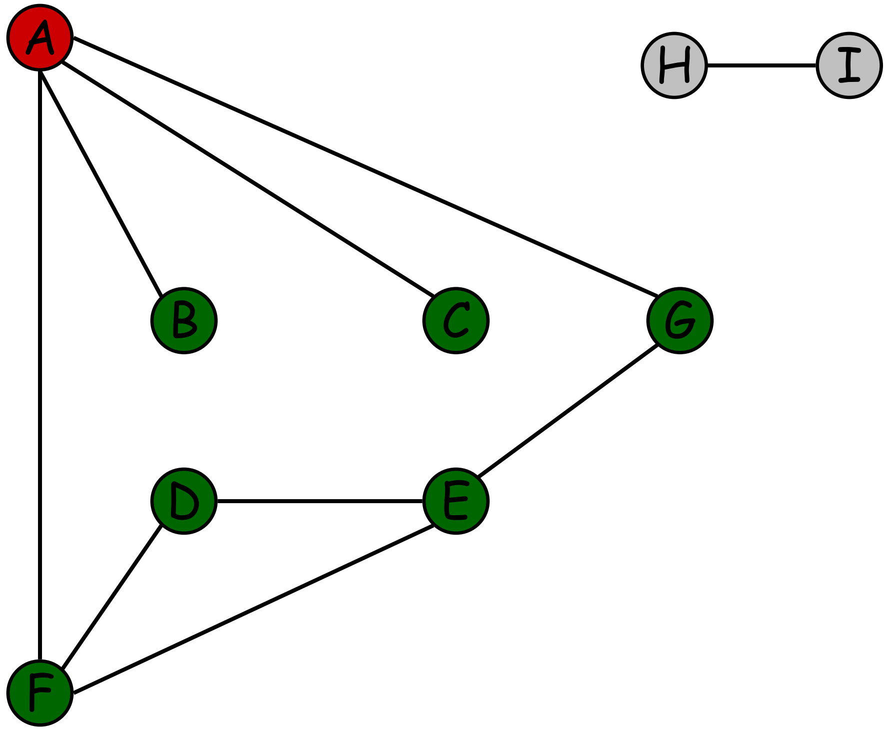
# Undirected Depth First Search



# Undirected Depth First Search



# Undirected Depth First Search




Undiscovered

Marked

Active

Finished

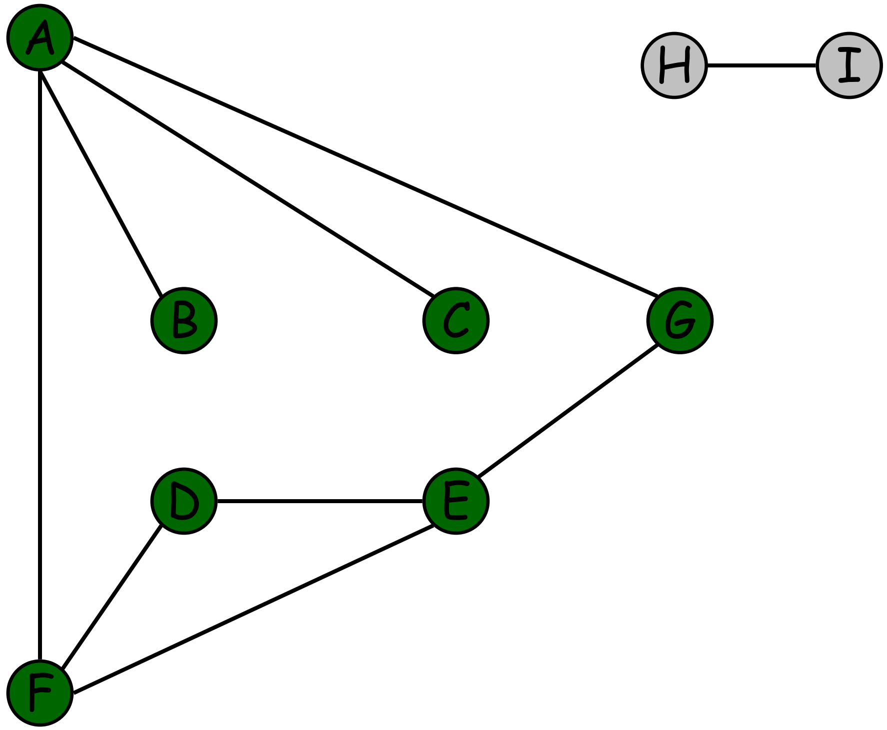
Finished A

visit(A)

