DAT600: Algorithm Theory - Assignment 3

Antón Maestre Gómez - Daniel Linfon Ye Liu

Task 1 Problem Basic Graph:

a) Adjacency Matrix

- Converting Adjacency Matrix to Adjacency List:

We are provided with an adjacency matrix that describes the relationships between six nodes. The goal is to convert this adjacency matrix into an adjacency list, where the lists of adjacent nodes should be in alphabetical order.

The given adjacency matrix is as follows:

```
Adj = [[0 , 1 , 0 , 0 , 0 , 0],  # Node A
       [0 , 1 , 1 , 1 , 0 , 0],  # Node B
       [1 , 1 , 0 , 0 , 1 , 0],  # Node C
       [0 , 0 , 0 , 0 , 1 , 1],  # Node D
       [0 , 0 , 1 , 1 , 0 , 0],  # Node E
       [0 , 0 , 0 , 1 , 0 , 0]]  # Node F
```

Each row in the matrix corresponds to a node in the graph, and each column in that row indicates whether the node is adjacent to another node. A value of `1` means there is an edge between the nodes, while `0` means no connection between them.

To convert the adjacency matrix into an adjacency list, we need to examine each row of the matrix and extract the neighboring nodes for each node.

1. For **Node A (1)**, we look at row 1. There is only a `1` in column 2, meaning node A is connected to node B.

   A → B

2. For **Node B (2)**, we look at row 2. The `1`s in columns 2, 3, and 4 indicate that node B is connected to nodes B, C, and D.

   B → B, C, D

3. For **Node C (3)**, we look at row 3. The 1s in columns 1, 2, and 5 indicate that node C is connected to nodes A, B, and E.

   C → A, B, E

4. For **Node D (4)**, we look at row 4. The 1s in columns 5 and 6 indicate that node D is connected to nodes E and F.

   D → E, F

5. For **Node E (5)**, we look at row 5. The 1s in columns 3 and 4 indicate that node E is connected to nodes C and D.

   E → C, D

6. For **Node F (6)**, we look at row 6. The 1 in column 4 indicates that node F is connected to node D.

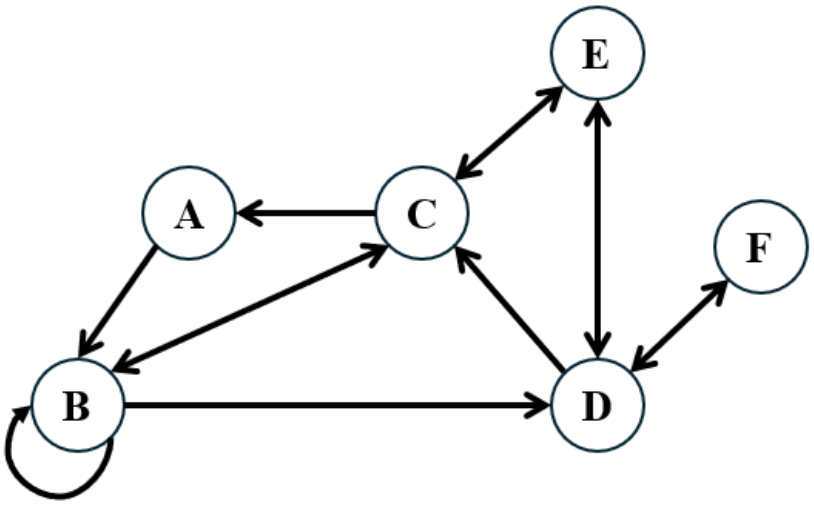
   F → D

Now that we have identified all the neighboring nodes, we organize this information into an adjacency list:

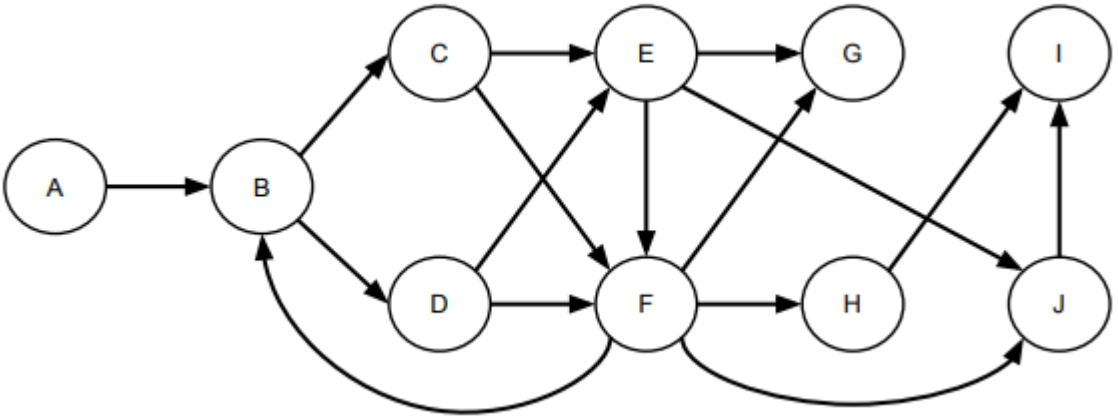
```
adj_list = {
    'A': ['B'],
    'B': ['B', 'C', 'D'],
    'C': ['A', 'B', 'E'],
    'D': ['E', 'F'],
    'E': ['C', 'D'],
    'F': ['D']
    }
```

- Drawing the Adjacency Matrix in graph form:

This graph is **directed** because the connections (edges) between the vertices are not necessarily reciprocal. For example, while there is a directed edge from vertex `C` to vertex `A`, there is no directed edge from `A` to `C`. This asymmetry in the connections is characteristic of a directed graph.



- Adjacent List of Figure 1:



Looking at the nodes and edges of the graph in Figure 1, the adjacency list is obtained:

```
adj_list = {
    'A': ['B'],
    'B': ['C', 'D'],
    'C': ['E', 'F'],
    'D': ['E', 'F'],
    'E': ['F', 'G', 'J'],
    'F': ['B', 'G', 'H', 'J'],
    'G': [],
    'H': ['I'],
    'I': [],
    'J': ['I']
}
```

We can appreciate that this graph has two terminal nodes (sink nodes): G and I. This type of node has **incoming edges** (other nodes point to it) but **no outgoing edges** (it does not point to any other node).

b) Depth-First Search (DFS) vs. Breadth-First Search (BFS)

DFS and BFS are two fundamental graph traversal algorithms used to explore nodes and edges in a graph. Both algorithms have been executed manually as well as in Python code. The file is included among the GitHub files.

- **Depth-First Search (DFS):**

DFS explores as far as possible along each branch before backtracking. It follows a **recursive or stack-based** approach.

How DFS Works:

1. Start at the root (or any arbitrary node).
2. Mark the node as visited.
3. Recursively visit all its unvisited neighbors in **alphabetical order** (if required).
4. Backtrack when no unvisited neighbors remain.
5. Continue until all nodes are visited.

Applying it to the graph in Figure 1, the procedure would be as follows:

Visit `A` → `Start(A) = 1`

Visit `B` → `Start(B) = 2`

Visit `C` → `Start(C) = 3`

Visit `E` → `Start(E) = 4`

Visit `F` → `Start(F) = 5`

Visit `B` (already visited, backtrack)

Visit `G` → `Start(G) = 6`, `Finish(G) = 7` (no children, finish)

Visit `H` → `Start(H) = 8`

Visit `I` → `Start(I) = 9`, `Finish(I) = 10` (no children, finish)

Finish `H` → `Finish(H) = 11`

Visit `J` → `Start(J) = 12`

Visit `I` (already visited, backtrack)

Finish J → `Finish(J) = 13`

Finish F → `Finish(F) = 14`

Visit G (already visited, backtrack)

Finish E → `Finish(E) = 15`

Finish C → `Finish(C) = 16`

Visit D → `Start(D) = 17`

Visit E (already visited, backtrack)

Visit F (already visited, backtrack)

Finish D → `Finish(D) = 18`

Finish B → `Finish(B) = 19`

Finish A → `Finish(A) = 20`

DFS Order (Preorder Traversal)

`A → B → C → E → F → G → H → I → J → D`

DFS Start & Finish Times

In Depth-First Search (DFS), we keep track of two timestamps for each node:

1. **Start Time** → When we **first visit** the node.

2. **Finish Time** → When we **have finished exploring** all its neighbors (i.e., when we backtrack).

Node	Start Time	Finish Time
A	1	20
B	2	19
C	3	16
D	17	18
E	4	15
F	5	14
G	6	7
H	8	11
I	9	10
J	12	13

- **Breadth-First Search (BFS):**

BFS explores all neighbors at the present depth before moving deeper. It follows a **queue-based** approach.

How BFS Works:

1. Start at the root (or any arbitrary node).

2. Mark the node as visited and enqueue it.

3. Dequeue a node, visit its **unvisited** neighbors, and enqueue them.

4. Repeat until all nodes are visited.

Applying it to the graph in Figure 1, the procedure would be as follows:

Start at A, enqueue B → **Queue:** `[B]`

Visit B, enqueue C, D → **Queue:** `[C, D]`

Visit C, enqueue E, F → **Queue:** `[D, E, F]`

Visit D, enqueue E, F (already in queue) → **Queue:** `[E, F, E, F]`

Visit E, enqueue F, G, J → **Queue:** `[F, E, F, F, G, J]`

Visit F, enqueue B, G, H, J → **Queue:** `[E, F, F, G, J, B, G, H, J]`

Visit E (already visited), remove from queue → **Queue:** `[F, F, G, J, B, G, H, J]`

Visit F (already visited), remove from queue → **Queue:** `[F, G, J, B, G, H, J]`

Visit F (already visited), remove from queue → **Queue:** `[G, J, B, G, H, J]`

Visit G, (no children) → **Queue:** `[J, B, G, H, J]`

Visit J, enqueue I → **Queue:** `[B, G, H, J, I]`

Visit B (already visited), remove from queue → **Queue:** `[G, H, J, I]`

Visit G (already visited), remove from queue → **Queue:** `[H, J, I]`

Visit H, enqueue I → **Queue:** `[J, I, I]`

Visit J (already visited), remove from queue → **Queue:** `[I, I]`

Visit I, (no children) → **Queue:** `[I]`

Visit I (already visited), remove from queue → **Queue:** `[]`

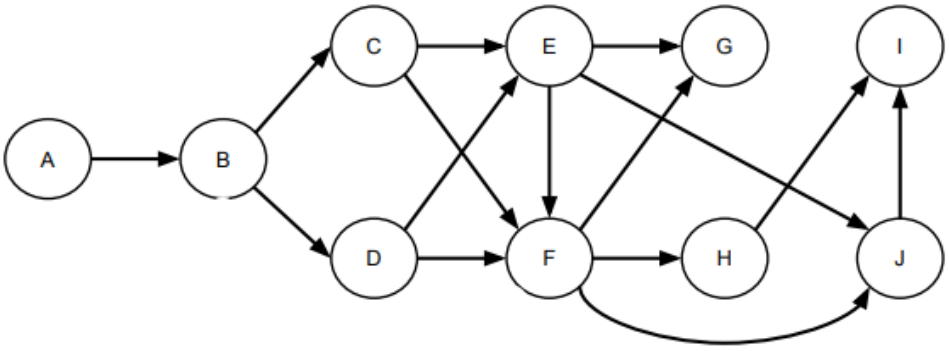
BFS Order

A → B → C → D → E → F → G → J → H → I

c) Directed Acyclic Graph (DAG)

To transform the graph into a Directed Acyclic Graph (DAG), we need to remove a single edge that creates a cycle. In this graph, we can see that there's a cycle between nodes `B`, `C`, `F`, and `G`. Specifically, the edge `F -> B` creates a cycle in the graph.

