Class Scheduling Algorithm Description

Ranty Wang, Yupei Sun, Tianyun Song

November 2024

1 Algorithm

1.1 Description

The class scheduling algorithm is designed to minimize scheduling conflicts while respecting room capacity limitations and teacher availability constraints. This conflict-driven, popularity-adjusted approach utilizes a conflict matrix to identify and prioritize class pairs with the highest potential for overlap based on student preferences. The algorithm proceeds as follows:

1. **Conflict Matrix Creation**: For each student, the conflict count is incremented for every pair of their preferred classes. The conflict matrix records the number of overlapping preferences between each pair of classes, with higher values signifying greater potential conflicts if scheduled concurrently.

2. **Priority Scheduling of High-Conflict Classes**: Classes with the highest conflict values are prioritized and assigned to distinct time slots to minimize overlap. This step ensures that classes with significant student preference overlap do not occur at the same time.

3. **Scheduling of Remaining Classes**: After scheduling high-conflict classes, the remaining classes are prioritized based on students' preference counts. Classes are assigned to rooms that closely match their capacity requirements, minimizing resource wastage, while avoiding conflicts with previously scheduled classes. In addition, teacher availability is verified to prevent any professor from being scheduled to teach multiple classes simultaneously.

4. **Student Enrollment**: Students are enrolled in their preferred classes if the classes are available in non-conflicting time slots. If a time-slot conflict arises, students are not assigned to the overlapping class, thereby ensuring no double booking occurs.

1.2 Pseudocode

Algorithm 1 Class Scheduling Algorithm

```
Input: classes[], rooms[], teachers[], students[], T
Output: schedule[]

// Step 1: Create the conflict matrix
conflictMatrix = CreateConflictMatrix(students)
// Step 2: Assign high-conflict classes
topClasses = SelectTopClassesByConflict(conflictMatrix, T)
for each class in topClasses do
    AssignClassToDistinctTimeSlot(class, rooms, schedule)
end for
// Step 3: Assign remaining classes
remainingClasses = SortByPopularity(classes - topClasses)
for each class in remainingClasses do
    bestSlot = FindSlotWithMinConflict(class, schedule, conflictMatrix)
    if TeacherNotInRowConflict(class.teacher, bestSlot, schedule) then
        AssignClassToRoom(class, bestSlot, rooms, schedule)
    end if
end for
return schedule
```

Algorithm 2 Create Conflict Matrix

```
Input: students[]
Output: conflictMatrix
Initialize conflictMatrix as a zero matrix
for each student in students do
    preferredClasses = GetPreferredClasses(student)
    for each pair (class1, class2) in preferredClasses do
        Increment conflictMatrix[class1][class2]
        Increment conflictMatrix[class2][class1]
    end for
end for
return conflictMatrix
```

Algorithm 3 Select Top Conflict Classes

Input: conflictMatrix, T
Output: topClasses
Initialize topClasses as an empty list
// Calculate the total conflict value for each class
conflictSum[class] = $\sum$ conflictMatrix[class][otherClass] for all otherClass
// Sort classes by their total conflict value in descending order
Sort classes by conflictSum in descending order
// Select the top T classes
Add the first T classes to topClasses
return topClasses

Algorithm 4 Teacher Conflict Check

Input: teacher, row, schedule
Output: True if no conflict, False otherwise
// Check each time slot in the specified row
for each time slot slot in row **do**
    assignedClass = schedule[row][slot]
    if assignedClass.teacher == teacher **then**
        return False // Conflict found: teacher already assigned to another class in this row
    end if
end for
return True // No conflict detected

1.3 Time Complexity

The time complexity of the algorithm is analyzed step by step as follows:

Conflict Matrix Creation

- **Process:** For each student, the algorithm identifies their class preferences (up to P classes) and calculates pairwise conflicts between these preferred classes. The pairwise conflicts are stored in a conflict matrix.
- **Analysis:** Since P, the maximum number of preferred classes per student, is a small constant, this operation scales linearly with the number of students (S). The pairwise evaluation $\binom{P}{2}$ is constant across all students.
- **Time Complexity:** $O(S)$.

Conflict Sum Calculation and Sorting

- **Process:**
    1. Iterate through the conflict matrix (a symmetric $C \times C$ matrix) to calculate the total conflict for each class by summing over all C^2 entries.
    2. Sort the conflict sums to identify the top T classes with the highest conflicts.
- **Analysis:** Sorting the C^2 conflict values requires $O(C^2 \log C^2)$. Selecting the top T classes after sorting is $O(T)$, which is negligible compared to the sorting step.
- **Time Complexity:** $O(C^2 \log C^2)$.

