

# *Mini Project: Timetable Scheduling*

---

*February 2023*

FUNDAMENTAL OF OPTIMIZATION

Nguyễn Mạnh Dương

20210243

Nguyễn Quang Tri

20210860

Đoàn Thế Vinh

20210940

# Table of Contents

<b>I</b>	<b>Introduction</b>	<b>2</b>
I.1	Problem statement . . . . .	2
I.2	Mathematical modelling . . . . .	3
I.2.1	Decision variable . . . . .	3
I.2.2	Constraints . . . . .	4
<b>II</b>	<b>Proposed solutions</b>	<b>6</b>
II.1	Backtracking . . . . .	6
II.2	Improved backtracking - Value ordering . . . . .	8
II.3	Improved backtracking - Forward checking . . . . .	10
II.4	Constraint programming . . . . .	12
II.5	Heuristic Methods . . . . .	15
II.5.1	Timetable Scheduling Problems . . . . .	15
II.5.2	Hill Climbing - Iterated Hill Climbing . . . . .	15
II.5.3	Simulated Annealing . . . . .	17
II.5.4	Generate Neighbour Solutions . . . . .	18
II.5.5	Evaluation Function . . . . .	20
<b>III</b>	<b>Result analysis</b>	<b>22</b>
<b>IV</b>	<b>Difficulties during execution</b>	<b>26</b>
<b>V</b>	<b>Conclusion and Possible extensions</b>	<b>27</b>
<b>VI</b>	<b>Task assignment</b>	<b>28</b>

# I Introduction

## I.1 Problem statement

### Original statement

**N** classes: **1**, **2**, ..., **N** need to be assigned. Each class **i** has:

1.  $t_i$ : number of periods needed
2.  $g_i$ : teacher
3.  $s_i$ : number of students.

**M** rooms: **1**, **2**, ..., **M** are available. Each room **i** has  $c_i$  number of seats.

There are **5** scholar days a week, from Monday to Friday, each day is divided into **12** periods. (Morning: **6** periods/ Afternoon: **6** periods).

Create a timetable for those classes satisfying the following constraints: Classes with the same teacher can not be on the same period Number of students need to be smaller than number of seats

### Academic statement

The task involves creating a timetable for **N** classes, numbered **1** to **N**, each of which has specific attributes, namely the number of periods needed ( $t(i)$ ), the assigned teacher ( $g_i$ ), and the number of students ( $s_i$ ). There are **M** rooms available, numbered **1** to **M**, each with a specified capacity ( $c_i$ ) in terms of the number of seats. The timetable is to be structured over **5** scholar days, from Monday to Friday, with each day divided into **12** periods (**6** in the morning and **6** in the afternoon).

The schedule must abide by the constraints that classes with the same teacher cannot be scheduled during the same period, and the number of students must be smaller than the number of seats in the assigned room.

## I.2 Mathematical modelling

### Decision Variables

$$X_{i,m,k,b} \in \{0,1\} \quad \forall (i,m,k,b) \in D_i \times D_m \times D_k \times D_b$$

### Constraints

1.  $D_i = [1, N]$
2.  $D_m = [1, M]$
3.  $D_k = [1, 10]$
4.  $D_b = [1, 6]$
5.  $X_{i,m,k,b} = 0 \Leftarrow s(i) \geq c(m) \quad \forall i \leq N \quad \forall m \leq M \quad \forall k \leq 10 \quad \forall b \leq 6$
6.  $\sum_{m=1}^M \sum_{b=1}^6 X_{i,m,k,b} \in \{t_i, 0\} \quad \forall i \in [1, N] \quad \forall k \leq 10$
7.  $\sum_{m=1}^M \sum_{\substack{i=1 \\ g_i=g}}^N X_{i,m,k,b} \leq 1 \quad \forall g \in G \quad \forall b \leq 6 \quad \forall k \leq 10$
8.  $\sum_{m=1}^M \sum_{\substack{i=1 \\ g_i=g}}^N \sum_{b=1}^6 X_{i,m,k,b} \leq 6 \quad \forall g \in G \quad \forall k \leq 10$
9.  $\sum_{k=1}^{10} \sum_{b=1}^6 X_{i,m,k,b} \in \{t_i, 0\} \quad \forall m \leq M \quad \forall i \leq N$
10.  $\sum_{k=1}^{10} \sum_{b=1}^6 \sum_{m=1}^M X_{i,m,k,b} = t_i \quad \forall i \leq N$
11.  $\sum_{i=1}^N X_{i,m,k,b} \leq 1 \quad \forall k \leq 10 \quad \forall m \leq M \quad \forall b \leq 6$
12.  $\sum_{m=1}^M \sum_{i=1}^N \sum_{\substack{b'=b \\ X_{i,m,k,b}=1}}^{b+t_i-1} X_{i,m,k,b'} = X_{i,m,k,b} \times t_i \quad \forall k \leq 10 \quad \forall b \leq 6$

### I.2.1 Decision variable

Let  $X_{i,m,k,b} \in \{0,1\}$  a boolean decision variable for the option: Assign class (i) to room (m) in period (b) of block (k)

$$\begin{aligned} 1 \leq i \leq N & \quad 1 \leq m \leq M \\ 1 \leq k \leq 10 & \quad 1 \leq b \leq 6 \end{aligned}$$

### I.2.2 Constraints

We will formulate the mathematical equality or inequality based on the constraints mentioned above.

- **The chosen room has enough seats for students**

$$X_{i,m,k,b} = 0 \Leftarrow s_i \geq c_m \quad \forall i \leq N \quad \forall m \leq M \quad \forall k \leq 10 \quad \forall b \leq 6$$

Where:

- $s_i$ : Number of students in class  $i$
- $c_m$ : Number of seats in room  $m$

- **All of the classes can not be assigned in 2 separated blocks**

On each block, if a class ( $i$ ) is assigned, then it is assigned into at least  $t(i)$  periods. Then what we need to do is limit the periods a class can take in a block to  $t(i)$  when it is assigned and 0 when not being assigned.

$$\sum_{m=1}^M \sum_{b=1}^6 X_{i,m,k,b} \in \{t_i, 0\} \quad \forall i \in [1, N] \quad \forall k \leq 10$$

Where:  $t_i$  is the periods needed of chosen class  $i$ .

