



Class - Course - Teacher Assignment and TimeTabling



Group 1

Group members:

Phạm Đức Duy - 20235588	Exact method + Experimental Results
Ngô Mạnh Hiếu - 20235590	Greedy
Dương Ngô Hoàng Vũ - 20235629	Hill climbing
Hoàng Quốc Huy 20235594	Tabu Search

Content

01

**Problem
Description and
Application**

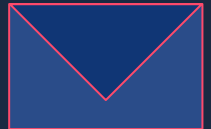
03

**Experimental
Results**

02

Algorithms

1. Greedy
2. Exact Method (Constraint programming)
3. Hill Climbing
4. Tabu Search





Problem Description



Problem Description

- Có T giáo viên, M môn học, và N lớp học.
- Mỗi lớp học có một danh sách các môn lớp đó cần học.
- Mỗi môn học có số tiết $d(m)$
- Mỗi giáo viên có danh sách các môn có thể dạy.
- Có 5 ngày học, mỗi ngày chia thành 2 buổi (sáng và chiều), mỗi buổi gồm 6 tiết học.
- Phân công giáo viên dạy các lớp-môn sao cho:
- Các lớp-môn trong cùng một lớp không được xếp thời khóa biểu chồng chéo.
- Các lớp-môn phân công cho cùng một giáo viên không được xếp thời khóa biểu chồng chéo.
- Số lớp-môn được phân công cho giáo viên là lớn nhất, tối đa hóa số lớp-môn được phân công.
- In ra thời gian biểu phân công giáo viên và các lớp-môn, đảm bảo các điều kiện

Input

- Dòng 1: T (số giáo viên), N (số lớp), M (số môn) ($1 \leq N \leq 100, 1 \leq M \leq 100, 1 \leq T \leq 100$)
- Dòng $i+1$ ($i = 1, \dots, N$): ghi danh sách các môn mà lớp i cần phải học (kết thúc bởi 0)
- Dòng thứ $t + N + 1$ ($t = 1, 2, \dots, T$): ghi danh sách các môn mà giáo viên t có thể dạy (kết thúc bởi 0)
- Dòng thứ $N + T + 2$: ghi $d(m)$ là số tiết của môn m ($m = 1, \dots, M$)

Output

- Dòng 1: ghi số nguyên dương K
- Dòng $k + 1$ ($k = 1, \dots, K$): ghi 4 số nguyên dương x, y, u, v trong đó lớp-môn $x-y$ được phân vào kíp bắt đầu là u và giáo viên dạy là v

Example

Input	Output
3 5 4	12
2 4 0	1 2 1 2
2 3 4 0	1 4 7 3
2 3 0	2 2 1 3
1 2 4 0	2 3 7 1
1 3 0	2 4 13 3
1 3 0	3 2 7 2
2 3 0	3 3 1 1
1 2 4 0	4 1 5 1
2 4 4 4	4 2 13 2
	4 4 19 3
	5 1 5 3
	5 3 13 1

REAL-WORLD APPLICATIONS

01

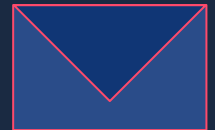
Educational management
Business Management

02

Event organization

03

Online education system





Greedy Method



Tại sao nghĩ đến việc sử dụng thuật toán Greedy?

-Thuật toán **Greedy** (Tham lam) là một cách tiếp cận rất phổ biến trong các bài toán lập lịch (**scheduling problems**) vì nó giúp ta đưa ra **quyết định tối ưu cục bộ tại từng bước** mà không cần quay lui hay thử tất cả các khả năng như **Brute Force**.

1. Feasibility Check(can_assign)

2. Iterate to assign Class-Subjects

3. Output results

Input parsing:

T:teachers

N:classes

M:Subjects

Class_subjects: List of subjects required by each class.

Teacher_subjects: List of subjects each teacher can teach.

Subject_periods: The number of periods required for each subject.