Assigning Classes by Popularity

- **Process:** Sort all C classes based on the number of students enrolled in each class and assign them to time slots and rows with minimal conflicts.
- **Analysis:**
  1. Sorting the C classes by popularity requires $O(C \log C)$.
  2. For each class, evaluate all possible time slots and rows to find the placement with the minimum conflict. Assuming the number of rows is proportional to C, this step requires $O(C \cdot T)$, where T is the number of time slots.
  3. Teacher conflict checks use a hash table or set for efficient conflict detection with $O(1)$ per check, which is negligible compared to the other steps.
- **Time Complexity:** $O(C \log C + C \cdot T)$.

Overall Time Complexity Combining all steps, the overall time complexity of the algorithm is:

$$O(S) + O(C^2 \log C^2) + O(C \log C + C \cdot T).$$

The $O(C^2 \log C^2)$ term dominates when the number of classes is large, making it the primary factor in the algorithm's complexity.

2 Experimental Analysis

In this section, we analyze the runtime performance of the algorithm by varying the number of students and comparing the actual runtime with the predicted linear trend.

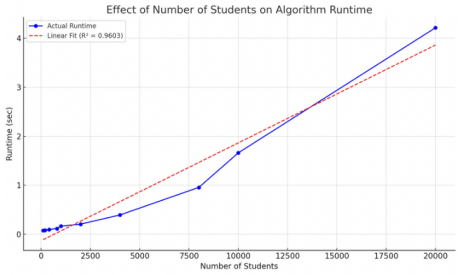

Figure 1: Effect of Number of Students on Algorithm Runtime. The blue line represents the actual runtime, and the red dashed line indicates the linear fit with $R^2 = 0.9603$.

Number of Students	Runtime (sec)
100	0.0812
200	0.0796
400	0.0918
800	0.1149
1000	0.1397
2000	0.2018
4000	0.396
8000	0.9671
10000	1.7333
20000	4.2357

Figure 2: Runtime Data Table: Number of Students vs Runtime in Seconds.

As shown in Figure 1, the runtime increases approximately linearly with the number of students. The linear fit (red dashed line) has an R^2 value of 0.9603, indicating a strong correlation between the number of students and the runtime.

- **Runtime Trend:** The runtime increases linearly as the number of students grows, validating the expected time complexity of $O(S)$ for the conflict matrix creation step.
- **Linear Fit:** The red line in Figure 1 represents the linear trend, closely matching the actual runtime. This suggests that the algorithm is efficient and scales well with the number of students.
- **Table Insights:** The runtime data table (Figure 2) provides detailed runtime measurements for different numbers of students, further supporting the observed linear trend.
- **Consistency with Complexity Analysis:** The experimental results align with the complexity analysis, confirming that the algorithm operates within the theoretical bounds for large student datasets.

In this section, we analyze the runtime performance of the algorithm by varying the number of classes and comparing the actual runtime with the predicted quadratic trend. The results are presented in Figure **??** and the accompanying table.

As shown in Figure 3, the runtime exhibits a clear quadratic growth trend as the number of classes increases. The quadratic fit (red dashed line) has an R^2 value of 0.9421, demonstrating strong agreement with the actual runtime.

The runtime analysis includes the following observations and insights:

- **Quadratic Runtime Growth:** The runtime increases quadratically with the number of classes. This is consistent with the theoretical time complexity of $O(C^2 \log C^2)$, as the primary operations include summing over all class conflicts and sorting these conflicts.
- **Accuracy of the Quadratic Fit:** The red dashed line represents the quadratic fit, which has an R^2 value of 0.9421, indicating a strong correlation between the actual runtime and the expected trend.

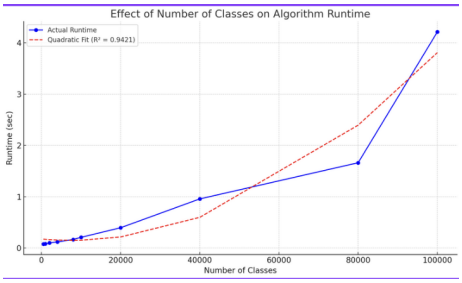

Figure 3: Effect of Number of Classes on Algorithm Runtime. The blue line represents the actual runtime, and the red dashed line indicates the quadratic fit with $R^2 = 0.9421$.

