Solving the Course Scheduling Problem Using Brute-Force Search with Pattern Restriction

Mahmoud Hatem El Badawy Mohammed

ENG : Ahmed El Demoksy

DR : Mohamed Abd Elfattah

Introduction

Efficient scheduling of exams is crucial to reduce conflicts among students and optimize resource usage. In this context, we address a specific problem: arranging courses into exam schedules in a way that minimizes conflicts.

To solve this, we employ **Brute-Force Search with Pattern Restriction**, leveraging predefined patterns for efficient exploration of possible solutions.

Problem Statement

The problem revolves around designing an optimal exam schedule while minimizing student conflicts (overlapping students between courses) under certain constraints.

Objective

To determine the sequence of course scheduling (based on conflict costs) that adheres to the given constraints and minimizes conflicts using the **Brute-Force Search with Pattern Restriction** algorithm.

Inputs

1. Conflict Table

This table defines overlapping relationships between courses:

- **SUB_ID:** The ID of a course.
- **CONFLICT_SUB_ID:** The ID of another course with overlapping students.

- **NUMOFINTERSECTION:** The number of students that overlap between these two courses.

```

[
  {"sub_id": 101, "conflict_sub_id": 201, "numOfIntersection": 5},
  {"sub_id": 201, "conflict_sub_id": 301, "numOfIntersection": 3},
  {"sub_id": 301, "conflict_sub_id": 101, "numOfIntersection": 2}
]
```

2. Levels Table

Courses are grouped by their academic levels (e.g., First, Second, Third)

```
{
  1: [101, 102, 103],// 1st level
  2: [201, 202, 203],// 2nd level
  3: [301, 302, 303] // 3rd level
}
```

3. Constraints

The problem introduces two scheduling patterns to explore:

1. **PATTERN 1:** Level 1 → Level 2 → Level 3 (repeated in that order).

2. **Pattern 2:** Level 3 → Level 1 → Level 2 (repeated in that order).

The goal is to determine the sequence that minimizes the **total cost** (sum of the number of conflicts for consecutive courses).

Algorithm

We use the **Brute-Force Search with Pattern Restriction** to find the optimal sequence.

How the Algorithm Works

Sequence Generation:

- Generate two sequences based on the two provided patterns:
  - **Pattern 1:** First Level → Second Level → Third Level.
  - **Pattern 2:** Third Level → First Level → Second Level.

Conflict Cost Calculation:

- For each sequence, compute the total number of conflicts (cost) by summing the conflicts between consecutive days.

Select Optimal Sequence:

- Compare the costs of the two patterns and select the sequence with the lowest cost.

Time & Space Complexity

Time Complexity:

- Generating sequences runs **in O(n)O(n)O(n)**, where n is the number of courses.
- Conflict calculations for each sequence run in O(n2)O(n^2)O(n2).

Space Complexity:

- The memory required is minimal, primarily storing sequences and conflict tables.

Implementation in Dart

```dart
// Generate sequences based on patterns
List<int> generateSequence(Map<int, List<int>> levels, int pattern) {
  List<int> sequence = [];
  List<int> order = (pattern == 1) ? [1, 2, 3] : [3, 1, 2];

  while (sequence.length < levels.values.expand((x) => x).length) {
    for (var level in order) {
      // Use null-aware operator to ensure non-null value
      if (levels[level] != null) {
        sequence.addAll(levels[level]!);
      }
    }
  }
  return sequence;
}

// Calculate cost of a sequence
int calculateCost(List<int> sequence, List<Map<String, dynamic>> conflicts) {
  int cost = 0;

  for (int i = 0; i < sequence.length - 1; i++) {
    for (var conflict in conflicts) {
      if ((conflict["sub_id"] == sequence[i] &&
              conflict["conflict_sub_id"] == sequence[i + 1]) ||
          (conflict["conflict_sub_id"] == sequence[i] &&
              conflict["sub_id"] == sequence[i + 1])) {
        cost += (conflict["numOfIntersection"] as int); // Cast to int
      }
    }
  }
  return cost;
}

// Find the best sequence
List<int>? bestSequence;
int minCost = double.maxFinite.toInt(); // Use maxFinite.toInt() for the largest int

for (int pattern = 1; pattern <= 2; pattern++) {
  List<int> sequence = generateSequence(levels, pattern);
  int cost = calculateCost(sequence, conflictTable);
  if (cost < minCost) {
    minCost = cost;
    bestSequence = sequence;
  }
}
```

Explanation of the Code

1. Input Data

- **Conflict Table:** Defines the conflicts between courses and how many students overlap.
- **Levels Table:** Groups courses by their academic level.

2. Sequence Generation

We generate sequences using two predefined patterns:

- **Pattern 1:** First → Second → Third → Repeat.
- **Pattern 2:** Third → First → Second → Repeat.

3. Conflict Cost Calculation

The function calculateCost computes the total number of conflicts between consecutive days for a given sequence.

4. Optimal Sequence Selection

Both patterns are calculated, and the sequence with the lowest conflict cost is selected.

Results

Example Output

```
[Running] dart "d:\bestalgo\bin\bestalgo.dart"
Best Sequence: [101, 102, 103, 201, 202, 203, 301, 302, 303]
Minimum Cost: 0
```

Conclusion

The proposed **Brute-Force Search with Pattern Restriction** algorithm effectively solves the course scheduling problem by evaluating two predefined patterns for conflicts. The approach balances simplicity, efficiency, and effectiveness, making it a viable solution for small to medium-scale scheduling tasks.