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 list or an adjacency matrix. An adjacency list 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(Figure 1):

- 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.

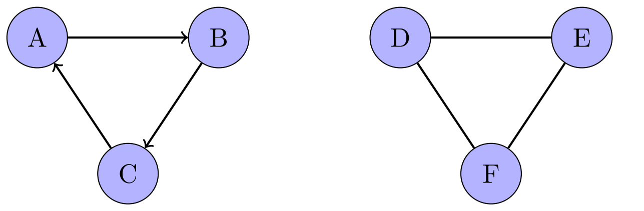

Figure 1: Left: Directed Graph Right: Undirected Graph

Weighted vs Unweighted Graphs(Figure 2):

- 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.

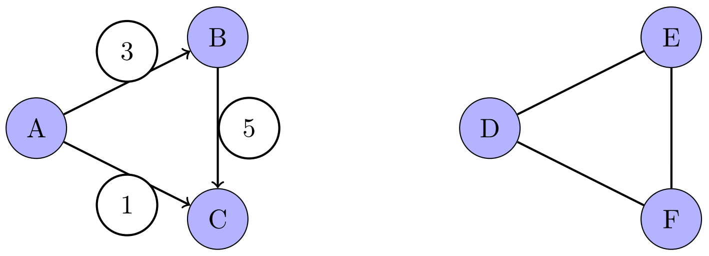

Figure 2: Left: Weighted Graph Right: Unweighted Graph

Simple vs Multi Graphs(Figure 3):

- 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.

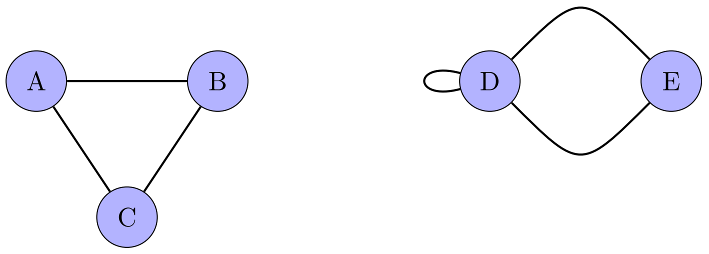

Figure 3: Left: Simple Graph Right: Multi Graph

Connected vs Disconnected Graphs(Figure 4):

- 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.

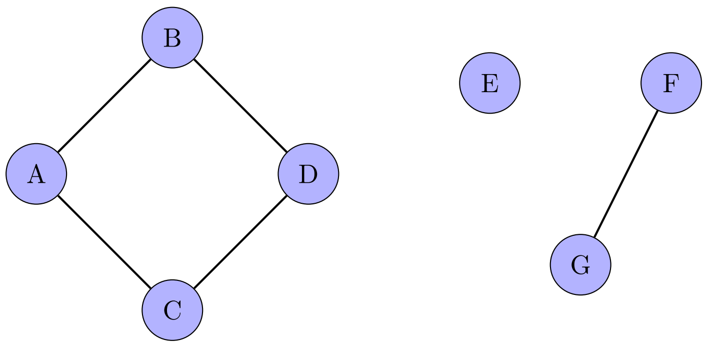

Figure 4: Left: Connected Graph   Right: Disconnected Graph

Cyclic vs Acyclic Graphs(Figure 5):

- 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.

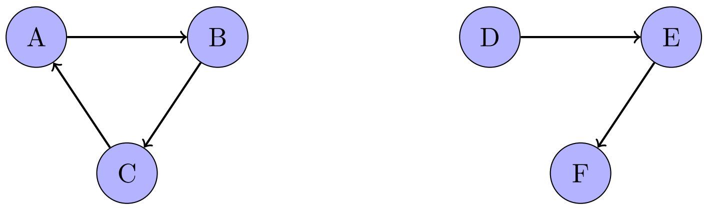

Figure 5: Left: Cyclic Graph   Right: Acyclic Graph

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.
- An adjacency list can represent social networks like Facebook or LinkedIn. Each person maintains a list of friends or connections, it's like if Kai is friends with Will and Raj, Kai's list will contain Will and Raj.

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 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.
- For example, think of it like building a travel map for a group of cities. The adjacency matrix works like a table where each city is listed along the top and side. If there's a direct route between two cities, the corresponding cell in the table records the distance or travel time.

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.

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.

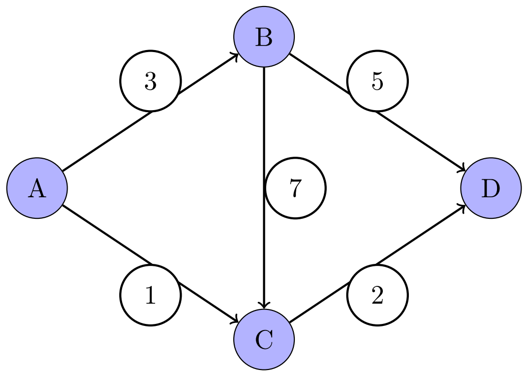

Figure 6: Weighted Graph with vertices A, B, C, D

Represent this graph(Figure 6) using both an adjacency list and an adjacency matrix.

4.23 Solution

Adjacency List Representation(Figure 7):

- 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	$A \to B(3), A \to C(1)$
B	$B \to C(7), B \to D(5)$
C	$C \to D(2)$
D	None

Figure 7: Adjacency List Representation with Arrows

Adjacency Matrix Representation(Figure 8):

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.
- Imagine you enter a large shopping mall, and you want to find the quickest way to reach a special store. You start at the entrance and explore all the nearby stores on the same floor first. After that, you move to the next closest section, continuing this process level by level.

Algorithm 3 Breadth First Search

```
Input  : A graph $G = (V, E)$, starting vertex $s$
Output: All vertices reachable from $s$ along with their shortest path distances
procedure BFS($G, s$)
Create a queue $Q$
Initialize $dist[v] \leftarrow \infty$ for all $v \in V$
$dist[s] \leftarrow 0$ // Set the starting vertex distance to 0
Enqueue $s$ into $Q$
while $Q$ is not empty do
    $v \leftarrow$ Dequeue from $Q$
    foreach neighbor $u$ of $v$ in $E$ do
        if $dist[u] = \infty$ then
            $dist[u] \leftarrow dist[v] + 1$ // Update distance
            Enqueue $u$ into $Q$
```

Explanation of the Pseudocode of BFS

- Step 1: Create a queue and initialize all vertex distances to infinity.
- Step 2: Set the starting vertex distance to 0 and enqueue it.
- Step 3: While the queue is not empty, dequeue a vertex.
- Step 4: For each unvisited neighbor, update its distance and enqueue it.
- Step 5: Continue until the queue is empty, ensuring all reachable vertices are visited.

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.
- Imagine you are exploring a maze with many paths, and you want to find a way out. You decide to go as far as possible along one path until you hit a dead end, and then backtrack to explore the next possible path.

Algorithm 4 Depth First Search

```
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[$v$] ← true // Mark vertex $v$ as visited
pre[$v$] ← 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[$v$] ← post[$v$] + 1 // Assign postvisit number to $v$
```

Explanation of the Pseudocode of DFS

- 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.

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, and 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

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

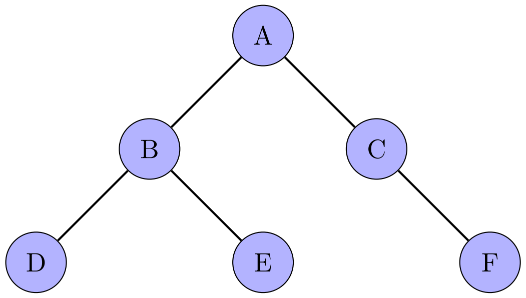

Figure 9: Graph for DFS Traversal

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].

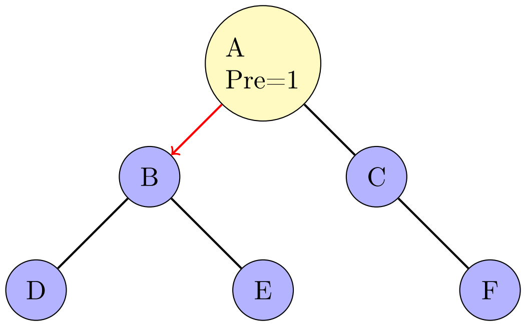

Figure 10: DFS Step 1: Visit A, Pre-number = 1

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].

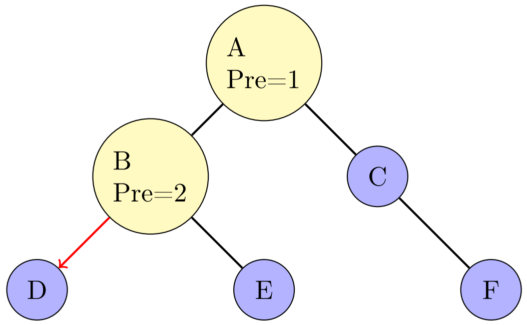

Figure 11: DFS Step 2: Visit B, Pre-number = 2

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].

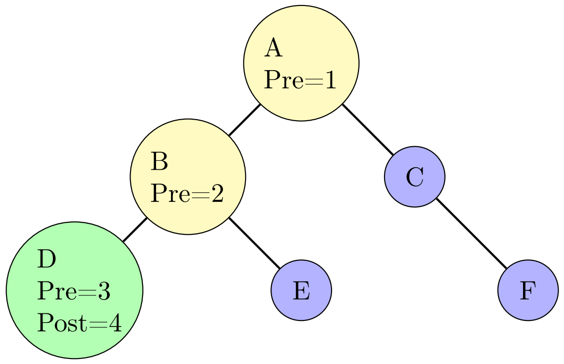

Figure 12: DFS Step 3: Visit D, Pre-number = 3, Post-number = 4

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].

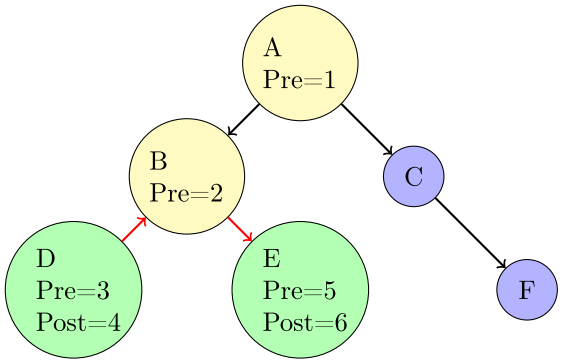

Figure 13: DFS Step 4: Visit E, Pre-number = 5, Post-number = 6

5. Backtrack to Vertex B(Figure 14):

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

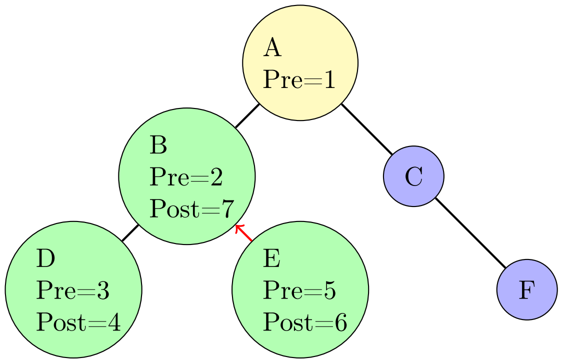

Figure 14: DFS Step 5: Backtrack to B, Post-number = 7

6. Backtrack to Vertex A and Visit Neighbor C(Figure 15):

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

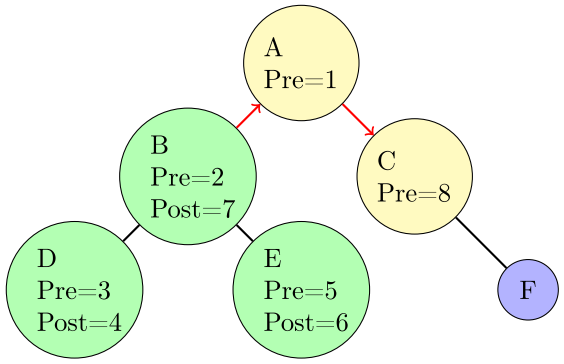

Figure 15: DFS Step 6: Visit C, Pre-number = 8

7. Visit Neighbor F of C(Figure 16):

- 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].

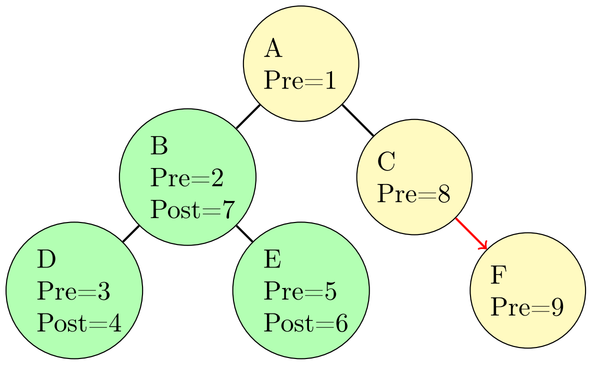

Figure 16: DFS Step 7: Visit F, Pre-number = 9

8. Backtrack to Vertex C(Figure 17):

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

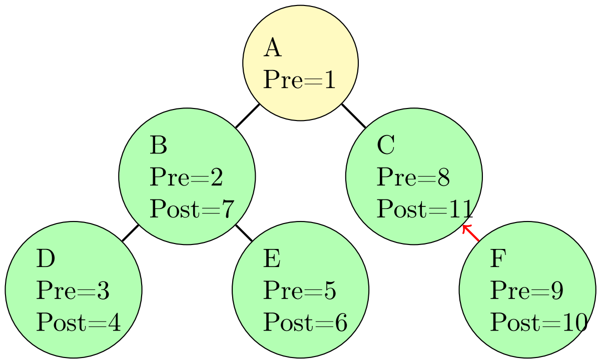

Figure 17: DFS Step 8: Backtrack to C, Post-number = 11

9. Backtrack to Vertex A(Figure 18):

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

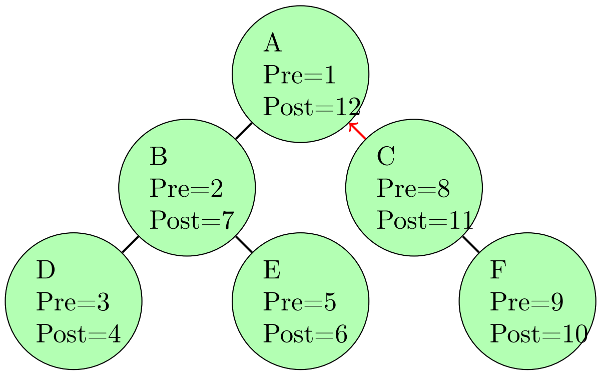

Figure 18: DFS Step 9: Backtrack to A, Post-number = 12

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

Connectivity Region

- In graph theory, the concept of connectivity refers to how vertices in a graph are connected. In an undirected graph, a graph is said to be connected if there is a path between every pair of vertices. The graph is disconnected if there is no such path between at least one pair of vertices.
- Imagine a network of bus routes connecting multiple towns. The network is connected if you can travel from one town to another even with multiple transfers. This is like a graph being connected if there is a path between any two nodes.

Strongly Connected Regions

- 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.
- Imagine specific cities where you can take a direct round-trip that you can travel from City A to City B and back to City A without switching transport modes. In graph terms, a directed graph is strongly connected if there is a path from any vertex to every other vertex, and vice versa.

The most common and efficient algorithm for finding SCC in a directed graph is the Kosaraju algorithm. It operates by performing two DFS on the graph:

First DFS:

- Perform a DFS on the original graph to determine the finishing order in which vertices are fully processed.
- The finishing order indicates the sequence in which vertices are fully explored during the DFS traversal, with the most recently finished vertex listed first.

Transpose of the graph:

- Reverse the direction of all edges in the graph to create the transposed graph.
- If there is an edge $u \to v$ in the original graph, the transposed graph will have $v \to u$.

Second DFS:

- Process vertices in decreasing order of finishing times from the first DFS on the transposed graph.
- Each DFS run on the transposed graph identifies an SCC, where all vertices in the same SCC are reachable from one another.

For example, think of power grids where electricity flows between stations. Kosaraju's algorithm can identify clusters of stations where power can flow in both directions between any two stations. This helps engineers find critical components in the grid that ensure electricity flows reliably, even if certain routes are one-way.

