	Pimpri Chinchwad Education Trust's **Pimpri Chinchwad College of** **Engineering**
Assignment 7	

ASSIGNMENT - 07: University Timetable Scheduling

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 Context

- **Course Objectives:**
  - To know the basics of computational complexity of various algorithms.
  - To select appropriate algorithm design strategies to solve real-world problems.
- **Course Outcomes:**
  - Apply Backtracking strategies (and related heuristics like graph coloring) to solve various problems.

Concepts

1. Problem Modeling: The Conflict Graph

The core of this problem is modeling the relationships. We use **Graph Theory** to do this.

- A graph $G(V,E)$ consists of **Vertices (V)** and **Edges (E)**.
- **Vertices (V):** Each course is represented as a vertex (e.g., "Data Structures", "Calculus").
- **Edges (E):** An edge is created between two vertices if they have a conflict. In this scenario, an edge exists if at least one student is enrolled in both courses.

This resulting graph is called a **Conflict Graph**. It is **undirected** (a conflict between C1 and C2 is the same as C2 and C1) and **unweighted**.

2. Graph Colouring

Once we have the conflict graph, the problem becomes a classic **Graph Colouring** problem.

- **Concept:** Assign a "colour" to each vertex (course) such that no two *adjacent* vertices (conflicting courses) have the same colour.
- **Application:** Each "colour" represents a unique exam time slot.
- **Goal:** The primary goal is to minimize the total number of colours used. This minimum number is called the

graph's **Chromatic Number**.

By solving the coloring problem, we guarantee that no two exams with common students are scheduled at the same time.

3. Colouring Algorithms (Heuristics)

Finding the absolute minimum (the Chromatic Number) is an NP-hard problem. Therefore, we use efficient heuristics to find a *good* solution, which may not always be the absolute minimum.

- **a. Greedy Coloring:**
  - **How it works:** Processes vertices one by one. For each vertex, it assigns the smallest possible colour (e.g., 1, 2, 3...) that is not already used by its adjacent, already-coloured neighbors.
  - **Performance:** Simple, but the result heavily depends on the order in which vertices are processed. Its time complexity is $O(V^2)$.
- **b. Welsh-Powell Algorithm (Implementation Used):**
  - **How it works:** An improvement on the simple greedy method. It first **sorts all vertices in descending order of their degree** (most connected/conflicting courses first). Then, it applies the greedy coloring method to this sorted list.
  - **Performance:** This heuristic often performs better than simple greedy coloring because it prioritizes coloring the most "constrained" vertices first. The complexity is dominated by the sort and coloring, remaining $O(V^2)$.
- **c. DSATUR (Degree of Saturation):**
  - **How it works:** A more dynamic heuristic. It first colors the vertex with the highest degree. After that, it iteratively selects the next vertex to color based on its "saturation degree"—the number of *different* colors already used by its neighbors.
  - **Performance:** Often produces near-optimal solutions and is very effective for complex conflict graphs. Time complexity is typically $O(V^2)$.

4. Post-Processing: Room Allocation

After the graph is colored (i.e., time slots are assigned), a second-pass algorithm is needed for room allocation.

1. Group all exams (courses) scheduled for the same time slot (colour).
2. For each group, sort the exams by the number of students enrolled (descending).
3. Sort the available rooms by capacity (ascending).
4. Iterate through the sorted exams and assign each one to the **first available room** that can accommodate the number of students (a "best-fit" strategy).
5. Once a room is assigned for a time slot, it is marked as occupied for that slot and cannot be used by another exam in the *same* slot.

This approach efficiently packs exams into the smallest suitable rooms, maximizing resource utilization.

5. Optimization for Large Datasets

To handle thousands of courses and students:

- **Adjacency Lists:** Use adjacency lists (like vector<vector<int>>) to represent the graph, which is much more memory-efficient than an $O(V^2)$ adjacency matrix for sparse graphs.
- **Hash Maps:** Use hash maps (like unordered_map) for $O(1)$ average-time lookups to map course names

(strings) to their vertex indices (integers).
- **Efficient Data Handling:** Read student enrollments and build the conflict graph in a single pass.