- **Teacher can not be at the different rooms or classes simultaneously**

Let  $G$  set of all given teachers. The above constraint can be divided into two following conditions:

1. Each teacher can only teach one class in each period

$$\sum_{m=1}^M \sum_{\substack{i=1 \\ g_i=g}}^N X_{i,m,k,b} \leq 1 \quad \forall g \in G \quad \forall b \leq 6 \quad \forall k \leq 10$$

Where:  $g_i$  is the teacher of class  $i$

2. Total periods taught by him/her in each block cannot exceed max periods (6 in this problem)

$$\sum_{m=1}^M \sum_{\substack{i=1 \\ g_i=g}}^N \sum_{b=1}^6 X_{i,m,k,b} \leq 6 \quad \forall g \in G \quad \forall k \leq 10$$

- **A class  $i$  cannot change room during its section**

When a class is assigned to a room in some periods of block, it is necessary to ensure that they will attached each other in exactly  $t(i)$  periods. With the constraint *All classes can not be assigned in 2 separated blocks*, the total periods of a class in a block and in a room assigned equals to exactly  $t(i)$ , which leads to impossibility for the class to appear in 2 different rooms or blocks simultaneously.

$$\sum_{k=1}^{10} \sum_{b=1}^6 X_{i,m,k,b} \in \{t_i, 0\} \quad \forall m \leq M \quad \forall i \leq N$$

Where:  $t_i$  is the periods needed of chosen class (i).

- **Each class has to be assigned once in whole week**

In order to satisfy this condition, the total periods assigned for the given class should be exactly equal to the number of periods it needs.

$$\sum_{k=1}^{10} \sum_{b=1}^6 \sum_{m=1}^M X_{i,m,k,b} = t_i \quad \forall i \leq N$$

Where:  $t_i$  is the periods needed of chosen class i.

- **One room contains only one class at each period**

It could be specified by two following sub-constraints:

1. At every period, there must be less than 1 class assigned to the given room

$$\sum_{i=1}^N X_{i,m,k,b} \leq 1 \quad \forall k \leq 10 \quad \forall m \leq M \quad \forall b \leq 6$$

2. When assigning class  $i$  in the given room to a period, the next  $t_i - 1$  periods need to be blocked.

$$\sum_{m=1}^M \sum_{i=1}^N \sum_{\substack{b'=b \\ X_{i,m,k,b}=1}}^{b+t_i-1} X_{i,m,k,b'} = X_{i,m,k,b} \times t_i \quad \forall k \leq 10 \quad \forall b \leq 6$$

## II Proposed solutions

### II.1 Backtracking

#### Description

Backtracking is a general algorithmic technique to find some (or all) solutions to a problem, where there exists the notion of a solution consisting of different variables—each of which has a domain, or a list of candidates—that can be built incrementally. The algorithm abandons a partial solution, or backtracks, as soon as it determines that given the assigned variables, there are no other acceptable candidates for the next unassigned variable.

By being able to eliminate many candidates at once, backtracking is more time efficient than simple brute-force enumeration.

```
procedure EXPLORE(node n)
  if REJECT(n) then return
  if COMPLETE(n) then
    OUTPUT(n)
  for  $n_i$  : CHILDREN(n) do EXPLORE( $n_i$ )
```

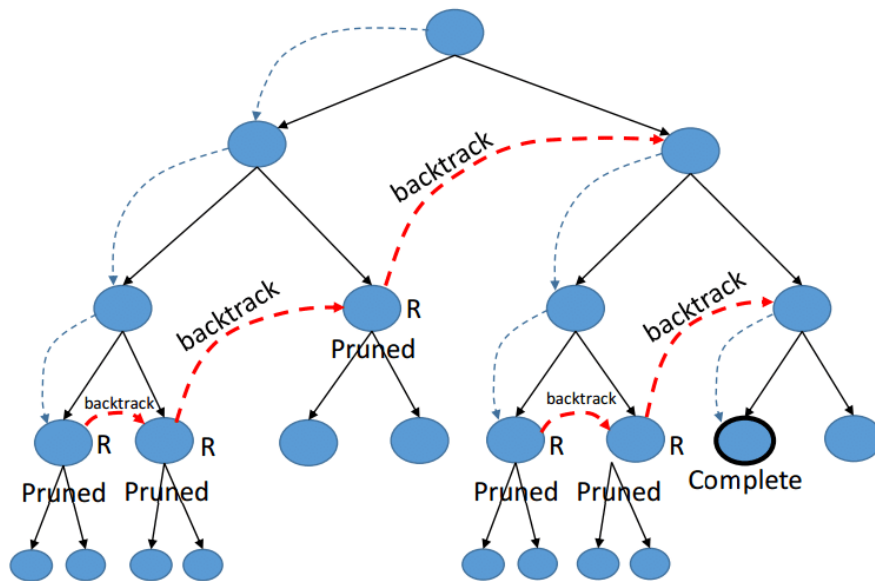


Fig. 1: Backtracking pseudocode and search tree implementation

Backtracking is an important tool for constraint satisfaction problems (CSPs),

which is exactly the type of practical problem introduced in this topic. Since only one valid solution is needed, the algorithm does not need to traverse the entire search tree. As can be seen in the following analysis, backtracking can be extremely time efficient.

