ASSIGNMENT – 07

Title – Scenario: University Timetable Scheduling A university is facing challenges in scheduling exam timetables due to overlapping student enrolments in multiple courses. To prevent clashes, the university needs to assign exam slots efficiently, ensuring that no two exams taken by the same student are scheduled at the same time.
To solve this, the university decides to model the problem as a Graph Colouring Problem, where:

- Each course is represented as a vertex.
- An edge exists between two vertices if a student is enrolled in both courses.
- Each vertex (course) must be assigned a colour (time slot) such that no two adjacent vertices share the same colour (no two exams with common students are scheduled in the same slot).

As a scheduling system developer, you must:
1. Model the problem as a graph and implement a graph colouring algorithm (e.g., Greedy Colouring or Backtracking).
2. Minimize the number of colours (exam slots) needed while ensuring conflict-free scheduling.
3. Handle large datasets with thousands of courses and students, optimizing performance.
4. Compare the efficiency of Greedy Colouring, DSATUR, and Welsh-Powell algorithms for better scheduling.

Extend the solution to include room allocation constraints where exams in the same slot should fit within available classrooms.

Aim:
To design an efficient exam timetable scheduling system using Graph Colouring, ensuring that no two exams with common students are held at the same time while minimizing the total number of exam slots.

Course Outcomes:
Apply Backtracking strategies to solve various problems.

Course Objectives:
1. To know the basics of computational complexity of various algorithms.
2. To select appropriate algorithm design strategies to solve real-world problems.

Theory:

1. Graph Theory:
Graph Theory is a mathematical concept used to model pairwise relationships between objects.
A **graph G(V, E)** consists of:

- **Vertices (V):** Represent entities such as courses.
- **Edges (E):** Represent relationships between entities, such as shared students between courses.

Graphs can be:
- **Directed or Undirected:** Depending on whether relationships have direction.
- **Weighted or Unweighted:** Depending on whether edges carry weights like distance or cost.

In this scheduling problem, the graph is **undirected and unweighted**, as the relationship (conflict) between two courses is mutual and has no weight.

2. Graph Colouring:
Graph Colouring is the process of assigning colors to vertices of a graph so that no two adjacent vertices have the same colour.
In the context of timetable scheduling:
- Each course is a vertex**.**
- An edgeconnects two vertices if at least one student is enrolled in both courses.
- Each colour represents a distinct exam time slot**.**

The goal is to minimize the total number of colours (time slots) used.
The **minimum number of colours** required to colour a graph without conflicts is called its **Chromatic Number**.

3. Conflict Graph (Problem Modelling):
To prevent overlapping exams:

- Courses with common students are **connected by edges**.
- If two courses share an edge, they **cannot** have the same exam slot.
  Thus, the timetable scheduling becomes equivalent to a **Graph Coloring Problem**, where no two connected nodes (conflicting courses) can share the same color (exam time).

This approach efficiently models real-world exam conflicts and ensures no student faces two exams at the same time.

4. Graph Colouring Algorithms:

a. Greedy Coloring Algorithm:

- Colors one vertex at a time using the smallest possible color not used by its adjacent vertices.
- Although simple, it does not always produce the minimum number of colors.
- **Time Complexity:** $O(V^2)$
- **Advantages:** Easy to implement and works well for large graphs.

b. DSATUR (Degree of Saturation) Algorithm:

- Chooses the next vertex to color based on the **number of different colors** used by its adjacent vertices.
- Produces near-optimal solutions and handles complex conflict graphs effectively.
- **Time Complexity:** $O(V^2)$

c. Welsh-Powell Algorithm:

- Sorts vertices in descending order of their degrees (most connected first).
- Then applies the greedy method to color vertices.
- Performs better than simple greedy in most cases.
- **Time Complexity:** $O(V^2)$

5. Room Allocation Constraint:

After time slots (colours) are assigned, each exam in the same slot must be assigned to available classrooms.