(A, F) (A, C) (A, B) (A, G)

Stack

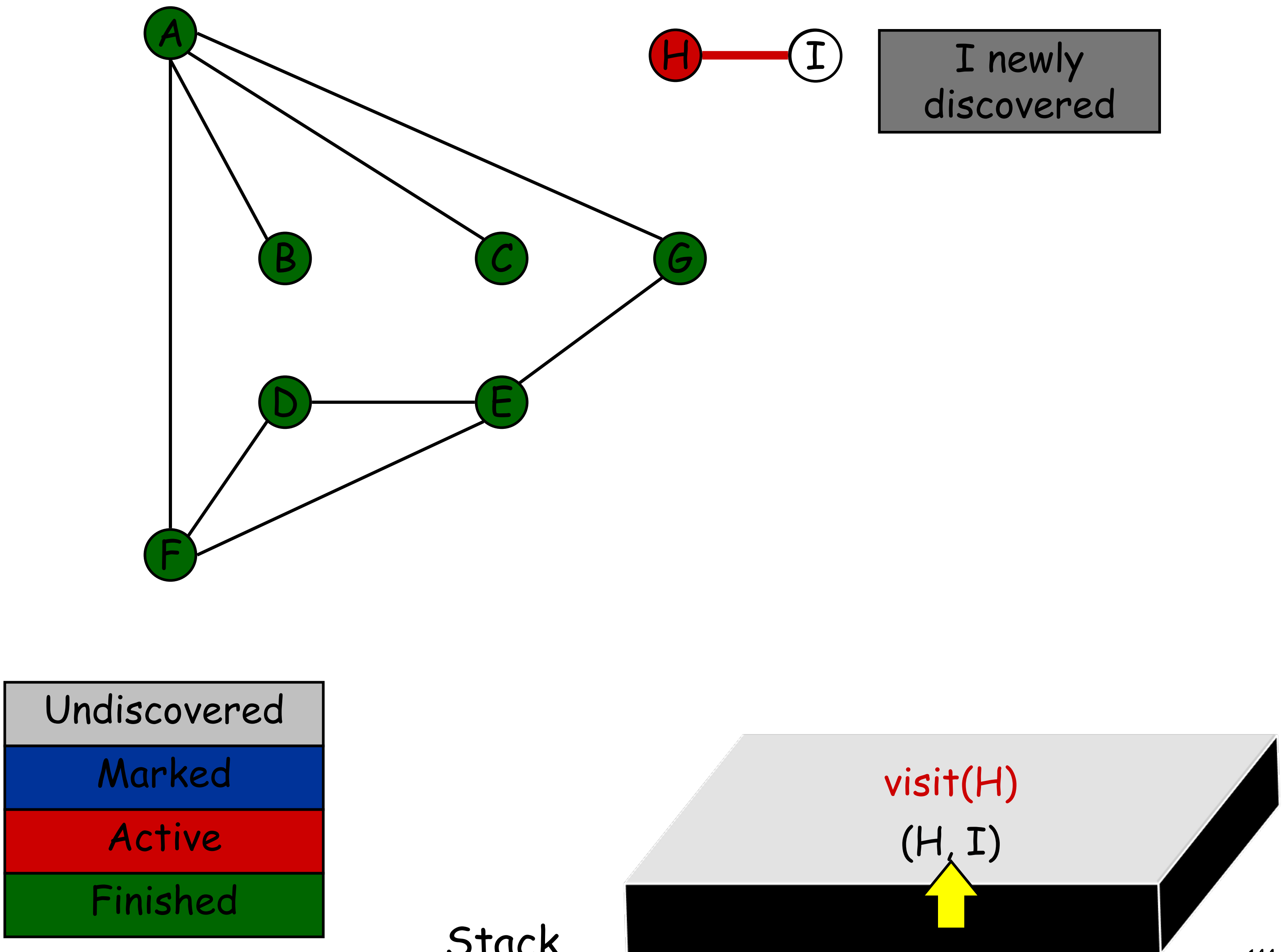
# Undirected Depth First Search



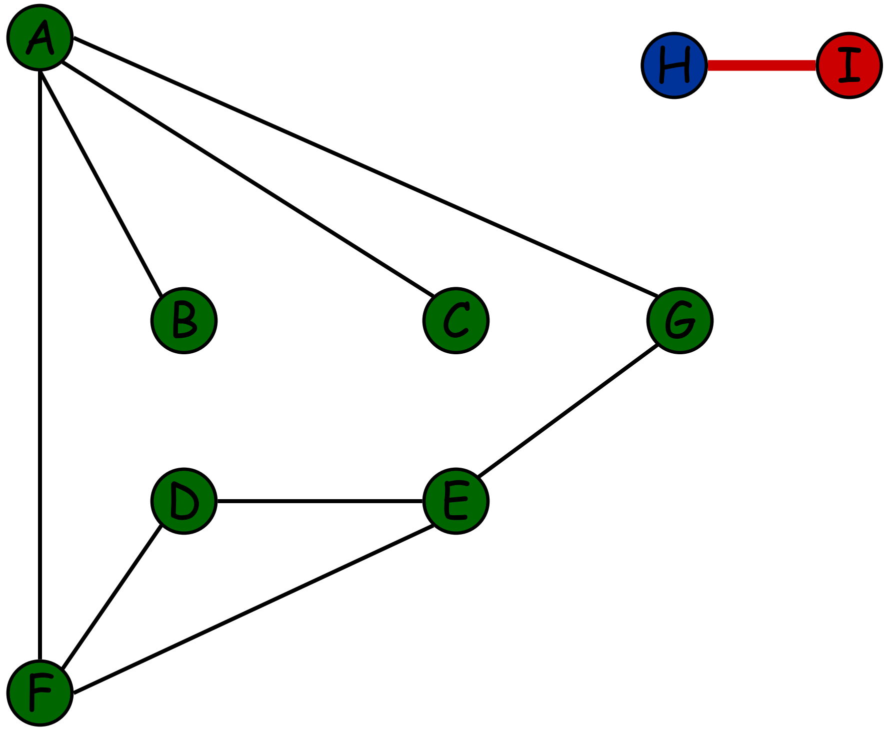

