Assignment 3. Graph Algorithms.

Student name: Evgenii Dudkin
e-mail: e.dudkin@stud.uis.no

A solution called «Graph» was implemented to solve the tasks in the assignment. It comprises a C# library «Graph» containing graph abstractions and various graph algorithms. A separate C# project was created for each task in the assignment: «BasicGraph» for Task 1, «CableNetwork» for Task 2, «FindingChampion» for Task 3, «ShortestPath» for Task 4, and «MaximumFlow» for Task 5. Each project utilizes the library and its functionality to solve the problems specified in the tasks. Further details about each project, including their algorithms and the components of the core library, will be provided in the subsequent sections of the report.

For access to the implementations, code, comments, and other related materials, please refer to the GitHub repository [1].

1 Problem Basic Graph

1.1

At the core of the «Graph» library lies an abstract class called GraphBase, which enables various operations such as initializing graphs, retrieving vertices, adding and removing edges, visualizing graphs, and more. GraphBase serves as the blueprint for graph abstraction, allowing users to execute all necessary operations on graphs, including initialization, traversal of vertices or edges, graph modification (adding or removing edges), and graph display. Concrete implementations like AdjacencyMatrixGraph and AdjacencySetGraph extend this abstraction to actualize specific graph representations.

1.1.1

The given adjacency matrix was converted into an adjacency list, where adjacent lists were organized alphabetically. This conversion was achieved through the utilization of the AdjacencyMatrixGraph and AdjacencySetGraph classes. The former employs an adjacency matrix, while the latter uses an adjacency list for graph representation. Additionally, the implementations of the GraphBase are capable to accept another graph as an argument in the constructor and initialize the instances based on the provided graph. Consequently, an instance of AdjacencyMatrixGraph representing the given adjacency matrix was created and passed as an argument to the AdjacencySetGraph constructor (Listing 1). Thus, the conversion from the adjacency matrix to an adjacency list was executed, with adjacent lists being organized alphabetically in the AdjacencySetGraph as per specification.

```
AdjacencyMatrixGraph matrixGraph = new(new[,]
{
    { 0, 1, 0, 0, 0, 0 },
    { 0, 1, 1, 1, 0, 0 },
    { 1, 1, 0, 0, 1, 0 },
    { 0, 0, 0, 0, 1, 1 },
    { 0, 0, 1, 1, 0, 0 },
    { 0, 0, 0, 1, 0, 0 },
}, true);
AdjacencySetGraph setGraph = new(matrixGraph);
```

Listing 1: Adjacency matrix convertion

1.1.2

GraphBase class featured a display method, enabling both AdjacencySet-Graph and AdjacencyMatrixGraph implementations to visualize the graph. For the adjacency matrix, the matrix itself is outputted in the console. The adjacency set graph is visually represented using the R language. Leveraging the R.Net library allowed for the utilization of R functions and capabilities within the C# code. As a result, the given adjacency matrix was depicted in graph form (Figure 1), as the matrix had been previously converted into adjacency set form during the initial step.

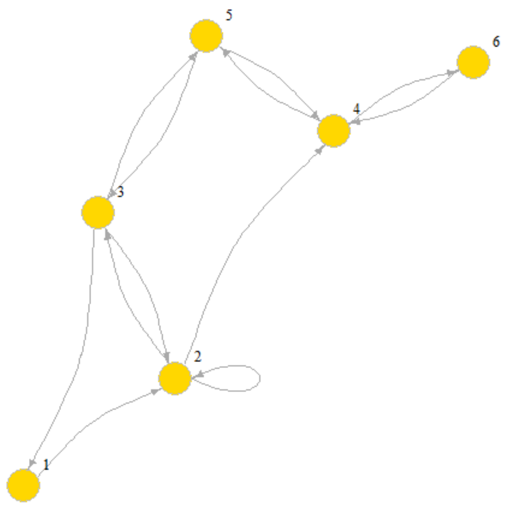


Figure 1: Adjacency matrix in a graph form

1.1.3

In the project «BasicGraph» an AdjacencySetGraph object was instantiated out of the graph from the assignment (Listing 2). Subsequently, the graph was displayed (Figure 2).

```
AdjacencySetGraph graph = new(new List<VertexBase>
{
    new(1, "A"),
    new(2, "B"),
    new(3, "C"),
    new(4, "D"),
    new(5, "E"),
    new(6, "F"),
    new(7, "G"),
    new(8, "H"),
    new(9, "I"),
    new(10, "J"),
}, true);
graph.AddEdge( 1, 2 ); // A – B
graph.AddEdge( 2, 3 ); // B – C
graph.AddEdge( 2, 4 ); // B – D
graph.AddEdge( 3, 5 ); // C – E
graph.AddEdge( 3, 6 ); // C – F
graph.AddEdge( 4, 5 ); // D – E
graph.AddEdge( 4, 6 ); // D – F
graph.AddEdge( 6, 2 ); // F – B
graph.AddEdge( 5, 6 ); // E – F
graph.AddEdge( 5, 7 ); // E – G
graph.AddEdge( 6, 7 ); // F – G
graph.AddEdge( 5, 10 ); // E – J
graph.AddEdge( 6, 8 ); // F – H
graph.AddEdge( 6, 10 ); // F – J
graph.AddEdge( 8, 9 ); // H – I
graph.AddEdge( 10, 9 ); // J – I

graph.Display();
```

Listing 2: Graph initialization

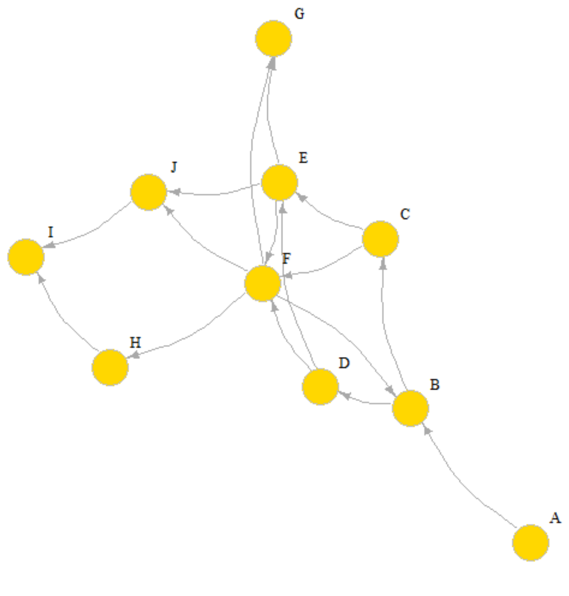


Figure 2: Adjacency list in a graph form

1.2

1.2.1 Breadth-First Search

The Breadth-First Search (BFS) algorithm was implemented in the «Graph» library. This algorithm is designed to find the shortest paths from a source vertex to all reachable vertices in a graph. The figures below illustrate the process of the BFS algorithm on a given graph with source vertex A. A white color indicates that the vertex has not been visited, gray signifies that the vertex is placed in the processing queue and awaits its turn, while black denotes that the vertex has been visited and all necessary parameters have been calculated for it.

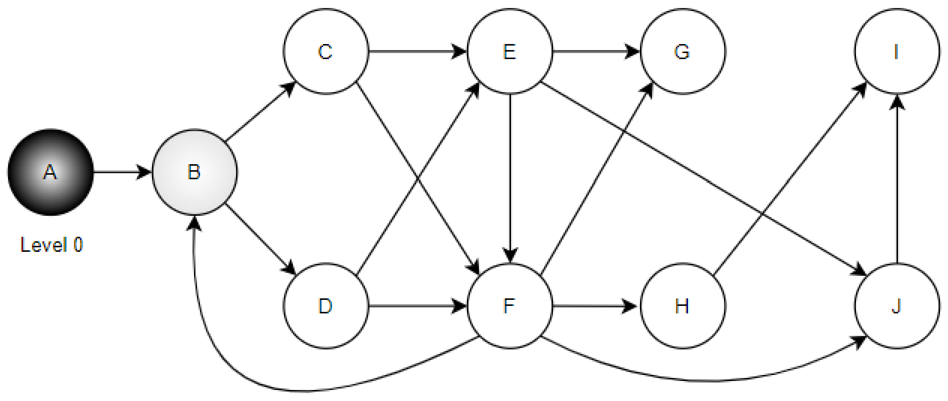


Figure 3: BFS. Step 1.