## Implementation

```
# class: class.duration, class.professor, class.no_students
# classroom: classroom.capacity

SET (all_classes, all_classrooms) = read_input()

SET all_slots = [(slot.session, slot.period, slot.classroom)
  FOR session from Monday morning to Friday afternoon
  FOR period from 1 to 6
  FOR classroom from 1 to M]

SET timetable = [] * N

FUNCTION assign(class):
  FOR slot in all_slots:
    if slot_is_taken(timetable, slot): CONTINUE
    if not_enough_periods_left_in_session(class.duration, slot.period): CONTINUE
    if below_capacity(class.no_students, classroom.capacity): CONTINUE
    if duplicate_prof(timetable, class.professor): CONTINUE

    timetable[class] <- slot

    IF all_classes_assigned:
      break_recursive_calls()
  END FUNCTION

assign(class <- 1)
```

Fig. 2: Pseudocode for backtracking implementation

Before elaborating on the implementation, it is important to note that the backtracking algorithms do not strictly follow the previously described mathematically modeling, though the underlying idea remains the same. Rather, for simplicity in implementing, the algorithms define variables and constants as follows.

A ‘class’ has three attributes: ‘class.duration’ (suggesting the number of periods it takes up, realistically an integer between 2 and 4), ‘class.professor’ (suggesting the instructor of the class, denoted by an integer ID), and ‘class.no students’ (suggesting the number of students in the class, realistically an integer between 30 and 160).



A ‘classroom’ has one attribute: `class.capacity` (suggesting the number of seats available in the classroom)

Given the practical problem, ‘all slots’ is developed as a list of possible slots that can be assigned to each ‘class’. A ‘slot’ has three attributes: ‘`slot.session`’ (suggesting the session, indexed out of ten, in which the class takes place, from Monday morning to Friday afternoon), ‘`slot.period`’ (suggesting the period within the session, realistically an integer between 1 and 6), and ‘`slot.classroom`’ (suggesting the classroom used). In other words, a slot corresponds to a “time and place” where a class takes place.

In this specific problem, the solution to be achieved consists of  $N$  variables, placed in order in an array. ‘all slots’ acts like the domain for each of these variables, and the value assigned to one is the “starting time and place” of a class. For example, if class 10 is assigned slot (4, 4, 5), this class will start in period 4 of Tuesday afternoon, in classroom 5. It is also important to see that if class 10 takes two periods to complete, slot (4, 5, 5) is also “taken”, even if it is not explicitly assigned to class 10.

With this in mind, we want to obtain a mapping from a class to its slot, or its “starting time and place”. We then use backtracking to traverse ‘all slots’ (the domain of each variable) and incrementally build our solution, starting from class 1. When checking whether a slot is allowed for class  $i$ , we determine (1) whether that slot has been taken by class 1, 2, ...,  $i-1$ , (2) whether that starting time allows for enough periods left in the session to complete the class, (3) whether the classroom can accommodate the number of students, and (4) whether the professor is already teaching in another classroom at the same time.

Once a solution has been found (when all variables have been assigned), the algorithm breaks out of the nested recursive calls, and the ‘timetable’ array of length  $N$  is returned.

## II.2 Improved backtracking - Value ordering

### Description

Value ordering, in the context of constraint satisfaction problems (CSPs) and backtracking algorithms, refers to the order in which the members of the candidate set for each variable are tried. Value ordering is used to optimize the performance of the backtracking algorithm by reducing the number of possibilities that need to be explored. By trying the most promising values first, the algorithm is able to reduce the number of possibilities that need to be explored and make it more likely that a solution will be found quickly. The heuristic used to re-order often cannot, and need not, be exact, but rather simple enough to implement while likely to improve time efficiency in finding a solution.

For this particular practical problem, the intuition comes from an observation that many trials take a long time because a class of only 30 students is assigned a classroom of 160 capacity at the beginning, while larger classes struggle to find a slot.

Thus, the order of slots in ‘all slots’ should be changed according to the class.no students attribute of each class. Specifically, the domain for each variable (each class’ “starting time and place”) is re-organized so that slots with insufficient classroom capacity are dropped, and the remaining are put in order of increasing capacity. This ensures that for each class, the classrooms with the smallest (capacity - no students) difference are prioritized. Smaller classes get smaller rooms, and larger classes get larger rooms.

### Implementation

```
# class: class.duration, class.professor, class.no_students
# classroom: classroom.capacity

SET (all_classes, all_classrooms) = read_input()

SET all_slots = [(slot.session, slot.period, slot.classroom)
  FOR session from Monday morning to Friday afternoon
  FOR period from 1 to 6
  FOR classroom from 1 to M]

SET timetable = [] * N

FUNCTION reorder_slots(all_slots, class):
  remove_insufficient_capacity_slots(all_slots, class.no_students)
  sort_slots_by_increasing_capacity(all_slots)
  RETURN all_slots

FUNCTION assign(class):
  FOR slot in reorder_slots(all_slots, class):
    if slot_is_taken(timetable, slot): CONTINUE
    if not_enough_periods_left_in_session(class.duration, slot.period): CONTINUE
    if duplicate_prof(timetable, class.professor): CONTINUE

    timetable[class] <- slot

  IF all_classes_assigned:
    break_recursive_calls()
END FUNCTION

assign(class <- 1)
```

Fig. 3: Pseudocode for backtracking with value ordering

The majority of the algorithm remains unchanged compared to the normal backtracking algorithm, suggesting a high level of simplicity and maintainability in implementation. What does change is that rather than using one array as the domain for all variables, the algorithm now has to re-compute and store a distinct domain, as described above, for each, suggesting a trade-off between time efficiency and space complexity. Another minor change is since the reordering of a domain already accommodates the constraint of classroom capacity, this test is dropped while backtracking unfolds.

In practical analysis, since value ordering does not increase time complexity by much (since the heuristic is deliberately chosen to be simple to implement) while improving performance significantly, this method is preferred for this problem.

