Technical University of Denmark
Department of Applied Mathematics

April 28, 2015

42137 Optimization using Metaheuristics

University Scheduling with Greedy Heuristics

Authors:
Adrien Haegel
Maria Albert Gimeno

April 28, 2015

Contents

Introduction

Scheduling is a well-known problem that arises from a willing of a better utilization of the resources. In this project we focus in a university schedule example where constraints such as lecturer availability, room capacity, curriculum compatibility, allocation of lectures are relevant for the timetable cost. The formulation of this problem is not of particular complexity however due to the large size of the data, the search for the optimal solution becomes a difficult task. For this reason, the application of heuristics which may not find the optimal solution but a good solution in a short period of time, is convenient. The effectiveness of these methods depends upon their ability to adapt to a particular problem and data, avoid entrapment at local optima, and exploit the basic structure of the problem.

The goal of this project is to find potential heuristics that are suitable for this specific problem. Thereafter implement them and study how good they perform in the problem. The first method that we use is the so well-known *GRASP*[4] method which is based on a greedy procedure. Then we develop a new version of this method in order to improve computational time. Then we apply *LNS*[3] as an extension of the first method. Finally, we combine the mentioned methods into one using *ALNS*[3].

In the first part of the report we introduce the scheduling problem that we want to optimize, as well as the constraints that it should satisfy and the objectives that should optimize. In order to control this, we develop a computational model that will be the basis for the heuristic method. In the second part, we explain the methods applied and how they have been implemented to this particular problem. In addition, we consider the different parameters of a given method and tune them in order to achieve best performance. We also analyze qualitatively the computational time of each method. Finally we compare the methods and establish their efficacy.

Part I
The Scheduling problem

1 Problem Description

The scheduling problem we are to solve is known as curriculum-based course timetabling. A curriculum is a set of courses that the same students follow. This problem consist of the weekly scheduling of the lectures of courses within a given number of rooms and time periods, where conflicts between courses are set according to the curricula published by the university. The problem considers the following entities:

- **Days, Periods and Timeslots**
  We are given a number of lecturing days in the week (typically 5 or 6). Each day is split in a fixed number of periods, which is equal for all days. A time slot is a pair composed of a day and a period. The total number of time slots is the product of the days times the periods.

- **Courses and Lectures**
  Each course consists of a fixed number of lectures to be scheduled in distinct periods, it is attended by a given number of students, and is taught by a lecturer. For each course there is a minimum number of days that the lectures of the course should be spread in, moreover there are some periods in which the course cannot be scheduled.

- **Rooms**
  Each room has a capacity, expressed in terms of number of available seats. All rooms are equally suitable for all courses (if large enough).

- **Curricula**
  A curriculum is a group of courses such that any pair of courses in the group have students in common. Based on curricula, we have the conflicts between courses and other soft constraints.

The solution of the problem is a number of assignments of lectures to a time slot (day and period) and a room which satisfies the following constraints:

Constraint 1 (Lectures) *Each course has a predetermined amount of lectures that must be given. Each lecture must be scheduled in distinct time slots and the total number of lectures cannot be exceeded.*

Constraint 2 (Room Occupancy) *Two lectures cannot take place in the same room in the same time slot.*

Constraint 3 (Conflicts) *Lectures of courses in the same curriculum or taught by the same lecturer must all be scheduled in different time slots.*

Constraint 4 (Availabilities) *Some courses cannot be scheduled at specific time slots. This can be due to the lecturer teaching the course or some students attending the course are unavailable at those time slots.*

The goal of the problem is to plan lectures in order to minimize the cost. This cost is given by the following unwanted attributes. Each unwanted attribute has an associated penalty, the sum of all these penalties constitute the objective value of the solution.