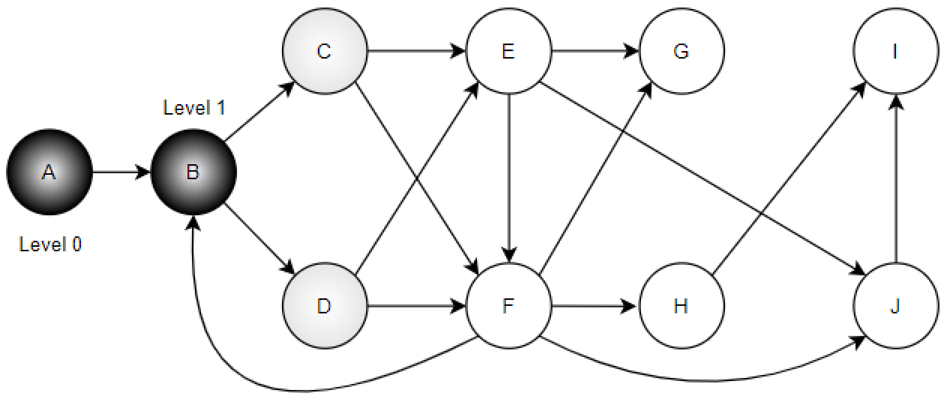


Figure 4: BFS. Step 2.

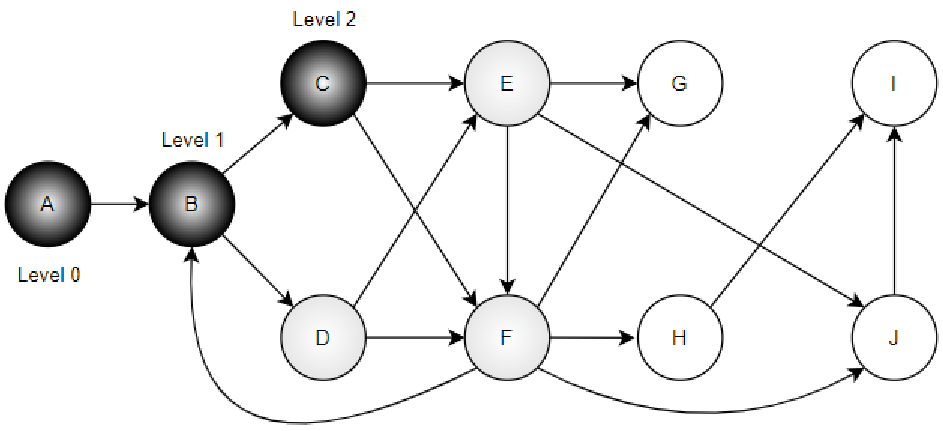


Figure 5: BFS. Step 3.

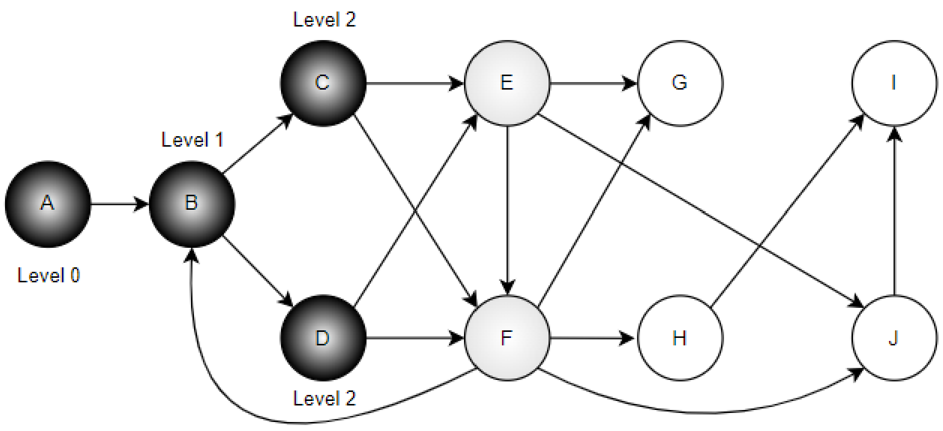


Figure 6: BFS. Step 4.

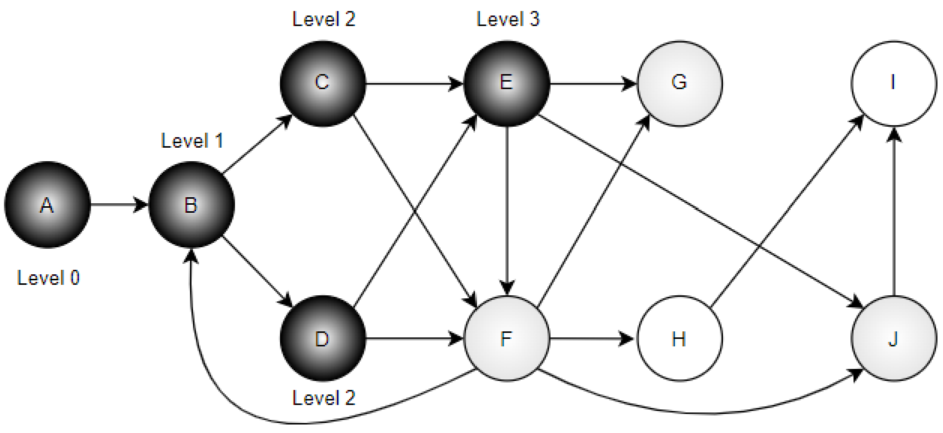


Figure 7: BFS. Step 5.

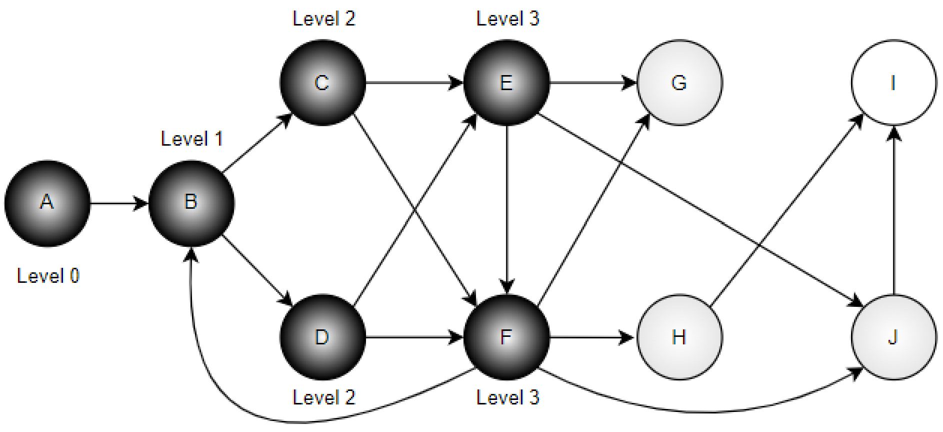


Figure 8: BFS. Step 6.

