FEU INSTITUTE OF TECHNOLOGY

Task Scheduling with Random Dependency Graphs (Variable Size Decrease)

CS0007 - ALGORITHMS

Dr. Hadji J. Tejuco

Acuña, Amiel Josiah - Leader
Nuyda, Matthew
Lizardo, Gideon
Viacrusis, Sean Matthew

March 2, 2025

TABLE OF CONTENTS

INTRODUCTION

Task scheduling is a critical challenge in project management, particularly when dependencies between tasks are complex and not predefined. Traditional scheduling methods often assume fixed dependency structures, but in dynamic environments, these relationships can vary unpredictably. According to Gurnov (2024), Successful time management, particularly for professional service businesses, depends on efficient project scheduling. This project addresses this issue by developing a tool that generates random dependency graphs and computes a valid task execution order, offering a practical solution to optimize project workflows.

The main purpose of this project is to design and simulate a task scheduling system that uses a randomly generated directed acyclic graph (DAG) to represent task dependencies. This simulation is significant as it provides insights into handling real-world scheduling problems where task interdependencies are uncertain, aiming to enhance project efficiency and completion timelines.

The objectives of the project are to create a system that generates a random DAG based on user-specified numbers of tasks and dependencies, determines a correct task order using topological sorting, manages exceptions such as cyclic dependencies, and simulates the sorting process step-by-step. The system also aims to ensure robustness by handling invalid inputs and providing clear feedback on the generated graph and task sequence.

The scope includes the simulation of a single DAG with N tasks and M dependencies, topological sorting using Depth-First Search (DFS), and basic error handling for cycles and invalid data. It does not cover advanced scheduling algorithms beyond topological sorting, real-time task execution, or integration with project management software

.

PROJECT OVERVIEW

The project "Task Scheduling with Random Dependency Graphs (Variable Size Decrease)" aims to develop a simulation system that efficiently manages task dependencies in a randomly generated DAG. The system must generate a graph with user-defined parameters, compute a valid task order, and address potential errors such as cycles or invalid inputs.

The key features of "Task Scheduling with Random Dependency Graphs" are:

1. **Random Graph Generation**: Creates a DAG with N tasks and M dependencies based on user input.
2. **Task Ordering**: Uses topological sorting to determine a feasible sequence of tasks.
3. **Step-by-Step Simulation**: Displays the sorting process incrementally for transparency.
4. **Error Management**: Identifies and handles cyclic dependencies and invalid data gracefully.

REQUIREMENTS ANALYSIS

The project was built with the following requirements:

Functional Requirements

1. **Graph Generation**:
   - Accept user inputs for the number of tasks (N) and dependencies (M).
   - Generate a random DAG ensuring no cycles are present.
2. **Task Sequencing**:
   - Implement topological sorting using DFS to order tasks based on dependencies.
   - Simulate the sorting process, showing each task as it is processed.
3. **Error Handling**:
   - Detect cyclic dependencies and notify the user if they occur.
   - Validate inputs to ensure N and M are positive and M does not exceed the maximum possible edges.

Non-Functional Requirements

1. **Performance**:
   - The system should process and sort tasks within a reasonable time frame, targeting O(N + M) complexity.
2. **Reliability**:
   - The system must consistently generate valid DAGs and produce accurate task orders.

3. **Usability**:
    - Output should be clear and easy to understand, detailing the graph and sorted sequence.
4. **Scalability**:
    - The design should allow for future expansion, such as increasing N and M or adding new features.