Number of Class	Runtime(sec)
500	0.0812
1000	0.0796
2000	0.0918
4000	0.1149
8000	0.1397
10000	0.2018
20000	0.396
40000	0.9671
80000	1.7333
100000	4.2357

Figure 4: Runtime Data Table: Number of Classes vs Runtime in Seconds.

- **Behavior for Small and Large Class Counts:** For smaller class counts (e.g., $C < 10,000$), the actual runtime closely aligns with the predicted trend, demonstrating the algorithm's efficiency even at smaller scales. For larger class counts, the quadratic growth becomes more evident, validating the scalability of the algorithm.
- **Table Analysis:** The runtime data table (Figure 4) provides detailed runtime measurements for different class counts. These measurements show a smooth increase in runtime as the class count doubles or quadruples, further reinforcing the quadratic trend.
- **Consistency with Complexity Analysis:** The observed results confirm that the algorithm's performance is dominated by the conflict sum calculation and sorting operations, which scale quadratically with the number of classes.

Overall, the results demonstrate that the algorithm performs as expected, with runtime increasing quadratically with the number of classes. These findings align with theoretical complexity analysis and suggest that the algorithm is well-suited for handling large-scale datasets.

2.1 Solution Quality Analysis

In this section, we analyze the solution quality of the scheduling algorithm under varying parameters, including the number of students, courses, time slots, and rooms. The results are presented below, with corresponding observations and insights.

2.1.1 Varying Student Number

As shown in Figure 5, the percentage of fit (%Fit) remains generally stable and consistent as the number of students increases. The following observations can be made:

- **Stability:** The algorithm demonstrates consistent quality in scheduling as the number of students increases, with only minor fluctuations in %Fit.
- **High Quality:** The stability of the %Fit highlights the algorithm's effectiveness in balancing student preferences, even as the dataset size grows.
- **Scalability:** These results indicate that the algorithm scales well with the number of students, maintaining high solution quality.

Classes	Students	Time Slot	Rooms	Runtime (sec)	Optimal	Real Fit	**% Fit**
30	200	20	100	0.0926	800	784	**0.98**
50	200	20	100	0.1124	800	752	**0.94**
50	200	20	100	0.0986	800	763	**0.9538**
80	200	20	100	0.1028	800	745	**0.9313**
100	200	20	100	0.1056	800	724	**0.905**
150	200	20	100	0.1199	800	701	**0.8763**

Figure 5: Quality Analysis: Student Number as a Variable.

2.1.2 Varying Course Number

Classes	**Students**	Time Slot	Rooms	Runtime (sec)	Optimal	Real Fit	**% Fit**
200	**300**	10	100	0.132	1200	1035	**0.8625**
200	**500**	10	100	0.1325	2000	1772	**0.886**
200	**800**	10	100	0.143	3200	2919	**0.9122**
200	**1200**	10	100	0.1379	4800	4416	**0.92**
200	**1600**	10	100	0.1557	6400	5853	**0.9145**
200	**2000**	10	100	0.1649	8000	7289	**0.9111**

Figure 6: Quality Analysis: Course Number as a Variable.

Figure 6 shows the impact of increasing the number of courses on the solution quality. Key insights include:

- **Slight Decline in %Fit:** As the number of courses increases, the %Fit shows a gradual decline, indicating increased complexity in balancing preferences.
- **Algorithm Challenges:** With more courses, the algorithm faces higher difficulty in assigning classes without conflicts, leading to a slight reduction in fit quality.
- **Acceptable Performance:** Despite the challenges, the %Fit remains relatively high, indicating reliable performance even under more demanding scenarios.

2.1.3 Varying Time Slot Number

In Figure 7, the number of time slots is varied to analyze its effect on scheduling quality. Observations include:

Classes	Students	**Time Slot**	Rooms	Runtime (sec)	Optimal	Real Fit	**% Fit**
50	200	**5**	100	0.106	800	652	**0.815**
50	200	**10**	100	0.1185	800	750	**0.9375**
50	200	**20**	100	0.0981	800	764	**0.955**
50	200	**40**	100	0.097	800	791	**0.989**
50	200	**80**	100	0.0905	800	800	**1**
50	200	**100**	100	0.0911	800	800	**1**

Figure 7: Quality Analysis: Time Slots as a Variable.