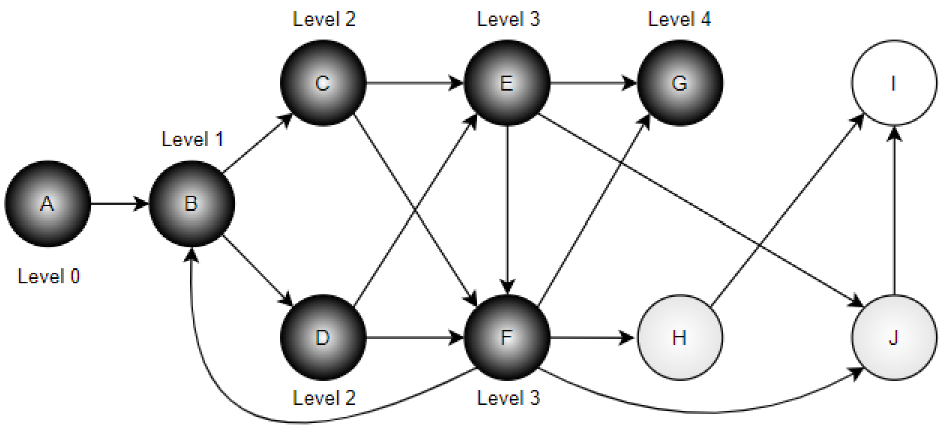


Figure 9: BFS. Step 7.

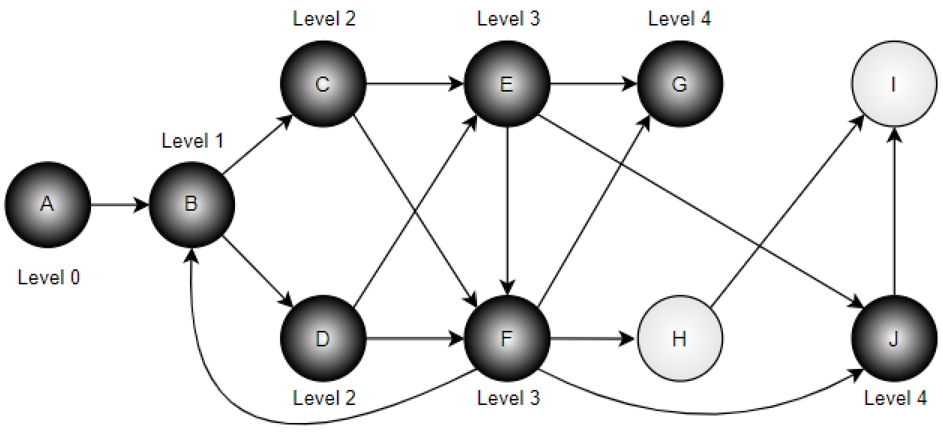


Figure 10: BFS. Step 8.

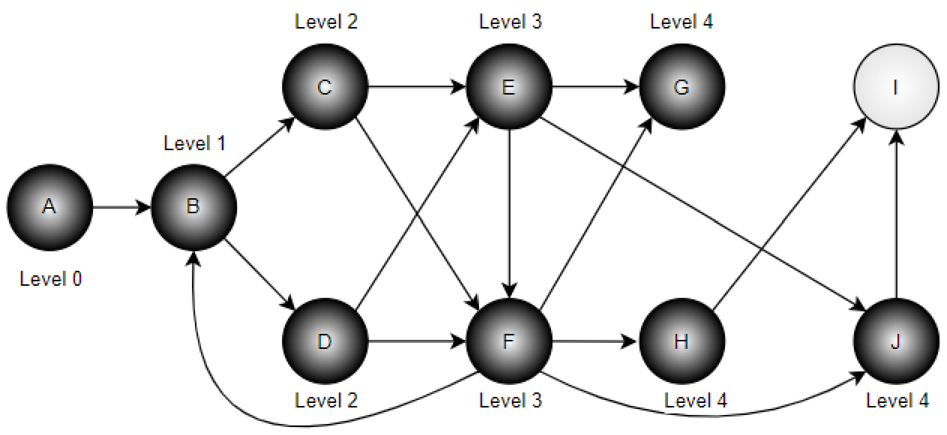


Figure 11: BFS. Step 9.

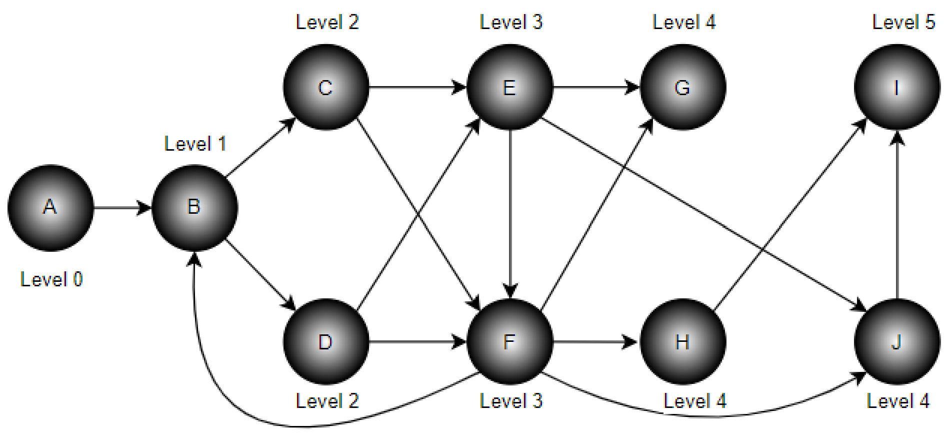


Figure 12: BFS. Step 10.

1.2.2 Depth-First Search

The Depth-First Search (DFS) algorithm has been implemented in the ”Graph” library. The figures below illustrate the process of the DFS algorithm on a given graph. A white color indicates that the vertex has not been visited, gray signifies that the vertex is discovered, while black denotes that the vertex has been visited and all necessary parameters have been calculated for it.

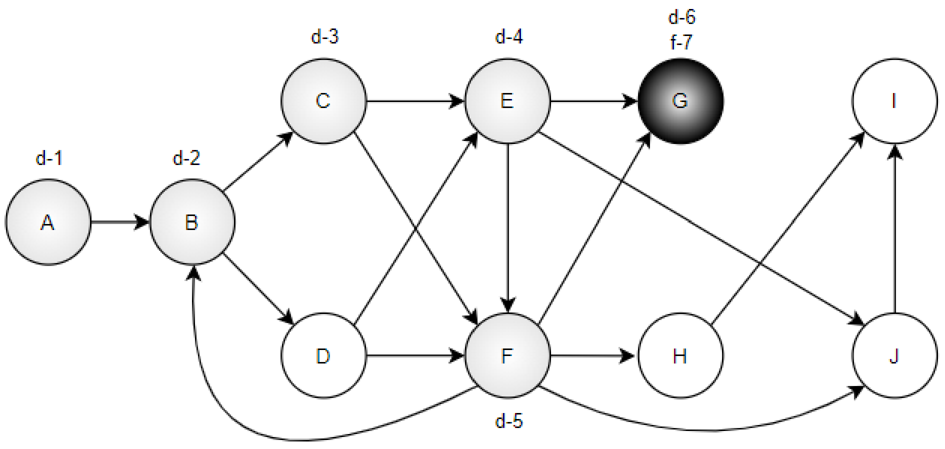


Figure 13: DFS. Step 1.

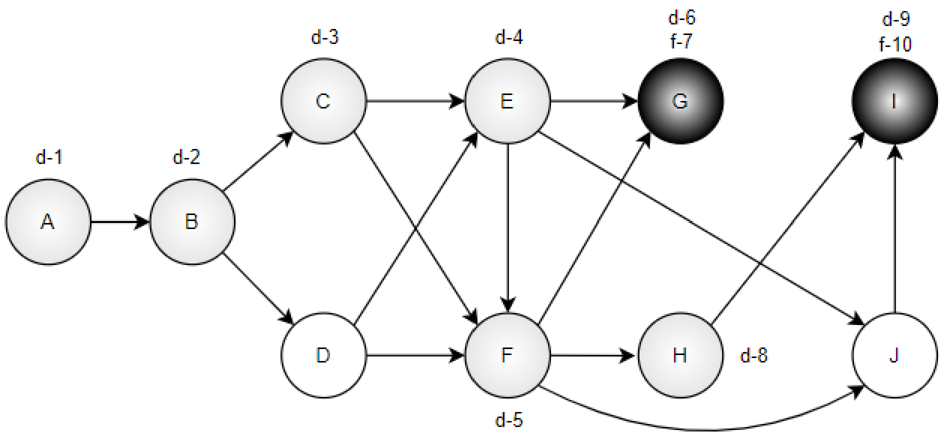


Figure 14: DFS. Step 2.