## **II.3 Improved backtracking - Forward checking**

### **Description**

Forward checking, in the context of CSPs, is used to prevent future conflicts. It performs a restricted form of arc consistency to the not yet instantiated variables. Specifically, when a value is assigned to the current variable, any value in the domains of the unassigned variables which conflicts with this assignment, according to the tests specified in the normal backtracking algorithm, is temporarily removed from the domains. When the search backtracks, the domains get reverted back to the original state. The advantage of this is that if the domain of any unassigned variable becomes empty, it is known immediately that the current partial solution is inconsistent. Forward checking therefore allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking.

Note that whenever a new variable is considered, all the remaining values in its domain are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no longer necessary.

## Implementation

```
# class: class.duration, class.professor, class.no_students, class.domain
# classroom: classroom.capacity

SET (all_classes, all_classrooms) = read_input()

SET all_slots = [(slot.session, slot.period, slot.classroom)
  FOR session from Monday morning to Friday afternoon
  FOR period from 1 to 6
  FOR classroom from 1 to M]

FOR class in all_classes:
  class.domain <- all_slots

FUNCTION forward_checking(class, assigned_slot):
  SET deleted = []
  FOR class in remaining_classes:
    FOR slot in class.domain:
      deleted.append(remove_and_return_inconsistent_slots(assigned_slot))
  RETURN deleted

SET assignments = []

FUNCTION assign(class):
  FOR slot in all_slots:
    IF any_empty_domain(classes):
      RETURN False
    assignments.append(slot)
    SET deleted = forward_checking(class, slot)

    IF all_classes_assigned:
      break_recursive_calls()

  revert_assignment_and_restore_domains(assignments, classes, deleted)
END FUNCTION
assign(class <- 1)
```

Fig. 4: Pseudocode for backtracking with forward checking

Here, a ‘class’ has an extra attribute: `class.domain`, as a computational trick to easily update and revert domains as backtracking unfolds. The function ‘forward checking’ removes the inconsistent slots of the domain of a variable, and returns the deleted slots to revert the change once the algorithm backtracks. It is also necessary to note that ‘forward checking’ will parse the domains of the unassigned variables only, rather than all  $N$  variables. In a way, forward checking is a looser deviation of arc consistency, leading to more simplicity to implement.

In the main ‘assign’ function, whenever a slot is assigned to a variable, forward checking is performed, and the entire search tree is pruned immediately if there is

an unassigned variable with an empty domain. 'deleted' is kept as an N-length array of lists, with each list being the slots deleted for the class at that index. 'deleted' is then used to revert the domain back to its previous state so that backtracking can try another slot for the variable in question. Another important change is that the 'assign' function no longer checks the tried slots, since the domains invariably contain only acceptable candidates.

Since forward checking loops through at most N variables and is executed whenever a slot is assigned, this poses a much higher time complexity than simple backtracking. However, generally in theory, this effect is countered by the fact that the search tree is greatly reduced.

## II.4 Constraint programming

Constraint Programming (CP) is a paradigm for solving combinatorial problems that draws on a wide range of techniques from artificial intelligence, computer science, and operations research. In constraint programming, users declaratively state the constraints on the feasible solutions for a set of decision variables. CP is based on feasibility (finding a feasible solution) rather than optimization (finding an optimal solution) and focuses on the constraints and variables rather than the objective function. In fact, a CP may not even have an objective function - the goal may simply narrow down a very large set of possible solutions to a more manageable subset by adding constraints to the problem

### Constraint Satisfaction Problems

A constraint is a relation between multiple variables which limits the values these variables can take simultaneously. Constraint satisfaction is the process of finding a solution through a set of constraints that impose conditions that the variables must satisfy. Problems that can be expressed as constraint satisfaction problems are the eight queens puzzle, the Sudoku solving problem, and many other logic puzzles, scheduling problems,...

**Definition:** A constraint satisfaction problem on finite domain (or CSP) is defined by a triplet  $(X, D, C)$  where:

1.  $X = \{x_1, x_2, \dots, x_n\}$  is the set of variables of the problem
2.  $D = \{D_1, D_2, \dots, D_n\}$  is the set of domains of variables, i.e,  $\forall k \in [1, n]$ , then  $x_k \in D_k$
3.  $C = \{C_1, C_2, \dots, C_n\}$  is the set of constraints. A constraint  $C_i = (X_i, R_i)$  is defined by a set  $X_i = \{x_{i1}, x_{i2}, \dots, x_{ik}\}$  and a relation  $R_i \subset D_{i1} \times D_{i2} \times \dots \times D_{ik}$

During the search for the solutions for CSP, a user can wish for:

- Finding a solution (satisfying all the constraints)
- Finding all the solutions to the problem
- Proving the unsatisfiability of the problem

### Constraint Optimization Problems

A constraint optimization problem (COP) is a constraint satisfaction problem associated with an objective function. An optimal solution to a minimization/ maximization COP is a solution that minimizes/ maximizes the value of the objective function. During the search for the solutions of a CSP, users can wish for:

- Finding a solution (satisfying all the constraints)
- Finding the best solution with respect to the objective
- Proving the optimality of the best found solution
- Proving the unsatisfiability of the problem

### Google OR-Tools

OR-Tools is open source software for combinatorial optimization, which seeks to find the best solution to a problem out of a very large set of possible solutions. OR - Tools includes solver for:

- Constraint Programming (which is used in this study)
- Linear and Mixed-Integer Programming

- Vehicle Routing
- Graph Algorithms

With the supports of OR-Tools, we can generate a constraint programming solving model to generate all feasible solutions.