Removing the edge `F -> B`, we get the following TAG:

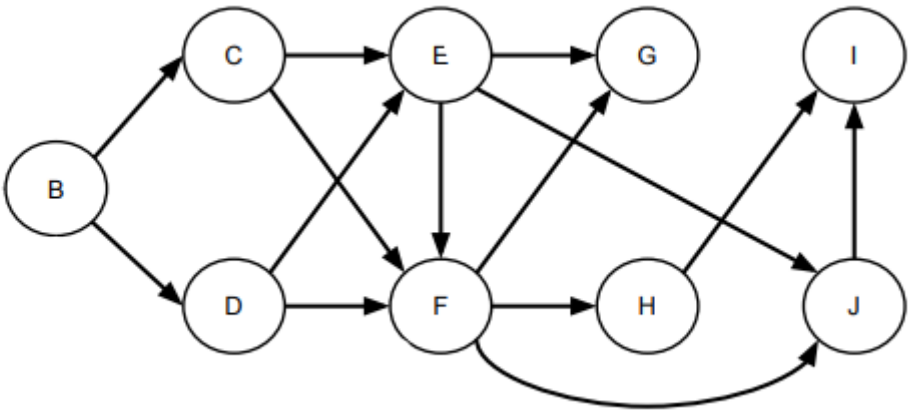


Topological Sort:

Initially, we start with nodes that have no incoming edges. These are the nodes that have no dependencies. In this case, we have the node A.

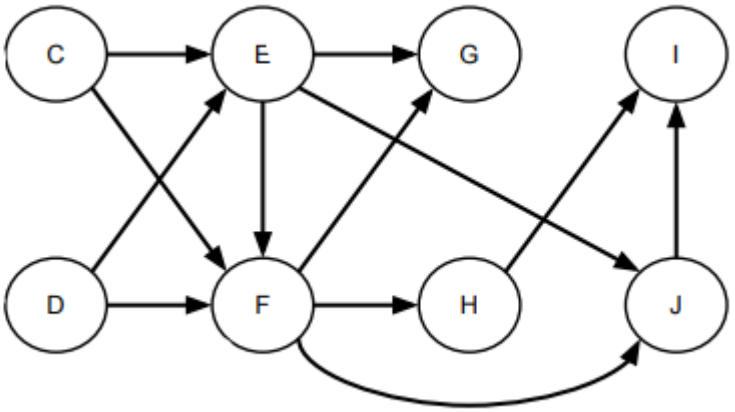
1. Process Node `A`

Process `A` and remove it along with its outgoing edge `A -> B`.



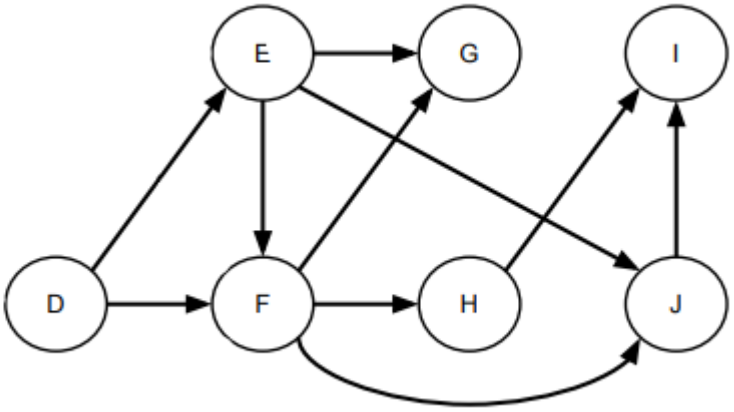
After removing `A`, the node `B` now has no incoming edges because `A -> B` is removed.

2. Process Node B

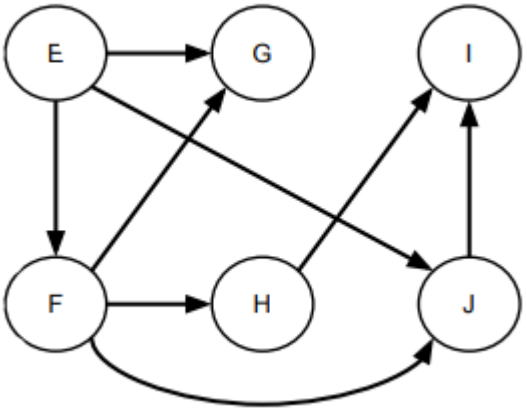


After removing B, the nodes C and D now have no incoming edges.

3. Process Node C

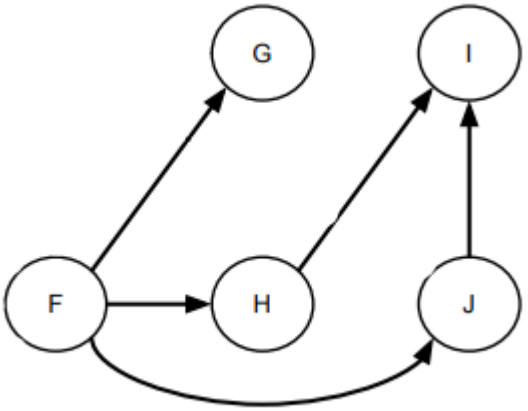


4. Process Node D



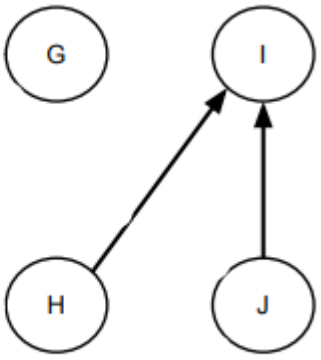
After removing `C` and `D`, the node `E` now has no incoming edges.

5. Process Node E



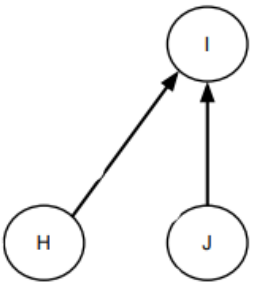
After removing `E`, the node `F` now has no incoming edges.

6. Process Node F



After removing `E`, the nodes `G, H, J` now have no incoming edges.

7. Process Node G



8. Process Node H

9. Process Node J

After removing `J`, the node `I` now has no incoming edges.

10. Process Node I

The graph is now fully processed.

Final Topological Order:

A, B, C, D, E, F, G, H, J, I

d) Algorithm Implementation to create DAGs

Given a directed graph represented as a set of **vertices** (nodes) and **edges** (directed connections), the goal is to:

1. Detect the presence of **cycles** in the graph.
2. Remove an edge from each cycle to convert it into a **DAG**.

Algorithm Implementation:

The implementation is done in Python using the **`Graph`** class, which allows for graph construction, cycle detection, and topological sorting.

1. The Graph Class:

- Initialization:

```
from collections import defaultdict, deque
```

```
class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices
```

- **`defaultdict(list)`** is used to store the graph's adjacency list.
- **`self.V`** stores the number of nodes.

- Add Edges function:

```
def add_edge(self, u, v):
    self.graph[u].append(v)
```

- The **`add_edge(u, v)`** method adds a directed edge from node u to node v in the adjacency list.

- Cycle Detection and Removal:

```
def detect_and_remove_cycles(self):
    def find_cycle():
        visited = set()
        rec_stack = set()
        back_edges = []

        def dfs(node, parent):
            visited.add(node)
            rec_stack.add(node)
            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    if dfs(neighbor, node):
                        return True
                elif neighbor in rec_stack:
                    back_edges.append((node, neighbor))
            rec_stack.remove(node)
            return False

        for node in list(self.graph.keys()):
            if node not in visited:
                dfs(node, None)
        return back_edges

    while True:
        back_edges = find_cycle()
        if not back_edges:
            break  # No more cycles, stop
        edge_to_remove = back_edges[0]  # Remove the first back edge found
        self.graph[edge_to_remove[0]].remove(edge_to_remove[1])
        print(f"Removed edge: {edge_to_remove}")
```

```
print("DAG after cycle removal:")
for node in self.graph:
    print(f"{node} -> {self.graph[node]}")
```

- **Detect cycles** using **DFS**.
- If a cycle is found, remove one of its back edges.
- Repeat until no cycles remain.
- Finally, print the resulting DAG.

2. Graph Creation and Execution:

```
# Create graph from Figure 1 with extra edges I → C and C → A
graph = Graph(10)  # 10 nodes

edges = [
    ('A', 'B'), ('B', 'C'), ('B', 'D'), ('C', 'E'), ('C', 'F'), ('D', 'E'),
    ('D', 'F'), ('E', 'F'), ('E', 'G'), ('E', 'J'), ('F', 'B'), ('F', 'G'),
    ('F', 'H'), ('F', 'J'), ('H', 'I'), ('J', 'I'), ('I', 'C'), ('C', 'A')
]

for u, v in edges:
    graph.add_edge(u, v)

# Remove cycles
graph.detect_and_remove_cycles()
```

- The graph is initialized with 10 nodes.
- The given edges are added to represent the input graph.
- Cycle detection and removal are executed until a DAG is obtained.
- The DAG is printed.

3. Results:

```
Removed edge: ('F', 'B')
Removed edge: ('I', 'C')
Removed edge: ('C', 'A')
DAG after cycle removal:
A -> ['B']
B -> ['C', 'D']
C -> ['E', 'F']
D -> ['E', 'F']
E -> ['F', 'G', 'J']
F -> ['G', 'H', 'J']
H -> ['I']
J -> ['I']
I -> []
G -> []
```

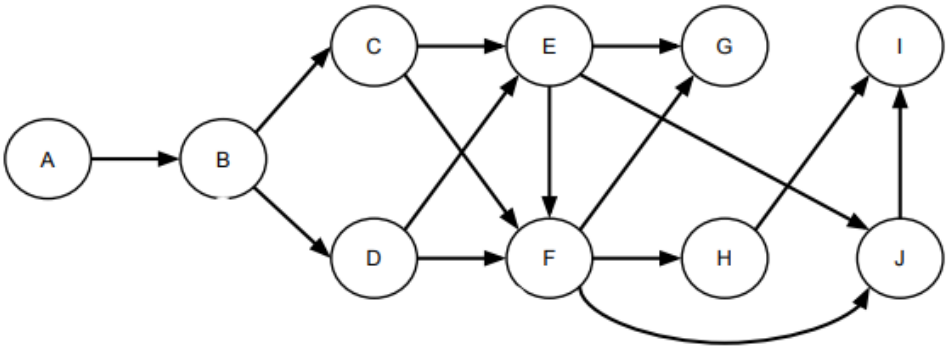
After executing the cycle detection and removal algorithm, the following edges were removed to transform the graph into a **Directed Acyclic Graph (DAG):**

1. **Removed edge (`F` → `B`)**: This edge was part of a cycle that involved `B` and `F`. Removing it helped break one of the cyclic dependencies.

