Synthesis Assignment 2
Erdun E
October 18, 2024
Dr. Alan Jamieson

CS 5800: Algorithm
Synthesis Assignment 2

4 Graphs

4.1 Basic Definition

4.11 Basic Definition

Graphs are important data structures used in computer science and mathematics to represent relationships between entities. Graphs consist of two main parts:

Vertices (V):

- Also known as a node, it is the basic unit of a graph. Vertices represent entities or points and are usually denoted by symbols such as v1, v2....., vn and other symbols. The set of all vertices is denoted by V.

Edges (E):

- Edge (E): This is a connection or relationship between a pair of vertices. An edge can be represented as a pair of vertices such as (u, v), where u and v are elements of V.

4.12 Mathematical Notation

Graphs can be represented in many ways, for example using an adjacency table or an adjacency matrix. An adjacency table representation stores a list of neighbors for each vertex and is therefore suitable for sparse graphs. On the other hand, an adjacency matrix is a two-dimensional array in which each cell indicates the presence or absence of an edge between two vertices, which is very effective for dense graphs.

A graph is typically represented as G = (V, E), where:

- V is a finite set of vertices, and |V| represents the number of vertices.
- E is a finite set of edges, |E| represents the number of edges.

4.13 Classes of Graphs

Graphs can be categorized into different classes based on their properties:

Directed vs Undirected graphs:

- Directed graphs: Edges have a direction, such as from one vertex to another. Directed graphs are often used to model one-way relationships, such as hyperlinks between web pages or task dependencies.
- Undirected graphs: Edges have no direction, meaning the connection between vertices u and v is bidirectional. Edges are represented as unordered pairs $\{u, v\}$. Undirected graphs often represent bidirectional relationships, such as friendships in social networks.

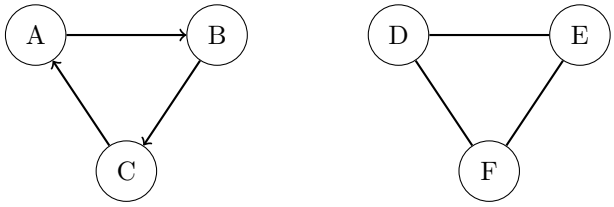

Figure 1: Left: Directed Graph Right: Undirected Graph

Weighted vs Unweighted Graphs:

- Weighted graph: Each edge has an associated weight or cost, often used to represent distances or other quantifiable relationships between vertices. For example, in a road network, weights may represent distances or travel times between different locations.
- Unweighted graph: Edges do not have any weights, which means that all connections are considered equivalent. This graph is used when the relationship between vertices is uniform, such as simple connections in a network.

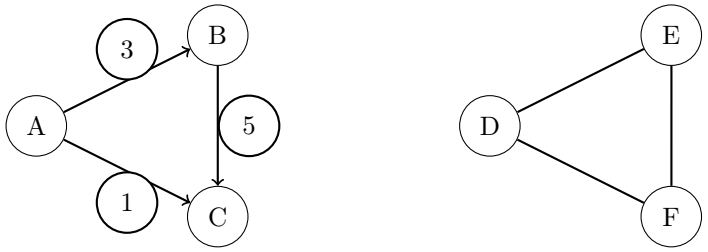

Figure 2: Left: Weighted Graph Right: Unweighted Graph

Simple vs Multi Graphs:

- Simple Graphs: Simple graphs are graphs that have no loops, no edges connecting vertices to themselves, and no multiple edges between the same pair of vertices. Simple graphs are typically used in scenarios that allow only a single, unique relationship between entities.
- Multi Graph: A graph that allows multiple edges between the same set of vertices. Multiple graphs are used in modeling scenarios where multiple relationships may exist between the same entities, such as multiple flights between two airports.

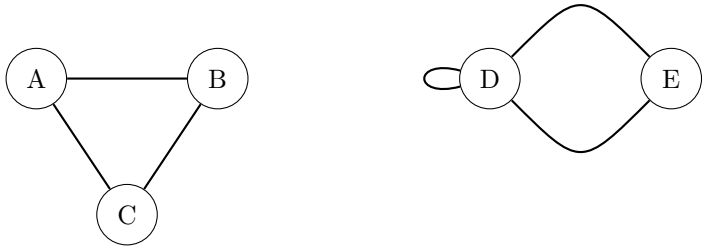

Figure 3: Left: Simple Graph Right: Multi Graph

Connected vs Disconnected Graphs:

- Connected Graph: In an undirected graph, a graph is said to be connected if there is a path between every pair of vertices. Connected graphs are often used to represent systems where all components are mutually accessible, such as a fully operational communication network.
- Disconnected Graph: A graph is said to be disconnected if one or more pairs of vertices do not have a path between them. Disconnected graphs represent systems where components or points of failure are isolated from each other.