Algorithm 5 Kosaraju's Algorithm for SCCs

```
Input  : A directed graph $G = (V, E)$
Output: All SCCs
Initialize an empty stack $S$
Initialize a visited set $visited \leftarrow \emptyset$
procedure FirstDFS($v$)
Mark $v$ as visited
foreach neighbor $u$ of $v$ in $G$ do
    if $u$ is not visited then
        FirstDFS($u$)
Push $v$ onto $S$  // Record the finishing order
foreach vertex $v \in V$ do
    if $v$ is not visited then
        FirstDFS($v$)
Transpose the graph $G$ to get $G^T$
Clear the visited set: $visited \leftarrow \emptyset$
    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
    $v \leftarrow S.pop()$
    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 a DFS on the original graph G to record the finishing order of vertices.
- As each vertex finishes, push it onto the stack based on its finishing time.
- This ensures that the vertex with the latest finishing time will be processed first in the second DFS.

Step 2: Transpose the Graph:

- Reverse all edges in the graph to create the transposed graph.
- This transposed graph will allow the second DFS to explore the SCCs correctly.

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

- Continue the process until all vertices have been processed from the stack, printing the SCCs as identified by the DFS on the transposed graph.

4.42 Exercise

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

- $A \to B$
- $A \to C$
- $B \to D$
- $C \to E$
- $D \to A$
- $D \to F$
- $E \to F$
- $F \to G$
- $G \to E$

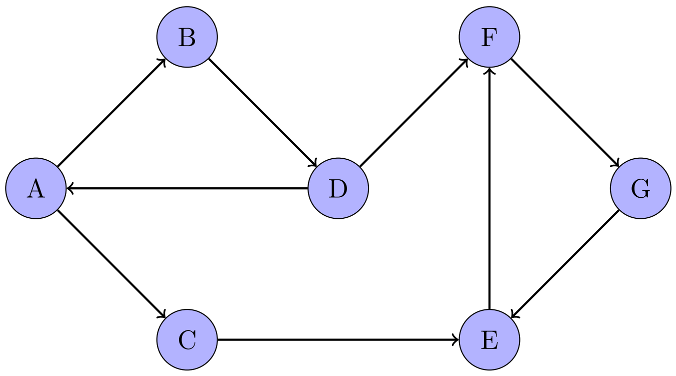

Figure 19: Original Graph for SCC Detection

Perform a step-by-step execution of Kosaraju's algorithm to determine the strongly connected components of this graph(Figure 19).

4.43 Solution

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.

Start at A and Move to B(Figure 20):

- Pre-visit A.
- Explore B from A

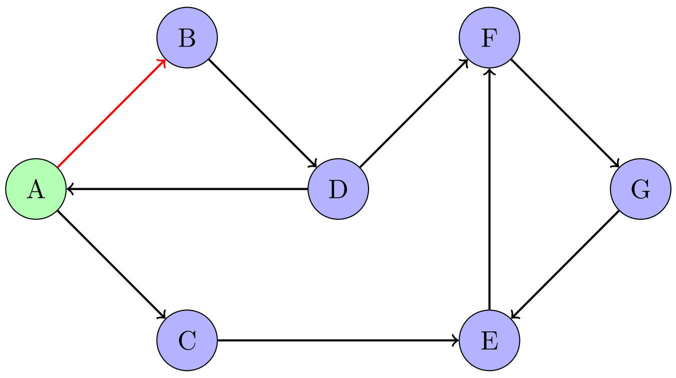

Figure 20: From A to B

From B to D(Figure 21):

- Pre-visit B.
- Move to D

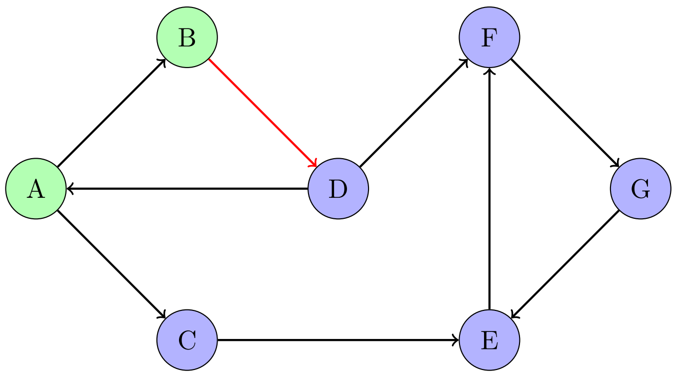

Figure 21: From B to D

From D to F(Figure 22):

- Pre-visit D, and A already visited
- Explore F from D

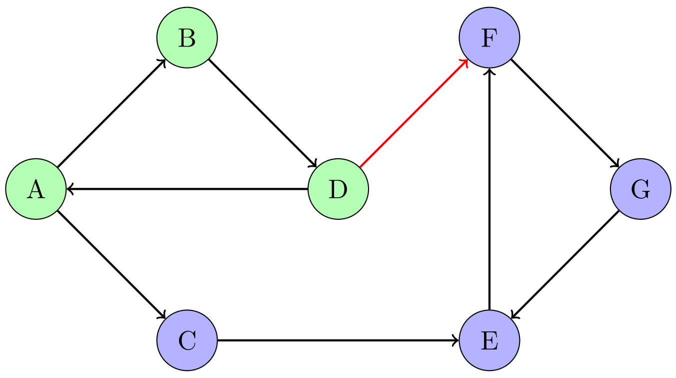

Figure 22: From D to F

From F to G(Figure 23):

- Pre-visit F
- Move to G

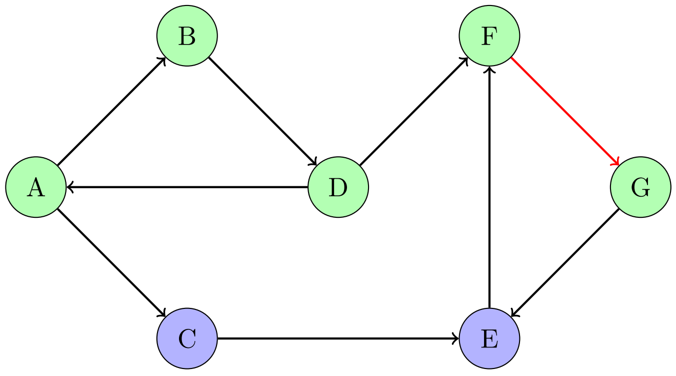

Figure 23: From F to G

From G to E(Figure 24):

- Pre-visit G.
- Move to E

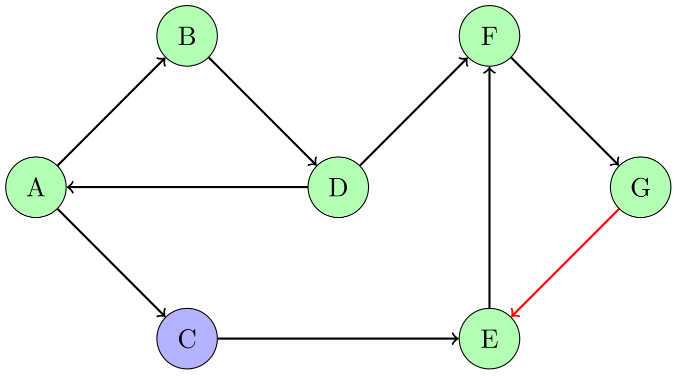

Figure 24: From G to E

Backtrack to A and Explore C(Figure 25):

- F already visited
- Backtrack to A and explore C

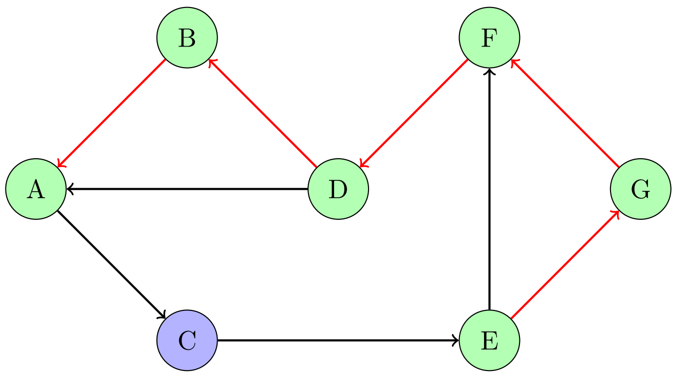

Figure 25: From A to C

Explore C(Figure 26):

- Explore C from A

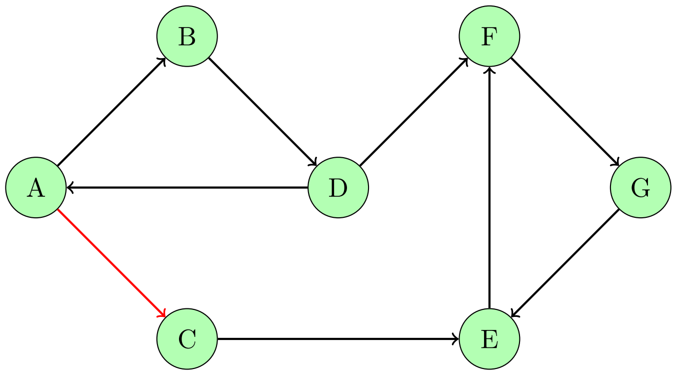

Figure 26: Explore C from A

Finishing Times:

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

Step 2: Transpose the Graph

Then I reverse the direction of all edges to get the transposed graph(Figure 27):

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

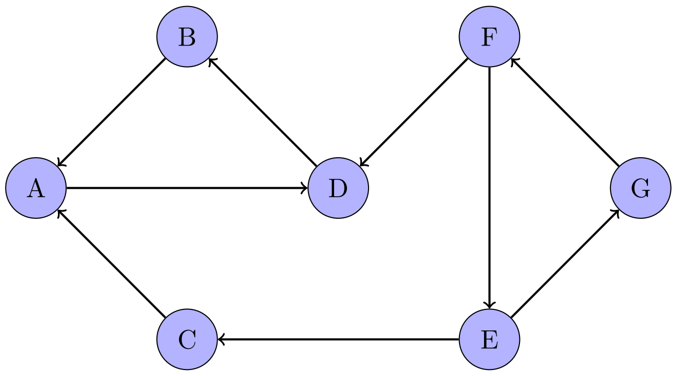

Figure 27: Transposed Graph for SCC Detection

Step 3: Second DFS on Transposed Graph

Start DFS from Vertex E(Figure 28):

- Pre-visit E and explore $E \to G$

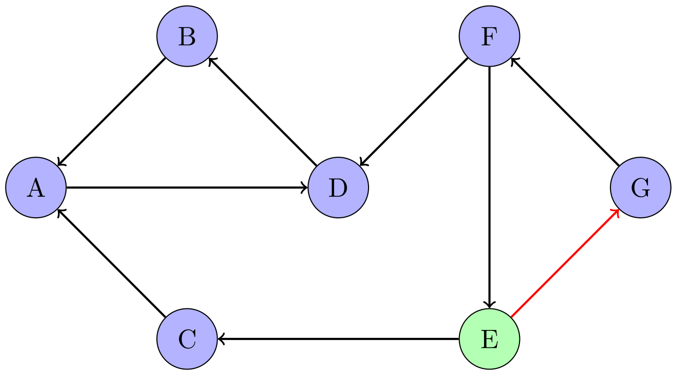

Figure 28: Second DFS: Start from E and Explore G

Explore G from E(Figure 29):

- Pre-visit G and explore $G \to F$

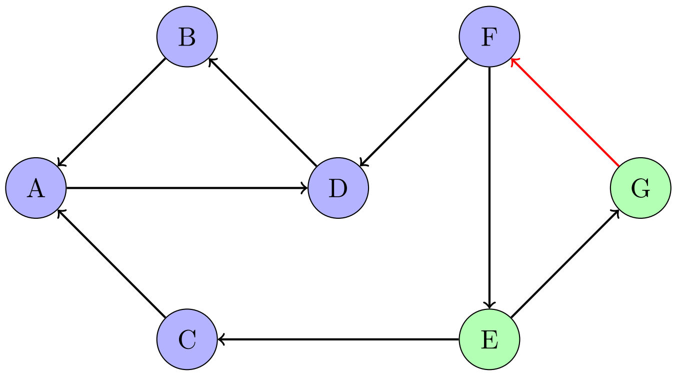

Figure 29: From G to F

Finish DFS for SCC {E, F, G}(Figure 30):

- Pre-visit F and finish exploring
- This completes the SCC {E, F, G}.

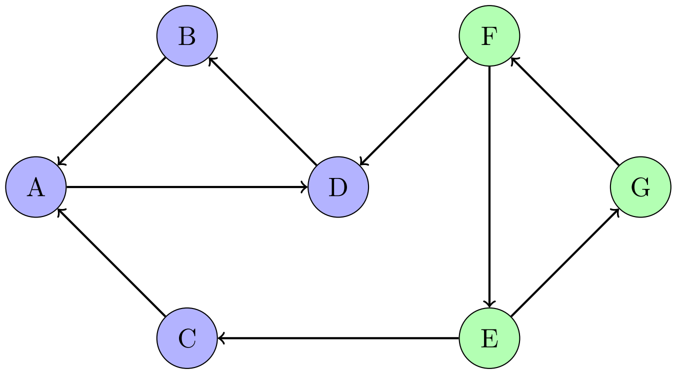

Figure 30: Finish SCC $\{E, F, G\}$

Start DFS from D(Figure 31):

- Pre-visit D and explore $D \to B$

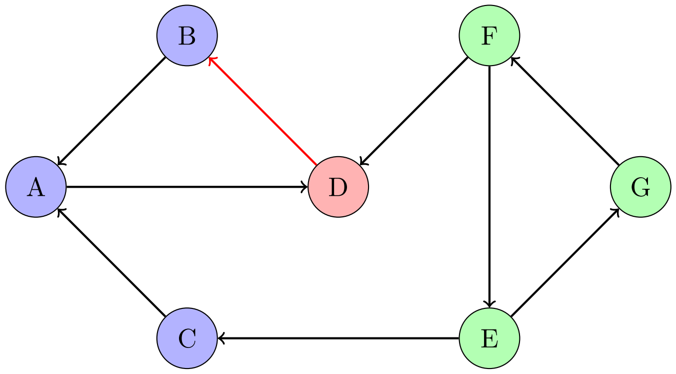

Figure 31: Start from D and Explore B

Explore from B to A(Figure 32):

- Pre-visit B. From B, explore A.

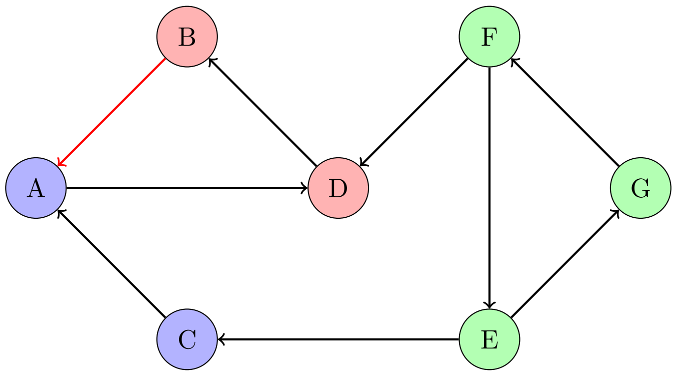

Figure 32: Explore from B to A

Complete SCC {A, B, D}(Figure 33):

- Finish DFS from A, B, and D.

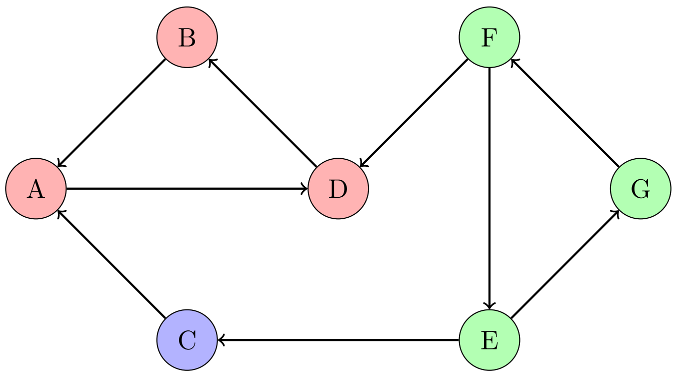

Figure 33: Finish SCC $\{A, B, D\}$

Explore C(Figure 34):

- Pre-visit C. Since C has no unvisited neighbors, it forms an SCC by itself

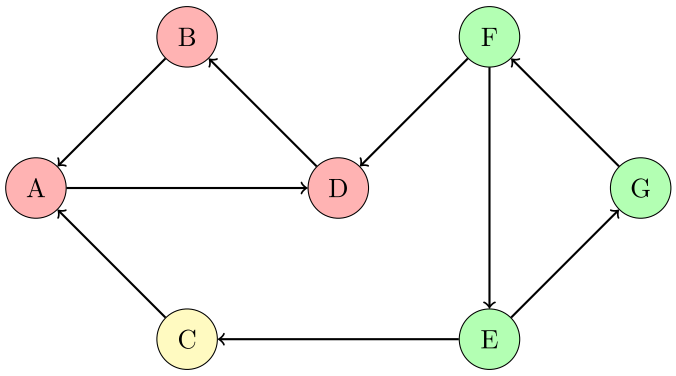

Figure 34: C forms an SCC by itself

Step 4: Collect All SCCs

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

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

5 Graph Algorithms

5.1 Dijkstra's Algorithm

5.11 Explanation

Dijkstra's algorithm is a famous shortest path algorithm used to find the minimum path cost from a given source vertex to all other vertices in a weighted graph. The graph must have non-negative weights, as the algorithm assumes that once a node is visited with the minimum path cost, its value won't need to be updated again.

The idea is to explore the shortest path from the source to every other vertex incrementally. The algorithm uses a priority queue to repeatedly extract the vertex with the smallest known distance, marking it as visited, and updating the distances of its neighbors if shorter paths are found. This process continues until all vertices have been visited or the shortest paths are known.

Algorithm 6 Dijkstra's Algorithm for Shortest Paths

```
Input  : A weighted graph $G = (V, E)$, source vertex $s$
Output: Shortest path distances from $s$ to every vertex $v \in V$

