



**FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY**

**GROUP PROJECT REPORT**

**CSC4202-1 (DESIGN AND ANALYSIS OF ALGORITHMS)**

**SEMESTER II 2022/2023**

**PROGRAMME :**

**BACHELOR OF COMPUTER SCIENCE WITH HONORS**

---

**COURSE : DESIGN AND ANALYSIS OF ALGORITHMS**

**COURSE CODE : CSC4202**

**GROUP : 1**

**ONLINE PORTFOLIO LINK: <https://github.com/Ramesh260402/CSC4202-G1-ProjectReport>**

**GROUP MEMBERS:**

NAME	MATRIC NUM
RAMESH ARVIN A/L ANPALAGAN	213111
PONMUGILLAN A/L MANI	210301
NIK QISTINA NURIN BINTI NIK SHAFUL ANWAR	210839
NOR SYAKILA BINTI SALIM	212927

<b>1.0 Original Scenario.....</b>	<b>3</b>
<b>2.0 Development of A Model.....</b>	<b>3</b>
<b>3.0 Importance of Optimal Solution.....</b>	<b>4</b>
<b>4.0 Suitability of Algorithms As Solution Paradigm (Specification of Algorithm).....</b>	<b>5</b>
3.1 Sorting.....	5
3.2 Divide and Conquer (DAC).....	5
3.3 Dynamic Programming (DP).....	6
3.4 Greedy Algorithms.....	6
3.5 Graph Algorithms.....	6
3.6 Final Choice of Algorithm.....	7
3.7 Consideration.....	8
3.7.1 Implementing Dynamic Programming.....	8
3.7.2 Sample of Implementation.....	9
<b>4.0 Design The Algorithm.....</b>	<b>10</b>
4.1 Topological Sorting.....	10
4.1.1 Kahn's algorithm for topological sorting (Python).....	10
<b>5.0 Analysis of The Algorithm's Correctness.....</b>	<b>13</b>
5.1 Topological Sorting.....	13
5.1.1 Kahn's algorithm for topological sorting (Python).....	13
<b>8.0 Implementation of An Algorithm.....</b>	<b>15</b>
<b>9.0 Program Testing.....</b>	<b>15</b>
<b>10.0 Conclusion.....</b>	<b>17</b>
<b>11.0 Online Portfolio.....</b>	<b>18</b>
<b>12.0 References.....</b>	<b>18</b>
<b>13.0 Project Progress.....</b>	<b>20</b>
13.1 Week 10 , Milestone 1.....	20
13.2 Week 11 , Milestone 2.....	21
13.3 Week 12 , Milestone 3.....	22
13.4 Week 13 , Milestone 4.....	23
13.5 Week 14 , Milestone 6.....	24

## 1.0 Original Scenario

### Optimization of University Course Scheduling

Scenario: Timetabling issues can occur in a range of industries, such as healthcare, sports, transportation, and education. Here, we concentrate on a particular subset of scheduling issues known as the university course scheduling issue. This issue is frequently experienced in a lot of colleges around the world. Teacher assignment, course scheduling, class-teacher timetabling, student scheduling, and classroom assignment are the five sub-problems that can be further divided into.

The scheduling of students is the topic of this project. Imagine you are a student at university who has to come up with the **optimal course schedule** or best course plan for the forthcoming semester. The institution offers a wide range of courses with **various start times, requirements, and seat capacity**.

The number of desirable courses should be maximized while avoiding timing conflicts and fulfilling prerequisite requirements. An organized schedule ought to guarantee an equal distribution of professor and student groups. The planning of the class schedule is a crucial and challenging undertaking. Because there are so many courses, manual scheduling ultimately results in conflicts of various types, and its flaws become more obvious. The drawbacks of manual scheduling will be remedied by computer scheduling as computer science and technology advance.

## 2.0 Development of A Model

Data Type:

The data type required to build the scenario using Kahn's topological sorting approach is a directed graph, which is how the courses and their dependencies are often represented. Each course would be represented as a node in the network, and the connections between them as directed edges.

Objective Function:

In this instance, the aim function would be developing a suitable course schedule that fits with prerequisites and credit criteria. Students should be able to go through the curriculum without exceeding their allowed credits or failing any requirements if the courses are designed in this way.

Constraints:

1. Prerequisite Constraints: Each course may have one or more requirements, which means that it must be taken after passing a particular prior course. The created schedule must adhere to these restrictions.

2. Credit Restrictions: A student is only permitted to enroll in a certain number of credits during any given semester. These credit restrictions should be followed by the schedule, ensuring that each semester's total credits do not go over the set number.

Examples:

1. Prerequisite Restrictions: Enrollment in Course B requires successful completion of Course A. The schedule calls for taking Course A before Course B.

