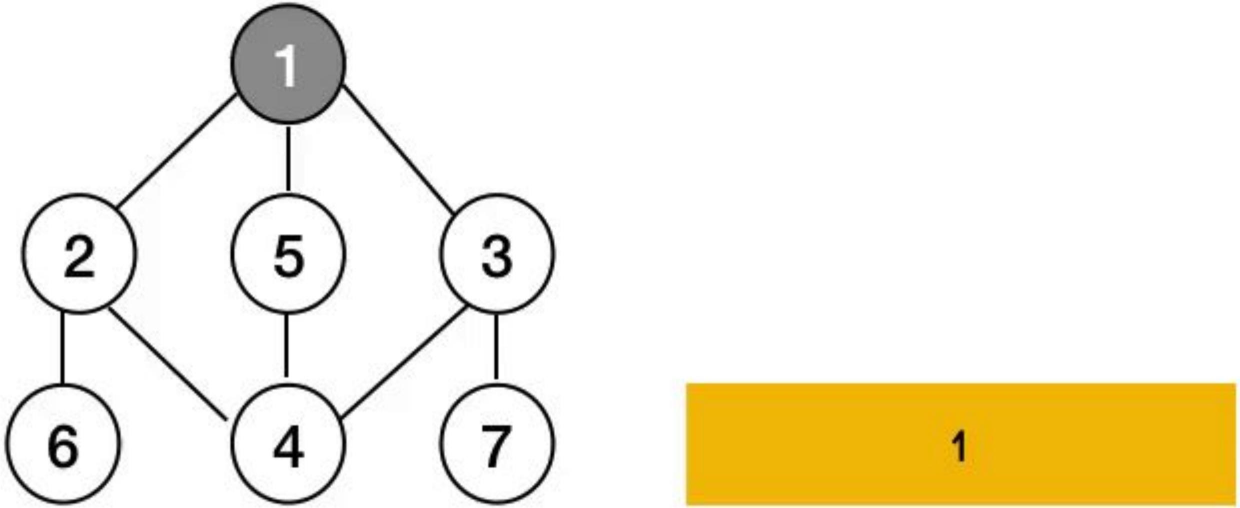
DFS

(Depth First Search)



Lecture Flow

1) Pre-requisites
2) Problem definitions and uses
3) Different approaches
4) Applications of DFS
5) Pair Programming
6) Things to pay attention(common pitfalls)
7) Practice questions
8) Resources
9) Quote of the day

Prerequisites

- Recursion
- Graph
- Stack

Objectives

- Learn about DFS graph traversal

- Learn different operations we can do on graphs using DFS

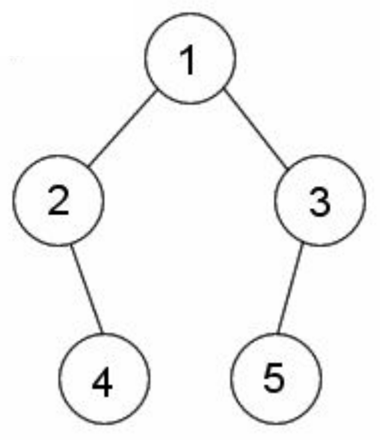
- Learn about applications of DFS

Definition

- DFS is a graph **traversal algorithm**.
- It **aims to visit all nodes or vertices** of a graph in a systematic way.

Definition

- The algorithm starts at a particular node, known as **the source or starting node**, and **explores** as far as possible along each branch before **backtracking**.



Implementation`

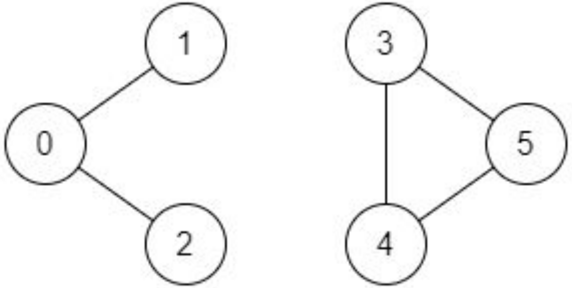
```
def dfs(vertex, visited):
    # base case


    visited.add(vertex)
    for neighbour in graph[vertex]:
        if neighbour not in visited:
            dfs(neighbour, visited)
```

When to Use

- Finding Connected Components
- Detecting Cycles
- Path Finding
- Maze Solving
- Solving Puzzles
- Generating Permutations
- Topological Sorting

Recursive and Iterative Approach of Implementing DFS

Find if Path Exists in Graph



Easy 2.8K 144

Companies

There is a **bi-directional** graph with `n` vertices, where each vertex is labeled from `0` to `n - 1` (**inclusive**). The edges in the graph are represented as a 2D integer array `edges`, where each `edges[i] = [`u_i`, `v_i`]` denotes a bi-directional edge between vertex u_i and vertex v_i. Every vertex pair is connected by **at most one** edge, and no vertex has an edge to itself.

You want to determine if there is a **valid path** that exists from vertex `source` to vertex `destination`.

Given `edges` and the integers `n`, `source`, and `destination`, return `true` *if there is a* ***valid path*** *from* `source` *to* `destination`*, or* `false` *otherwise.*

Recursive Approach

Recursive Implementation

- Since it's a recursive implementation we have to obey the **3 rules of recursive functions**.

- What are those 3 rules?

Rule 1: State

- We need to know what node we are on.

- We need to keep track of the visited nodes.

```
def dfs(node, visited):
```

Rule 2: Base case

- For the base case, when should we stop the recursion when **the current node is our target**.

- Alternatively, we can also finish when we have **traversed over all the nodes**.

```
if node == destination:
    return True
```

Rule 3: Recurrence relation

- We want to **traverse all the adjacent nodes**.

```
for neighbour in graph[node]:
    found = dfs(neighbour)

    if found:
        return True
```

Build Graph

```python
def validPath(self, n, edges,
    source, destination):

    graph = defaultdict(list)

    for node1, node2 in edges:
        graph[node1].append(node2)
        graph[node2].append(node1)

    return dfs(source)
```

Graph traversal

```
def dfs(node):
    if node == destination:
        return True

    for neighbour in graph[node]:
        found = dfs(neighbour)

        if found:
            return True
    return False
```

What’s wrong with the above code?

Build Graph

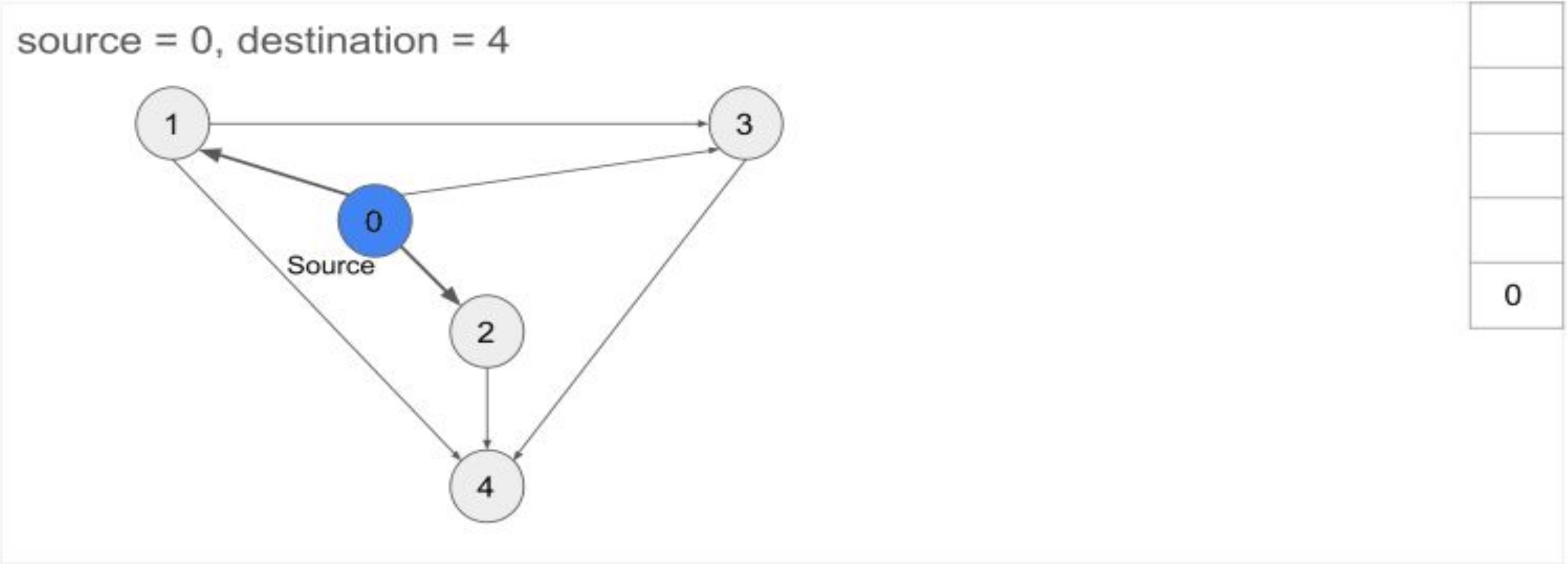
```
def validPath(n, edges,
    source, destination)

    graph = defaultdict(list)

    for node1, node2 in edges:
        graph[node1].append(node2)
        graph[node2].append(node1)

    visited = set()
    return dfs(source, visited)
```

Graph traversal

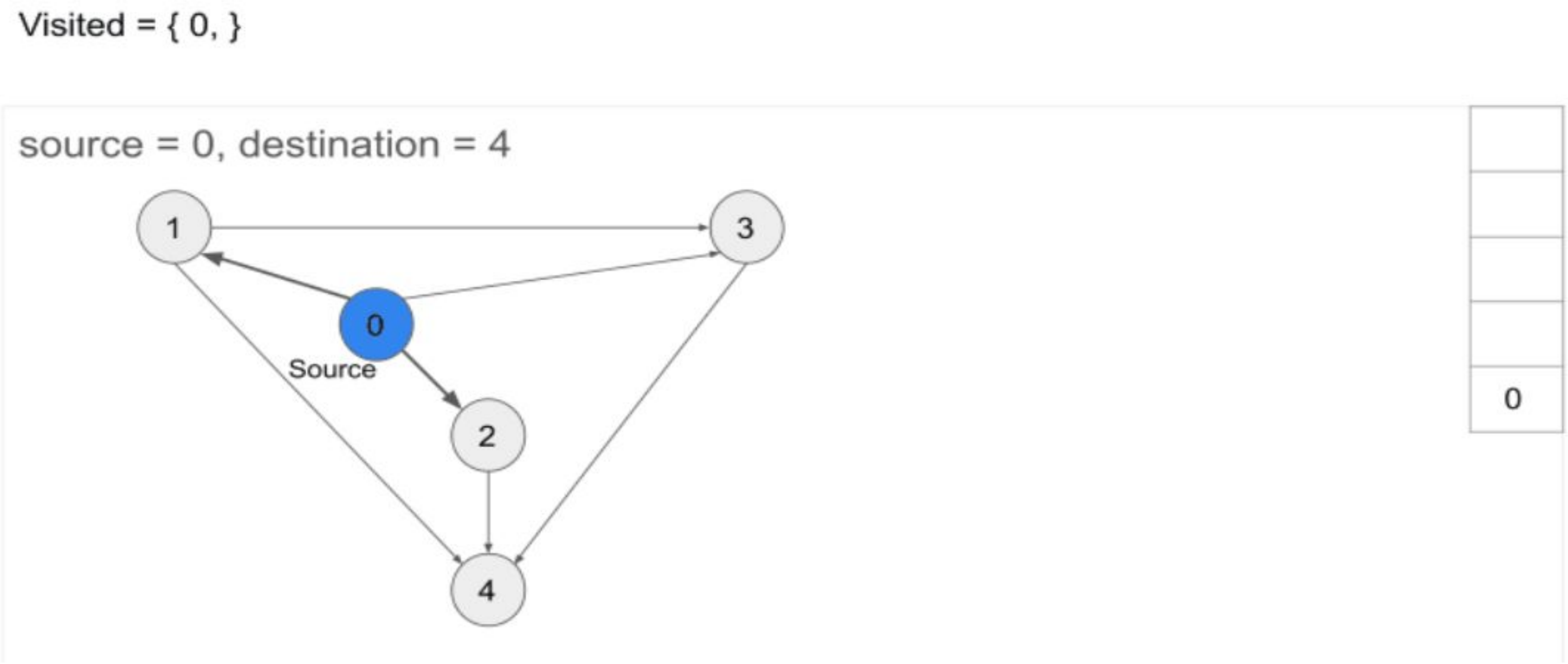
```
def dfs(node, visited):
    if node == destination:
        return True

    visited.add(node)

    for neighbour in graph[node]:
        if neighbour not in visited:
            found = dfs(neighbour, visited)
            if found:
                return True
    return False
```

