Class Scheduling Problem

Cărămidă Iustina-Andreea - 332CA

Faculty of Automatic Control and Computer Science
University Politehnica of Bucharest
`iustina.caramida@stud.acs.upb.ro`

Abstract. With the rapid growth of student enrollment and the expansion of academic offerings in universities and colleges worldwide, the task of scheduling classes within existing timetables and facilities has become increasingly complex. Today, class scheduling requires consideration of multiple factors, including room availability, capacity, instructors' preferences, and more. This problem is considered to be NP-complete and has received some research during the past few years. Several formulations and algorithms have been proposed to solve scheduling problems, most of which are based on local search techniques. In this paper, we propose to compare 2 different types of algorithms to solve the class scheduling problem: the random restart Hill-Climbing algorithm and the A-Star algorithm.

Keywords: NP-complete · A Star · Hill-Climbing · searching algorithms · class scheduling.

1 Introduction

1.1 Problem Declaration

The Class Scheduling Problem represents a critical challenge in the field of educational operational research. Universities contend with an ever-growing course catalog, necessitating the efficient allocation of a diverse range of classes to classrooms with varying capacities. This optimization problem seeks to construct a course schedule that adheres to a comprehensive set of university constraints while simultaneously maximizing the effective and efficient utilization of existing facilities.

The complexity of the Class Scheduling Problem presents a multifaceted challenge due to a confluence of factors. First, there is significant variety in the number of students enrolled in each course. Similarly, classroom capacities exhibit a wide range of variation. These factors are further compounded by the existence of course-specific classroom constraints. Additionally, faculty members often express preferences regarding the day of the week, time slot, and even break schedule for their assigned courses. These factors, combined with the need to satisfy a multitude of university-specific regulations, render the task of assigning courses to classrooms a highly intricate endeavor.

Merely ensuring a classroom can accommodate a course's enrolled students is insufficient. Such an approach would lead to suboptimal space utilization, potentially hindering the educational experience and fostering student dissatisfaction. Consider a scenario with two courses: one with six students and another with nineteen. Furthermore, imagine two classrooms exist, one seating twenty and another seating fifty. While technically feasible to schedule either course in either room, a more strategic allocation would be to assign the larger course to the larger classroom.

This example exemplifies the multifaceted nature of Class Scheduling Problem. Researchers continue to explore sophisticated methods to address this intricate problem, with the ultimate goal of ensuring the smooth operation and optimal resource allocation within universities.

1.2 Problem Statement

Statement: Receiving a set of courses, each with a specific number of students,a set of rooms with a specific capacity, and a list of instructors with their list of preferences, the goal is to assign each course to a room and a time slot, according to a list of constraints.

There are 2 types of constraints that need to be satisfied:

- **Hard constraints:** These constraints must be satisfied in order for the solution to be valid. If any of these constraints are not satisfied, the solution is invalid.
- **Soft constraints:** These constraints are not mandatory, but they are taken into account when evaluating the quality of the solution.

Hard constraints

- Within a designated time slot and in a given room, only one subject may be taught by a single instructor.
- During any given time slot, an instructor may teach only one subject, and this instruction must occur within a singular room.
- An instructor may conduct classes in a maximum of 7 time slots per week.
- Within a specified time slot, a room may accommodate a number of students equal to or less than its predetermined maximum capacity.
- All students enrolled in a particular subject must have designated class hours allocated for that subject.
- Instructors are restricted to teaching only the subjects in which they are specialized.
- All rooms are designated for classes pertaining only to the subjects for which they have been assigned

Soft constraints

- An instructor may express preferences regarding specific days of the week for teaching or may wish to avoid teaching on certain days.

– An instructor may have preferences regarding specific time slots during the day for teaching or may wish to avoid teaching during certain time slots.
– An instructor may prefer not to have a break exceeding a certain number of hours between consecutive classes.

1.3 Methods of Approach

Random Restart Hill Climbing is a local search algorithm commonly applied in optimization problems. It begins with an initial solution and iteratively explores neighboring solutions, selecting the one that improves the objective function the most. This process continues until a local optimum is reached, where no better solution can be found in the immediate neighborhood. When the algorithm reaches a local optimum, it restarts the search from a random initial solution. The algorithm terminates when a specified number of iterations have been completed or when a solution that satisfies all constraints is found.

A Star is an informed search algorithm widely used for pathfinding and graph traversal tasks. It intelligently combines both actual path costs and heuristic estimates to guide the search towards the goal efficiently. A* maintains a priority queue of nodes to be explored and selects the most promising node based on a combination of the cost incurred so far and the estimated cost to reach the goal. This allows A* to efficiently find the optimal path while intelligently pruning the search space, making it highly effective for solving a wide range of optimization problems.