- **Significant Increase in %Fit:** As the number of time slots increases, the %Fit improves substantially, reaching 100% with sufficient time slots.
- **Flexibility:** More time slots provide greater flexibility for scheduling, allowing the algorithm to better satisfy student preferences.
- **Implications:** These results suggest that increasing time slots is a practical way to enhance solution quality in scenarios with high conflict potential.

2.1.4 Varying Room Number

Classes	Students	Time Slot	**Rooms**	Runtime (sec)	Optimal	Real Fit	**% Fit**
50	200	10	**20**	0.1013	800	737	**0.9213**
50	200	10	**50**	0.0972	800	738	**0.9225**
50	200	10	**80**	0.1156	800	746	**0.9325**
50	200	10	**100**	0.0979	800	752	**0.94**
50	200	10	**150**	0.0982	800	755	**0.9438**
50	200	10	**200**	0.0994	800	751	**0.9388**

Figure 8: Quality Analysis: Room Number as a Variable.

Figure 8 analyzes the effect of varying the number of rooms on the scheduling quality. Key findings include:

- **Slight Improvement in %Fit:** As the number of rooms increases, the %Fit shows a gradual improvement.
- **Better Flexibility:** More rooms allow the algorithm to better distribute classes, reducing scheduling conflicts and improving satisfaction.
- **Practical Recommendations:** These results highlight the importance of sufficient room allocation in achieving high-quality scheduling outcomes.

2.2 Summary

The analysis demonstrates the robustness and scalability of the algorithm under varying parameters. While challenges arise with increasing courses, the solution quality remains high across different scenarios, providing practical insights for optimizing scheduling performance.

3 BMC Data

3.1 Real Data Constraints

3.1.1 Department-Specific Room Assignments

Courses must be scheduled in rooms specifically designated for the department offering the class. This ensures that resource allocation aligns with departmental needs and maintains scheduling feasibility.

3.1.2 Student Enrollment Limits

Students are allowed to enroll in up to 5 courses. Preferences beyond the top 5 will not be considered, ensuring fair allocation of courses and adherence to realistic enrollment limits.

3.1.3 Teacher Assignment Limits

Teachers are restricted to teaching courses (e.g. 4 courses) per semester to ensure a balanced workload. Any assignments exceeding this limit will be reassigned or canceled.

3.1.4 Enrollment Capacity

Each classroom has a predefined enrollment capacity to prevent overcrowding. This constraint ensures that no course exceeds the seating and resource limits of the assigned classroom.

3.1.5 Course Level Restrictions

Courses are divided into levels (e.g., 100, 200, 300). Students are prohibited from enrolling in both a prerequisite and its higher-level course during the same term to maintain logical academic progression.

3.1.6 Courses with Both Lectures and Labs

Certain courses include both lecture and lab components. These components must be scheduled in a coordinated manner to avoid time conflicts, ensuring students can attend both sessions without overlap.

3.1.7 Time Slot and Weekday Alignment

All courses must be scheduled within realistic time slots and weekdays, adhering to established patterns derived from real-world data. This ensures compatibility with institutional scheduling practices.

3.2 Extensions Implementation

3.3 Solution Quality Analysis

The following is an analysis of the fit percentages calculated for various semesters from Spring 2010 to Spring 2015. The fit percentage is derived from the proportion of successful enrollments relative to real preferences for each dataset.

Dataset	StudentNum	ClassNum	RoomNum	TimeSlots	Real Preferences	Optimal	Fit (%)
S2015	1703	307	69	69	2999	4682	64.05
S2014	1719	318	74	75	2965	5028	58.97
S2013	1731	332	76	77	3071	5117	60.02
S2012	1698	325	70	71	3029	4883	62.03
S2011	1528	324	67	66	2855	4549	62.76
F2014	1635	280	67	74	2815	4863	57.89
F2013	1644	320	69	78	2934	5076	57.80
F2012	1659	293	70	79	2888	5047	57.22
F2011	1600	280	64	76	2790	4870	57.29
F2010	1475	288	68	68	2627	4659	55.95

Table 1: Summary of datasets and calculated fit percentages.