### Implementation

Our implementation includes 3 fundamental steps, namely Initialization, Add Constraints, and Solving. Since the problem in this study is a standard constraint satisfaction problem then it is unnecessary to add an objective function to the model.

```
CONSTRAINT PROGRAMMING

# Step 1: Initialization

classes = { i: i = 1 --> N }
rooms = { m: m = 1 --> M }
blocks = { k: k = 1 --> 10 }
periods = { b: b = 1 --> 6 } # Available periods in one block
model = ORTools.CP_Model ( )

for i in classes:
    for m in rooms:
        for k in blocks:
            for b in periods:
                define X(i, m, k, b) boolean variable of model

constraints = { f(X(i, m, k, b)): X(i, m, k, b) is decision variable of model}

# Step 2: Add constraints

for c in constraints:
    model.Add ( c )

# Step 3: Solve the problem using solver

solver = ORTools.solver ( )
solutions = solver.solve ( model )

return solutions
```

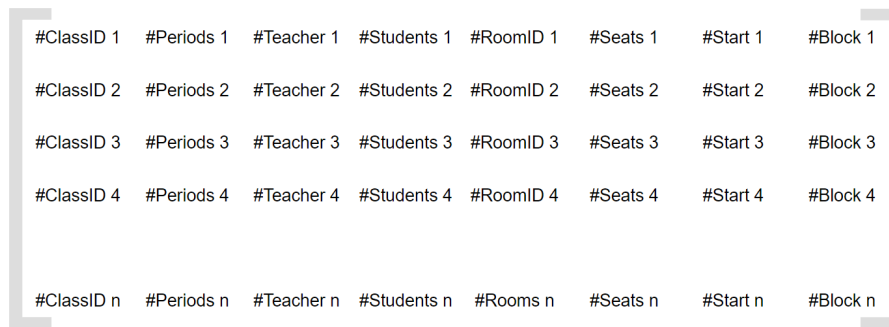
Fig. 5: Pseudocode for Constraint Programming using OR - Tools

## II.5 Heuristic Methods

**Heuristic**, or **heuristic technique**, is any approach to problem-solving or self-discovery that employs a practical method that is not guaranteed to be optimal, perfect, or rational, but is nevertheless sufficient for reaching an immediate, short-term goal or approximation. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution. In this study, Hill Climbing and Simulated Annealing algorithms are used to solve the constraint satisfaction problem.

### II.5.1 Timetable Scheduling Problems

In this study, for the heuristic method, we define our timetable as a list of assignments, which forms a 2D matrix. The general visualization is as follows:



#ClassID 1	#Periods 1	#Teacher 1	#Students 1	#RoomID 1	#Seats 1	#Start 1	#Block 1
#ClassID 2	#Periods 2	#Teacher 2	#Students 2	#RoomID 2	#Seats 2	#Start 2	#Block 2
#ClassID 3	#Periods 3	#Teacher 3	#Students 3	#RoomID 3	#Seats 3	#Start 3	#Block 3
#ClassID 4	#Periods 4	#Teacher 4	#Students 4	#RoomID 4	#Seats 4	#Start 4	#Block 4
#ClassID n	#Periods n	#Teacher n	#Students n	#Rooms n	#Seats n	#Start n	#Block n

Fig. 6: Timetable Formula in Heuristic Method

Specifically, each row is considered as an assignment that contains class ID (ClassID i), the required periods (Periods i), teacher (Teacher i), number of students (Students i), room ID (RoomID i), number of seats (Seats i), start period (Start i), block (Block i).

### II.5.2 Hill Climbing - Iterated Hill Climbing

Hill Climbing is a heuristic search used for mathematical optimization problems which belong to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found.



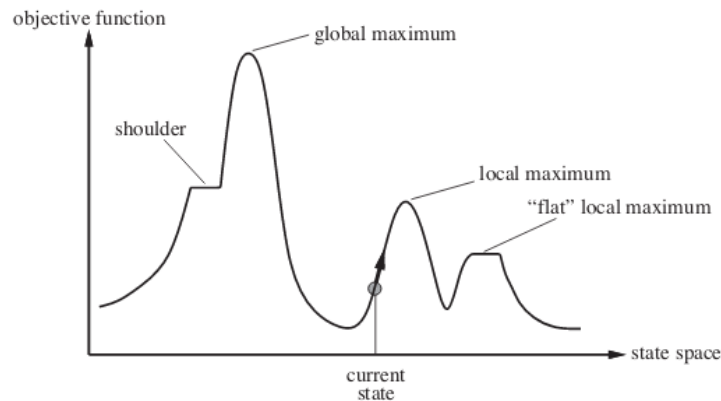


Fig. 7: Hill Climbing searching process

An obvious advantage of the algorithm is its effectiveness in running time to reach an acceptable solution. However, since the method only "climb" in one direction, it somehow limits the search space and might ignore the global optimum. There are some techniques that can improve this drawback such as iterated hill climbing and simulated annealing.

Algorithm 1: Hill Climbing

Hill Climbing ():

```
# Generate the initial solution
s = generate_initial_solution()
current_score = evaluation_function(s)
current_solution = s

neighbour_solutions = generate_neighbours(current_solution)
best_neighbour, best_neighbour_score = get_best_neighbour(neighbour_solutions)

# Start greedy search
while best_neighbour_score > current_score:

    # Get the neighbour solutions and the best one

    current_score = best_neighbour_score
    current_solution = best_neighbour

    neighbour_solutions = generate_neighbours(current_solution)
    best_neighbour, best_neighbour_score = get_best_neighbour(neighbour_solutions)

return current_score, current_solution
```

Fig. 8: Pseudocode for Hill Climbing algorithm

To implement iterated hill climbing techniques, we randomly generated as many initial solutions as possible. The number of starting solutions in reality strongly

depends on the input data size because of time limitations. In our case, that number is usually from 100 to 200 for simple data and from 10 to 15 for more complex situations.

### II.5.3 Simulated Annealing

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem, especially when the search space is discrete. In general manner, SA algorithm adopts an iterative movement according to the variable temperature parameter which imitates the annealing transaction of the metals.