Visualization

Visited = { 0, }

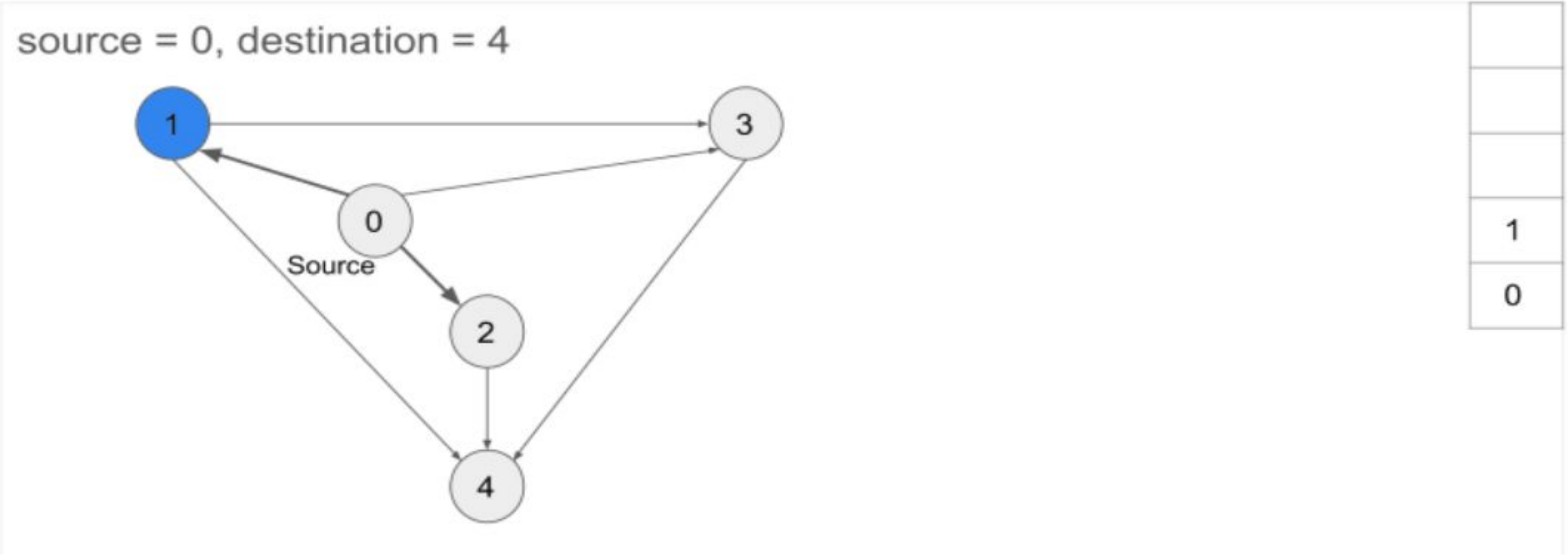


Visualization



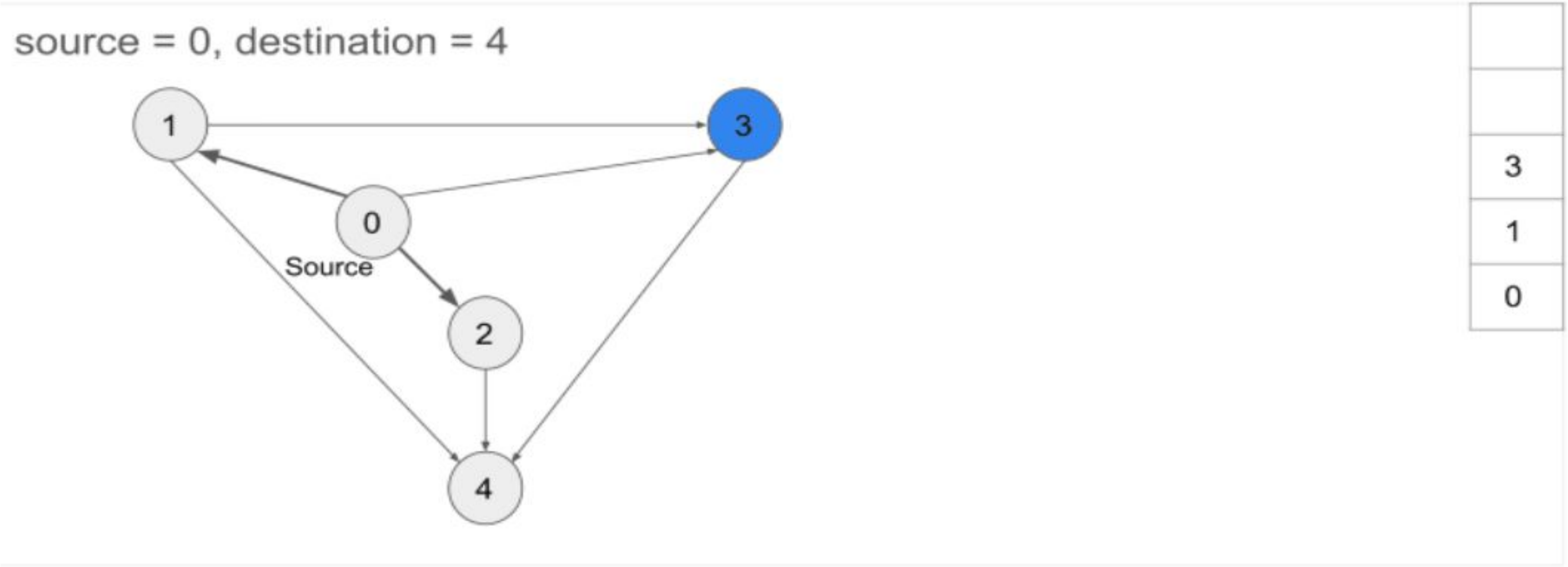
Visualization

Visited = { 0, 1, }



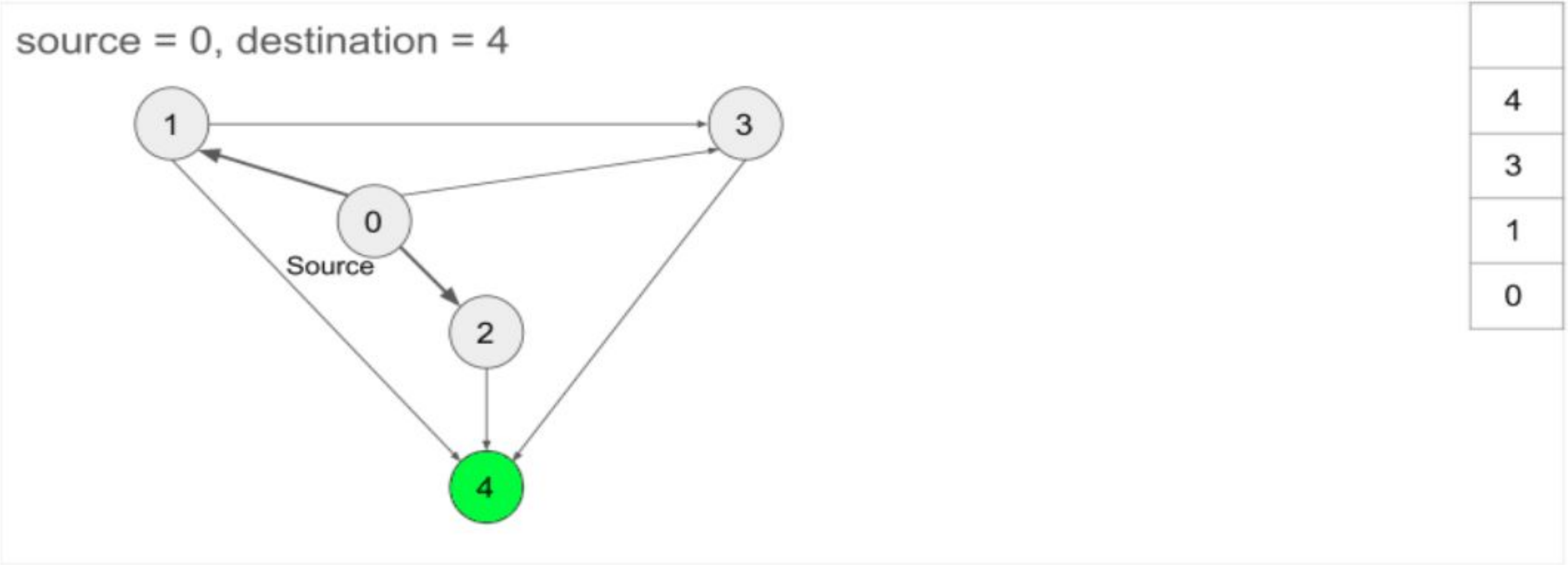
Visualization

Visited = { 0, 1, 3, }



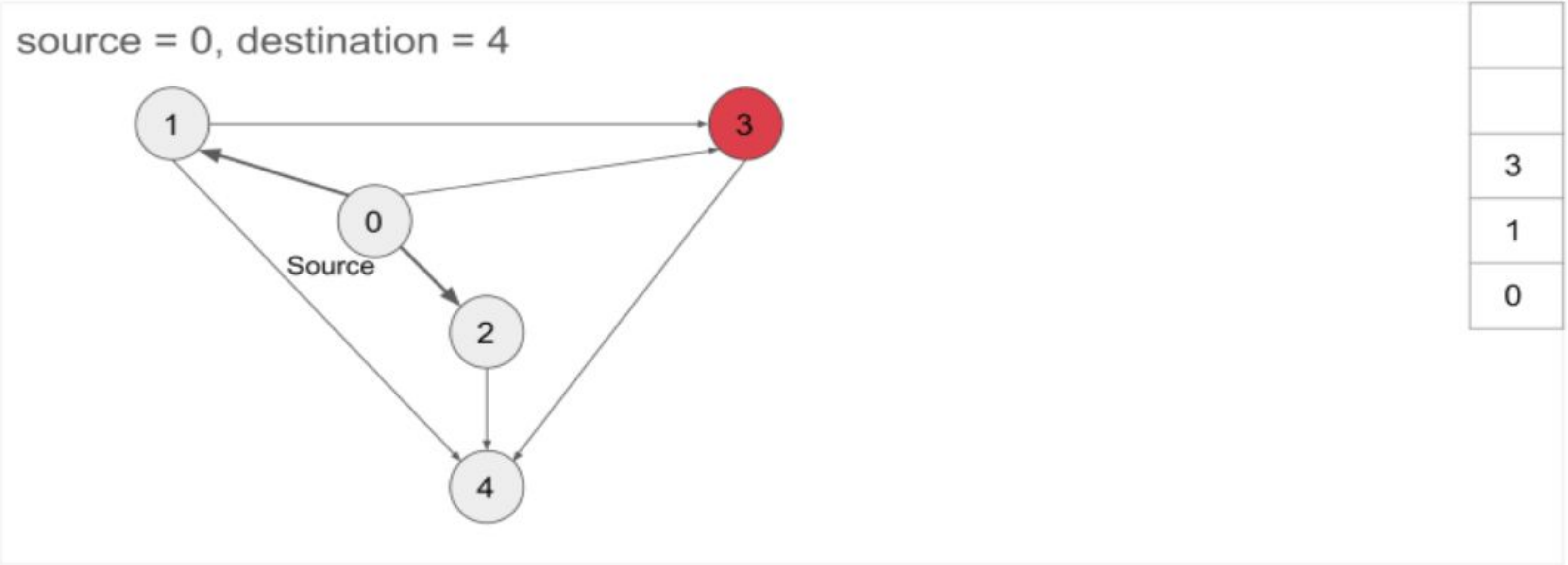
Visualization

Visited = { 0, 1, 3, 4 }



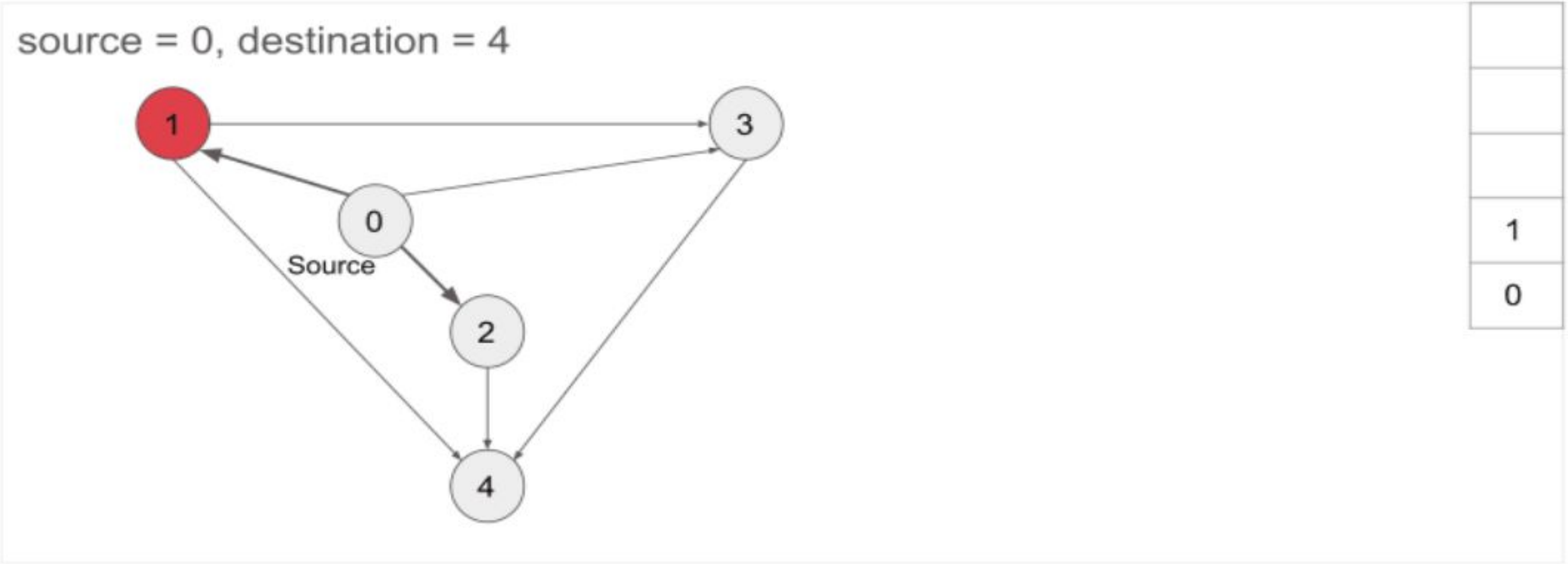
Visualization

Visited = { 0, 1, 3, 4 }



Visualization

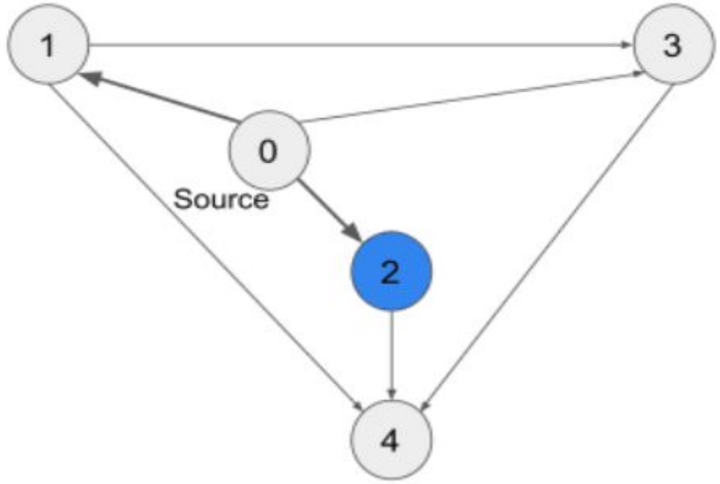
Visited = { 0, 1, 3, 4 }



Visualization

Visited = { 0, 1, 3, 4, 2 }

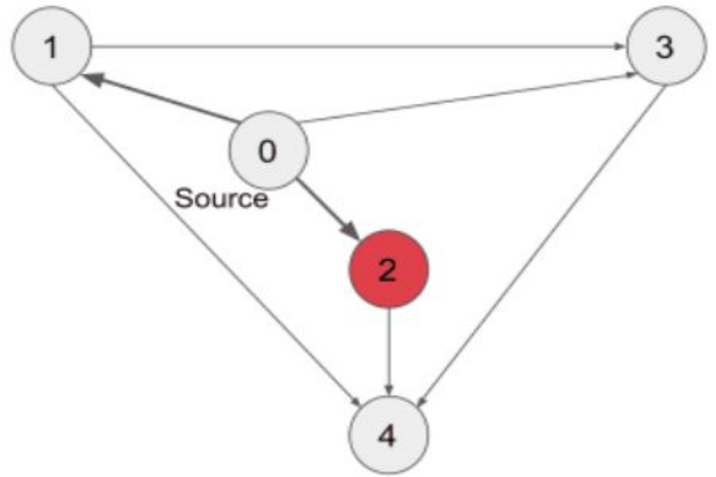
source = 0, destination = 4



Visualization

Visited = { 0, 1, 3, 4, 2 }

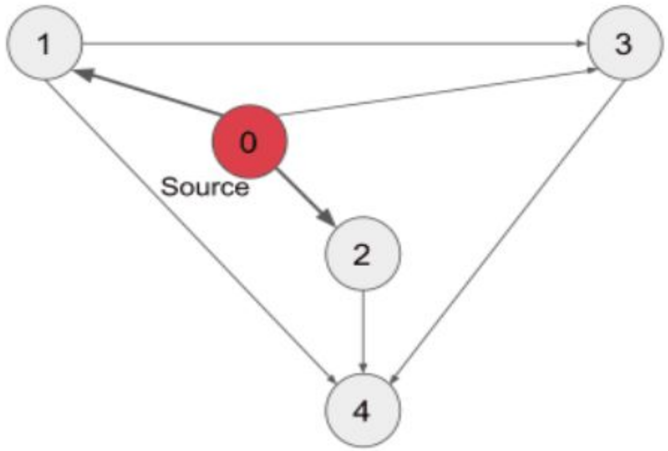
source = 0, destination = 4



Visualization

Visited = { 0, 1, 3, 4, 2 }

source = 0, destination = 4



Time Complexity

- We have **V nodes** and **E edges**
- We will only **visit a node once and edge once**

  Time Complexity = **O (V + E)**

- Note: if the graph is a complete graph, the time complexity is

  = **O(V + (V * (V - 1)))**

  = **O(V^2)**

Space Complexity

- We have **V nodes** and **E edges**
- We will store **the nodes**

  Space Complexity = **O (V)**

- Why not O(V + E)?
- Does it matter if the graph is complete or not?