Initialize a min-priority queue $Q$
Initialize $dist[v] \leftarrow \infty$ for all vertices $v \in V$
Set $dist[s] \leftarrow 0$
Insert $(0, s)$ into $Q$
while $Q$ is not empty do
    Extract the vertex $u$ with the smallest distance from $Q$
    foreach neighbor $v$ of $u$ with edge weight $w(u, v)$ do
        if $dist[u] + w(u, v) < dist[v]$ then
            $dist[v] \leftarrow dist[u] + w(u, v)$
            Insert or update $(dist[v], v)$ in $Q$
return $dist$
```

For example, think of it like mapping out a road trip, that the Dijkstra's algorithm ensures you take the most efficient path from point A to point B by considering every possible route and minimizing the overall travel time.

Explanation of the Pseudocode of the Dijkstra's Algorithm

- Initialize the priority queue Q and set the distances of all vertices to infinity, except the source vertex s which is set to 0.
- Extract the vertex u with the smallest known distance from the priority queue.
- For each neighbor v of u, if a shorter path is found, update the distance of v.
- Continue this process until all vertices have been visited, and the shortest paths to all nodes are computed.

5.12 Exercise

Design a weighted graph with 9 vertices labeled, A, B, C, D, E, F, G, H, I. Use the following edges with weights:

- $A \rightarrow B$: 4
- $A \rightarrow H$: 8

- $B \to C$: 8
- $B \to H$: 11
- $C \to D$: 7
- $C \to F$: 4
- $D \to E$: 9
- $D \to F$: 14
- $E \to F$: 10
- $F \to G$: 2
- $G \to H$: 1
- $G \to I$: 6
- $H \to I$: 7

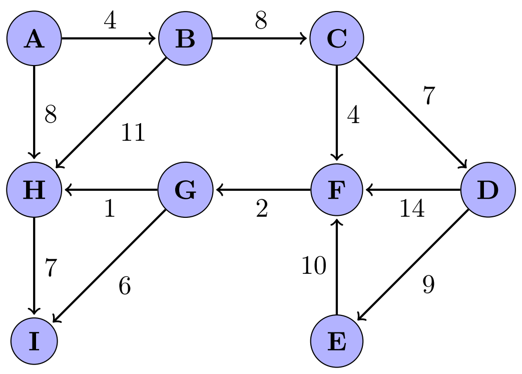

Figure 35: 5.12 Exercise

Perform a step-by-step execution of Dijkstra's algorithm to find the shortest path from the source vertex A to all other vertices.

5.13 Solution

Step by Step execution for the given graph

Step 1: Initial the graph

$$\begin{aligned} dist[A] &= 0, \\ dist[B] &= \infty, \\ dist[C] &= \infty, \\ dist[D] &= \infty, \\ dist[E] &= \infty, \\ dist[F] &= \infty, \\ dist[G] &= \infty, \\ dist[H] &= \infty, \\ dist[I] &= \infty. \end{aligned}$$

- Insert A into the queue: $Q = [(0, A)]$.

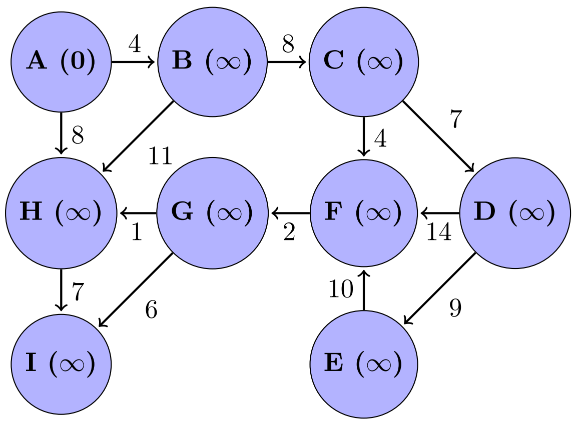

Figure 36: Initial Graph with 9 Vertices and Weighted Edges

Step 2: Extract A

- Update distances:

$$\begin{aligned}
dist[A] &= 0,\\
dist[B] &= 4,\\
dist[C] &= \infty,\\
dist[D] &= \infty,\\
dist[E] &= \infty,\\
dist[F] &= \infty,\\
dist[G] &= \infty,\\
dist[H] &= 8,\\
dist[I] &= \infty.
\end{aligned}$$

- Queue: $Q = [(4, B), (8, H)]$.

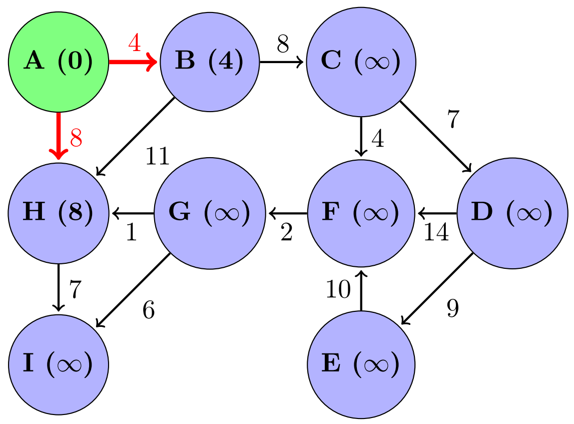

Figure 37: Step 2: Extract A and Update Neighbors B and H

Step 3: Extract B

- Update distances:

$$\begin{aligned}
dist[A] &= 0, \\
dist[B] &= 4, \\
dist[C] &= 12, \\
dist[D] &= \infty, \\
dist[E] &= \infty, \\
dist[F] &= \infty, \\
dist[G] &= \infty, \\
dist[H] &= \min(8, 11) = 8, \\
dist[I] &= \infty.
\end{aligned}$$

- Queue: $Q = [(8, H), (12, C)]$.

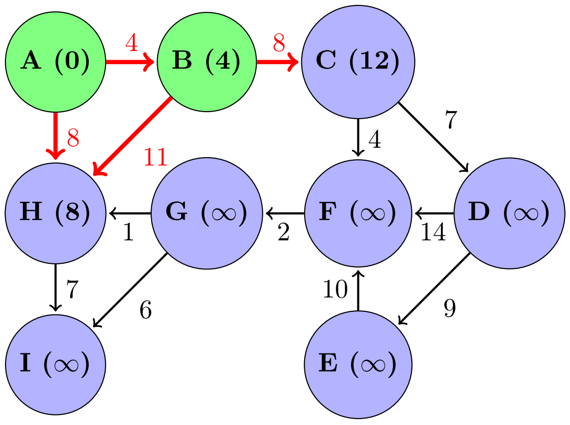

Figure 38: Step 3: Extract B and Update Neighbors C and H

Step 4: Extract H

- Update distances:

$$\begin{aligned}
dist[A] &= 0, \\
dist[B] &= 4, \\
dist[C] &= 12, \\
dist[D] &= \infty, \\
dist[E] &= \infty, \\
dist[F] &= \infty, \\
dist[G] &= 9, \\
dist[H] &= 8, \\
dist[I] &= 15.
\end{aligned}$$

- Queue: $Q = [(9, G), (12, C), (15, I)]$.

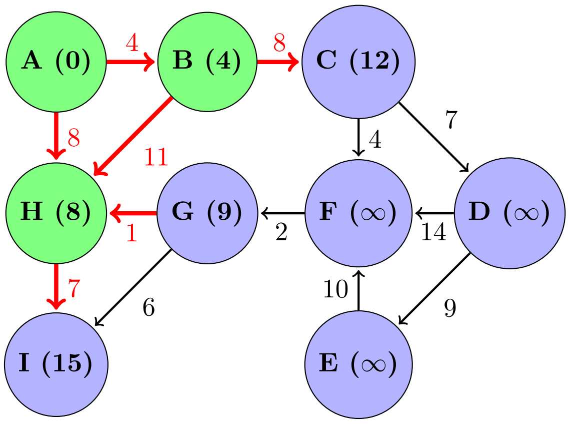

Figure 39: Step 4: Extract H and Update Neighbors G and I

Step 5: Extract G**.**

- Update distances:

$$\begin{aligned} dist[A] &= 0, \\ dist[B] &= 4, \\ dist[C] &= 12, \\ dist[D] &= \infty, \\ dist[E] &= \infty, \\ dist[F] &= 11, \\ dist[G] &= 9, \\ dist[H] &= 8, \\ dist[I] &= 15. \end{aligned}$$

- Queue: $Q = [(11, F), (12, C), (15, I)]$.

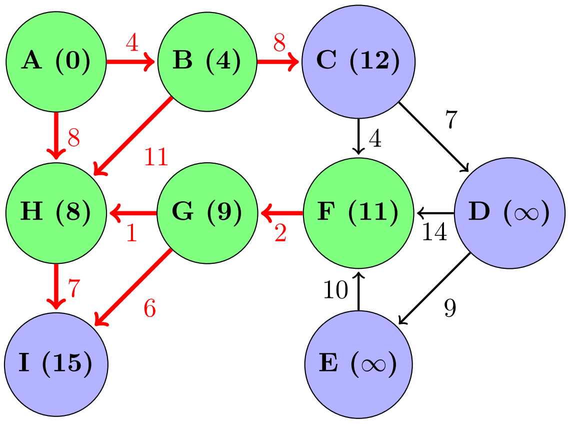

Figure 40: Step 5: Extract G and Update Neighbor F

Step 6: Extract F

- Update distances:

$$\begin{aligned}dist[A] &= 0,\\ dist[B] &= 4,\\ dist[C] &= 12,\\ dist[D] &= 18,\\ dist[E] &= 21,\\ dist[F] &= 11,\\ dist[G] &= 9,\\ dist[H] &= 8,\\ dist[I] &= 15.\end{aligned}$$

- Queue: $Q = [(12, C), (15, I), (18, D), (21, E)]$.

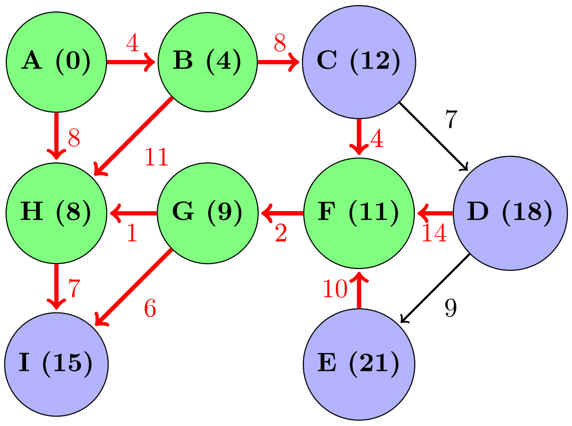

Figure 41: Step 6: Extract F and Update Neighbor D, E

Step 7: Extract C

- No updates required.

$$\begin{aligned}dist[A] &= 0,\\ dist[B] &= 4,\\ dist[C] &= 12,\\ dist[D] &= 18,\\ dist[E] &= 21,\\ dist[F] &= 11,\\ dist[G] &= 9,\\ dist[H] &= 8,\\ dist[I] &= 15.\end{aligned}$$

- Queue: $Q = [(15, I), (18, D), (21, E)]$.

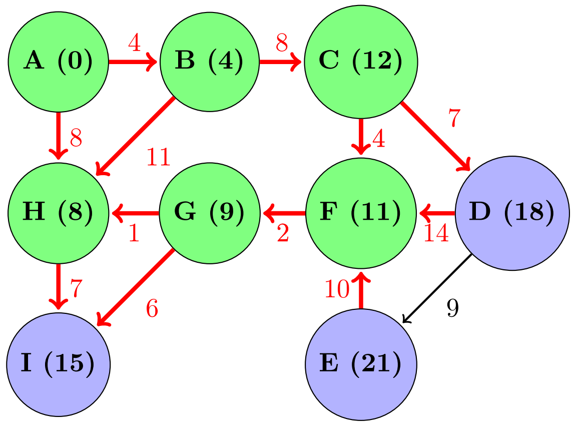

Figure 42: Step 7: Extract C, No Updates Needed

Step 8: Extract I

- No updates required.

$$\begin{aligned} dist[A] &= 0, \\ dist[B] &= 4, \\ dist[C] &= 12, \\ dist[D] &= 18, \\ dist[E] &= 21, \\ dist[F] &= 11, \\ dist[G] &= 9, \\ dist[H] &= 8, \\ dist[I] &= 15. \end{aligned}$$

- Queue: $Q = [(18, D), (21, E)]$.

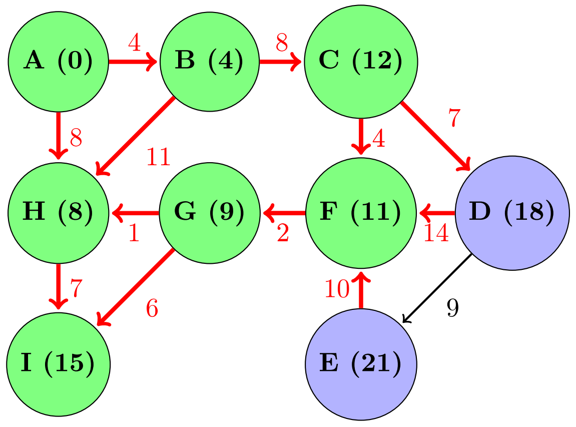

Figure 43: Step 8: Extract I, No Updates Needed

Step 9: Extract D

- No updates required.

$$\begin{aligned}
dist[A] &= 0, \\
dist[B] &= 4, \\
dist[C] &= 12, \\
dist[D] &= 18, \\
dist[E] &= 21, \\
dist[F] &= 11, \\
dist[G] &= 9, \\
dist[H] &= 8, \\
dist[I] &= 15.
\end{aligned}$$

- Queue: $Q = [(21, E)]$.

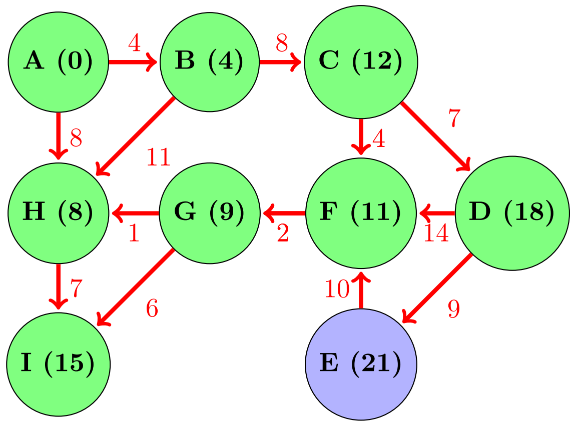

Figure 44: Step 9: Extract D and Update Neighbor E

Step 10: Extract E

- All vertices processed.

$$\begin{aligned}
dist[A] &= 0, \\
dist[B] &= 4, \\
dist[C] &= 12, \\
dist[D] &= 18, \\
dist[E] &= 21, \\
dist[F] &= 11, \\
dist[G] &= 9, \\
dist[H] &= 8, \\
dist[I] &= 15.
\end{aligned}$$

- Queue is now empty.

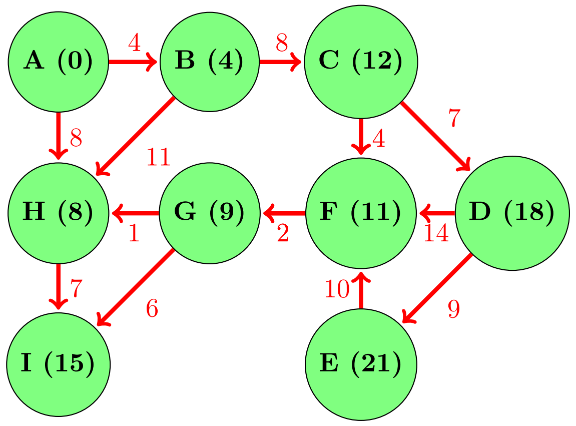

Figure 45: Step 10: Extract E. All vertices processed.

Final Shortest Paths from A:

$$\begin{aligned} A \to A &= 0, \\ A \to B &= 4, \\ A \to H &= 8, \\ A \to G &= 9, \\ A \to F &= 11, \\ A \to C &= 12, \\ A \to I &= 15, \\ A \to D &= 18, \\ A \to E &= 21. \end{aligned}$$

5.2 Bellman-Ford Algorithm

5.21 Explanation

The Bellman-Ford algorithm finds the shortest path from a single source vertex to all other vertices in a weighted graph. The difference between Dijkstra's algorithm and Bellman-Ford's work with graphs containing negative weight edges makes it better for solving real-world problems that might have negative costs. However, it does not work with graphs containing negative weight cycles, as those cycles lead to infinitely decreasing path lengths.

The algorithm works by repeatedly relaxing all edges, meaning it attempts to improve the current shortest path to each vertex by checking if a better path is available through another vertex. Bellman-Ford guarantees the shortest paths if they exist by relaxing all edges |V| - 1 times, where |V| is the number of vertices.

Imagine you are planning a trip where the distances between cities aren't the only point need to think about, probably some routes between cities offer discounts (negative weights) that reduce the overall cost. You need to determine the cheapest path from your home city to all other destinations. This is where Bellman-Ford shines: it will compute the minimum travel cost, even considering routes with discounts.

Algorithm 7 Bellman-Ford Algorithm for Shortest Paths