Pseudocode

This pseudocode outlines the logic for both coloring and room allocation.

```
Begin
  // Step 1: Build Conflict Graph
  Input: Courses (C), Students (S), Rooms (R)
  Create empty graph G with vertices for each course in C

  For each student s in S:
    courses_list = courses enrolled by s
    For each unique pair (c1, c2) in courses_list:
      Add an edge between c1 and c2 in G
    EndFor
  EndFor

  // Step 2: Initialize Colors
  For each course c in C:
    color[c] = -1 // -1 means uncolored
  EndFor

  // Step 3: Apply Graph Coloring (Welsh-Powell Method)
  Sort all courses in a list by degree (descending)

  For each course u in the sorted list:
    Create set of available_colors (initially all true)

    For each adjacent course v of u:
      If color[v] is not -1:
        mark available_colors[color[v]] as false
      EndIf
    EndFor

    Assign color[u] = smallest color i where available_colors[i] is true
  EndFor

  // Step 4: Room Allocation for Each Time Slot
```

```
    For each time_slot t in set of used colors:
        exams_in_slot = all courses where color[c] == t
        Sort exams_in_slot by number_of_students (descending)
        Sort rooms R by capacity (ascending)

        For each course e in exams_in_slot:
            For each room r in R:
                If capacity(r) >= students_in(e) AND r is available for slot t:
                    Assign room r to course e
                    Mark r as occupied for slot t
                    Break // Move to next exam
                EndIf
            EndFor
        EndFor
    EndFor

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

C++ Implementation

```
#include <iostream>     // For cin, cout, ios::sync_with_stdio, cin.tie
#include <vector>       // For vector
#include <string>       // For string, getline
#include <unordered_map>  // For unordered_map
#include <unordered_set>  // For unordered_set
#include <algorithm>    // For sort, max, unique
#include <numeric>      // For iota
#include <iomanip>      // For setw (if needed, though not in final output)

using namespace std;

int main() {
```

```
// Optimize C++ stream I/O for speed
ios::sync_with_stdio(false);
cin.tie(nullptr);

// --- 1. Read Courses ---
int numCourses;
if (!(cin >> numCourses)) return 0;

vector<string> courseNames(numCourses);
unordered_map<string, int> courseIndex;
for (int i = 0; i < numCourses; i++) {
    cin >> ws; // Consume trailing newline
    getline(cin, courseNames[i]);
    courseIndex[courseNames[i]] = i;
}

// --- 2. Build Conflict Graph ---
int numStudents;
cin >> numStudents;

vector<int> studentsIn(numCourses, 0); // Count students per course
vector<vector<int>> adj(numCourses);   // Adjacency list for conflicts

for (int si = 0; si < numStudents; si++) {
    string sid;
    int k; // Number of courses for this student
    cin >> ws;
    cin >> sid >> k;

    vector<int> enrolled;
    for (int j = 0; j < k; j++) {
        string cname;
        cin >> ws;
        getline(cin, cname);
        if (courseIndex.find(cname) == courseIndex.end()) continue;

        int idx = courseIndex[cname];
        enrolled.push_back(idx);
        studentsIn[idx]++;
```

```
    }

    // Add edges between all pairs of courses this student is enrolled in
    sort(enrolled.begin(), enrolled.end());
    enrolled.erase(unique(enrolled.begin(), enrolled.end()), enrolled.end());

    for (size_t a = 0; a < enrolled.size(); a++) {
        for (size_t b = a + 1; b < enrolled.size(); b++) {
            int u = enrolled[a], v = enrolled[b];
            adj[u].push_back(v);
            adj[v].push_back(u);
        }
    }
}

// Clean up adjacency lists (remove duplicate edges)
vector<int> degree(numCourses);
for (int i = 0; i < numCourses; i++) {
    sort(adj[i].begin(), adj[i].end());
    adj[i].erase(unique(adj[i].begin(), adj[i].end()), adj[i].end());
    degree[i] = adj[i].size();
}

// --- 3. Graph Coloring (Welsh-Powell Heuristic) ---
// Create an 'order' vector [0, 1, 2, ..., C-1]
vector<int> order(numCourses);
iota(order.begin(), order.end(), 0);