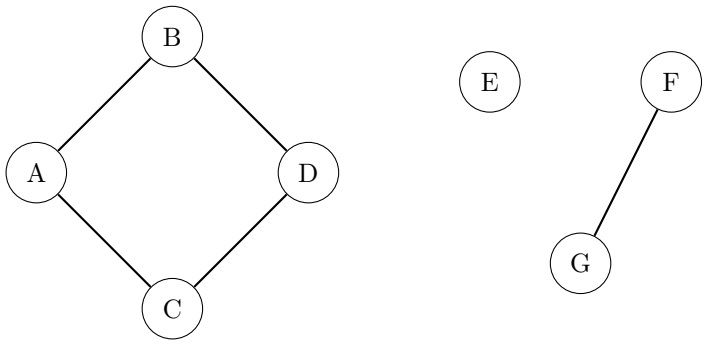

Figure 4: Left: Connected Graph Right: Disconnected Graph

Cyclic vs Acyclic Graphs

- Cyclic graphs: Graphs that contain at least one loop, that is a path with the same start and end vertices. Cyclic graphs are useful for modeling systems with feedback loops such as circuits.
- Acyclic graphs: Graphs that do not contain loops. Directed acyclic graphs are often used in scenarios such as task scheduling, where dependencies need to be respected and loops can lead to conflicts.

Figure 5: Left: Cyclic Graph Right: Acyclic Graph

4.2 Graph Representations

4.21 Explanation

Graphical representation is fundamental to understanding how to implement graphics in a computer system. Two common methods for representing graphs are adjacency lists and adjacency matrices. Both methods have advantages and disadvantages, and the choice usually depends on the type of graph and the operations to be performed.

- Adjacency List: In an adjacency list, each vertex has a list of all vertices connected by edges. For example, if vertex A is connected to vertices B and C, then A's list will contain B and C. This representation saves space, especially for sparse graphs. An adjacency list is good for sparse.

- Adjacency Matrix: An adjacency matrix is a two-dimensional array of size |V| x |V|, where |V| is the number of vertices. In an unweighted graph, each element of the matrix is either 0 for no edges or 1 for edges. In a weighted graph, the elements can store the weights of the edges. The adjacency matrix is straightforward and allows for constant-time edge lookups, but it can be spatially inefficient for large sparse graphs because every possible edge is represented, even if it does not exist. Adjacency Matrix is good for dense graphs.

In summary, for sparse graphs, adjacency lists are more space-efficient, while adjacency matrices are faster to check for the existence of an edge between two vertices. The choice of which representation to use should be consistent with the type of graph and the operations most often performed.

4.22 Exercise

Consider the following weighted graph with 4 vertices (A, B, C, D):

- A is connected to B with a weight of 3, and to C with a weight of 1.
- B is connected to C with a weight of 7, and to D with a weight of 5.
- C is connected to D with a weight of 2.

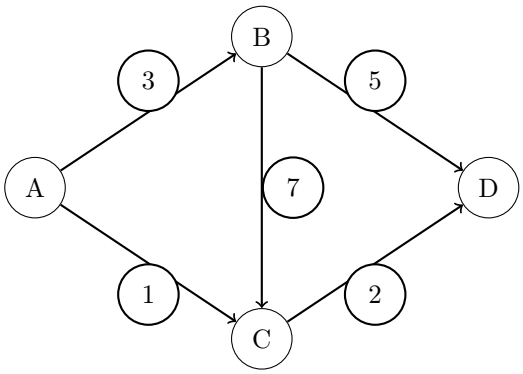

Figure 6: Weighted Graph with vertices A, B, C, D

Represent this graph using both an adjacency list and an adjacency matrix.