SYSTEM DESIGN

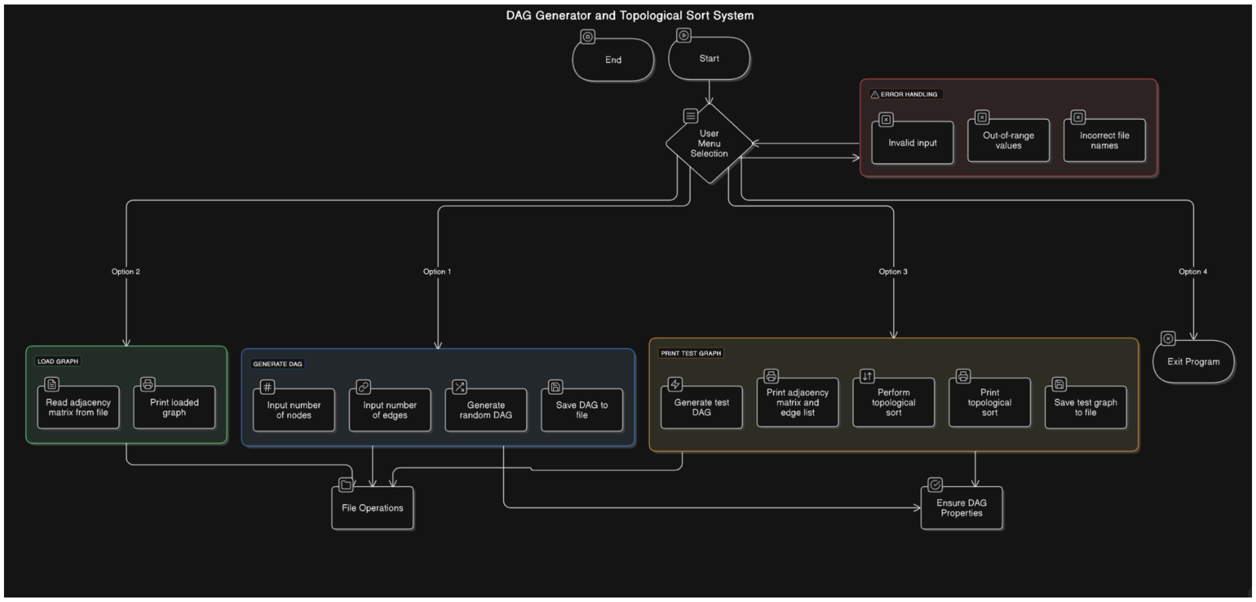


Figure 1. System Design Diagram

IMPLEMENTATION

Technologies Used:

A. Programming Languages

a. C++: The system is designed to be utilized with C++ version 23 standalone.

B. Libraries Used

a. iostream - utilized basic input output functions.

b. stdexcept - utilized for error handling.

c. cstdlib - utilized for randomization

d. ctime - utilized for randomization

C. Development Environment

a. Code::Blocks

TESTING

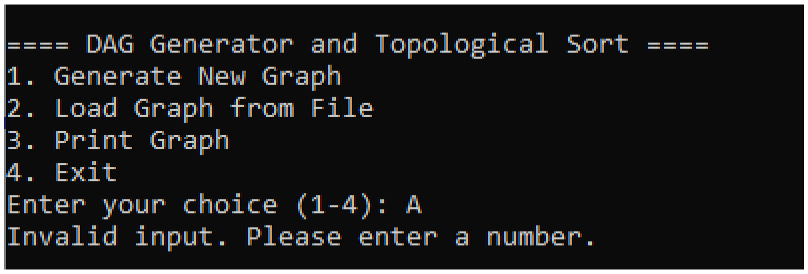
Test Cases: List test cases with inputs, expected outputs, and actual results.

Results: Summarize the outcomes of the testing process.

Test Case 1 (Wrong Input):

Input: A

Output:

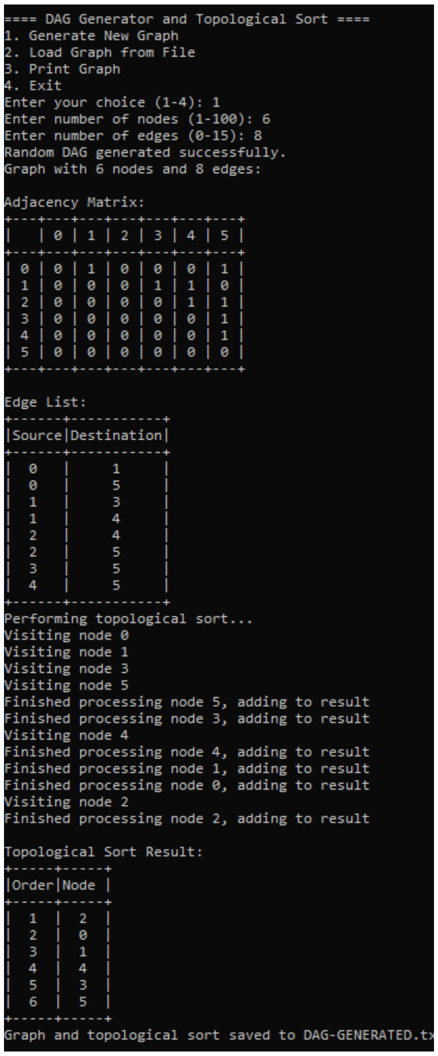


```
==== DAG Generator and Topological Sort ====
1. Generate New Graph
2. Load Graph from File
3. Print Graph
4. Exit
Enter your choice (1-4): A
Invalid input. Please enter a number.
```

Test Case 2 (Generate New Graph):

Input: 1 6 8

Output:

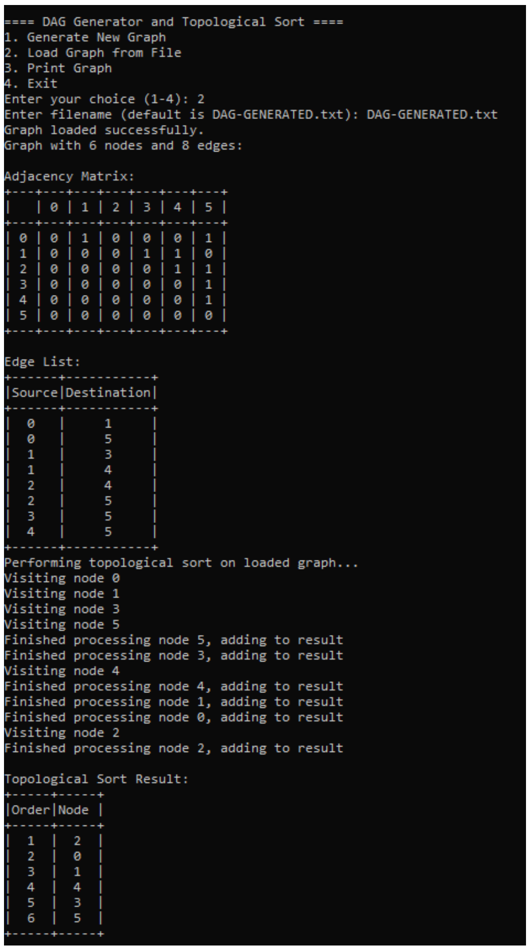


```
==== DAG Generator and Topological Sort ====
1. Generate New Graph
2. Load Graph from File
3. Print Graph
4. Exit
Enter your choice (1-4): 1
Enter number of nodes (1-100): 6
Enter number of edges (0-15): 8
Random DAG generated successfully.
Graph with 6 nodes and 8 edges:

Adjacency Matrix:
+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
| 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 1 |
| 3 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
+---+---+---+---+---+---+---+

Edge List:
+------+-----------+
|Source|Destination|
+------+-----------+
|  0   |     1     |
|  0   |     5     |
|  1   |     3     |
|  1   |     4     |
|  2   |     4     |
|  2   |     5     |
|  3   |     5     |
|  4   |     5     |
+------+-----------+
Performing topological sort...
Visiting node 0
Visiting node 1
Visiting node 3
Visiting node 5
Finished processing node 5, adding to result
Finished processing node 3, adding to result
Visiting node 4
Finished processing node 4, adding to result
Finished processing node 1, adding to result
Finished processing node 0, adding to result
Visiting node 2
Finished processing node 2, adding to result

Topological Sort Result:
+-----+-----+
|Order|Node |
+-----+-----+
|  1  |  2  |
|  2  |  0  |
|  3  |  1  |
|  4  |  4  |
|  5  |  3  |
|  6  |  5  |
+-----+-----+
Graph and topological sort saved to DAG-GENERATED.tx
```

Test Case 3 (Load Graph From File):

Input: 2 DAG-GENERATED.txt

Output:

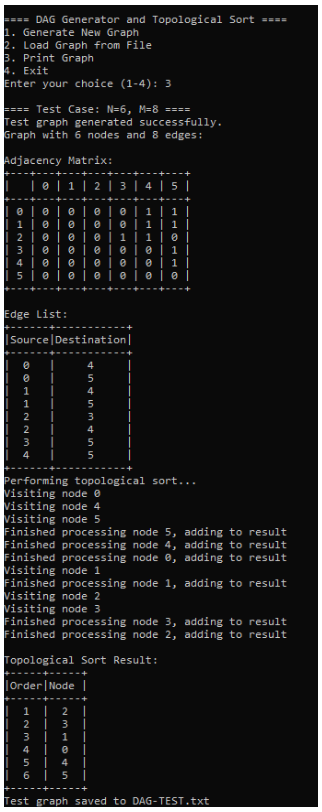


```
==== DAG Generator and Topological Sort ====
1. Generate New Graph
2. Load Graph from File
3. Print Graph
4. Exit
Enter your choice (1-4): 2
Enter filename (default is DAG-GENERATED.txt): DAG-GENERATED.txt
Graph loaded successfully.
Graph with 6 nodes and 8 edges:

Adjacency Matrix:
+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
| 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 1 |
| 3 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
+---+---+---+---+---+---+---+

Edge List:
+------+-----------+
|Source|Destination|
+------+-----------+
|  0   |     1     |
|  0   |     5     |
|  1   |     3     |
|  1   |     4     |
|  2   |     4     |
|  2   |     5     |
|  3   |     5     |
|  4   |     5     |
+------+-----------+
Performing topological sort on loaded graph...
Visiting node 0
Visiting node 1
Visiting node 3
Visiting node 5
Finished processing node 5, adding to result
Finished processing node 3, adding to result
Visiting node 4
Finished processing node 4, adding to result
Finished processing node 1, adding to result
Finished processing node 0, adding to result
Visiting node 2
Finished processing node 2, adding to result

Topological Sort Result:
+-----+-----+
|Order|Node |
+-----+-----+
|  1  |  2  |
|  2  |  0  |
|  3  |  1  |
|  4  |  4  |
|  5  |  3  |
|  6  |  5  |
+-----+-----+
```

Test Case 4 (Print Graph):

Input: 3

Output:

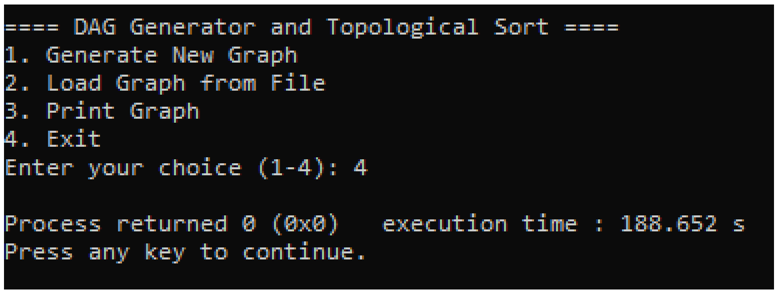


```
==== DAG Generator and Topological Sort ====
1. Generate New Graph
2. Load Graph from File
3. Print Graph
4. Exit
Enter your choice (1-4): 3

==== Test Case: N=6, M=8 ====
Test graph generated successfully.
Graph with 6 nodes and 8 edges:

Adjacency Matrix:
+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |
+---+---+---+---+---+---+---+

Edge List:
+------+-----------+
|Source|Destination|
+------+-----------+
|  0   |     4     |
|  0   |     5     |
|  1   |     4     |
|  1   |     5     |
|  2   |     3     |
|  2   |     4     |
|  3   |     5     |
|  4   |     5     |
+------+-----------+
Performing topological sort...
Visiting node 0
Visiting node 4
Visiting node 5
Finished processing node 5, adding to result
Finished processing node 4, adding to result
Finished processing node 0, adding to result
Visiting node 1
Finished processing node 1, adding to result
Visiting node 2
Visiting node 3
Finished processing node 3, adding to result
Finished processing node 2, adding to result

Topological Sort Result:
+-----+-----+
|Order|Node |
+-----+-----+
|  1  |  2  |
|  2  |  3  |
|  3  |  1  |
|  4  |  0  |
|  5  |  4  |
|  6  |  5  |
+-----+-----+
Test graph saved to DAG-TEST.txt
```

Test Case 5 (Exit):

Input: 4

Output:



```
==== DAG Generator and Topological Sort ====
1. Generate New Graph
2. Load Graph from File
3. Print Graph
4. Exit
Enter your choice (1-4): 4

Process returned 0 (0x0)   execution time : 188.652 s
Press any key to continue.
```

USER MANUAL

INSTALLATION GUIDE

Prerequisites:

Ensure you have a working C++ compiler installed on your device. The following compilers are recommended:

- Online GDB
- Dev-C++
- Code::Blocks

Downloading the Code

1. Visit the GitHub repository: https://github.com/MatthewNuyda/CS0007-ACT3-GROUP8/blob/main/DAG/DAG-ACTIVITY3-TN22G8.cpp
2. Click on **Raw** and right-click **Save As…** to download the DAG-ACTIVITY3-TN22G8.cpp file.
3. Save the file to your desired directory

Compiling the Code

A. Using Online GDB

1. Visit the Online GDB C++ compiler website: https://www.onlinegdb.com/online_c++_compiler
2. Paste the entire code into the editor.
3. Click Run.

B. **Using Dev-C++**

    1. Open **Dev-C++.**
    2. Click **File > Open,** then select DAG-ACTIVITY3-TN22G8.cpp.
    3. Click **Execute > Compile & Run.**

C. **Using Code::Blocks**

    1. Open **Code::Blocks.**
    2. Click **File > Open,** then select DAG-ACTIVITY3-TN22G8.cpp.
    3. Click **Build > Build and Run.**

USAGE INSTRUCTIONS

Program Overview

This program generates a Directed Acyclic Graph (DAG) and performs Topological Sorting using Depth First Search (DFS). Users can:

- Generate a new DAG with a specified number of nodes and edges.
- Load an existing graph from a file.
- Display the adjacency matrix and edge list.
- Perform and display topological sorting.

Running the Program

1. Launch the compiled executable.
2. The main menu will appear with the following options:
    - 1. Generate New Graph
    - 2. Load Graph from File
    - 3. Print Graph
    - 4. Exit

3. Choose an option by entering the corresponding number.

Generating a New Graph

1. Select Option 1: Generate New Graph.
2. Enter the number of nodes (between 1 and 100).
3. Enter the number of edges (between 0 and the maximum possible edges for the given nodes).
4. The program will generate a random DAG.
5. The graph will be displayed in Adjacency Matrix and Edge List format.
6. The program will automatically perform Topological Sorting.

Loading a Graph from a File

1. Select Option 2: Load Graph from File.
2. Enter the filename of the saved graph.
3. If successful, the program will load the graph and display it.

Displaying the Graph

1. Select Option 3: Print Graph.
2. The program will display:
    - Adjacency Matrix
    - Edge List
    - Topological Sort Result

Saving the Graph

After generating a graph, it can be saved using the built-in function:

- The graph will be stored in a .txt file containing the adjacency matrix, edge list, and topological sort result.