At each step, the simulated annealing heuristic considers some neighboring state  $s^*$  of the current state  $s$  and probabilistically decides between moving the system to state  $s^*$  or staying in state  $s$ . The advantage of this method is to allow the computer to find the global optimal without getting stuck at the local one.

```
Algorithm 2: Simulated Annealing
Simulated Annealing ():
    # Initialization

    current_solution = generate_initial_solution()
    current_score = evaluation_function(current_solution)
    pb = probability_bound # (0 < pb < 1)
    T = initial_temperature

    # Start iteration (step by step)

    for iter in length:
        neighbour_solutions = generate_neighbours (current_solution)
        best_neighbour, best_neighbour_score = get_best_neighbour(neighbour_solutions)

        T* = T / (iter + 1)

        if best_neighbour_score < current_score: probabilistic_function = 1
        else: probabilistic_function = exp[ (current_score - best_neighbour_score) / T* ]

        # Consider the choice of "worse" solution probabilistically

        if probabilistic_function >= pb:
            current_score = best_neighbour_score
            current_solution = best_neighbour

    return current_score, current_solution
```

Fig. 9: Pseudo code of Simulated Annealing algorithm

Specifically, our initial temperature  $T$  is predefined as a constant by many trials and the maximum number of iterations strongly depends on the input size.

After each iteration (or step), the temperature gradually decreases following the rule: the temperature at  $i^{th}$  iteration is  $T_i = \frac{T}{i+1}$  which leads to a change in the probabilistic score. The probabilistic function is determined as:

$$p = \begin{cases} \exp \left[ -\frac{f(x_{i+1}) - f(x_i)}{T_i} \right], & \text{if } f(x_{i+1}) > f(x_i) \\ 1, & \text{otherwise} \end{cases}$$

Where:

1.  $x_i$ : the best solution (candidate) at iteration  $i$
2.  $T_i$ : the temperature at iteration  $i$

Our evaluation function in this study is defined to be minimized, then if  $f(x_{i+1}) < f(x_i)$ , the probability for that solution is 1, i.e, definitely get it. Otherwise, after each iteration, the decreases in temperature result in the fact that  $p$  tends to diminish gradually. In the case that  $p$  does not exceed some probabilistic bound we predefined, some "worse" solutions are considered to be accepted. By this mechanism, the search process might avoid being stuck at the local optimum.

In addition, heuristic methods are usually mentioned with two main questions: How we can keep track of the process to goal, and how we can find a better solution compared to the current one.

#### II.5.4 Generate Neighbour Solutions

Note that all given classes have to be assigned in the feasible timetable, then by fixing class ID with their attributes, the searching process might be more efficient. Generally, in an arbitrary assignment, since ClassID, Periods, Teacher, Students are unchanged and Seats is dependent on RoomID, the straightforward method to generate new solutions is to adjust Room ID, Start and Block which are assigned to the classes. In order to maintain simplification and feasibility, changes can only be applied if the new candidates do not violate constraints that are satisfied by old ones.

```

Algorithm 3: Get_neighbour_solutions

# Change room of the assignment if possible

ChangeRoom(current_timetable):
    new_timetable <- []
    for assignment in timetable:
        temporary_timetable = copy(current_timetable)
        for room in rooms:
            if room != current_room and NumOfStudents <= NumOfSeats:
                temporary_timetable = ReplaceRoom(assignment)
                new_timetable.add(temporary_timetable)
        stop loop
    return new_timetable

# Change start period of assignment (by 1) if possible

ChangeStartPeriod(current_timetable):
    new_timetable <- []
    for assignment in current_timetable:
        temporary_timetable = copy(current_timetable)
        if (end_period < max_period) or (start_period < min_period):
            AdjustPeriodByOne(temporary_timetable)
            new_timetable.add(temporary_timetable)
    return new_timetable

# Change block of assignment (by 1) if possible

ChangeBlock(current_timetable):
    new_timetable <- []
    for assignment in current_timetable:
        temporary_timetable = copy(current_timetable)
        if (min_block < current_block < max_block):
            AdjustBlockByOne(temporary_timetable)
            new_timetable.add(temporary_timetable)
    return new_timetable

# Get all possible solutions from the above methods

GET_NEIGHBOURS(current_timetable):
    new_timetable <- []
    new_timetable.add ( ChangeRoom, ChangeBlock, ChangeStartPeriod )
    return new_timetable

```

Fig. 10: Pseudocode for Neighbour Solutions Generating function

Our function satisfies the constraint of the number of students, the number of seats obviously. Furthermore, it is ensured that the starting period and the

corresponding ending period can not exceed the limitation in one block, and the assigned block has to be allowed in a week. Thus the only unsolved problems which are the foundation for our evaluation function are:

(C1) Teacher is assigned to 2 different classes at the same period, same block

(C2) A room is used for 2 different classes at the same period, same block

### II.5.5 Evaluation Function

The evaluation function or Heuristic function can be considered a mathematical function that evaluates the performance of the process, and also gives our program a direction to the goal node. In general constraint satisfaction problems, a conflict-based function is usually used due to its simplicity and strong relevance to constraint.

```
Algorithm 4: Evaluation Function
Evaluation(Timetable):
    TheScore = 0

    # Consider 2 seperated assignments

    for i and j in Timetable, i != j:
        if Block[i] == Block[j]:

            # A room cannot be used for 2 classes simultaneously
            if Room[i] == Room[j]:
                if Start[j] <= End[i] <= End[j]: TheScore += Penalty
                else: TheScore -= Reward
            else: TheScore -= Reward

            # A teacher cannot turn up at 2 classes simultaneously
            if Teacher[i] == Teacher[j]:
                if Room[i] == Room[j]:
                    if Start[j] <= End[i] <= End[j]: TheScore += Penalty
                    else: TheScore -= Reward
                else: TheScore -= Reward
            else: TheScore -= MaximumReward
    return TheScore
```

Fig. 11: Pseudocode for Neighbour Solutions Generating function

Our evaluation function is strongly related to the neighbor solution-generating function which has been mentioned above. Since the period constraint can not be violated by 2 assignments in 2 different blocks, we only consider those on the same blocks. The idea is to “punish” the program amount of a positive score called a Penalty, and give it a negative score called a Reward. For more details, suppose

there are assignments X and Y in the timetable attached with the same teacher or the same room, which might lead to (C1) or (C2). For instance, X starts from period 1 to 4, and Y starts from period 3 to 6, then the conflict occurs. Otherwise, both assignments are valid. Thus the mission of our evaluation function is to give a Penalty in the case conflicts occur, otherwise give a Reward. Minimization of this score by using some local search methods such as Hill Climbing, Simulated Annealing,... possibly allows us to reach feasible solutions.