2. Credit Constraints: Each semester, a maximum of 15 credits may be accepted. The maximum number of credits for each semester should be 15, hence the course distribution should be done in that fashion.

Topological sorting (Kahn's algorithm), which can be established as a viable model for resolving the course scheduling problem, can be developed by taking into account these restrictions, aims, and other needs.

### **3.0 Importance of Optimal Solution**

1. *Maximizing Course Availability*

Students who have the finest course schedule can enroll in the most appropriate courses. When the course offerings are well-organized, students have a better chance of finding the courses they need or want to take, which leads to a more rewarding educational experience.

2. *Avoiding Timing Conflicts*

Scheduling issues may arise when a student wants to enroll in two or more courses, yet there are concurrent offerings of those courses. By optimizing the course schedule, allowing students more opportunity to select their own classes, and reducing the need for time-consuming changes or compromises, conflicts can be minimized.

3. *Fulfilling Prerequisite Requirements*

Some courses have prerequisites, which means that students must successfully complete lower-level courses before enrolling in higher-level ones. These needs are taken into account when designing the course programme, ensuring that students can fulfill the requirements rationally and successfully.

4. *Efficient Resource Allocation*

To make the best use of resources like classrooms, lecturers, and teaching assistants, the course schedule can be adjusted. By equally distributing these resources among different courses and time slots, the institution may make the most use of its resources and avoid any potential bottlenecks or inefficiencies.

#### 5. *Improved Student and Professor Satisfaction*

When the course schedule is organized, both students and professors feel less stressed and irritated. Students can ensure a balanced workload and lessen conflicts between their courses by effectively organizing their semester. Professors gain from a fair distribution of student groups because it makes for more manageable class sizes and improves learning and participation.

#### 6. *Time and Cost Savings*

Manual scheduling can be time-consuming and error-prone when dealing with several courses, students, and constraints. Automating the scheduling process and choosing the best option can save a lot of time and effort for students, teachers, and support staff. As a result, the institution could be able to make financial savings.

The optimal course schedule must be chosen in the context of university course scheduling in order to maximize course availability, minimize conflicts, fulfill requirements, allocate resources effectively, boost satisfaction, and save time and money. By applying computer scheduling and optimisation methodologies, the drawbacks of manual scheduling can be eliminated, leading to a more effective and straightforward scheduling procedure.

### **4.0 Suitability of Algorithms As Solution Paradigm (Specification of Algorithm)**

#### **3.1 Sorting**

##### 1. *Strengths*

Sorting can be helpful for some aspects of the course scheduling issue, such ordering the courses according to prerequisites or arranging them according to factors like course credits or semester availability.

##### 2. *Weaknesses*

The course scheduling problem is extremely complex, and sorting alone cannot solve it because it doesn't take prerequisites or credit requirements into account. To make sure that the sorted schedule complies with the restrictions, it could be necessary to use extra algorithms or methods.

#### **3.2 Divide and Conquer (DAC)**

##### 1. *Strengths*

The course scheduling issue could be divided into smaller subproblems using Strengths DAC, with each sub problem's solution being combined after it has been solved. When dealing with intricate dependence systems, this strategy can be useful.

##### 2. *Weaknesses*

The qualifications or credit constraints might not be immediately addressed by DAC, thus further measures would be necessary to make sure the whole schedule complies with

the criteria. It can be difficult to manage the combination of solutions and keep consistency across different subproblems.

### 3.3 Dynamic Programming (DP)

1. *Strengths*

DP can be helpful when there are overlapping subproblems and ideal substructure qualities. It might be able to solve the issue of course scheduling by taking dependencies and credits into account, creating a schedule incrementally, and identifying the best approach.

2. *Weaknesses*

It may be necessary to define an appropriate state and recursive connection that reflects the course dependencies and credit constraints because DP can be computationally expensive for high input sizes. Managing real-time updates or scheduling adjustments can also be difficult.

### 3.4 Greedy Algorithms

1. *Strengths*

Greedy algorithms can offer straightforward and effective solutions by selecting options that are locally optimal at each stage. A greedy solution to the course scheduling problem can entail choosing the courses according to a set of standards, like the quantity of prerequisites or credits.

2. *Weaknesses*

When taking into account complex constraints like requirements and credit constraints, greedy algorithms may not always ensure an ideal overall result. The greedy algorithm's locally optimal decisions could result in inefficient or impractical scheduling.

### 3.5 Graph Algorithms

1. *Strengths*

The relationships and prerequisites in the problem of scheduling courses can be directly addressed by graph algorithms like topological sorting or shortest path algorithms. They offer a realistic depiction for simulating the connections between courses, and they can aid in creating a schedule that complies with the restrictions.

2. *Weaknesses*

Credit constraints might not be taken into account by graph algorithms by default, therefore additional procedures would be necessary to include them in the scheduling process. To guarantee the accuracy and effectiveness of the timetable, they can also call for careful consideration of edge cases and uncommon circumstances.

