# Graph Algorithm Applications

SWE2016-44

# Applications of BFS

1. Shortest Path in a graph
2. Social Network
3. Cycle Detection in undirected graph
4. To test if a graph is bipartite
5. Broadcasting in a network
6. Path Finding

# Applications of BFS

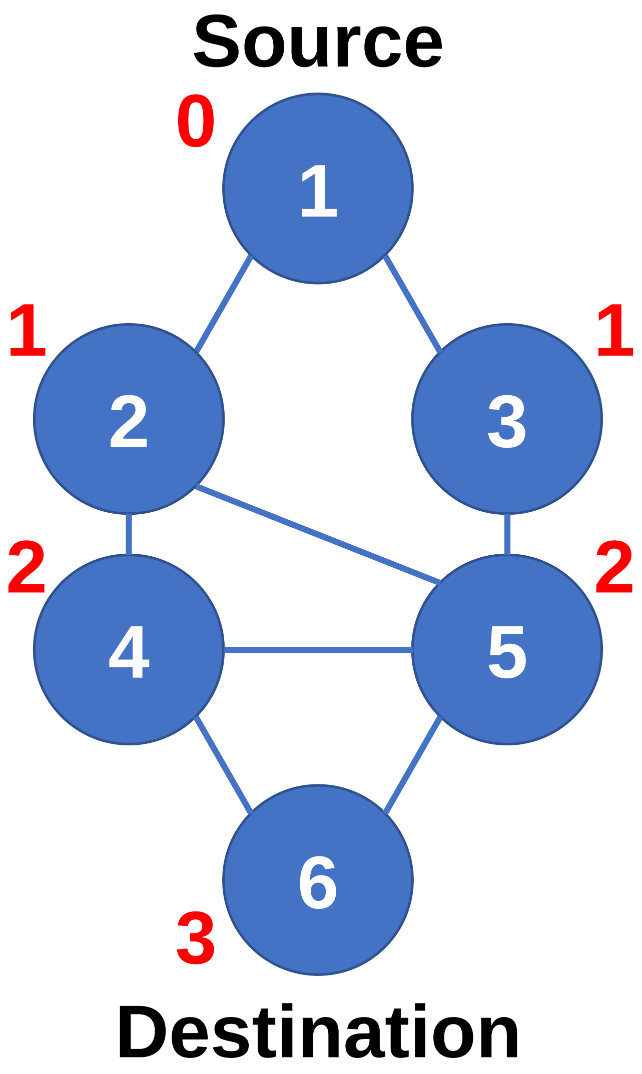
1. <u>Shortest Path in an unweighted graph</u>

**Initialize:**

$$\text{Dist\_Shortest(F)} \begin{cases} 0 \text{ if source=destination} \\ \infty \text{ otherwise} \end{cases}$$

**For each edge E={v, w}:**

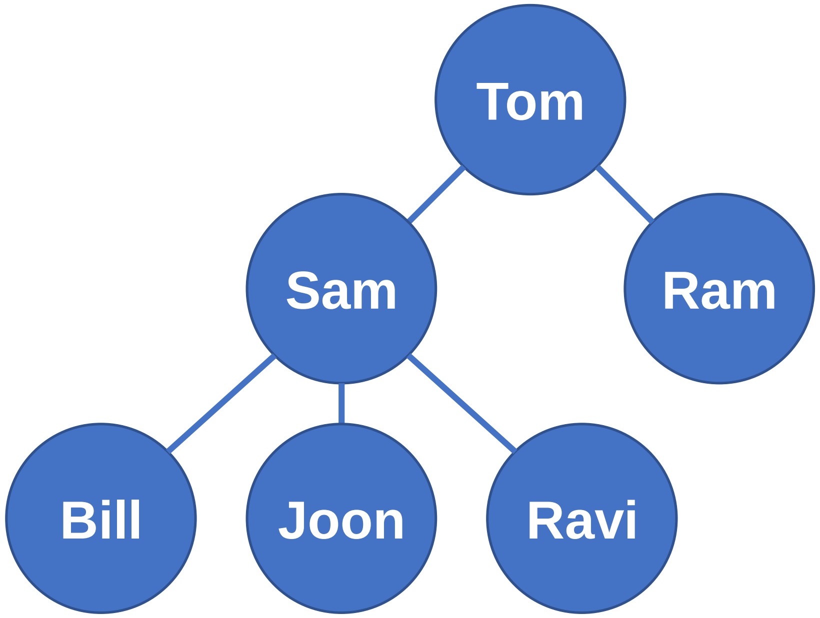
- If w is unvisited,
  Dist_Shortest(w) = Dist_Shortest(v)+1



# Applications of BFS

2. <u>Social Network</u>

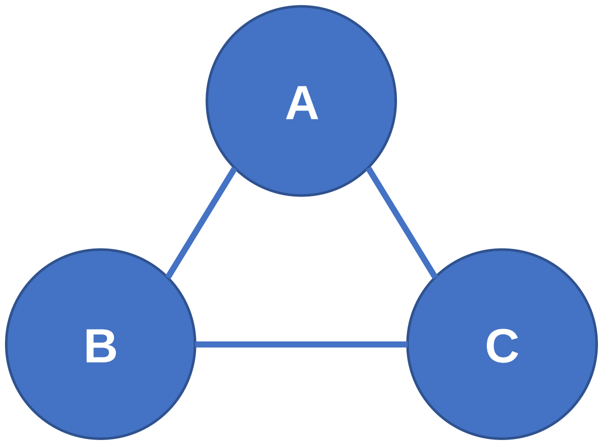
   **In social networks, we can find people within a given distance ‘k’ from a person BFS until ‘k’ levels.**



# Applications of BFS

3. <u>Detecting cycle in undirected graph</u>

   For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph

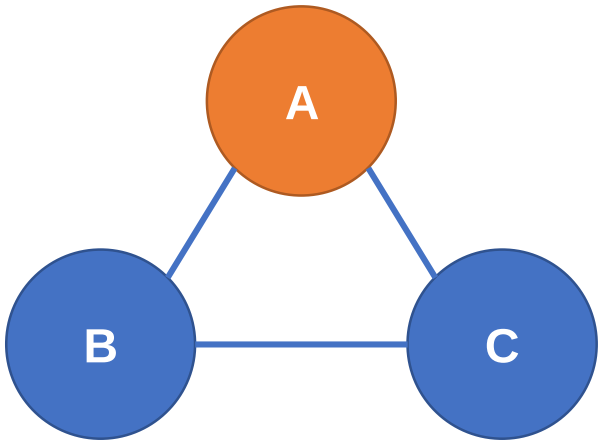



BFS + Visited

# Applications of BFS

3. <u>Detecting cycle in undirected graph</u>

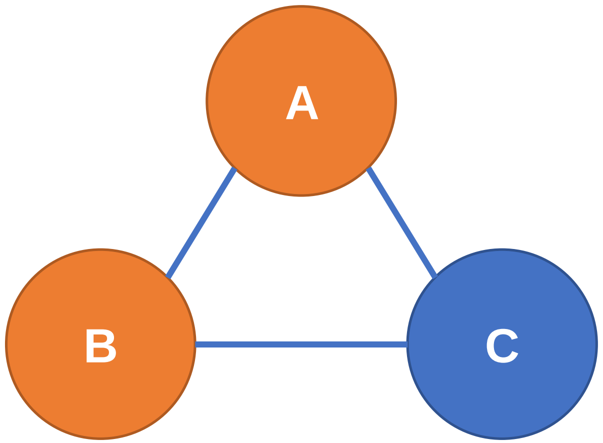
   **For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph**



# Applications of BFS

3. <u>**Detecting cycle in undirected graph**</u>

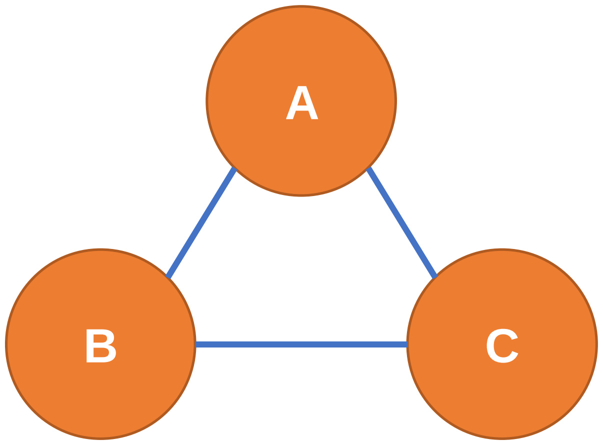
   **For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph**



# Applications of BFS

3. <u>Detecting cycle in undirected graph</u>

   **For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph**



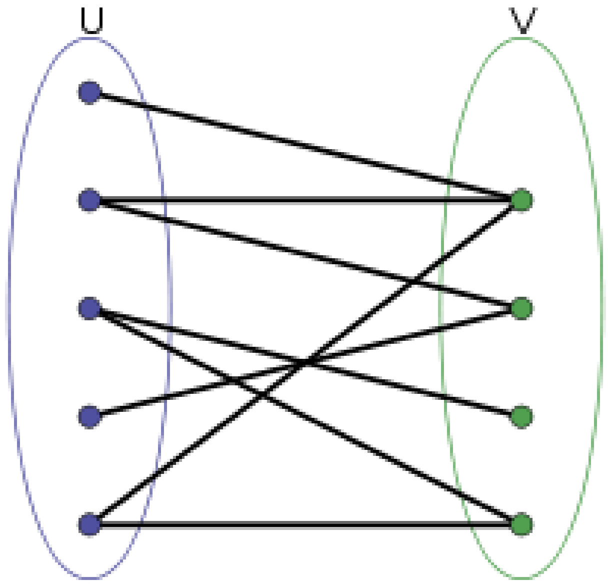
# Applications of BFS

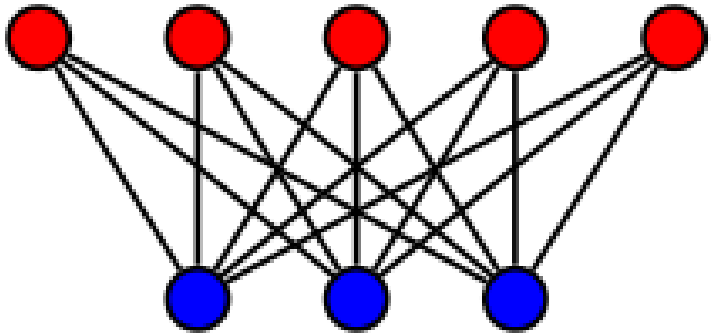
4. <u>Check if graph is bipartite or not</u>: Bipartite Graph is a graph whose vertices can be divided into two disjoint and independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.