4.23 Solution

Algorithm 1 Pseudocode of Adjacency List

Input : A graph represented as a list of edges: $(u, v, weight)$
Output: Adjacency List

Initialize an empty adjacency *list*
foreach *edge* $(u, v, weight)$ *in the input graph* **do**
    Add $(v, weight)$ to $list[u]$
    if *graph is undirected* **then**
        Add $(u, weight)$ to $list[v]$
return *list*

Explanation of the Pseudocode of Adjacency List

- Step 1: Initialize an empty adjacency list.
- Step 2: Iterate over each edge in the graph.

- Step 3: For each edge, add the node and weight to the adjacency list of the source node.
- Step 4: Add the reverse edge to the list if the graph is undirected.
- Step 5: Finally, return the completed adjacency list.

Adjacency List Representation:

- A: [(B, 3), (C, 1)], given A is connected to B with weight 3 and to C with weight 1, so A's list contains (B, 3) and (C, 1)
- B: [(C, 7), (D, 5)], given B is connected to C with weight 7 and to D with weight 5, so B's list contains (C, 7) and (D, 5)
- C: [(D, 2)], given C is connected to D with weight 2, so C's list contains (D, 2)
- D: []

Vertex	Adjacency List
A	(B, 3), (C, 1)
B	(C, 7), (D, 5)
C	(D, 2)
D	None

Figure 7: Adjacency List Representation of the Graph

Algorithm 2 Pseudocode of Adjacency Matrix

Input : A graph represented as a list of edges: $(u, v, weight)$, number of vertices n
Output: Adjacency Matrix $matrix$ of size $n \times n$

Initialize an $n \times n$ matrix with all entries as 0
foreach *edge* $(u, v, weight)$ *in the input graph* **do**
    $matrix[u][v] \leftarrow weight$
    if *graph is undirected* **then**
        $matrix[v][u] \leftarrow weight$
return $matrix$

Explanation of the Pseudocode of Adjacency Matrix

- Step 1: Initialize a matrix of size $n \times n$ with all entries set to 0.
- Step 2: For each edge in the input graph, set the corresponding matrix entry to the weight of the edge.
- Step 3: Set the symmetric entry if the graph is undirected.
- Step 4: Finally, return the constructed adjacency matrix.

Adjacency Matrix Representation:

Represent the vertices A, B, C, D as 0, 1, 2, 3 in the martix

- The value at matrix[0][1] is 3, representing an edge from A to B with weight 3
- The value at matrix[0][2] is 1, representing an edge from A to C with weight 1.
- The value at matrix[1][2] is 7, representing an edge from B to C with weight 7.
- The value at matrix[1][3] is 5, representing an edge from B to D with weight 5.
- The value at matrix[2][3] is 2, representing an edge from C to D with weight 2.
- All other values are 0, representing no direct edge between those vertices.

	A	B	C	D
A	0	3	1	0
B	0	0	7	5
C	0	0	0	2
D	0	0	0	0

Figure 8: Adjacency Matrix Representation of the Graph

4.3 Graph Traversal Algorithms (BFS and DFS)

4.31 Explanation

Graph traversal algorithms are used to explore nodes and edges in a graph. BFS and DFS are two of the most common traversal methods. Each has unique properties and is suitable for different types of graph problems.

- BFS: Breadth First Search is a level-order traversal method that explores the graph layer by layer, starting from a given root node. It uses a queue data structure to keep track of nodes that need to be visited. BFS is ideal for finding the shortest path in an unweighted graph and for scenarios where we need to explore all nodes at the current depth level before moving to the next level.
- DFS: Depth First Search is a traversal method that explores as far down a branch as possible before backtracking. It uses a stack to keep track of the nodes to be visited. DFS is useful for pathfinding in scenarios where we need to explore all possible paths, such as solving mazes or detecting cycles in a graph.

In both BFS and DFS, we can keep track of the visitation status of nodes using a set of markers such as pre-visit and post-visit numbers. These markers help in understanding the order in which nodes are visited and processed, which is especially useful for tasks like topological sorting or identifying strongly connected components.

4.32 Exercise

Consider the following graph with vertices A, B, C, D, E, F:

- A is connected to B and C.
- B is connected to D and E.
- C is connected to F.
- D, E, F have no outgoing edges.

Perform a DFS on this graph starting from vertex A. Assign pre-numbers and post-numbers to each vertex.