2. **Removed edge (`I` → `C`)**: This edge contributed to a cycle involving `C` and `I`. Eliminating it helped further reduce cyclic dependencies.

3. **Removed edge (`C` → `A`)**: This was the final edge removed to ensure the graph was completely acyclic.

The final result is a DAG:

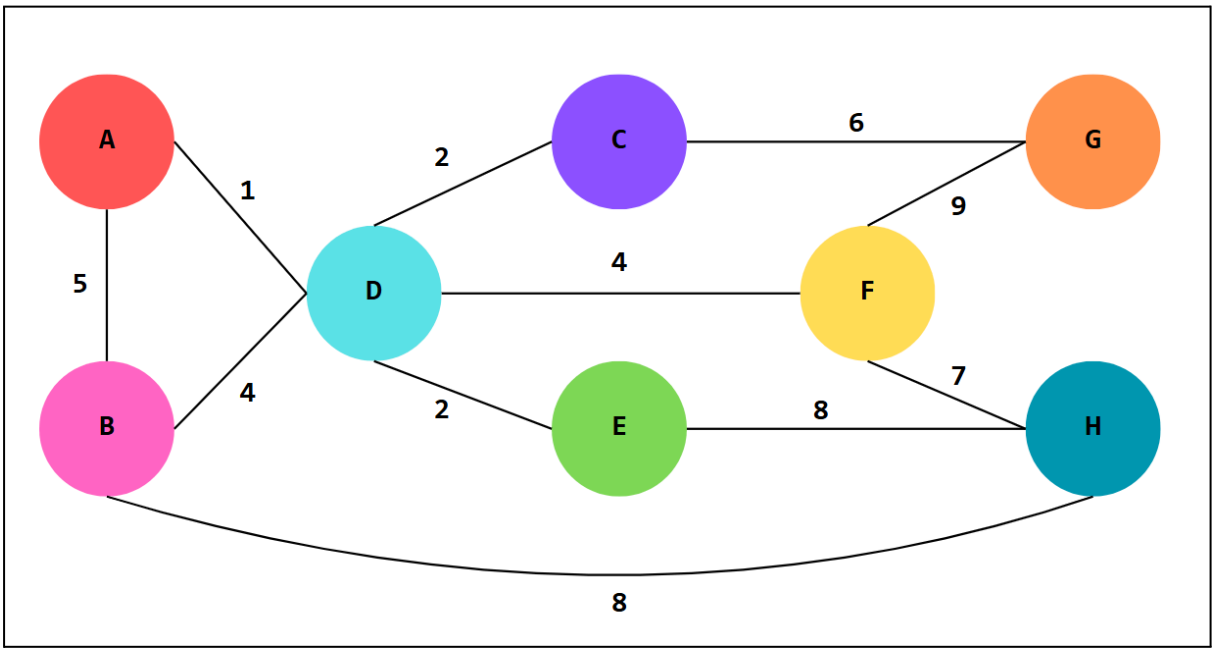


Task 2 Problem Cable Network

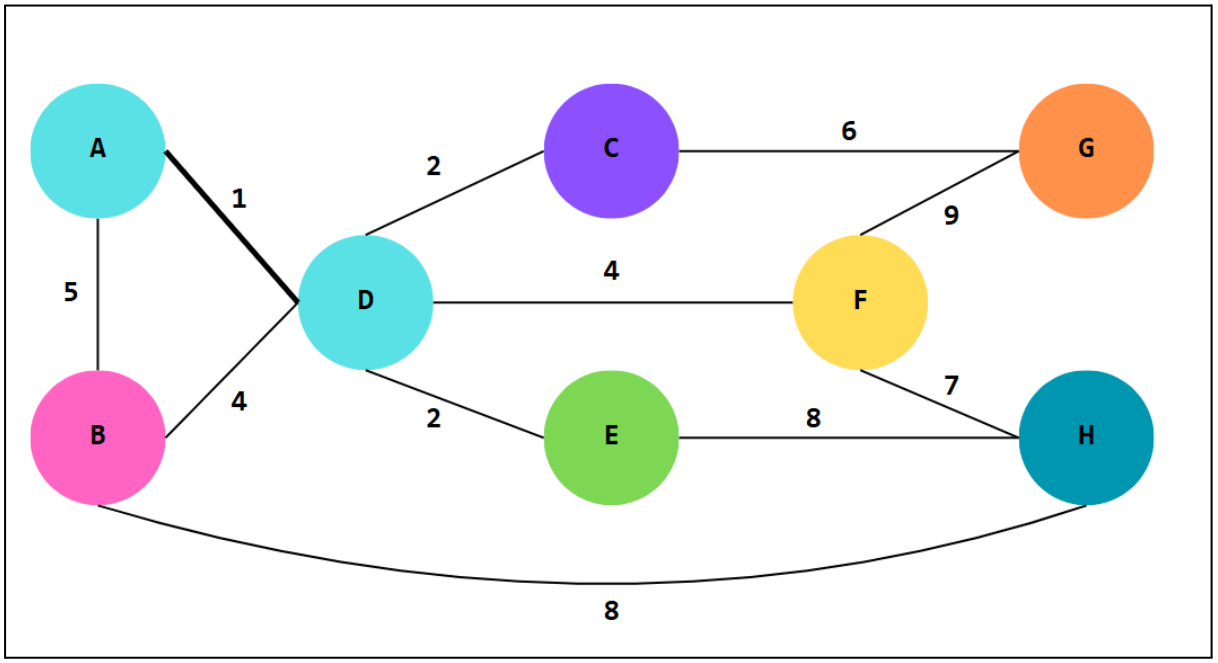
a) Is it possible to connect all nodes within the budget $b = 30$?

To determine if we can connect all nodes within the budget, we apply Kruskal's algorithm.