```
Input  : A weighted graph $G = (V, E)$ with weights $w(u, v)$, source vertex $s$
Output: Shortest path distances from $s$ to every vertex $v \in V$, or detection of a negative weight cycle

Initialize $dist[v] \leftarrow \infty$ for all vertices $v \in V$
Set $dist[s] \leftarrow 0$
for $i \leftarrow 1$ to $|V| - 1$ do
    foreach edge $(u, v) \in E$ with weight $w(u, v)$ do
        if $dist[u] + w(u, v) < dist[v]$ then
            $dist[v] \leftarrow dist[u] + w(u, v)$

foreach edge $(u, v) \in E$ with weight $w(u, v)$ do
    if $dist[u] + w(u, v) < dist[v]$ then
        return Negative weight cycle detected

return $dist$
```

There is Bellman-Ford Algorithm Pseudocode:

- Set the distance to all vertices as infinity, except for the source vertex s, which is set to 0.
- For each of the |V| - 1 iterations, relax all edges. If a shorter path to a vertex is found, update its distance.
- After all relaxations, go through all edges once more. If any distance can still be improved, a negative weight cycle exists.
- The algorithm returns the shortest path distances or reports if a negative weight cycle is detected.

5.22 Exercise

Design a weighted graph with 9 vertices labeled, A, B, C, D, E, F, G, H, I and some negative weights. Use the following edges with weights:

- $A \rightarrow B$: 4
- $A \rightarrow C$: 5

- $B \to C$: -2
- $B \to D$: 6
- $C \to E$: 3
- $D \to F$: 2
- $E \to D$: -4
- $E \to F$: 7
- $F \to G$: 1
- $G \to H$: -1
- $H \to I$: 2
- $I \to G$: 3

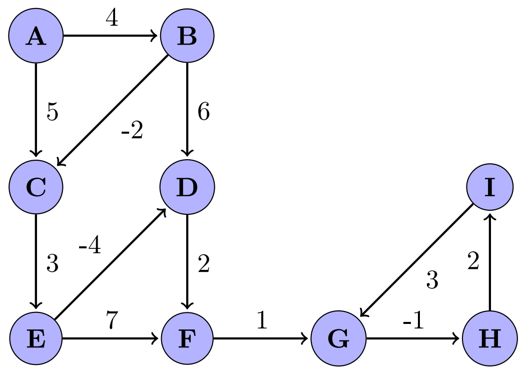

Figure 46: 5.22 Exercise

Perform a step-by-step execution of the Bellman-Ford algorithm starting from the source vertex A. Ensure to detect any negative weight cycles if they exist.

5.23 Solution

Step-by-Step Execution for the Given Graph

Step 1: Initialize the graph, set all distances to infinity except for the source vertex A, which is initialized to 0.

$$\begin{gathered} dist[A] = 0, \\ dist[B] = \infty, \\ dist[C] = \infty, \\ dist[D] = \infty, \\ dist[E] = \infty, \\ dist[F] = \infty, \\ dist[G] = \infty, \\ dist[H] = \infty, \\ dist[I] = \infty. \end{gathered}$$

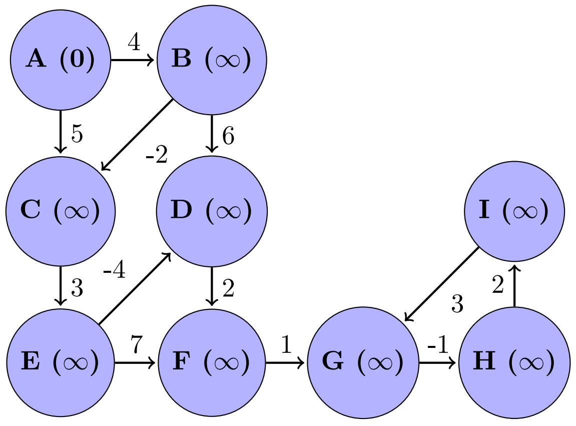

Figure 47: Graph Initialization with All Distances Set to Infinity except A

Iteration 1: Relax all edges, several paths are improved, such as D through E, and H through G.

- Relax edge $A \to B$:
$$dist[B] = \min(\infty, 0 + 4) = 4$$
- Relax edge $A \to C$:
$$dist[C] = \min(\infty, 0 + 5) = 5$$
- Relax edge $B \to C$:
$$dist[C] = \min(5, 4 - 2) = 2$$
- Relax edge $B \to D$:
$$dist[D] = \min(\infty, 4 + 6) = 10$$
- Relax edge $C \to E$:
$$dist[E] = \min(\infty, 2 + 3) = 5$$
- Relax edge $D \to F$:
$$dist[F] = \min(\infty, 10 + 2) = 12$$
- Relax edge $E \to D$:
$$dist[D] = \min(10, 5 - 4) = 1$$
- Relax edge $E \to F$:
$$dist[F] = \min(12, 5 + 7) = 12$$
- Relax edge $F \to G$:
$$dist[G] = \min(\infty, 12 + 1) = 13$$
- Relax edge $G \to H$:
$$dist[H] = \min(\infty, 13 - 1) = 12$$
- Relax edge $H \to I$:
$$dist[I] = \min(\infty, 12 + 2) = 14$$

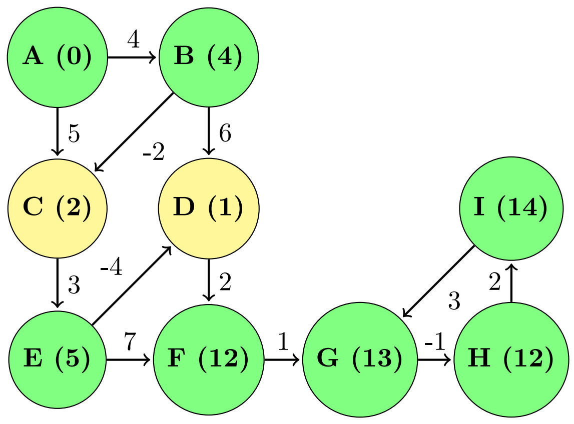

Figure 48: Iteration 1: Updated Distances After Relaxing All Edges

Iteration 2: Relax all edges again, further improvements are made, especially to F and G.

- Relax edge $A \to B$:
$$dist[B] = \min(4, 0 + 4) = 4$$
- Relax edge $A \to C$:
$$dist[C] = \min(2, 0 + 5) = 2$$
- Relax edge $B \to C$:
$$dist[C] = \min(2, 4 - 2) = 2$$
- Relax edge $B \to D$:
$$dist[D] = \min(1, 4 + 6) = 1$$
- Relax edge $C \to E$:
$$dist[E] = \min(5, 2 + 3) = 5$$
- Relax edge $D \to F$:
$$dist[F] = \min(12, 1 + 2) = 3$$
- Relax edge $E \to D$:
$$dist[D] = \min(1, 5 - 4) = 1$$
- Relax edge $E \to F$:
$$dist[F] = \min(3, 5 + 7) = 3$$
- Relax edge $F \to G$:
$$dist[G] = \min(13, 3 + 1) = 4$$
- Relax edge $G \to H$:
$$dist[H] = \min(12, 4 - 1) = 3$$
- Relax edge $H \to I$:
$$dist[I] = \min(14, 3 + 2) = 5$$

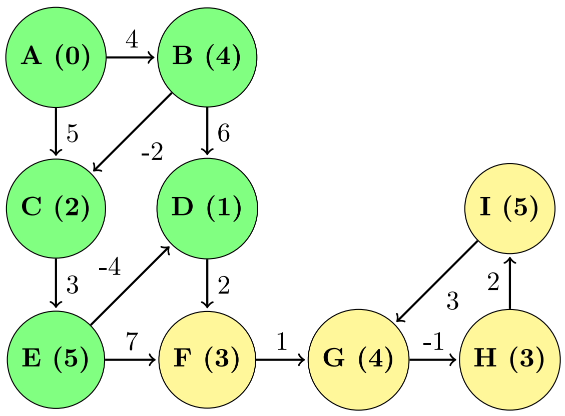

Figure 49: Iteration 2: Further Improved Distances

Iteration 3: Relax all edges again, and perform a final pass over all edges. Since no distances change, the algorithm confirms that it has found the shortest paths.

- Relax edge $A \to B$:
$$dist[B] = \min(4, 0 + 4) = 4$$
- Relax edge $A \to C$:
$$dist[C] = \min(2, 0 + 5) = 2$$
- Relax edge $B \to C$:
$$dist[C] = \min(2, 4 - 2) = 2$$
- Relax edge $B \to D$:
$$dist[D] = \min(1, 4 + 6) = 1$$
- Relax edge $C \to E$:
$$dist[E] = \min(5, 2 + 3) = 5$$
- Relax edge $D \to F$:
$$dist[F] = \min(3, 1 + 2) = 3$$
- Relax edge $E \to D$:
$$dist[D] = \min(1, 5 - 4) = 1$$
- Relax edge $E \to F$:
$$dist[F] = \min(3, 5 + 7) = 3$$
- Relax edge $F \to G$:
$$dist[G] = \min(4, 3 + 1) = 4$$
- Relax edge $G \to H$:
$$dist[H] = \min(3, 4 - 1) = 3$$
- Relax edge $H \to I$:
$$dist[I] = \min(5, 3 + 2) = 5$$
- No updates were made in this iteration, indicating convergence.

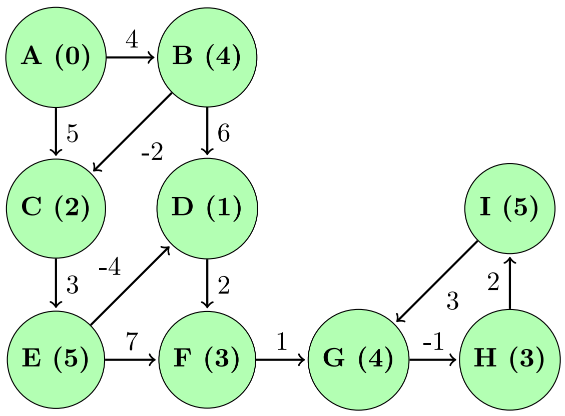

Figure 50: Iteration 3: No More Improvements

Negative Cycle Check

- For each edge $(u, v) \in E$, check if:
$$dist[u] + w(u, v) < dist[v]$$
- No further improvements were found, confirming no negative weight cycle.

Final Shortest Paths from A:

$$\begin{aligned} A \to A &= 0, \\ A \to B &= 4, \\ A \to C &= 2, \\ A \to D &= 1, \\ A \to E &= 5, \\ A \to F &= 3, \\ A \to G &= 4, \\ A \to H &= 3, \\ A \to I &= 5. \end{aligned}$$

5.3 Subset Parameters (Matchings and Domination)

5.31 Explanation

In graph theory, subset parameters involve finding specific sets of edges or vertices that meet particular criteria. Two essential concepts in this category are maximum matching and minimum dominating set.

1.Maximum Matching:

- A matching is a set of edges such that no two edges share a common vertex. A maximum matching is the largest possible matching in a given graph. This concept is often used in task allocation problems where each resource can be assigned to at most one other resource.
- Imagine assigning students to project groups, where each student can only join one group. The goal is to create as many groups as possible without overlapping students, which translates to finding a maximum matching.

Algorithm 8 Maximum Matching Algorithm

```
Input  : Graph $G = (V, E)$ with vertices $V$ and edges $E$
Output: A maximum matching $M \subseteq E$
Initialize an empty matching $M \leftarrow \emptyset$
Mark all vertices in $V$ as unmatched
foreach edge $(u, v) \in E$ do
    if both $u$ and $v$ are unmatched then
        Add edge $(u, v)$ to the matching $M$
        Mark $u$ and $v$ as matched
return $M$
```

Explanation of Maximum Matching Pseudocode:

- Initialize an empty matching M.
- Iterate over all edges. For each edge, if both vertices are unmatched, add it to the matching.
- Mark the matched vertices to ensure they are not reused.
- Return the final matching.

2.Minimum Dominating Set:

- A dominating set is a subset of vertices such that every vertex in the graph is either in this set or adjacent to at least one vertex in the set. A minimum dominating set is the smallest such set.
- Imagine placing security cameras in a building. The goal is to use the fewest number of cameras while ensuring that every area is monitored either directly or by a neighboring camera. This corresponds to finding a minimum dominating set where each camera covers itself and adjacent areas.

Algorithm 9 Minimum Dominating Set Algorithm