Room allocation is done based on:

- **Room capacity:** Each exam must fit within the capacity of the assigned room.
- **Availability:** A room can only host one exam per time slot.
- **Optimization:** Allocate the smallest available room that fits all students to maximize space utilization.

This ensures effective use of university resources and avoids scheduling conflicts both in time and space.

6. Optimization for Large Datasets:

For large universities with thousands of students and courses:

- Use **Adjacency Lists** (instead of matrices) to save memory.
- Use **Hash Maps or Dictionaries** to store student-course relationships.
- Apply **sorting and efficient traversal** to reduce complexity.
- Implement **parallelization** when building the conflict graph for faster processing.

Pseudocode:
Input:

C = list of all courses

S = list of all students with their enrolled courses

R = list of available rooms with capacities

Output:

Timetable showing (Course → Time Slot → Room)

Begin

// Step 1: Build Conflict Graph

Create an empty graph G with vertices as courses in C

For each student s in S do

courses_list = courses enrolled by s

For each pair (c1, c2) in courses_list do

Add an edge between c1 and c2 in G

EndFor

EndFor

// Step 2: Initialize Colors

For each course c in C do

color[c] ← -1 // no color assigned initially

EndFor

// Step 3: Apply Graph Coloring (Greedy Method)

Sort all courses in descending order of their degree (number of edges)

color[first_course] ← 0

For each course u in sorted list (excluding first_course) do

Create available_colors[] ← true for all colors

For each adjacent course v of u do

If color[v] ≠ -1 then

available_colors[color[v]] ← false

EndIf

EndFor

Assign the smallest colori where available_colors[i] = true

```
color[u] ← i
EndFor

   // Step 4: Room Allocation for Each Time Slot
   For each time_slot t in set of used colors do
exams_in_slot ← all courses having color = t
      Sort exams_in_slot by number_of_students (descending)
      Sort rooms R by capacity (descending)
      For each course e in exams_in_slot do
         For each room r in R do
            If capacity(r) ≥ students_in(e) AND r is available for slot t then
               Assign room r to course e
               Mark r as occupied for slot t
               Break
EndIf
EndFor
EndFor
EndFor

   // Step 5: Display Final Timetable
   For each course c in C do
      Print "Course:", c, "→ Time Slot:", color[c], "→ Room:", assigned_room[c]
EndFor
End
```

Example Input:

Student	**Enrolled Courses**
S1	C1, C2
S2	C2, C3
S3	C1, C3
S4	C4
Room	**Capacity**
R1	100
R2	80
R3	50

Expected Output:

Course	Time Slot	Room
C1	Slot 1	R1
C2	Slot 2	R2
C3	Slot 3	R3
C4	Slot 1	R2

Explanation of Output:

- Courses **C1, C2, and C3** conflict with each other (same students), hence different slots.
- **C4** has no conflict, so it shares **Slot 1** with **C1**.
- Each course is assigned a room according to its capacity and availability.