Initial

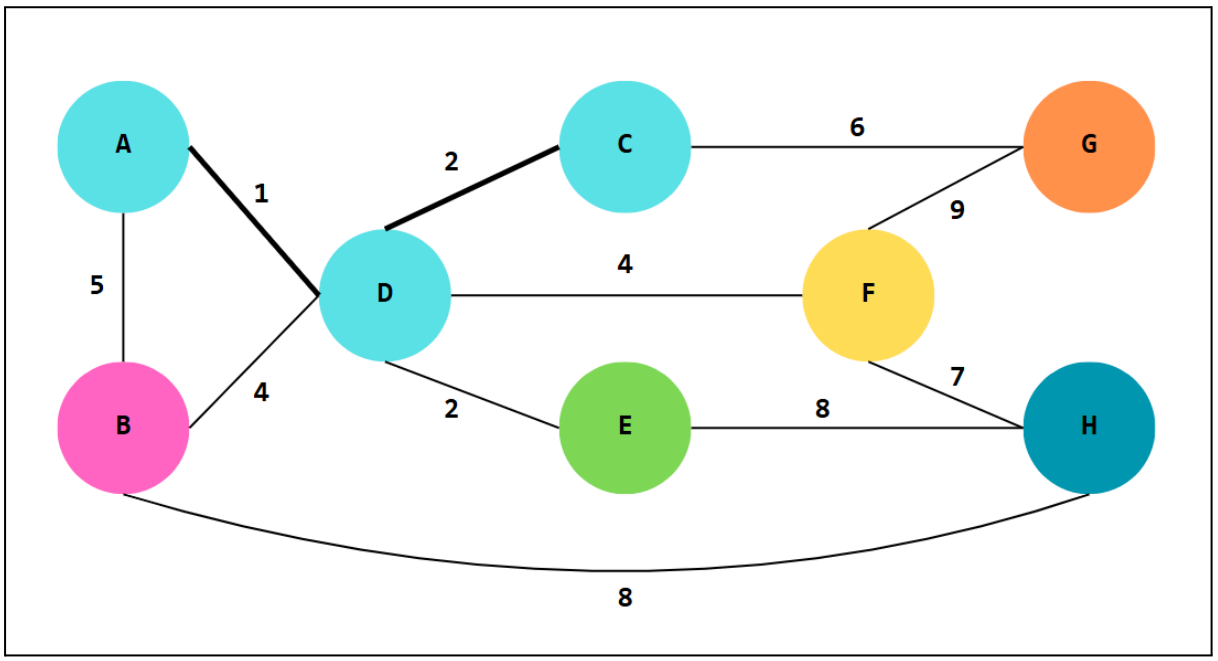


Iteration 1



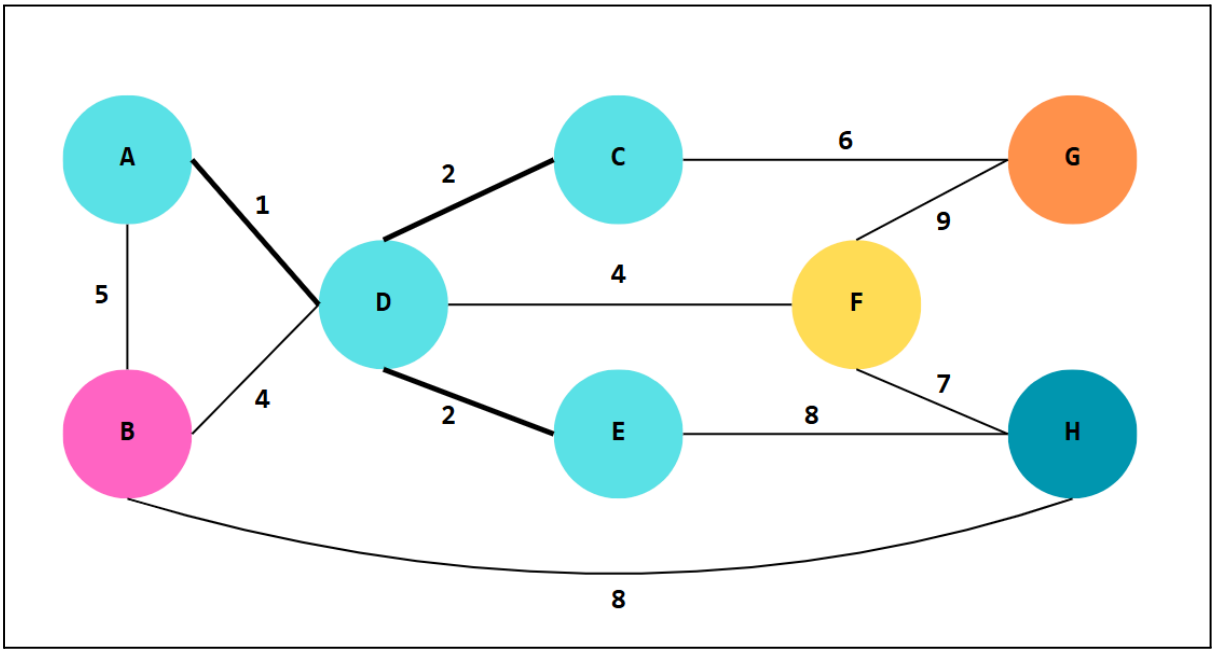
A → D: cost 1

Iteration 2



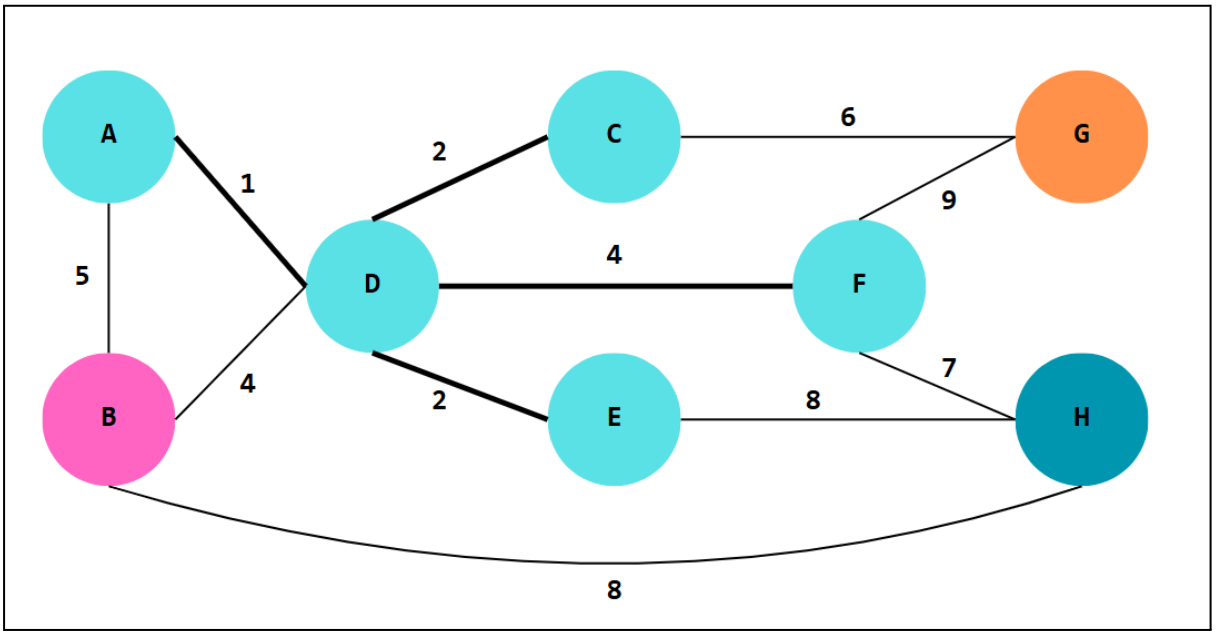
C → D: cost 2

Iteration 3



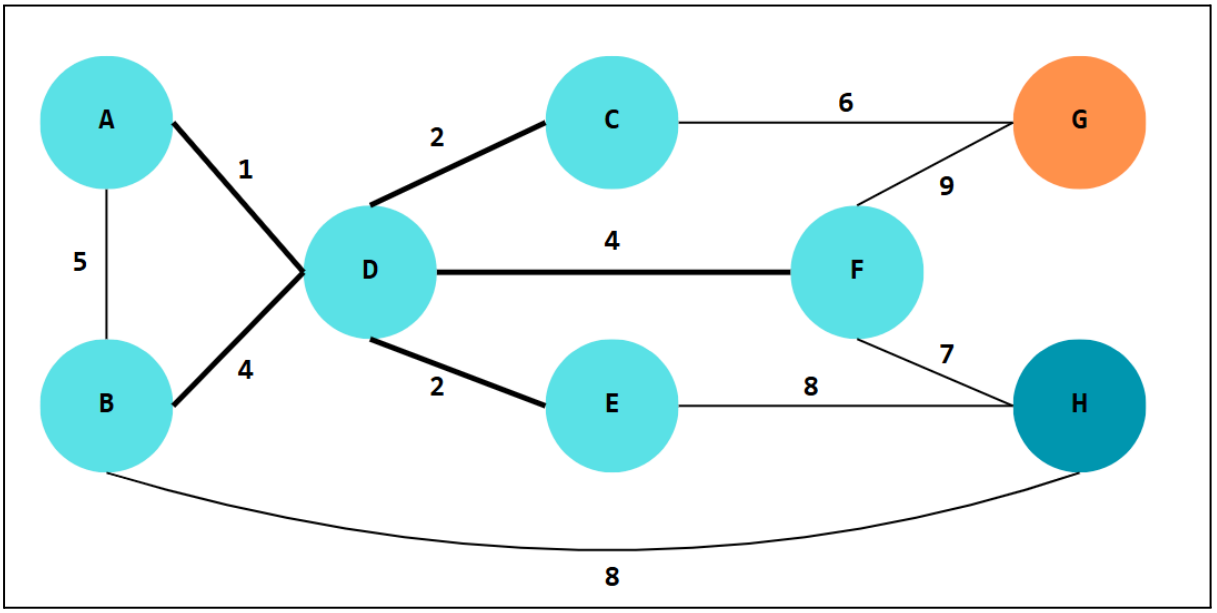
D → E: cost 2

Iteration 4



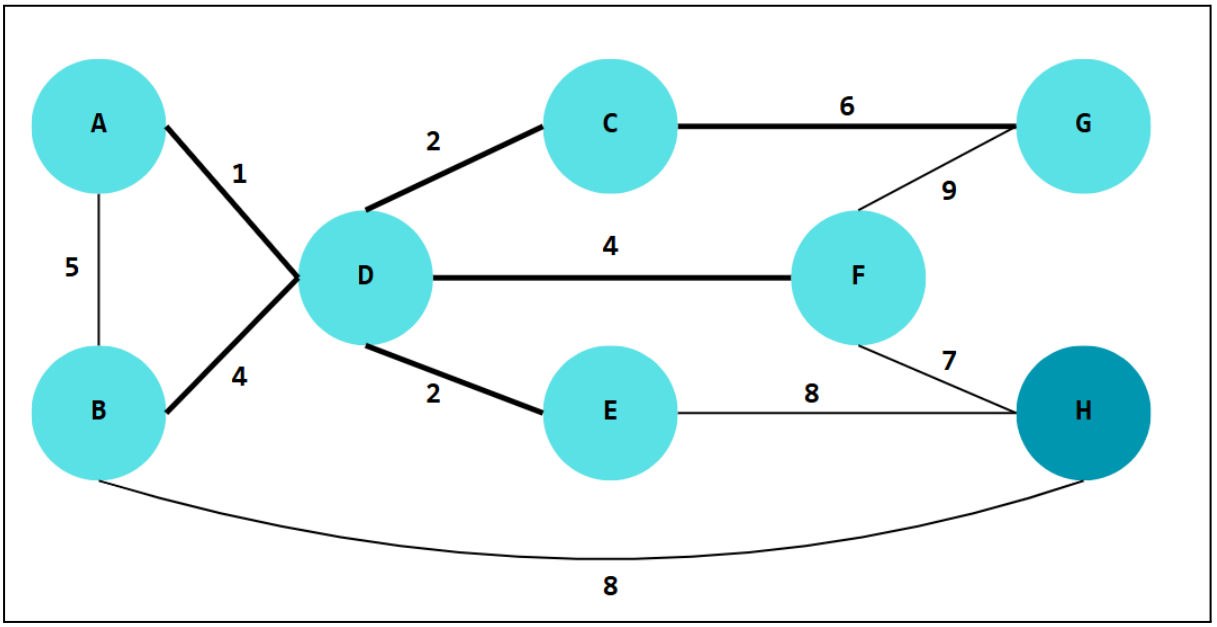
D → F: cost 4

Iteration 5



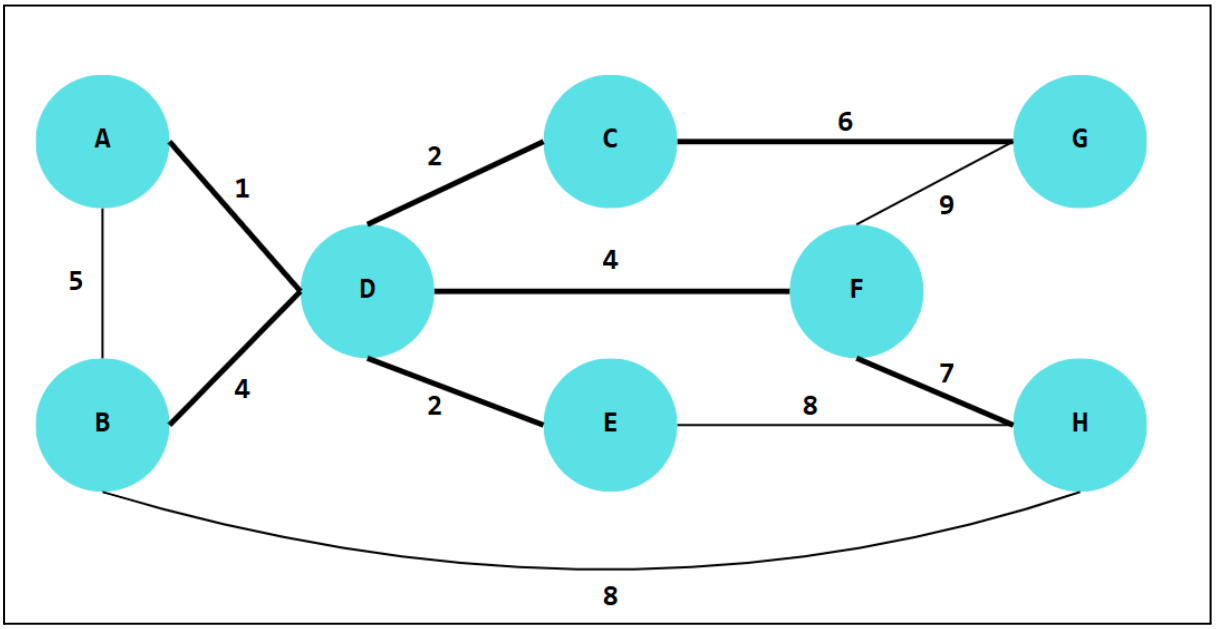
B→ D: cost 4

Iteration 6



C → G: cost 6

Iteration 7



F → H: cost 7

MST cost

We add the weights of the edges in the MST:

1 + 2 + 2 + 4 + 4 + 6 + 7 = **26 < b**

The total cost of MST (26) is less than the budget $b = 30$, so it is possible to connect all nodes within the budget.

b) Restriction: D can only have 3 connections

In the current MST, D has 5 connections, which exceeds the restriction. D's current connections are:

(A, D) = 1

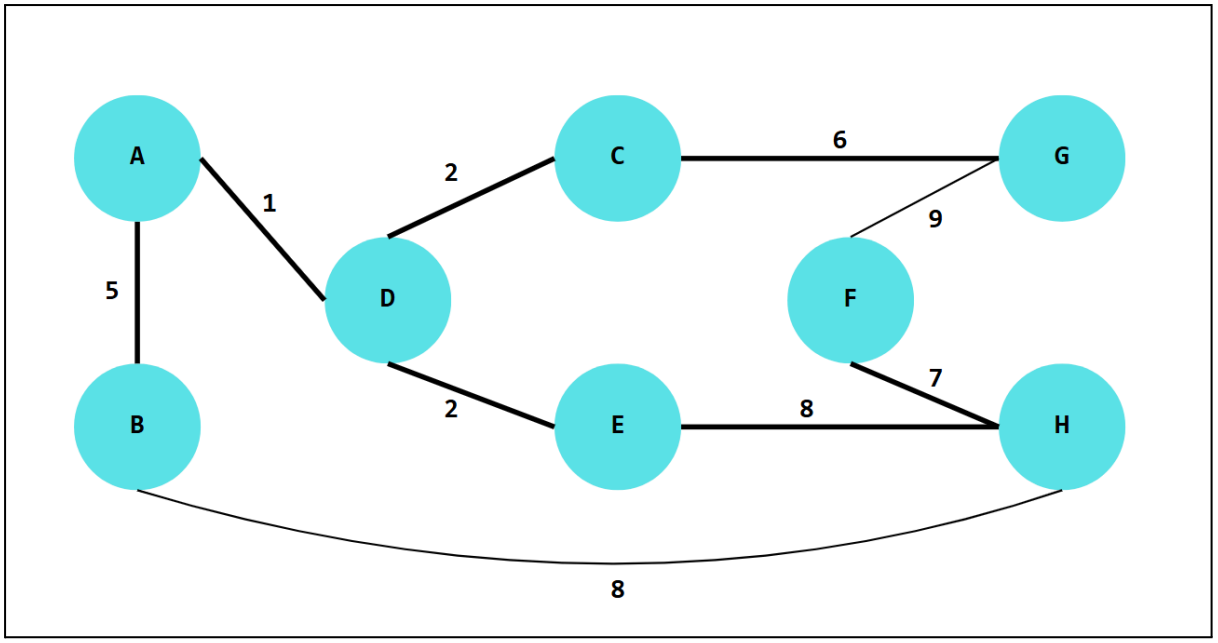
(C, D) = 2

(D, E) = 2

(D, F) = 4

(B, D) = 4

We eliminated the 2 most expensive connections from D and the graph would look like this after applying Kruskal:



We add the weights of the edges in the MST:

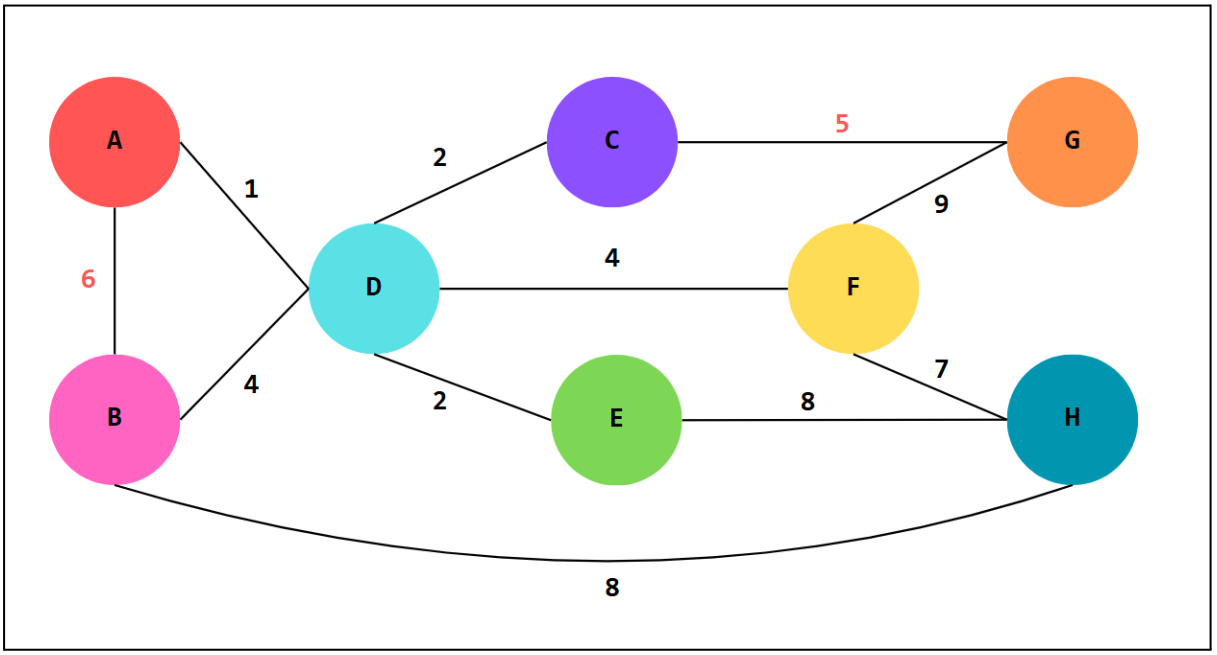
1 + 2 + 2 + 5 + 6 + 7 + 8 = **31 > b**

If we impose the restriction of maximum 3 connections in D, the cost of MST increases to 31, which exceeds the budget, so it is not possible to meet both conditions simultaneously.

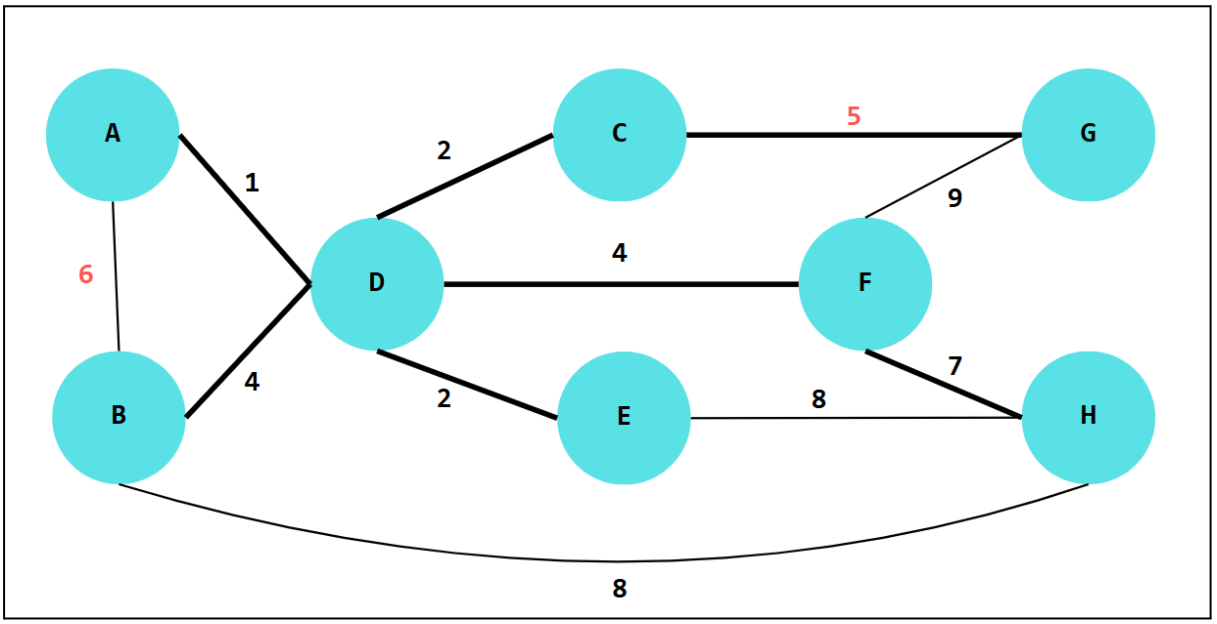
The solution is not always globally optimal, since modifying the MST to meet the constraint in D can lead to higher costs. Suppose that in another graph, removing connections from a node requires using more expensive edges. This demonstrates that the constraint in D does not guarantee the most globally optimal solution.

c) Adjust the MST to comply with $b' = 25$

We change **A** → **B** with edge **C** → **G**:



After applying Kruskal's algorithm:



MST cost

We add the weights of the edges in the MST:

$1 + 2 + 2 + 4 + 4 + 5 + 7 = \mathbf{25} = b'$

The total cost of MST meets the new budget $b' = 25$, so it is possible to connect all nodes within the budget.

Task 3 Problem Finding Champion

a) Algorithm for champion nodes detection:

A **champion** in the directed graph $G = (V, E)$ is a node that has defeated every other node, either directly or indirectly. That means, for a node u to be a champion, there must be a path from u to every other node in the graph.

To determine the champions, we implement a **Depth-First Search (DFS)** approach to check which nodes can reach all other nodes.

- Algorithm Implementation:

The algorithm follows these steps:

1. **Define a DFS function** to explore the graph and track visited nodes.
2. **Iterate through each node** in the graph and perform DFS starting from that node.
3. **Check reachability**: If a node can reach all other nodes in the graph, it is considered a champion.
4. **Return the list of champions**.

The implementation is as follows:

```
def find_champions(graph):
    def dfs(node, visited):
        for neighbor in graph.graph[node]:  # Access neighbors
            if neighbor not in visited:
                visited.add(neighbor)
                dfs(neighbor, visited)

    champions = []
    for node in graph.graph.keys():  # Iterate over the dictionary keys (nodes)
        visited = set()
        dfs(node, visited)
        if len(visited) == len(graph.graph.keys()):  # Check if the node reaches all others
            champions.append(node)

    return champions
```

1. **`dfs(node, visited)`**

It is a recursive function that explores all reachable nodes from a given starting node. It marks each visited node and continues exploring unvisited neighbors.

2. **Main Loop (Finding Champions)**

Iterates through each node in the graph. Initializes a **`visited`** set to track which nodes can be reached from the current node. Then, it calls **`dfs(node, visited)`** to explore the graph. If the **`visited`** set contains all nodes in the graph, the current node is a champion and is added to the list.

3. **Return Champions**

- Graph Creation and Execution:

For the graph representation, the class described in Task 1, part d will be used. For that reason, the creation of the graph will be similar, and we will simply call the **`find_champions`** function to later print them.