# Applications of BFS

4. <u>Check if graph is bipartite or not</u>

**Bipartite Graph**



# Applications of BFS

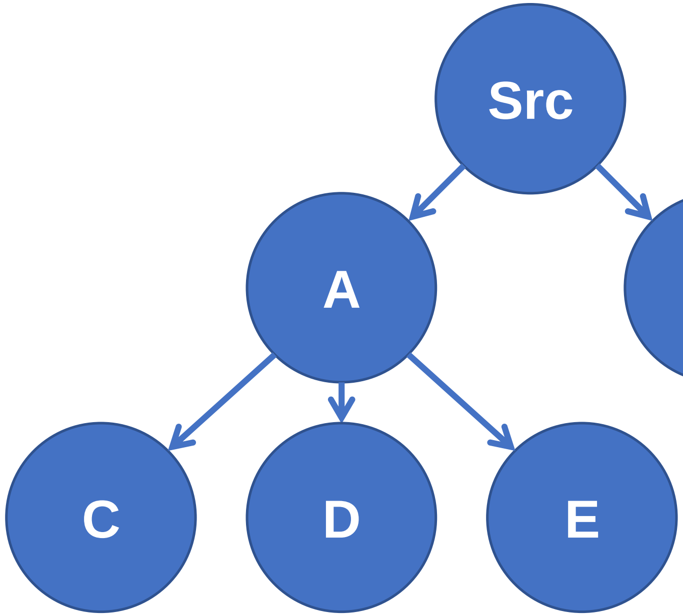
4. <u>Check if graph is bipartite or not</u>

Use the vertex coloring algorithm:

1) Start with a vertex and give it a color (RED).
2) Run BFS from this vertex. For each new vertex, color it opposite its parents. (p:RED → v:BLUE, p:BLUE → v:RED)
3) Check for edges that it doesn't link two vertices of the same color.
4) Repeat steps 2 and 3 until all the vertices are colored RED or BLUE.

# Applications of BFS

5. <u>Broadcasting in a Network</u>: Transferring data to all recipients simultaneously. BFS ensures that each node maintain shortest route to the source. Thus, reduces transmission delay and saves battery power.



# Applications of BFS

6. <u>Path Finding</u>: find a path between two given vertices u and v

    1) Create a queue which will store path(s).
    2) Now run a loop till queue is not empty.
        a) Get the frontmost path from queue.
        b) Check if the last vertex of this path is destination. If true then return it.
        c) Run a loop for all the vertices connected to the current vertex.
        d) If the vertex is not visited in current path,
            ① Create a new path from earlier path and append this vertex.
            ② Insert this new path to queue.

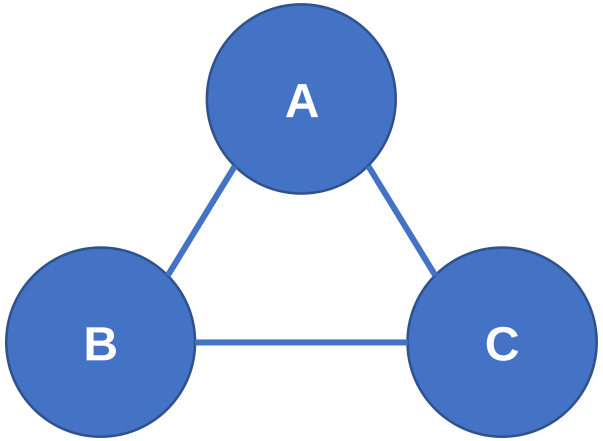
# Applications of DFS

1. Detecting cycle in a graph
2. Path Finding
3. To test if a graph is bipartite
4. Topological Sort
5. Strongly Connected Components

# Applications of DFS

1. <u>Detecting cycle in a graph</u>

   For every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not parent of v, then there is a cycle in graph

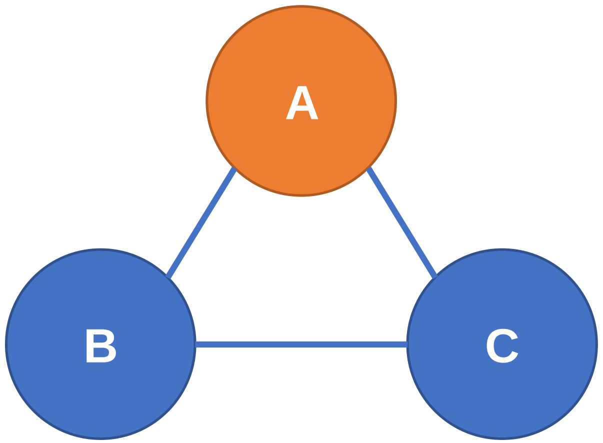



DFS + Visited

# Applications of DFS

1. **Detecting cycle in a graph**

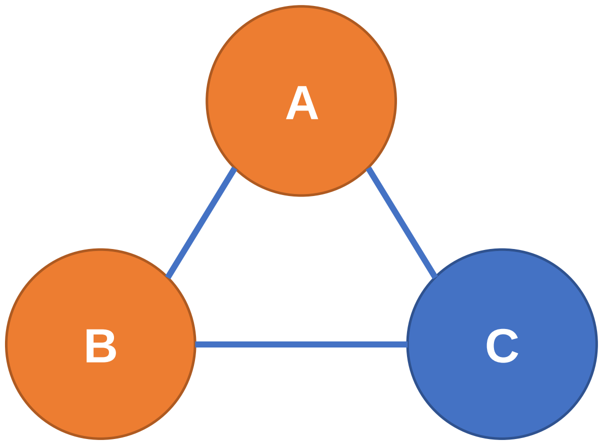
   **For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph**



# Applications of DFS

1. <u>Detecting cycle in a graph</u>

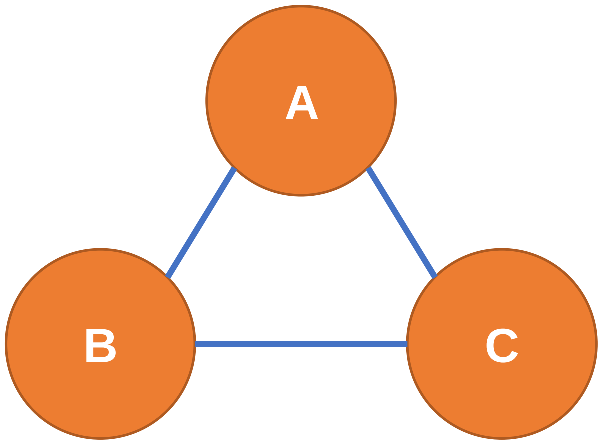
   For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph



# Applications of DFS

1. <u>Detecting cycle in a graph</u>

   For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph




Only DFS for directed graph

# Applications of DFS

2. Path Finding: find a path between two given vertices u and v

   Algorithm:

   1) Call DFS(G, u) with u as the start vertex.
   2) Use a stack S to keep track of the path between the start vertex and the current vertex.
   3) As soon as destination vertex v is encountered, return the path as the contents of the stack.

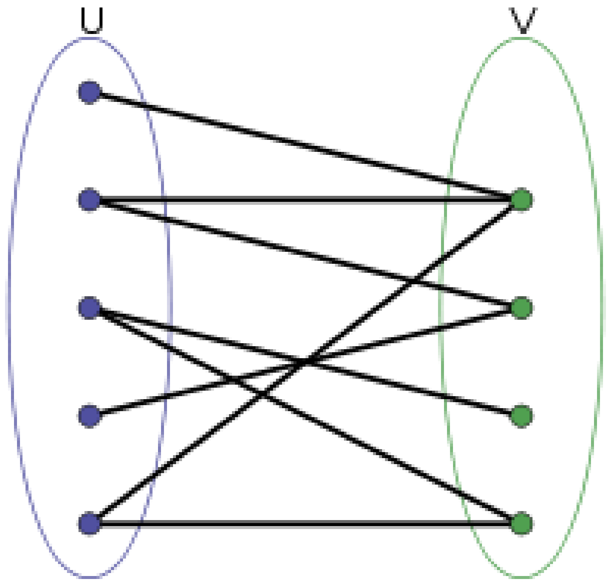
# Applications of DFS

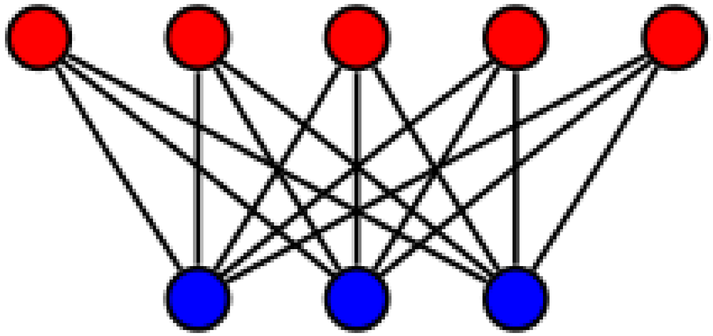
3. **<u>Check if graph is bipartite or not</u>: Bipartite Graph is a graph whose vertices can be divided into two disjoint and independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.**

# Applications of DFS

3. <u>Check if graph is bipartite or not</u>

**Bipartite Graph**



# Applications of DFS

3. <u>**Check if graph is bipartite or not**</u>

**Use the vertex coloring algorithm:**

1) **Start with a vertex and give it a color (RED).**
2) **Run DFS from this vertex. For each new vertex, color it opposite its parents. (p:RED → v:BLUE, p:BLUE → v:RED)**
3) **Check for edges that it doesn't link two vertices of the same color.**
4) **Repeat steps 2 and 3 until all the vertices are colored RED or BLUE.**

# Applications of DFS

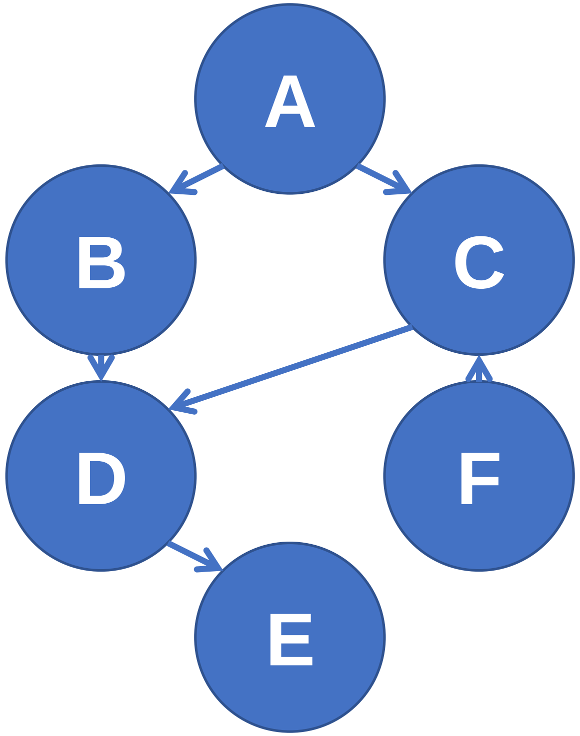
4. <u>Topological Sort</u>: for a DAG (Directed Acyclic Graph) G=(V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in ordering.