Exiting the Program

1. Select Option 4: Exit.
2. The program will terminate.

CHALLENGES AND SOLUTIONS

Challenges Faced

Generating a random DAG posed difficulties in ensuring acyclic properties, as random edge additions could inadvertently create cycles. Detecting and managing these cycles efficiently was another hurdle, alongside explaining the variable size decrease concept inherent in topological sorting.

Solutions Implemented

To maintain a DAG, the system checks each potential edge for cycles before inclusion, using a DFS-based approach with a recursion stack to verify acyclicity. If a cycle is detected during sorting, the system halts and reports an error, ensuring invalid schedules are avoided. This addresses the guide question: *How will you ensure that the randomly generated graph is a Directed Acyclic Graph (DAG) and does not contain cycles?* and *What happens if the graph* *contains a cycle? How will you detect and handle this situation?* For variable size decrease, topological sorting reduces the problem by processing one task and its dependencies per step, with the reduction varying by the number of unvisited dependent tasks, answering: *Why is Topological Sorting an example of variable size decrease? How does the reduction pattern vary at each step?*

FUTURE ENHANCEMENTS

Potential Improvements

Future iterations could incorporate weighted dependencies, where tasks have varying durations, using a greedy DFS approach to prioritize shorter or critical tasks, addressing: *How would you modify the program to handle weighted dependencies (e.g., tasks that take different amounts of time to complete)?* Additional enhancements might include a visual interface to display the DAG and sorting process or support for parallel task execution to optimize schedules further.

CONCLUSIONS

Summary

The project successfully developed a task scheduling simulation that generates random DAGs and computes valid task orders using topological sorting. It handles errors effectively and provides a clear sequence for N=6 tasks and M=8 dependencies, fulfilling its objectives and offering a foundation for real-world scheduling applications.

Lessons Learned

The importance of cycle prevention in random graph generation was a key takeaway, alongside the effectiveness of DFS in topological sorting. Understanding variable size decrease deepened our appreciation for algorithmic problem reduction, and robust error handling proved essential for system reliability.

REFERENCES

Gurnov, A. (2024, August 21). *What Is Scheduling in Project Management*. Wrike. Retrieved March 2, 2025, from https://www.wrike.com/project-management-guide/faq/what-is-scheduling-in-project-management/

Jain, S. (2025, January 23). *Topological Sorting*. GeeksforGeeks. Retrieved March 2, 2025, from https://www.geeksforgeeks.org/topological-sorting/

Mount, D. (2003). *COMP 271: Design and Analysis of Algorithms*. Lecture notes - COMP 271. Retrieved March 2, 2025, from http://home.cse.ust.hk/faculty/golin/COMP271Sp03/Notes/

APPENDICES

Appendix A: Additional diagrams or charts.