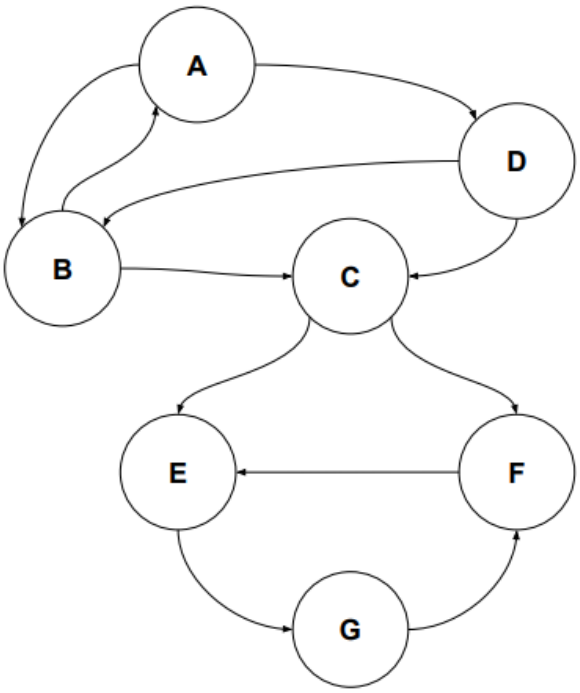
```
# Create graph from Figure 3
graph = Graph(7)  # 7 nodes

edges = [
    ('A', 'B'), ('B', 'A'), ('A', 'D'), ('D', 'B'), ('B', 'C'), ('D', 'C'),
    ('C', 'E'), ('C', 'F'), ('F', 'E'), ('E', 'G'), ('G', 'F')
]

for u, v in edges:
    graph.add_edge(u, v)

# Find Champions
champions = find_champions(graph)
print("Champion nodes:", champions)
```

- Results:



After executing the algorithm on the given graph in Figure 3, the output obtained is:

```
Champion nodes: ['A', 'B', 'D']
```

This result indicates that nodes **A, B, and D** can reach all other nodes in the graph through directed or undirected edges.

The presence of multiple champion nodes suggests that the graph has multiple strong connections that allow different starting points to reach all other nodes. In particular, nodes **A, B, and D** likely have access to key branching points in the structure, allowing them to cover the entire graph.

b) Algorithm for champion nodes detection:

In this section, we aim to identify groups of nodes in a directed graph where each node can reach every other node within the same group, either directly or indirectly. For that reason, we need to obtain the **Strongly Connected Components (SCCs)** of a directed graph.

- Algorithm Implementation:

We extend the **`Graph`** class (as described in Task 1, Section D) to include SCC detection. The key modifications include this methods:

1. **Perform Depth-First Search (DFS) on the original graph** to compute the **finishing times** of each node. This helps determine the order in which nodes should be processed in the next step.

```
def dfs(self, node, visited, stack):
    """Performs DFS and records finishing order in a stack."""
    visited.add(node)
    for neighbor in self.graph[node]:
        if neighbor not in visited:
            self.dfs(neighbor, visited, stack)
    stack.append(node)  # Push node to stack when finished
```

2. **Transpose the graph** by reversing all its edges, resulting in a new graph G^T.

```
def transpose(self):
    """Returns the transposed graph (reversed edges)."""
    transposed = Graph(self.V)
    for node in self.graph:
        for neighbor in self.graph[node]:
            transposed.add_edge(neighbor, node)  # Reverse direction
    return transposed
```

3. **Perform DFS on** G^T, but process nodes in the order of **decreasing finishing times** obtained in Step 1. Each DFS tree in this step forms an SCC, meaning all nodes within the same tree belong to the same strongly connected component.

```
def dfs_scc(self, node, visited, scc):
    """Performs DFS on the transposed graph to collect SCC nodes."""
    visited.add(node)
    scc.append(node)
    for neighbor in self.graph[node]:
        if neighbor not in visited:
            self.dfs_scc(neighbor, visited, scc)
```

4. **Perform all the process** in a method:

```
def find_sccs(self):
    """Finds and returns all strongly connected components."""
    stack = []
    visited = set()

    # Step 1: Perform DFS on original graph to determine finishing times
    for node in self.graph:
        if node not in visited:
            self.dfs(node, visited, stack)

    # Step 2: Transpose the graph
    transposed = self.transpose()

    # Step 3: Process nodes in decreasing order of finishing times
    visited.clear()
    sccs = []

    while stack:
        node = stack.pop()
        if node not in visited:
            scc = []
            transposed.dfs_scc(node, visited, scc)
            sccs.append(scc)

    return sccs
```

- Graph Creation and Execution:

The creation of the graph will be similar to the previous task, and we will simply call the **`find_sccs`** function to later print the groups.

```
# Create graph from Figure 3
graph = Graph(7)  # 7 nodes

edges = [
    ('A', 'B'), ('B', 'A'), ('A', 'D'), ('D', 'B'), ('B', 'C'), ('D', 'C'),
    ('C', 'E'), ('C', 'F'), ('F', 'E'), ('E', 'G'), ('G', 'F')
]

for u, v in edges:
    graph.add_edge(u, v)

sccs = graph.find_sccs()
print("Strongly Connected Components:", sccs)
```

```
Strongly Connected Components: [['A', 'B', 'D'], ['C'], ['E', 'F', 'G']]
```

- Running Time:

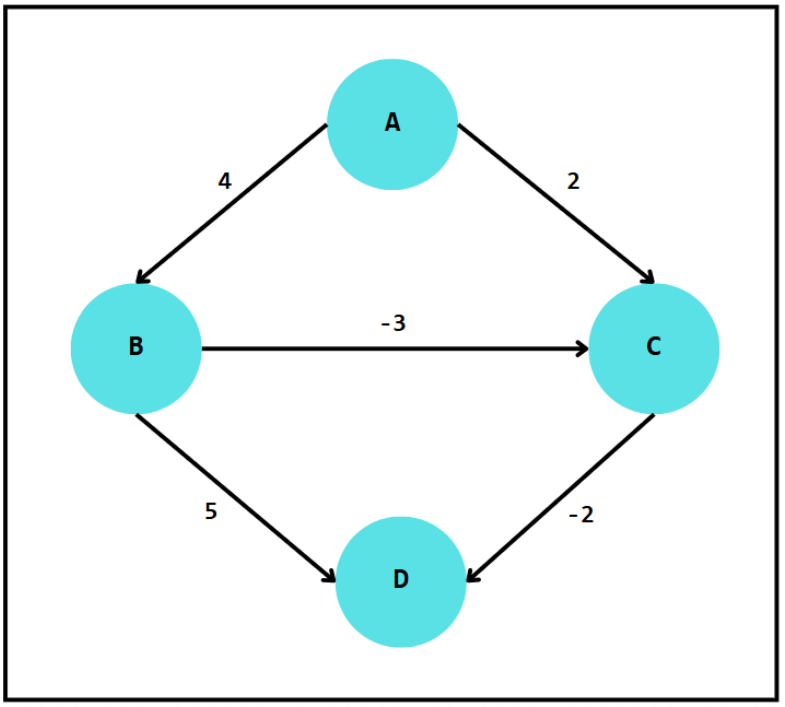
The algorithm consists of three main steps:

- **First DFS traversal**: $O(V + E)$
- **Graph transposition**: $O(V + E)$
- **Second DFS traversal**: $O(V + E)$

Since each step runs in $O(V + E)$, the total running time of the algorithm is $O(V + E)$, making it efficient for SCC detection.

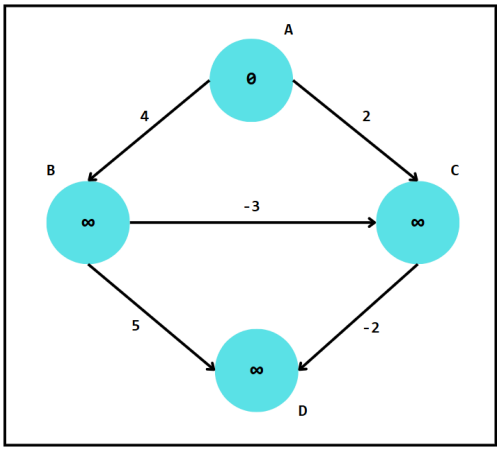
Task 4 Problem Shortest Path

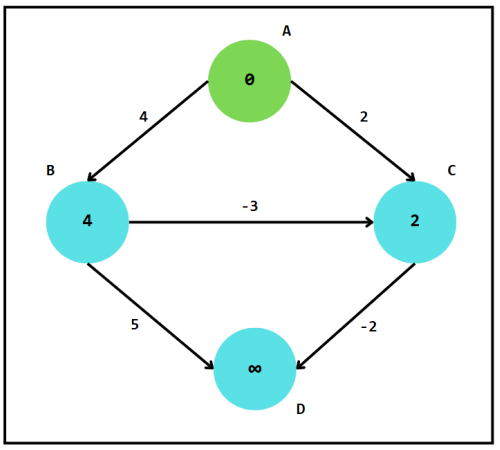
Dijkstra does not work correctly when there are negative weights because it assumes that once it finds the smallest distance to a node, that distance cannot change, which is false if there are edges with negative weights. For example, we have the following graph:



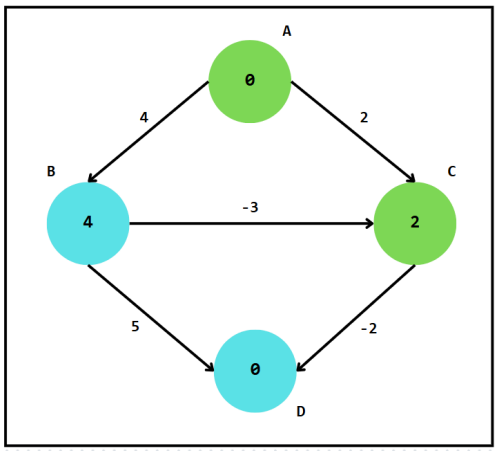
If we execute Dijkstra from vertex **A:**

Iteration	Visited	Vertices	Distance(B)	Distance(C)	Distance(D)
initial		A,B,C,D	∞	∞	∞
1	A	B,C,D	4	2	∞
2	A,C	B,D	4	2	0
3	A,C,D	B	4	2	0

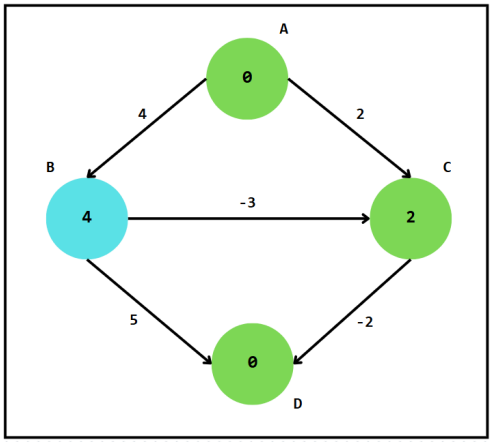




Initial



Iteration 1



Iteration 3

Iteration 4

Dijkstra gives the result:
A → **C** → **D** with cost **0**.
A → **B** with cost **4**.
A → **C** with cost **2**.