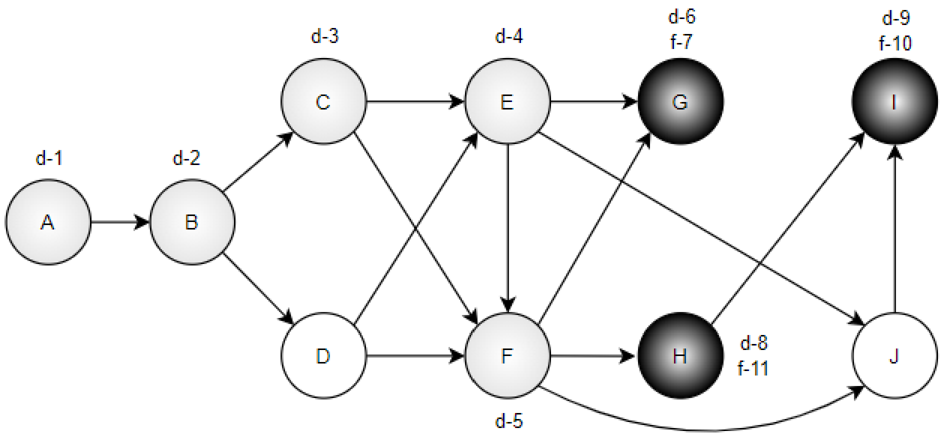


Figure 15: DFS. Step 3.

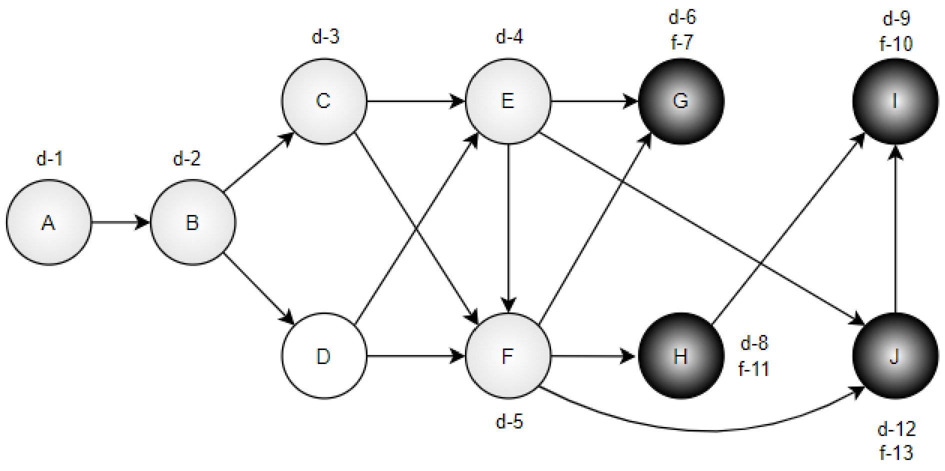


Figure 16: DFS. Step 4.

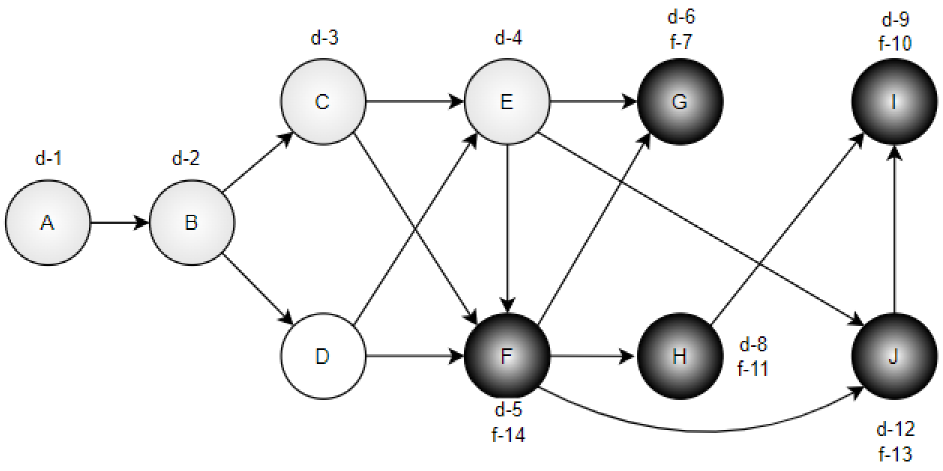


Figure 17: DFS. Step 5.

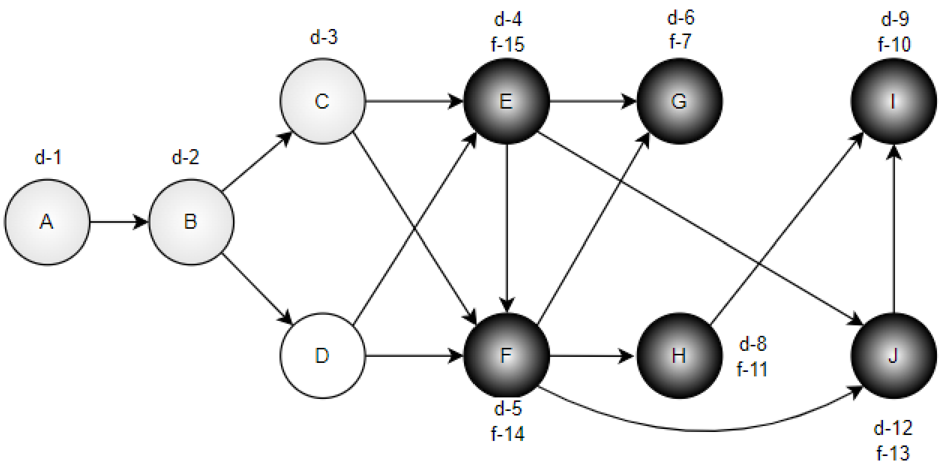


Figure 18: DFS. Step 6.

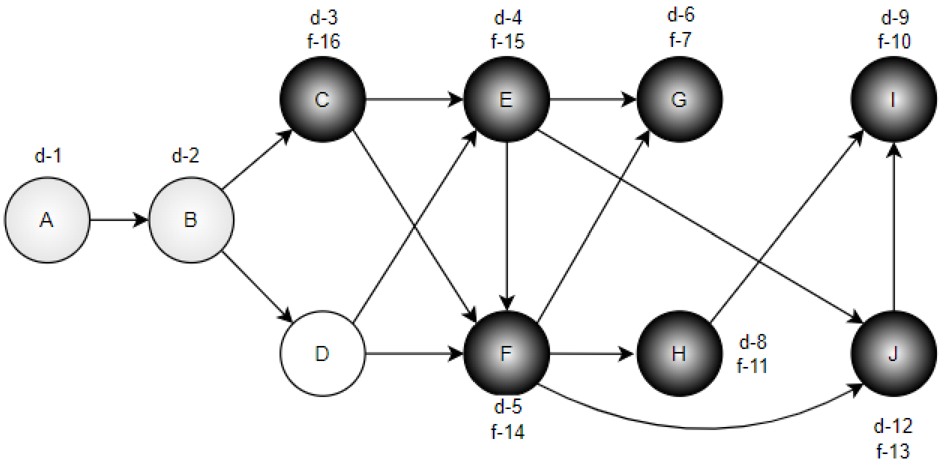


Figure 19: DFS. Step 7.