Objective 1 (Unscheduled lectures) *Each course has a predetermined amount of lectures that must be given. As many of these lectures as possible must be scheduled. Each course that has a lecture which is not scheduled gives a penalty of 10 points.*

Objective 2 (Room capacity) *For each lecture, the number of students that attend the course must be less or equal than the number of seats of all the rooms that host its lectures. Each student above the capacity counts as 1 point of penalty.*

Objective 3 (Minimum working days) *The lectures of each course must be spread into a given minimum number of days. Each day below the minimum counts as 5 points of penalty.*

Objective 4 (Curriculum compactness) *Lectures belonging to a curriculum should be adjacent to each other (i.e., in consecutive time slots). For a given curriculum we call a lecture from the curriculum secluded if it is not scheduled adjacent to any other lecture from the same curriculum within the same day. Each secluded lecture in a curriculum counts as 2 points of penalty.*

Objective 5 (Room stability) *All lectures of a course should be given in the same room. Each distinct room used for the lectures of a course, but the first, counts as 1 point of penalty.*

2 Computational Model

In order to satisfy the above mentioned constraints, the structure of the code has been created bearing in mind the relationships between the entities and the relevance of those. Here we explain constraint by constraint how we have modeled them in our algorithm.

The core of the different entities in the algorithm is the object *Course*. This object contains main fields such as *room stability*, *curriculum list*, *availability*, *working days*, ... Furthermore *Course* is a central object from where we seek for available time slots and rooms to construct a new solution. To make this solution feasible we insert the mentioned constraints in the fields intrinsically. This is specified constraint by constraint in the following paragraph.

For constraint 1 we have created a list of lectures to be scheduled, so it can happen that a lecture is not scheduled (due to infeasibility or not improving solution) but we will never find an excess number of scheduled lectures. Similarly for constraint 2 we have a list of available rooms for each time slot which is updated every time a lecture takes place or releases a time slot. To satisfy constraint 3 and 4, we use the field *Availability* which has size of the total number of time slots. It is initialized with the pre-unavailabilities of a course given by constraint 4 and is updated when a course of the same curriculum or lecturer takes or releases that time slot. This ensures that we will end up with a feasible solution.

Now we focus in optimizing the solution step by step by adding lectures to the schedule. Computing the change in the objective solution for each of these steps is of crucial importance since the faster we calculate the value of a solution the more solutions we can assess. A shortcut to compare different options is to create a delta function that calculates the gain of scheduling a new lecture instead of calculating the total objective value. In this way, we will only consider a lecture addition if the value of the corresponding delta function happens to be negative. The delta function is the sum of the gain of the different objectives for adding one lecture.

The gain for objective 1 is straightforward, -10 when the lecture is assigned. For objective 2 we need to count the excess of students with respect to the capacity of the room and add this amount if positive.

For the objective 3 we use the field *working days* which is an array of size the number of days where each element contains the number of lectures of this course in a given day. Then we create an array that contains the cost of adding a lecture in a certain day. This array is initialized with -5, updated to zero when a lecture takes place in a certain day and set to zero when the actual working days are bigger or equal to the minimum of working days.

In the same fashion as we evaluate the gain for the minimum of working days constraint, we create an array of the size of the time slots that gives the

cost of scheduling a course in that time slot given by objective 4. This array is initialized by 2 and updated depending on the adjacent planned courses of the same curriculum.

Finally, for objective 5 we use the field *room stability* which is a list of rooms that are used by the course. This allows us to update the room stability objective by just looking at this list and adding one if the room assigned is not in *room stability*.

Part II
Metaheuristic methods

3 Grasp

3.1 Algorithm theory

The *GRASP* algorithm consists of constructing a greedy solution of the problem. At each step of the algorithm, a new lecture is added to the schedule. This change is performed in a greedy way: the algorithm chooses randomly amongst the changes with the lowest cost, that is, the changes that improve the whole solution the most. We keep adding courses to the schedule until we obtain a solution.

This procedure is repeated until a certain time is reached, and we select the best solution amongst all those who have been computed.