1.4 Evaluation

Both algorithms will be evaluated based on the quality of the solutions they provide, the time required to find these solutions and the total number of states explored during the search. The quality of the solutions will be evaluated based on the number of constraints satisfied. The time required to find the solutions will be measured in milliseconds, and the total number of states explored will be counted during the search process.

There will be 5 types of tests, describe in *Tests Description* section.

2 Algorithm Design

2.1 State Representation

The state representation for the Class Scheduling Problem consists of a schedule that assigns each course to a room and a time slot. The state is represented as a class that contains the following fields:

– **file_name:** The name of the file from which the data was read.
– **yaml_dict:** A dictionary containing the data read from the file.

- **size:** A tuplet (days, time_slots) representing the size of the schedule.
- **schedule:** A dictionary that has the following structure: {(day: str) : {(time_slot: (int, int)) : {(classroom : str) : ((instructor : str), (subject : str))}}}.
- **strudents_per_subject:** A dictionary that contains the number of students for each subject that needs to be scheduled.
- **count_techer_slots:** A dictionary that contains the number of scheduled slots for each instructor.
- **trade_off:** A number that represents the trade-off between the number of constraints satisfied and the the choosen classroom at each step (used in A Star).

Initial State The initial state is generated as an empty schedule, which is initialized with the parameters derived from the input file, including the number of days, time slots, and classrooms. Additionally, the number of students per subject is obtained from the input file, while the count of scheduled slots for each instructor is set to zero.

This initial state serves as the starting point for the search algorithms, enabling them to iteratively allocate courses to rooms and time slots until a valid schedule is achieved.

An alternative method for generating the initial state involves randomly assigning courses, instructors to rooms and time slots in a manner that adheres to the hard constraints. A comparative analysis of these two approaches will be conducted in the *Initial State Selection* section.

Generating Neighbors The neighbors of a state are generated by considering all possible combinations of assigning a course to a room, a time slot and an instructor, while ensuring adherence to the hard constraints.

In the initial phase, all potential neighbors that adhere to both the hard and soft constraints are generated. Subsequently, in the event that no neighbors satisfying all soft constraints are found, a secondary phase ensues where only neighbors satisfying the hard constraints are generated. This strategy reduces the total number of neighbors generated, allowing the algorithm to prioritize those that satisfy all constraints. Consequently, the algorithm minimizes time wastage by avoiding the generation of neighbors that would not be utilized, resulting in a reduced number of states generated.

Initial State Selection As previously mentioned, the initial state can be generated in two methodologies: either as an empty schedule or through the random assignment of courses, instructors, and rooms to time slots. The former method exhibits a greater degree of determinism, initializing the schedule with vacant slots, whereas the latter introduces stochasticity into the initial state generation process.

Upon experimentation with both approaches, it became evident that the random initialization method could often yield solutions that fail to adhere to all specified soft constraints. This underscores the importance of carefully considering all potential subsequent states that may arise from the current state when designing the random initialization method.

For instance, in instances where an instructor's preferences are violated within a particular time slot, the algorithm should endeavor to substitute instructor with another whose preferences remain unviolated for that time slot. Moreover, the algorithm should endeavor to substitute the time slot with another or substitute the room with anothers that have the same capacity in total. This approach was found more difficult to implement. When tring to implement the first two substitutes, the algorithm was not able to find a solution that satisfies all the soft constraints and it took quite a lot of time to find a partial solution due to a large number of constraints that should be checked while creating the neighbors. On the other hand, this approach will always find a partial solution that satisfies all the hard constraints. Thus, this one is recommended in cases when the soft constraints are not that important.

The method that I used in the end was the empty schedule initialization. It is acknowledged that without the random initialization component, the Hill-Climbing algorithm may struggle to assign all students to a room, thereby violating a hard constraint. However, with the inclusion of the random restart, the algorithm will always find a solution that satisfies all the hard constraints, the soft ones being satisfied in most of the cases. This approach is recommended in cases when the soft constraints are more important than in the previous case and it is easier to generate the neighbors.

2.2 Random Restart Hill Climbing

As previously mentioned, the Random Restart Hill Climbing algorithm is the most suitable for this approach. The algorithm is initialized with an empty schedule and generates neighbors that adhere to the hard constraints. The algorithm iteratively explores the neighborhood of the current state, selecting randomly one of the neighbors that satisfies the most constraints. This process continues until a local optimum is reached, at which point the algorithm restarts the search. The algorithm terminates when a solution that satisfies all constraints is found or when a specified number of iterations / restarts have been completed. A pseudocode of the algorithm is presented in Algorithm **??**.