```
Input  : Graph $G = (V, E)$ with vertices $V$ and edges $E$
Output: A minimum dominating set $D \subseteq V$
Initialize $D \leftarrow \emptyset$
Mark all vertices in $V$ as uncovered
while there are uncovered vertices do
    Select a vertex $v \in V$ with the maximum number of uncovered neighbors
    Add $v$ to the dominating set $D$
    Mark $v$ and all its neighbors as covered
return $D$
```

Explanation of Minimum Dominating Set Pseudocode:

- Start with an empty dominating set D.
- While there are uncovered vertices, select the vertex with the most uncovered neighbors.
- Add this vertex to the dominating set and mark it and its neighbors as covered.
- Return the final dominating set D.

5.32 Exercise

Given the following graph G = (V, E) with 12 vertices V = {A, B, C, D, E, F, G, H, I, J, K, L}, Design a graph with 12 vertices and solve for the Minimum Dominating Set.

- $A \leftrightarrow B$
- $A \leftrightarrow D$
- $B \leftrightarrow C$
- $B \leftrightarrow E$
- $C \leftrightarrow F$
- $D \leftrightarrow G$
- $E \leftrightarrow F$
- $E \leftrightarrow H$
- $G \leftrightarrow H$
- $G \leftrightarrow I$
- $H \leftrightarrow J$
- $I \leftrightarrow K$
- $J \leftrightarrow L$
- $K \leftrightarrow L$

Here is the initial graph, where all vertices are unselected (blue):

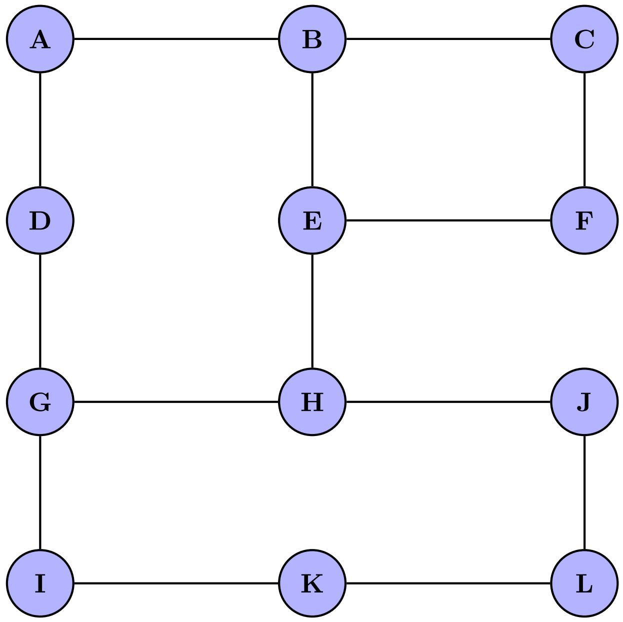

Figure 51: Initial Graph with 12 Vertices

5.33 Solution

Step-by-Step Solution for Minimum Dominating Set

Step 1: Select Vertex B because it has maximum uncovered neighbors

- B is connected A, C and E
- Add B to the dominating set
- Mark A, C, and E as covered
- Uncovered Vertices is D, F, G, H, I, J, K, L.

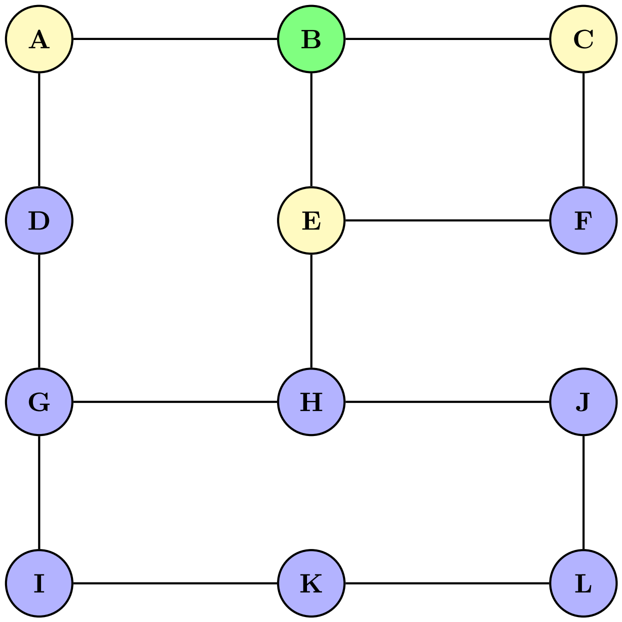

Figure 52: Step 1: Select B, Cover A, C, and E

Step 2: Select Vertex G because it's the most uncovered neighbor)

- G is connected to D, H, and I
- Add G to the dominating set
- Mark D, H, and I as covered
- Uncovered Vertices is F, J, K, L

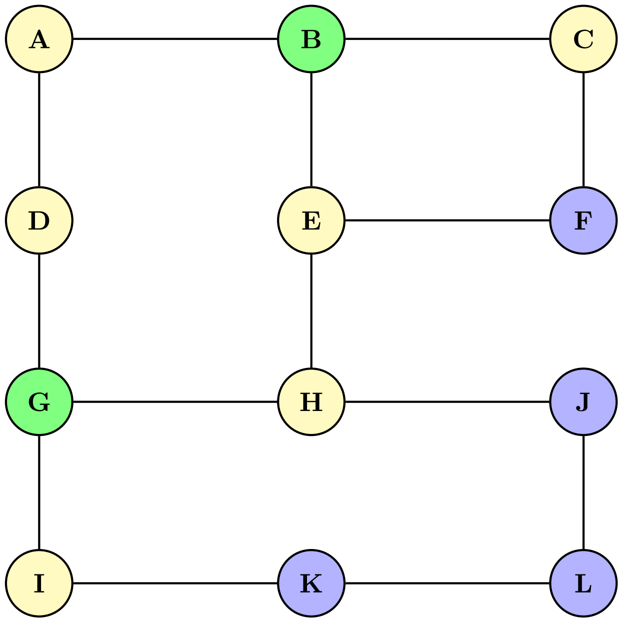

Figure 53: Step 2: Select G, Cover D, H, and I

Step 3: Select Vertex L

- L is connected to J and K
- Add L to the dominating set
- Mark J and K as covered
- Uncovered Vertices is L

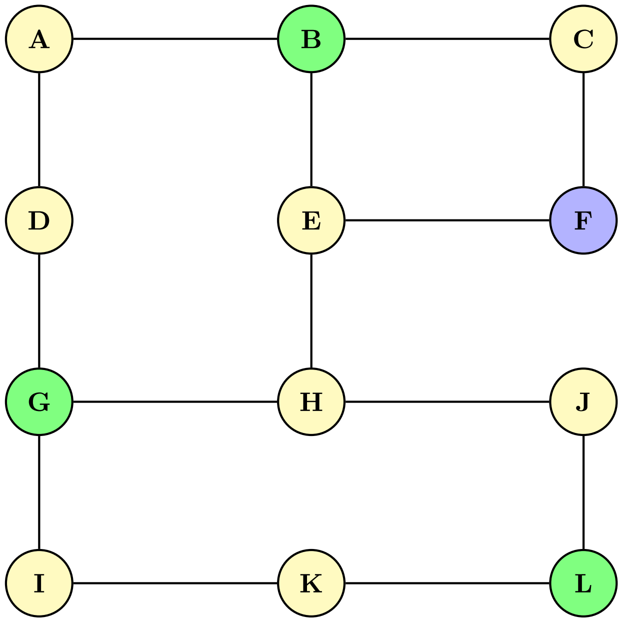

Figure 54: Step 3: Select L, Cover J and K

Step 4: Select Vertex F

- F cover itself
- Add F to the dominating set.
- All vertices are now covered.

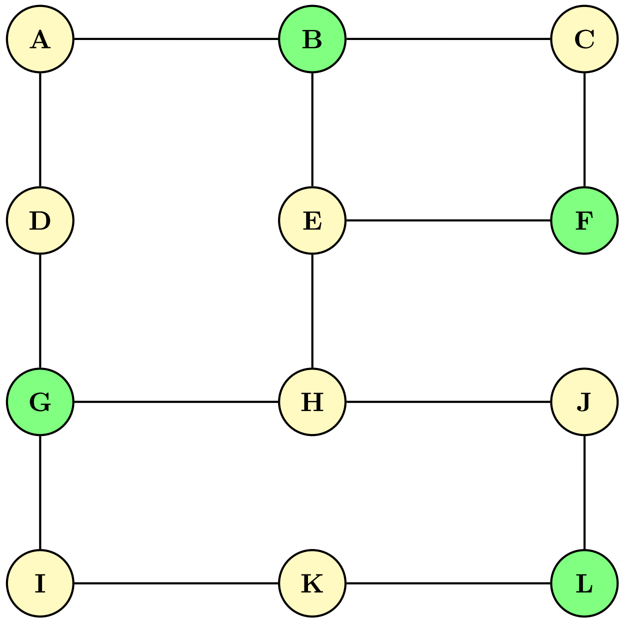

Figure 55: Step 4: Select Vertex F

The final dominating set is:

- D = {B, G, L, F}

6 Greedy Algorithms

6.1 Technique Definition

6.11 Explanation

Greedy algorithms are a class of algorithms that solve optimization problems by making the best possible choice at each step, with the hope that these locally optimal choices will make a globally optimal solution. What makes a greedy algorithm unique is that once a decision is made, it is not revisited, so the algorithm proceeds step-by-step, never backtracking to revise earlier decisions.

A greedy algorithm can be applied to problems that meet two important points:

- A global optimum can be achieved by choosing local optima at each step.
- The solution to the overall problem can be built using solutions to its subproblems.

Goddard defines greedy algorithms as those that "make locally optimal choices at each step with the hope of achieving a globally optimal solution." This approach works only for problems where making the best local decision leads to an optimal global solution, otherwise, the algorithm may result in a suboptimal result.

Every time I use a greedy algorithm, I can't help but think, this is just like life. We make decisions step by step, choosing what seems best in the moment, without knowing the full picture. And just like in life, the path unfolds through these seemingly small choices. So my life word is, every choice you make, no matter how small, is the best decision you could make with what you knew at the time. Life, like a greedy algorithm, moves forward without looking back, the only way is keep moving and trust your path, and never regret where it takes you.

6.12 Example and Pseudocode

Imagine planning an event schedule for a conference. Each session has a start time and an end time, and no two sessions can overlap. The goal is to fit in as many sessions as possible without conflicts.

A greedy algorithm for this problem would:

- Sort all sessions by their end times.
- Select the earliest finishing session first.
- Skip any sessions that overlap with the currently selected session.
- Continue adding sessions until no more non-overlapping sessions can be added.

Algorithm 10 Greedy Algorithm for Scheduling

```
Input  : A list of sessions with start and end times
Output: A set of non-overlapping sessions
Sort the sessions by their end times
Initialize an empty schedule $S \leftarrow \emptyset$
foreach session $i$ in the sorted list do
    if session $i$ does not overlap with the last session in $S$ then
        Add session $i$ to $S$
return $S$
```

This approach works because select the earliest finishing session can be maximizes the number of future sessions that can still fit into the schedule, get an optimal solution.

Explanation of the Pseudocode:

- The sessions are sorted by their end times to prioritize those that finish earlier.
- For each session in the sorted list, the algorithm checks if it overlaps with the last selected session.
- If there is no overlap, the session is added to the schedule.
- The final schedule is returned with the maximum number of non-overlapping sessions.

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

6.21 Explanation

A Minimum Spanning Tree (MST) is a subset of edges from a connected, weighted graph that connects all the vertices together with the minimum possible total edge weight and without any cycles. Basically, an MST ensures that all nodes are connected while minimizing the overall cost.

Prim's Algorithm:

- Prim's algorithm starts with a single vertex and grows the MST by adding the smallest possible edge from the visited vertices to a new, unvisited vertex. It continues until all vertices are included.

Algorithm 11 Prim's Algorithm for Minimum Spanning Tree

```
Input  : A connected graph $G = (V, E)$ with weights $w(u, v)$
Output: A minimum spanning tree $T$
Select an arbitrary vertex $s \in V$ as the starting point
Initialize $T \leftarrow \emptyset$
Initialize a priority queue $Q$ containing all edges adjacent to $s$
Mark $s$ as visited
while there are unvisited vertices do
    Extract the smallest edge $(u, v)$ from $Q$
    if $v$ is unvisited then
        Add $(u, v)$ to $T$
        Mark $v$ as visited
        Add all edges adjacent to $v$ to $Q$
return $T$
```

Imagine you are setting up a local electric power grid. You want to connect every home using the shortest possible total length of wires. Starting from the power station, Prim's algorithm ensures you build the network incrementally by adding the shortest new connections possible, minimizing the overall cost.

Explanation of Prim's Algorithm Pseudocode

- Start with an arbitrary vertex s.
- Use a priority queue to store edges connected to the visited vertices.
- At each step, select the smallest edge that connects a visited vertex to an unvisited vertex.
- Continue until all vertices are visited, forming the MST.

Kruskal's Algorithm:

- Kruskal's algorithm builds the MST by sorting all edges by weight and adding the smallest edges one by one. It only adds an edge if it doesn't create a cycle, using a disjoint-set data structure to track connected components efficiently.

Algorithm 12 Kruskal's Algorithm for Minimum Spanning Tree