There are several methods that could be used to resolve the course scheduling problem. With the help of graph algorithms like topological sorting, prerequisites and dependencies may be managed, and additional constraints or scheduling process optimisation can be accomplished with the help of sorting, DAC, DP, or greedy algorithms. The technique selected

will be influenced by the particular needs, problem complexity, input data that is accessible, and performance variables.

### 3.6 Final Choice of Algorithm

To execute our scenario, we decided to use topological sorting graph methods. This is because the Course Scheduling problem, which takes into account prerequisites and credit limits, is amenable to graph algorithms, particularly topological sorting. The following justifies why topological sorting is a viable paradigm for solving this issue:

1. *Prerequisite dependencies*

Prerequisites, where some courses must be completed before taking others, are a fundamental component of the course scheduling problem. Dependencies between nodes in a graph are handled by graph algorithms like topological sorting. Topological sorting effectively determines the order in which the courses should be taken by portraying the courses and requirements as nodes and edges in a directed graph.

2. *Credit constraints*

When planning a course schedule, credit constraints are just as important as prerequisites. Credit constraints are not directly addressed by topological sorting, but it offers the framework for including them in the scheduling process. Once the topological order has been determined, additional checks and algorithms can be used to allocate courses to particular semesters while taking credit constraints into account.

3. *Efficiency*

The time complexity of topological sorting is  $O(V+E)$ , where  $V$  is the number of courses and  $E$  is the number of prerequisites. Given that the number of prerequisites and courses is normally controllable, this time complexity is frequently effective for the course scheduling problem. The technique may provide a proper course schedule in an acceptable amount of time by using topological sorting.

4. *Correctness*

The courses are organized in a manner that satisfies all prerequisites thanks to topological sorting. In order to maintain the accuracy of the timetable, it makes sure that no course is taken before its prerequisites have been satisfied. The Course Scheduling solution can deliver precise and dependable results since it uses a tried-and-true algorithm.

5. *Scalability*

Larger examples of the Course Scheduling problem can be handled by graph algorithms, including topological sorting. Topological sorting may effectively construct the course schedule without experiencing any serious performance concerns, as long as the number of courses and requirements stays within a tolerable range.

Due to its capacity to manage precondition dependencies, potential integration with credit limits, efficiency, correctness, and scalability, topological sorting is an appropriate solution paradigm for the Course Scheduling problem. You can efficiently build a course schedule while taking prerequisites and credit constraints into account by using this graph algorithm.

### **3.7 Consideration**

#### **3.7.1 Implementing Dynamic Programming**

The implementation of Dynamic Programming for Course Scheduling Problem in the subset of Class Capacity Problem is a consideration, nevertheless, as seat capacity is one of the criterias for an optimized schedule that we defined in the original scenario that we produced. A good paradigm for solving the Classroom Selection problem is dynamic programming.

1. *Optimal substructure*

When a problem can be broken down into overlapping subproblems and the optimal solution can be built from the optimal solutions of its subproblems, dynamic programming is effective. Finding the class with the least surplus capacity is the objective of the Classroom Selection issue, which may be viewed as determining the best solution by taking unique class capacities into account. Because the issue displays an ideal substructure, dynamic programming is a workable solution.

2. *Overlapping subproblems*

The code snippet's dynamic programming technique stores interim findings in a 2D table (dp). The minimum number of classrooms needed to hold a specific number of students for a specific combination of classes is shown in each cell of the table. The approach efficiently resolves overlapping subproblems and prevents redundant computations by incrementally filling in this table.

3. *Time complexity*

The dynamic programming solution has an  $O(n * m)$  time complexity, where  $n$  denotes the number of classes and  $m$  the total number of students. The code's nested loops run through the number of students and the class sizes, accordingly. The table's dimensions are proportional to the number of classes and the total number of students, thus even for larger issue cases, the time complexity is still reasonable.

4. *Efficiency*

The Classroom Selection problem has an effective solution provided by dynamic programming. The approach reduces the need for repeated computations and streamlines the search for the class with the least amount of extra capacity by computing and saving interim results in the dp table. The answer is produced in a timely and organized manner thanks to the usage of dynamic programming.

5. *Correctness*



The code snippet's dynamic programming technique successfully identifies the class with the least amount of extra capacity. It determines the class with the capacity that is closest to the total number of students and calculates the minimal number of classrooms needed to accommodate various amounts of students for each subset of courses.

The Classroom Selection problem is a good fit for dynamic programming as a solution paradigm. To quickly identify the class with the least surplus capacity, it makes use of the problem's ideal substructure and overlapping subproblems. The method guarantees accuracy, effectiveness, and acceptable temporal complexity, making it a workable solution for this issue.