Total_Hours = $5 \times 2 \times 6 = 60$ periods per week

Data structures:

Schedule: A $N \times 60$ matrix to track the schedule of each class(initialized as False for all periods)

Teacher_schedules: A $T \times 60$ matrix to track of the schedule of the each teacher

```
# Initialize schedules
total_periods = 5 * 2 * 6 # 5 days * 2 sessions * 6 periods
schedule = [[False] * total_periods for _ in range(N)] # Class schedule
teacher_schedules = [[False] * total_periods for _ in range(T)] # Teacher schedules
```

1. Feasibility Check(can_assign):

This function checks:

1. Whether the class schedule allows assigning the subject at the given time
2. Whether the teacher's schedule allows it

```
#PYTHON
def can_assign(schedule, class_id, subject, start_time, periods, teacher_schedule):
    end_time = start_time + periods - 1
    for t in range(start_time, end_time + 1):
        if schedule[class_id][t] or teacher_schedule[t]:
            return False
    return True
```

2. Assign Class-Subjects

1. Iterate over each class.
2. Iterate for each subject required by the class:
 - +Determine the number of periods needed.
 - +Iterate through each teacher who can teach the subject.
 - +Try all possible start times in the timetable.
3. If a valid assignment is found:
 - +Update the class and teacher schedules.
 - +Add the assignment to the assignments list.

```
# Store assignments
assignments = [] # (class_id, subject, start_time, teacher_id)
```

1. Iterate all possible start times
2. If `can_assign` return True, it means this class can be assigned at that time

Example:

Giả sử chúng ta có 3 giáo viên, 2 lớp học, và 3 môn học. Lớp 1 cần học môn 1 và 2, lớp 2 cần học môn 2 và 3. Số tiết của môn 1 là 2, môn 2 là 2, môn 3 là 2. Sau khi áp dụng thuật toán Greedy, ta sẽ có kết quả như sau:

- (1, 1, 1, 1) - Lớp 1, môn 1, bắt đầu từ tiết 1, giáo viên 1.
- (1, 2, 3, 1) - Lớp 1, môn 2, bắt đầu từ tiết 3, giáo viên 1.
- (2, 2, 1, 2) - Lớp 2, môn 2, bắt đầu từ tiết 1, giáo viên 2.
- (2, 3, 3, 2) - Lớp 2, môn 3, bắt đầu từ tiết 3, giáo viên 2.

3. Output results

1. Print the total number of assignments.
2. Print the details of each assignments.

```
# Output results
print(len(assignments))
for class_id, subject, start_time, teacher_id in sorted(assignments):
    print(class_id, subject, start_time, teacher_id)
```


4. Solution Evaluation

1. Advantages:

- +Ensures valid schedules with no conflicts.
- +Maximizes the number of class-subject assignments.

2. Drawbacks:

- +Accuracy



Exact Method

(Constraint programming)



Data parsing

T : number of teachers

N : number of classes

M : number of subjects

$class_subjects[c]$: list contain subjects that class c need to study

$subject_teachers[m]$: list contain teachers that teach subject m

$subject_periods[m]$: number of periods of subject m

$TOTAL_PERIODS = 5 * 2 * 6 = 60$: number of periods in a week



Variables

for c in $[1, N]$:

for m in $class_subjects[c]$:

for h in $range(TOTAL_PERIODS)$:

$$x[(c, m, h)] = \begin{cases} True = 1, & \text{if class } c \text{ study subject } m \text{ at period } h \\ False = 0, & \text{otherwise} \end{cases}$$

for c in $[1, N]$:

for m in $class_subjects[c]$:

for t in $subject_teachers[m]$:

$$y[(c, m, t)] = \begin{cases} True = 1, & \text{if class-subject } c-m \text{ is assigned to teacher } t \\ False = 0, & \text{otherwise} \end{cases}$$



Given c, m

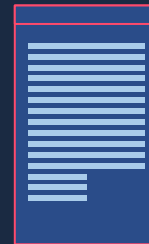
$\Rightarrow \text{sum}(y[(c, m, t)] \text{ for } t \text{ in } \text{subject_teachers}[m])$: number of teachers that class–subject $c-m$ is assigned to