```
Input  : A connected graph $G = (V, E)$ with weights $w(u, v)$
Output: A minimum spanning tree $T$
Initialize $T \leftarrow \emptyset$
Sort all edges in $E$ by their weights
Initialize a disjoint-set data structure $DS$
foreach edge $(u, v) \in E$ in sorted order do
    if $u$ and $v$ are in different components in $DS$ then
        Add $(u, v)$ to $T$
        Union the components of $u$ and $v$ in $DS$
return $T$
```

Imagine you are designing highways between cities to minimize the overall cost. Kruskal's algorithm helps select the shortest road segments between pairs of cities while ensuring all cities are connected without forming any loops or unnecessary roads.

Explanation of Kruskal's Algorithm Pseudocode

- Sort all edges in ascending order by weight.
- Use a disjoint-set data structure to track connected components.
- Add the smallest edge to the MST, ensuring it doesn't create a cycle.
- Continue until all vertices are connected.

6.22 Exercise

Given the following weighted graph with 9 vertices V = {A, B, C, D, E, F, G, H, I}, find the minimum spanning tree (MST) using both Prim's Algorithm and Kruskal's Algorithm.

- $A \leftrightarrow B : 4$
- $A \leftrightarrow H : 8$
- $B \leftrightarrow C : 8$
- $B \leftrightarrow H : 11$
- $C \leftrightarrow D : 7$
- $C \leftrightarrow F : 4$
- $D \leftrightarrow E : 9$
- $D \leftrightarrow F : 14$
- $E \leftrightarrow F : 10$
- $F \leftrightarrow G : 2$
- $G \leftrightarrow H : 1$
- $G \leftrightarrow I : 6$
- $H \leftrightarrow I : 7$

6.23 Solution of Prim's Algorithm

Prim's Algorithm step-by-step execution to find an MST

Step 1: Initialize the Graph

- All distances are set to infinity, except for the starting vertex A, which is set to 0.
- No edges have been added yet, and all vertices are unvisited.

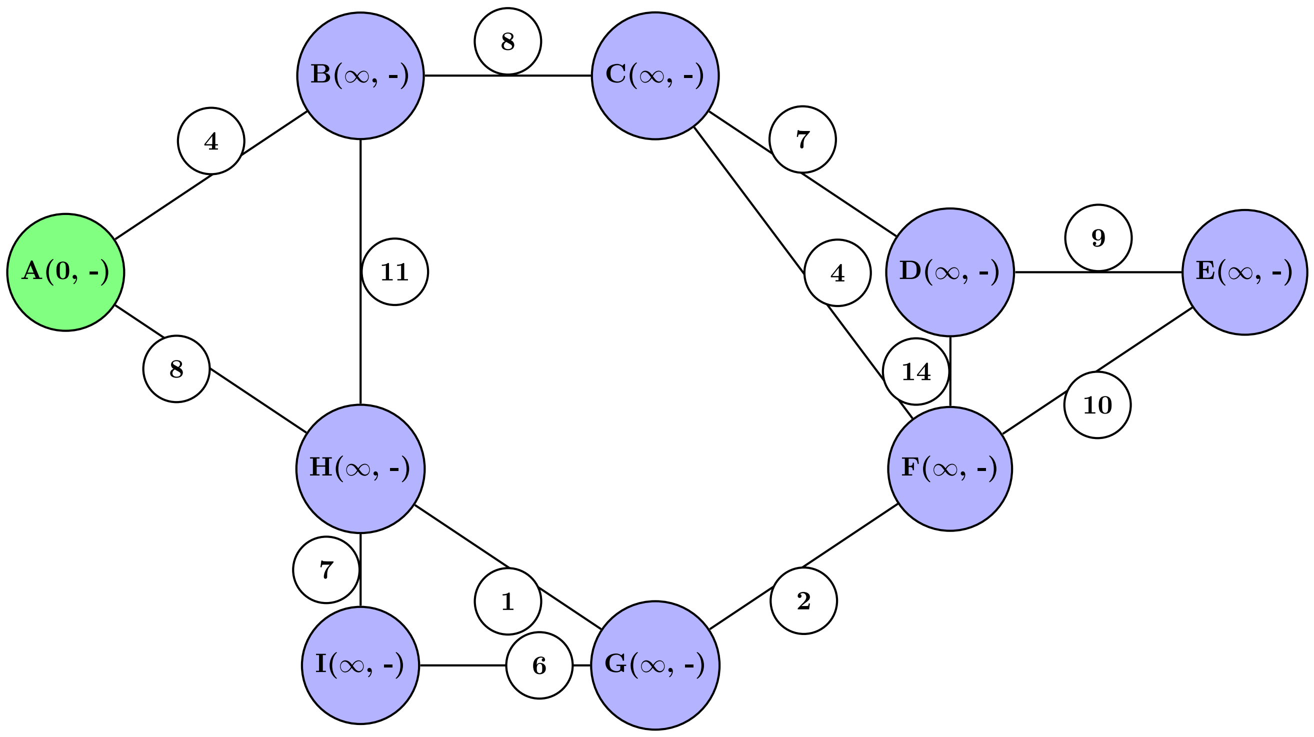

Figure 56: Step 1: Initial Graph with Distances

Step 2: Start at Vertex A

- Start with vertex A.
- Update neighbors B, distance becomes 4, Previous = A.
- Update neighbors H, distance becomes 8, Previous = A.

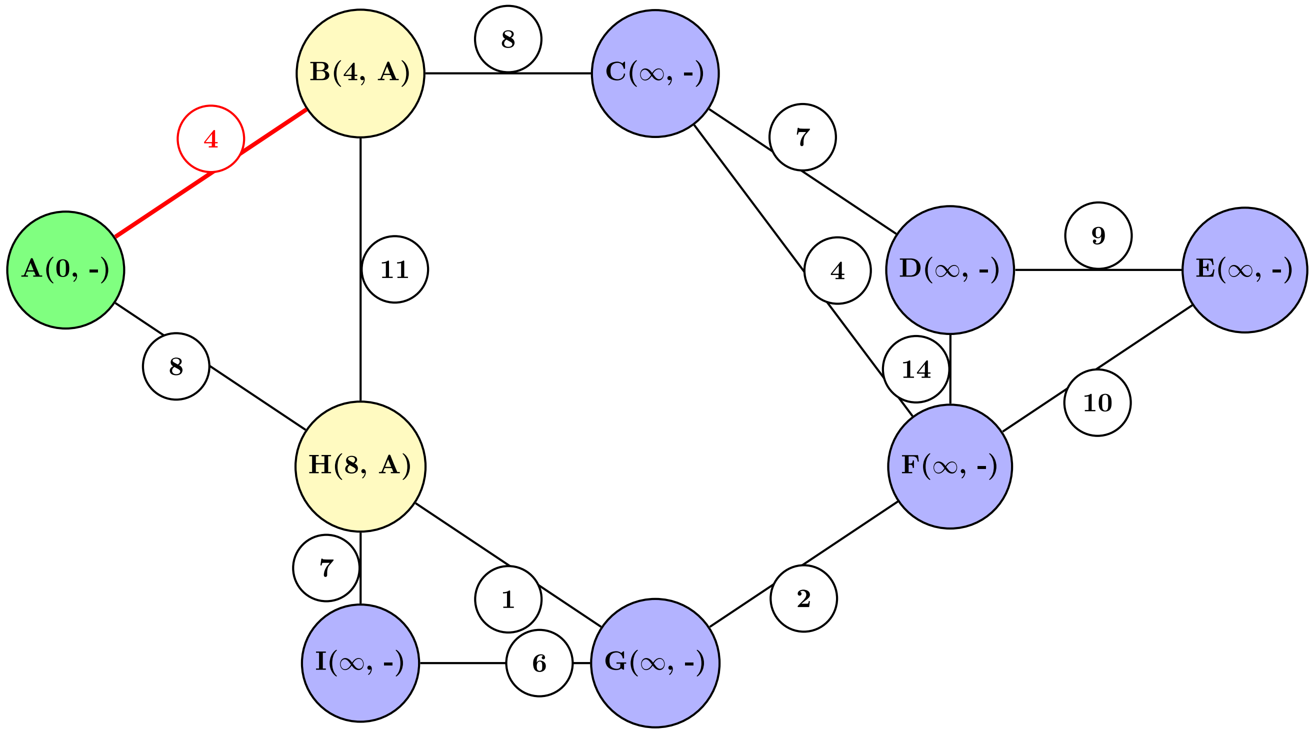

Figure 57: Step 2: Update B and H from A

Step 3: Select B and Update Neighbors

- Select B (smallest distance = 4), Mark B as visited.
- Update neighbors C, distance becomes 8, Previous = B.
- No update to the neighbor H since 11 > 8

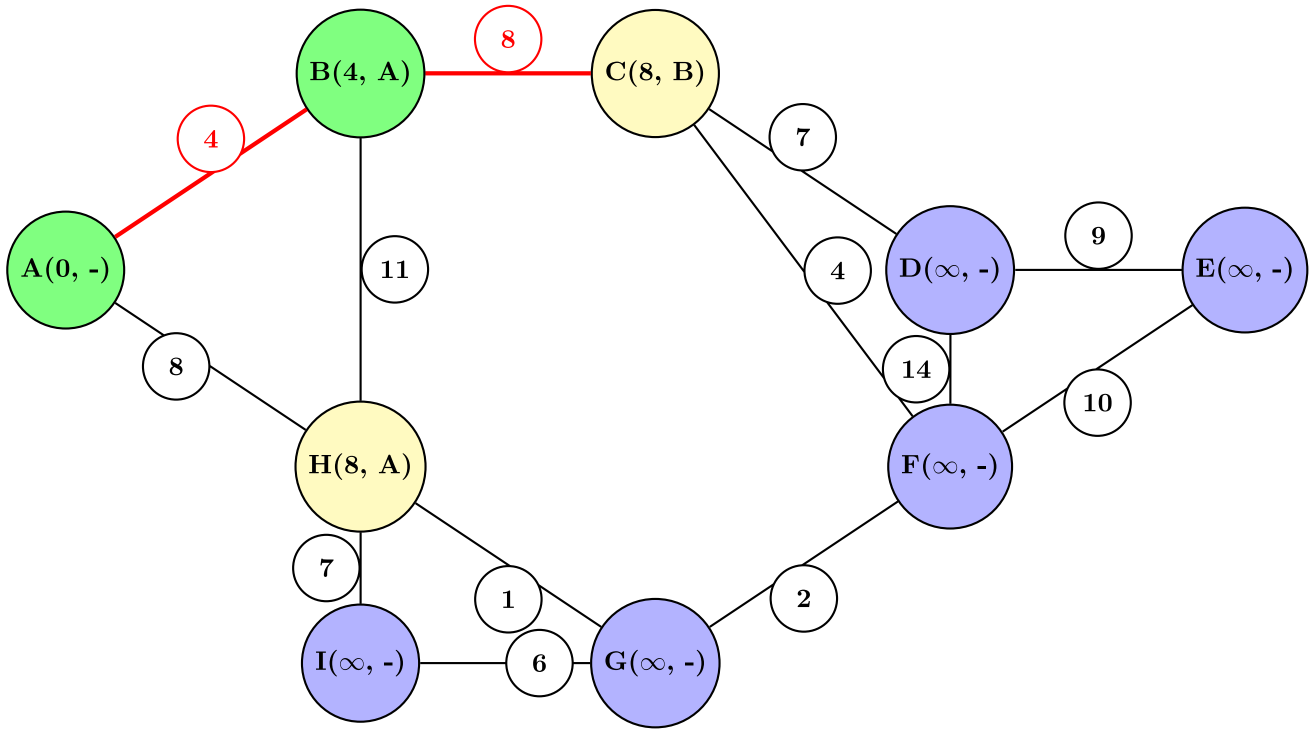

Figure 58: Step 3: Select B and Update C