### 3.7.2 Sample of Implementation

This is the example of implementation of Dynamic Programming for Course Scheduling Problem in the subset of class capacity

```
import java.util.Arrays;

public class ClassroomSelection {
    public static void main(String[] args) {
        int[] classCapacity = {10, 20, 30, 40, 50}; // Capacity of each class
        int totalStudents = 50; // Total number of students

        String[] classNames = {"A", "B", "C", "D", "E"}; // Class names

        int[][] dp = new int[classCapacity.length + 1][totalStudents + 1];

        // Initialize the table with a large value
        int largeValue = totalStudents + 1;
        for (int i = 0; i <= classCapacity.length; i++) {
            Arrays.fill(dp[i], largeValue);
        }

        // Base case: If there are no students, no classrooms are needed
        for (int i = 0; i <= classCapacity.length; i++) {
            dp[i][0] = 0;
        }

        // Fill the table using dynamic programming
        for (int i = 1; i <= classCapacity.length; i++) {
            for (int j = 1; j <= totalStudents; j++) {
                // If the current class can accommodate j students
                if (classCapacity[i - 1] <= j) {
                    dp[i][j] = Math.min(dp[i - 1][j], 1 + dp[i][j - classCapacity[i - 1]]);
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
    }
}
```

```

}

// Find the class with the minimum excess capacity (closest to the total number
of students)
int bestClassIndex = -1;
int minExcessCapacity = Integer.MAX_VALUE;
for (int i = 0; i < classCapacity.length; i++) {
    int excessCapacity = classCapacity[i] - totalStudents;
    if (excessCapacity >= 0 && excessCapacity < minExcessCapacity) {
        minExcessCapacity = excessCapacity;
        bestClassIndex = i;
    }
}

// Print the best class if found
if (bestClassIndex != -1) {
    String bestClass = classNames[bestClassIndex];
    System.out.println("Best Class: " + bestClass);
    System.out.println("Minimum Classrooms Required: 1");
} else {
    System.out.println("No valid class found.");
}

}
}

```

## 4.0 Design The Algorithm

### 4.1 Topological Sorting

#### 4.1.1 Kahn's algorithm for topological sorting (Python)

1. Initialize the in-degree dictionary to store the in-degree of each node in the course\_graph as 0.
2. By iterating through the nodes and their neighbors, incrementing the in-degree of each neighbor in the in\_degree dictionary, you can determine the in-degree for each node. This step does not require any recurrence.
3. Iterate over the nodes in the in\_degree dictionary and enqueue any nodes with an in-degree of 0 by adding them to the queue. This step also does not require any recurrence.
4. Utilize the Kahn's algorithm to perform the topological sorting. This section doesn't involve recurrence.
  - To store the sorted nodes, initialize an empty list called sorted\_nodes.
  - Dequeue a node from the front of the queue even when it is not empty.

- Dequeued nodes are appended to the list of sorted\_nodes.
  - Iterate through the dequeued node's neighbors:
    - Decrement according to the in\_degree dictionary.
    - Enqueue a neighbor into the queue if its in-degree decreases to 0.
5. Make a new directed nx type graph called new\_graph.DiGraph().
  6. Select nodes from the sorted\_nodes list to add to new\_graph. This step does not require recurrence.
  7. Copy the edges from the original course\_graph and add them to the new\_graph. This step does not require recurrence.
  8. Transfer the relevant nodes' data from the original course\_graph to the new\_graph's associated nodes. This step does not require recurrence.
  9. Assign the self the new\_graph.Variable top\_graph.

The algorithm does not require any recurrence and instead uses an iterative method. The key concept is to compute the in-degrees of nodes, enqueue nodes with an in-degree of 0, perform topological sorting using Kahn's algorithm, generate a new graph with the sorted nodes and edges, and then copy the node data from the unsorted graph to the sorted graph. The provided code doesn't specifically reference any optimisation functions.

#### **Pseudocode:**

START:

FUNCTION topologicalSort(graph):

inDegree <- createInDegreeArray(graph)

queue <- createEmptyQueue()

sortedNodes <- createEmptyList()

FOR node IN graph.nodes:

IF inDegree[node] = 0:

enqueue(queue, node)

WHILE queue is not empty:

node <- dequeue(queue)

```
append node to sortedNodes
```

```
FOR neighbor IN graph.neighbors(node):
```

```
    inDegree[neighbor] -= 1
```

```
    IF inDegree[neighbor] = 0:
```

```
        enqueue(queue, neighbor)
```

```
newGraph <- createEmptyGraph()
```

```
addNodes(newGraph, sortedNodes)
```

```
addEdges(newGraph, graph.edges())
```

```
copyNodeData(newGraph, graph)
```

```
RETURN newGraph
```

```
FUNCTION createInDegreeArray(graph):
```

```
    inDegree <- empty dictionary
```

```
    FOR node IN graph.nodes:
```

```
        inDegree[node] <- 0
```

```
    FOR node IN graph.nodes:
```

```
        FOR neighbor IN graph.neighbors(node):
```

```
            inDegree[neighbor] += 1
```

RETURN inDegree

```
graph <- createGraph() // Create a directed graph  
addNodesAndEdges(graph) // Add nodes and edges based on course prerequisites  
sortedGraph <- topologicalSort(graph) // Perform topological sorting  
drawGraph(sortedGraph) // Visualize the sorted graph
```

END.