2.3 A Star

For the A Star algorithm, the state representation is the same as for the Random Restart Hill Climbing algorithm. The algorithm is initialized with an empty schedule and generates neighbors that adhere to the hard constraints. The algorithm iteratively explores the neighborhood of the current state.

The frontier represents a heap that contains the states that need to be explored. The diccovered is a dictionary that contains as keys the number of students that need to be scheduled for each subject and as values the cost of the state that brought us to this configuration.

Algorithm 1 Random Restart Hill Climbing Algorithm

```
1: procedure HILL_CLIMBING($max\_restarts$) return [$is\_final, total\_iters, total\_states, best\_state$]
2:     $total\_iters = 0$
3:     $total\_states = 0$
4:     $best\_state = None$
5:     for $index$ in $range(max\_restarts)$ do
6:         $state = InitialState()$
7:         $is\_final, iters, states, state = stochastic\_hill\_climbing(state)$
8:         $total\_iters += iters$
9:         $total\_states += states$
10:        if $is\_final$ then
11:            return [$is\_final, total\_iters, total\_states, state$]
12:        if $state$ does not have hard constraints then
13:            if $best\_state == None$ or $state$ has less soft constraints
                   unsatisfied than $best\_state$ then
14:                $best\_state = state$
15:    return [$is\_final, total\_iters, total\_states, best\_state$]
```

Algorithm 1 Stochastic Hill Climbing

```
1: procedure STOCHASTIC_HILL_CLIMBING($state, max_iters$) return [$is\_final, total\_iters, total\_states, best\_state$]
2:     $total\_iters = 0$
3:     $total\_states = 0$
4:     while $total\_iters < max\_iters$ do
5:         $total\_iters += 1$
6:         if $state$ is $final$ then return [$True, total\_iters, total\_states, state$]
7:         $neighbors = state.generate\_neighbors()$
8:         $total\_states += len(neighbors)$
9:         if $neighbors == None$ then return [$False, total\_iters, total\_states, state$]
10:        $state = random.choice($from neighbors one of the neighbors with
11:      minimum number of constraints unsatisfied$)$
12:    return [$False, total\_iters, total\_states, state$]
```

The function used in the A Star algorithm is:

$$f(state) = g(state) + h(state) \tag{1}$$

where

$$h(state) = total\ number\ of\ students\ that\ are\ not\ assigned \tag{2}$$

$$g(state) = number\ of\ constraints\ unsatisfied * weight + trade\_off \tag{3}$$

$$trade\_off = \frac{number\ of\ classrooms(subject)}{total\ number\ of\ classrooms} \tag{4}$$

The heuristic function is admisible, because the return value is always less than or equal to the actual cost of the state to reach a final one, and is equal to 0 in the final states. On the other hand, the function h is not consistent, because a state from discovered can be added to the frontier with a smaller cost.

The cost function is calculated as the number of constraints unsatisfied multiplied by a weight and the trade off. In cases where the number of constraints unsatisfied is equal, multiplying it by a weight will prioritize the states that have the trade off smaller. Moreover, when the number of constraints unsatisfied is different, the prioritization will be made based on the number of constraints unsatisfied, not on the trade off.

The trade off is calculated as the number of classrooms that are assigned to a subject divided by the total number of classrooms. We want to prioritize adding a subject to a classroom that has fewer subjects assigned to it. For instance, if we have 2 subjects: A and B, 2 classrooms: 1 and 2, and classroom 1 is assigned to subject A and classroom 2 is assigned to both subjects, while tring to assign a subject to classroom 2 we will choose subject B (trade off = 0.5) instead of subject A (trade off = 1).

The pseudocode of the algorithm is presented in Algorithm **??**.

2.4 Complexities

The complexity of the Random Restart Hill Climbing algorithm is $O(n)$, where n is the total number of iterations. The complexity of the A Star algorithm is $O(b^d)$, where b is the branching factor and d is the depth of the solution.

The complexity of the generate neighbors function is $O(d * t * c * i * s)$, where d is the number of days, t is the number of time slots, c is the number of classrooms, i is the number of instructors and s is the number of subjects. Because the number of days does not exceed 7 (the worst case) and the number of time slots does not exceed 12, the complexity becomes $O(c * i * s)$.

3 Evaluation

3.1 Tests Description

orar_mic_exact: Contains 3 courses, 2 rooms, and 13 instructors. The number of students for each course are 300, 330, and 330. The capacity of the rooms are 20 and 30. The instructors have few preferences.

