Bianchi C., Chiapponi E., Coral R., Djettchou Nana L. W., Lorenzo F. Group 11 [A-L]

Examination Timetabling Problem – a multithreaded approach.

In order to solve the Examination Timetabling Problem, we decided to split it into three subproblems:

- Initialization: its goal is to provide an initial solution that satisfies the hard constraints of the problem as quickly as possible.
- Processing: we apply a constructive heuristic to improve the quality of the initial feasible solution before applying any metaheuristic.
- Optimization: in this step, we implement different algorithms that take advantage of the qualities of all the metaheuristics covered during the course (Simulated Annealing, Tabu Search, Genetic Algorithms).

Furthermore, we chose to follow a multi-start approach, hence allowing a parallel initialization of multiple initial feasible schedules. This is done both to improve our chances of finding the global optimum and to obtain an initial population of solutions, which are used as an input to the genetic algorithm implementation realized in the Optimization step.

Initialization. In this phase, our goal is to group exams in a number *Ns* of subsets, such that each subset does not contain couples of *conflicting exams*[1], and having $Ns <= t_{max}$. We consider this problem as a graph coloring problem where:

- A node of the graph represents an exam
- An arch between 2 nodes represents a conflict between 2 exams
- We can use at most t_{max} colors.

To do this we implemented an algorithm that follows this idea:

1. Considering that nodes with many adjacent nodes are probably more difficult to color, we will color them with a higher priority. To do so, we create a list of nodes sorted in descending order of the number of arches having them as a vertex. Moreover, we group colors in buckets (subsets of colors with a fixed size). We initialize *nBuckets=1.*
2. We consider *nBuckets* buckets.
3. We consider the node at the top of this list.
4. We try to color the current node with a random color among the ones in the considered buckets. If we manage to color the node without causing any conflicts, we remove it from the list and go to step 6. Otherwise, after a fixed number of failed attempts, we go to step 4a.
    a) We randomly change the colors of the nodes that are already colored (changes are allowed only if they lead to a feasible solution). We go back to step 4. This loop (4 -4a) can be repeated at most N times; after N iterations, we go to step 5.
5. Since we did not manage to color the current node, we need to extend the number of available colors, hence we increase *nBuckets* and go back to step 3. If we cannot increase the number of buckets (we're already considering t_{max} colors), we reset everything and restart from step 1.
6. If there are no more node to be colored, END, otherwise, go to step 3.

This algorithm works fine, but it has a deterministic component since the order in which exams are considered for a given instance is always the same. To add some randomness, after creating the ordered list, we consider 5 consecutive exams (1-5; 6-10; 11-15,…) and swap randomly their positions (e.g. e1-e2-e3-e4-e5-|-e6-e7-e8-e9-e10-| can result in e2-e5-e3-e1-e4-|-e7-e6-e10-e8-e9).

[1] Two exams are "conflicting" if they have at least one common student enrolled in both of them. We call conflict a couple of conflicting exams.

Results: the algorithm provides feasible solutions in <1s for all the public instances.

Processing. From the initialization phase, we obtain at most t_{max} subsets of non-conflicting exams. The goal of this phase is to place each subset in a timeslot to obtain the lowest possible objective function. To do so, we find the best possible order of the subsets by using a random insertion heuristic:

1. We build an empty list.
2. We randomly pick two subsets and we place them in the first two positions of the list. Then we compute the overall penalty we would pay if this list was an actual timetable.
3. We consider another subset and we try all possible placements (at the beginning or the end of the list or between two already inserted subsets). We choose the position that ensures us the lowest value of the objective function.
4. We go back to step 3 until all the subsets are positioned. The position of the subsets in the list determines the timeslot in which they will be placed in the schedule.

The algorithm worked fine, but in order to find an even better solution, once the insertion is completed, we try to swap the order in which subsets are placed in the list (allowing only improving swaps).

Results: the algorithm provides a solution in some seconds. The gap between the current objective function and the benchmark is usually halved, with respect to the random order determined by the initialization phase.

Optimization. The goal of this phase is to optimize the value of the objective function of the schedule. At the beginning of this phase, we have multiple solutions provided by the exploitation of the algorithms previously presented, executed on parallel threads.

This phase is divided in two parts:

1. Optimization 1:
   Multiple threads run the DeepDiveAnnealing algorithm, a mixed approach between simulated annealing and tabu search. We reserve about 40% of the computational time that we have at our disposal to this phase.
   This algorithm begins as a classic simulated annealing. Starting from a solution, we modify it executing certain moves. These moves are executed only if they lead to a better cost function, or their penalty is accepted by the simulated annealing logic. The kind of moves we consider in this phase are mostly moving one exam from a timeslot to another one. Occasionally, we swap the entire content of 2 timeslots. The temperature is decreased in an exponential way ($temperature_{i+1} = temperature_i * K$, where i is the current platform) and for each temperature a fixed number of iterations K is performed. When the temperature goes below a certain threshold, the tabu search logic is activated: we start keeping track of the moves that we have executed and add them in a tabu list, so that these moves cannot be undone for a defined number of iterations (unless they satisfy the aspiration criterion: the move leads to the best solution ever found). If the algorithm is stuck in a local optimum the temperature rises again, allowing an efficient exploration of the solution space (see Figure 1).