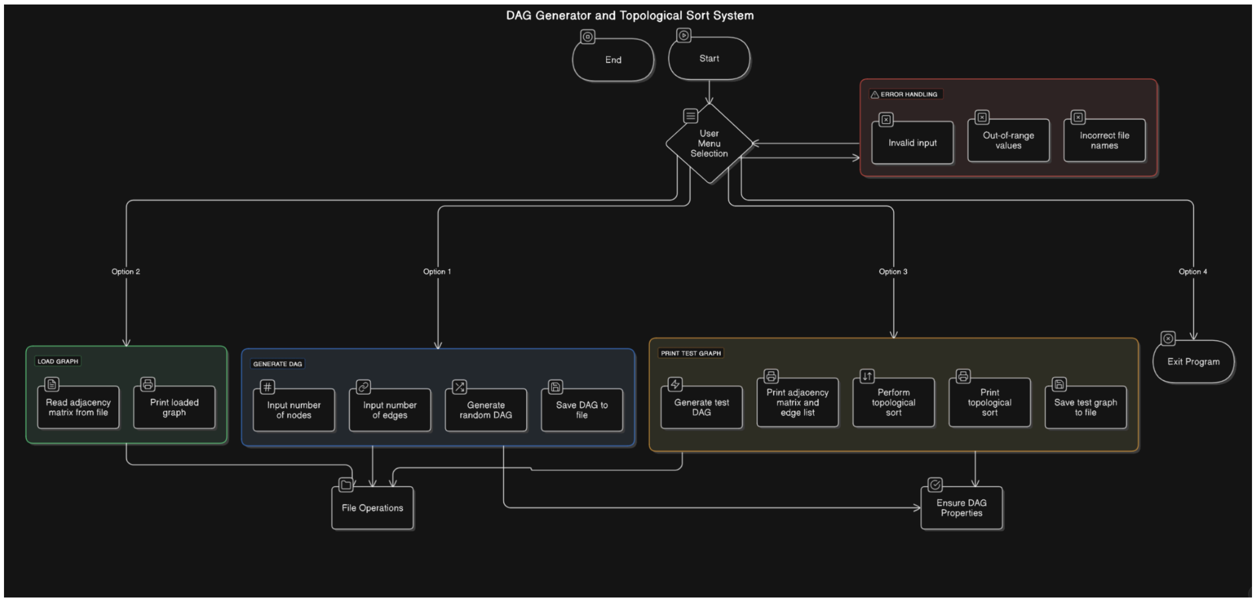


Figure 1. System Design Diagram

Appendix B: Complete Code

```
#include <iostream> //used for input output

#include <fstream> //used for file handling

#include <cstdlib> //used for randomization

#include <ctime>  //used for randomization


using namespace std;


class Graph {

private:

    int n;                  // Number of nodes

    int m;                  // Number of dependencies

    bool** adjMatrix;       // Checker for adjacency

    int* visited;           // Depth First Search Visited Checking

    int* result;            // Store Sort result

    int resultIndex;        // Index for result array


public:

    // Constructor

    Graph(int nodes) {

        if (nodes <= 0 || nodes > 100) {

            cout << "Error: Invalid number of nodes. Must be between 1 and 100" << endl;

            exit(1);

        }
        n = nodes;

        m = 0;

        resultIndex = 0;


        // Construct the adjacency matrix

        adjMatrix = new bool*[n];

        for (int i = 0; i < n; i++) {

            adjMatrix[i] = new bool[n];

            for (int j = 0; j < n; j++) {

                adjMatrix[i][j] = false;

            }

        }

        //construct visited and result array

        visited = new int[n];

        result = new int[n];


        // Initialize visited array

        for (int i = 0; i < n; i++) {

            visited[i] = 0;

        }

    }


    // Destructor
```

```
~Graph() {

    for (int i = 0; i < n; i++) {

        delete[] adjMatrix[i];

    }

    delete[] adjMatrix;

    delete[] visited;

    delete[] result;

}



// function to add edge

void addEdge(int u, int v) {

    if (u < 0 || u >= n || v < 0 || v >= n) {

        cout << "Error: Node index out of range." << endl;

        return;

    }


    if (!adjMatrix[u][v]) {

        adjMatrix[u][v] = true;

        m++;

    }

}


//DAG Generator using stime
```

```
    bool generateRandomDAG(int edges) {
        if (edges < 0 || edges > n * (n - 1) / 2) {
            cout << "Error: Invalid number of edges. Must be between 0 and " << n * (n - 1) / 2 <<
endl;
            return false;
        }

        // Limit edges to 100
        if (edges > 100) {
            cout << "Warning: Requested edges (" << edges << ") exceeds maximum allowed (100)
edges." << endl;
            edges = 100;
        }

        // Clear existing edges
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                adjMatrix[i][j] = false;
            }
        }
        m = 0;

        //make sure that the graph made is acyclic low node -> high node
        int attempts = 0;
        while (m < edges && attempts < 1000) {
            int u = rand() % n;
            int v = rand() % n;

            // Ensure v > u to maintain acyclicity
            if (v > u && !adjMatrix[u][v]) {
                addEdge(u, v);
            }

            attempts++;

            // Just incase
            if (attempts >= 1000 && m < edges) {
                     cout << "Warning: Could only generate " << m << " edges out of " << edges << "
requested." << endl;
                break;
            }
        }

        return true;
    }

    // DFS for topological sort
    void DFS(int node, bool* onStack, bool printSteps) {
```

```
    visited[node] = 1;
    onStack[node] = true;

    if (printSteps) {
        cout << "Visiting node " << node << endl;
    }

    // Check all neighboring nodes
    for (int i = 0; i < n; i++) {
        if (adjMatrix[node][i]) {
            if (!visited[i]) {
                DFS(i, onStack, printSteps);
            } else if (onStack[i]) {
                 cout << "Error: Cycle detected in the graph! Path includes nodes: " << node << " ->
" << i << endl;
                    cout << "This graph is not a DAG. Topological sort is only possible on DAGs." <<
endl;
                exit(1);
            }
        }
    }

    // Add node to result in reverse order (for topological sort)
    result[n - 1 - resultIndex++] = node;
    onStack[node] = false;


    if (printSteps) {
        cout << "Finished processing node " << node << ", adding to result" << endl;
    }
}


// Topological Sort using DFS
void topologicalSort(bool printSteps) {
    // Reset visited array and result index
    for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }
    resultIndex = 0;


    // track node existing in stack to track if its a cyclic graph
    bool* onStack = new bool[n];
    for (int i = 0; i < n; i++) {
        onStack[i] = false;
    }


    // Perform DFS on all unvisited nodes
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            DFS(i, onStack, printSteps);
        }
    }

    delete[] onStack;
}

// Print the graph
void printGraph() {
    cout << "Graph with " << n << " nodes and " << m << " edges:" << endl;



    cout << "\nAdjacency Matrix:" << endl;
    int colWidth = 3;

    // top
    cout << "+";
    for (int i = 0; i < n + 1; i++) {
        for (int j = 0; j < colWidth; j++) {
            cout << "-";