Algorithm 2 A Star Algorithm

```
1: procedure ASTAR return [$is\_final, total\_iters, total\_states, best\_state$]
2:     $frontier = []$
3:     $discovered = \{\}$
4:     $state = InitialState()$
5:     $frontier.append((f(state), state))$
6:     $discovered[state] = 0$
7:     $total\_iters = 0$
8:     $total\_states = 1$
9:     while $frontier$ do
10:        $current\_state = frontier.pop(1)$
11:        $total\_iters += 1$
12:        if $current\_state$ $is_final$ $==$ $0$ then return [$True, total\_iters, total\_states, current\_state$]
13:        $neighbors = current\_state.generate\_neighbors()$
14:        $total\_states += len(neighbors)$
15:        for $neighbor$ in $neighbors$ do
16:            $new\_cost = g(neighbor) + h(neighbor)$
17:            students_per_subject = neighbor.students_per_subject
18:            if $students\_per\_subject$ $not$ $in$ $discovered$ or $new\_cost < discovered[students\_per\_subject]$ then
19:                $discovered[neighbor] = new\_cost$
20:                $frontier.append((new\_cost, neighbor))$
21:    return [$False, total\_iters, total\_states, current\_state$]
```

orar_mediu_relaxat: Contains 4 courses, 4 rooms, and 18 instructors. The number of students for each course is 660, 660, 665 and 685. The capacity of the rooms is 25, 25, 35 and 70. The instructors have few preferences.

orar_mare_relaxat: Contains 8 courses, 6 rooms, and 37 instructors. The number of students for each course is 470, 475, 475, 495, 500, 530, 535 and 550. The capacity of the rooms is 25, 30, 30, 35, 85 and 85. The instructors have few preferences.

orar_constrans_incalcat: Contains 4 courses, 2 rooms, and 17 instructors. The number of students for each course is 720, 750, 780 and 810. The capacity of the rooms is 15 and 90. The instructors have a lot of preferences.

orar_bonus_exact: Contains 5 courses, 5 rooms, and 23 instructors. The number of students for each course is 500, 510, 515, 520 and 545. The capacity of the rooms is 15, 15, 15, 15 and 50. The instructors have a lot of preferences, including breaks constraints.

3.2 Results

Pentru fiecare dintre algoritmii prezentați mai sus, am folosit *hyperfine* pentru a evalua timpul de rulare al acestora și am salvat datele în câte un fișier MD. Excepție a făcut algoritmul Brute Force, căruia i-am calculat timpul de rulare doar pentru testele 1-2, 3, 9-14, 15-16, deoarece pentru celelalte teste timpul de rulare era prea mare.

Pentru fiecare set de date, am făcut un grafic cu timpul de rulare al fiecărui algoritm, iar rezultatele sunt prezentate în Figurile **??** - **??**.

De asemenea, în tabelul **??** sunt prezentate rezultatele obținute pentru fiecare set de date în raport cu numărul cromatic teoretic al grafurilor (acest număr a putut fi precalculat doar pentru testr 1-26). Pentru anumite teste care au fost prea costisitoare în timp pentru algoritmul Brute Force, am lăsat liber câmul respectiv.

./graphics/fig1.png

Fig. 1. Graficele timpului de rulare pentru grafuri goale

Table 1. Rezultatele obținute pentru fiecare set de date

Nr. set	**Nr. noduri**	**Nr. muchii**	**Nr. cromatic**	**Brute Force**	**Greedy**	**DSatur**	**RLF**
1	887	0	1	1	1	1	1
2	1384	0	1	1	1	1	1
3	10	45	10	10	10	10	10
4	50	1225	50	50	50	50	50
5	100	4950	100	-	100	100	100

3.3 Interpretarea rezultatelor

În raport cu timpul de execuție, avem următoarele observații:

- Algoritmul Brute Force este cel mai lent, **nu** se folosește în practică deloc.
- Algoritmul Greedy pe toate tipurile de grafuri este cel mai rapid.
- Algoritmul DSatur este mai rapid decât algoritmul Brute Force, dar mai lent decât algoritmul Greedy.

./graphics/fig2.png

Fig. 2. Graficele timpului de rulare pentru grafuri complete

./graphics/fig3.png

Fig. 3. Graficele timpului de rulare pentru grafuri bipartite complete

./graphics/fig4.png

Fig. 4. Graficele timpului de rulare pentru arbori binari

`./graphics/fig5.png`

Fig. 5. Graficele timpului de rulare pentru grafuri planare

./graphics/fig6.png

Fig. 6. Graficele timpului de rulare pentru grafuri cu număr de muchii fixat

- Algoritmul RLF pe grafuri goale este mai lent și decât algoritmul Brute Force, pe grafuri planare și arbori binari este mai lent decât DSatur, iar pe restul de grafuri este între Greedy și DSatur (aici au apărut cazuri care în teorie nu sunt adevărate din cauza complexității agoritmului (vezi **??**). Acest lucru se datorează faptului că nu s-au putut genera teste suficient de mari și rulate de un număr suficient de ori pentru a se observa diferența dintre algoritmi.)