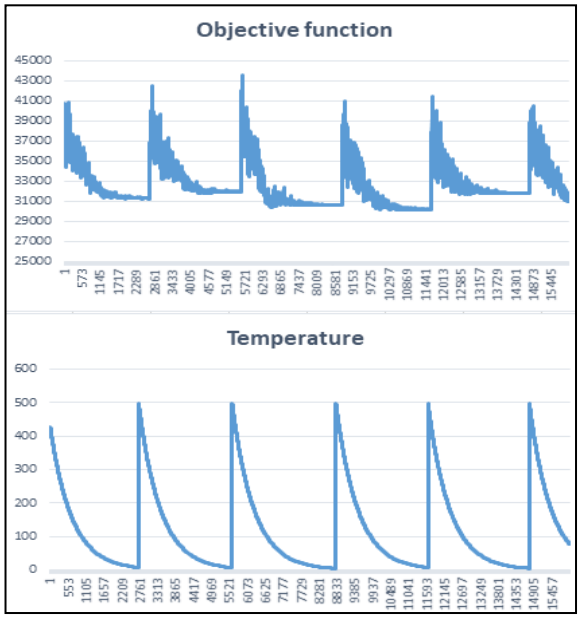

Figure 1: Progress of the objective function and temperature on instance07. As the objective function reaches a plateau, the temperature rises.

2. Optimization 2:
   We set a time limit *tlim*.
   Since we have a set of optimized solutions we can run the Genetic algorithm, which uses the solutions found by Optimization1 as its initial population. Since we don't want to waste any computational resource we run one more thread with the DeepDiveAnnealing. If we still have more than 40% of computational time at our disposal, these DeepDiveAnnealing instances will run for that long, otherwise they will run for tlim seconds (see Figure 2).
   Our implementation of the genetic algorithm tries both to improve the single solutions by means of the moves presented in Optimization1 but more importantly tries to merge them exploiting a crossover-like logic; no worsening moves are accepted.
   After *tlim* seconds passed, we join the population on which the genetic algorithm is working with the solution provided by the DeepDiveAnnealing, and order them by means of their objective function. We iterate through the ordered list from the worse solution to the best one. We pick the first solution and decide whether to remove it (with a probability of 80%). In case we don't, we move to the next solution in the list and repeat the process until one schedule is finally removed.
   In this way, we try to escape local minima.
   We restart this phase from the beginning, using the "depurated" population as initial population. The DeepDiveAnnealing works on the best of these solutions.

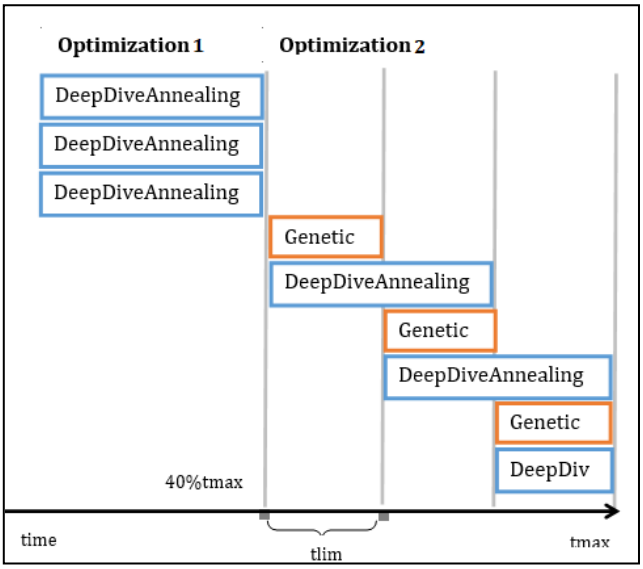

Figure 2: Representation of time management (simplified). Each box represents a different thread.

Results: since the optimization phase implements many different concepts, there are many parameters that need to be configured (e.g. the rate at which the temperature decreases, the length of the tabu list). This leads to algorithms with a high flexibility in which we can choose whether we want to obtain a decent solution in a small amount of time or we are willing to wait for more to get a higher quality solution. Figure 3 shows our results on the public instances.

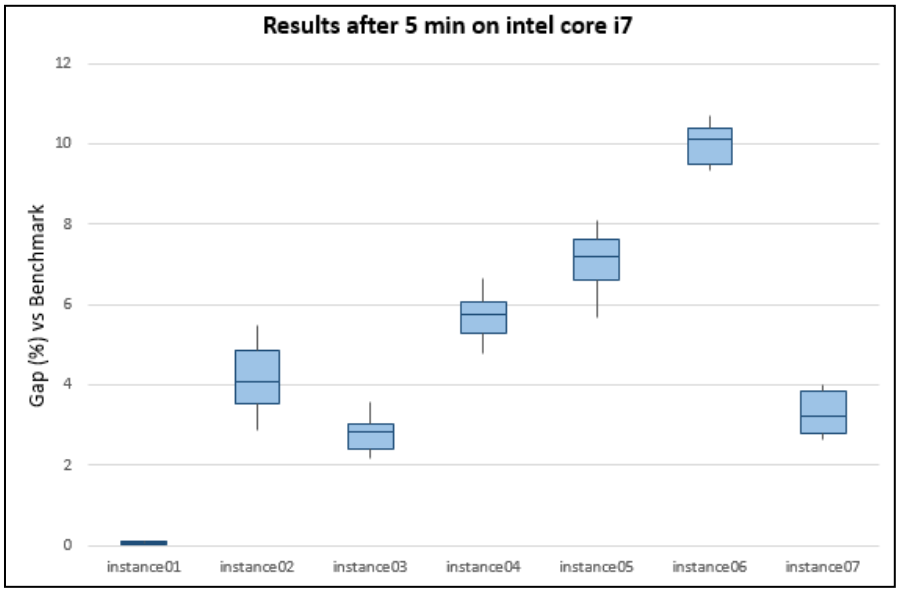

Figure 3: Boxplots that represents the statistics about the gap(%) from the benchmark on public instances. For these statistics the algorithm was run 8 times on a intel core i7 for 5 min