Topic 8

February 2023

Fundamental of Optimization

Bùi Minh Quang
Trịnh Thái Dương
Phạm Quang Tùng

Table of Contents

I Problem description

The problem is about arranging the schedule of exams of N subjects at M rooms satisfying some constraints.

- Given d(i) is the number of students participating in the subject i, with i = 1, ...,N.
- Give c(j) is the capacity of room j, with j = 1, ...,M.
- We also have a list of (i,j), where i, j are subjects that can't be arranged in same period.

The task is to set up a timetable such that all subjects is arranged in suitable rooms and suitable period to minimize the number of periods.

II Modelling

Notations

- n is the number of subjects.
- m is the number of rooms.
- $S = \{1, 2, \ldots, N\}$: the set of subjects.
- $R = \{1, 2, \ldots, M\}$: the set of rooms.
- $D = \{d(i) | i \in S\}$: the set of number of students participating of subjects.
- $C = \{c(j) | j \in R\}$: the set of capacities of rooms.
- $P = \{(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)\}$: the list of pairs of subjects that can't be arranged in the same period.

Variables

$$X(i,j,k) = \begin{cases} 1, \text{if there is exam of subject } i \text{ in period } j \text{ at room } k \\ 0, \text{otherwise} \end{cases}$$

Constraints

We assume that the maximum number of periods is equal to the number of subjects since the worst case is that each period only have one subject.

- $X(i,j,k) = 0, \text{if } D(i) > C(k), \forall j \in \{1, 2, \ldots, n\}$

  This constraint guarantee that if capacity of room k is less than the number of students participating in subject i, subject i can't be arranged in room k in every periods.

- $\sum_{j=1}^{n} \sum_{k=1}^{m} X(i,j,k) = 1, \forall i \in S$

  This constraint guarantee that each subject is arranged in a particular room and a particular period.

- $\sum_{i=1}^{n} X(i,j,k) \leq 1, \forall j \in \{1, 2, 3, \ldots, n\}, \forall k \in R$

  This constraint guarantee that each room in a particular period can set up only one subject.

- $\sum_{j=1}^{n} (X(i_1, j, k_1) + X(i_2, j, k_2)) \leq 1, \forall (i_1, i_2) \in P, \forall k_1, k_2 \in R$

  This constraint guarantee that two subjects in P cannot be arranged in the same period.

Objective function

The function represent the number of periods of a particular schedule is:

$$f = \sum_{i \in P} i, \text{where } P(j) = \max\{X(i, j, k) | \forall i, k\}$$

, where P is n-element list of boolean values.

Our job now is to optimize(minimize) the objective function.

III Algorithms used

III.1 CP-SAT

Overview

OR-Tools is an open source software suite for optimization, tuned for tackling the world's toughest problems in vehicle routing, flows, integer, constraint programming. And one of the most suitable tools to solve this problem is the CP-SAT solver, a constraint programming solver that uses SAT - based methods. We only need to define necessary variables, add all constraints to the model and the tool will do the rest.

Properties

- Completeness: The CP-SAT solver is complete, meaning that it will find an optimal solution if one exists.
- Running time: The algorithm's computational time is low but increase rapidly when it comes to large input.
- Determinism: The CP-SAT solver is non-deterministic, meaning that it might produce different results each time with the same input if there is more than 1 optimal solution.
- Efficiency: The CP-SAT solver can also be an efficient algorithm because it gives optimal solution in a small amount of time. Although it works slower with big input, we still can apply some methods to support it.

Pseudo Code

Algorithm 1: CP-SAT Algorithm

```
Input: standard input with subjects, rooms and conflicted pairs
Output: a schedule for examination
// Create the model
model = cp_model.CpModel()
// Define variables
x[i][j][k]: subject i, period j, room k
// Add constraints
model.Add(Constraint 1, 2, 3 ...)
// Add one more constraint to optimize the objective function
   constraint: number of subjects in previous period >= number of
   subjects in current period
model.Minimize(Objective function)
solver = cp_model.CpSolver()
status = solver.Solve(model)
if status == cp_model.OPTIMAL then
    return solution
else
    print("no optimal solution found")
```

III.2 Brute Force

Overview

Although the time complexity of this algorithm is exponential, but it always give optimal solution. Besides, we also come up with an idea that apply a little bit of Greedy algorithm to reduce the running time and optimize for the solution and many more algorithms could be combined to improve our original Brute Force.

Properties

- Completeness: The brute force algorithm guarantees to find the optimal solution, if it is run to completion.
- Running time: The running time of the brute force algorithm is exponential in the number of rooms and subjects, making it infeasible for large instances of the problem.
- Determinism: The brute force algorithm is deterministic, meaning that it produces the same result each time it is run with the same input.
- Efficiency: The brute force algorithm is inefficient. Despite optimal solution it provided, but to be practical, it can not handle the large amount of subjects and rooms in a particular university.