### III Result analysis

#### Backtracking and Improved Backtracking

Backtracking relies heavily on the quality of the input. If the input is “easy” to handle (e.g. there are many classrooms of large capacity, or professors do not teach too many classes,...), backtracking is extremely efficient. However, in the opposite case, the algorithm can take a long time, and should there be no available solution, backtracking is not a feasible algorithm to detect this—the number of iterations is exponential.

With this in mind, for testing, we generate the input randomly with 10 seeds—seeds are used to ensure all three algorithms are tested with the same input—to include the situations that the algorithm may and may not be able to handle. We use the normal backtracking algorithm as the control group to analyze value ordering and forward checking.

(N,M)	Time taken					
	(15,3)			(50,8)		
Seed	BT	VO	FC	BT	VO	FC
Seed 1	0,03383	0,00800	0,01622	0,51788	0,08845	0,31993
Seed 2	0,01812	0,01086	0,02080	0,48081	0,08081	0,97014
Seed 3	0,02303	0,00711	0,01877	0,46752	0,07688	0,34867
Seed 4	0,01849	0,00752	0,02473	0,43244	0,09247	0,36291
Seed 5	0,01563	0,00744	0,02551	0,39336	0,07911	0,38441
Seed 6	0,01781	0,00638	0,02015	0,25228	0,09552	0,49434
Seed 7	0,02870	0,00595	0,01613	0,37725	0,07609	0,41151
Seed 8	0,02158	0,00822	0,02109	0,33943	0,12037	0,37384
Seed 9	0,01812	0,00705	0,01662	0,30152	0,11245	0,46219
Seed 10	0,02010	0,00602	0,01764	0,34558	0,11004	0,33197
Average	<b>0,02154</b>	<b>0,00745</b>	<b>0,01977</b>	<b>0,39081</b>	<b>0,09322</b>	<b>0,44599</b>
* N, M: number of classes, rooms respectively * Red-filled: No solution computed in feasible time						

Fig. 12: Results for smaller test inputs, of size (15, 3) and (50, 8)

As can be seen from the table, the algorithms manage to find a solution for all cases, and the ratio between input sizes (50,8) versus (15,3) is also reflected in the times. In both cases, value ordering is the most efficient, decreasing the time needed by 3 to 4 times. It is also interesting to note that forward checking seems to perform worse than simple backtracking and is more irregular in its performance, across different inputs. In this regard, value ordering is more consistent.

(N,M)	Time taken					
	(100,10)			(200,10)		
Seed	BT	VO	FC	BT	VO	FC
Seed 1	2,27007	0,92961	4,12487		0.014116525650	
Seed 2	2,53536	0,59059	4,79845		0.042604923248	
Seed 3		0,49489	25,58242			
Seed 4	2,31622	0,72520				
Seed 5	2,28217	0,72925	2,40787			
Seed 6	2,35962	0,76346	6,29821			
Seed 7	2,22266	0,78716				
Seed 8	2,70895	1,11839				
Seed 9	2,72466	0,64906	4,23945			
Seed 10	2,73712	0,61519	3,20462			
<b>Average</b>		<b>0,74028</b>				
<i>* N, M: number of classes, rooms respectively</i> <i>* Red-filled: No solution computed in feasible time</i>						

Fig. 13: Results for larger test inputs, of size (100, 10) and (200, 10)

Things are more complex when the algorithms have to deal with more “difficult” inputs. Similar to the previous results, backtracking with value ordering still proves to be the most time-efficient algorithm. However, from the table, we also obtain this insight. For difficult inputs with hard-to-locate solutions, value ordering and forward checking has to be used, since normal backtracking cannot find this solution in a feasible time, as can be seen in the seed-3-trial of input (100, 10). However, when a solution is easily available, as in the remaining seeds, normal backtracking and value ordering boast a much more stable performance, while forward checking takes unpredictably longer and fails to give a solution in a feasible time for some cases. For an even harder size of (200, 10), backtracking fails altogether, apart from some exceptionally favorable inputs. Even in these cases, only backtracking with value ordering can locate the solution in a feasible time.

Overall, for among the three backtracking schemes used, it seems that using value ordering is the best measure to obtain a solution. Simplicity in implementation makes backtracking as a whole an effective method for CSPs, and modifications to improve backtracking prove simple enough to be worthwhile for their possible improvement in performance.

Forward checking, though in theory often an improvement over normal backtracking, does not really show benefits here. This suggests that forward checking may be optimal in different scenarios, such as detecting an overall lack of solution for small input. Further tests would be needed to test this claim.