4.33 Solution

Algorithm 3 DFS with Pre and Post Numbers

Input : A graph $G = (V, E)$, starting vertex v, arrays `pre`, `post`, and `visited`
Output: Pre and Post numbers for each vertex $v \in V$

procedure DFS($G, v, pre, post, visited$)
visited[v] $\leftarrow$ `true // Mark vertex` v `as visited`
pre[v] $\leftarrow$ pre[v] + 1 `// Assign previsit number to` v
foreach *neighbor* u *of* v *in* E **do**
  if *visited[u]* = `false` **then**
    DFS($G, u, pre, post, visited$) `// Recursively explore unvisited neighbor`
post[v] $\leftarrow$ post[v] + 1 `// Assign postvisit number to` v

Explanation of the Pseudocode of Adjacency Matrix

- Step 1: Initialize the pre and post numbers for all vertices to 0.
- Step 2: Mark the current vertex as visited and assign it a pre-visit number.
- Step 3: Recursively explore each unvisited neighbor.
- Step 4: Assign a post-visit number after all neighbors have been explored.
- Step 5: Continue the process until all vertices are visited.

To perform a DFS on the given graph starting from vertex A, the graph can be visually represented as Figure 9:

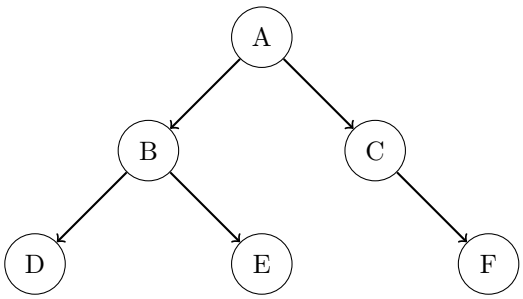

Figure 9: Graph for DFS Traversal

The traversal will assign pre-numbers when a vertex is first visited and post-numbers when all its neighbors have been explored. The DFS step-by-step execution is following :

1. **Start at Vertex A (Figure 10):**

- Assign Pre-number to A as 1
- Push A onto the stack. The stack now contains [A].

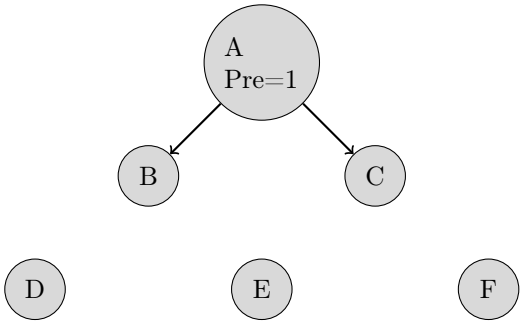

Figure 10: DFS Step 1: Visit A, Pre-number = 1

2. Visit Neighbor B of A (Figure 11):

- Assign Pre-number to B as 2
- Push B onto the stack. The stack now contains [A, B].

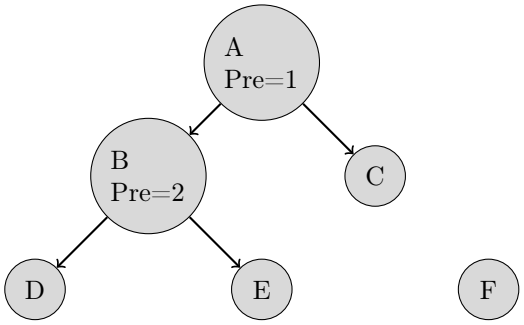

Figure 11: DFS Step 2: Visit B, Pre-number = 2

3. Visit Neighbor D of B (Figure 12):

- Assign Pre-number to D as 3
- Push D onto the stack. The stack now contains [A, B, D].
- D has no neighbors, so assign Post-number to D as 4
- Pop D from the stack. The stack now contains [A, B].

4. Backtrack to Vertex B and Visit Neighbor E (Figure 13):

- Assign Pre-number to E as 5
- Push E onto the stack. The stack now contains [A, B, E].
- E has no neighbors, so assign Post-number to E as 6
- Pop E from the stack. The stack now contains [A, B].

