Technical Framework for Subtree Selection and Node Weighting in Automated Prerequisite Course Planning

1 Introduction

For a given target course, prerequisites can be represented as a complex tree of requirements. The primary objective of this work is to define a technical framework for navigating this structure to find an optimal path for the student.
Specifically, given a prerequisite tree, the goal is to select an optimal directed subtree rooted at the target course. This subtree must satisfy all logical requirements (i.e., AND/OR conditions). The selection process is guided by a weighting system that reflects course quality and user preferences, aiming to maximize the total weight of the selected courses while promoting course reuse to ensure an efficient academic path.

2 Methodology

2.1 Prerequisite Graph Model

We model the prerequisite structure as a rooted Directed Acyclic Graph (DAG), denoted $G = (V, E)$, with the following properties:

- The vertex set V consists of two types of nodes:
  - **Course nodes**, which represent atomic courses, possibly with a minimum grade requirement.
  - **Logic nodes**, which represent either an AND ($\wedge$) or OR ($\vee$) condition over their children.
- The root of the DAG is the target course for which the prerequisite path is being planned.

- Edges in E represent dependencies, connecting parent nodes to their required sub-requirements (children).

2.2 Optimal Subtree Selection

The core of the methodology is a recursive function, $\mathsf{SelectSubtree}(T, S)$, designed to traverse the graph and identify the optimal set of courses. Here, T is the current node (either course or logic) under consideration, and S is the set of courses already selected, which is used to handle dependencies and prevent duplicates. The function is detailed in Algorithm 1.

Algorithm 1 Optimal Subtree Selection for Prerequisite Satisfaction

```
function SelectSubtree(T, S)
    if T is a course node then
        if T ∉ S then
            P ← {T}
            S ← S ∪ {T}
            for all prerequisite Q of T do
                P ← P ∪ SelectSubtree(Q, S)
            end for
            return P
        else
            return ∅
        end if
    else if T is an AND-node then
        P ← ∅
        for all child C of T do
            P ← P ∪ SelectSubtree(C, S)
        end for
        return P
    else if T is an OR-node then
        P* ← argmin_P Cost(P) over all P = SelectSubtree(C, S) for children C of T
        return P*
    end if
end function
```

The cost function, $\mathrm{Cost}(P)$, used at OR-nodes, is initially defined as the negative sum of weights of the courses in a potential path P: $\sum_{v \in P} -y_v$. This serves to maximize the total path weight. This function is expanded in Section 4.

2.3 Algorithmic Properties

- **Feasibility**: The algorithm guarantees a feasible path by only omitting courses that are already in the selected set S, ensuring that all dependencies are met.
- **Optimality at OR-nodes**: At each OR-node, the branch that yields the optimal cost (e.g., maximum weight, minimum courses) is chosen based on the defined cost function.
- **Completeness at AND-nodes**: At each AND-node, all branches are traversed, and their resulting course sets are unified to ensure all requirements are satisfied.
- **Theoretical Basis**: This approach frames course selection as a dynamic programming problem on a tree, which can be seen as a generalization of the AND/OR pathfinding problem in graphs.

3 Node Weighting Algorithm

Each course node is assigned a weight y_i to quantify its desirability based on historical student feedback and user-specified preferences.

3.1 Parameters

For each course i, we define the following parameters:

$$\begin{aligned} x_1 &: \text{liked score } [0, 100] \\ x_2 &: \text{easy score } [0, 100] \\ x_3 &: \text{useful score } [0, 100] \\ r_i &: \text{number of student ratings} \end{aligned}$$

User preferences are captured by a weight vector $\boldsymbol{\beta} = (\beta_1, \beta_2, \beta_3)$, where $\beta_k \geq 0$ and $\sum_k \beta_k = 1$. Predefined profiles for $\boldsymbol{\beta}$ include:

- **Focus on likeness**: $(0.7, 0.15, 0.15)$
- **Focus on easiness**: $(0.15, 0.7, 0.15)$
- **Focus on usefulness**: $(0.15, 0.15, 0.7)$
- **Balanced**: $(0.4, 0.3, 0.3)$