Code

```
import java.util.*;

public class ExamSchedulerSimple {

    // ---------------- Build Graph ----------------
    // Graph: adjacency list of conflicts between courses
    static List<List<Integer>> buildGraph(int numCourses, Map<Integer, List<Integer>> studentCourses) {
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < numCourses; i++) graph.add(new ArrayList<>());

        // For each student, add edges between all courses they take
        for (List<Integer> courses : studentCourses.values()) {
            int k = courses.size();
            for (int i = 0; i < k; i++) {
                for (int j = i + 1; j < k; j++) {
                    int a = courses.get(i);
                    int b = courses.get(j);
                    if (!graph.get(a).contains(b)) graph.get(a).add(b);
                    if (!graph.get(b).contains(a)) graph.get(b).add(a);
```

```
            }
        }
    }
    return graph;
}

// ---------------- Greedy Coloring ----------------
static int[] greedyColoring(List<List<Integer>> graph) {
    int n = graph.size();
    int[] colors = new int[n];
    Arrays.fill(colors, -1);

    colors[0] = 0; // first course gets slot 0

    for (int u = 1; u < n; u++) {
        boolean[] used = new boolean[n];
        for (int v : graph.get(u)) {
            if (colors[v] != -1) used[colors[v]] = true;
        }
        int c;
        for (c = 0; c < n; c++) if (!used[c]) break;
        colors[u] = c;
    }
    return colors;
}

// ---------------- Welsh-Powell Coloring ----------------
static int[] welshPowellColoring(List<List<Integer>> graph) {
    int n = graph.size();
    int[] colors = new int[n];
    Arrays.fill(colors, -1);

    Integer[] order = new Integer[n];
```

```
    for (int i = 0; i < n; i++) order[i] = i;

    // Sort courses by descending degree (number of conflicts)
    Arrays.sort(order, (a, b) -> graph.get(b).size() - graph.get(a).size());

    int color = 0;
    for (int i = 0; i < n; i++) {
        int u = order[i];
        if (colors[u] != -1) continue;
        colors[u] = color;
        for (int j = i + 1; j < n; j++) {
            int v = order[j];
            if (colors[v] == -1) {
                boolean conflict = false;
                for (int adj : graph.get(v)) {
                    if (colors[adj] == color) {
                        conflict = true;
                        break;
                    }
                }
                if (!conflict) colors[v] = color;
            }
        }
        color++;
    }
    return colors;
}

// ---------------- Utility: count number of slots used ----------------
static int countColors(int[] colors) {
    Set<Integer> set = new HashSet<>();
    for (int c : colors) set.add(c);
    return set.size();
```

```
    }

    // ---------------- Main Function ----------------
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of courses: ");
        int numCourses = sc.nextInt();
        System.out.print("Enter number of students: ");
        int numStudents = sc.nextInt();

        Map<Integer, List<Integer>> studentCourses = new HashMap<>();
          System.out.println("Enter student enrollments (studentId courseCount course1 course2
...):");
        for (int s = 0; s < numStudents; s++) {
            int studentId = sc.nextInt();
            int count = sc.nextInt();
            List<Integer> courses = new ArrayList<>();
            for (int i = 0; i < count; i++) {
                int c = sc.nextInt();
                courses.add(c);
            }
            studentCourses.put(studentId, courses);
        }

        // Build conflict graph
        List<List<Integer>> graph = buildGraph(numCourses, studentCourses);

        // Greedy Coloring
        int[] greedyColors = greedyColoring(graph);
        System.out.println("\n--- Greedy Coloring ---");
        System.out.println("Number of slots used: " + countColors(greedyColors));
        for (int i = 0; i < numCourses; i++)
```

```
            System.out.println("Course " + i + " -> Slot " + greedyColors[i]);

        // Welsh-Powell Coloring
        int[] wpColors = welshPowellColoring(graph);
        System.out.println("\n--- Welsh-Powell Coloring ---");
        System.out.println("Number of slots used: " + countColors(wpColors));
        for (int i = 0; i < numCourses; i++)
            System.out.println("Course " + i + " -> Slot " + wpColors[i]);

        sc.close();
    }
}
```

Output

```
Enter number of courses:
5
Enter number of students: 3
Enter student enrollments (studentId courseCount course1 course2 ...):
0 2 0 1
1 2 1 2
2 2 3 4

--- Greedy Coloring ---
Number of slots used: 2
Course 0 -> Slot 0
Course 1 -> Slot 1
Course 2 -> Slot 0
Course 3 -> Slot 0
Course 4 -> Slot 1

--- Welsh-Powell Coloring ---
Number of slots used: 2
Course 0 -> Slot 1
Course 1 -> Slot 0
Course 2 -> Slot 1
Course 3 -> Slot 0
Course 4 -> Slot 1
```

Conclusion:

The university timetable scheduling problem is efficiently solved using the **Graph Colouring technique**, where each course is represented as a vertex and conflicts are modelled as edges. By applying colouring algorithms such as the Greedy method, the system ensures that no two exams with common students are scheduled in the same time slot. Extending this approach with **room allocation** based on capacity further optimizes resource usage. Hence, the proposed model provides a **conflict-free, scalable, and practical** solution for exam scheduling in universities.