Given c, h

$\Rightarrow \text{sum}(x[(c, m, h)] \text{ for } m \text{ in } \text{class_subjects}[c])$: number of subjects that class c study at period h

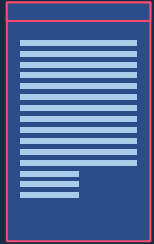
Given c, m

$\Rightarrow \text{sum}(x[(c, m, h)] \text{ for } h \text{ in } \text{range}(\text{TOTAL_PERIODS}))$: number of periods that class–subject $c-m$ was taught



Objective function

Maximize : $\sum(y[(c, m, t)])$
for c in $[1, N]$
for m in $class_subjects[c]$
for t in $subject_teachers[m]$)



Constraints

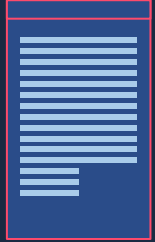
1. Each class-subject must be assigned to exactly one teacher



```
for  $c$  in  $[1, N]$ :  
    for  $m$  in  $class\_subjects[c]$ :  
         $sum(y[(c, m, t)] \text{ for } t \text{ in } subject\_teachers[m]) \leq 1$ 
```

Constraints

2. Each class-subject must be scheduled exactly the required number of periods, and those periods should be taught by the assigned teacher



for c in $[1, N]$:

for m in $class_subjects[c]$:

$A = \text{sum}(x[(c, m, h)] \text{ for } h \text{ in } \text{range}(TOTAL_PERIODS))$

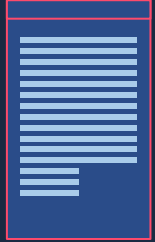
$B = \text{sum}(y[(c, m, t)] \text{ for } t \text{ in } \text{subject_teachers}[m])$

$A == B * \text{subject_periods}[m]$

Constraints

3. A class can only study one subject at a time

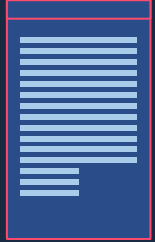
```
for  $c$  in  $[1, N]$ :  
    for  $h$  in  $range(TOTAL\_PERIODS)$ :  
         $sum(x[(c, m, h)] \text{ for } m \text{ in } class\_subjects[c]) \leq 1$ 
```



Constraints

4. A teacher cannot teach multiple subjects or classes at a time

```
for  $t$  in  $[1, T]$ :  
    for  $h$  in  $range(TOTAL\_PERIODS)$ :  
         $sum(x[(c, m, h)] * y[(c, m, t)]$   
            for  $c$  in  $[1, N]$   
            for  $m$  in  $class\_subjects[c]) \leq 1$ 
```



Constraints

5. Enforce continuous periods for each subject in each class

```
for  $c$  in  $[1, N]$ :  
  for  $m$  in  $class\_subjects[c]$ :  
     $subject\_duration = subject\_periods[m]$   
    for  $start\_period$  in  $range(TOTAL\_PERIODS - subject\_duration)$ :  
      if  $c - m$  start at  $start\_period$ :  
         $x[(c, m, h)] = True$  for  $h$  in  $[start\_period, start\_period + subject\_duration)$ 
```





Hill Climbing Method



Input data

T : number of teachers

N : number of classes

M : number of courses

$class_subjects$: list of subjects for each class

$teacher_subjects$: list of subjects each teacher can teach

$duration$: class time for each subject

Check validity in timetable

MAX_SLOT: 60 slots, each slot corresponds to 1 lesson.

Objective:

- Ensure that classes, teachers, and times are not duplicated.
- Check for validity before assigning a schedule.

Validity Check (is_valid_assignment):

Check conditions:

- Slot does not exceed the limit (60).
- Teacher teaches the subject.
- Classes do not have duplicate times.
- Teachers do not have duplicate schedules.