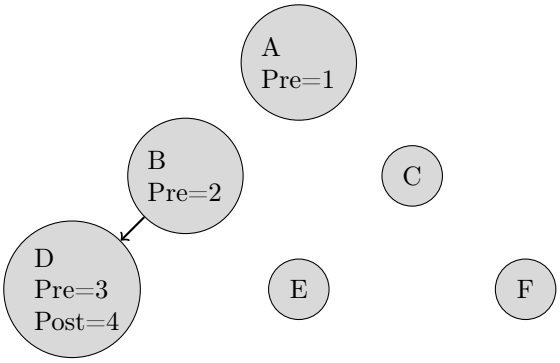

Figure 12: DFS Step 3: Visit D, Pre-number = 3, Post-number = 4

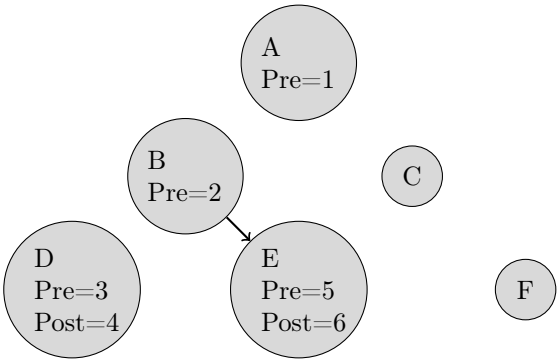

Figure 13: DFS Step 4: Visit E, Pre-number = 5, Post-number = 6

5. Backtrack to Vertex B:

- B has no more neighbors, so assign Post-number to B as 7
- Pop B from the stack. The stack now contains [A].

6. Backtrack to Vertex A and Visit Neighbor C:

- Assign Pre-number to C as 8
- Push C onto the stack. The stack now contains [A, C].

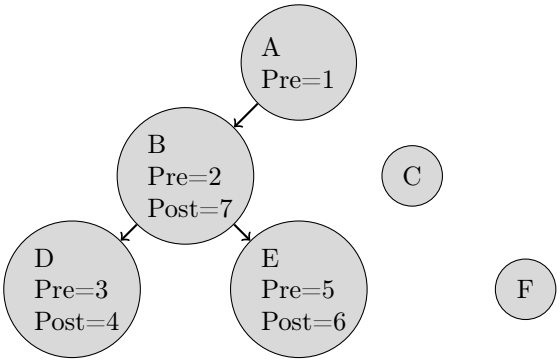

Figure 14: DFS Step 5: Backtrack to B, Post-number = 7

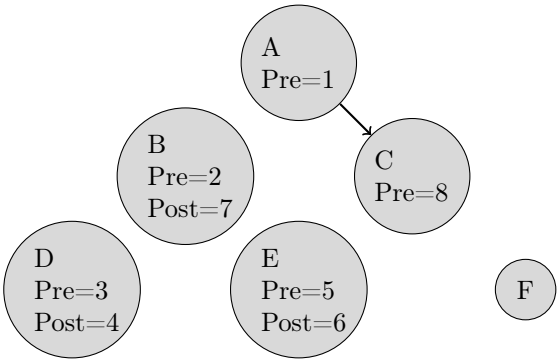

Figure 15: DFS Step 6: Visit C, Pre-number = 8

7. Visit Neighbor F of C:

- Assign Pre-number to F as 9
- Push F onto the stack. The stack now contains [A, C, F].
- F has no neighbors, so assign Post-number to F as 10
- Pop F from the stack. The stack now contains [A, C].

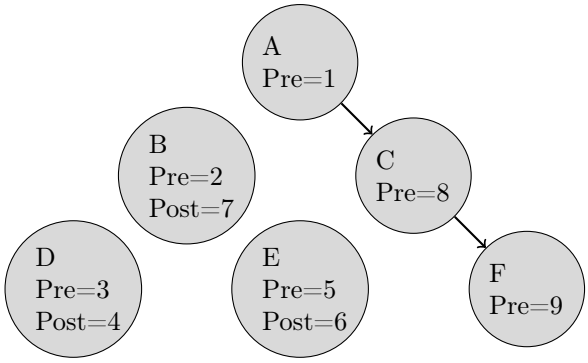

Figure 16: DFS Step 7: Visit F, Pre-number = 9

8. Backtrack to Vertex C:

- C has no more neighbors, so assign Post-number to C as 11
- Pop C from the stack. The stack now contains [A].