// Sort 'order' vector based on degree (descending)
sort(order.begin(), order.end(), [&](int a, int b) {
    if (degree[a] != degree[b]) return degree[a] > degree[b];
    return a < b;
});

vector<int> color(numCourses, -1); // -1 = uncolored
int maxColor = 0;

for (int idx : order) {
    vector<char> used(numCourses + 1, false); // Tracks used colors by neighbors
```

```
    for (int v : adj[idx]) {
        if (color[v] != -1) {
            used[color[v]] = true;
        }
    }

    // Find the smallest available color (1-indexed)
    int c = 1;
    while (used[c]) c++;

    color[idx] = c;
    maxColor = max(maxColor, c);
}

// --- 4. Room Allocation ---
int numRooms;
cin >> numRooms;

vector<pair<string, int>> rooms(numRooms);
for (int i = 0; i < numRooms; i++) {
    string rname;
    int cap;
    cin >> ws;
    getline(cin, rname);
    cin >> cap;
    rooms[i] = {rname, cap};
}

// Get order of rooms, sorted by capacity (ascending)
vector<int> roomOrder(numRooms);
iota(roomOrder.begin(), roomOrder.end(), 0);
sort(roomOrder.begin(), roomOrder.end(), [&](int a, int b) {
    if (rooms[a].second != rooms[b].second) {
        return rooms[a].second < rooms[b].second;
    }
    return a < b;
});
```

```
// Tracks which room (by index) is used in which slot
vector<unordered_set<int>> roomOccupied(maxColor + 1);
vector<string> assignedRoom(numCourses, "Unassigned");

// Assign rooms for each time slot
for (int t = 1; t <= maxColor; t++) {
    vector<int> exams;
    for (int i = 0; i < numCourses; i++) {
        if (color[i] == t) exams.push_back(i);
    }

    // Sort exams in this slot by student count (descending)
    sort(exams.begin(), exams.end(), [&](int a, int b) {
        if (studentsIn[a] != studentsIn[b]) {
            return studentsIn[a] > studentsIn[b];
        }
        return a < b;
    });

    // Assign rooms using best-fit
    for (int e : exams) {
        for (int ri : roomOrder) {
            // If room fits and is not occupied in this slot
            if (rooms[ri].second >= studentsIn[e] &&
                roomOccupied[t].count(ri) == 0) {

                assignedRoom[e] = rooms[ri].first;
                roomOccupied[t].insert(ri);
                break;
            }
        }
    }
}

// --- 5. Display Final Timetable ---
cout << "Course\tTimeSlot\tRoom\n";
for (int i = 0; i < numCourses; i++) {
    cout << courseNames[i] << "\tSlot " << color[i] << "\t" << assignedRoom[i] << "\n";
}
```

```
    return 0;
}
```

Example Input and Output

Example Input

4

C1

C2

C3

C4

4

S1 2 C1

C2

S2 2 C2

C3

S3 2 C1

C3

S4 1 C4

3

R1

100

R2

80

R3

50

Explanation of Input Conflicts

- S1 takes C1 and C2 -> Edge (C1, C2)
- S2 takes C2 and C3 -> Edge (C2, C3)
- S3 takes C1 and C3 -> Edge (C1, C3)
- S4 takes C4 -> No conflicts for C4.
- This creates a "triangle" conflict between C1, C2, and C3, requiring 3 different slots. C4 conflicts with none of them.

Expected Output

Course TimeSlot Room

C1	Slot 1	R1
C2	Slot 2	R1
C3	Slot 3	R1
C4	Slot 1	R2

Conclusion

The university timetable scheduling problem can be efficiently modeled and solved using Graph Colouring techniques. By representing courses as vertices and student overlaps as edges, a conflict graph is constructed.

Applying a heuristic algorithm like **Welsh-Powell (Greedy-Sort)** ensures that no two exams with common students are scheduled in the same time slot. The system successfully minimizes the total number of required time slots.

Furthermore, extending this solution with a **post-processing step for room allocation**, which considers room capacity and exam size, optimizes the use of university resources. This model provides a conflict-free, scalable, and practical solution for exam scheduling.