Step 4: Select C and Update Neighbors

- Select C (smallest distance = 8, from B).Mark vertex C as visited.
- Update neighbors D, distance becomes 8, Previous = 7.
- Update neighbors F, distance becomes 8, Previous = 4.

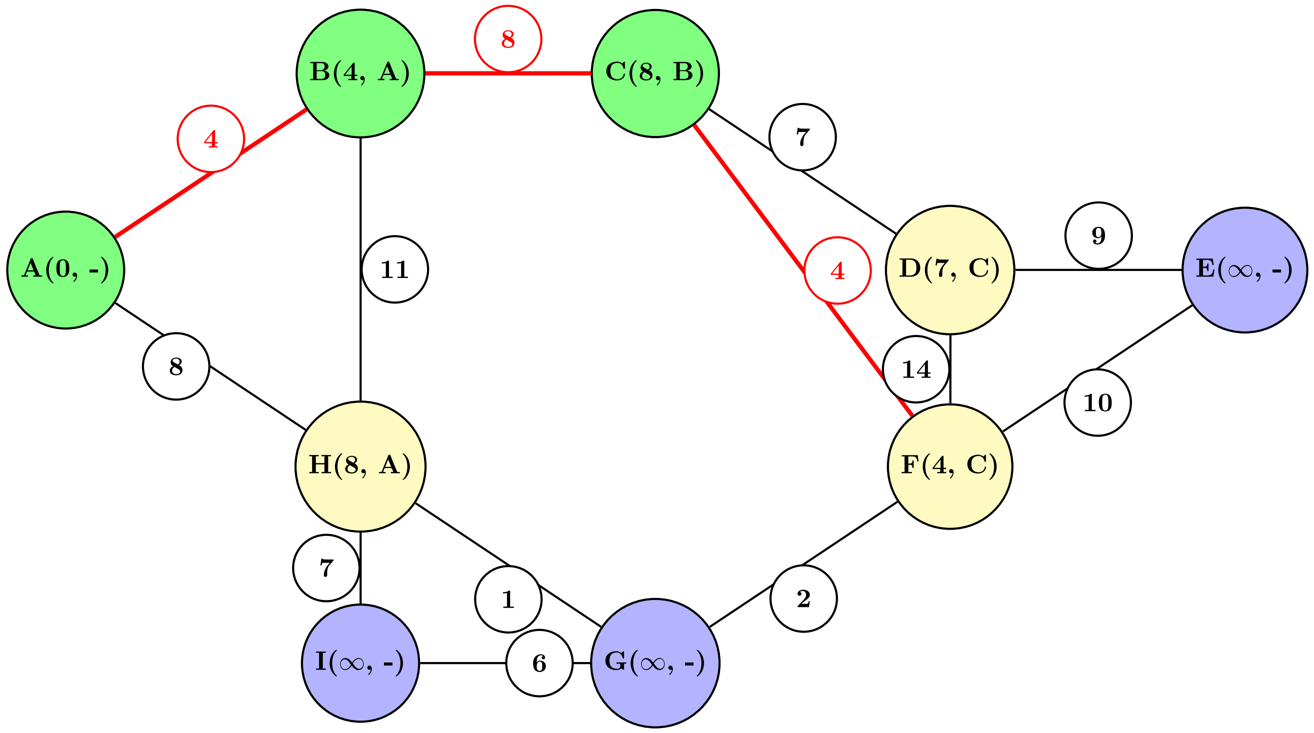

Figure 59: Step 4: Select C and Update Neighbors

Step 5: Select F and Update Neighbors

- Select F (smallest distance = 4 from C).
- Mark F as visited.
- No update to neighbors D, since $14 > 7$.
- Update neighbors E, distance becomes 10, Previous = F.
- Update neighbors G, distance becomes 2, Previous = F.

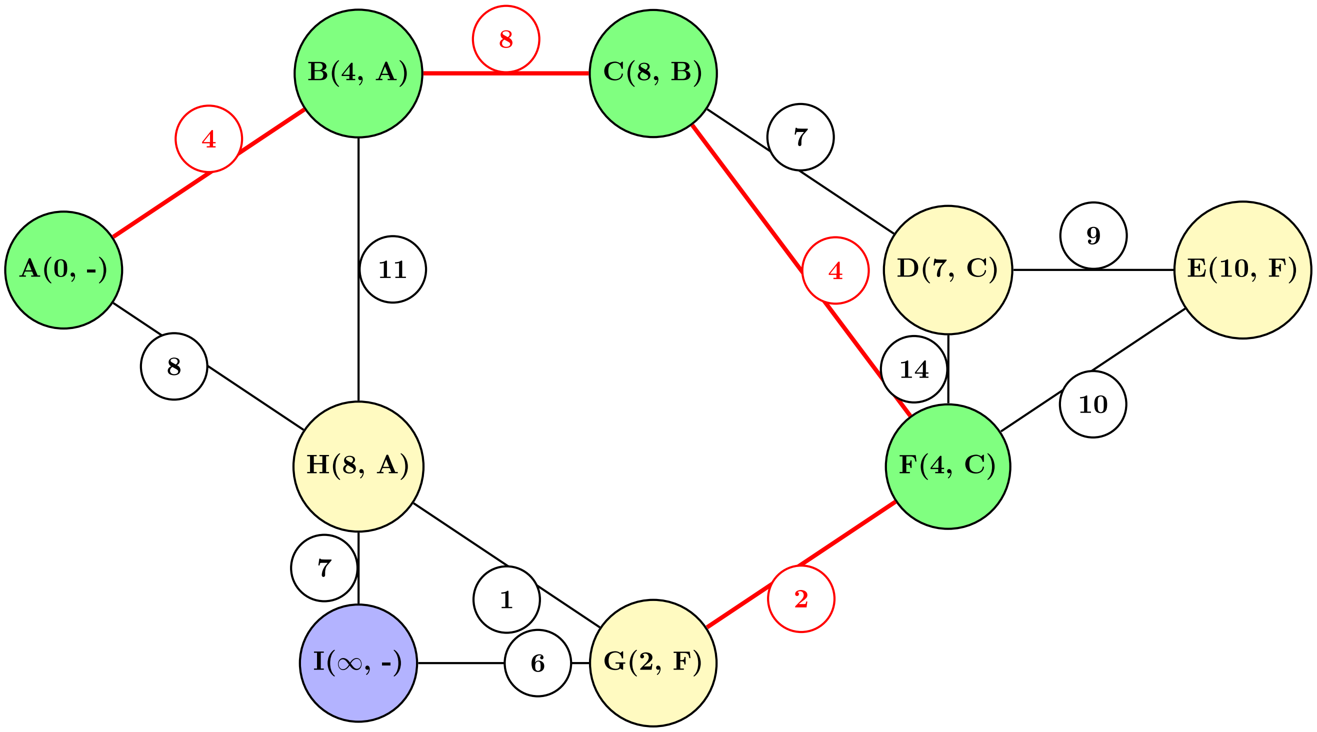

Figure 60: Step 5: Select F and Update G

Step 6: Select G and Update Neighbors

- Select G (smallest distance = 2 from F).
- Mark G as visited.
- Update neighbors H, distance becomes 1 since 1 <8, Previous = G.
- Update neighbors I, distance becomes 6, Previous = G.

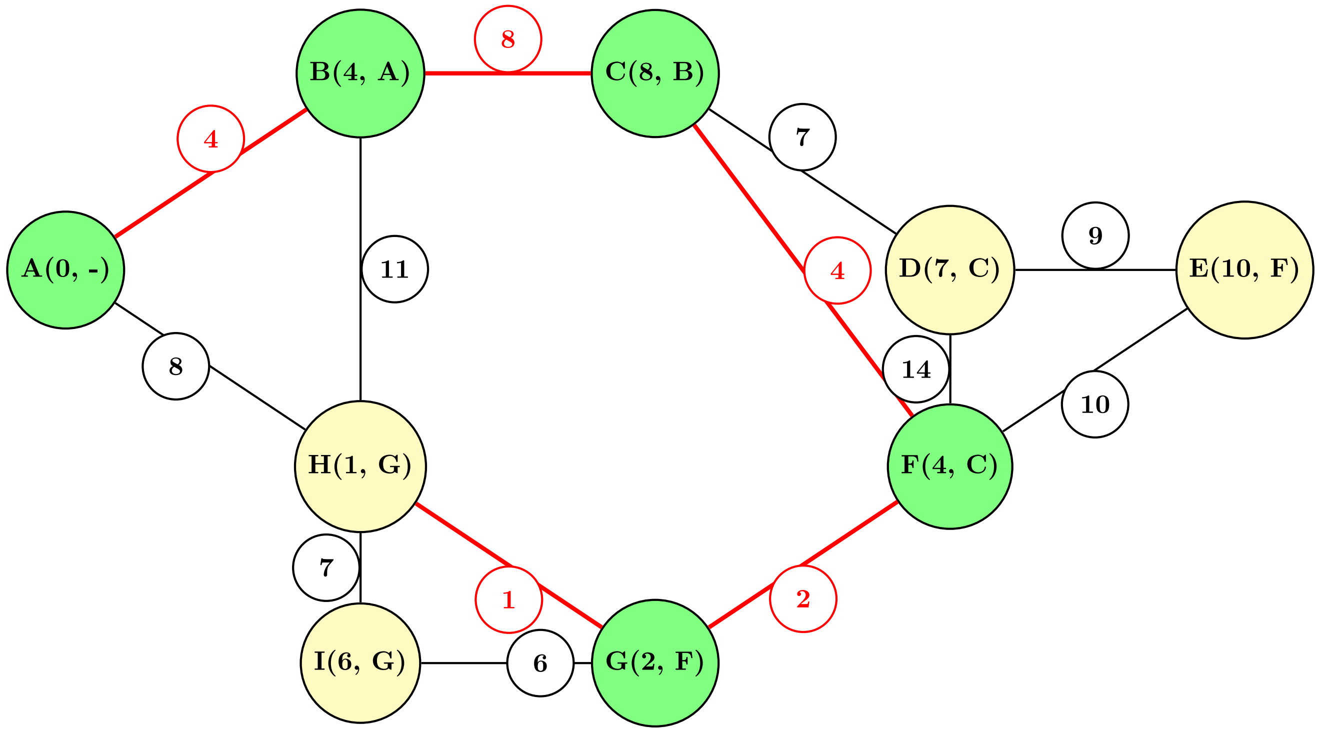

Figure 61: Step 6: Select G and Update H and I

Step 7: Select H and Update Neighbors

- Select H (smallest distance = 1).
- Mark H as visited.
- No update to neighbors I, since $7 > 6$.

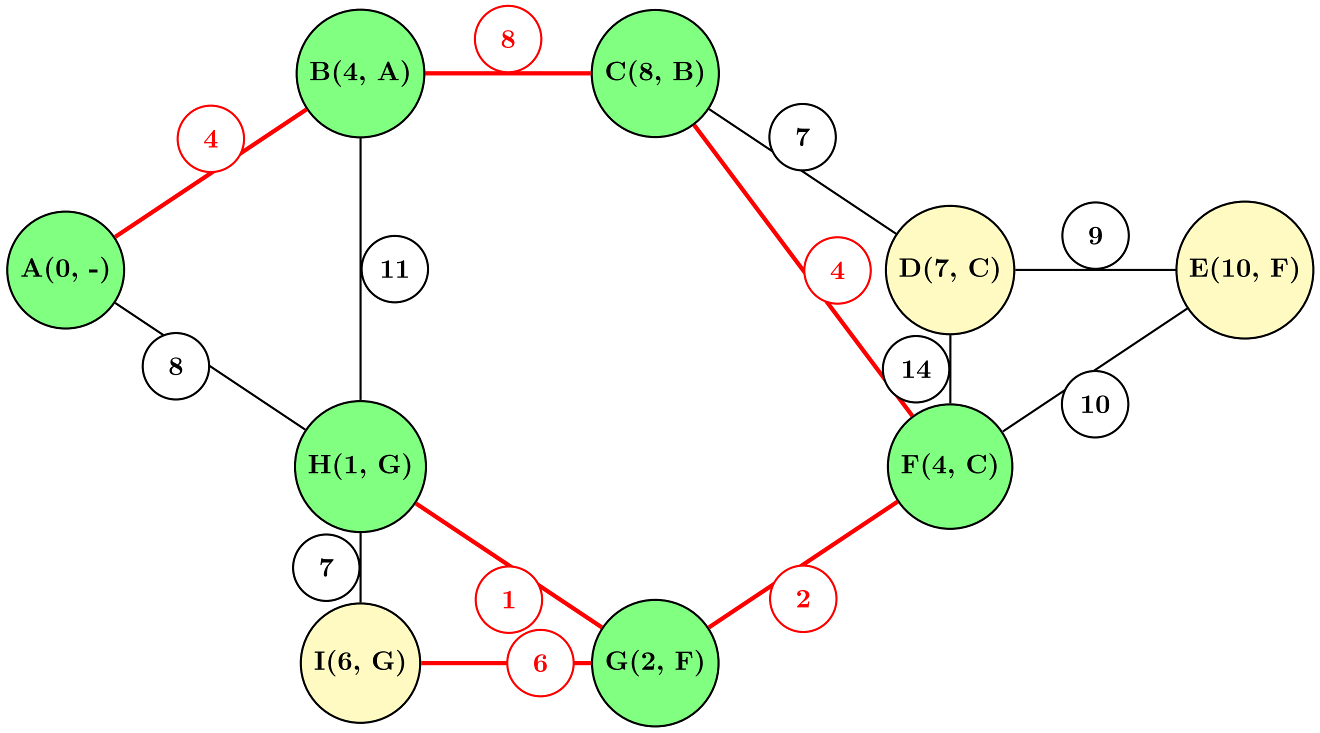

Figure 62: Step 7: Select H and Update Neighbors

Step 8: Select I and Update Neighbors

- Select I (smallest distance = 6).
- Mark I as visited.
- I has no neighbors with smaller weights than their current distances, so no updates are needed.

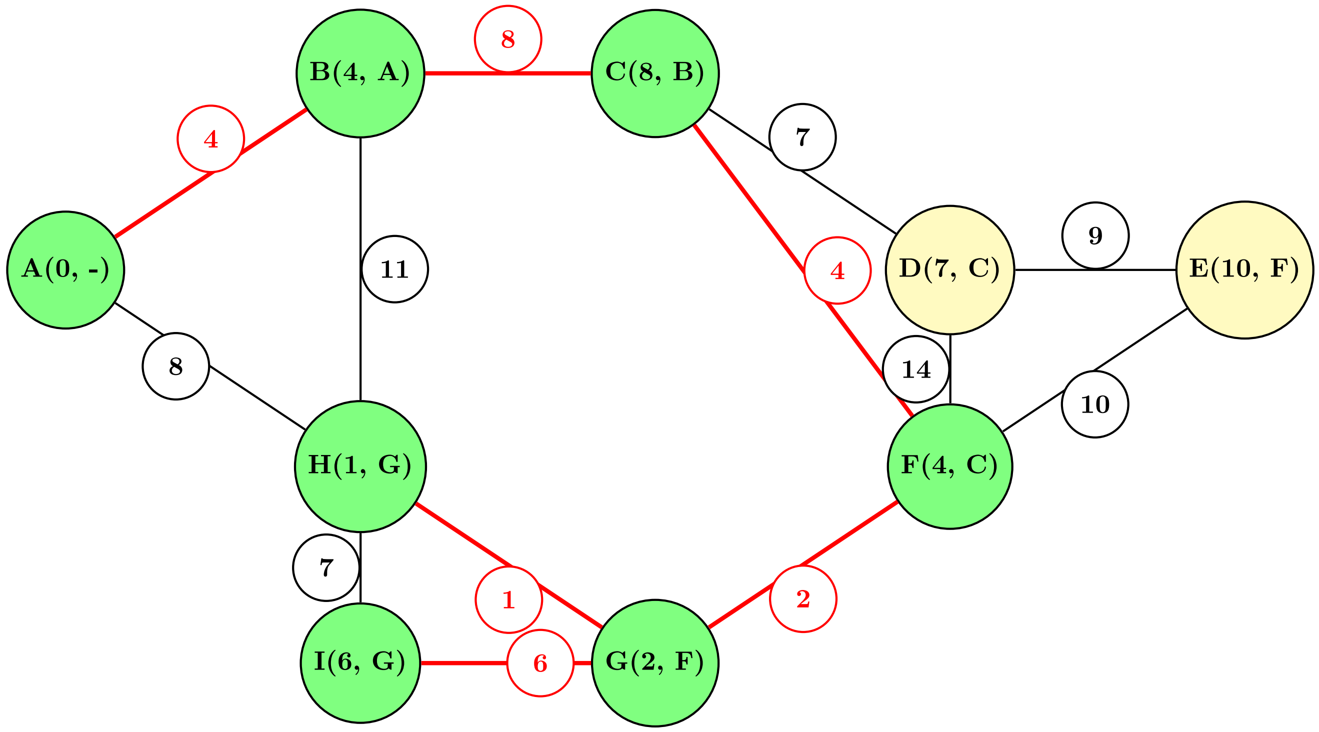

Figure 63: Step 8: Select I and Update Neighbors

Step 9: Select D and Update Neighbors

- Select D (smallest distance = 7).
- Mark D as visited.
- Update neighbors E, distance becomes 10 > 9, Previous = D.

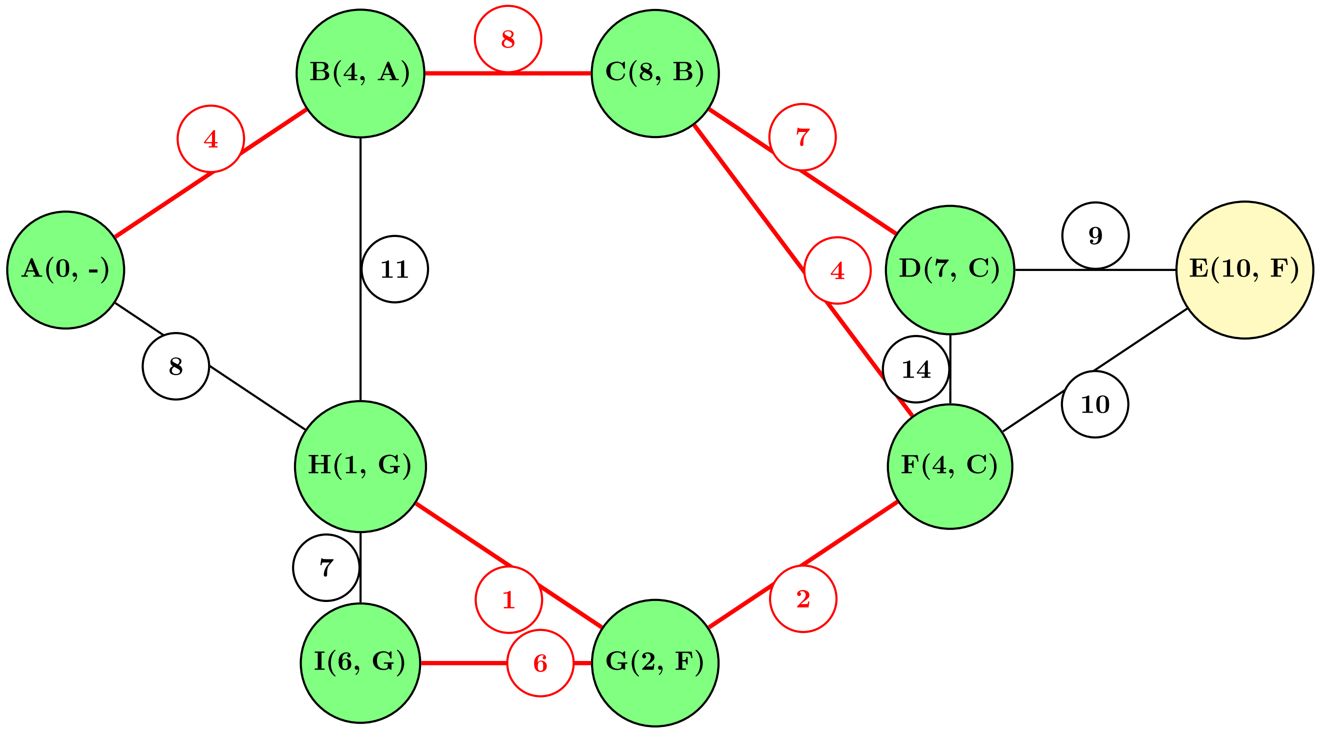

Figure 64: Step 9: Select D and Update E

Step 10: Select E

- Select E (smallest distance = 10).
- Mark E as visited.
- E has no unvisited neighbors, so no further updates are required.

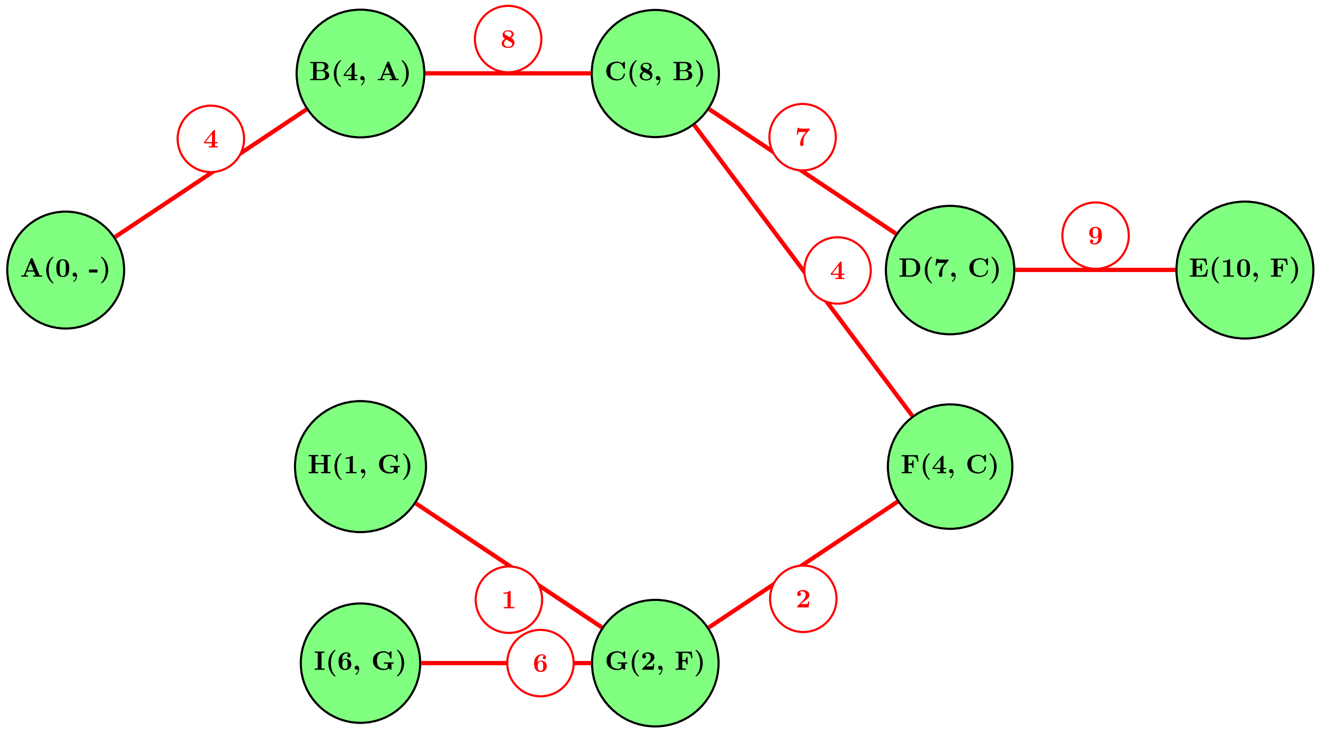

Figure 65: Step 10: Select E and Finalize MST

Final MST

- All vertices are now connected, and the MST is complete.
- The total weight is the sum of all selected edges:
- $4 + 8 + 4 + 2 + 1 + 6 + 7 + 9 = 41$

6.24 Solution of Kruskal's Algorithm

Kruskal's Algorithm step-by-step execution to find an MST

Step 1: Initialize the Graph

- Begin with the original graph, showing all vertices and their corresponding edges and weights.