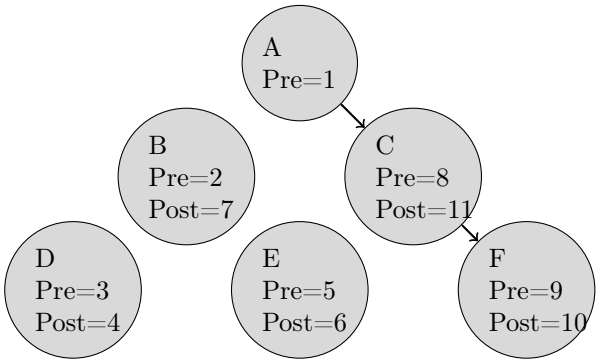

Figure 17: DFS Step 8: Backtrack to C, Post-number = 11

9. Backtrack to Vertex A:

- A has no more neighbors, so assign Post-number to A as 12
- Pop A from the stack. The stack is now empty.

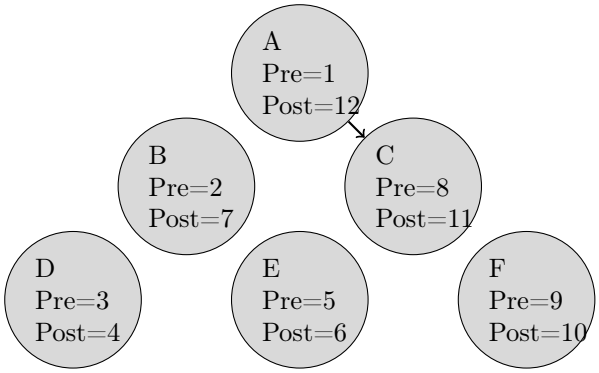

Figure 18: DFS Step 9: Backtrack to A, Post-number = 12

Summarize the pre and post numbers is following:

- A: Pre = 1, Post = 12
- B: Pre = 2, Post = 7
- C: Pre = 8, Post = 11
- D: Pre = 3, Post = 4
- E: Pre = 5, Post = 6
- F: Pre = 9, Post = 10

4.4 Connectivity And Strongly Connected Regions

4.41 Explanation

In graph theory, the concept of connectivity refers to the way vertices in a graph are connected to each other. In an undirected graph, a graph is said to be connected if there is a path between every pair of vertices. If there is no such path between at least one pair of vertices, then the graph is disconnected.

In directed graphs, use the concept of strong connectivity. A directed graph is strongly connected if there are paths from every vertex to every other vertex. In other words, for every pair of vertices u and v, there must be a path from u to v and a path from v to u. If the graph is not strongly connected, it can be partitioned into strongly connected components, such as maximal subgraphs where every vertex is reachable from any other vertex in the same subgraph.

The most common algorithm for finding strongly connected components is the Kosaraju algorithm, which consists of two DFS.

- First DFS: Perform a DFS on the original graph to determine the finishing order in which vertices are fully processed.
- Transpose the graph: Reverse the direction of all edges in the graph.
- Second DFS: Use the finishing order from the first DFS to process the transposed graph, identifying SCCs during the second DFS traversal.

4.42 Exercise

Consider the following directed graph with 7 vertices (A, B, C, D, E, F, G):

- A is connected to B and C.
- B is connected to D.
- C is connected to E.
- D is connected to A and F.
- E is connected to F.
- F is connected to G.
- G is connected to E.

Perform a step-by-step execution of Kosaraju's algorithm to determine the strongly connected components of this graph.

4.43 Solution

Algorithm 4 Kosaraju's Algorithm for Strongly Connected Components

Input : A directed graph $G = (V, E)$
Output: All Strongly Connected Components (SCCs)

```
Initialize an empty stack S
Initialize a visited set visited ← ∅
procedure FirstDFS(v)
Mark v as visited
foreach neighbor u of v do
    if u is not visited then
        FirstDFS(u)
Push v onto S
foreach vertex v ∈ V do
    if v is not visited then
        FirstDFS(v)
Transpose the graph G to get G^T
Clear the visited set
    procedure SecondDFS(v)
Mark v as visited
Print v (belongs to the current SCC)
foreach neighbor u of v in G^T do
    if u is not visited then
        SecondDFS(u)
while S is not empty do
    v ← S.pop()
    if v is not visited then
        SecondDFS(v)
        Print a new line (end of current SCC)
```

Explanation of the Pseudocode of Kosaraju's Algorithm

Step 1: First DFS:

- Perform DFS on the original graph to record the finishing order.
- Push vertices onto the stack based on their finishing time.