(Illustration code)

```
def is_valid_assignment(...):  
    if start_slot + d - 1 > MAX_SLOT: return False  
    if y not in teacher_subjects[v]: return False  
    for s in range(start_slot, start_slot + d):  
        if occupied_class[x][s] or occupied_teacher[v][s]: return False  
    return True
```

Schedule Acceptance and Cancellation

Record the schedule

(apply_assignment)

- Assign slots and teachers to the **solution**.
- Mark used slots.

Illustration code

```
def apply_assignment(...):  
    solution[(x, y)] = (start_slot, v)  
    for s in range(start_slot, start_slot + d):  
        occupied_class[x][s] = True  
        occupied_teacher[v][s] = True
```

Cancel schedule

(remove_assignment)

- Remove the assignment from the **solution**.
- Release the marked slots.

Illustration code

```
def remove_assignment(...):  
    start_slot, v = solution[(x, y)]  
    for s in range(start_slot, start_slot + d):  
        occupied_class[x][s] = False  
        occupied_teacher[v][s] = False  
    solution[(x, y)] = None
```

Scoring Function: Solution Evaluation

1. Objective:

- Measure the effectiveness of the solution: Number of **classes-subjects** successfully scheduled.
- Provide a basis for comparison when searching for the optimal solution.

2. How it works:

- Iterate through all pairs (**class, subject**) in the solution.
- Count the number of pairs with values other than **None** (ie, have been scheduled properly).

The higher the score => the better the solution.

```
def compute_score(solution):  
    count = 0  
    for k, val in solution.items():  
        if val is not None:  
            count += 1  
    return count
```


Initial Solution Creation

1. Goal:

- Create an initial solution to start the optimization process.
- Try to assign classes to valid slots and teachers.

2. How it works:

- Step 1: Iterate through all classes in random order.
- Step 2: For each class:
 - Select appropriate teachers and random slots.
 - Use the **is_valid_assignment** function to check for validity.
 - If valid, record it in **solution** and update the **occupied** status

(Illustration code)

The initial solution is not optimal but valid to start the improvement process.

```
def initialize_solution(T, N, M, class_subjects, teacher_subjects, duration):
    solution = {}
    occupied_class = [[False]*(MAX_SLOT+1) for _ in range(N+1)]
    occupied_teacher = [[False]*(MAX_SLOT+1) for _ in range(T+1)]

    # Duyệt ngẫu nhiên
    all_class_sub = [(x, y) for x in range(1, N+1) for y in class_subjects[x]]
    random.shuffle(all_class_sub)

    for (x, y) in all_class_sub:
        # Thử tất cả slot và giáo viên ngẫu nhiên
        ...
        solution[(x, y)] = None # Nếu không xếp được
    return solution, occupied_class, occupied_teacher
```

Get Neighbor Generation

1. Main idea:

- Create a "neighbor" solution from the current solution by applying small changes (moves).

2. Types of Moves:

- *Change slot*: Change the start time for an assigned class-subject.
- *Change teacher*: Change the teacher of an assigned subject.
- *Assign unassigned class-subject*: Try to assign an unassigned class-subject.

(Illustration code)

3. How to do it:

- Randomly select a possible move.
- Check the validity before applying the move.
- If there is no possible move, keep the current solution.

```
def get_neighbor_solution(...):  
    # Copy the current solution and occupancy data  
    ...  
    if move == "move_slot":  
        # Change start time of an assigned class-subject  
        ...  
    elif move == "move_teacher":  
        # Change teacher of an assigned class-subject  
        ...  
    elif move == "assign_unassigned":  
        # Try assigning an unassigned class-subject  
        ...  
    return new_solution, new_occupied_class, new_occupied_teacher
```

Hill Climbing

1. Idea:

- Optimize the solution by gradually improving it through neighbor generation.

2. Steps:

- Initialize: Start with the initial solution.
- Iterate:
 - Generate neighbors from the current solution.
 - If the neighbors are better, update the solution.
- Stop: When the maximum number of iterations is reached or no further improvement is made.