        }
        cout << "+";
    }
    cout << endl;
```

```
// header & col index
cout << "|" << string(colWidth, ' ') << "|";
for (int i = 0; i < n; i++) {
    cout << string((colWidth - 1) / 2, ' ') << i
         << string(colWidth - ((colWidth - 1) / 2) - 1, ' ') << "|";
}
cout << endl;

// divider
std::cout << "+";
for (int i = 0; i < n + 1; i++) {
    for (int j = 0; j < colWidth; j++) {
        std::cout << "-";
    }
    std::cout << "+";
}
std::cout << std::endl;

// Print matrix rows
for (int i = 0; i < n; i++) {
    // Print row index
    std::cout << "|" << std::string((colWidth - 1) / 2, ' ') << i
              << std::string(colWidth - ((colWidth - 1) / 2) - 1, ' ') << "|";
    // Print row values
    for (int j = 0; j < n; j++) {
        std::cout << std::string((colWidth - 1) / 2, ' ')
                  << (adjMatrix[i][j] ? "1" : "0")
                  << std::string(colWidth - ((colWidth - 1) / 2) - 1, ' ') << "|";
    }
    std::cout << std::endl;
}

// bottom
std::cout << "+";
for (int i = 0; i < n + 1; i++) {
    for (int j = 0; j < colWidth; j++) {
        std::cout << "-";
    }
    std::cout << "+";
}
std::cout << std::endl;

//Print Edges
cout << "\nEdge List:" << endl;
if (m == 0) {
    cout << "No edges in the graph." << endl;
    return;
}

int srcWidth = 6;
int destWidth = 11;

// top
cout << "+" << string(srcWidth, '-') << "+"
        << string(destWidth, '-') << "+" << endl;

// header
cout << "|" << "Source" << "|" << "Destination" << "|" << endl;

// divider
std::cout << "+" << std::string(srcWidth, '-') << "+"
          << std::string(destWidth, '-') << "+" << std::endl;

// Print edges
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (adjMatrix[i][j]) {
            std::cout << "|" << std::string((srcWidth - 1) / 2, ' ') << i
                      << std::string(srcWidth - ((srcWidth - 1) / 2) - 1, ' ') << "|"
                      << std::string((destWidth - 1) / 2, ' ') << j
```

```
                    << std::string(destWidth - ((destWidth - 1) / 2) - 1, ' ') << "|" << std::endl;
            }
        }
    }

    // bottom
    std::cout << "+" << std::string(srcWidth, '-') << "+"
              << std::string(destWidth, '-') << "+" << std::endl;
}

// Print the topological sort
void printTopologicalSort() {
    cout << "\nTopological Sort Result:" << endl;

    int colWidth = 5;  // Minimum width for "Order" header
    int nodeWidth = 5; // Minimum width for "Node" header

    // top
    std::cout << "+" << std::string(colWidth, '-') << "+"
              << std::string(nodeWidth, '-') << "+" << std::endl;

    // header
    std::cout << "|" << "Order" << "|" << "Node " << "|" << std::endl;
    // divider
    std::cout << "+" << std::string(colWidth, '-') << "+"
              << std::string(nodeWidth, '-') << "+" << std::endl;

    // results
    for (int i = 0; i < n; i++) {
        std::cout << "|" << std::string((colWidth - 1) / 2, ' ') << i + 1
                  << std::string(colWidth - ((colWidth - 1) / 2) - 1, ' ') << "|"
                  << std::string((nodeWidth - 1) / 2, ' ') << result[i]
                  << std::string(nodeWidth - ((nodeWidth - 1) / 2) - 1, ' ') << "|" << std::endl;
    }

    // bottom
    std::cout << "+" << std::string(colWidth, '-') << "+"
              << std::string(nodeWidth, '-') << "+" << std::endl;
}

// Save to File
bool saveToFile(const char* filename) {
    std::ofstream file(filename);

    if (!file.is_open()) {
        cout << "Error: Unable to open file for writing." << endl;
        return false;
    }

    // Write number of nodes and edges
    file << "Graph with " << n << " nodes and " << m << " edges" << std::endl;

    // Write adjacency matrix
    file << "\nAdjacency Matrix:" << std::endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            file << (adjMatrix[i][j] ? "1" : "0") << " ";
        }
        file << std::endl;
    }

    // Write edge list
    file << "\nEdge List:" << std::endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (adjMatrix[i][j]) {
                file << i << " -> " << j << std::endl;
            }
        }
    }
```

```
    // Write topological sort result
    file << "\nTopological Sort Result: ";
    for (int i = 0; i < n; i++) {
        file << result[i] << " ";
    }
    file << std::endl;

    file.close();
    return true;
}

// Load the graph from a file
bool loadFromFile(const char* filename) {
    std::ifstream file(filename);

    if (!file.is_open()) {
        cout << "Error: Unable to open file for reading." << endl;
        return false;
    }

    string line;

    // Skip graph info
    std::getline(file, line);
    // Skip adjacency matrix header
    std::getline(file, line);
    std::getline(file, line);

    // Read adjacency matrix
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            char c;
            file >> c;
            adjMatrix[i][j] = (c == '1');
            if (adjMatrix[i][j]) m++;
        }
    }

    file.close();
    return true;
}

// Get number of nodes