3.2 Reliability Scaling Factor

To account for the confidence in course ratings, a reliability scaling factor, λ_i, is introduced, which is monotonic in the number of ratings r_i. It can be defined using discrete thresholds:

$$\lambda_i = \begin{cases} 1.10 & \text{if } r_i > 100 \\ 1.00 & \text{if } 50 \leq r_i \leq 100 \\ 0.9 & \text{if } r_i < 50 \end{cases}$$

Alternatively, a continuous scaling function can be used for finer-grained adjustments:

$$\lambda_i = 1 + \alpha \cdot \left(\frac{r_i - \bar{r}}{\sigma_r} \right), \qquad \text{clamped to } [0.95, 1.05]$$

where $\bar{r} = 32.8743$ and $\sigma_r = 67.55$ are the mean and standard deviation of ratings across all courses, and $\alpha \approx 0.15$ is a small constant.

3.3 Null Value Handling

Missing rating values (x_1, x_2, x_3) are handled as follows:

- If **one or two** scores are null, they are imputed using the column median (or mean) over all courses. Alternatively, the remaining β_k weights can be re-normalized to sum to 1.
- If **all three** scores are null, they are assigned a low value to penalize the course while keeping it as a viable option:

  $$x_k^* = \gamma \cdot \min_{\text{all non-null values}} x_{j,\ell}$$

  for $k = 1, 2, 3$, with a penalty factor $\gamma \in [0.3, 0.7]$ (e.g., $\gamma = 0.5$). For such courses, λ_i is set to a penalty value like 0.97 to further reflect their unreliability.

3.4 Node Weight Calculation

The final weight y_i for a course is calculated as:

$$s_i = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$$

$$y_i = \min(\lambda_i s_i, 100)$$

The min operation ensures the weight does not exceed the maximum possible score of 100.

3.5 Edge Cases

- Courses with all rating scores imputed will have a small, non-zero weight. This ensures they are only selected if no other paths are available.
- If all available paths require traversing a node with imputed scores, the algorithm will still return the path with the highest possible total weight.

4 Global Reuse Preference (Depth-Aware New-Course Penalty)

The initial objective function encourages high-weight courses but does not explicitly favor solutions that reuse courses across different prerequisite branches. To promote efficient paths, we introduce a *new-course penalty*: a depth-aware cost added only when a course *ID* appears in the path for the first time.

4.1 Penalty Definition

Let S be the set of selected course IDs so far. For a course node v at depth d (root at $d = 0$), the base cost is

$$\text{baseCost}(v) = 1 - \frac{y_v}{100}.$$

A depth-aware penalty is charged only on first inclusion:

$$\text{newCoursePenalty}(v, S, d) = \begin{cases} \lambda(d) & v \notin S, \\ 0 & v \in S, \end{cases} \qquad \lambda(d) = \frac{\lambda_0}{1+d}, \;\; \lambda_0 \in [0.1, 0.4] \text{ (default 0.2)}.$$

The node cost is then

$$\text{cost}(v, S, d) = \text{baseCost}(v) + \text{newCoursePenalty}(v, S, d).$$

This is equivalent to a "reuse award" formulation up to a constant shift; we use the new-course penalty because it is simpler to implement and reason about.

4.2 Selection with Global Reuse

The recursive selection passes (S, d) so that OR-branches are evaluated from the same state:

- **Course node:** Return $(\text{cost}(v, S, d),\ S \cup \{v\})$.
- **AND node:** Sum child costs and accumulate S through the children.
- **OR node:** For each child C, evaluate with the same incoming S and pick the minimal-cost branch (stable tie-break by order).

Base-cost rule on reuse. When a course reappears in another branch, the base cost $1 - y_v/100$ and the penalty are charged *only* on the first inclusion of its ID in S. Later reuses add zero cost. This ensures that reusing a course strictly reduces $\mathcal{C}(P)$ relative to introducing a new course, all else equal.