We thought GRASP would be an efficient way to solve this kind of problems. First of all, it is close of the human way of solving this problem: Iteratively constructing the schedule by adding the lectures where they fit the most. Secondly, in this particular problem, if a scheduling of one lecture gives a very poor result, it is not likely to lead to a good solution. For example, scheduling a lecture in a room that is too small while better suited rooms are available. A greedy heuristic aims to build a sensible solution in very short time.

3.2 Implementation

The key method to success in having an efficient GRASP algorithm is to have an efficient "delta-evaluation", that means, to be able to compute the cost of scheduling a new lecture in the (current schedule) problem. Indeed, to construct the list of the N best choices, we need to go trough all the possible solutions and compute their associated cost, this procedure is explained in algorithm 2. Before adding something to the solution, this delta-evaluation is called a number of times equal to $\text{NB-OF-LECTURES}^2 \cdot \text{NB-OF-DAYS} \cdot \text{NB-OF-PERIODS-PER-DAY} \cdot \text{NB-OF-ROOMS}$, as it can be checked in the *GRASP* algorithm (algorithm 1). Even if time is lost during other parts of the algorithm, the slightest shortening of the time for one delta evaluation will make the whole algorithm faster.

For that reason, we aimed to have a delta-evaluation taking a constant time to compute. That is, there is not time cost due to arithmetical operations but just direct memory accesses for evaluating the delta function. These arithmetical calculations are only carried out when a lecture is added or removed from the schedule.

Algorithm 1 *GRASP* algorithm

```
min = +infinity
while time < timelimit do
    Current Solution ← new empty solution
    while lectures can still be scheduled do
        Construct N best choices for scheduling an unscheduled lecture in an empty Timeslot and Room
        Schedule randomly one of these best choices
        Update current solution value with delta evaluations functions
    end while
    if current solution value < min then
        min ← current solution value
        Best solution ← current solution
    end if
end while
```

Algorithm 2 Solution construction for *GRASP*

```
for each unscheduled lecture do
    for each available time slot do
        Calculate time slot delta adding cost Δt
        for each available room do
            Calculate room delta adding cost Δr
            if Δt + Δr < 0 then
                if Δt + Δr < max_N best solutions value(N) then
                    Remove solution whose value is the maximum of the best solutions
                    Store current solution in best solutions
                end if
            end if
        end for
    end for
end for
```

3.3 Tuning parameters

The only parameter to tune in this algorithm is the size N which is the number of the best changes amongst which the next change can be chosen. In our implementation, we only implemented this selection with an equal probability amongst these N elements.

When $N = 1$, we have a pure greedy algorithm, but this only leads to the same final solution, since there is no randomness. With higher values of N, we allow the algorithm to browse across a neighbourhood of this solution. The bigger N, the larger the neighbourhood is. However, with high N values, changes with poor values can be added to the schedule, leading to bad final solutions. There is therefore a compromise between the size of the neighbourhood and the efficiency of the greedy mechanism.

We compare the performance of the different parameters with the gap between the best solution value among the different parameters, z_t^*, and the average solution value among the number of trials for test t, z_t.

$$\text{Gap} = \frac{(z_t - z_t^*) \cdot 100}{z_t^*} \tag{1}$$

We calculate this measure for each parameter N and each test t, and show the performance of each parameter N in figure 1. This parameter tuning has been done during 5 minutes for 5 different trials for each parameter N.