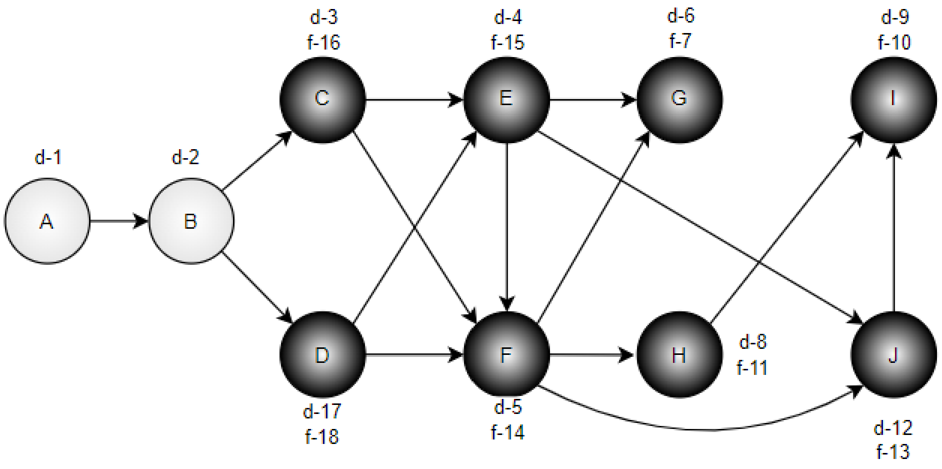


Figure 20: DFS. Step 8.

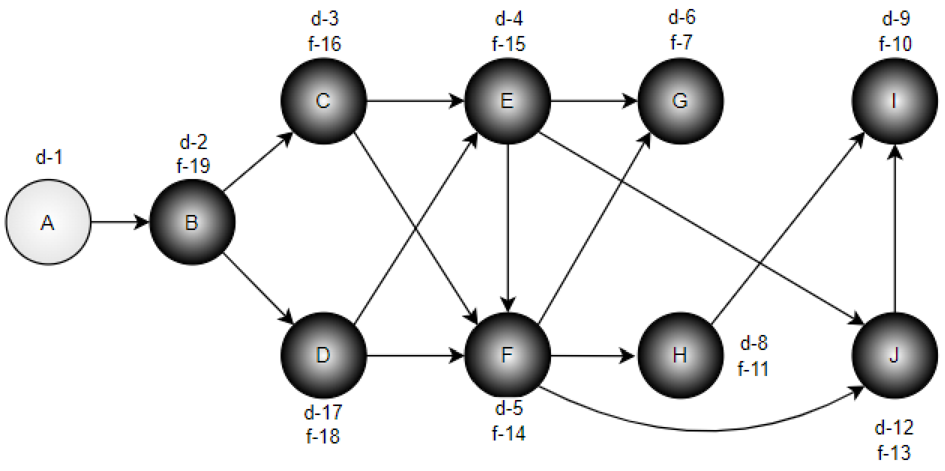


Figure 21: DFS. Step 9.

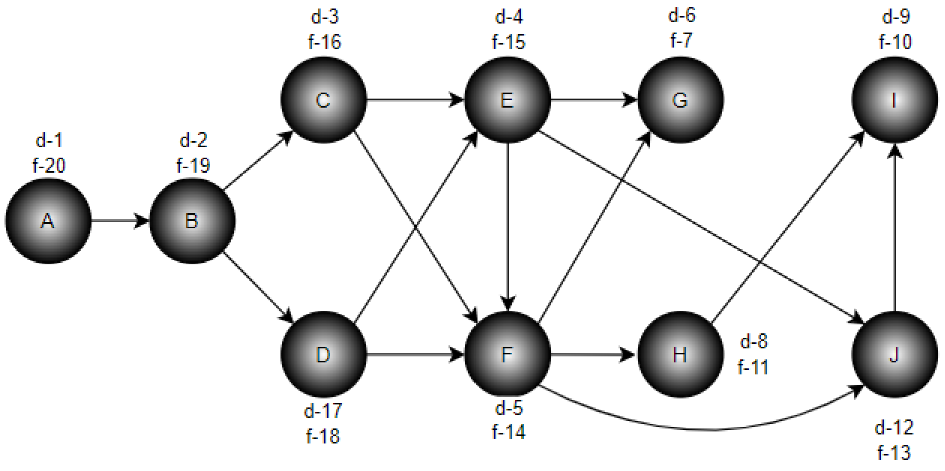
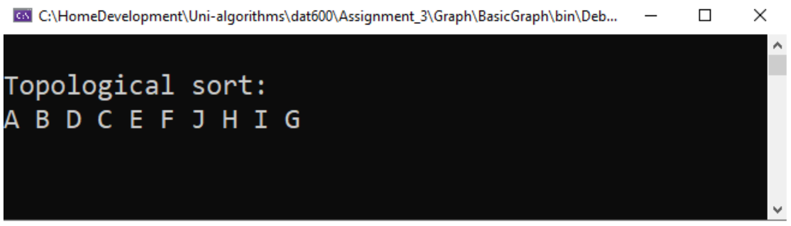


Figure 22: DFS. Step 10.

1.3

To create a Directed Acyclic Graph (DAG) from the graph G, the cycle B-D-F should be eliminated by removing the back edge F-B. The «Graph» library can perform a topological sort on a DAG. This process is accomplished by maintaining a side list of topologically sorted vertices in the DFS algorithm. When a vertex is visited and colored Black, it is added to the head of the list. Figure 23 displays the output of the program when the function PrintTopologicalSort, which prints the content of the list, was executed. Thus, A B D C E F J H I G is a topological sort on the resulting DAG.



```
C:\HomeDevelopment\Uni-algorithms\dat600\Assignment_3\Graph\BasicGraph\bin\Deb...

Topological sort:
A B D C E F J H I G
```

Figure 23: Topological sort on a DAG

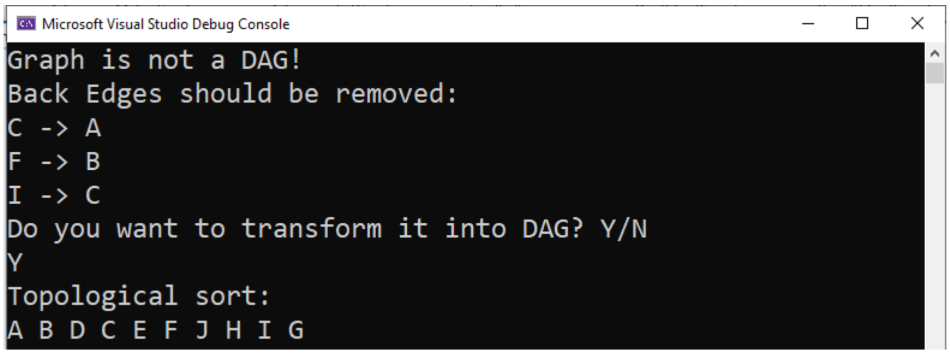
1.4

Considering the fact that in a directed graph, an edge (u,v) is a back edge if and only if $v.d \leq u.d < u.f \leq v.f$, the algorithm for transforming a directed graph into a DAG was implemented in the solution. First, a DFS is executed, and all discovery and finish times are recorded for the vertices. Then, the list of back edges is populated by iterating through all edges and checking the condition for a back edge (see Listing 3). Afterward, all back edges are removed, resulting in obtaining a DAG.

```
foreach( VertexBase fromVertex in _graph.Vertices )
    foreach( int adjacentVertexId in _graph.GetAdjacentVertices( fromVertex.Id ) )
    {
        VertexInfo from = _vertices[fromVertex.Id];
        VertexInfo to = _vertices[adjacentVertexId];
        if( to.DiscoveryTime <= from.DiscoveryTime &&
            from.DiscoveryTime < from.FinishTime
                && from.FinishTime <= to.FinishTime ) //back edge
        {
            backEdges.Add( (new VertexBase( from.Id, from.Name ),
                new VertexBase( to.Id, to.Name )) );
        }
    }
```