int getNodeCount() {
    return n;
}
```

```
    // Get number of edges
    int getEdgeCount() {
        return m;
    }
};

// Function to get integer input with validation
int getInteger(const char* prompt, int min, int max) {
    int value;
    bool valid = false;

    do {
        cout << prompt;
        if (cin >> value) {
            if (value >= min && value <= max) {
                valid = true;
            } else {
                cout << "Value must be between " << min << " and " << max << endl;
            }
        } else {
            cout << "Invalid input. Please enter an integer." << endl;
            cin.clear();
            cin.ignore(10000, '\n');
        }
    } while (!valid);

    return value;
}

// Main function
int main() {
    // Seed random number generator
    srand(time(NULL));

    int choice;
    bool exit = false;

    // Main menu loop
    while (!exit) {
        cout << "\n==== DAG Generator and Topological Sort ====" << endl;
        cout << "1. Generate New Graph" << endl;
        cout << "2. Load Graph from File" << endl;
        cout << "3. Print Graph" << endl;
        cout << "4. Exit" << endl;
        cout << "Enter your choice (1-4): ";
        cin >> choice;

        if (cin.fail()) {
            cout << "Invalid input. Please enter a number." << endl;
            cin.clear();
            cin.ignore(10000, '\n');
            continue;
        }

        switch (choice) {
            case 1: {
                // Generate new graph
                int n = getInteger("Enter number of nodes (1-100): ", 1, 100);
                int maxEdges = n * (n - 1) / 2;  //middle aka maximum number of edges
                if (maxEdges > 100) maxEdges = 100;  // Apply 100 edge limit
                 int m = getInteger(("Enter number of edges (0-" + to_string(maxEdges) + "): ").c_str(),
0, maxEdges);

                // Create new graph
                Graph graph(n);

                // Generate random DAG
                if (graph.generateRandomDAG(m)) {
                    cout << "Random DAG generated successfully." << endl;

                    // Print the graph
                    graph.printGraph();
                    cout << "Performing topological sort..." << endl;
                    graph.topologicalSort(true);  // true to show the process
                    graph.printTopologicalSort();

                    // save to DAG-GENERATED.txt
                    if (graph.saveToFile("DAG-GENERATED.txt")) {
                        cout << "Graph and topological sort saved to DAG-GENERATED.txt" << endl;
                    } else {
                        cout << "Failed to save to file." << endl;
                    }
                } else {
                    cout << "Failed to generate random DAG." << endl;
                }
                break;
            }
            case 2: {
                // Load graph from file
                char filename[100];
                cout << "Enter filename (default is DAG-GENERATED.txt): ";
                cin >> filename;

                // Create graph with default size (can be adjusted after loading)
                Graph graph(6);  // 6 is just an example default size

                if (graph.loadFromFile(filename)) {
                    cout << "Graph loaded successfully." << endl;
                    graph.printGraph();

                    // Perform topological sort on the loaded graph
                    cout << "Performing topological sort on loaded graph..." << endl;
                    graph.topologicalSort(true);
                    graph.printTopologicalSort();
                } else {
                    cout << "Failed to load graph." << endl;
                }
                break;
            }
            case 3: {
                // Create a test graph to display
                cout << "\n==== Test Case: N=6, M=8 ====" << endl;
                Graph testGraph(6);
                if (testGraph.generateRandomDAG(8)) {
                    cout << "Test graph generated successfully." << endl;
                    testGraph.printGraph();
                    cout << "Performing topological sort..." << endl;
                    testGraph.topologicalSort(true);
```

```
                testGraph.printTopologicalSort();


                // Save test graph to file
                if (testGraph.saveToFile("DAG-TEST.txt")) {
                    cout << "Test graph saved to DAG-TEST.txt" << endl;
                }
            } else {
                cout << "Failed to generate test graph." << endl;
            }
            break;
        }
        case 4: {
            // Exit
            exit = true;
            break;
        }
        default: {
            cout << "Invalid choice. Please enter a number between 1 and 4." << endl;
            break;
        }
    }
}

return 0;
```

```
}
```

Appendix C: Any other relevant material.

https://github.com/MatthewNuyda/CS0007-ACT3-GROUP8/tree/main

https://github.com/TechTutorialHub/AlgorithmCodes/tree/main