- **Higher Fit Percentages in Spring Semesters:** Spring semesters consistently show higher fit percentages compared to Fall semesters. For example, Spring 2015 has a fit percentage of 64.05% compared to Fall 2014 with 57.89%. One plausible reason is that Spring semesters offering more diverse elective courses, resulting in better alignment with student preferences. Fall semesters, on the other hand, include mandatory courses that most students need to take, like ESEM courses that are heavily enrolled by incoming freshmen in Bryn Mawr College, limiting flexibility and slightly lowering fit percentages.
- **Impact of Time Slots and Room Numbers:** The number of available time slots and rooms seems to impact fit percentages. For instance, Spring 2013, with 76 rooms and 77 time slots, has a fit percentage of 60.02%, whereas Fall 2013, despite having more rooms and time slots (69 and 78, respectively), achieves a slightly lower fit percentage of 57.80%. This indicates that other factors, such as class size distribution and student preferences, also play critical roles.

3.4 Limitation and Discussion

While the scheduling algorithm demonstrates overall good performance in accommodating student preferences, there are several limitations when applied to real-world datasets. After analyzing the results of specifically each class fit percentage, we identified three key findings that highlight these limitations:

3.4.1 Distribution of Fit Percentages Across Classes

For each dataset, the fit percentages of individual classes show a wide range. While the overall performance is relatively excellent — many classes achieve high fit percentages—there is noticeable variation:

- **Finding:** For example, in Spring 2015, **one-third of the 308 classes (102 classes)** achieved a fit percentage of 100%, meaning all students who wanted to enroll in these classes successfully got in. Additionally, **163 classes (more than half)** had a fit percentage of over 80%. This demonstrates good alignment for a majority of classes.
- **Limitation:** However, a minority of classes exhibit significantly lower fit percentages, indicating room for improvement in the scheduling process to better handle all preferences.

3.4.2 Population-Adjusted Algorithm Bias

The algorithm prioritizes classes with higher student demand, ensuring large population classes are assigned classrooms and timeslots first. While this approach maximizes the

overall fit percentage, it inherently deprioritizes smaller classes, leading to potential conflicts for low-enrollment classes. This results in a sharper decline in fit percentages for small classes compared to larger ones:

- **Example:** In Spring 2015, **ClassID: 008821** had a fit percentage of **0%**, despite having 10 students who expressed preferences for it. Similarly, in Spring 2014, **ClassID: 012353** also had a fit percentage of **0%**, with 3 students preferring the class.

3.4.3 Classroom Capacity Constraints

A significant limitation arises from classroom capacity constraints, where even when a classroom is assigned to a timeslot, its capacity may not be sufficient to accommodate all interested students.

- **Example:** In Spring 2015:
  - **ClassID: 003406** had a preference count of **42**, but only 7 students were enrolled. The classroom's capacity was **11**, leading to a fit percentage of **16.67%**.
  - **ClassID: 014569** had a preference count of **70**, but only 17 students were enrolled. The classroom's capacity was **18**, resulting in a fit percentage of **24.29%**.

Conclusion

These limitations highlight areas where the algorithm can be improved:

- Addressing fairness for low-enrollment classes.
- Implementing strategies to better match classroom capacity with demand.
- Improving flexibility in resource allocation to reduce extreme cases of underserviced classes.

Future work can focus on incorporating additional constraints, such as adaptive capacity scaling or alternative prioritization methods, to enhance scheduling effectiveness.

3.5 Recommendation

Based on the limitations identified in the scheduling algorithm, we propose the following recommendations for improving student satisfaction during the registration process. These suggestions aim to address the underlying issues and optimize resource allocation before the course enrollment period begins:

3.5.1 Adjust Scheduling for Small Classes with Low Preferences

- **Recommendation:** Allocate specific time slots and classrooms for low-population classes to ensure they are not overshadowed by high-demand courses during scheduling.
- **Purpose:** Mitigate the sharp decrease in fit percentages for small classes caused by the prioritization of larger courses.
- **Benefit:** This ensures that small classes receive adequate attention during the scheduling process, reducing unavoidable conflicts and improving satisfaction for students in niche or less popular courses.

3.5.2 Open Additional Sessions for High-Demand Courses

- **Recommendation:** Introduce additional sessions or sections for high-demand courses during pre-registration planning to ensure more students can be enrolled in their preferred classes.
- **Purpose:** Address the bottleneck caused by limited availability for courses with a high number of preferences.
- **Benefit:** This reduces over-enrollment conflicts in high-demand courses while maintaining a fair distribution of time slots and resources.

3.5.3 Pre-Registration Surveys for Demand Analysis

- **Recommendation:** Conduct a pre-registration survey to collect student preferences before the official enrollment period.
- **Purpose:** Use the survey results to predict demand, enabling better classroom allocation and scheduling adjustments in advance.
- **Benefit:** Proactively addressing high-demand courses and optimizing resource allocation leads to a smoother enrollment process and higher satisfaction rates.