Iterative Approach

"The Iterative implementation is just the recursive implementation done iteratively."

- Mahatma Gandhi

As such it obeys the 3 rules as well.

Rule 1: State

- Since we don’t have access to the **call stack** we need our own stack to keep track of the current state.

- For the state, we only need to **keep track of the node**, because the **visited set will be kept track of separately**.

```
stack = [source]
visited = set([source])
```

Rule 2: Base case

- The base case is **similar to the recursive** implementation.
- We only need to know if the **current node is the target node**.
- Alternatively, we can also **finish** when we have **visited all the nodes.**

```
if node == destination:
    return True
```

Rule 3: Iteration relation

- We **visit** any **adjacent vertices** that have not yet been visited.

- For each **unvisited adjacent vertex**, we **mark it** as visited and **push it** onto the **stack**.

```
for neighbour in graph[node]:
    if neighbour not in visited:
        stack.append(neighbour)
        visited.add(neighbour)
```

We will **continue to run** the loop **until** we reach the **base case** or until we **visit all the nodes**

we can visit by **starting from the source node.**

Implementation

```
Class Solution:
    def validPath(self, n: int, edges: List[List[int]], source: int, destination: int) -> bool:
        graph = defaultdict(list)

        for node1, node2 in edges:
            graph[node1].append(node2)
            graph[node2].append(node1)

        stack = [source]
        visited = set([source])

        while stack:
            node = stack.pop()
            if node == destination:
                return True

            for neighbour in graph[node]:
                if neighbour not in visited:
                    stack.append(neighbour)
                    visited.add(neighbour)

        return False
```

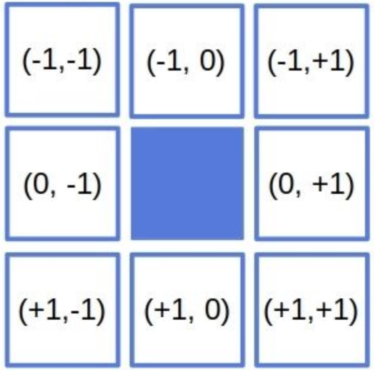
DFS on grid

DFS on grid

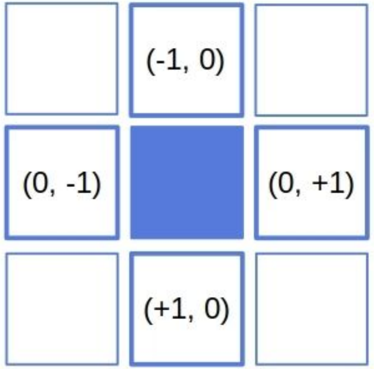
- **Grid vertices** are cells, and edges connect adjacent ones.
- DFS on a grid **starts at a cell**, **visits its neighbors**, and **repeats**.
- The process continues **until all cells are visited.**

Direction vectors

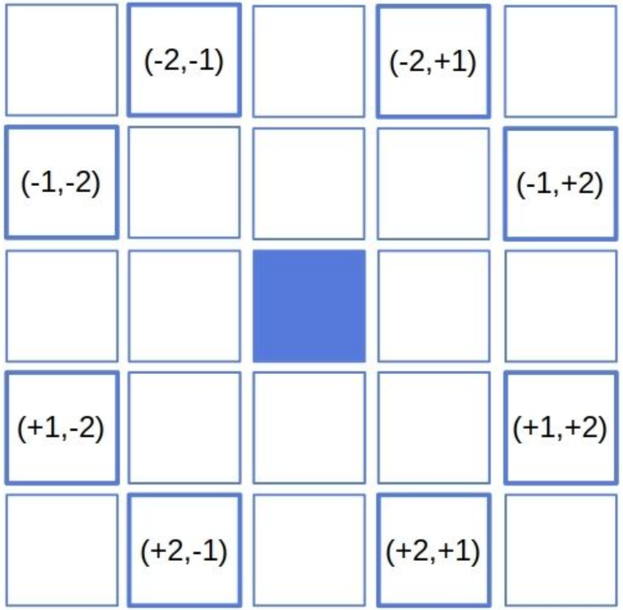
The standard square neighborhood



Only vertical and horizontal neighbors



The chess knight neighborhood



Code

```
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

visited = [[False for i in range(len(grid[0]))] for j in range(len(grid))]


def inbound(row, col):

     return (0 <= row < len(grid) and 0 <= col < len(grid[0]))

def dfs(grid, visited, row, col):
    # base case

    visited[row][col] = True


    for row_change, col_change in directions:

        new_row = row + row_change

        new_col = col + col_change

        if inbound(new_row, new_col) and not visited[new_row][new_col]:

            dfs(grid, visited, new_row, new_col)
```

Practice Problem

DFS Applications

Path Finding

- DFS is a graph **traversal algorithm** that can be used for **pathfinding**.
- DFS works by **starting at a vertex** and **exploring** as far as possible along **each branch** before backtracking.

Path Finding

- It is possible that DFS will find a **longer path before finding the shortest one.**

- While DFS can be used for pathfinding, it may **not always be the most efficient** or accurate method.

Pair Programming

Check if There is a Valid Path in a Grid

Medium 705 285 Add to List Share

You are given an `m x n` `grid`. Each cell of `grid` represents a street. The street of `grid[i][j]` can be:

- `1` which means a street connecting the left cell and the right cell.
- `2` which means a street connecting the upper cell and the lower cell.
- `3` which means a street connecting the left cell and the lower cell.
- `4` which means a street connecting the right cell and the lower cell.
- `5` which means a street connecting the left cell and the upper cell.
- `6` which means a street connecting the right cell and the upper cell.


Street 1


Street 2

You will initially start at the street of the upper-left cell `(0, 0)`. A valid path in the grid is a path that starts from the upper left cell `(0, 0)` and ends at the bottom-right cell `(m - 1, n - 1)`. **The path should only follow the streets**.

Notice that you are **not allowed** to change any street.

Return `true` *if there is a valid path in the grid or* `false` *otherwise*.

Implementation

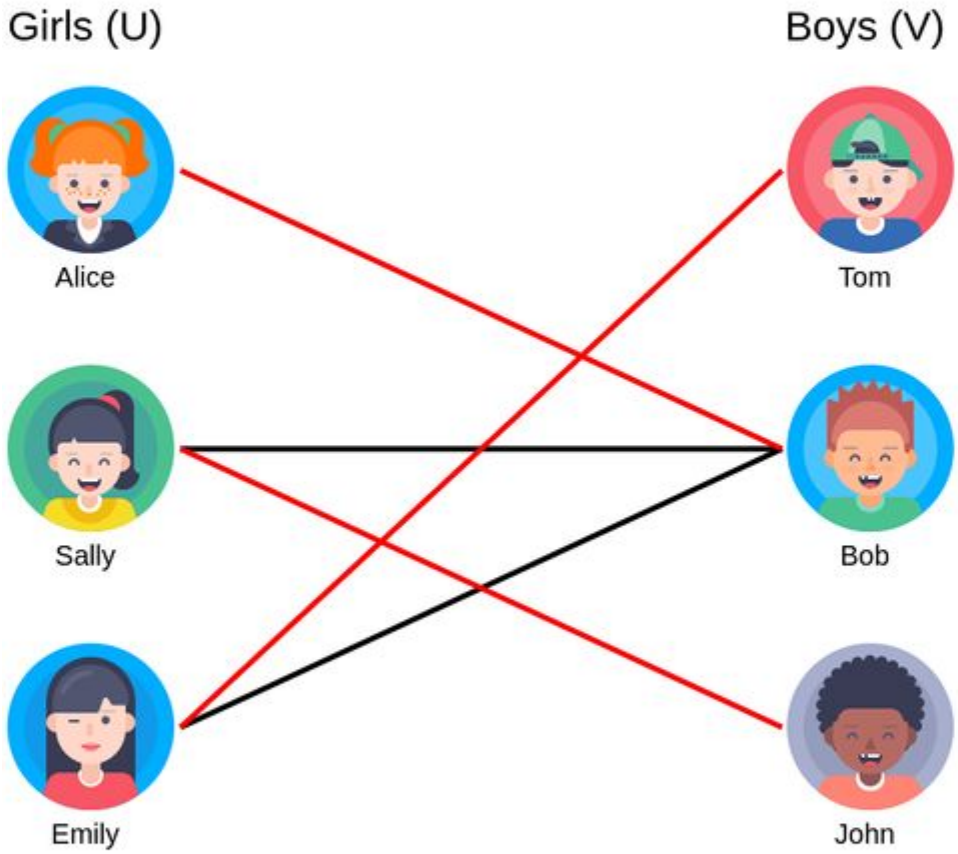
```
def hasValidPath(self, grid):
      destination = (len(grid)-1, len(grid[0]) -
1)
      directions =
            {1: [(0,-1),(0,1)],
            2: [(-1,0),(1,0)],
            3: [(0,-1),(1,0)],
            4: [(0,1),(1,0)],
            5: [(0,-1),(-1,0)],
            6: [(0,1),(-1,0)]}

      def inbound(row, col):
            return 0 <= row < len(grid)
            and 0 <= col < len(grid[0])

      visited = set([(0, 0)])
      return dfs(0, 0)
```

```
def dfs(row, col):
      if (row, col) == destination:
            return True

      for row_change, col_change in directions[grid[row][col]]:
            new_row= row + row_change
            new_col = col + col_change

            if (inbound(new_row, new_col) and
              (new_row, new_col) not in visited and
                (-row_change, -col_change) in
                  directions[grid[new_row][new_col]]):

                  visited.add((new_row, new_col))
                  found = dfs(new_row, new_col)
                  if found:
                        return True

                  return False
```

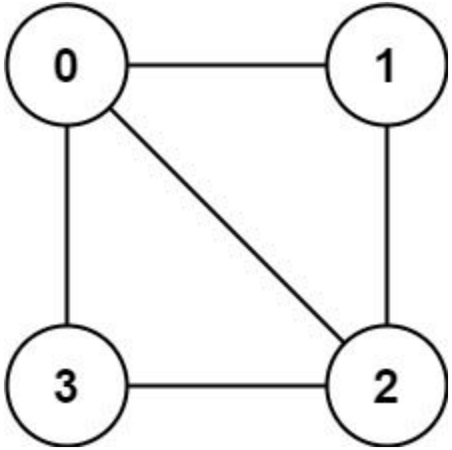
Determine if a graph is bipartite or not?

Bipartite Graph

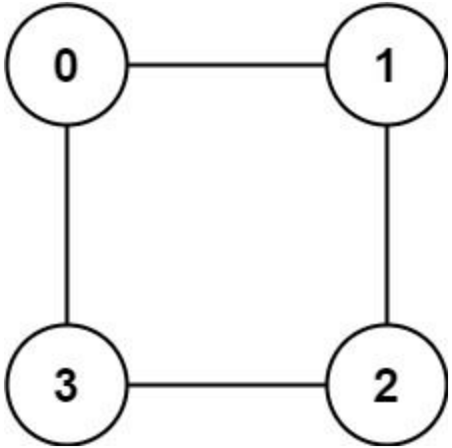
- Bipartite graphs have two sets of nodes.
- Each edge connects nodes from different sets.



Is this graph Bipartite?



Is this graph bipartite?

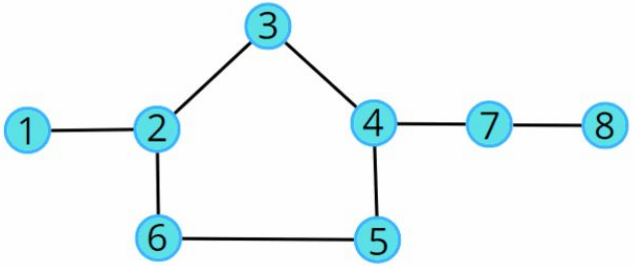


How do we identify if a graph is bipartite or not using dfs?

We use **DFS Coloring**

DFS Coloring

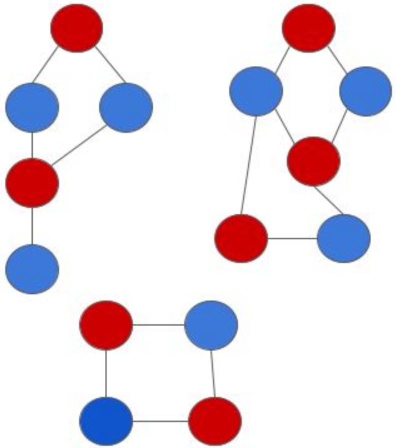
- Assign a color to **each vertex** of a graph in such a way that **no two adjacent vertices have the same color.**



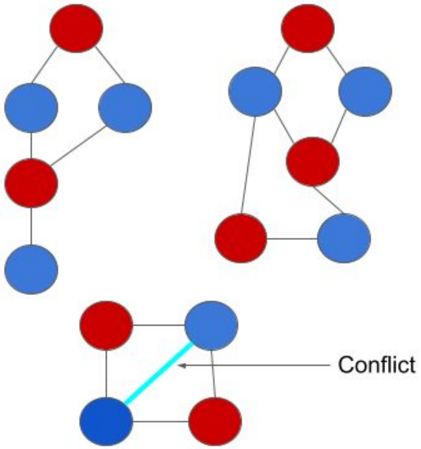
1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1

Colour Array

Is a graph bipartite?