Pseudo Code

Algorithm 2: Brute Force Algorithm

```
Input: standard input with subjects, rooms and conflicted pairs
Output: a schedule for examination
Def checkConflict(arrangement):
    for each pair in conflict subjects list do
        if i1 in arrangement and i2 in arrangement then
            return False
    return True
Def checkStudent(arrangement):
    for each subject in arrangement do
        if the subject has no student left then
            return False
    return True
// generate all the arrangements using permutation into a list and
   reverse the list to take the most subjects arrangement
// create a list to contains the solution called res
for each arrangement do
    if checkConflict(arrangement) and checkStudent(arrangement) then
        res.append(arrangement)
print(res)
```

III.3 Greedy Algorithm

Overview

Greedy Algorithm is a traditional approach to many NP decision problem. In this approach, the input is taken in a specific order, the algorithm will make the choice for variables in the input, the choice once has been made is irrevocable even when this choice leads to sub-optimal answer. The order of the input is probably the most important factor in a Greedy Algorithm. Inspired by that intuition, our Greedy Algorithm sort the input subject in a specific order corresponding to their attendants number and degree in conflicted pairs to achieve better answer.

Properties

- Completeness: The Greedy algorithm is incomplete, meaning that it is not guaranteed to find the optimal solution.
- Running time: The algorithm runs very fast, making it well-suited for large inputs.

- Determinism: The Greedy algorithm is deterministic, meaning that it produces the same result each time it is run with the same input.
- Efficiency: As the result is demonstrated in later section, the answer given by this algorithm is relatively good in complicated case with many conflicted pairs and near optimal in small case or large case with few conflicted pairs.

Although the quality of the solution when we apply Greedy search is not guaranteed to be good, it can be improved by combining this algorithm with other opimization techniques which we will mention in next section.

Pseudo code

Algorithm 3: Greedy Algorithm

```
Input: standard input with subjects, rooms and conflicted pairs
Output: a schedule for examination
c ← array of room capacity
d ← array of number of attendants of n subjects
conflicted_pairs ← array to store all pairs of conflicted subjects
p ← number of subject
sol ← array to store solution
// Following code is to sort the subjects in desired order
conflicted_count ← array to store the number of the conflicted pairs a subject in
tmp1 ← MinMaxScale(d)
tmp2 ← MinMaxScale(conflicted_count)
score ← tmp1 + tmp2
sub ← array of subjected sorted by their score
remain_sub ← sub
count ← 0 ;       // This variable is used to count number of timeslots used
for k in p do
    count ← count + 1
    available_rooms ← allrooms
    sub_registered_this_time ← = []
    for subject in remain_sub do
        psb ← True
        if subject is in conflicted with any subject in sub_registered_this_time then
            psb ← False
    if not psb then
        skip this subject this timeslot
    for room in available_rooms do
        if X[subject][room][k] == 1 then
            assign this subject to this room, timeslot and to sol
            remove this subject from remain_sub
            remove this room from available_room
            append this subject to sub_registered_this_time
    if length of remain_sub == 0 then
        terminate
return sol, count
```

III.4 Constraints Programming with Binary Search

Overview

This algorithm utilizes Google Ortools to solve the problem for a specific value of timeslots needed. If ortools can solve it then we decrease the number of timeslots needed, else we increase the number of timeslots needed. Although this method can solve for optimal answer, there is no big difference anwser between the one in timelimit and the optimal, as a result, for better utilization we set a time limit, leading to the possible sub-optimality of the anwser

Properties

- Completeness: We set a time limit for CP part to run, this part of the algorithm is still considered completed by the nature of CP, it will either return with answer satisfying all constraints or no answer at all for each number of timeslot needed.
- Running time: This CP version run relatively fast, under 3 minutes on average for the most complicated case. (If we do not set time limit, then this CP version always return optimal answer, however, for the largest and most complicated test case, the execution time varies greatly from 3 minutes to 20 minutes)
- Determinism: This CP model is guaranteed to return the same optimal answer for same test case in every run.
- Efficiency: This Algorithm achieves the balance in trade off between run time and efficiency as discussed later