Notes.

- The node weight y_v continues to guide the selection towards user-preferred courses, while the reuse penalty refines the selection to favor more efficient paths.
- The penalty schedule $\lambda_u(d)$ is a tunable hyperparameter and can be replaced by any monotonically decreasing function of depth.

4.3 Objective Function

The overall objective is to find a valid prerequisite path P that minimizes the total cost function $\mathcal{C}(P)$, which combines the base cost of courses with penalties for introducing new ones.

$$\min_P \mathcal{C}(P) = \min_P \sum_{v \in P} \text{cost}(v, S_v, d_v)$$

where for each node $v \in P$:

- d_v is its depth, and S_v is the set of courses selected prior to visiting v.
- $\text{cost}(v, S_v, d_v) = (1 - y_v/100) + \text{uniquePenalty}(v, S_v, d_v)$.

The $\mathsf{SelectSubtree}$ function acts as a recursive procedure to find the path P that minimizes this sum by making locally optimal decisions at OR-nodes while propagating the global set of selected courses.

5 Illustrative Examples

5.1 Example 1: Simple Prerequisite Tree

Consider the prerequisites for MATH249, which requires satisfying two groups of requirements: (MATH135 $\vee$ MATH145) $\wedge$ (MATH136 $\vee$ MATH146).

- **Scenario 1:** If the algorithm first selects MATH135, it satisfies the first group. For the second group, if MATH136 is chosen (perhaps due to a higher weight or because MATH135 is a prerequisite), the final path is $\{\text{MATH135, MATH136}\}$.
- **Scenario 2:** If MATH145 is chosen for the first group, then MATH146 must be chosen for the second, as MATH136 typically requires MATH135. The resulting path is $\{\text{MATH145, MATH146}\}$.

The algorithm selects the path with the optimal total cost, considering both course weights and dependencies.

5.2 Example 2: Complex Prerequisite Tree

The prerequisites for STAT330 involve multiple nested requirements:

- **STAT330**: requires MATH237 $\wedge$ (STAT230 $\vee$ STAT240) $\wedge$ STAT231.
- **MATH237**: requires (one of [MATH106, MATH114, MATH115, MATH136, MATH146]) $\wedge$ (one of [MATH128($\geq$70), MATH138($\geq$60), MATH148]).
- **STAT230**: requires (one of [MATH116, MATH118, MATH128]) $\wedge$ (one of [MATH137($\geq$80), MATH138]).
- **STAT231**: requires one of [MATH118($\geq$70), STAT220($\geq$70), STAT230].

A possible path that minimizes the number of unique courses could be constructed as follows:

1. To satisfy MATH237, select MATH136 and MATH138.
2. To satisfy STAT230, select MATH116 and reuse the already selected MATH138.
3. To satisfy STAT231, reuse the now-selected STAT230.

This yields the minimal path: {MATH136, MATH138 ($\geq$60), MATH116, STAT230 ($\geq$60), MATH237, The algorithm systematically avoids redundant courses by considering the global set of selected courses.

5.3 Example 3: Node Weight Calculation

Let's calculate the weight for ACTSC431, given ratings (`liked` = 68.42, `easy` = 28.57, `useful` = 100), number of ratings = 19, and a user preference for "likeness."

- $\boldsymbol{\beta} = (0.7, 0.15, 0.15)$.
- The reliability scalar λ is 0.9 (since $r_i = 19 < 50$).
- The weighted score is $s_{\text{ACTSC431}} = 0.7(68.42) + 0.15(28.57) + 0.15(100) = 67.18$.
- The final weight is $y_{\text{ACTSC431}} = 0.9 \times 67.18 = 60.46$.

If a course has all null ratings, assuming a global minimum rating of 18.18 and $\gamma = 0.5$:

- The imputed scores would be $x_1 = x_2 = x_3 = 9.09$.
- With $\lambda = 0.97$, the resulting weight y_i would be significantly lower, correctly flagging the course as a high-risk or unknown-quality option.