Bipartite Graph



No possible way to split

Practice Problem

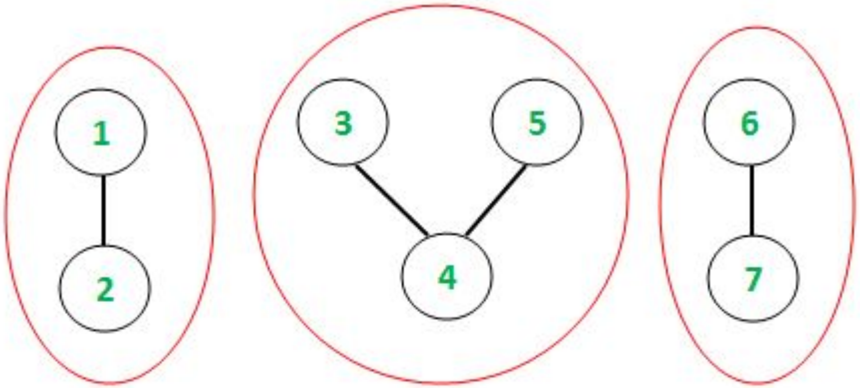
Implementation

```
def isBipartite(self, graph, n):
    color = [-1 for _ len(n)]
    result = True
    for node in range(n):
        if color[node] == -1:
            color[node] = 0
            result = result and dfs(node, graph)

    return result
```

```python
def dfs(node, graph):
    for neighbour in graph[node]:
        temp = True
        if color[neighbour] == -1:
            if color[node] == 0:
                color[neighbour] = 1
            else:
                color[neighbour] = 0
            temp = temp and dfs(neighbour, graph)
        else:
            return color[node] != color[neighbour]

    return temp
```

Connected components

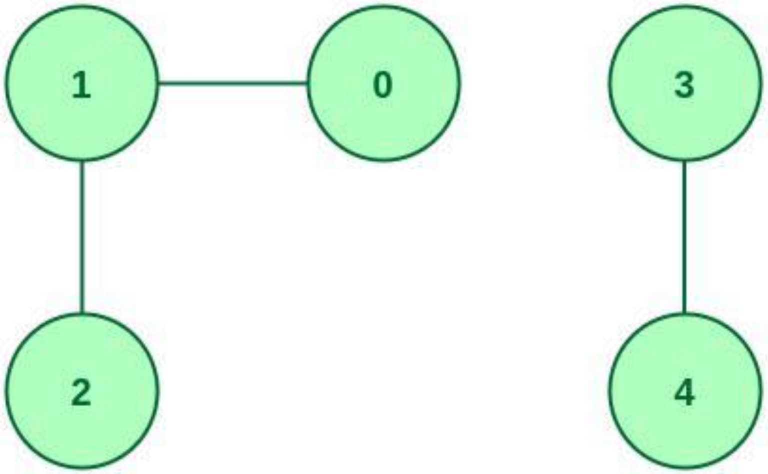
The connected parts of a graph are called its **components**



The **counts** of connected components are - **2, 3 and 2**

Finding Connected Components

A **connected component** of a graph is a **subset of vertices in the graph** such that there is a **path between any two vertices** in the subset.



Brainstorm on how to find connected components.

Number of Islands

Here's how we can use DFS to find the number of islands

1. **Initialize** all **vertices** as unvisited.

2. For each **unvisited vertex**, **perform** a **DFS starting** from that vertex

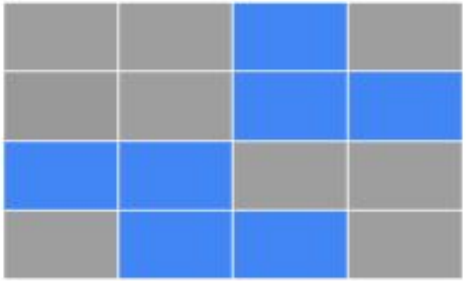
3. Mark all **visited vertices** as part of the **same connected component** as the **starting vertex.**

4. **Repeat** steps 2-3 for any r**emaining unvisited vertices** until **all vertices** have **been visited.**

After this process, **the set of marked vertices** for **each DFS traversal** will give you the **connected components** of the graph.

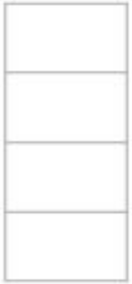
Visualization

Visited = { }



count = 0

Call stack



Implementation

```python
def numIslands(grid):
    rows = len(grid)
    cols = len(grid[0])
    islands = 0
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    def dfs(row, col):
        if row < 0 or row >= rows or col < 0 or col >= cols or grid[row][col] == '0':
            return
        grid[row][col] = '0'
        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            dfs(new_row, new_col)

    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == '1':
                islands += 1
                dfs(i, j)

    return islands
```

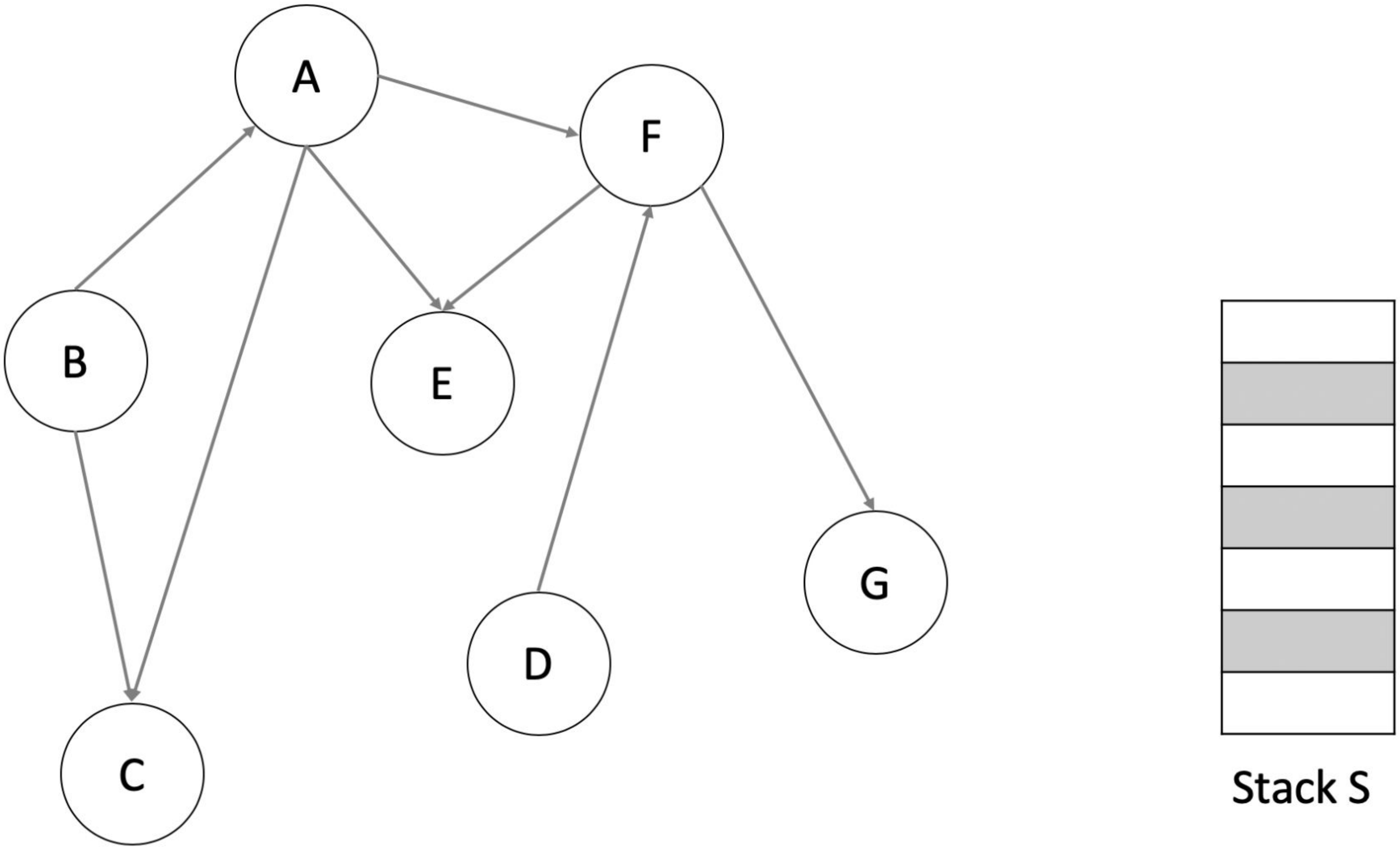
How can we detect cycles in directed graph using dfs?

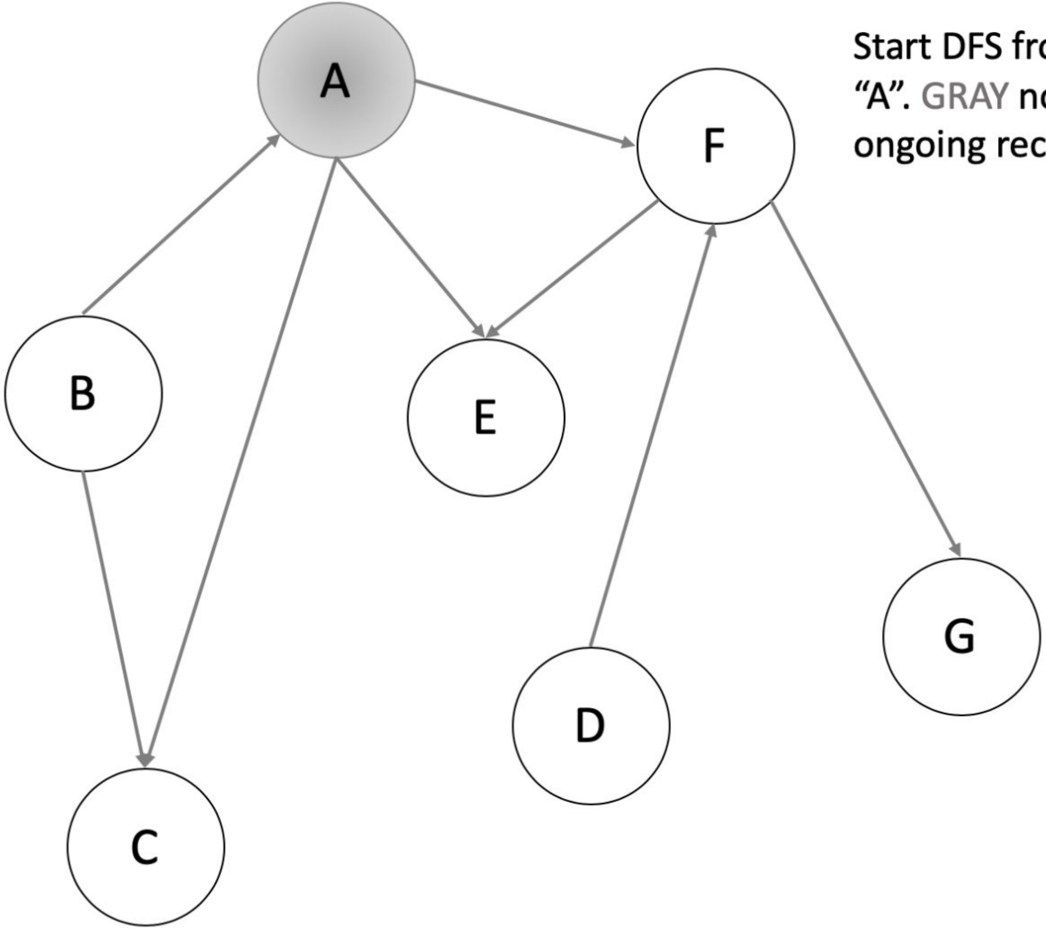
Cycle detection

- We will run a series of DFS in the graph.
- Initially all vertices are colored **white** (0)
- From each unvisited (**white**) vertex, start the DFS, mark it **gray** (1) while entering and mark it **black** (2) on exit
- If DFS moves to a **gray** vertex, then we have found a cycle