Topological Sort:

F A B C D E

F A C B D E

A B F C D E



# Applications of DFS

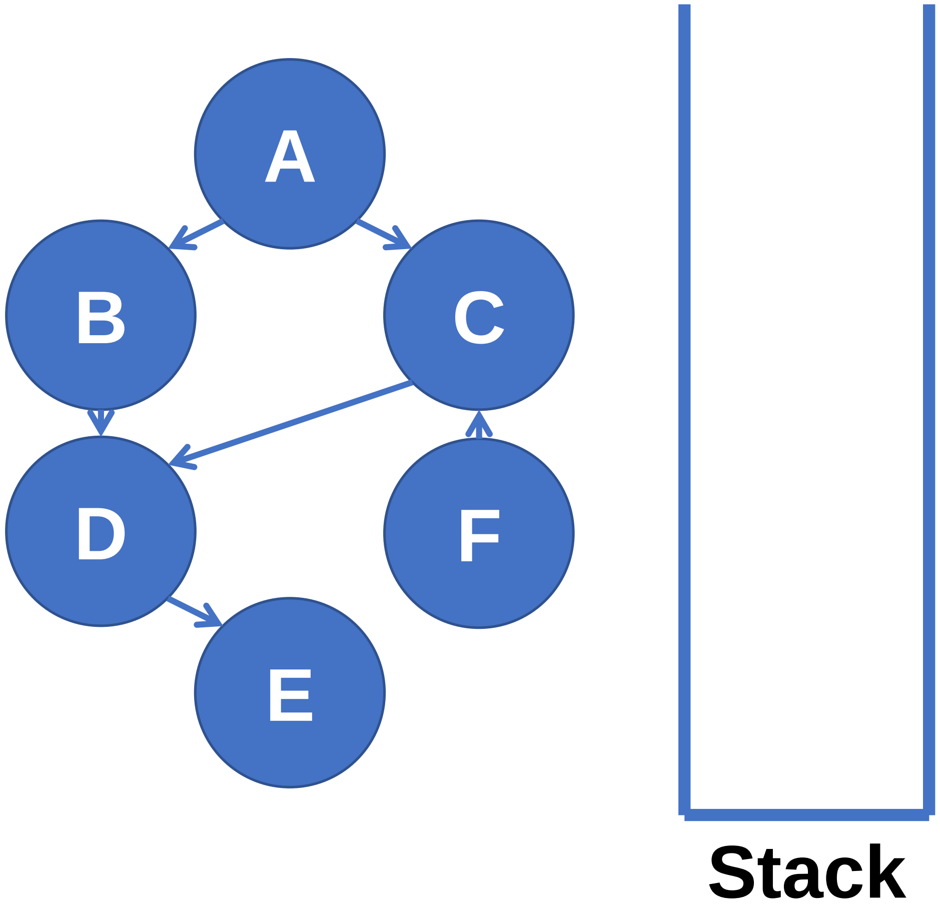
4. <u>Topological Sort</u>

   **Applications:**
   - **Build Systems**
   - **Advanced-Packaging Tool**
   - **Task Scheduling**
   - **Pre-requisite problems**

# Applications of DFS

**Topological Sort:**

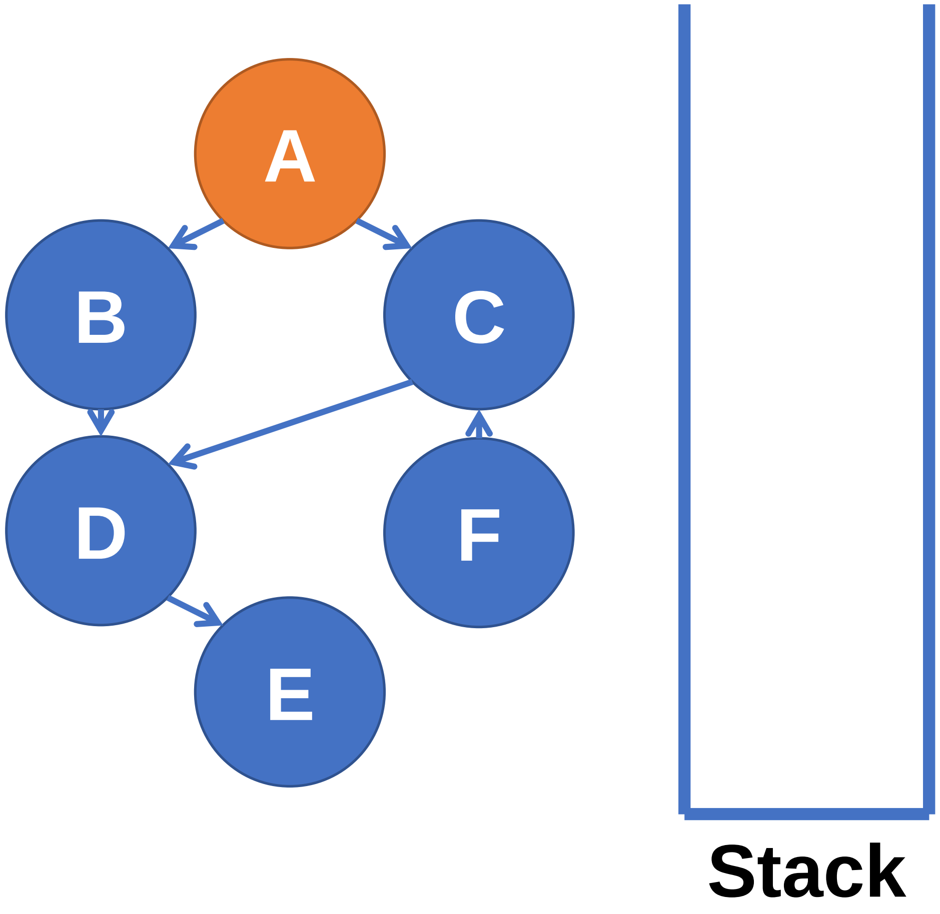
**Use Depth First Search using a temporary stack**



# Applications of DFS

**Topological Sort:**

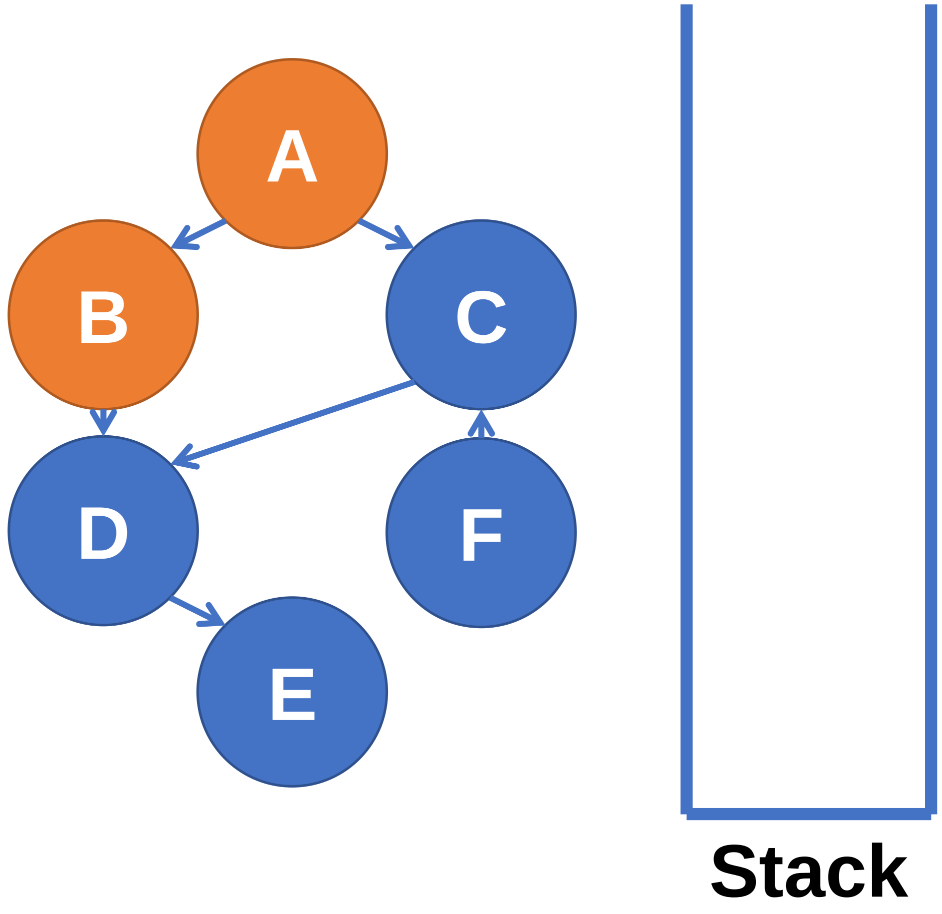
**Use Depth First Search using a temporary stack**



# Applications of DFS

**Topological Sort:**

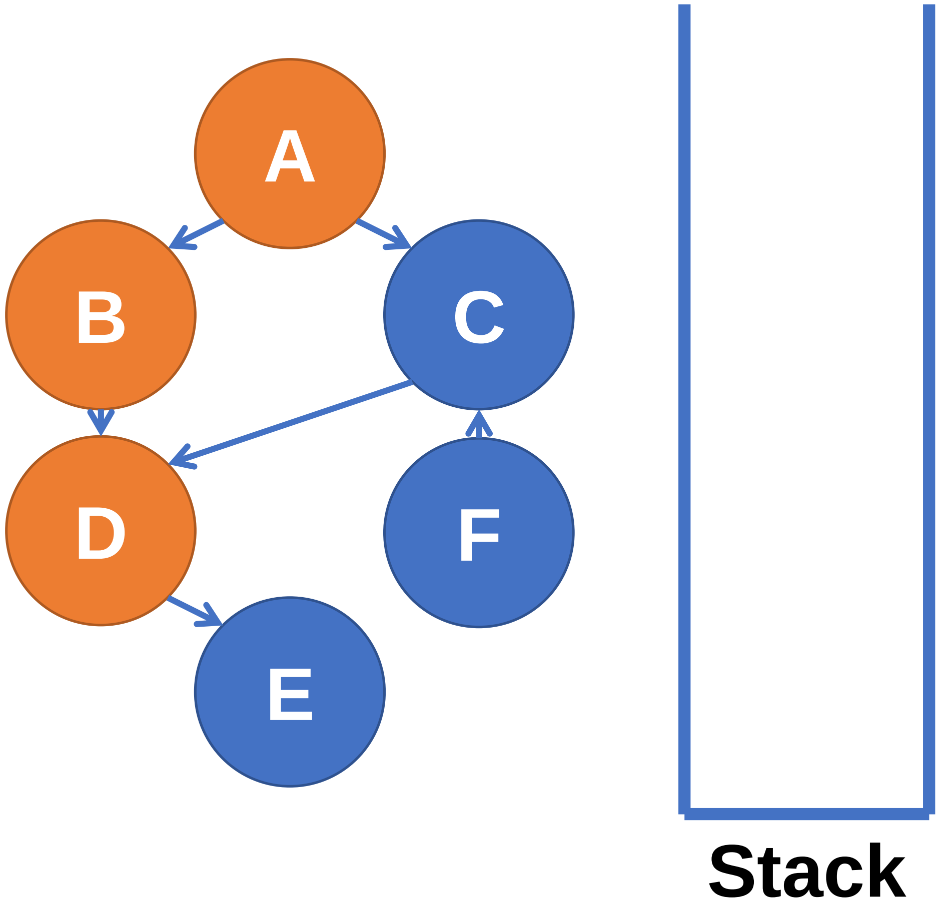
**Use Depth First Search using a temporary stack**