Listing 3: Adjacency matrix convertion

Figure 24 presents the result of the transformation algorithm and the topological sort of the resulting DAG, taking into account the presence of additional edges: one from I to C and another from C to A



```
Graph is not a DAG!
Back Edges should be removed:
C -> A
F -> B
I -> C
Do you want to transform it into DAG? Y/N
Y
Topological sort:
A B D C E F J H I G
```

Figure 24: Transformation into a DAG

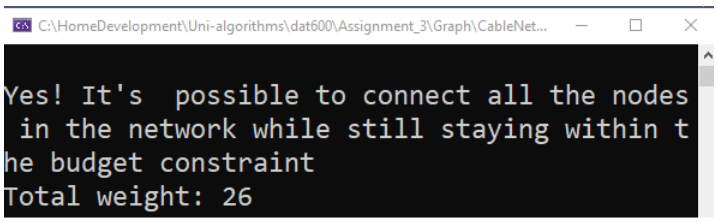
2 Problem Cable Network

2.1

To solve the problem of finding a cable network that can connect all neighborhoods with a total cost under the budget, Kruskal's algorithm for finding a minimum spanning tree was implemented in the «Graph» library. Listing 4 displays the code where the undirected graph from the assignment is used as input for Kruskal's algorithm. The result of the MST operation is indeed a minimum spanning tree, which is also in the form of a graph. This implies that the total weight property can be accessed and compared with a budget limit value. Figure 25 illustrates the output of the program with a network from the assignment and a budget of 30 as input parameters. It shows that it is possible to find such cable network, Figure 26 presents the obtained minimum spanning tree (needed cable network).

```
AdjacencySetGraph graph = new();
//init of a graph
{
    //
}
int budgetLimit = 30;
Kruskal kruskal = new(graph);
GraphBase minSpanningTree = kruskal.Mst();
bool isWithinBudget = minSpanningTree.TotalWeight <= budgetLimit;
```

Listing 4: Cable Network problem solution



C:\HomeDevelopment\Uni-algorithms\dat600\Assignment_3\Graph\CableNet...

```
Yes! It's  possible to connect all the nodes
 in the network while still staying within t
he budget constraint
Total weight: 26
```

Figure 25: Solution of Cable Netwrok Problem

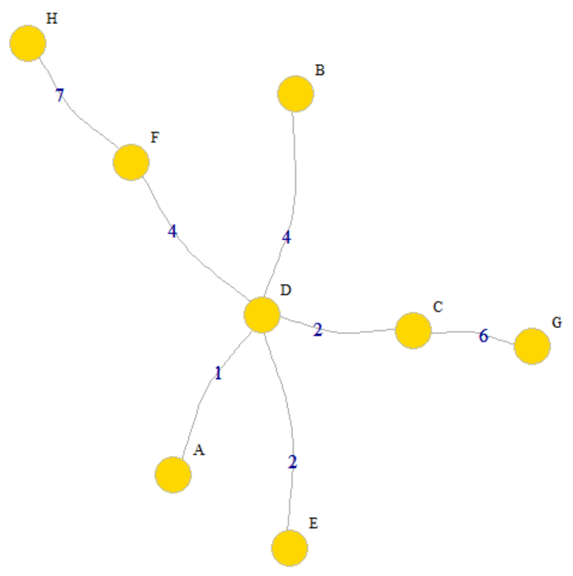
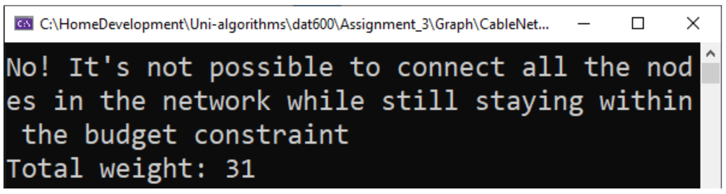


Figure 26: Obtained Cable Netwrok (1)

2.2

The Kruskal's algorithm used in the previous task was modified with the introduction of restrictions. In this context, a restriction is a structure that keeps the maximum edge count for the vertex in the final minimal spanning tree. The algorithm evaluates potential edges to include in the minimum spanning tree, ensuring they do not exceed the maximum edge count for the vertices, thus avoiding contradictions with the restrictions. However, when there is a restriction on vertex D to have a maximum of 3 edges, it becomes evident that meeting the budget constraint is not feasible. Figure 27 displays the output of the program with this restriction.

This example illustrates that such restrictions do not yield optimal solutions, and therefore, our specific assignment serves as a notable counter-example.

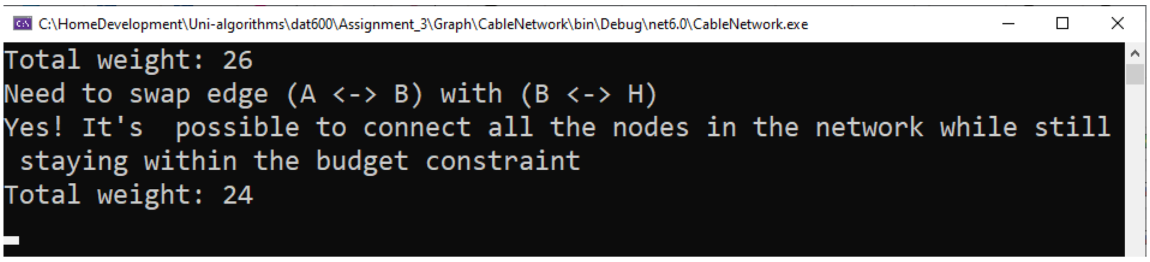


```
No! It's not possible to connect all the nod
es in the network while still staying within
 the budget constraint
Total weight: 31
```

Figure 27: Solution of Cable Netwrok Problem with restrictions

2.3

The algorithm for adjusting the position of a single edge to find a solution with a newly introduced constraint involves checking all possible combinations of edge swaps and running Kruskal's algorithm for each resulting graph until a solution is found. Executing the algorithm yields a solution (Figure 28 and Figure 29) by changing edges A - B and B - H, ensuring compliance with the budget constraint of 25 for the network.



```
Total weight: 26
Need to swap edge (A <-> B) with (B <-> H)
Yes! It's  possible to connect all the nodes in the network while still
 staying within the budget constraint
Total weight: 24
```

Figure 28: Solution of Cable Netwrok Problem with edge swap

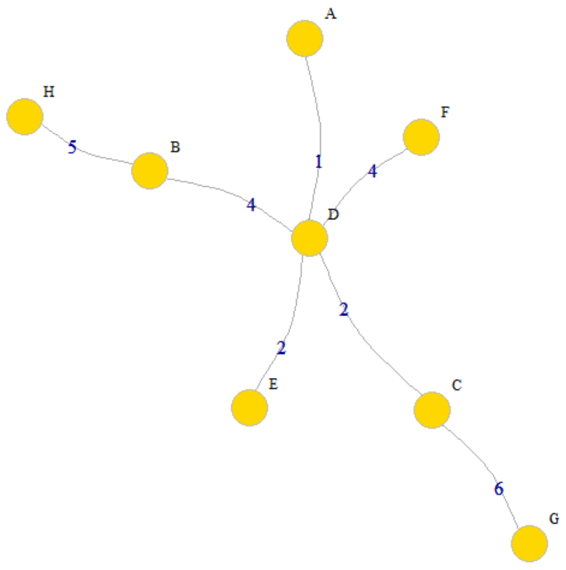


Figure 29: Obtained Cable Netwrok (2)