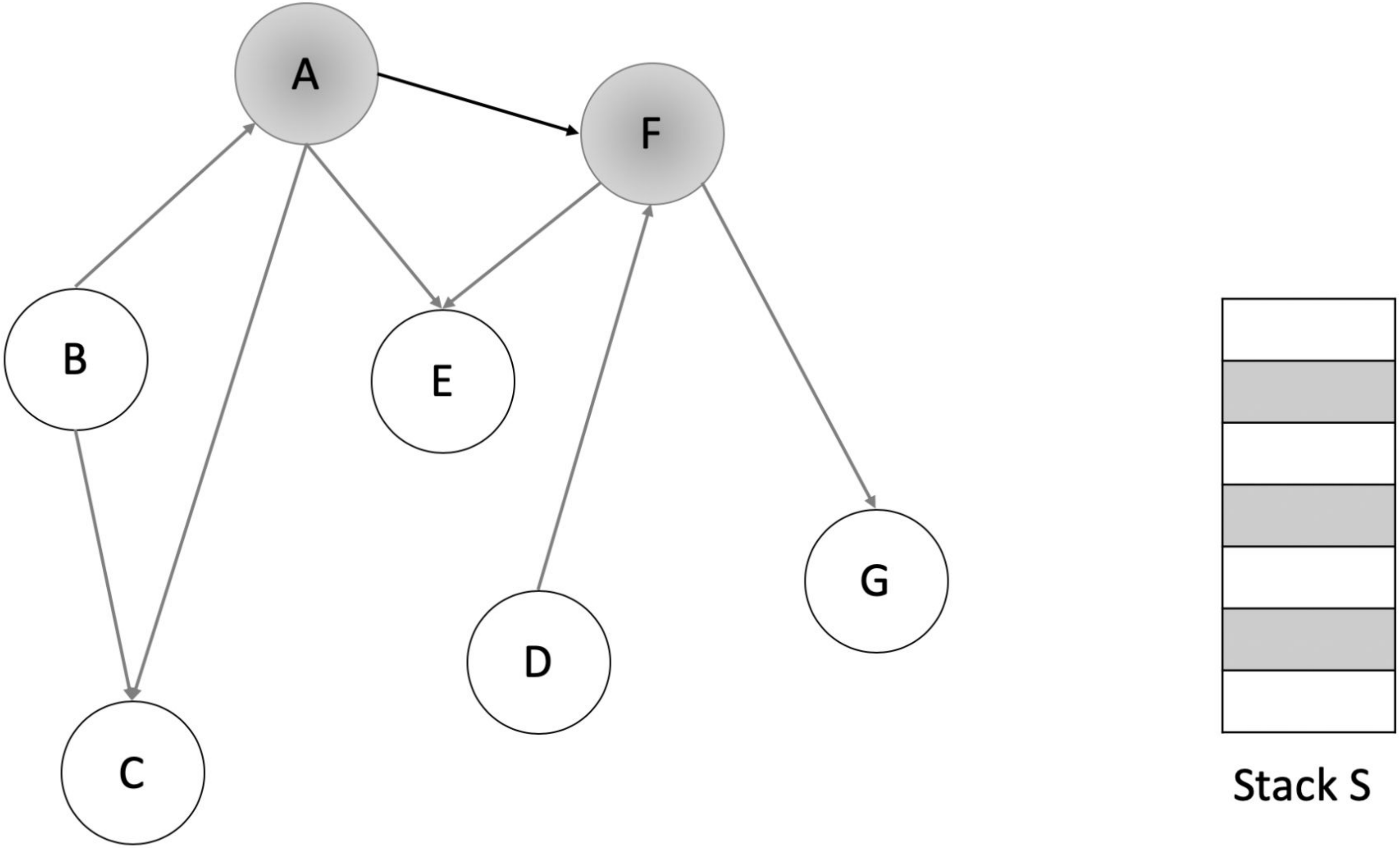
DFS Algorithm

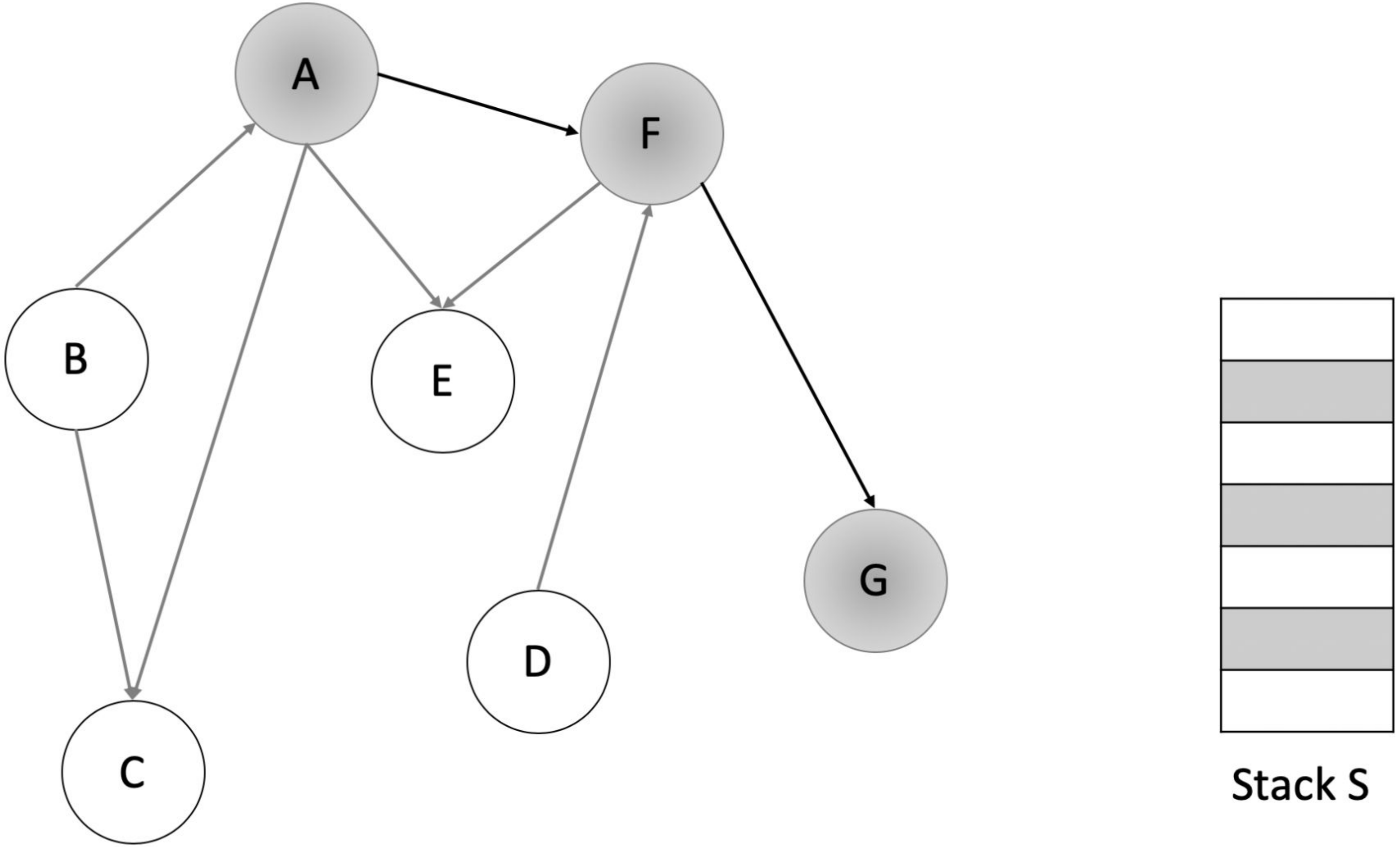
- During traversal:
  - Only traverse to white nodes.
  - If a black node is found, skip it - it has been processed.
  - If a grey node is found, this means there is a cycle. Why?

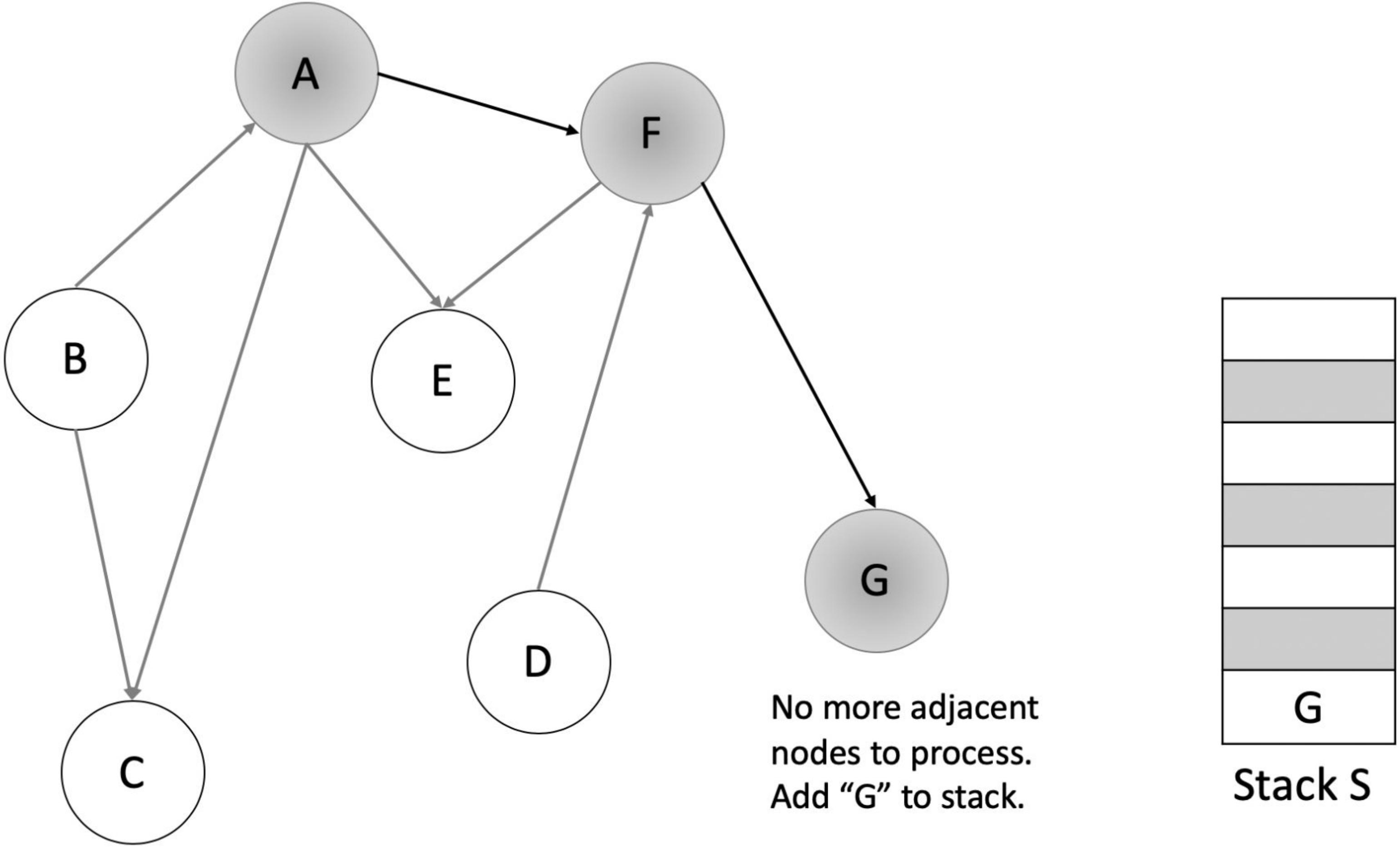

A
F
B
E
G
D
C
Stack S



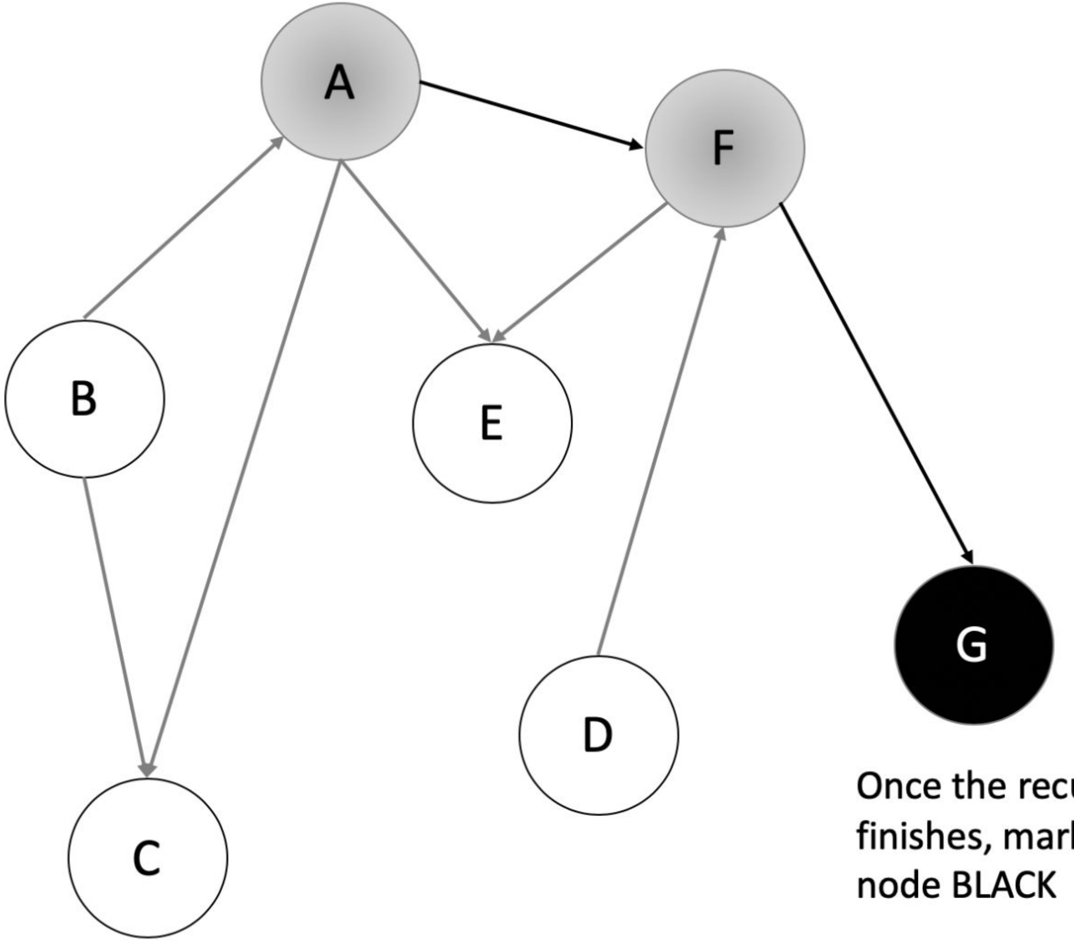
Start DFS from the node "A". GRAY nodes depict ongoing recursion.