Step 2: Transpose the Graph:

- Reverse all edges in the graph to create the transposed graph.

Second DFS on Transposed Graph

- Use the finishing order from the stack to perform DFS on the transposed graph.
- Each DFS traversal on the transposed graph identifies an SCC.

Collect All SCCs

- Repeat the DFS process until all vertices are processed, printing the SCCs as they are discovered.

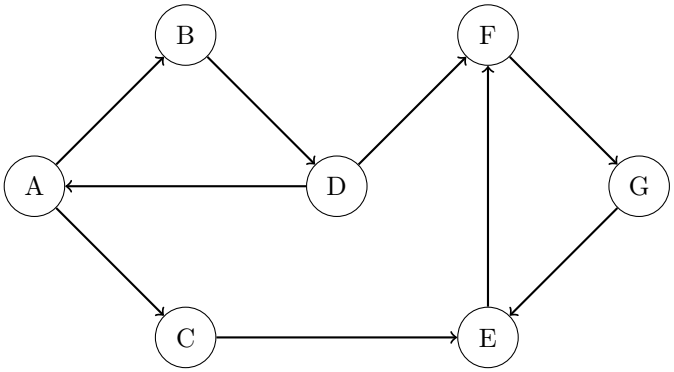

Figure 19: Original Graph for SCC Detection

Given the question, represent the graph as above, to find the strongly connected components of the given graph using Kosaraju's Algorithm, follow these steps:

Step 1: First DFS

I start with any unvisited vertex and perform DFS until all vertices are explored. Once a vertex finishes that means all its adjacent vertices are processed, then I record it. This determines the finishing order for the second DFS.

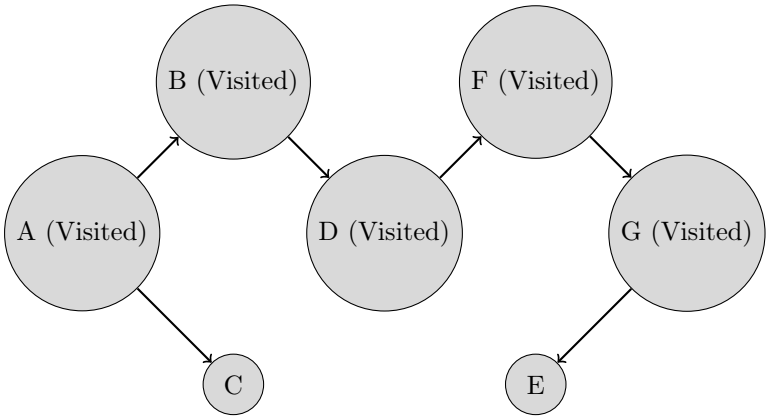

Figure 20: DFS Step 1: Traversing A → B → D → F → G → E

Start with A:

- A -> B -> D -> F -> G -> E
- E visited, backtrack to G
- G visited, backtrack to F
- F visited, backtrack to D
- D visited, backtrack to B
- B visited, backtrack to A
- Continue exploring to C, A -> C -> E(visited, backtrack)

Finishing Order (reverse of completion time):

- E, G, F, D, B, A, C

Step 2: Transpose the Graph

Then I reverse the direction of all edges to get the transposed graph:

- B -> A
- C -> A
- D -> B
- E -> C
- A -> D
- F -> D
- F -> E
- G -> F
- E -> G

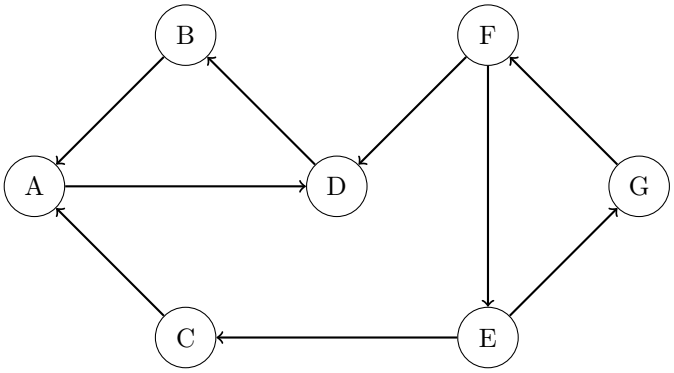

Figure 21: Transposed Graph for SCC Detection

The graph represents as follow

Step 3: Second DFS on Transposed Graph

Start with C:

- C has no unvisited neighbors.
- SCC 1: {C}

Next, A:

- A -> D -> B
- SCC 2: {A, B, D}

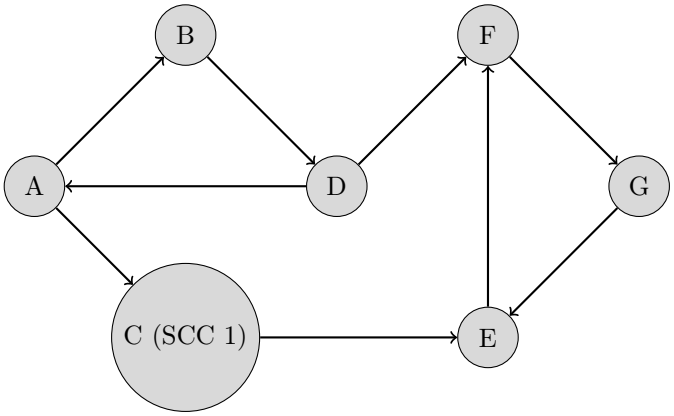

Figure 22: First SCC: {C}

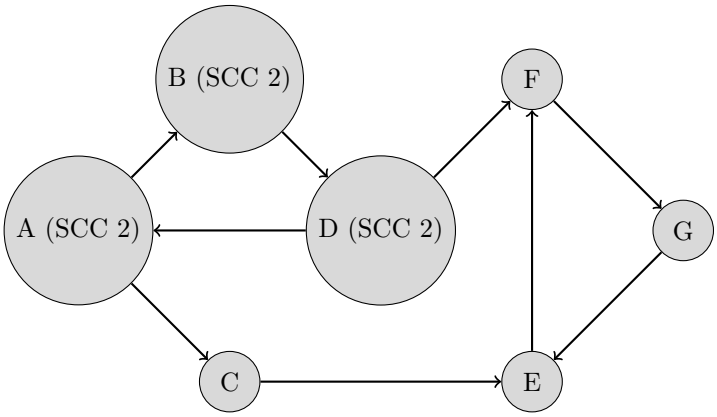

Figure 23: Second SCC: {A, B, D}

Next, E:

- E -> G -> F
- SCC 3: {E, F, G}

Final Result

After carefully applying the algorithm with the corrected transposed graph, I confirm that the SCCs are:

- SCC 1: {C}
- SCC 2: {A, B, D}
- SCC 3: {E, F, G}

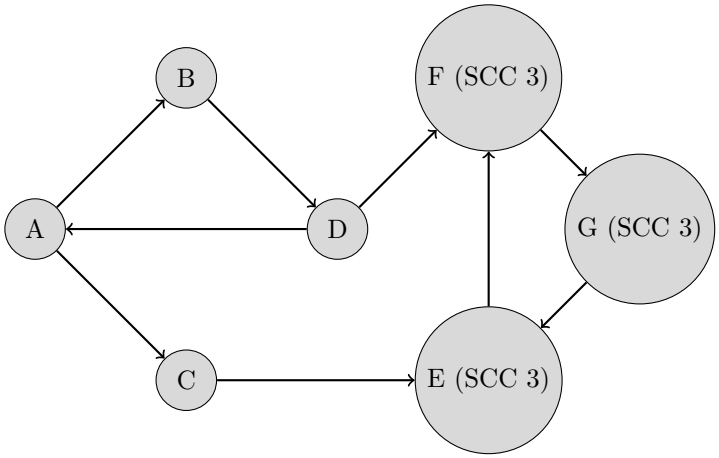

Figure 24: Third SCC: {E, F, G}

5 Graph Algorithms

5.1 Dijkstra's Algorithm

5.11 Explanation

5.12 Exercise

5.13 Solution

5.2 Bellman-Ford Algorithm

5.21 Explanation

5.22 Exercise

5.23 Solution

5.3 Subset Parameters (Matchings and Domination)

5.31 Explanation

5.32 Exercise

5.33 Solution

6 Greedy Algorithms

6.1 Technique Definition

6.11 Explanation

6.12 Exercise

6.13 Solution

6.2 Minimum Spanning Tree And Prim's & Kruskal's Algorithm

6.21 Explanation

6.22 Exercise

6.23 Solution