(Illustration code)

```
def hill_climbing(...):  
    solution = initialize_solution(...)  
    best_solution = solution  
    best_score = compute_score(solution)  
  
    for _ in range(max_iterations):  
        neighbor = get_neighbor_solution(...)  
        if compute_score(neighbor) > best_score:  
            best_solution = neighbor  
            best_score = compute_score(neighbor)  
    return best_solution
```



Tabu Search Method



1. Input structure

T : number of teachers

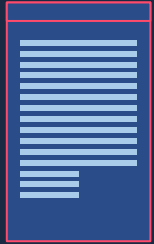
N : number of classes

M : number of subjects

Class_subjects : The dictionary contains a list of subjects to be taught for each class.

Teacher_subjects : The dictionary contains a list of subjects that each teacher can teach.

Subject_periods : The dictionary contains the number of periods required to teach each subject.



```
class_subjects = {  
    1: [2, 4],      # Lớp 1 cần học môn 2 và môn 4  
    2: [2, 3, 4],  # Lớp 2 cần học môn 2, môn 3, và môn 4  
    3: [1, 3]      # Lớp 3 cần học môn 1 và môn 3  
}
```

2.Tabu_search_with_heuristic()

- `Total_slots = 60`
- `Current_solution` : Current solution (list of assigned classes).
- `Best_solution` : The best solution is found throughout the entire process.
- `Best_K` : Number of classes assigned in the best solution
- `Tabu_list` : List of neighboring solutions prohibited to avoid repetition.
- `Generate_neighbors(solution)`
- `Is_feasible(solution)`

2.1.Generate_neighbors(solution)

- **Add unassigned classes** : For each unassigned class, try creating a new assignment with possible teachers and start times.

```
unassigned = []
assigned = {(x, y) for x, y, _, _ in solution}
for class_id, subjects in class_subjects.items():
    for subject_id in subjects:
        if (class_id, subject_id) not in assigned:
            unassigned.append((class_id, subject_id))
```

```
for class_id, subject_id in unassigned[:min(len(unassigned), 5)]:
    for teacher_id in teacher_subjects:
        if subject_id in teacher_subjects[teacher_id]:
            for start_time in range(1, total_slots - subject_periods[subject_id] + 1, 3):
                new_assignment = (class_id, subject_id, start_time, teacher_id)
                neighbors.append(solution + [new_assignment])
```

- **Delete existing assignments** : Delete some assignments from the current solution to create simpler neighbors.


```
if len(solution) > 0:
    for i in range(min(len(solution), 5)):
        neighbors.append(solution[:i] + solution[i + 1:])
```

2.2.Is_feasible(solution)


- **Check teacher's ability** : Teachers must be able to teach the subject.
- **Check if the subject belongs to the class** : The subject must be in the class's subject list.
- **Check for timing conflicts** :
 - + Classes cannot have two subjects at the same time.
 - + Teachers cannot have two classes at the same time.

3.Tabu Search loop

- **Generate neighbors** : Call `generate_neighbors()` function to generate a list of neighbor solutions.
- **Filter Feasible Neighbors** : Eliminate infeasible solutions using `is_feasible`.
- **Select Best Solution** : Select the best solution from the neighbors, not in the Tabu list (`tabu_list`). If the neighbor list is empty, continue the loop.
- **Update Tabu List** : Add new solution to the Tabu list.If the list exceeds `tabu_tenure`, delete the oldest element.
- **Update Best Solution** : If the current solution is better (`len(current_solution) > best_K`), update `best_solution`.



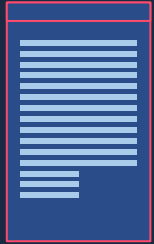
Experimental Results



Exact Method : Slow but highest accuracy

Greedy & Hill Climbing : Fast, good enough accuracy

Tabu search : ???





**Thank you
for listening**



Red Blue White Blue

Blue White Blue Blue Red