A
F
B
E
D
G
C
Stack S


A
F
B
E
G
D
C
Stack S

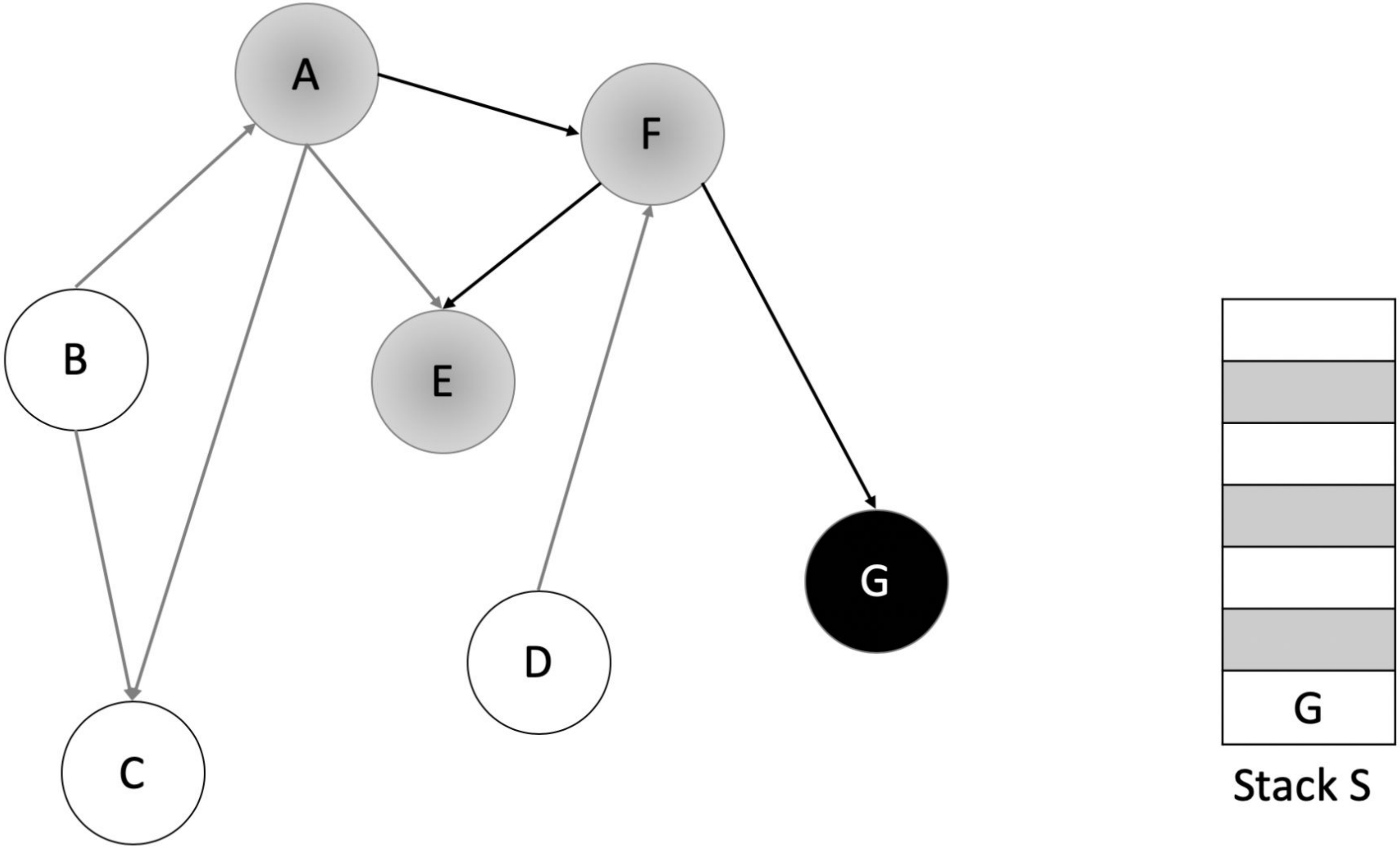


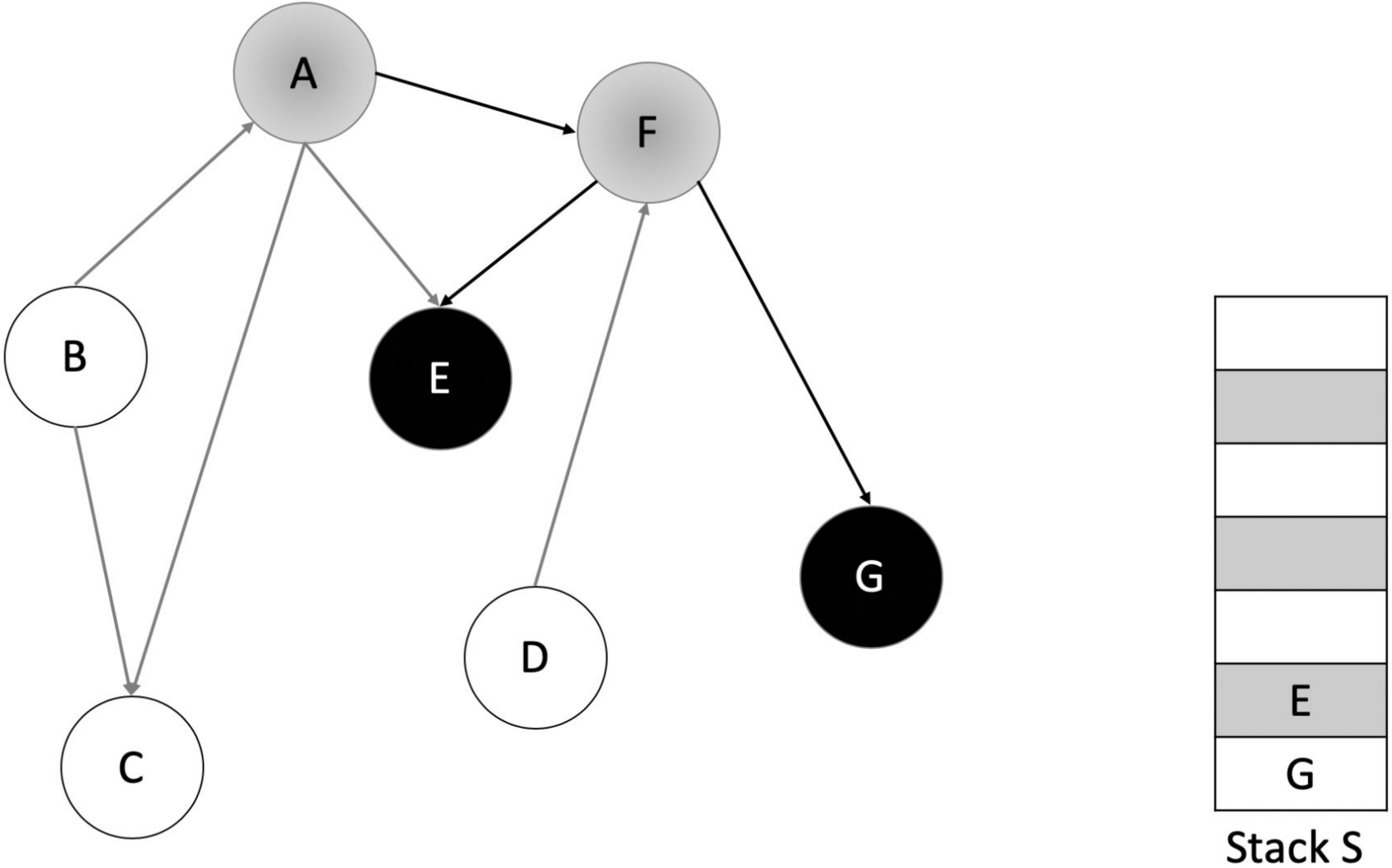
No more adjacent nodes to process.
Add “G” to stack.