# Applications of DFS

**Topological Sort:**

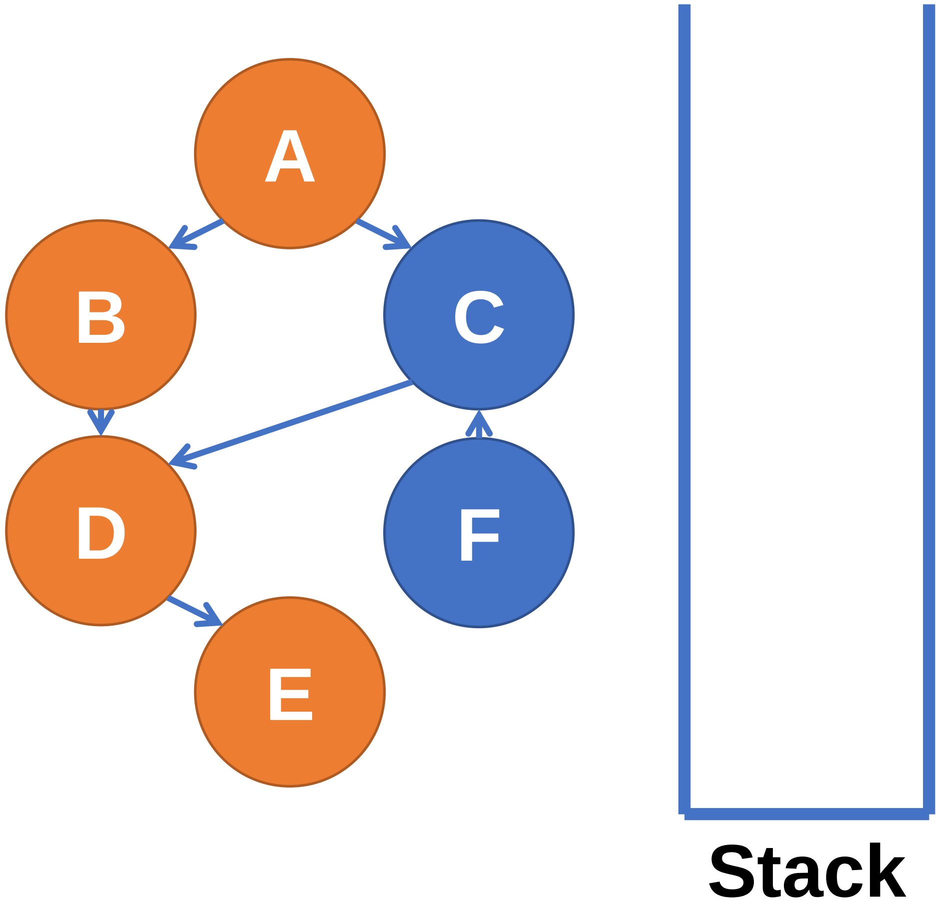
**Use Depth First Search using a temporary stack**



# Applications of DFS

**Topological Sort:**

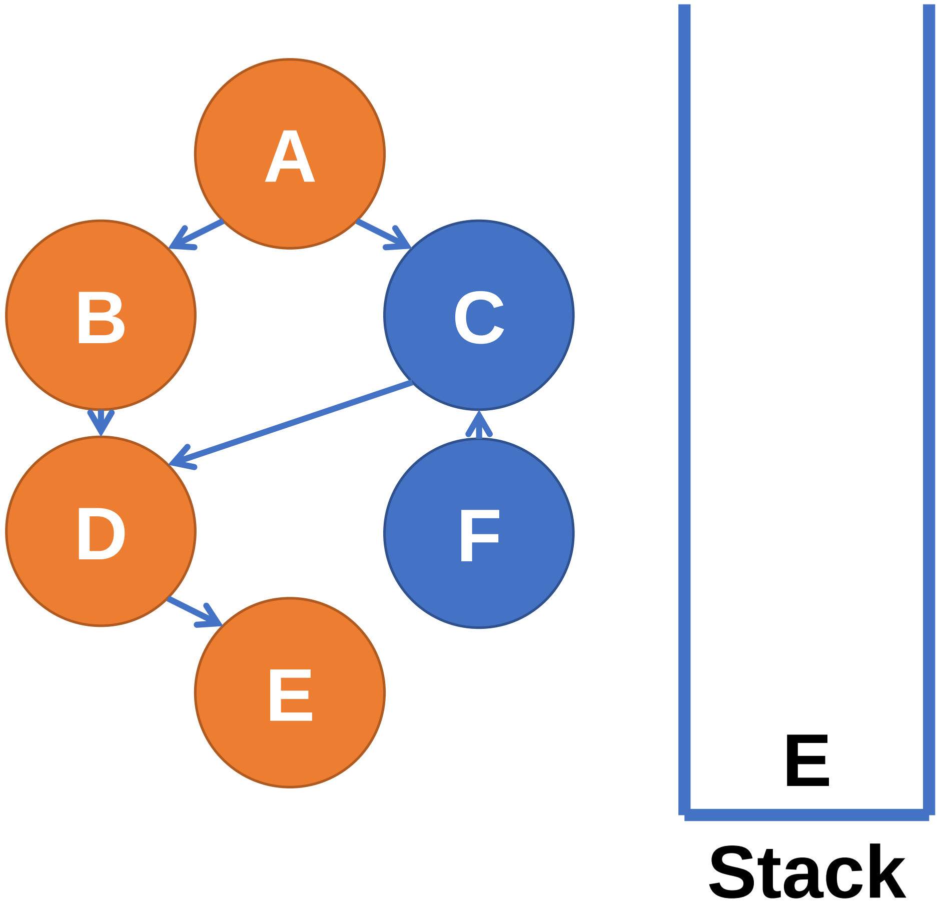
**Use Depth First Search using a temporary stack**



# Applications of DFS

**Topological Sort:**

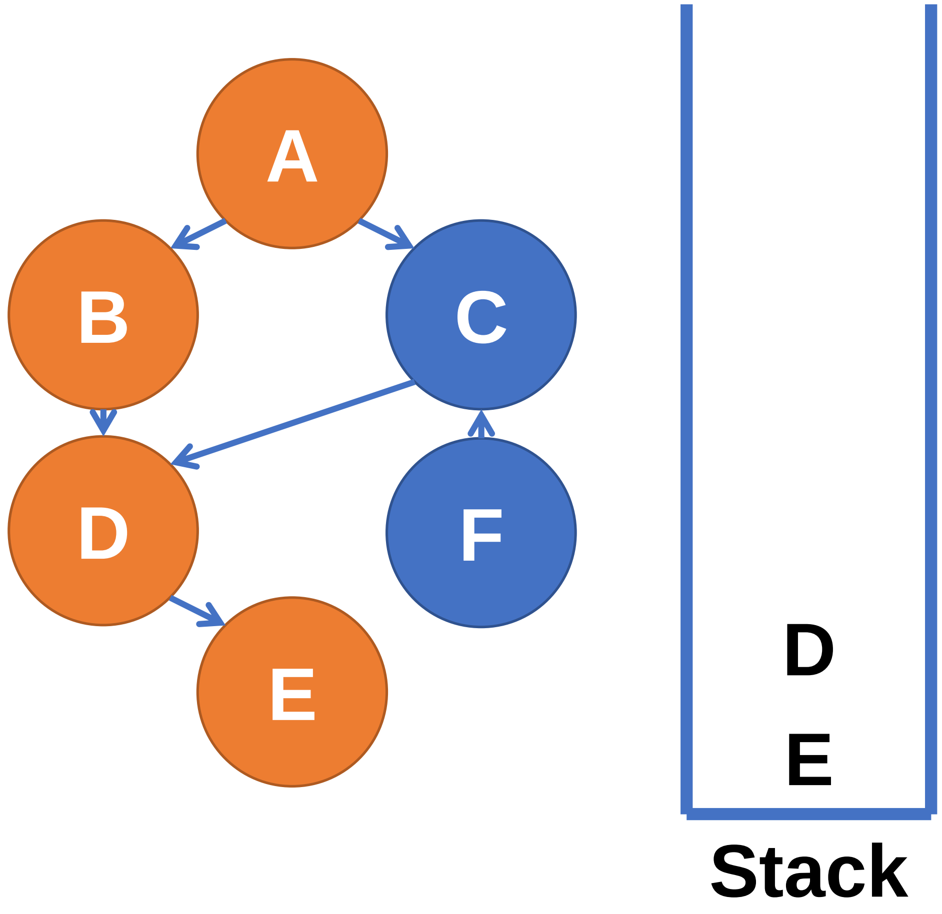
**Use Depth First Search using a temporary stack**



# Applications of DFS

**Topological Sort:**

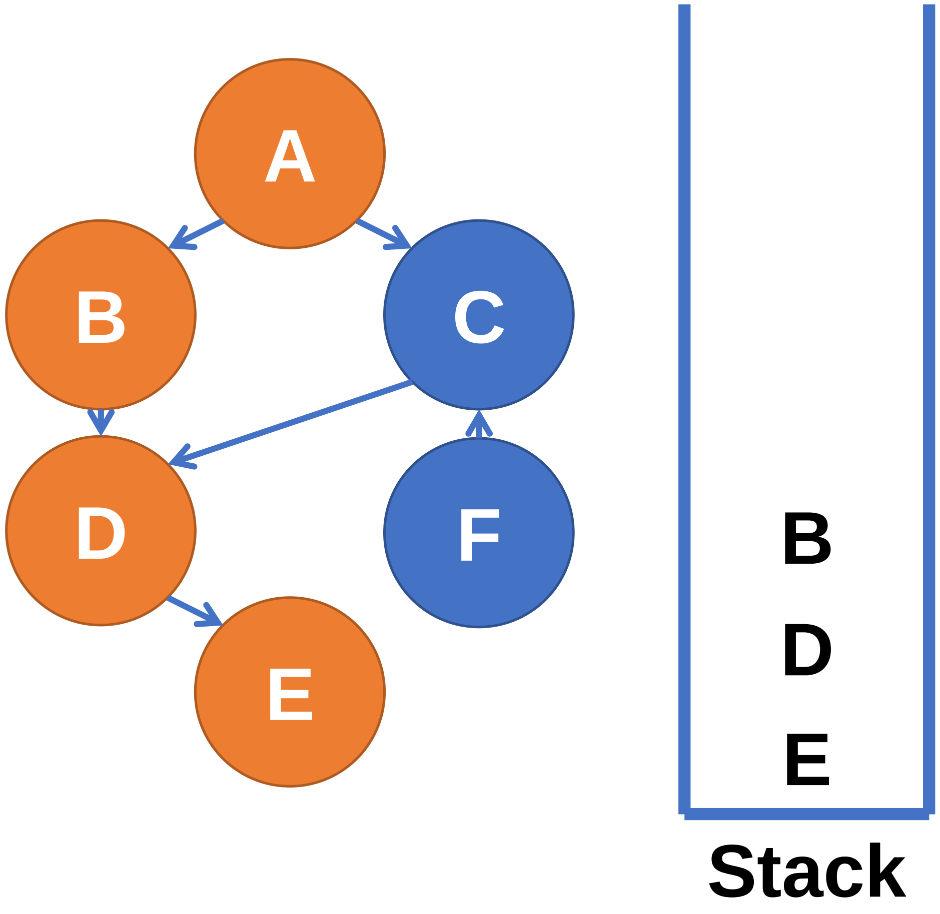
**Use Depth First Search using a temporary stack**