## 5.0 Analysis of The Algorithm's Correctness

### 5.1 Topological Sorting

#### 5.1.1 Kahn's algorithm for topological sorting (Python)

The provided code implements a course scheduling algorithm using a directed graph representation and performs topological sorting to create a course schedule.

#### **Correctness**

1. Creating the graph: The 'create\_graph' method builds a directed graph using the course data as input. Each time a course and its requirements are iterated over, the graph's nodes and edges are updated to reflect this. Each node's properties are stored with the course information. This step accurately depicts the graph's course dependencies.
2. Topological sorting: The 'create\_top\_sort' method employs Kahn's algorithm to conduct topological sorting on the course graph. It determines each node's in-degree, enqueues nodes with in-degree 0, and then sorts the nodes by going through the queue. A new graph is made using the nodes that were sorted as a result. This process produces a legitimate list of courses that satisfy prerequisites.
3. Creating the schedule: A course schedule is created using the 'create\_schedule' method using the sorted graph. The prerequisites and credits for each course are taken into account when it iterates across the sorted nodes. It allocates courses to semesters while taking the desired credits and the beginning semester into consideration. The created schedule is a dictionary of the years and semesters. The courses are correctly scheduled in this stage using the provided constraints.

Overall, the algorithm correctly constructs a course graph, performs topological sorting, and generates a valid course schedule taking into account prerequisites, desired credits, and the starting semester.

### ***Time Complexity of Implemented Topological Sorting in Code***

1. Creating the graph: The code iterates over the course input and prerequisites, adding nodes and edges to the graph. This step takes  $O(V + E)$  time, where  $V$  is the number of courses and  $E$  is the number of prerequisites.
2. Topological sorting: The topological sorting algorithm used is Kahn's algorithm, which has a time complexity of  $O(V + E)$ . It involves calculating in-degrees, enqueueing nodes with in-degree 0, and processing nodes in the queue to update in-degrees and generate the sorted order.
3. Creating the schedule: The code iterates over the sorted nodes and performs various operations based on the course data and constraints. In the worst case, it may iterate over all the courses and perform checks for each prerequisite and semester. Thus, the time complexity is dependent on the number of courses and their attributes.

Therefore, the overall time complexity of the algorithm is  $O(V + E)$ , where  $V$  is the number of courses and  $|E|$  is the number of prerequisites.

Note: The space complexity of the algorithm is  $O(V + E)$  as it stores the course graph, topologically sorted graph, and the course schedule.

The algorithm, where  $V$  is the number of courses and  $E$  is the number of prerequisites, has a time complexity of  $O(V + E)$ . The course graph is correctly created, topological sorting is carried out, and a legitimate course plan that takes into account requirements, desired credits, and the first semester is produced.

### ***Time Complexity of The Whole System***

#### **1. Graph Creation**

- The creation of the directed graph and adding nodes and edges based on course prerequisites takes  $O(N)$  time, where  $N$  is the number of courses.
- While performing addition of data to each node in the graph takes  $O(N)$  time.
- Therefore, time complexity for graph creation would be  $O(N)$ .

#### **2. Topological Sorting**

- The overall time complexity for topological sorting is  $O(V + E)$ , as explained in previous section.

#### **3. Course Scheduling:**

- Initializing all the variables and validating inputs takes  $O(1)$  time.

- Then, iterating through the nodes of the graph takes  $O(N)$  time.
- The process of checking prerequisites and other conditions takes  $O(M)$  time, where  $M$  is the maximum number of prerequisites for a course.
- Next, updating the schedule and credits per semester takes  $O(1)$  time.
- Therefore, the time complexity for course scheduling can be approximated as  $O(N * M)$ , where  $N$  is the number of courses and  $M$  is the maximum number of prerequisites for a course.

#### 4. File Reading and UI:

- The process of reading the input file and validating the format takes  $O(N)$  time, where  $N$  is the number of lines in the file.
- Lastly, creating the graph UI and displaying the graph takes  $O(1)$  time.

Overall, the time complexity of the program is dominated by the course scheduling component, which is approximately  $O(N * M)$ , where  $N$  is the number of courses and  $M$  is the maximum number of prerequisites for a course.