But the shortest actual path to D is **A** → **B** → **C** → **D** with cost 4 + (-3) + (-2) = -1, which Dijkstra does not find.
To handle negative weights, the **Bellman-Ford** algorithm should be used, which allows finding shortest paths even with negative weights (as long as there are no negative cycles). Using the previous graph:

Iteration	Distance(B)	Distance(C)	Distance(D)
initial	∞	∞	∞
1	**4**	**1**	**-1**
2	**4**	**1**	**-1**

Iteration 1
Relax all edges:
A → B (4) → dist(B) = min(∞, 0 + 4) = 4
A → C (2) → dist(C) = min(∞, 0 + 2) = 2
B → C (-3) → dist(C) = min(2, 4 + (-3)) = 1
B → D (5) → dist(D) = min(∞, 4 + 5) = 9
C → D (-2) → dist(D) = min(9, 1 + (-2)) = -1

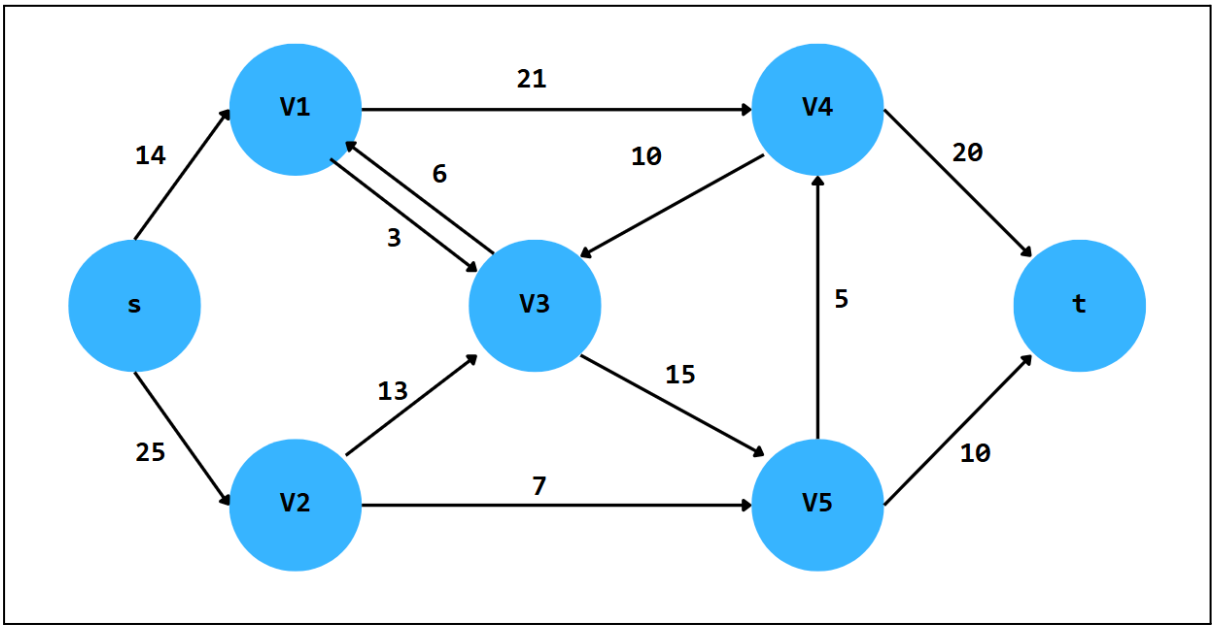
Iteration 2
We relax all edges again:
A → B (4) → dist(B) = min(4, 0 + 4) = 4 (no change)
A → C (2) → dist(C) = min(1, 0 + 2) = 1 (no change)
B → C (-3) → dist(C) = min(1, 4 + (-3)) = 1 (no change)
B → D (5) → dist(D) = min(-1, 4 + 5) = -1 (no change)
C → D (-2) → dist(D) = min(-1, 1 + (-2)) = -1 (no change)

No distance has changed, indicating that the algorithm has converged.

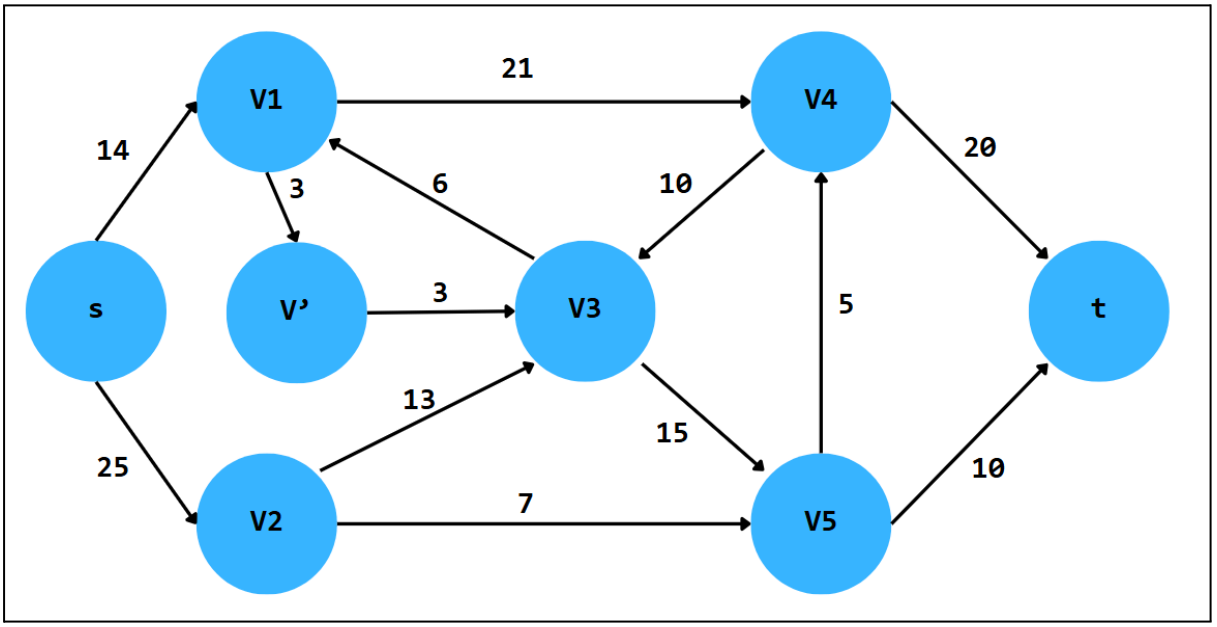
Comparison with Dijkstra
Dijkstra gave **dist(D) = 0** (incorrect).
Bellman-Ford gave **dist(D) = -1** (correct).

Task 5 Problem Maximum Flow

Given the flow network G:



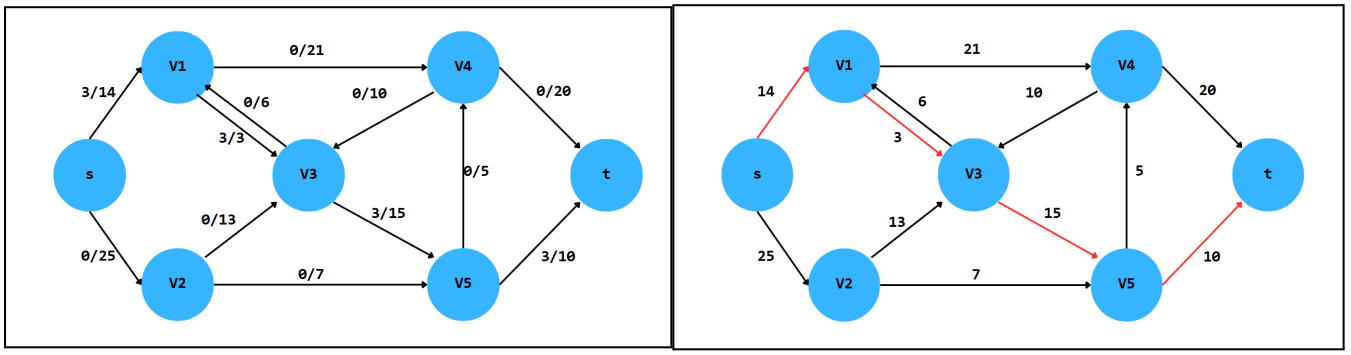
a) Resolve the **antiparallel edge issue** in G



b) Walk through the **Ford-Fulkerson algorithm** by hand, starting from source s and ending at sink t, on the flow network G.