# Applications of DFS

Topological Sort:

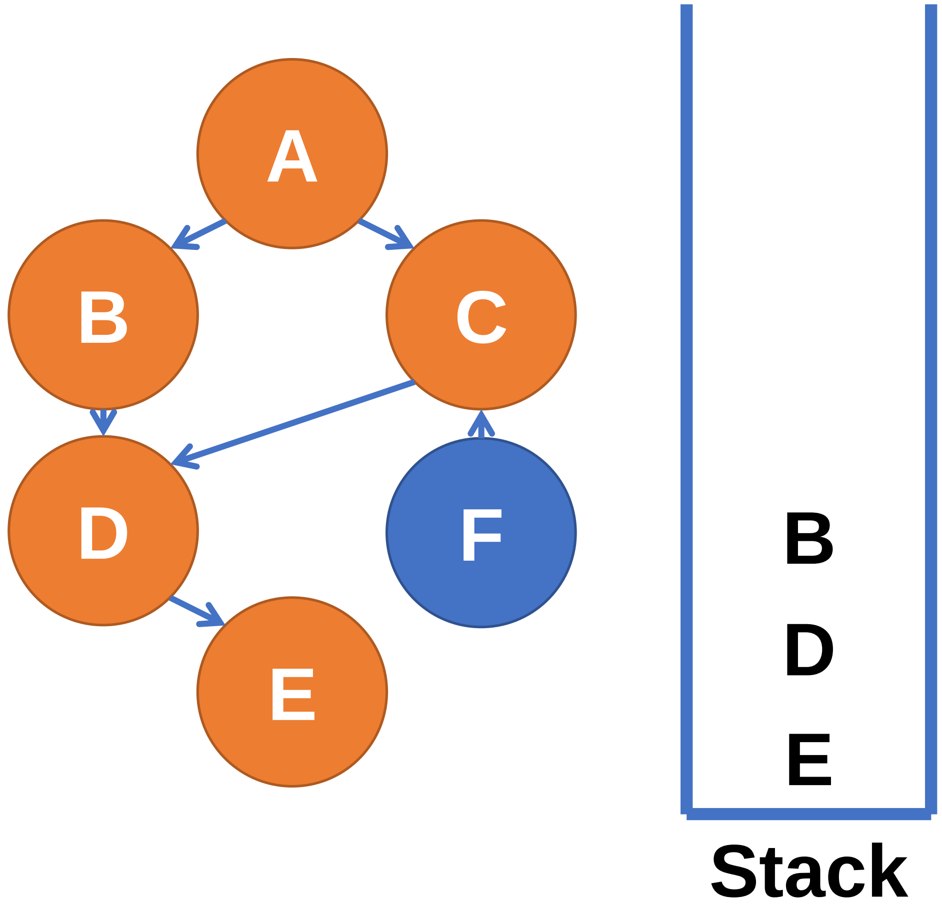
Use Depth First Search using a temporary stack



# Applications of DFS

**Topological Sort:**

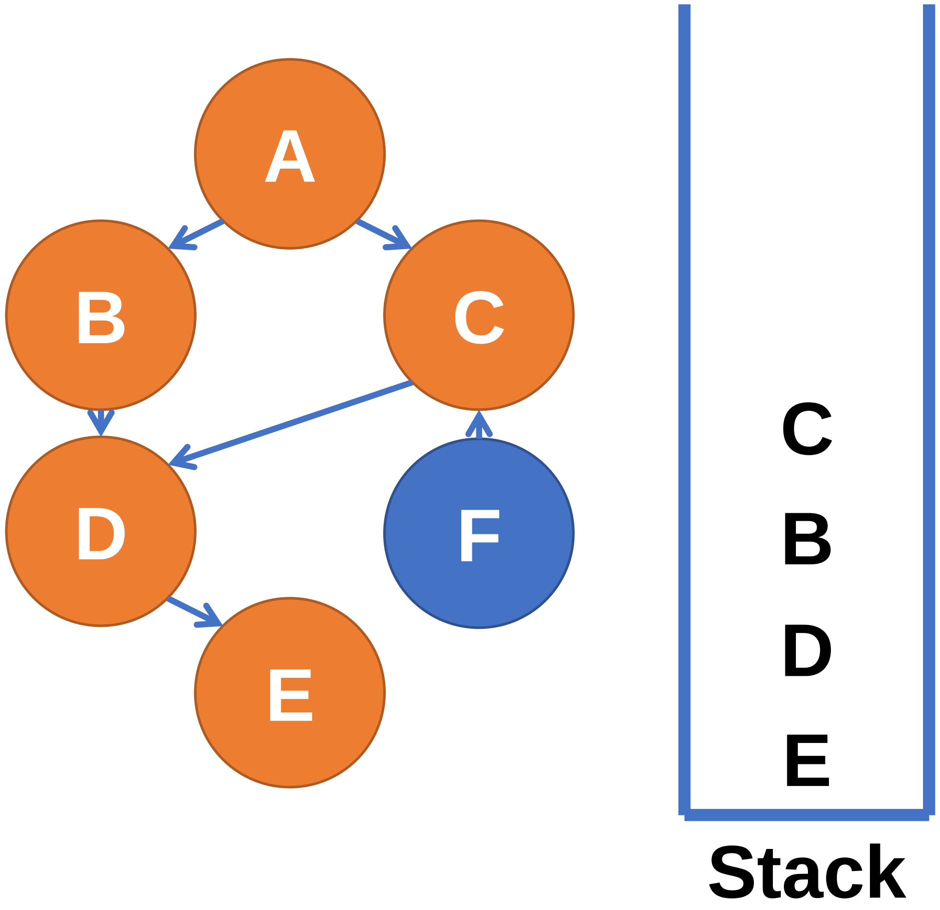
**Use Depth First Search using a temporary stack**



# Applications of DFS

**Topological Sort:**

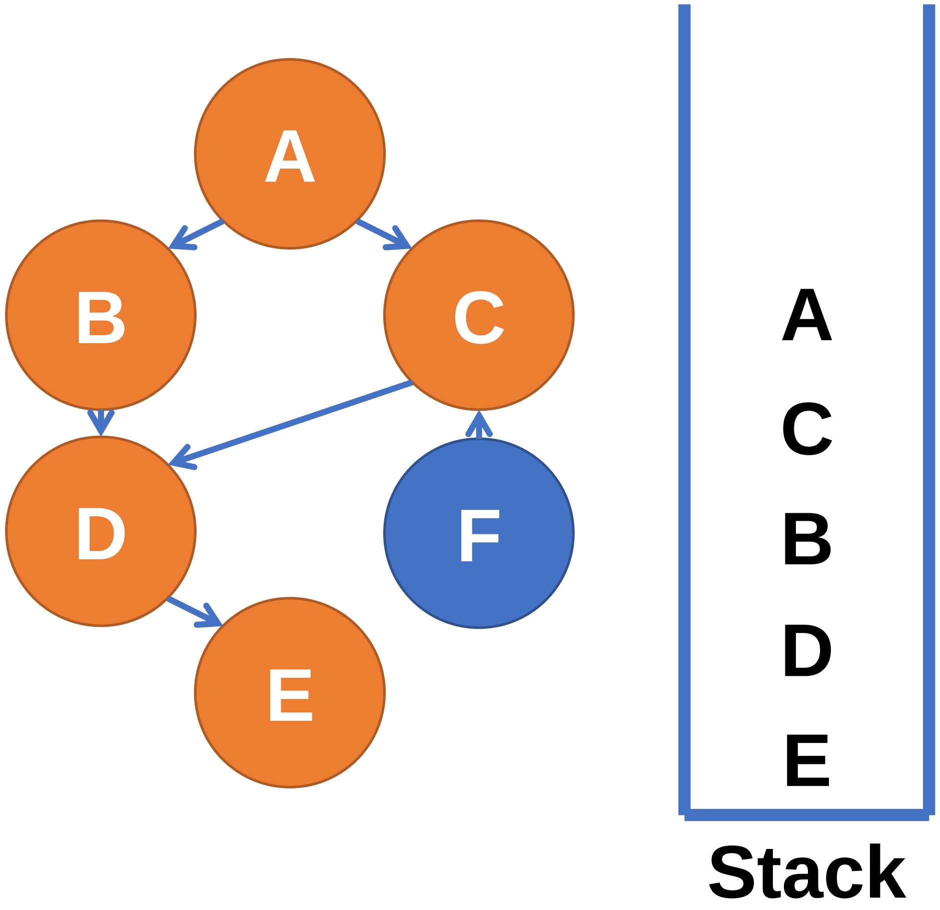
**Use Depth First Search using a temporary stack**



# Applications of DFS

**Topological Sort:**

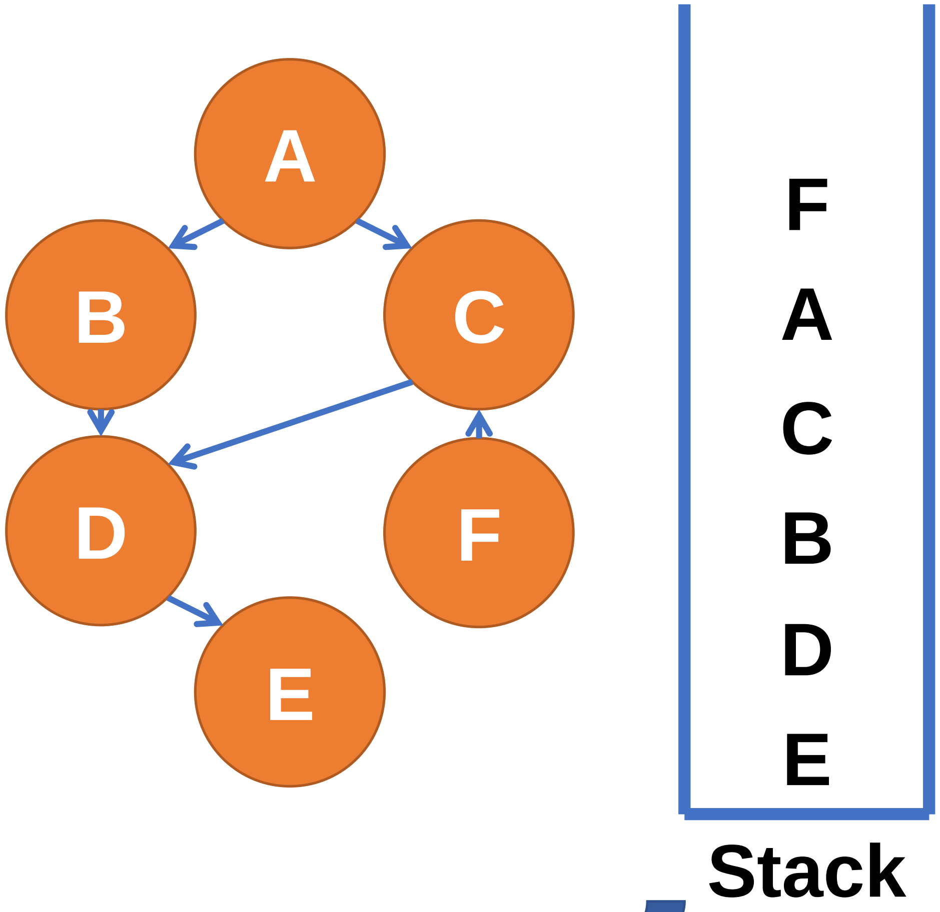
**Use Depth First Search using a temporary stack**