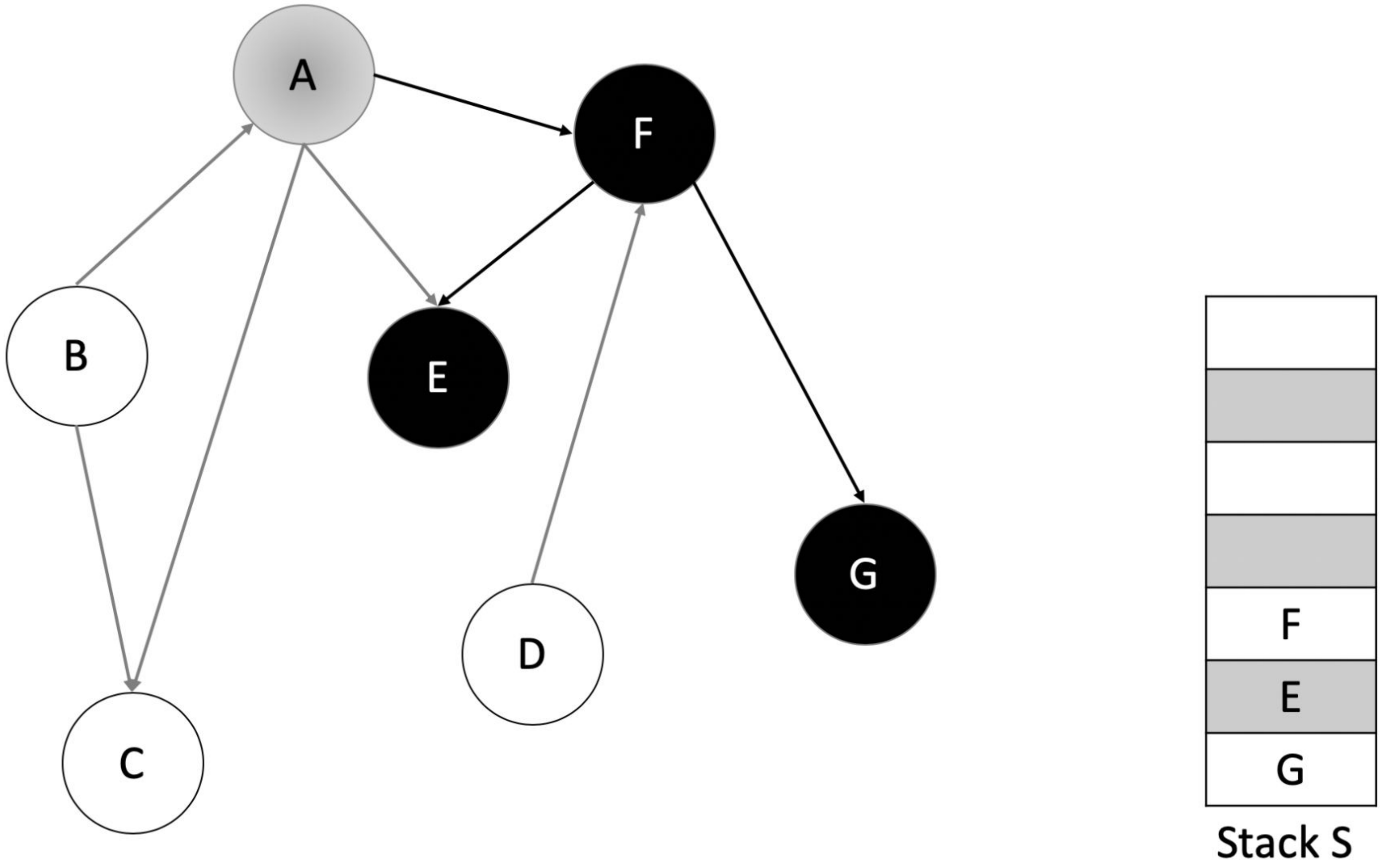


Once the recursion finishes, mark the node BLACK

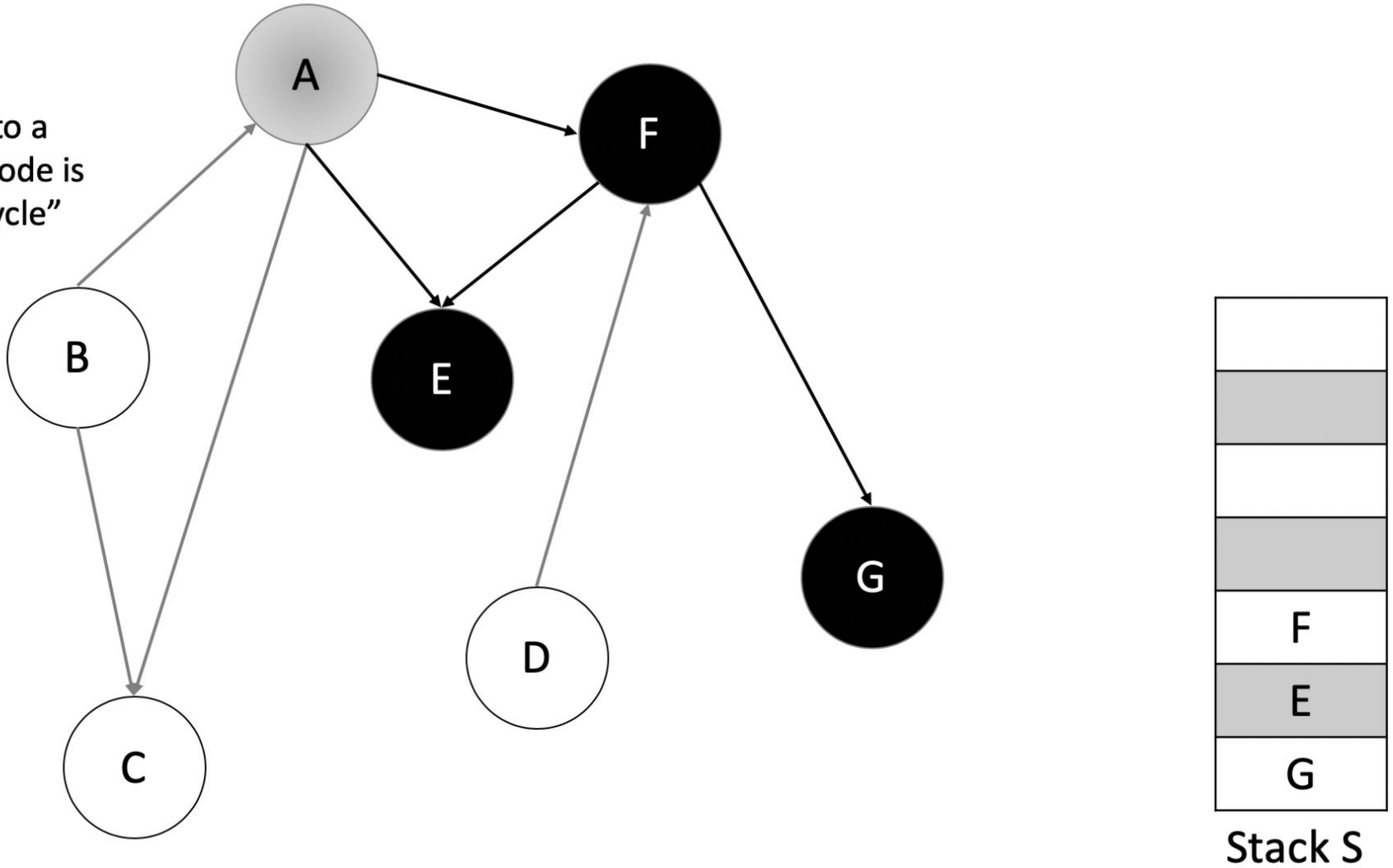
Stack S

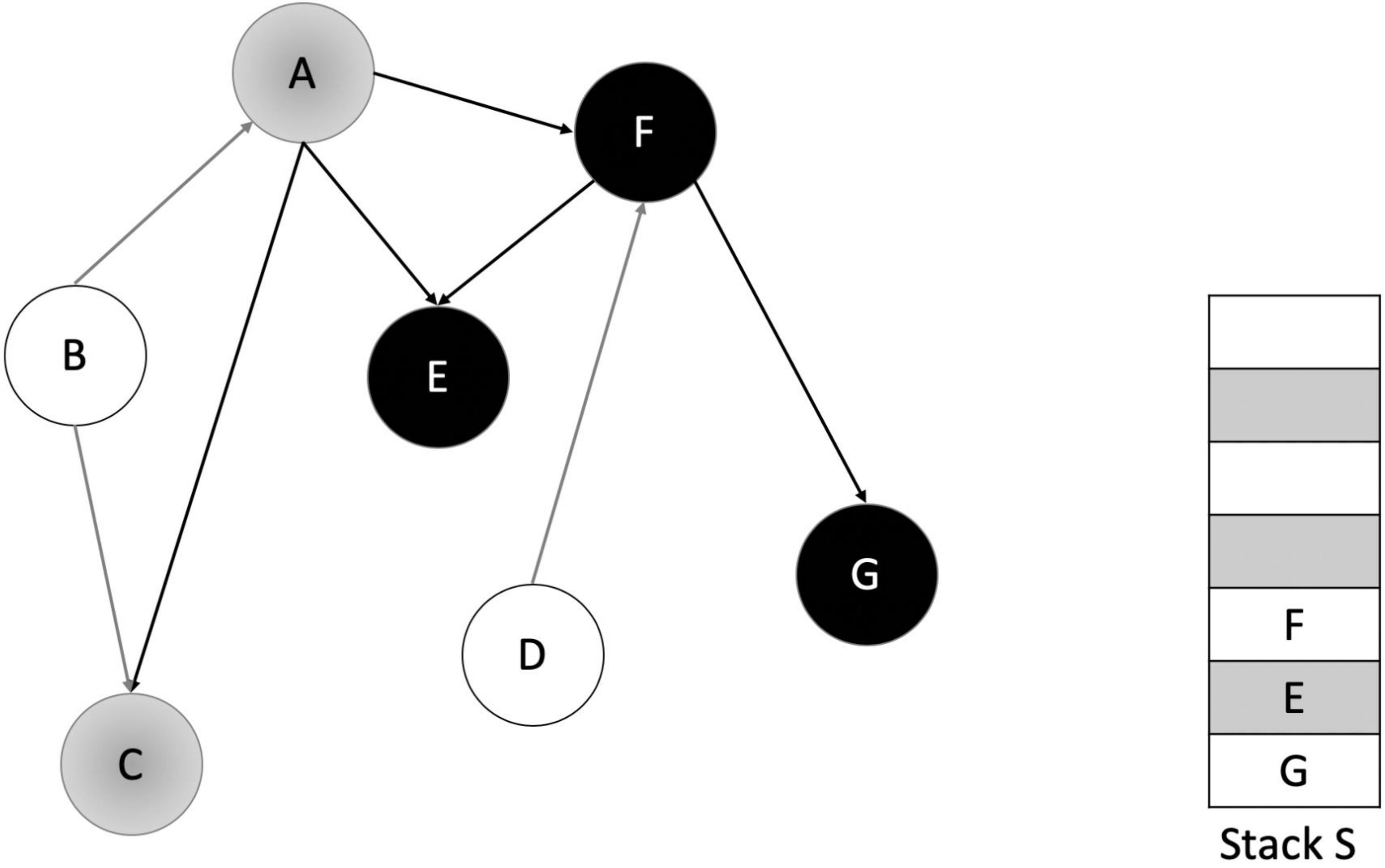

A
F
B
E
G
D
C
G
Stack S

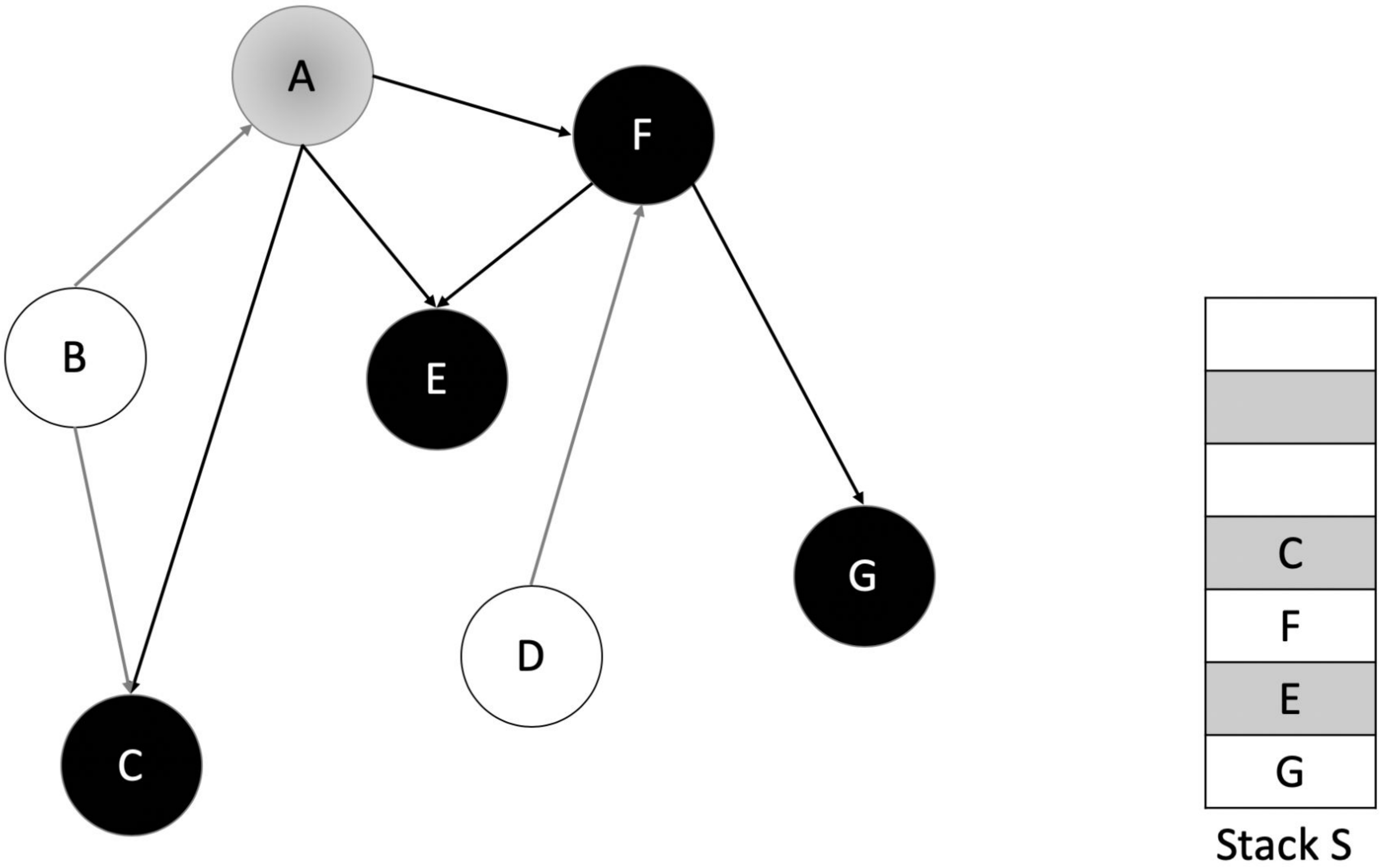

A
F
B
E
D
G
C
E
G
Stack S


A
F
B
E
D
G
C
F
E
G
Stack S

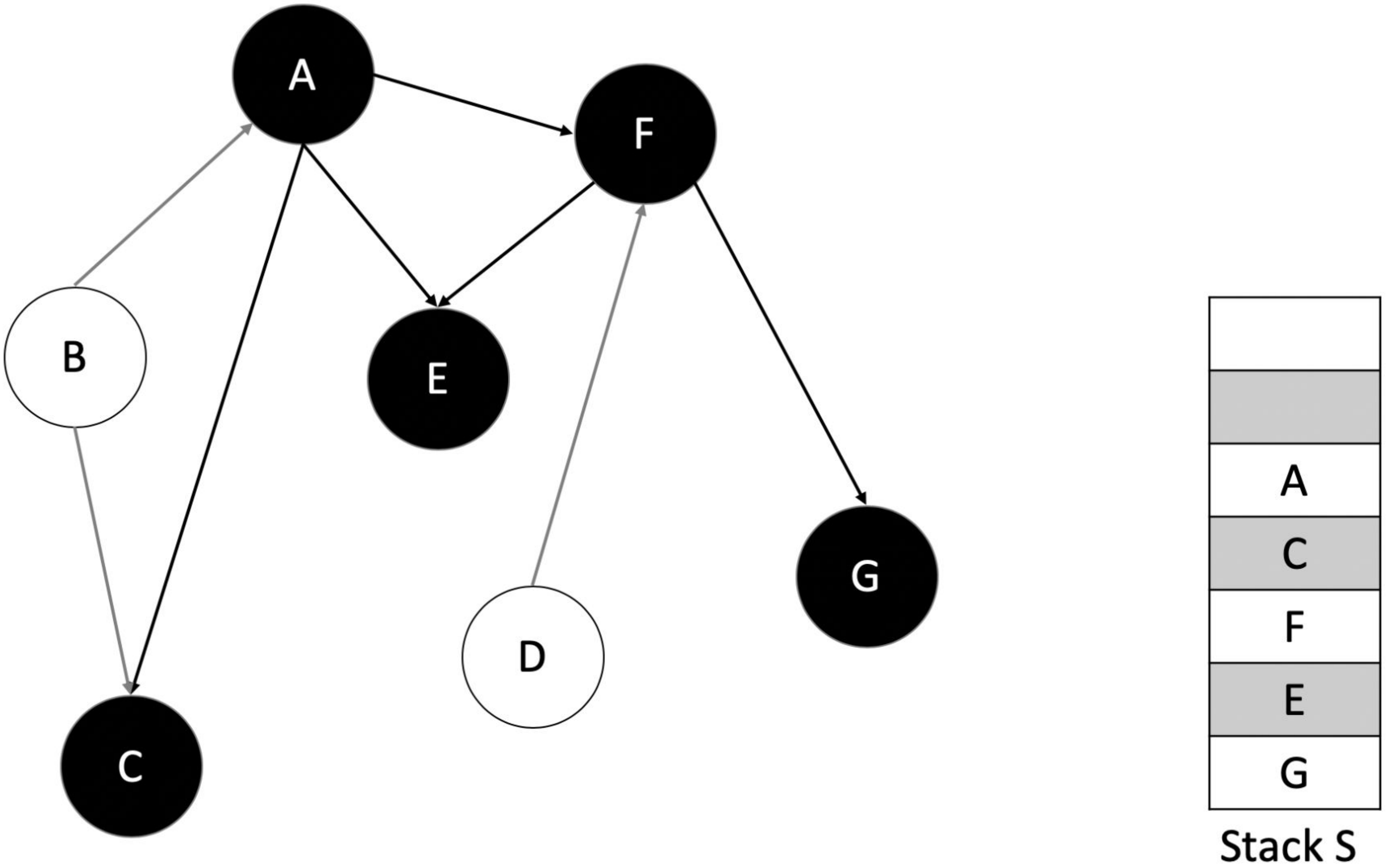
An edge leading to a BLACK node is not a "cycle"