| |
|---|
| Undiscovered |
| Marked |
| Active |
| Finished |

Stack

# Undirected Depth First Search

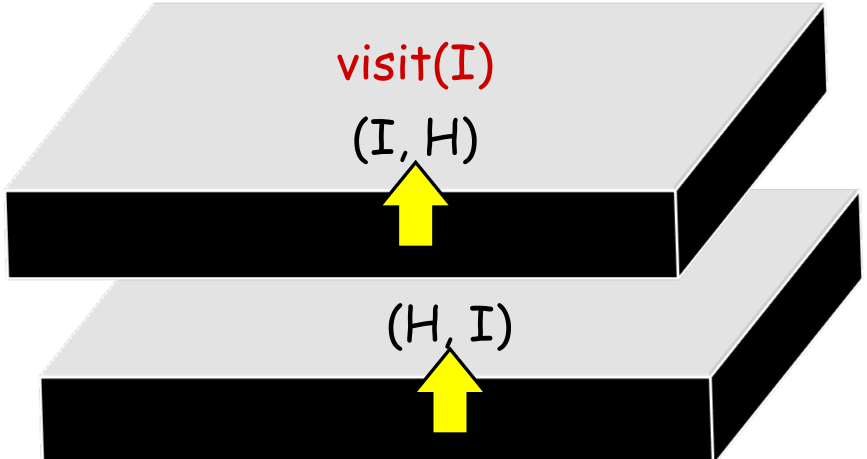


