Optimization methods and algorithms

Report Examination Timetabling Problem

Group 18 (AA-LZ)

Students:

Enrico Cecchetti s253823

Matteo D'Ospina s252973

Federico Gianno s214525

Salvatore Giordano s252811

Carlo Maria Negri s 214809

1. Introduction

The problem we have to deal with can be identified as “Examination Timetabling Problem” (ETP). As initial data we consider a set E of exams which have to be scheduled during an examination period and a set of student S enrolled at the university. Each student is enrolled in a subset of exam and the examination period is divided in t_{max} timeslots. If students are enrolled in more exams in the same timeslots these are considering conflicting. The aim is to schedule exams in given timeslots in order to maximize the efficacy of the exam distribution but at the same time avoiding the possibility that two conflicting exams will be placed in the same timeslot. We also need to reduce a given penalty function in order to create a more sustainable timetable for the students.

The programming language we choose is C to ensure that our program would have been highly performing and a better manage of memory.

2. Data structure

Three different files are given:

- The first file has the format INT1 INT2 where the first integer is the exam ID and the second one is the number of enrolled students for the specific exam. We used this file to obtain the number of total exams that we stored in a variable called *nexams.*
- The second file contains the number of timeslots. We save it in a variable called *tmax*.
- At least, the third file has the format sINT1 INT2 where the first integer is the student ID and the second one is the exam ID where he’s enrolled. We save the number of the students in a variable called *nstudents* and we generate a matrix called *table_schedule* that has student in rows and exams in columns. We assigned ‘1’ if a student is enrolled for a given exam, ‘0’ otherwise.

It is important to underline that exam, student, and timeslot IDs goes from 1 to *nexams*, *nstudents*, *tmax*. That allows us to use a correspondence between index and value in our data structures. **(ho provato a spiegare la corrispondenza indice valore nelle nostre strutture dati)**

Once we finished reading the files we generate a conflict matrix called *conflicts* that has exam ID in both rows and columns. Each position contains the number of conflicting students between exam ‘i’ and exam ‘j’. We can consider *conflicts* as an “adjency matrix” for the graph G(V,E) that has exams as vertex and conflicts as weighted edges.

We finally collect the solution in a vector of struct composed as follows:

```c
typedef struct solution{
    int *e;
    int dim;
    int currpos;
} Solution;
```

The length of this vector is the number of timeslots, “*e” is a vector of exams in a given timeslot, dim is the allocation size of “e” and “currpos” is the number of exams in vector “e”.

3. Algorithm

The algorithm is described as follows:

To find out the best possible solution to the problem we decided to apply a metaheuristic method. As a requirement for the meta-heuristic we started from finding a first feasible solution.

For this step we used “Graph Coloring” algorithm on our graph G. This algorithm assigns a “color” (a timeslot in our case) to every vertex such that no two adjent vertex can share the same color, so in our case, no two conflicting exams can share the same timeslot. Our aim is to “color” the graph with the minimum number of color (given by *tmax*) in order to reach feasibility. For this reason, we order vertices according to their degree using quicksort. This technique is able to improve performances of our algorithm, but the program still was not able in every case to find a feasible solution. So, we added an additional timeslot (*tmax +1*) and we place there all the uncolored vertex.

Summarizing, after Graph Coloring, we can stand in two different situations; in the first one we may have found a feasible solution and the timeslot *tmax+1* is empty, on the other hand in the second one *tmax+1* is not empty so we would schedule all uncolored exams that *tmax+1* contains.

To solve this last issue, we used another algorithm called “Tabu search”. We take exams placed in the extra timeslot one by one and we schedule them in a timeslot that contains exams with the lower number of conflicts with mine. Once our exam is scheduled we unschedule from the considering timeslot all conflicting exams placing them in *tmax+1*. We sign the move as done in a Tabu List and we keep doing it until *tmax+1* will be empty. Tabu list is important because avoid the generation of loop in Tabu Search algorithm. This algorithm allows us to find a starting feasible solution in a reasonable time.

Once we have an initial feasible solution we use another algorithm, called “Simulated Annealing”, which help us to decrease the value of our objective function working for the remaining time available. It starts from an initial value of *t* (as temperature) set as 20000 and decrease it after each iteration through the function *cooling_schedule.* During each iteration the algorithm uses these two different functions:

- The function *neighbourhood_ssn* takes a scheduled exam, searches a timeslot where this exam doesn’t have conflicts and put it there.
- The function *neigbourhood_swn* swap two different exams making sure the new positions will be without any conflicts for them.

If these two functions find solutions with a lower value of objective function the algorithm continue with the solution just found. And if the solution is not better then the previous one, through a probabilistic calculation, we decide if it is necessary to keep the solution (we found a local minimum) or to discard it.

If t reaches 0 we know that a local minimum is found so the temperature is reset and the algorithm can move to another feasible space.