Iteration 1

Network G | **Residual G_f and augmented path**

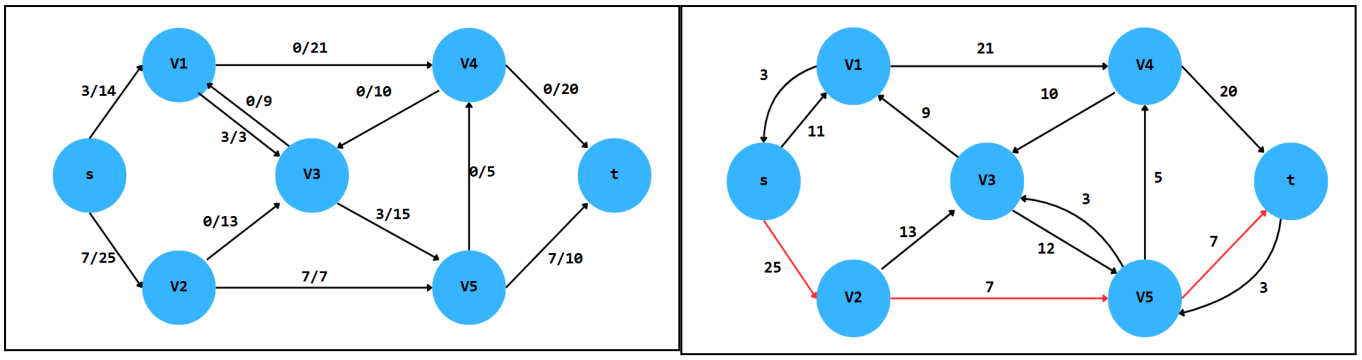


Path: s → V1 → V3 → V5 → t

$c_f\,(p)$: $min\{14, 3, 15, 18\} = 3$

Maximum flow = 0

Iteration 2

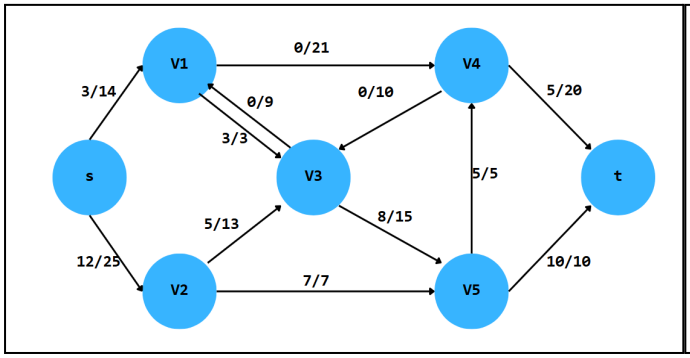


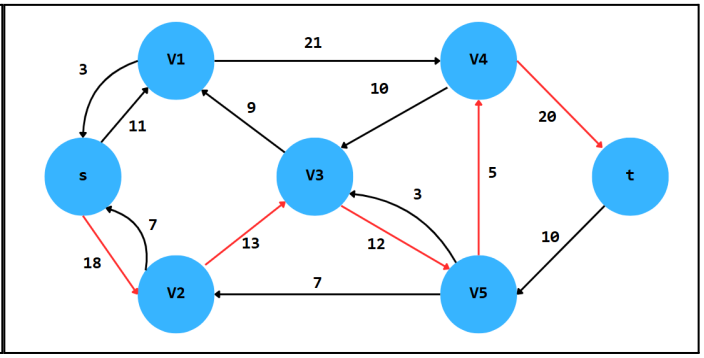
Path: s → V2 → V5 → t

$c_f\,(p)$: $min\{25, 7, 10\} = 7$

Maximum flow = 3

Iteration 3



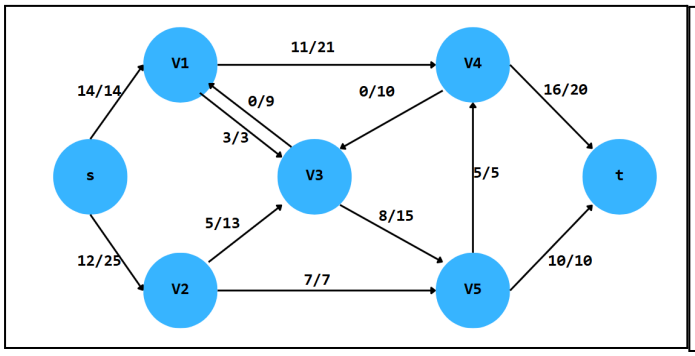


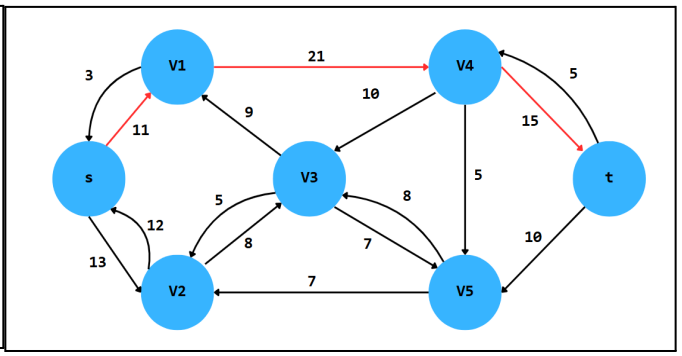
Path: s → V2 →V3 → V5 → V4 → t

$c_f\,(p)$: $min\{18, 13, 12, 5, 20\} = 5$

Maximum flow = 10

Iteration 4



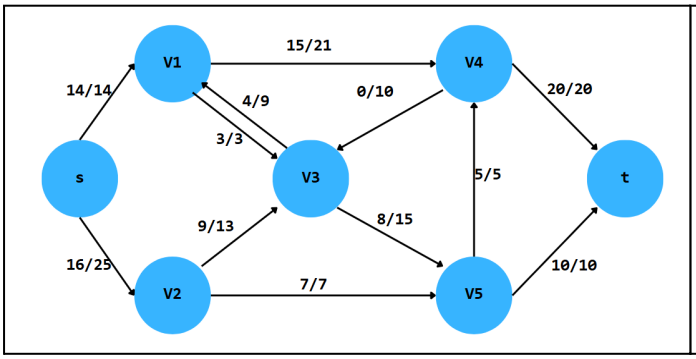


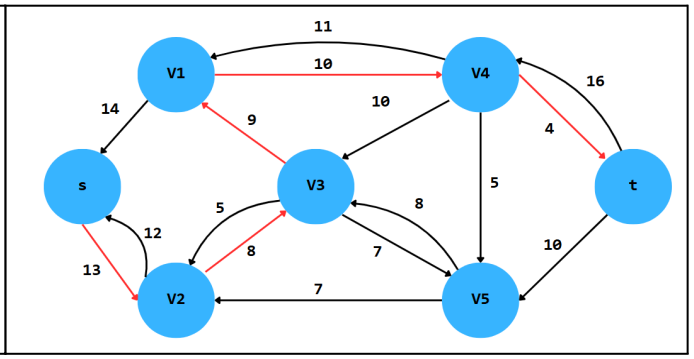
Path: s → V1 → V4 → t

$c_f\,(p)$: $min\{11, 21, 15\} = 11$

Maximum flow = 15

Iteration 5



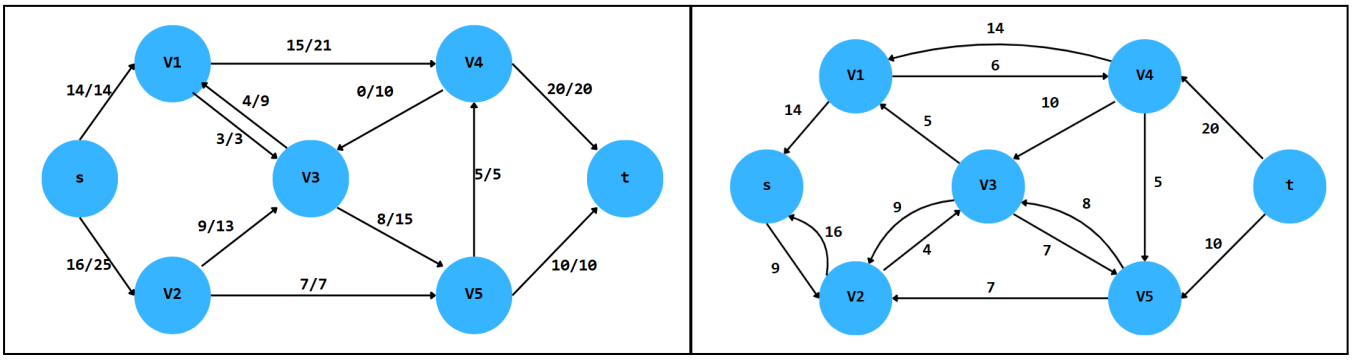


Path: s → V2 → V3 → V1 → V4 → t

$c_f\,(p)$: $min\{12, 8, 9, 21, 4\} = 4$

Maximum flow = 26

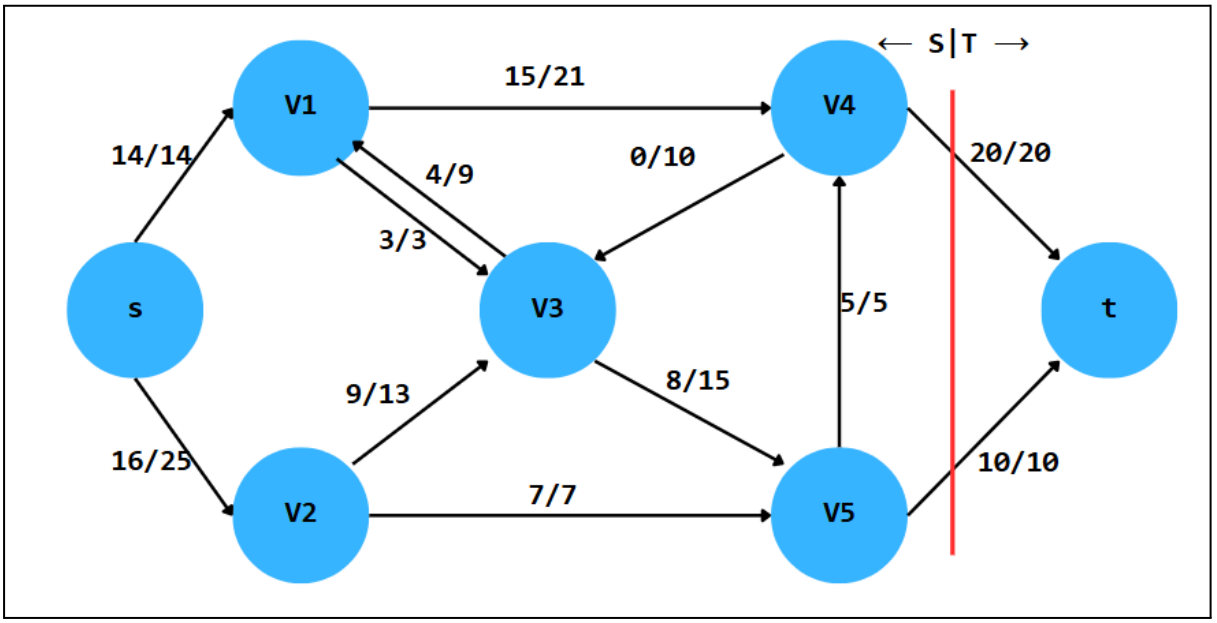
Iteration 6



No more augmented paths

Maximum flow = 30

c) Show the **bottleneck** of the flow network by using **cuts.**



S = {s, V1, V2, V3, V4, V5}
T = {t}

f (S, T) = 20 +10 = **30**
c (S, T) = 20 + 10 = **30**

The minimum cut always has a **capacity equal to the maximum flow**, according to the maximum flow and minimum cut theorem.

d) Describe the **running time** of the Ford-Fulkerson algorithm. Suggest any improvement to the Ford-Fulkerson algorithm.

The running time of the **Ford-Fulkerson algorithm** depends on how augmenting paths are found. The two most common approaches are:

1. **Using Depth-First Search (DFS) – Basic Ford-Fulkerson:**
    - In the worst case, the algorithm can run in **exponential time** if small bottleneck edges are always chosen first.
    - If edge capacities are integers, the worst-case time complexity is $O(E \cdot C)$, where CCC is the maximum capacity in the network.
2. **Using Breadth-First Search (BFS) – Edmonds-Karp Algorithm:**
    - A possible improvement to Ford-Fulkerson is using **BFS** instead of **DFS** to always find the shortest augmenting path.
    - This ensures that the algorithm runs in **polynomial time** with a complexity of $O(VE^2)$**.**
    - Using BFS prevents the worst-case exponential behavior of DFS and guarantees faster convergence.

A key improvement to the Ford-Fulkerson algorithm is to use **BFS instead of DFS**, which leads to the **Edmonds-Karp algorithm** with a guaranteed running time of $O(VE^2)$. This makes the algorithm more efficient and avoids potential inefficiencies in finding augmenting paths.

Github files

https://github.com/AntonMG4/DAT600_AlgorithmTheory/tree/main/Assignment3