# Applications of DFS

**Topological Sort:**

**Use Depth First Search using a temporary stack**




**F A C B D E**

# Applications of DFS

## Topological Sort

```
void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int> &Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack which stores result
    Stack.push(v);
}
```

```
void Graph::topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
      if (visited[i] == false)
        topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false)
    {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}
```

# Applications of DFS

**Topological Sort**

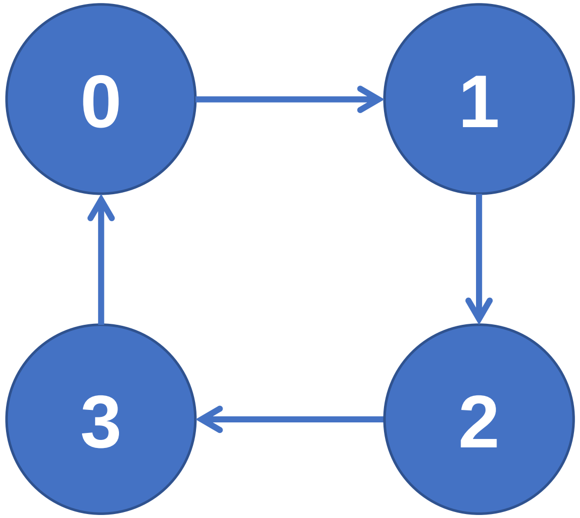
**<u>Time Complexity</u>: O(V+E)**

**V: Vertices**
**E: Edges**

# Applications of DFS

5. <u>Strongly Connected Components (SCC)</u>

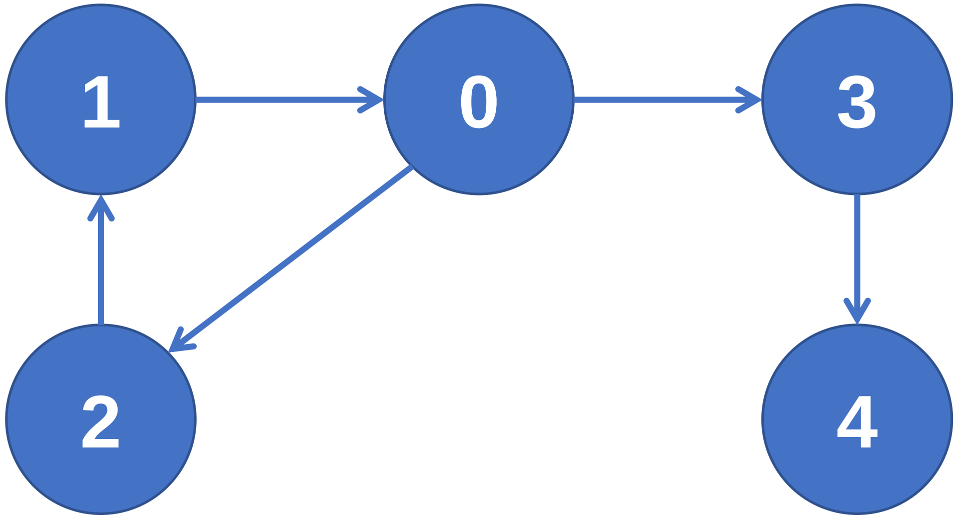
   A directed graph is strongly connected if there is a path between all pairs of vertices.



# Applications of DFS

5. <u>Strongly Connected Components (SCC)</u>

A strongly connected components (SCC) of a directed graph is a maximal strongly connected subgraph.




strongly connected components
1. 1-0-2
2. 3
3. 4

# Applications of DFS

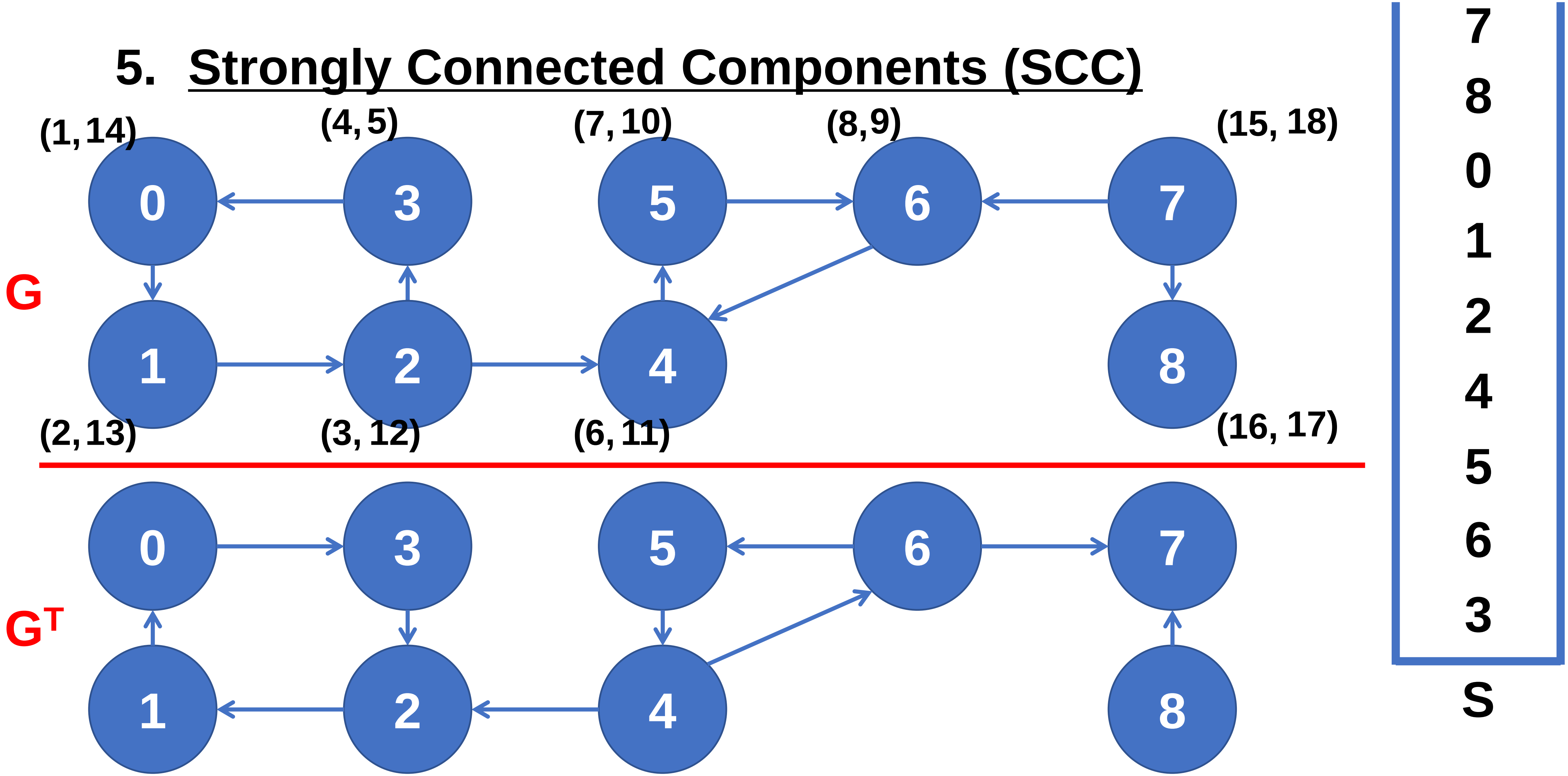
5. **Strongly Connected Components (SCC)**

**Kosaraju's algorithm:**

1) Create an empty stack S.
2) Do DFS of a graph. In DFS, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack.
3) Reverse directions of all arcs to obtain the transpose graph.
4) One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS call on v. The DFS starting from v prints strongly connected component of v.

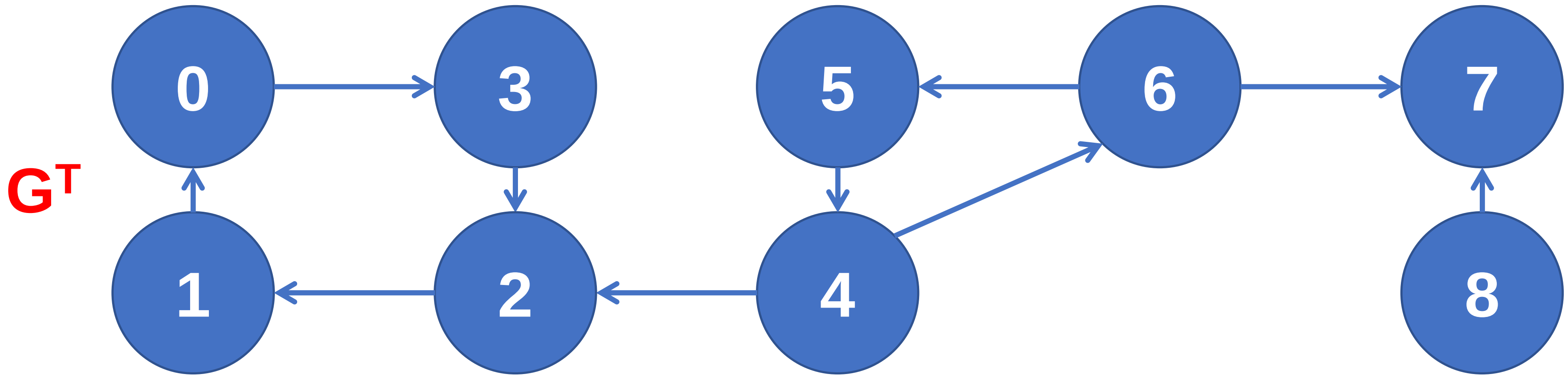
# Applications of DFS

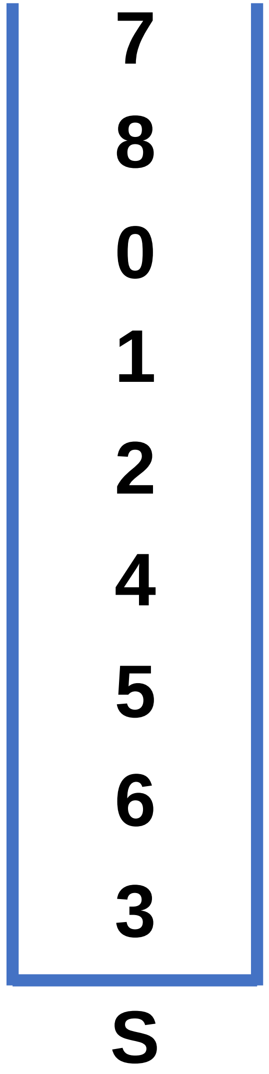
## 5. Strongly Connected Components (SCC)