## 8.0 Implementation of An Algorithm

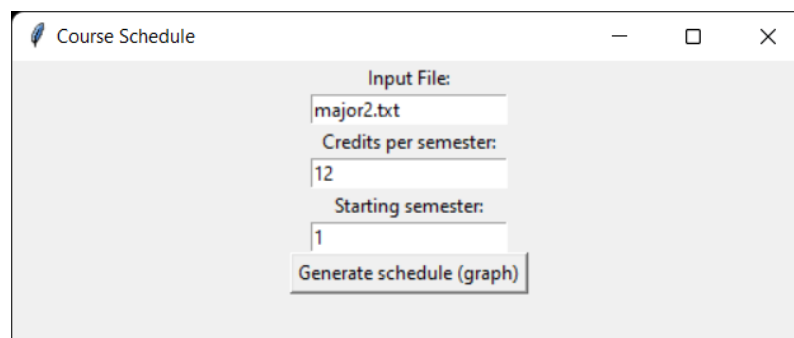
In our first try, we successfully obtained the fundamental output by implementing Topological Sorting in Java using DFS. But as we tried to add visualization using JavaFX to improve its usefulness, we ran into problems that made it difficult to fix them and get the results we wanted.

We chose to investigate Kahn's Algorithm for Topological Sorting in order to overcome these challenges, and we chose to put it into practice using Python. Python provides advantages in terms of ease of implementation and practical libraries for visualization because it is a simpler language than Java. We anticipated a more seamless integration of the visualization component into our course scheduling solution by using Python's modules.

We were able to implement Kahn's Algorithm in Python to schedule courses effectively and aesthetically by using NetworkX to represent graphs and Matplotlib for visualization. We think that Python is better to complete this project due to its concise syntax and the availability of specialized visualization libraries. Python's libraries often provide higher-level abstractions, enabling users to create visualizations quickly and easily with fewer lines of code.

The implementation of both Java and Python is compiled in the online portfolio.

## 9.0 Program Testing



The screenshot shows a window titled "Course Schedule". Inside the window, there are three input fields and one button. The first input field is labeled "Input File:" and contains the text "major2.txt". The second input field is labeled "Credits per semester:" and contains the number "12". The third input field is labeled "Starting semester:" and contains the number "1". Below these fields is a button labeled "Generate schedule (graph)".