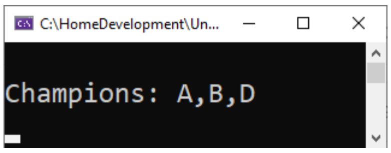
3 Problem Finding Champion

3.1

The algorithm for identifying and listing all the champions in the directed graph involves traversing the list of all vertices, calling BFS for each, and verifying that there are paths to all other vertices. Listing 5 demonstrates the implementation of this algorithm, and Figure 30 displays the output of the program applying this algorithm to a graph from the assignment.

```
List<VertexBase> champions = new();
BreadthFirstSearch bfs = new(graph);
foreach( VertexBase vertex in graph.Vertices )
{
    bfs.BFS( vertex.Id );
    bool isChampion = true;
    foreach( VertexBase tmpVertex in graph.Vertices )
    {
        if( vertex.Id == tmpVertex.Id )
            continue;
        if( !bfs.IsPath( vertex.Id, tmpVertex.Id ) )
        {
            isChampion = false;
            break;
        }
    }
    if( isChampion )
        champions.Add( vertex );
}
PrintChampions( champions );
```

Listing 5: Finding Champion. Naive algorithm



```
Champions: A,B,D
```

Figure 30: Finding Champion problem solution

3.2

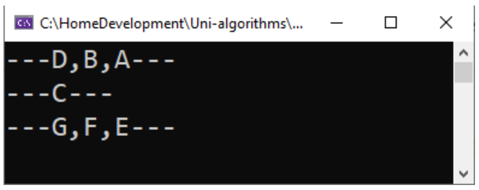
The task of dividing the graph into groups, where each group consists of nodes that have defeated each other, either directly or indirectly, is equivalent to finding strongly connected components (SCC) in the graph. Algorithm for finding SCC:

1. call DFS(G) to compute finishing times u.f for each vertex u - O(V+E)
2. compute G^T - O(V+E)
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing u.f - O(V+E)
4. output the vertices of each tree in the depth-first forest formed in 3 as a separate strongly connected component - O(V)

Listing 6 demonstrates the implementation of the algorithm, and Figure 31 displays the output of the program applying this algorithm to a graph from the assignment. Running time of the algorithm is O(V+E).

```
//DFS
DepthFirstSearch dfs = new(graph);
//G_t
AdjacencySetGraph transposedGraph = new(graph.Vertices, graph.IsDirected);
foreach( VertexBase vertex in graph.Vertices )
{
    foreach( int adjacentVertexId in
        graph.GetAdjacentVertices( vertex.Id ) )
    {
        int weight =
            graph.GetEdgeWeight( vertex.Id, adjacentVertexId );
        transposedGraph.AddEdge( adjacentVertexId, vertex.Id, weight );
    }
}
//DFS G_t
HashSet<int> mainLoopOrder =
    new(dfs.VerticesTimes.OrderByDescending( x => x.FinishTime )
        .Select( x => x.Id ));
dfs = new(transposedGraph, mainLoopOrder);
dfs.PrintDepthFirstForest();
```

Listing 6: Finding Champion. SCC algorithm



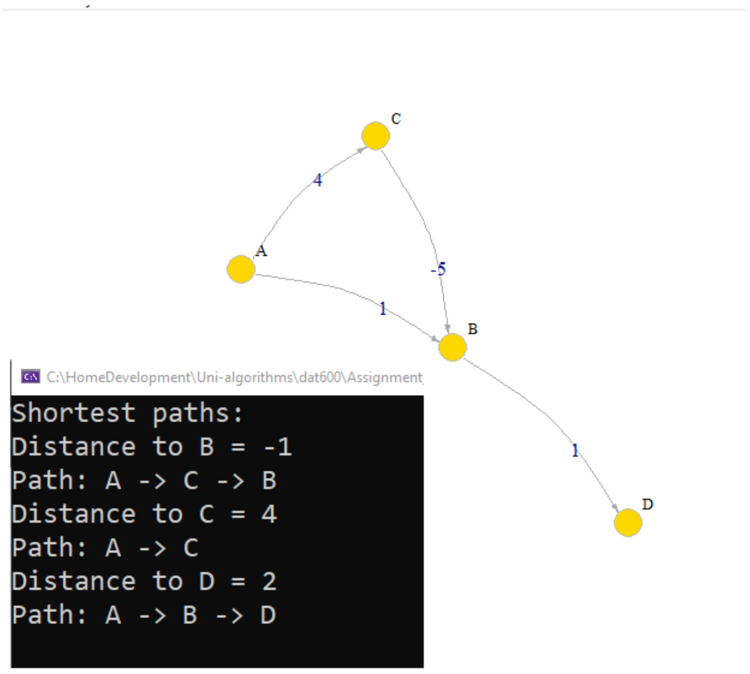
```
---D,B,A---
---C---
---G,F,E---
```

Figure 31: SCC solution

4 Problem Shortest Path

4.1

Figure 32 depicts a graph in which executing Dijkstra's algorithm fails to accurately determine the shortest path lengths from vertex A to every node in the graph. The output of the program, which utilized the implementation of Dijkstra's algorithm, reveals that the shortest path from A to D is the path A - B - D (total distance 2), whereas it is evident that the actual path is A - C - B - D (distance 0). This discrepancy occurs because once vertices are processed, they cannot be reprocessed again if necessary.

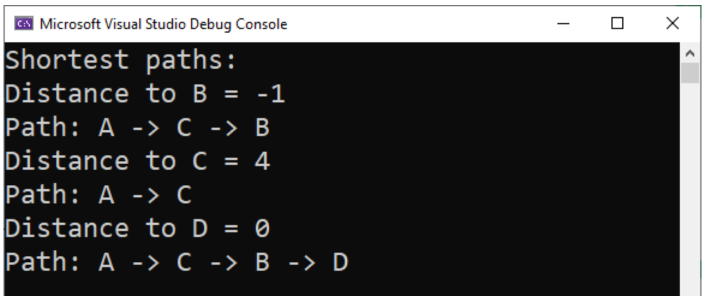


```
C:\HomeDevelopment\Uni-algorithms\dat600\Assignment
Shortest paths:
Distance to B = -1
Path: A -> C -> B
Distance to C = 4
Path: A -> C
Distance to D = 2
Path: A -> B -> D
```

Figure 32: Dijkstra's algorithm. Problem of negative edges

4.2

The Bellman-Ford algorithm is a solution that can address the issue of negative edges. Figure 33 demonstrates the output of the program using this algorithm on the same graph. Now, the shortest path from A to D is correctly found (A - C - B - D with a total distance of 0).



```
Shortest paths:
Distance to B = -1
Path: A -> C -> B
Distance to C = 4
Path: A -> C
Distance to D = 0
Path: A -> C -> B -> D
```

Figure 33: Solution with Bellman-Ford algorithm

5 Problem Maximum Flow

5.1

In the flow network G between $v_1 - v_3$ there is an antiparallel edge issue. This issue can be resolved by adding additional node v_6 between v_1 and v_3. Figure 34 presents the obtained flow netwrok.