## Hill Climbing and Simulated Annealing

Simulated annealing is one of the probabilistic techniques to improve the hill climbing algorithm. The difference in their performances is not obvious with the simple data input. However, for some more complicated scenarios as the input size increases, simulated annealing seems to reach closer to the goal than the original



hill climbing which is shown by its lower evaluated score. Theoretically, simulated annealing prefers to expand the search space to find the global optimal rather than search in only one direction like hill climbing. Due to this mechanism, it tends to barter time-effectiveness for space-effectiveness. The mentioned theoretical attribute is somewhat indicated as follows:

Input size (N, M)	Hill Climbing		Iterated Hill Climbing		Simulated Annealing (300 steps)	
	Time	Score	Time	Score	Time	Score
(15, 3)	0.05s	-3140	5.73s	-3140	2.05s	-3140
(50, 8)	6.73s	-36210	1085.50s	-36296	57.91s	-36292
(100, 10)	82.81s	-146920	1180.77s	-146966	1110.73s	-146990
<i>* N, M: number of classes, rooms respectively</i> <i>* Color - filled: Found at least 1 feasible solution</i>						

Fig. 14: Hill Climbing and Simulated Annealing Comparison

As can be observed from the above table, both original hill climbing and its improvements reach an optimum (-3140) in which iterated and simulated annealing techniques increase the time needed by 114 and 41 times, respectively for the most simple test (15, 3). However, in other complicated situations such as (50, 8) and (100, 10) of input size, both techniques seem to have an outstanding performance with a more negative value compared to the original one. Interestingly, for our last test, simulated annealing's performance surpasses the remaining methods in the minimal value (-146990) in the feasible time (1110.73s).

## Constraint Programming and its efficiency

All the methods mentioned above have some disadvantages despite their time efficiency. In detail, backtracking and its improved version rely heavily on the quality of the input, not only the size. If there exists at least one feasible solution for the problem with some given domains, the backtracking algorithm continuously searches for only one solution and then stops, which explains its outstanding performance in the time-consuming aspect. It is, however, suitable for constraint satisfaction problems due to the nonexistence of objective function. Besides that, heuristics, or conflict-based heuristics in our study, are inexact methods that try to satisfy constraints as much as possible. To some extent, the feasibility of solutions by heuristics is usually unexpected and strongly depends on the evaluation function, which is one of the most crucial parts of the local search family.

Our constraint programming model (CPM) is set up to find all the feasible solutions and output one of them. Things are more complex for all the methods

when the input data is more "difficult" to solve, i.e, size increases and quality decreases. In this case, CPM is more effective to come to conclusion compared to the enumeration of all possible solutions of backtracking

Input Sample (N, M)	Backtracking	FC	VO	Constraint Programming	Hill Climbing	Simulated Annealing
(15, 3)	0.022s	0.008s	0.021s	0.69s	0.15s	1.9s
(50, 8)	0.458s	0.084s	0.477s	4.78s	9.24s	26.59s
(100, 10)			0.740s	449.5s	143s	200s
<i>* N, M: number of classes, rooms respectively</i> <i>* Color - filled box: Found at least 1 feasible solution</i>						

Fig. 15: A comparison between methods on different sample sizes

As shown in the above table, we surprisingly observed that the time complexity of CPM surpasses that of other methods. There are some possible explanations for this phenomenon. In detail, CPM is programmed to enumerate all feasible solutions while others only try to find one leading to its being extremely time-consuming. Another explanation is the fact that we use too many independent factors  $(i, m, k, b)$  to formulate the decision variable  $X_{i,m,k,b}$  which might be unnecessary for solving the problem. Our recommendation for this drawback will be discussed in the later section.

## IV Difficulties during execution

During the finalization of the report, the research team encountered various challenges in the areas of data collection, algorithm implementation, and optimization of the performance of the individual algorithms.

The foremost issue faced by the team was related to the mathematical modeling of the problem. Owing to the different approaches used by each member of the team in terms of the number of constraints, the number of variables, and the interpretation of each variable, arriving at a consensus model proved to be a complex task. Ultimately, a unified model was agreed upon and documented in the report.

The first algorithm recommended by the team was backtracking, which demonstrated accurate results for smaller test cases. However, for larger test cases, the algorithm exhibited prolonged execution time. As such, the team proposed two additional algorithms, namely value ordering and forward checking, to improve the performance of the backtracking algorithm.

One more challenge faced by the team pertained to algorithm diversity. In order to address this, the team undertook an exploration of alternative algorithms to incorporate more diverse approaches to the problem.

## V Conclusion and Possible extensions

### Conclusion

The results have indicated a notable dependence between the performance of algorithms and the input data. Specifically, with small instances, all methods - including the heuristic approaches - yielded the correct results. However, as the size and complexity of the input data increased, the efficiency of constraint programming emerged as superior in comparison to smaller inputs.

### Possible extensions

In this report, we have explored and evaluated **06 (six)** different algorithms to solve this problem: backtracking, value ordering (improved backtracking), forward checking (improved backtracking), constraint programming, simulated annealing, and hill climbing. But the problem of scheduling still has more efficient approaches to be considered. In the near future, we are planning to:

1. Seeking to enhance the developed methods by implementing diverse algorithms utilizing advanced mathematical models that incorporate a greater number of constraints.
2. Furthermore, it is recommended that further testing be conducted, which includes an expansion of the time limit to evaluate the performance of the approaches on problem instances of larger sizes.
3. Additionally, improvements to the evaluation and generate-neighbors functions are proposed.
4. Since some independent factors in constraint programming are redundant as mentioned above, it might be more efficient to reduce the number of variables. For example,  $k^{th}$  block can be removed and calculated from periods with some corresponding changes in constraints.
5. Outside the scope of the current problem, it is also intended to apply these methods to more complex constraint satisfaction problems, such as general scheduling and game-solving.

## VI Task assignment

Throughout the process of finalizing the report, each member has made contributions to the overall outcome. A comprehensive list of the specific contributions made by each member is documented below:

1. Nguyen Manh Duong - 20210243:
  - (a) Mathematical modeling
  - (b) Constraint programming coding
  - (c) Hill climbing coding
  - (d) Simulated annealing coding
2. Nguyen Quang Tri - 20210860:
  - (a) Data generator coding
  - (b) Data processor coding
  - (c) Presentation slide
3. Doan The Vinh - 20210940:
  - (a) Backtracking coding
  - (b) Improved backtracking - Value ordering coding
  - (c) Improved backtracking - Forward checking coding