Figure 1. Pop-up window prompting for user input



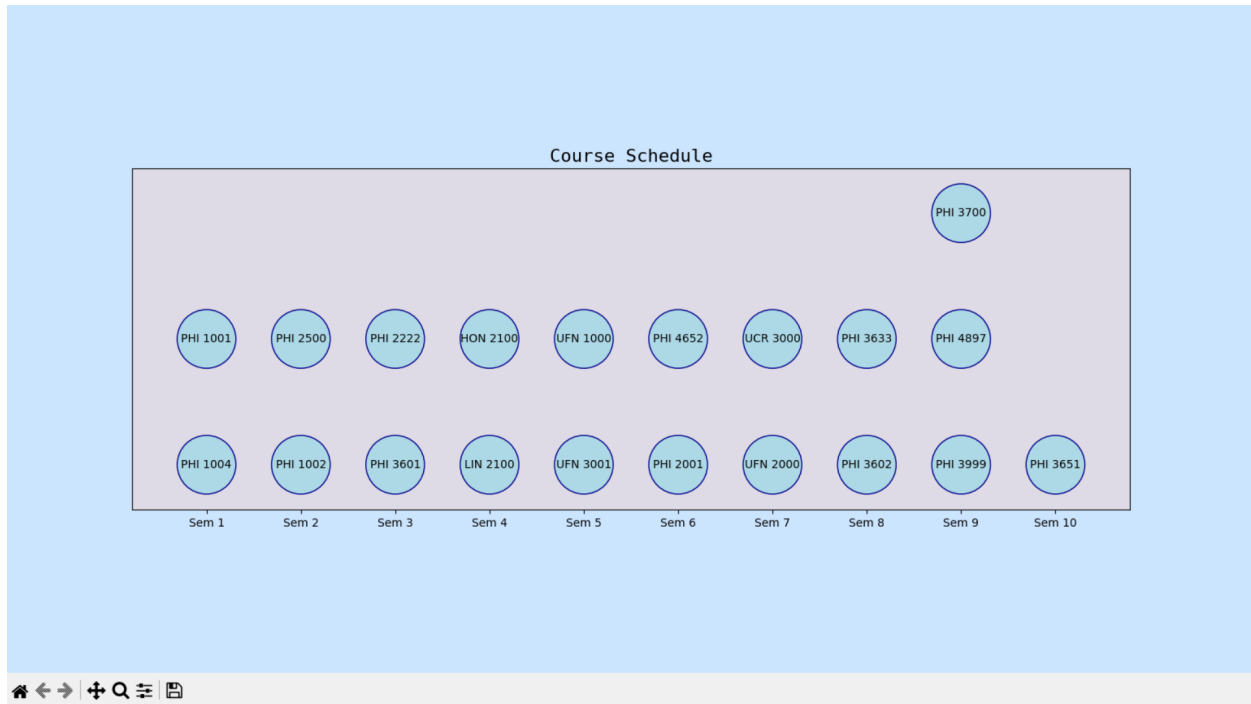


Figure 2. Generated Schedule Based on Topological Sorting Algorithm

## 10.0 Conclusion

In summary, the optimization of university course scheduling is a complex task that requires careful consideration of various factors. Manual scheduling processes are prone to errors, and graph algorithms, specifically topological sorting, provide a suitable solution paradigm for the course scheduling problem. The usage of graph algorithms has a number of benefits, including effective dependency resolution, streamlined schedule creation, and the creation of a sorted graph. While credit limits and extraordinary situations may call for additional considerations, graph algorithms typically address prerequisites and dependencies. Colleges and universities can optimize course scheduling, streamline offerings, enhance the student experience, and guarantee effective progression towards degree completion by using graph algorithms. This strategy provides a trustworthy and practical way to improve the entire course scheduling procedure.

## **11.0 Online Portfolio**

The following is the link to the Online Portfolio:

<https://github.com/Ramesh260402/CSC4202-G1-ProjectReport>

## 12.0 References

- Ajwani, D., Friedrich, T. (2010). Average-case analysis of incremental topological ordering. *\*Discrete Applied Mathematics\**, 158(4), 240–250.
- Bellman, R. E., & Dreyfus, S. E. (2015). *Applied dynamic programming* (Vol. 2050). Princeton university press.
- Haeupler, B., Kavitha, T., Mathew, R., Sen, S., Tarjan, R.E. (2012). Incremental cycle detection, topological ordering, and strong component maintenance. *\*ACM Transactions on Algorithms\**, 8(1), 3:1–3:33.
- Goodrich, M. T., & Tamassia, R. (2015). *Algorithm design and applications* (Vol. 363). Hoboken: Wiley.
- Kaveh, A., & Rahami, H. (2006). Analysis, design and optimization of structures using force method and genetic algorithm. *International Journal for Numerical Methods in Engineering*, 65(10), 1570-1584.
- Pearce, D. J., & Kelly, P. H. (2007). A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics (JEA)*, 11, 1-7.
- Prabhumoye, S., Salakhutdinov, R., & Black, A. W. (2020). Topological sort for sentence ordering. *arXiv preprint arXiv:2005.00432*.
- Sherine, A., Jasmine, M., Peter, G., & Alexander, S. A. (2023). *\*Algorithm and Design Complexity\**. CRC Press.
- Vince, A. (2002). A framework for the greedy algorithm. *Discrete Applied Mathematics*, 121(1-3), 247-260.

## 13.0 Project Progress

### 13.1 Week 10 , Milestone 1

Milestone 1	Planning											
Date (Wk)	26/05/23 (Week 10)											
Description/ sketch	1. Established scenario refinement  2. Enhance accuracy and realism of the project											
Role	<table><tr><td>Ramesh Arvin A/L Anpalagan</td><td>Nik Qistina Nurin Binti NikShaiful Anwar</td><td>Ponmugillan A/L Mani</td><td>Nor Syakila binti Salim</td></tr><tr><td>Documentati on, Researching</td><td>Documentati on, Researching</td><td>Documentati on, Researching</td><td>Documentati on, Researching</td></tr></table>				Ramesh Arvin A/L Anpalagan	Nik Qistina Nurin Binti NikShaiful Anwar	Ponmugillan A/L Mani	Nor Syakila binti Salim	Documentati on, Researching	Documentati on, Researching	Documentati on, Researching	Documentati on, Researching
Ramesh Arvin A/L Anpalagan	Nik Qistina Nurin Binti NikShaiful Anwar	Ponmugillan A/L Mani	Nor Syakila binti Salim									
Documentati on, Researching	Documentati on, Researching	Documentati on, Researching	Documentati on, Researching									