Pseudo Code

Algorithm 4: CP with Binary Search

Input: standard input with subjects, rooms and conflicted pairs
Output: a schedule for examination
$c \leftarrow$ array of room capacity
$d \leftarrow$ array of number of attendants of n subjects
$conflicted\_pairs \leftarrow$ array to store all pairs of conflicted subjects
$p \leftarrow$ number of subject ; `// This variable is the assumed number of timeslots needed in the optimal answer`
$p\_checked \leftarrow []$; `// This array is used to store all the value of p`
`// Following code generate the binary variable according to room capacity constraints`
for *i in range(n)* **do**
  for *j in range(m)* **do**
    if *d[i]* $\leq$ *c[j]* **then**
      model.Add(X[i][j][k] = [0,1])
    else
      model.Add(X[i][j][k] = 0)
`// Following code is used to generate conflicted_pairs contraints`
for *pair in conflicted_pairs* **do**
  $i1, i2 = pair$
  for *k in range(p)* **do**
    model.Add($\sum_{j=1}^{m} X[i1][j][k] + \sum_{j=1}^{m} X[i1][j][k] \leq 1$)
`// Following code guarantees all the rooms at any given timeslot can only hold at max one subject`
for *j in range(m)* **do**
  for *k in range(p)* **do**
    model.Add($\sum_{i=1}^{m} X[1][j][k] <= 1$)
`// Following code guarantees all subjects will be organised exactly one time`
for *i in range(n)* **do**
  model.Add($\sum_{j=1}^{m} \sum_{k=1}^{p} X[i][j][k] = 1$)
$solver \leftarrow cp\_model.CpSolver()$
$solver.parameters.max\_time\_in\_seconds \leftarrow 50$
$status \leftarrow solver.Solve(model)$
if *model.status == OPTIMAL or FEASIBLE* **then**
  $upper\_bound \leftarrow p$
  $new\_p \leftarrow (upper\_bound + lower\_bound)//2$
else
  $lower\_bound \leftarrow p$
  $new\_p \leftarrow (upper\_bound + lower\_bound)//2$
$sol \leftarrow X[i][j][k]$
if *p in p_checked* **then**
  append p to p_checked
  return sol, p_$checked[-1]appendptop_checked$
  $p \leftarrow new\_p$
  return sol, p_checked[-1]

III.5 Stochastic Hill-Climbing Algorithm

Overview

The idea behind Hill-Climbing is to start with an initial solution (an initial schedule for N subjects), and then repeatedly make small changes to the solution in an attempt to improve it. We will use a variant of Hill-Climbing algorithm which is Stochastic Hill-Climbing. The main point here is that the initial solution is a random solution among feasible solutions. The algorithm moves from the current solution to a random "neighboring" solution, and if the neighboring solution is better (i.e. has a lower objective function), it becomes the current solution. We define the neighboring solutions of a solution in this problem is a solution that is created by two methods which are changing the rooms of a particular period and combining two periods if the intersection of two sets of rooms is empty. This process continues until a local optimum (a solution that is better than all of its neighbors) is reached.

Properties

- Completeness: Hill-climbing is not guaranteed to find the optimal solution, as it only moves to locally optimal solutions and may get stuck in local minimal.
- Running time: The algorithm runs very fast compared to other algorithms.
- Determinism: Hill-climbing is deterministic, meaning that it produces the same result each time it is run with the same input.
- Efficiency: Hill-climbing can be an efficient algorithm for our topic, as it does not need to consider all possible solutions like the brute force algorithm. Instead, it only considers neighbors of the current solution, which can be a much smaller search space. Last but not least, Hill-climbing gives us a reasonable solution in an acceptable amount of time(see in Result Analyst).

Pseudo code

Algorithm 5: Hill-Climbing

```
Input: standard input with subjects, rooms and conflicted pairs
Output: a schedule for examination
Def findBetterNeighbour(currentSchedule):
    flag = random(1, 2)
    if flag==1 then
        newSchedule=changeRoom(currentsSchedule) ;        // change room of a
         particular period in current schedule
    newSchedule=combineTwoPeriod(currentSchedule) ;      // choose two random
     periods and combine that two periods into one
    return newSchedule
Def hillClimbing():
    initialSchedule = randomSchedule
    solution = initialSchedule
    while findBetterNeighbour(solution) ≠ ∅ do
        solution=findBetterNeighbour(solution)
    return solution
```

IV Result Analyst

Test and Result

We will use 6 test sets to evaluate whether the algorithm is efficient or not by judging two factors

- Completeness
- Running time

Here is the informations of 6 test sets that we use:

- Small input, few constraints: $N = 10, M = 2, K = 10$
- Small input, many constraints: $N = 10, M = 2, K = 40$
- Medium input, few constraints: $N = 50, M = 10, K = 50$
- Medium input, many constraints: $N = 50, M = 10, K = 500$
- Large input, few constraints: $N = 100, M = 20, K = 100$
- Large input, few constraints: $N = 100, M = 20, K = 1000$

Here are the result we get:

Test set	CP-SAT	CP-Binary	Brute force	Greedy	Hill Climbing
Small input, few constraints	0.068	0.067	0.074	0.006	0.03
Small input, many constraints	0.1313	0.143	9.8495	0.005	0.03
Medium input, few constraints	15.5177	3.27	N/A	0.013	0.29
Medium input, many constraints	39.2961	43.47	N/A	0.013	1.86
Large input, few constraints	35.1282	23.768	N/A	0.05	4.12
Large input, many constraints	N/A	127.8675	N/A	0.045	48.5

Table 1: Time comparison(second)

Test set	CP-Binary	Greedy	Hill Climbing
Small input, few constraints	6.32	6.42	7.33
Small input, many constraints	7.71	7.76	8.92
Medium input, few constraints	8.38	8.52	15.89
Medium input, many constraints	9.52(48% exact)	11.69	24.58
Large input, few constraints	8.61	8.83	22.12
Large input, many constraints	9.33(47% exact)	11.53	36.32

Table 2: Result comparison(number of periods)

- 'N/A' stands for 'No Answer' or our computer can't give the solution in a limited amount of time.

CP-SAT and CP-Binary)

- The running time of both algorithms is likely equal and quite low in small-size test set.
- They take more time at larger size test set.
- CP-SAT can't give the solution of large input, many constraints test set.

Bruteforce

- The running time of the algorithm is reasonable in small-size test set.
- Brute force require many space of memory so it can't return the solution in limited time

Greedy and Hill Climbing

It can be seen that Greedy and Hill Climbing mostly require a small amount of time, except for the required time of HC in large input, many constraints.

CP-Binary and Greedy

- This algorithm return a reasonable score in all test sets. It gives the optimal solution in 48 out of 100 test samples in medium input, many constraints and gives the optimal solution in 47 out of 100 samples in large input, many constraints.
- Greedy also return reasonable score in all test sets.

Hill-Climbing Algorithm

- Hill-Climbing not always give the optimal solution since it can converge to local optimal.

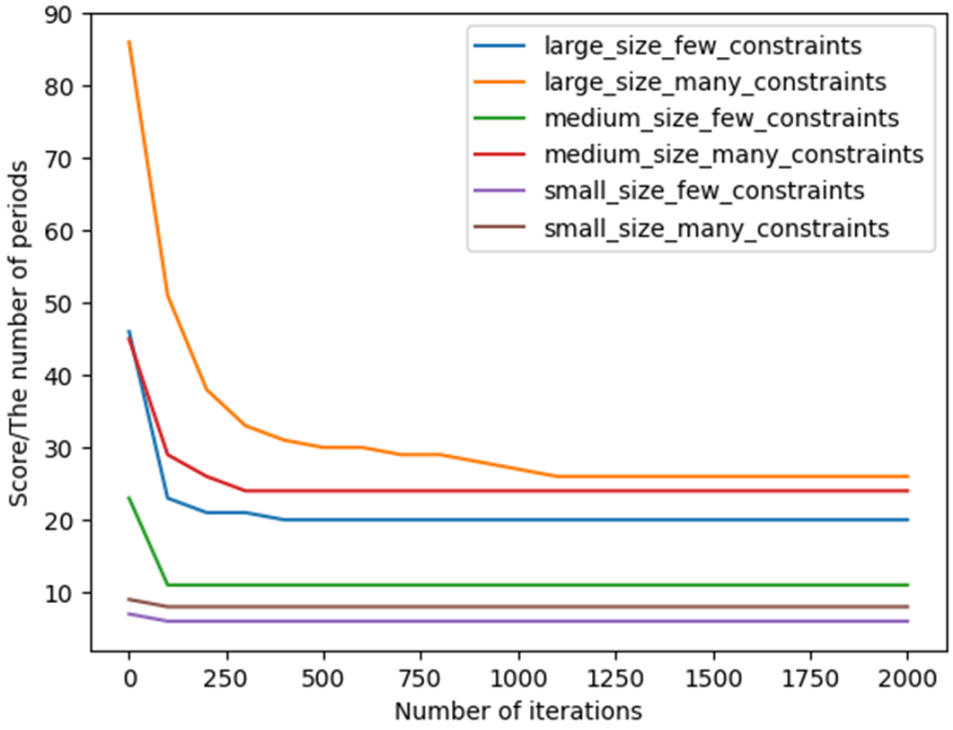

Fig. 1: Graph presenting the number of periods in term of the number of iterations when implementing the Hill Climbing algorithm

V Work Sharing

- Modelling: Dương
- CP-SAT: Dương
- Brute force: Dương
- CP-Binary: Quang
- Greedy: Quang
- Hill Climbing: Tùng
- Result and analysis: Tùng