4 Further Improvements

4.1 Simulated Annealing for Class Schedule Optimization

Our algorithm could be improved by employing **Simulated Annealing** as a method to iteratively optimize the generated class schedule. During the development process, we experimented with a variant of Simulated Annealing designed to systematically enhance the schedule by focusing on the classes with the lowest student satisfaction levels. The goal was to gradually maximize the overall "fit percentage" of the schedule, which measures how well student preferences align with final enrollments. Below, we describe the optimization approach and discuss its limitations.

4.1.1 Objective

The primary objective of this Simulated Annealing implementation is to maximize student satisfaction across all courses by improving the enrollment alignment for classes that exhibit the lowest fit percentages.

4.1.2 Approach

- **Initialization**: Start with an initial schedule, which may have been generated by a greedy algorithm or another scheduling method.
- **Identifying Problematic Classes**: In each iteration, identify the course with the lowest fit percentage (i.e., the course with the highest mismatch between student preferences and actual enrollments).
- **Same Department Swap**: Attempt to swap the problematic course with another course within the same department to maximize compatibility:
  - Courses from the same department share similar room and resource requirements, making them suitable candidates for swapping.
  - By swapping two courses, we aim to optimize the placement of both, thus improving student enrollment and satisfaction.
- **Swap Evaluation**: After performing the swap:
  - Calculate the new fit numbers for both classes.
  - Keep the swap if it improves the overall fit percentage; otherwise, revert the change.
- **Iterative Optimization**: Repeat this swapping process for a fixed number of iterations, aiming to escape local optima and explore potentially better configurations.

4.1.3 Limitations

- **Departmental Constraints**: The algorithm only allows swapping between courses within the same department to maintain consistency with resource requirements. This constraint limits the possible swaps and may cause the algorithm to converge prematurely to a local optimum.
- **Local Optima**: Due to the inherent nature of Simulated Annealing, there is always a possibility of getting stuck in a local optimum where no beneficial swaps are available. This can make it challenging to guarantee finding a globally optimal solution.

- **Fixed Iterations**: The effectiveness of Simulated Annealing is often dependent on the number of iterations and the cooling schedule. In our implementation, a fixed number of iterations was used, which might limit the potential for fully exploring the solution space.

4.1.4 Algorithm Description

The algorithm starts with an initial schedule, which can be generated using any baseline scheduling algorithm. It then iteratively attempts to improve the schedule by identifying the course with the lowest satisfaction (preference score) in its current time slot and attempting to swap it with another course within the same department. If the swap improves the overall satisfaction, it is accepted; otherwise, the algorithm continues to try other swaps. If all possible swaps for the least preferred course fail, the algorithm moves to the second least preferred course, and so on, until all courses in the department have been evaluated.

This process continues until no further improvement can be made for the current department, at which point the algorithm can proceed to the next department or terminate if all departments have been processed.

4.2 Pseudocode

Algorithm 5 Iterative Schedule Improvement (Attempted)

```
Input: initialSchedule, departments[], preferenceMatrix
Output: improvedSchedule

currentSchedule = initialSchedule
for each department in departments do
    Step 1: Sort courses by preference in ascending order
    sortedCourses = SortByPreference(currentSchedule, department, preferenceMatrix)
    for each course in sortedCourses do
        currentSatisfaction = CalculateSatisfaction(currentSchedule, preferenceMatrix)
        Step 2: Attempt to swap with other courses in the same department
        for each candidateCourse in department.courses do
            tempSchedule = SwapCourses(currentSchedule, course, candidateCourse)
            newSatisfaction = CalculateSatisfaction(tempSchedule, preferenceMatrix)
            if newSatisfaction ¿ currentSatisfaction then
                currentSchedule = tempSchedule
                Break and restart with the next least preferred course
                Break
            end if
        end for
    end for
end for
return currentSchedule
```

4.3 Conclusion

We applied this approach multiple times, but it did not yield significant improvements in the schedule quality. Despite numerous adjustments and iterations, the algorithm

struggled to produce meaningful results, largely due to the inherent limitations outlined earlier.
Theoretically, simulated annealing should allow exploration beyond local optima, but in practice, several factors restricted its effectiveness here. The nature of our constraints, including the limited capacity for swapping within departmental schedules, led to a high probability of reverting back to suboptimal states. This resulted in the algorithm often failing to escape local minima and ultimately converging to an unsatisfactory solution.