### 13.2 Week 11 , Milestone 2

Milestone 4	Exploring											
Date (Wk)	02/06/23 (Week 11)											
Description/ sketch	<div>1. Explore various example of solutions</div> <div>2. Identify the solutions most suitable for the project/scenario</div> <div>3. Determine why the selected solutions and the example of problems are closely related to our problem scenario and project's objective</div>											
Role	<table><tr><td>Ramesh Arvin A/L Anpalagan</td><td>Nik Qistina Nurin Binti Nik Shaiful Anwar</td><td>Ponmugillan A/L Mani</td><td>Nor Syakila binti Salim</td></tr><tr><td>Documentati on, Researching</td><td>Documentati on, Researching</td><td>Documentati on, Researching</td><td>Documentati on, Researching</td></tr></table>				Ramesh Arvin A/L Anpalagan	Nik Qistina Nurin Binti Nik Shaiful Anwar	Ponmugillan A/L Mani	Nor Syakila binti Salim	Documentati on, Researching	Documentati on, Researching	Documentati on, Researching	Documentati on, Researching
Ramesh Arvin A/L Anpalagan	Nik Qistina Nurin Binti Nik Shaiful Anwar	Ponmugillan A/L Mani	Nor Syakila binti Salim									
Documentati on, Researching	Documentati on, Researching	Documentati on, Researching	Documentati on, Researching									

### 13.3 Week 12 , Milestone 3

Milestone 4	Developing											
Date (Wk)	09/06/23 (Week 12)											
Description/ sketch	<div>1. Develop the program</div> <div>2. Edit the codes if necessary</div> <div>3. Completing the implementation</div> <div>4. Debugging if encounter any issues</div>											
Role	<table><tr><td>Ramesh Arvin A/L Anpalagan</td><td>Nik Qistina Nurin Binti Nik Shaiful Anwar</td><td>Ponmugillan A/L Mani</td><td>Nor Syakila binti Salim</td></tr><tr><td>Documentati on, Code Debugger</td><td>Documentati on, Code Debugger</td><td>Developer of Program</td><td>Developer of Program</td></tr></table>				Ramesh Arvin A/L Anpalagan	Nik Qistina Nurin Binti Nik Shaiful Anwar	Ponmugillan A/L Mani	Nor Syakila binti Salim	Documentati on, Code Debugger	Documentati on, Code Debugger	Developer of Program	Developer of Program
Ramesh Arvin A/L Anpalagan	Nik Qistina Nurin Binti Nik Shaiful Anwar	Ponmugillan A/L Mani	Nor Syakila binti Salim									
Documentati on, Code Debugger	Documentati on, Code Debugger	Developer of Program	Developer of Program									

### 13.4 Week 13 , Milestone 4

Milestone 5	Analyzing											
Date (Wk)	16/06/23 (Week 13)											
Description/ sketch	<div>1. Conduct an analysis of correctness of the implementation</div> <div>2. Verify the accuracy and reliability of the code</div> <div>3. Perform analysis of the time complexity for the implementation</div> <div>4. Carefully examine the algorithm utilized in the code to determine efficiency of the code</div>											
Role	<table><tr><td>Ramesh Arvin A/L Anpalagan</td><td>Nik Qistina Nurin Binti Nik Shaiful Anwar</td><td>Ponmugillan A/L Mani</td><td>Nor Syakila binti Salim</td></tr><tr><td>Documentati on</td><td>Documentati on</td><td>Developer of Program</td><td>Developer of Program</td></tr></table>				Ramesh Arvin A/L Anpalagan	Nik Qistina Nurin Binti Nik Shaiful Anwar	Ponmugillan A/L Mani	Nor Syakila binti Salim	Documentati on	Documentati on	Developer of Program	Developer of Program
Ramesh Arvin A/L Anpalagan	Nik Qistina Nurin Binti Nik Shaiful Anwar	Ponmugillan A/L Mani	Nor Syakila binti Salim									
Documentati on	Documentati on	Developer of Program	Developer of Program									

### 13.5 Week 14 , Milestone 6

Milestone 6	Finalization											
Date (Wk)	18/06/23 (Week 14)											
Description/ sketch	<div>1. Prepare online portfolio (by using GitHub, Google Sites or Google Colab)</div> <div>2. Showcase project progress, compiling the codes, pseudocodes, illustrating the problem and describe the algorithm analysis</div> <div>3. Preparing for the presentation</div> <div>4. Deliver the presentation</div>											
Role	<table><tr><td>Ramesh Arvin A/L Anpalagan</td><td>Nik Qistina Nurin Binti Nik Shaiful Anwar</td><td>Ponmugillan A/L Mani</td><td>Nor Syakila binti Salim</td></tr><tr><td>Documentati on, Presentor</td><td>Documentati on, Presentor</td><td>Documentati on, Presentor</td><td>Documentati on, Presentor</td></tr></table>				Ramesh Arvin A/L Anpalagan	Nik Qistina Nurin Binti Nik Shaiful Anwar	Ponmugillan A/L Mani	Nor Syakila binti Salim	Documentati on, Presentor	Documentati on, Presentor	Documentati on, Presentor	Documentati on, Presentor
Ramesh Arvin A/L Anpalagan	Nik Qistina Nurin Binti Nik Shaiful Anwar	Ponmugillan A/L Mani	Nor Syakila binti Salim									
Documentati on, Presentor	Documentati on, Presentor	Documentati on, Presentor	Documentati on, Presentor									