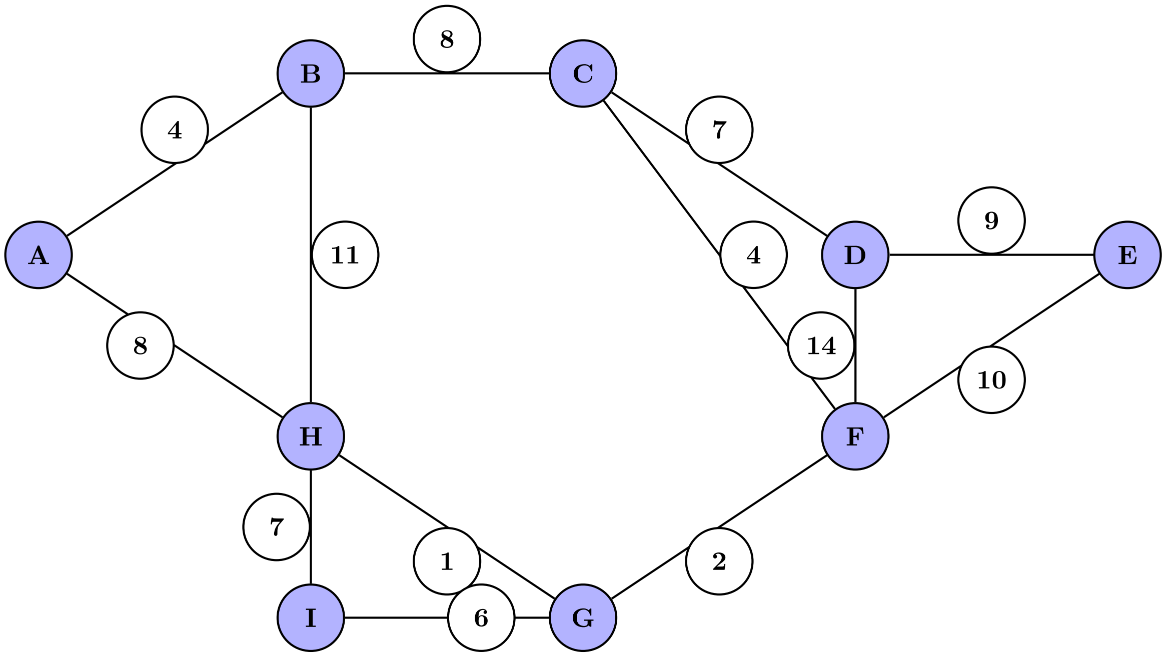

Figure 66: Step 1: Initial Graph with 9 Vertices and Weighted Edges

Step 2: Sort All Edges by Weight

- $G \leftrightarrow H$ (Weight 1)
- $F \leftrightarrow G$ (Weight 2)
- $A \leftrightarrow B$ (Weight 4)
- $C \leftrightarrow F$ (Weight 4)
- $G \leftrightarrow I$ (Weight 6)
- $C \leftrightarrow D$ (Weight 7)
- $H \leftrightarrow I$ (Weight 7)
- $A \leftrightarrow H$ (Weight 8)
- $B \leftrightarrow C$ (Weight 8)
- $D \leftrightarrow E$ (Weight 9)
- $E \leftrightarrow F$ (Weight 10)
- $B \leftrightarrow H$ (Weight 11)
- $D \leftrightarrow F$ (Weight 14)

Step 3: Add Edge $G \leftrightarrow H$ (Weight 1)

- Since G and H are not connected yet, so add this edge.

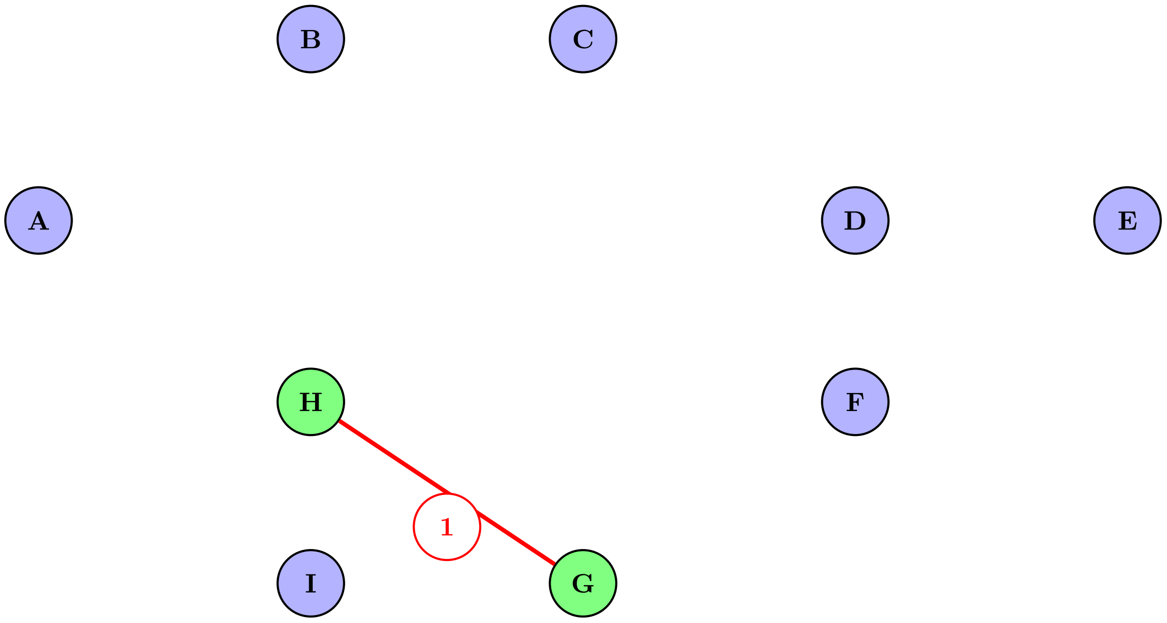

Figure 67: Step 3: Add Edge $G \leftrightarrow H$ (Weight 1)

Step 4: Add Edge $F \leftrightarrow G$ (Weight 2)

- Since F and G are not yet connected, so add this edge.

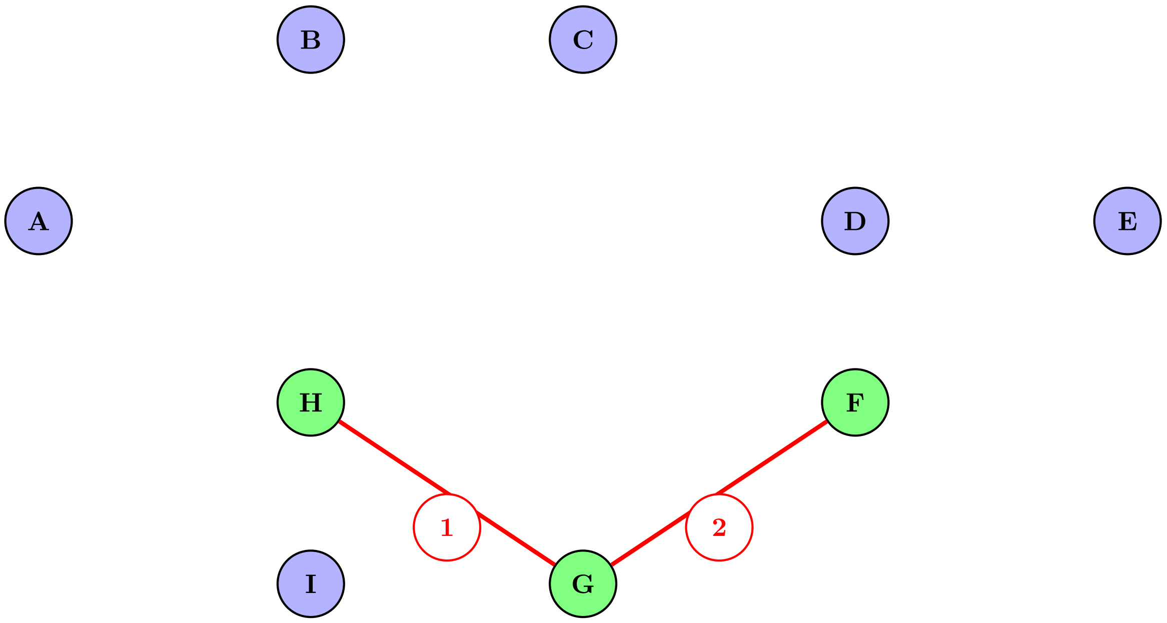

Figure 68: Step 4: Add Edge $F \leftrightarrow G$ (Weight 2)

Step 5: Add Edge $A \leftrightarrow B$ (Weight 4)

- Since A and B are not connected yet, so add this edge.

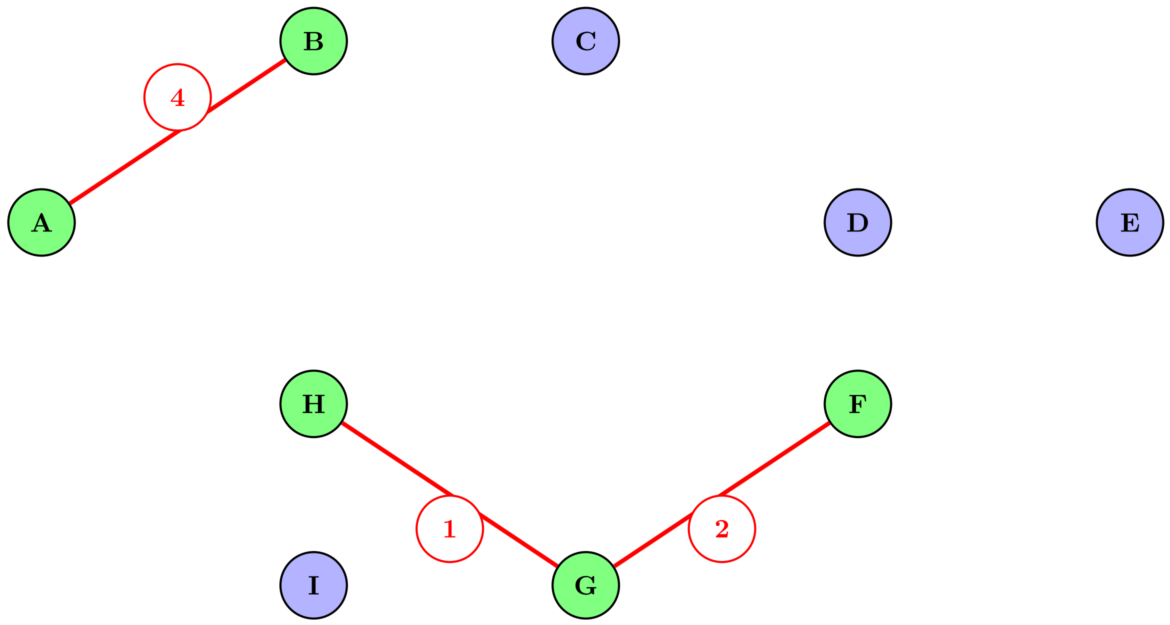

Figure 69: Step 5: Add Edge $A \leftrightarrow B$ (Weight 4)

Step 6: Add Edge $C \leftrightarrow F$ (Weight 4)

- C and F are not yet connected, so add this edge.

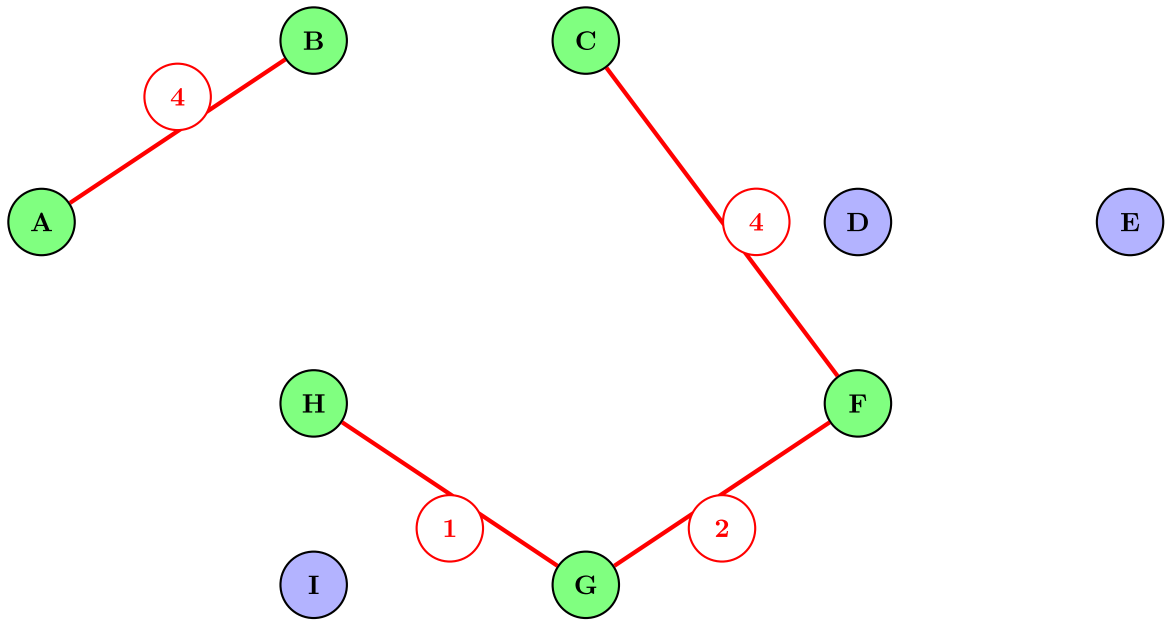

Figure 70: Step 6: Add Edge $C \leftrightarrow F$ (Weight 4)

Step 7: Add Edge $G \leftrightarrow I$ (Weight 6)

- G and I are not yet connected, so add this edge

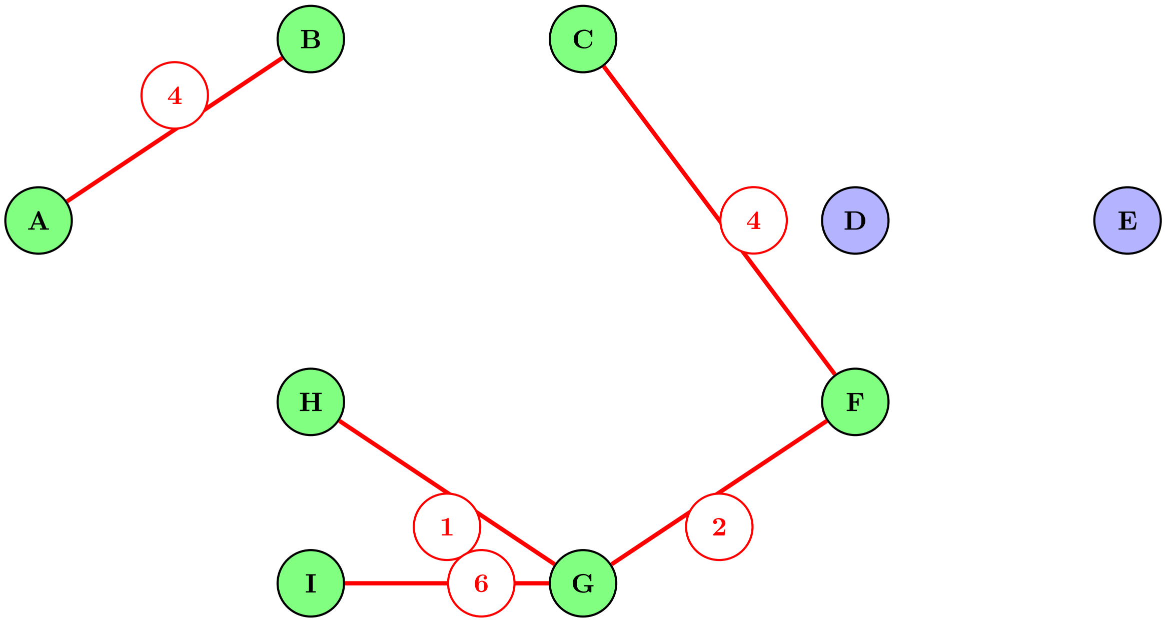

Figure 71: Step 7: Add Edge $G \leftrightarrow I$ (Weight 6)

Step 8: Add Edge $C \leftrightarrow D$ (Weight 7)

- C and D are not yet connected, so we add this edge

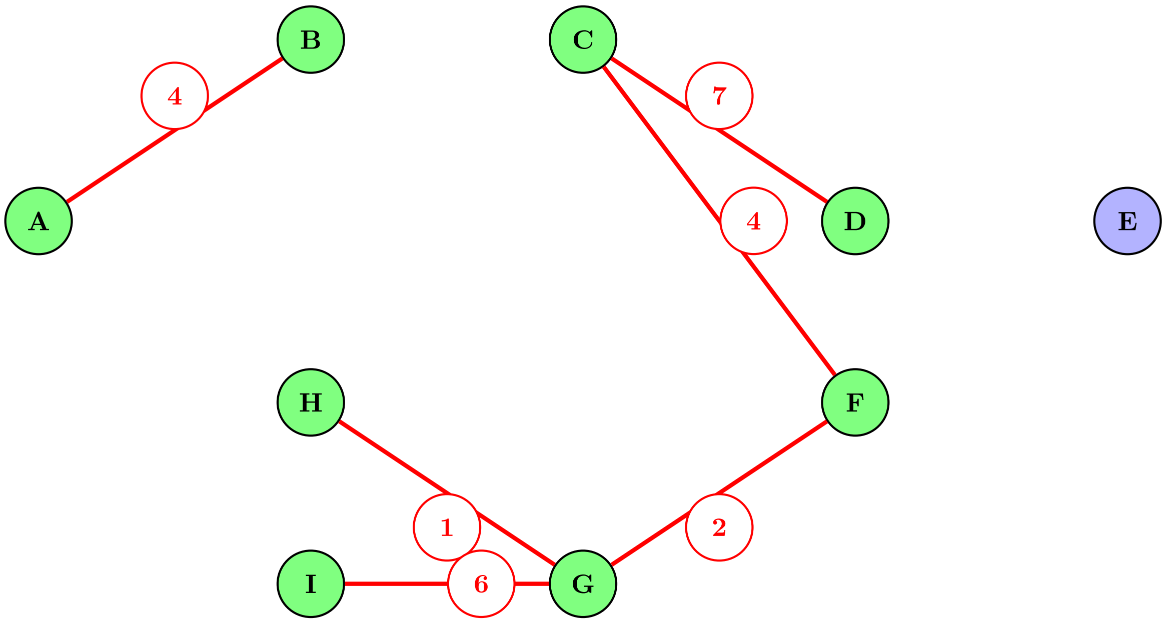

Figure 72: Step 8: Add Edge $C \leftrightarrow D$ (Weight 7)

Step 9: Add Edge $B \leftrightarrow C$ (Weight 8)

- B and C are not yet connected, so add this edge

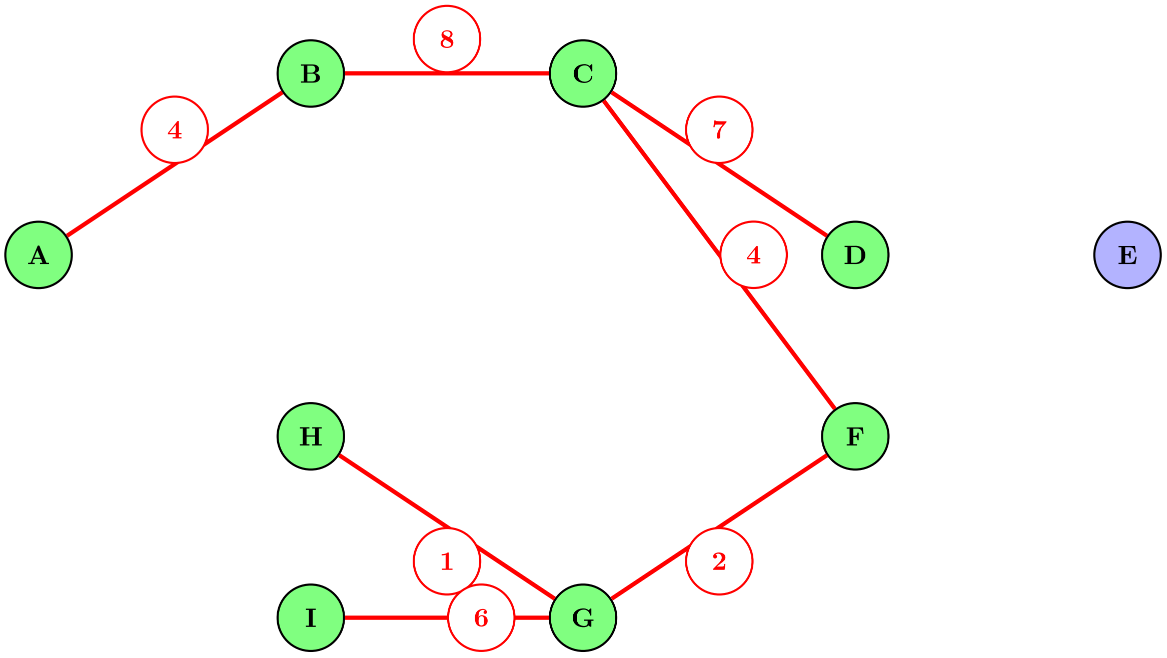

Figure 73: Step 9: Add Edge $B \leftrightarrow C$ (Weight 8)

Step 10: Add Edge $D \leftrightarrow E$ (Weight 9)

- Finally, add the edge $D \leftrightarrow E$ (Weight 9) to complete the MST

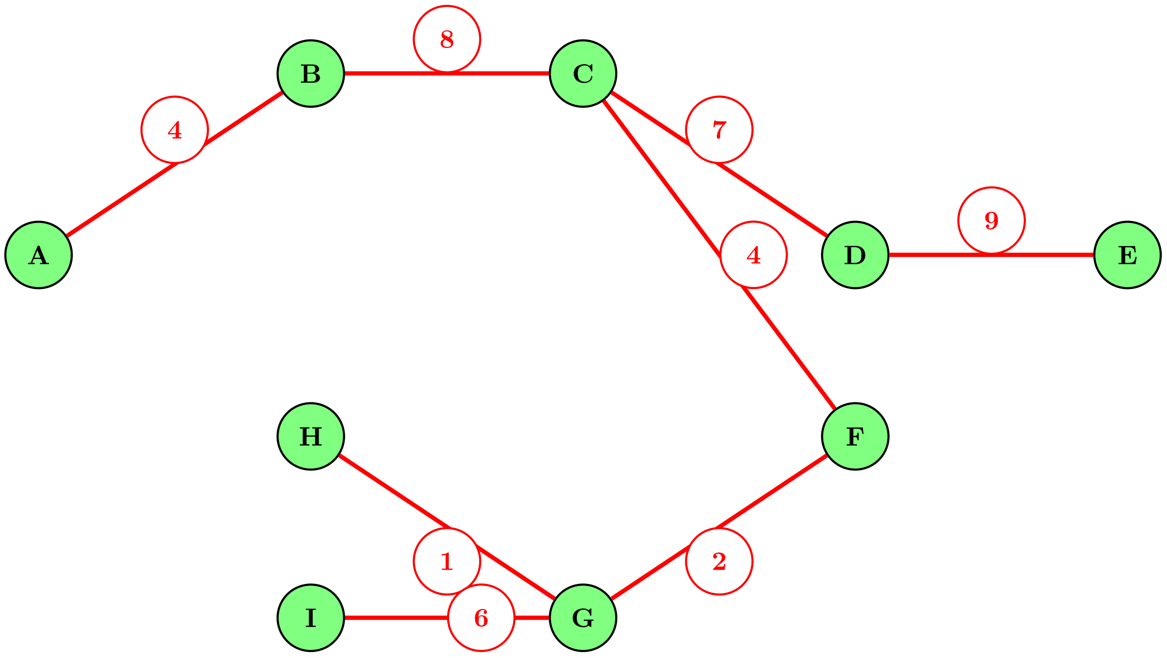

Figure 74: Step 10: Add Edge $D \leftrightarrow E$ (Weight 9)

Step 11: Final MST

- The final MST with a total weight of:

- $1 + 2 + 4 + 4 + 6 + 7 + 8 + 9 = 41$