# Applications of DFS

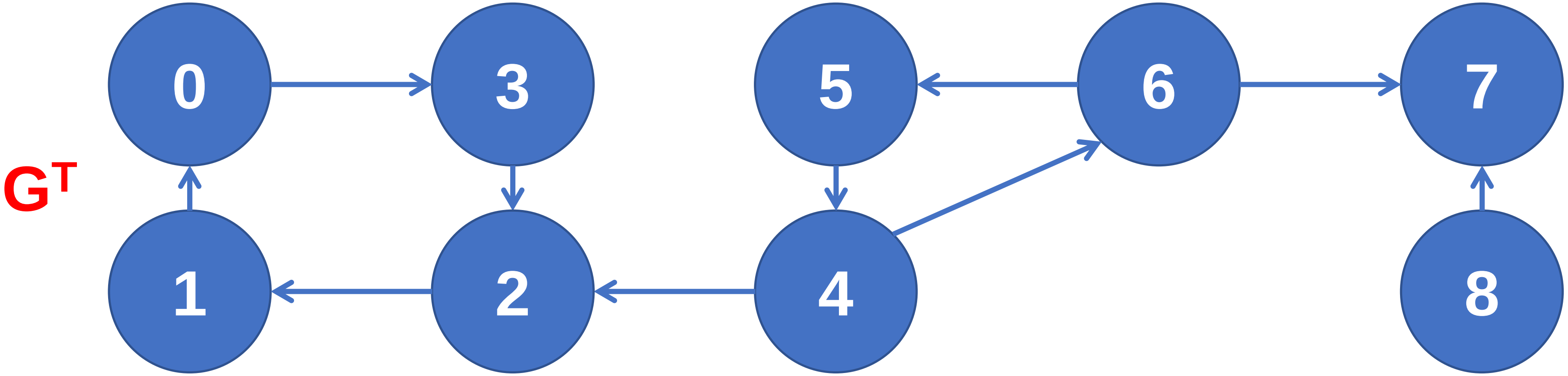
5. <u>Strongly Connected Components (SCC)</u>