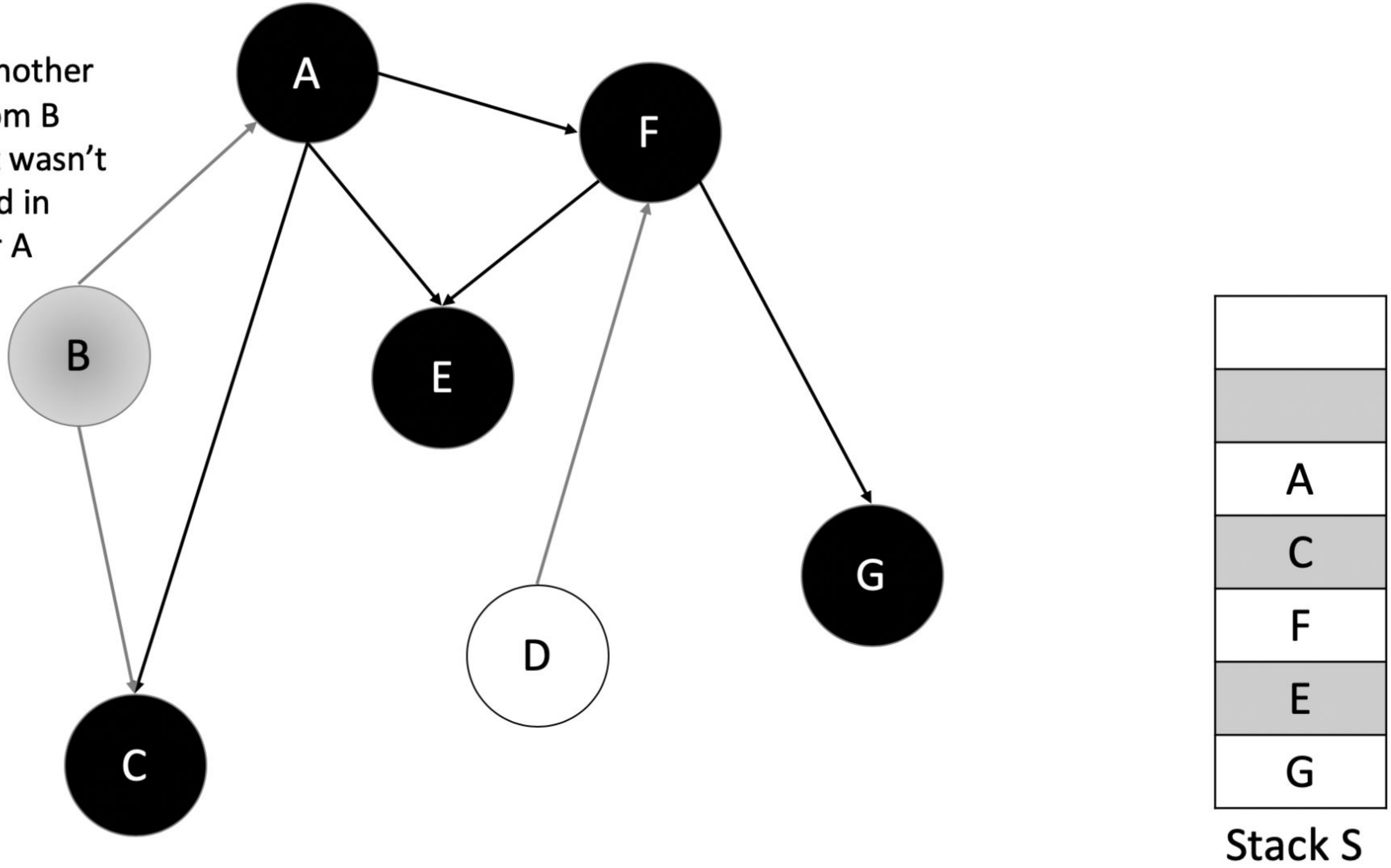

A
F
B
E
D
G
C
F
E
G
Stack S

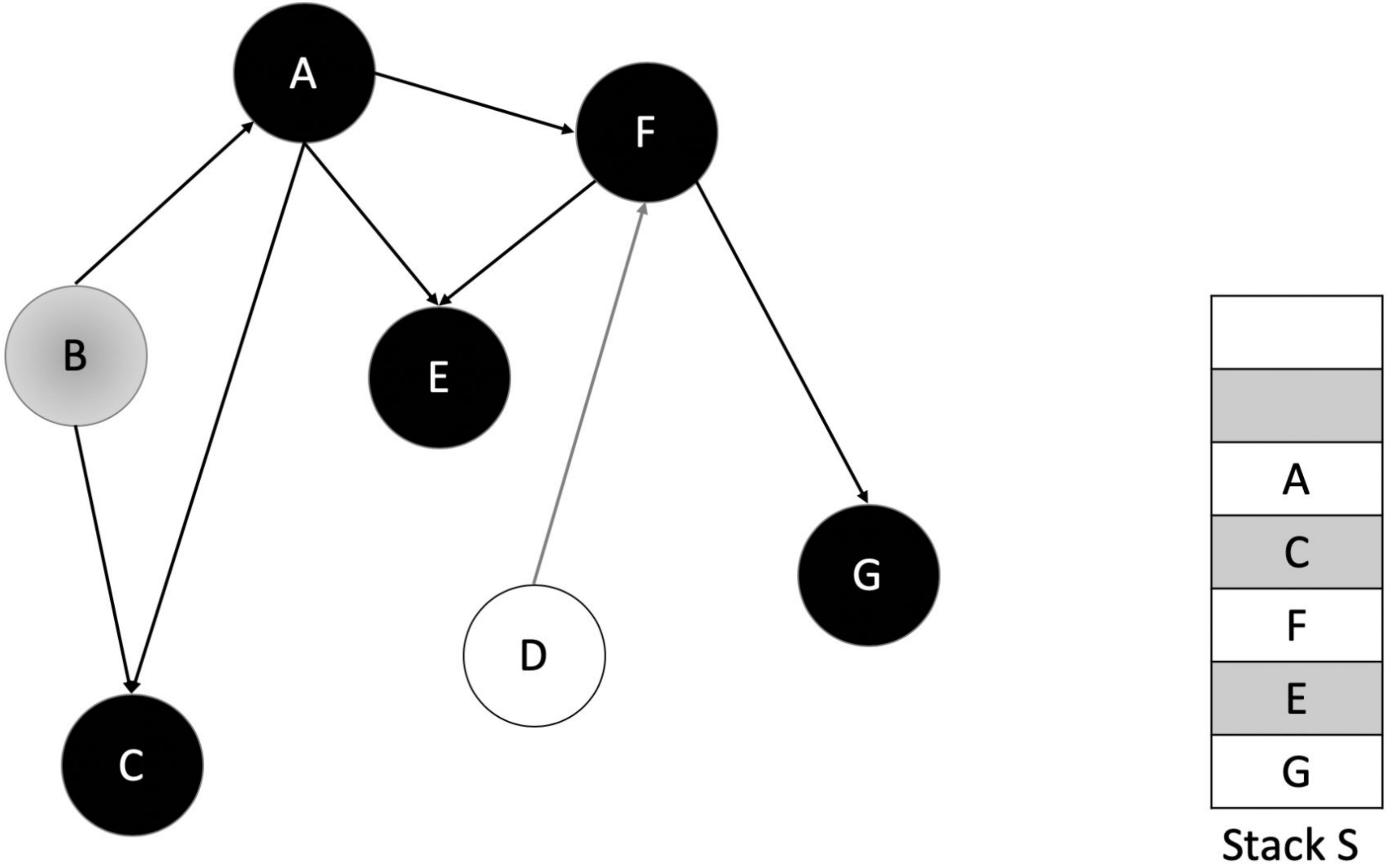


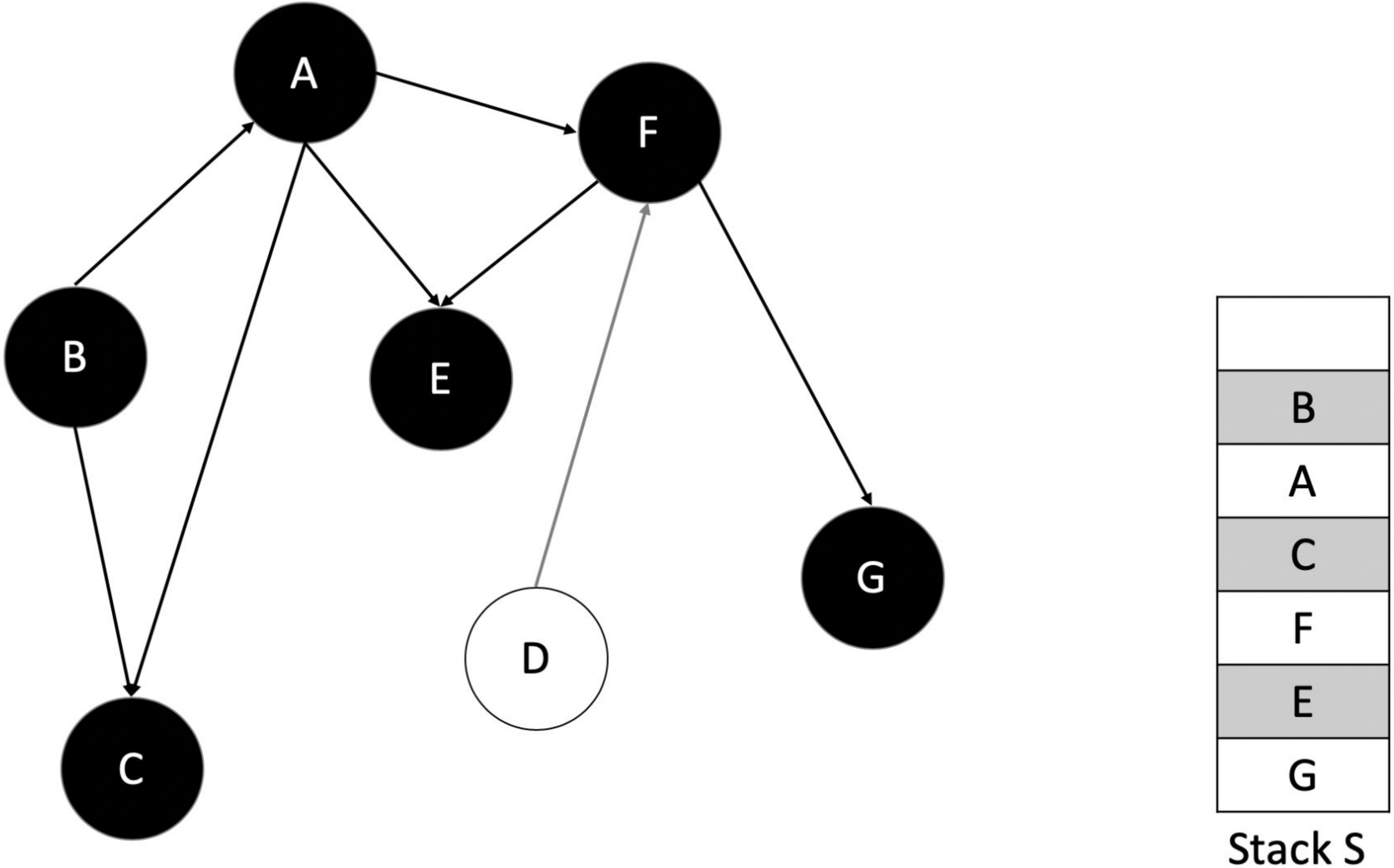
A
F
B
E
D
G
C
C
F
E
G
Stack S

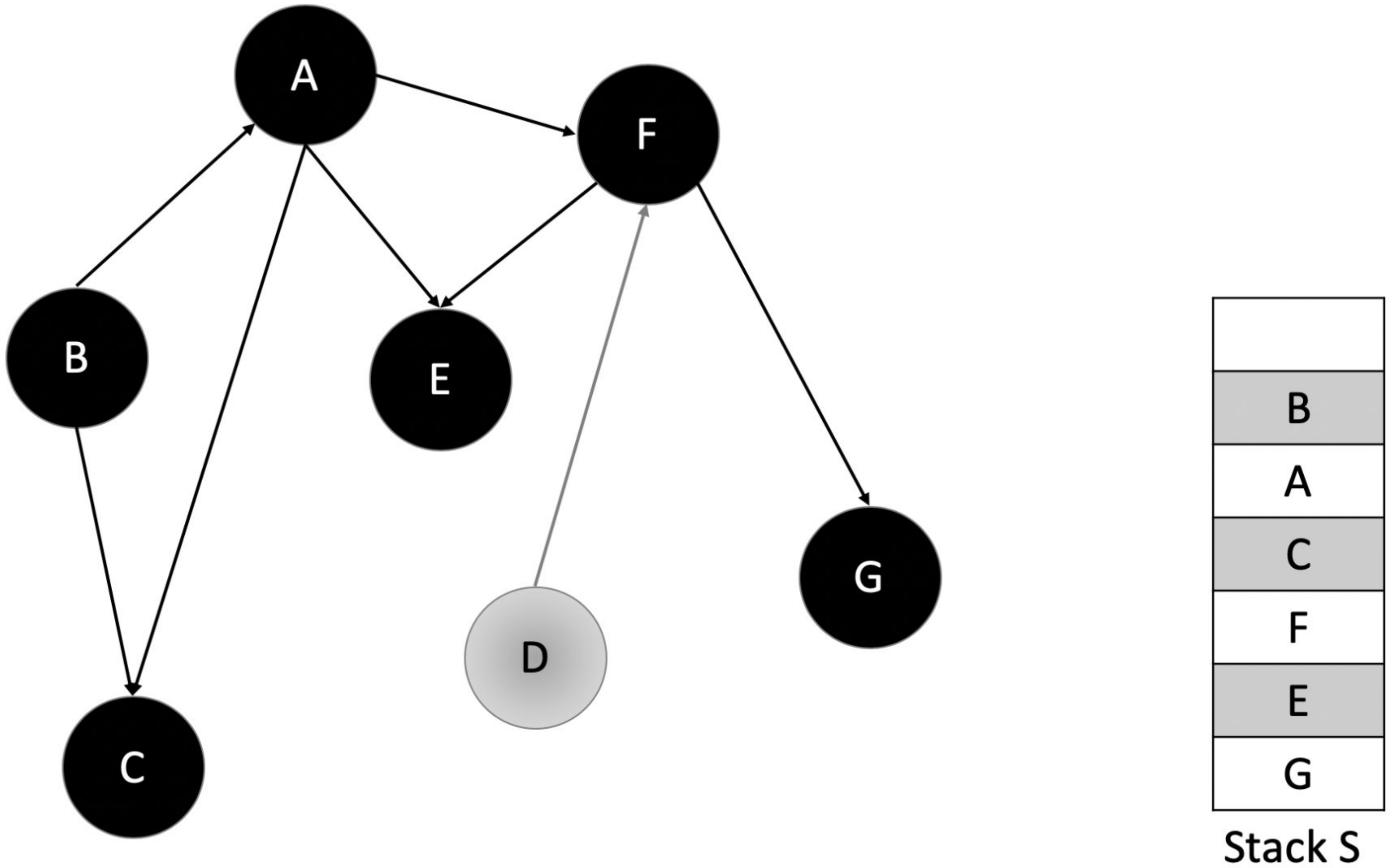

A
F
B
E
D
G
C
A
C
F
E
G
Stack S

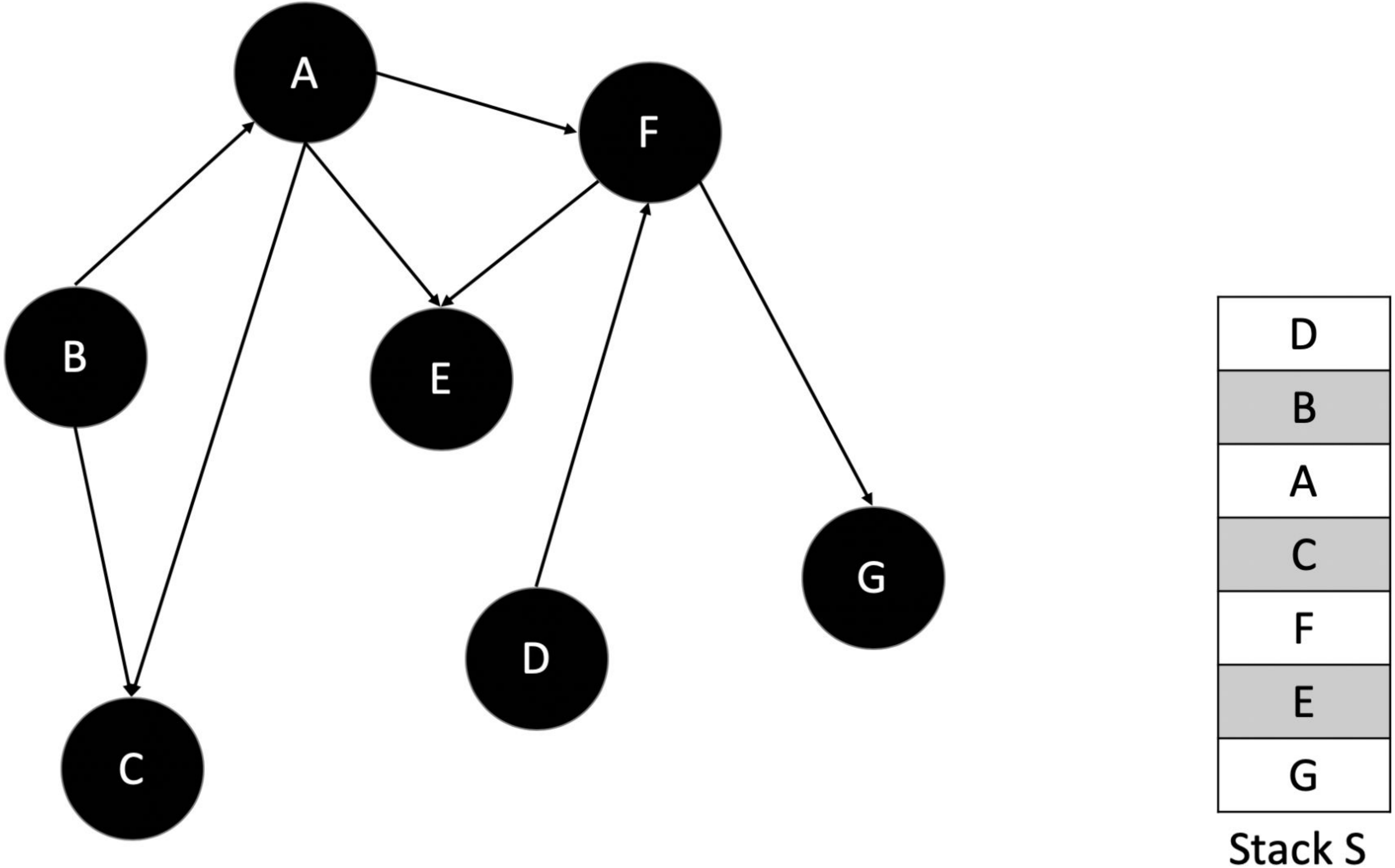
Start another DFS from B since it wasn't covered in DFS for A




A
F
B
E
G
D
C
A
C
F
E
G
Stack S


A
F
B
E
D
G
C
B
A
C
F
E
G
Stack S


A
F
B
E
G
D
C
B
A
C
F
E
G
Stack S


A
F
B
E
D
G
C
D
B
A
C
F
E
G
Stack S

Cycle detection

```python
WHITE = 1
GRAY = 2
BLACK = 3
# By default all vertces are WHITE
color = {k: WHITE for k in range(num_nodes)}
is_possible = True
def dfs(node):
    nonlocal is_possible

    # Don't recurse further if we found a cycle already
    if not is_possible:
        return

    # Start the recursion
    color[node] = GRAY

    # Traverse on neighboring vertices
    if node in adj_list:
        for neighbor in adj_list[node]:
            if color[neighbor] == WHITE:
                dfs(neighbor)
            elif color[neighbor] == GRAY:
                # An edge to a GRAY vertex represents a cycle
                is_possible = False

    # Recursion ends. We mark it as black
    color[node] = BLACK
```

Practice problem

Common Pitfalls

Infinite loops: DFS can get stuck in an infinite loop if it encounters a cycle in the graph. To avoid this, it is important to keep track of visited nodes and avoid revisiting them.

Common Pitfalls

Stack overflow & Exceeding Maximum Recursion Depth

- DFS uses a stack to keep track of nodes to visit. If the graph is very deep or has a large number of branches, the stack can become very large and cause a stack overflow.

- If you are using Python you are aware that the maximum recursion depth is around 1000, in some cases we might have more than 1000 nodes in our call stack in these cases we might be faced by **maximum recursion depth exceeded error**

Common Pitfalls

- To fix the maximum recursion depth exceeded error we can manually increase the recursion limit.

- To fix the stack overflow error we can manually increase the stack size for our python program. Taken together it will look like the image on the right.

```python
import threading
from sys import stdin,stdout,setrecursionlimit
from collections import defaultdict

setrecursionlimit(1 << 30)
threading.stack_size(1 << 27)

def main():
    # Enter your code here
    pass

main_thread = threading.Thread(target=main)
main_thread.start()
main_thread.join()
```

Common Pitfalls

Choosing the wrong starting node: The output of DFS can depend on the starting node. If the starting node is not chosen carefully, it may not be possible to reach some nodes in the graph. It is important to consider the structure of the graph and the problem at hand when choosing the starting node.

Recap

Recap Points

- DFS Definition and Algorithm
- Visual: summary of DFS algorithm on a graph
- DFS Applications

Resources

[GeeksForGeeks]

[Visualization]

Practice Problems

- Employee-importance ✔
- Number-of-provinces ✔
- Sum-of-nodes-with-even-valued-grandparent ✔
- Max-area-of-island ✔
- Evaluate-division ✔
- Sum-root-to-leaf-numbers ✔
- Detonate-the-maximum-bombs ✔
- Surrounded-regions ✔
- Minesweeper ✔
- Lowest-common-ancestor-of-deepest-leaves ✔
- Recover-binary-search-tree ✔

Quote of the day

"Turn your face to the sun and the shadows fall behind you"

- Maori Proverb