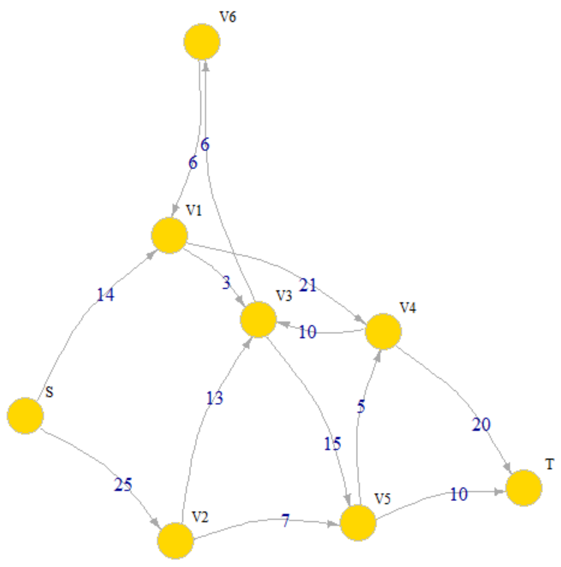


Figure 34: Flow network without antiparallel edges

5.2

In the «Graph» library, the Ford-Fulkerson algorithm has been implemented to solve maximum flow problems. Listing 7 displays the code of the method GetMaximumFlowNetwork, which contains the main logic and returns the maximum flow network in the form of a graph. This algorithm iteratively finds the residual network from the flow network, identifies augmenting paths in it, and, if such paths exist, augments or cancels the flow in the flow network until there are no augmenting paths from the source to the sink in the residual network. Figure 35 illustrates the obtained maximum flow network. By utilizing the graph and calculating the sum of all edges that reach the sink, the maximum flow is determined to be 30 (V_4 -> T + V_5 -> T = 20 + 10 = 30).

```
public GraphBase GetMaximumFlowNetwork()
{
    //init
    AdjacencySetGraph flowNetwork = new(_graph.Vertices, _graph.IsDirected);
    AdjacencySetGraph residualNetwork = new(_graph.Vertices, _graph.IsDirected);
    ModifyResidualNetwork( flowNetwork, residualNetwork );
    //search path
    BreadthFirstSearch bfs = new(residualNetwork, _sourceId);
    while( bfs.IsPath( _sourceId, _targetId ) )
    {
        //find augmenting flow
        int prevId = _targetId;
        int? parentId = bfs.GetParentIdInPath( prevId );
        int? min = null;
        while( parentId.HasValue )
        {
            int tmpWeight = residualNetwork.GetEdgeWeight( parentId.Value, prevId );
            if( !min.HasValue || tmpWeight <= min.Value )
                min = tmpWeight;
            prevId = parentId.Value;
            parentId = bfs.GetParentIdInPath( prevId );
        }
        //augment/cancel
        prevId = _targetId;
        parentId = bfs.GetParentIdInPath( prevId );
        while( parentId.HasValue && min.HasValue )
        {
            if( _graph.GetAdjacentVertices( parentId.Value ).Contains( prevId ) )
            {
                int flow = flowNetwork.GetEdgeWeight( parentId.Value, prevId );
                flowNetwork.RemoveEdge( parentId.Value, prevId );
                flowNetwork.AddEdge( parentId.Value, prevId, flow + min.Value );
            }
            else if( _graph.GetAdjacentVertices( prevId ).Contains( parentId.Value ))
            {
                int flow = flowNetwork.GetEdgeWeight( prevId, parentId.Value );
                flowNetwork.RemoveEdge( prevId, parentId.Value );
                if( flow − min.Value > 0 )
                    flowNetwork.AddEdge( prevId, parentId.Value, flow − min.Value );
            }
            prevId = parentId.Value;
            parentId = bfs.GetParentIdInPath( prevId );
        }
        ModifyResidualNetwork( flowNetwork, residualNetwork );
        bfs.BFS( _sourceId );
    }
    return flowNetwork;
}
```

Listing 7: Ford-Fulkerson algorithm

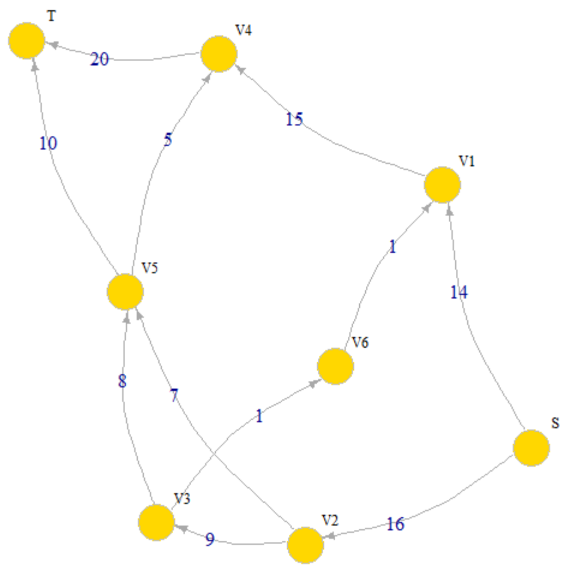


Figure 35: Maximum Flow network

5.3

Bottleneck of the flow network is 30 (Figure 36). Since:

- Capacity of the Cut_0, where $S = \{s\}$ and $T = V - S$ is $C(0) = 14 + 25 = 39$

- $C(1) = 25 + 3 + 21 = 49$, $Cut_1 : S = \{s, V_1\}$, $T = V - S$
- $C(2) = 21 + 3 + 13 + 7 = 44$, $Cut_2 : S = \{s, V_1, V_2\}$, $T = V - S$
- $C(3) = 21 + 6 + 15 + 7 = 49$, $Cut_3 : S = \{s, V_1, V_2, V_3\}$, $T = V - S$
- $C(4) = 20 + 6 + 15 + 7 = 48$, $Cut_4 : S = \{s, V_1, V_2, V_3, V_4\}$, $T = V - S$
- $C(5) = 20 + 10 + 6 = 36$, $Cut_5 : S = \{s, V_1, V_2, V_3, V_4, V_5\}$, $T = V - S$
- $C(6) = 20 + 10 = 30$, $Cut_6 : S = \{s, V_1, V_2, V_3, V_4, V_5, V_6\}$, $T = V - S$

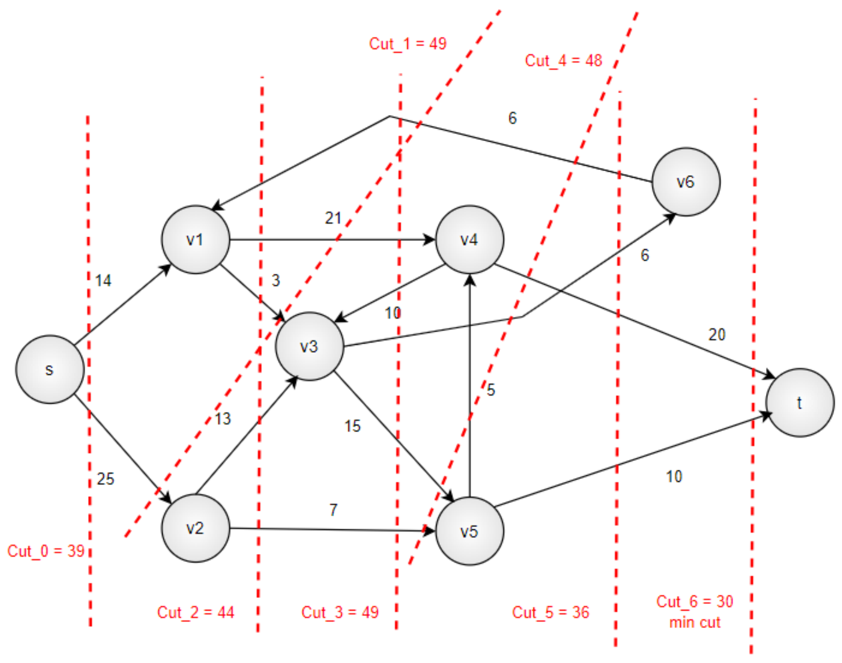


Figure 36: Flow network. Cuts

5.4

If all capacities are integers, then each augmenting path increases the value of |f| by at least 1. If the maximum flow is denoted as f*, then the algorithm requires at most |f*| iterations. Therefore, the time complexity is O(E|f*|).

The Edmonds-Karp Algorithm represents an enhancement of the Ford-Fulkerson algorithm. The sole modification involves choosing the augmenting path through breadth-first search on the residual network. This adjustment significantly accelerates its operation, allowing it to run in much faster polynomial time, specifically $O(V * E^2)$.

References

[1] Github Evgenii Dudkin. — URL: `https://github.com/EvgeneDudkin/dat600` (online; accessed: 20.03.2024).