# Applications of DFS

5. <u>Strongly Connected Components (SCC)</u>

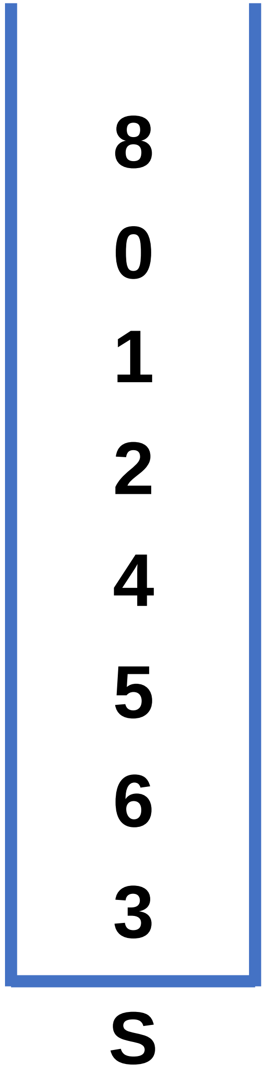



Pop 7

Visited: {7}

strongly connected components

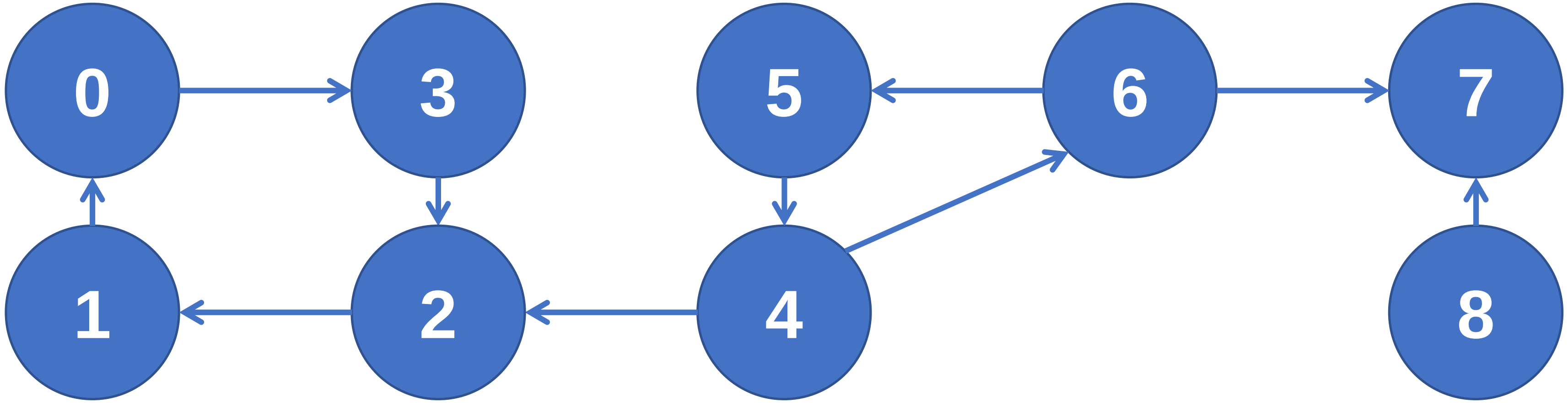
1. 7

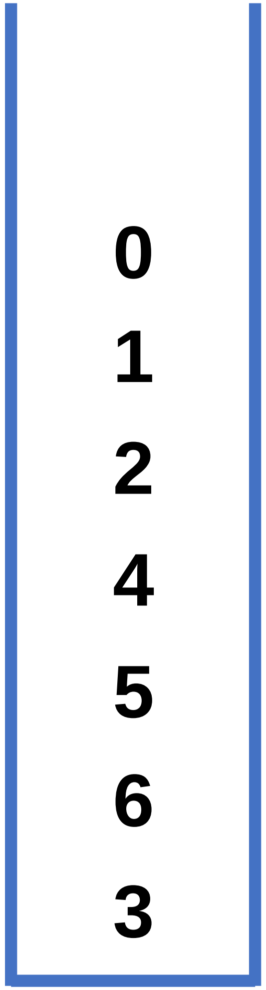


# Applications of DFS

## 5. Strongly Connected Components (SCC)

$G^T$




S

**Pop 8**

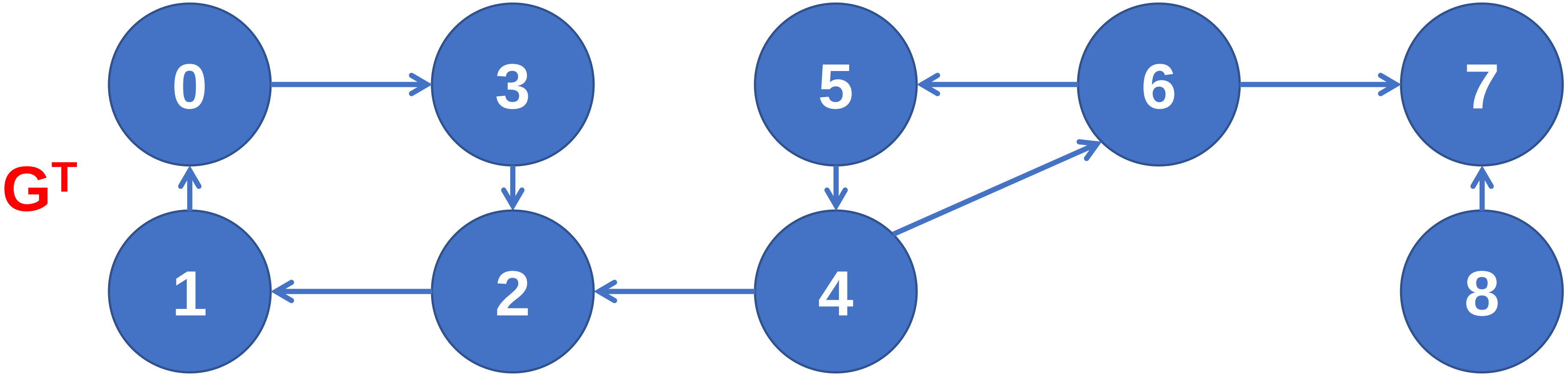
**Visited: {7,8}**

**strongly connected components**

1. 7
2. 8

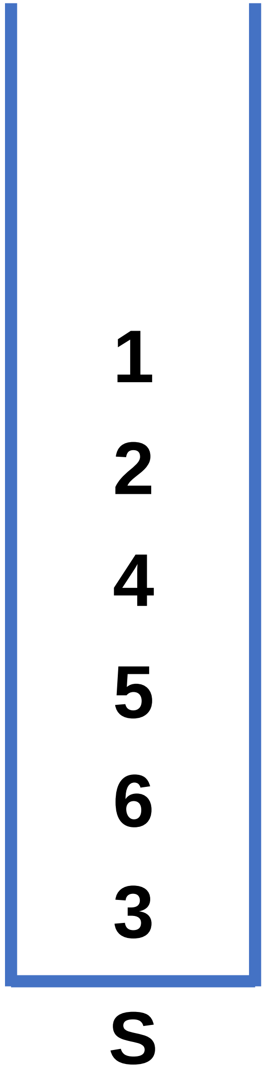
# Applications of DFS

## 5. Strongly Connected Components (SCC)




Pop 0

Visited: {7,8,0,3,2,1}

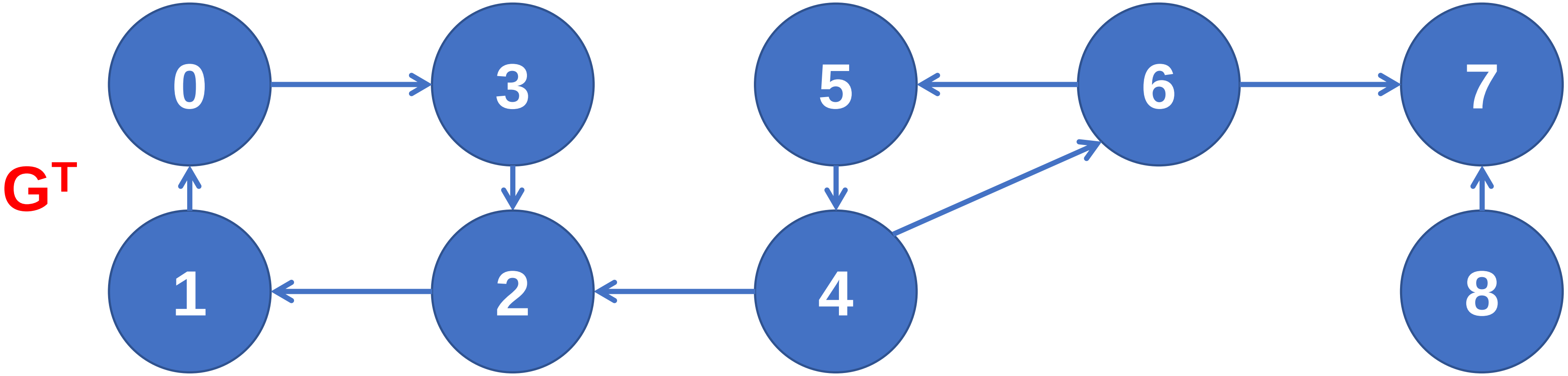



strongly connected components

1. 7
2. 8
3. 0-3-2-1

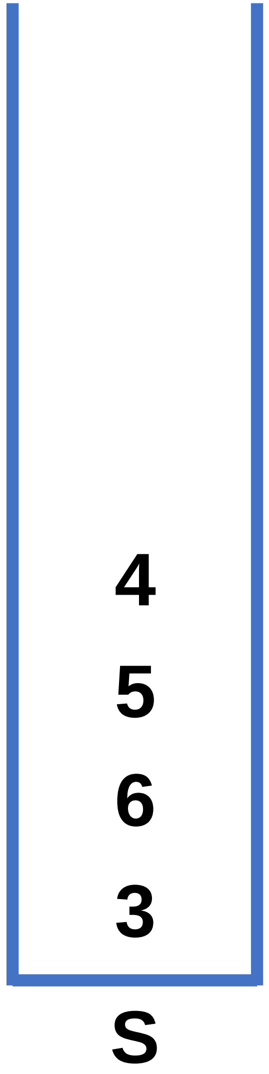
# Applications of DFS

## 5. Strongly Connected Components (SCC)




Pop 1 and 2

Visited: {7,8,0,3,2,1}

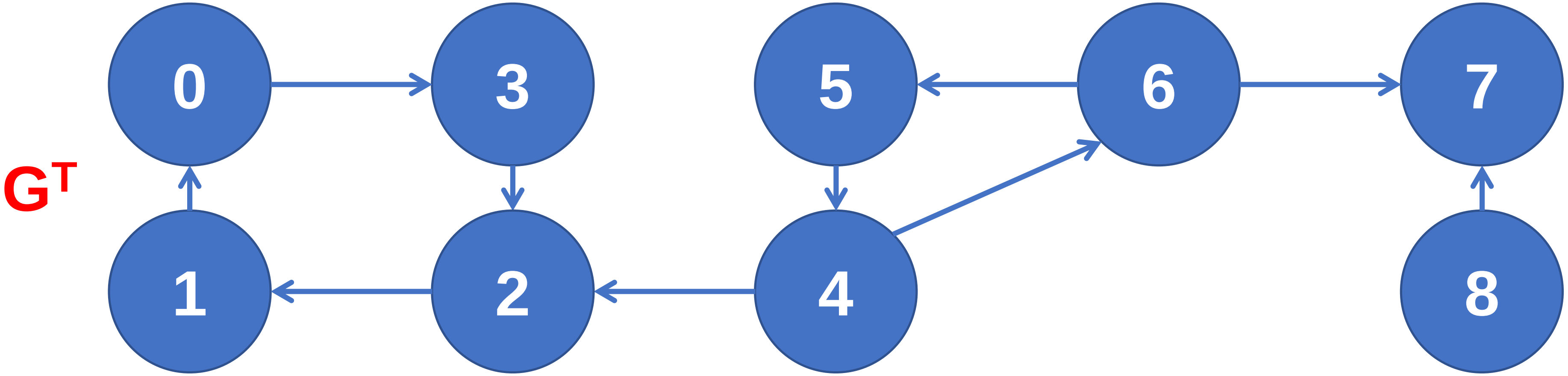



strongly connected components

1. 7
2. 8
3. 0-3-2-1

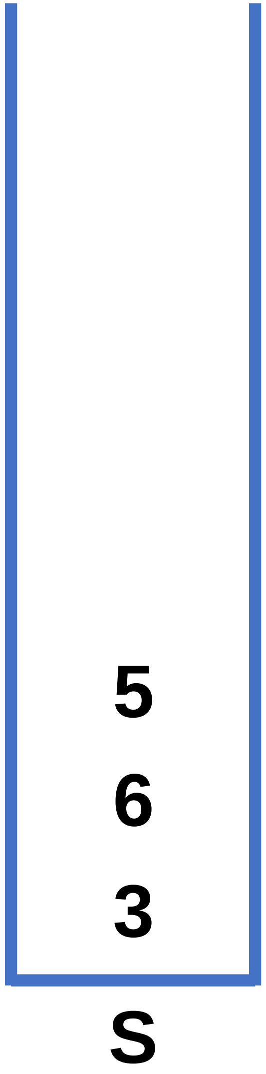
# Applications of DFS

## 5. Strongly Connected Components (SCC)




**Pop 4**

**Visited: {7,8,0,3,2,1,4,6,5}**

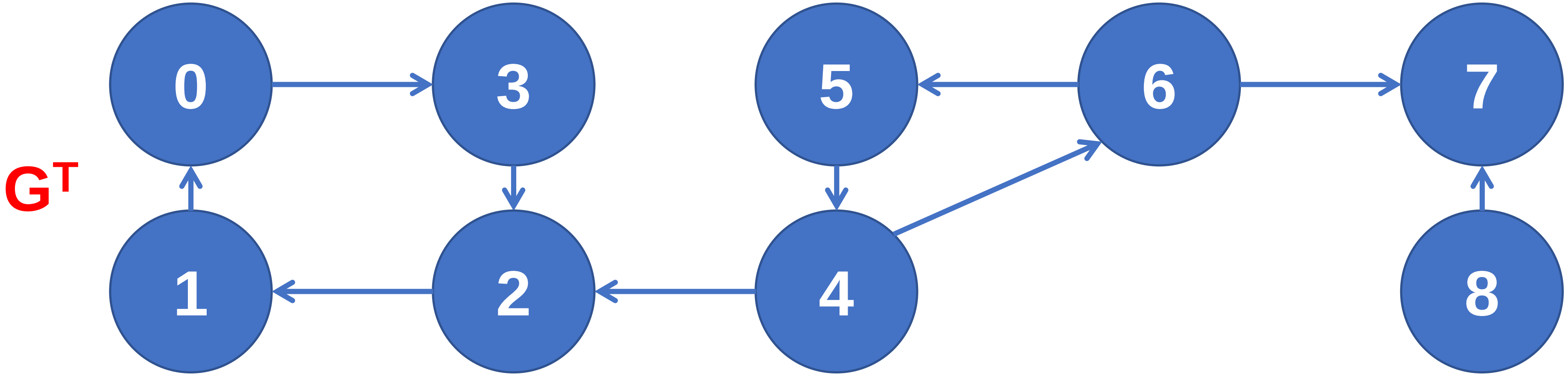


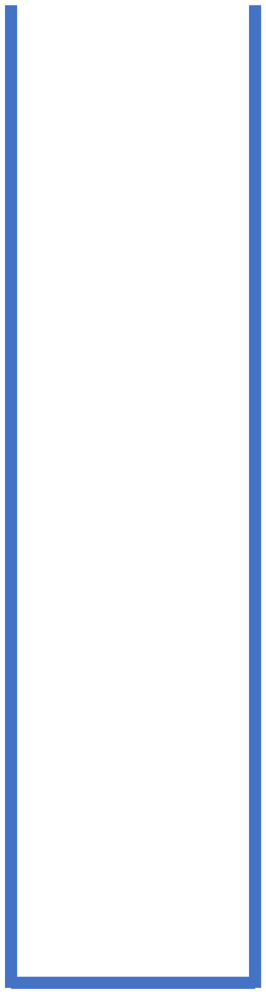

strongly connected components

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

# Applications of DFS

## 5. Strongly Connected Components (SCC)



S

**Pop 5, 6 and 3**

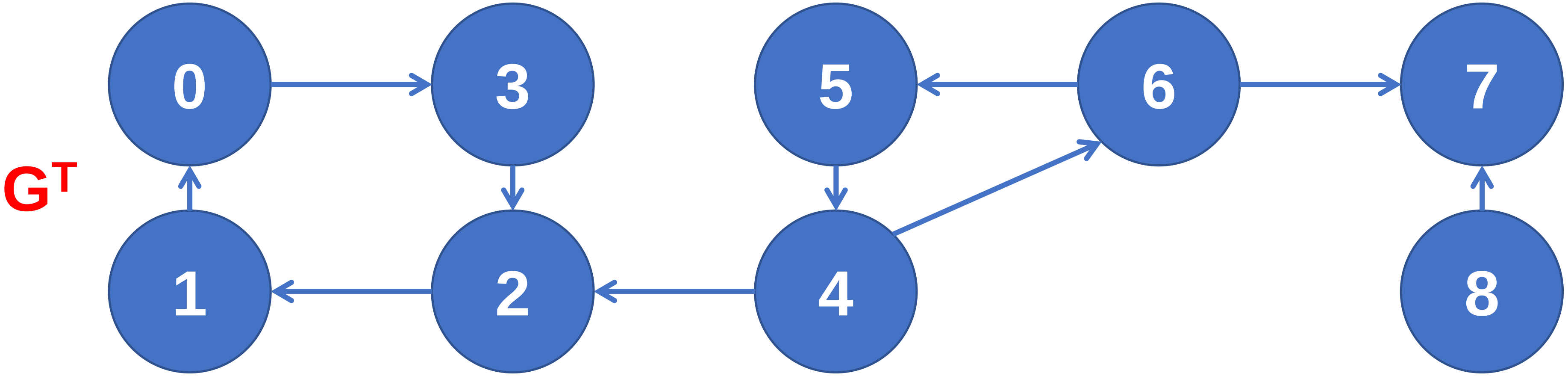
**Visited: {7,8,0,3,2,1,4,6,5}**

**strongly connected components**

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

# Applications of DFS

## 5. Strongly Connected Components (SCC)



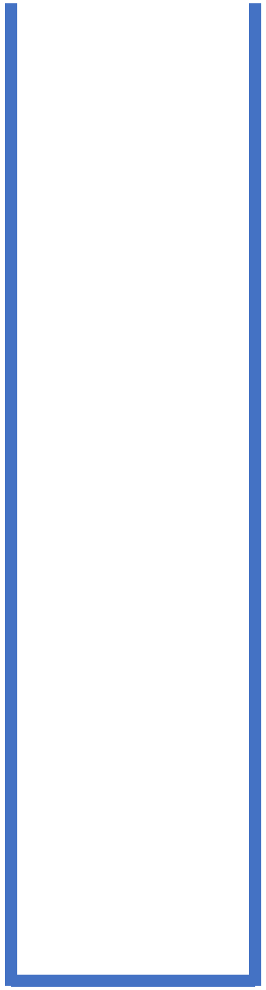

Stack is empty → Terminate

strongly connected components

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


Graph of SCCs

SCCs in reverse graph



S

# Applications of DFS

## Strongly Connected Components

```
void Graph::fillOrder(int v, bool visited[], stack<int> &Stack)
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for(i = adj[v].begin(); i != adj[v].end(); ++i)
        if(!visited[*i])
            fillOrder(*i, visited, Stack);

    // All vertices reachable from v are processed by now, push v
    Stack.push(v);
}
```

```
void Graph::printSCCs()
{
    stack<int> Stack;

    // Mark all the vertices as not visited (For first DFS)
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Fill vertices in stack according to their finishing times
    for(int i = 0; i < V; i++)
        if(visited[i] == false)
            fillOrder(i, visited, Stack);

    // Create a reversed graph
    Graph gr = getTranspose();

    // Mark all the vertices as not visited (For second DFS)
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Now process all vertices in order defined by Stack
    while (Stack.empty() == false)
    {
        // Pop a vertex from stack
        int v = Stack.top();
        Stack.pop();

        // Print Strongly connected component of the popped vertex
        if (visited[v] == false)
        {
            gr.DFSUtil(v, visited);
            cout << endl;
        }
    }
}
```

# Reference


- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms*”, September 29, 2004

- https://www.geeksforgeeks.org