În raport cu numărul cromatic, se poate observa cu ușurință faptul că algoritmul Brute Force este un algoritm exact, iar restul sunt aproximări (deși în teste nu se observă o diferență între numărul cromatic teoretic și rezultatul de la algoritmul DSatur, dacă am fi avut un număr mai mare de teste cu numărul cromatic precalculat, ar fi apărut diferența). Dintre toți1 algoritmii euristici, algoritmul DSatur este cel mai exact.

4 Concluzii

În cadrul acestei teme am observat 4 algoritmi diferiți de colorare a grafurilor.

Se poate afirma că nu există o soluție complet generală pentru colorarea grafurilor. Fiecare din cele prezentate mai sus vin cu avantajele și dezavantajele lor.

Algoritmul Brute Force este cel mai lent, dar este exact, algoritmul Greedy este cel mai rapid, dar nu este exact, algoritmul DSatur deși este mai lent decât Greedy, este cel mai exact dintre algoritmi, iar algoritmul RLF este mai lent decât Brute Force și DSatur, dar este aproximativ exact.

În practică:

- Dacă lucrați cu grafice bipartite sau ciclice, algoritmul Greedy nu este suficient de eficient; pe de altă parte, algoritmul RLF este destul de exact și total recomandat pentru aceste cazuri.
- Dacă prioritar este timpul, se poate înclina spre utilizarea algoritmul Greedy.
- Dacă prioritatea este un număr mai apropiat de cel cromatic, trebuie să se utilizeze algoritmul DSatur.

References

1. J. Bondy and U. Murty, Graph Theory - Graduate Texts in Mathematic. Springer, 2008.
2. Z. Àdàm Mann and A. Szajkò, “Average-case complexity of backtrack search for coloring sparse random graphs,” Journal of Computer and System Sciences, vol. 79, no. 8, pp. 1287–1301, 2013.
3. “Backtrack: An o(1) expected time algorithm for the graph coloring problem,” Information Processing Letters, vol. 18, no. 3, pp. 119–121, 1984.
4. N. Barnier and P. Brisset, “Graph coloring for air traffic flow management,” Annals of Operations Research, vol. 130, 03 2002.
5. “The application of a graph coloring method to an examination scheduling problem,” Institute for Operations Research and the Management Sciences (INFORMS), vol. 11, no. 5.
6. University Exam Scheduling System Using Graph Coloring Algorithm and RFID Technology
7. A. Murat and B. Nurdan, “A performance comparison of graph coloring algorithms,” International Conference on Advanced Technology Sciences (ICAT’16), vol. 4, pp. 1–19, 12 2016
8. Z. Mann, “Complexity of coloring random graphs: An experimental study of the hardest region,” Journal of Experimental Algorithmics, vol. 23, pp. 1–19, 03 2018
9. Enunțul problemei
10. M. Garey and D. Johnson, Computer and Intractability: A Guide to the Theory of NP-Completeness, 01 1979.
11. D. Porumbel, J.-K. Hao, and P. Kuntz, “An evolutionary approach with diversity guarantee and well-informed grouping recombination for graph coloring,” Computers Operations Research, vol. 37, pp. 1822–1832, 10 2010.
12. L. Ouerfelli and H. Bouziri, “Greedy algorithms for dynamic graph coloring,” 2011 International Conference on Communications, Computing and Control Applications, CCCA 2011, 03 2011.
13. À. E. Eiben, J. K. Van Der Hauw, and J. I. van Hemert, “Graph coloring with adaptive evolutionary algorithms,” Journal of Heuristics, vol. 4, no. 1, pp. 25–46, 1998.
14. D. Brèlaz, “New methods to color the vertices of a graph,” Commun. ACM, vol. 22, pp. 251–256, 04 1979.
15. R. Janczewski, K. Manuszewski, and K. Piwakowski, “The smallest hard-to-color graph for algorithm dsatur,” Discrete Mathematics, vol. 236, pp. 151–165, 06 2001.
16. GitHub - Jngen: preparing tests made simpler
17. Graph coloring - Wikipedia
18. A Comparison of Parallel Graph Coloring Algorithms
19. A Performance Comparison of Graph Coloring Algorithms
20. Moodle - Analiza algoritmilor