# Undirected Depth First Search



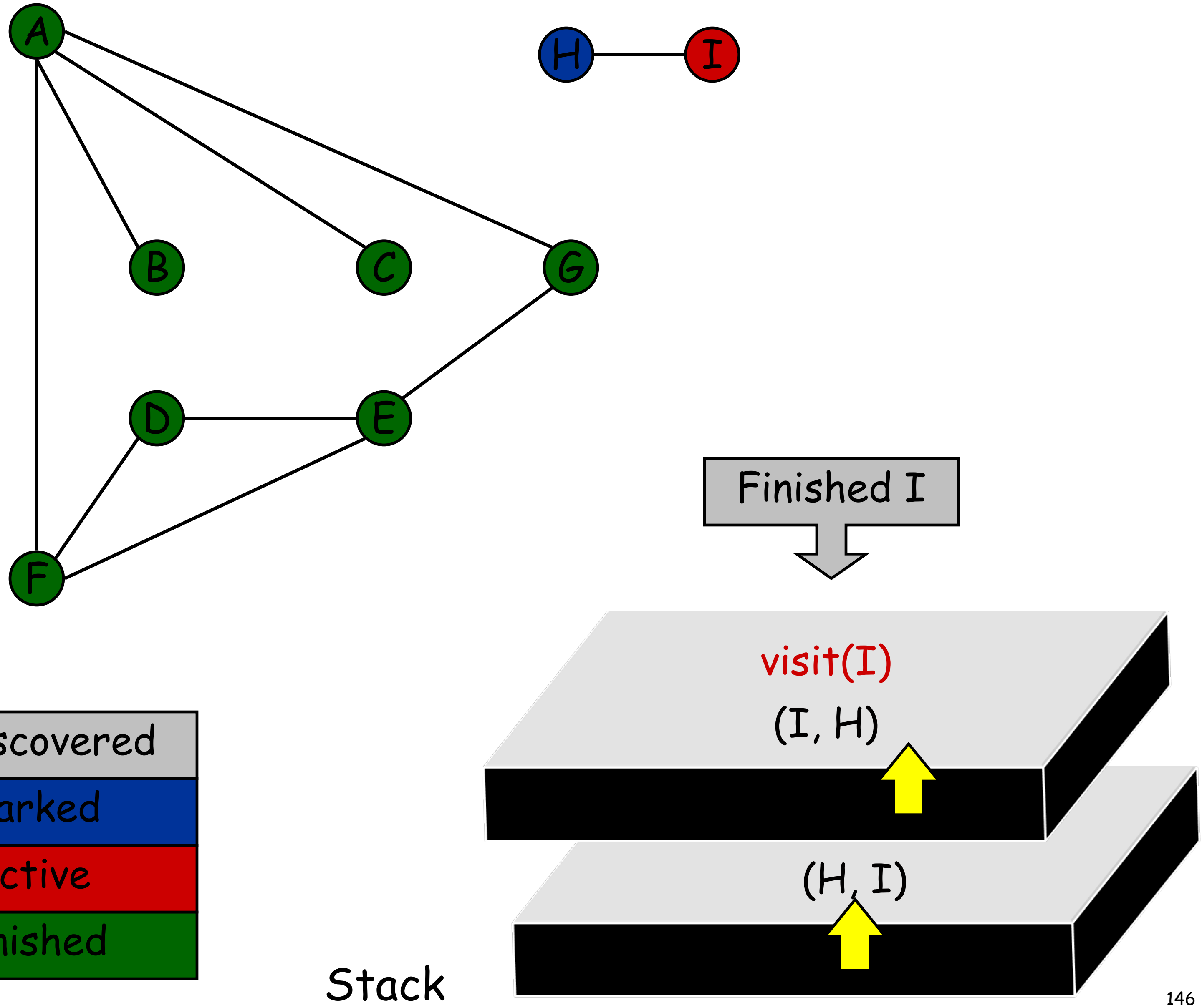

H already marked

| |
|---|
| Undiscovered |
| Marked |
| Active |
| Finished |



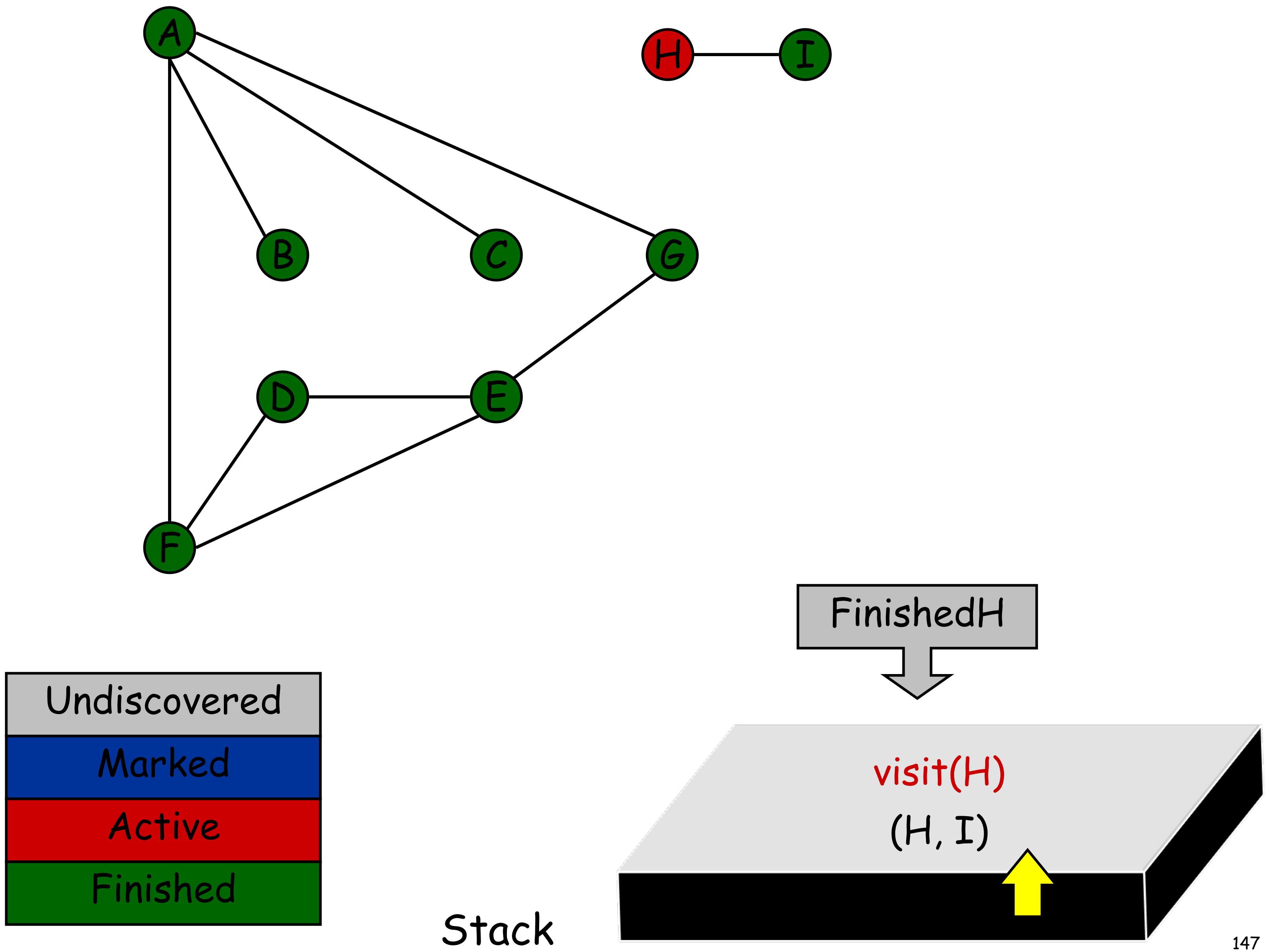

Stack

# Undirected Depth First Search

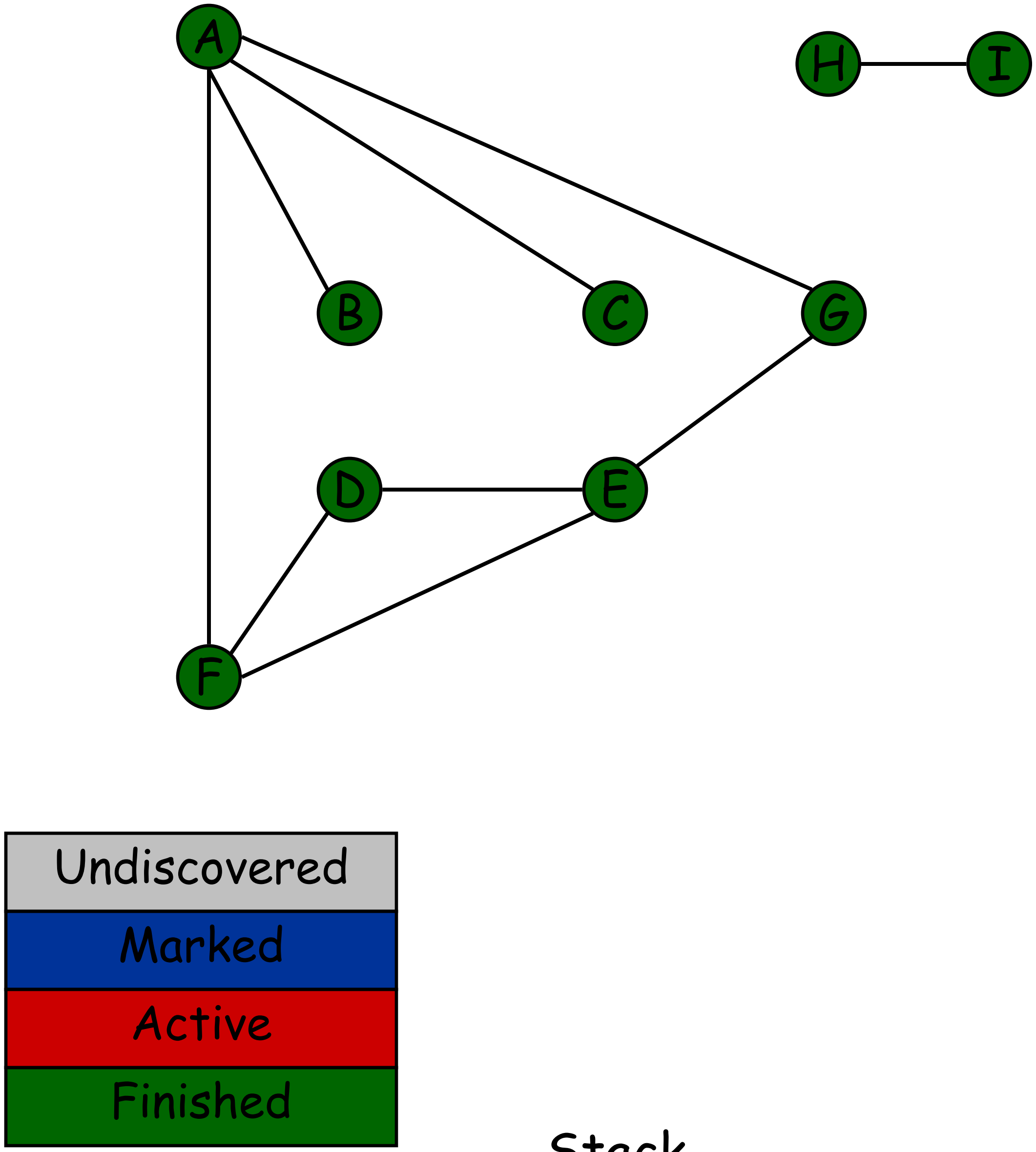



| |
|---|
| Undiscovered |
| Marked |
| Active |
| Finished |

# Undirected Depth First Search



# Undirected Depth First Search




Stack