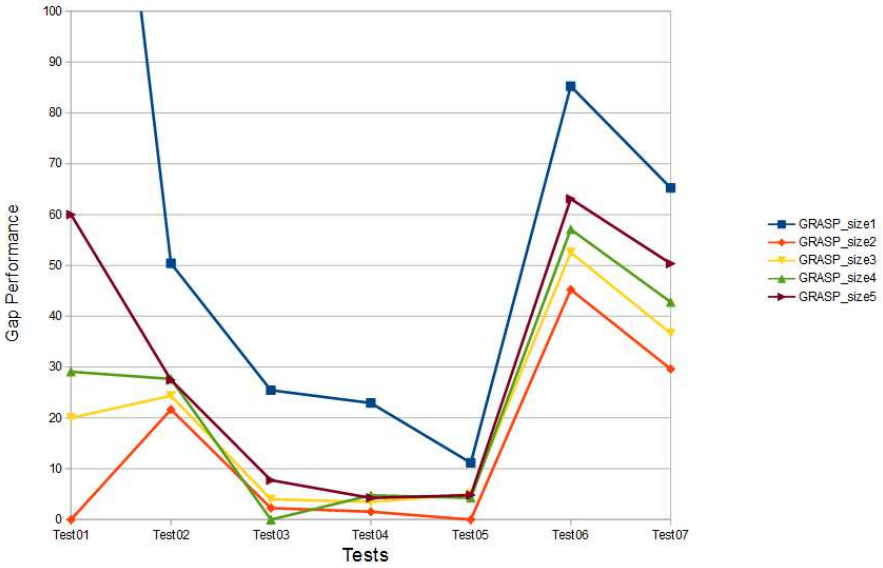

Figure 1: Gap performance for different window sizes

$N = 2$ seems clearly to be the best parameter in this problem for the test data we consider. As expected, $N = 1$, leading to a unique solution, gives poor results. However, $N = 2$ is still a very low value, and it is surprising to obtain the best solutions with this value. This can be seen as a confirmation that greedy methods are well suited for this problem: $N = 2$ is the greediest parameter that

still adds randomness. We corroborate this conclusions with the average gap over the different tests which summarizes figure 1, the results can be found in table 1.

Window size N	Average gap performance	Deviation
$N = 1$	68.11	0
$N = 2$	14.33	10.87
$N = 3$	20.85	12.96
$N = 4$	23.66	10.94
$N = 5$	31.10	11.99

Table 1: Average gap and deviation for *GRASP* tuning with window sizes

Hereafter, we will refer to GRASP as this algorithm with a window size $N = 2$.

4 A modified version of GRASP: Lecture-GRASP

4.1 Algorithm theory

The main issue with the GRASP algorithm of the previous section is that looping through all the possible changes can be very expensive computationally. Indeed, we need to compute the cost for every lecture, every time slot, and every room. We develop a new method with the same logic of *GRASP* but trying to restrict the computational cost, bearing in mind that the lectures are, in general the largest set.

The way we approach this method is by choosing ahead the lecture that is likely to be added. By doing so, we only need to use the *GRASP* method on the rooms and time slots for that given lecture, which would greatly reduce the time needed for this operation. However, it could happen that a lecture is assigned to a time slot and a room that could be more profitable for another lecture.

In order to implement such a method, we need to find a good way to select the order of the lectures that will be added to the schedule. It is however possible to have a very problem-dependent method which can therefore make the algorithm efficient. For example, if we start adding the lectures that have the highest number of constraints, these constraints can be fulfilled more easily when the schedule is still empty. Furthermore, a course with only one lecture and no curriculum constraints can be added in the end, since it does not interfere with the rest of the solution. By cleverly choosing the order of the lectures we add to the schedule, we can obtain both greedy construction quality and lower computational costs.

Algorithm 3 *Lecture-GRASP* algorithm

```
1: min = +infinity
2: while time < timelimit do
3:     Current Solution ← new empty solution
4:     Sort lectures
5:     for each lecture to be scheduled do
6:         Construct N best choices for scheduling this lecture in an empty Timeslot and Room
7:         Current Solution ← Schedule randomly one of these best choices
8:         Update current solution value with delta evaluations functions
9:     end for
10:    if current solution value < min then
11:        min ← current solution value
12:        Best solution ← current solution
13:    end if
14: end while
```

The way the N best choices are constructed for scheduling a given lecture in an empty Timeslot and Room is exactly the same procedure as algorithm 2 but without the outer loop over the courses.

4.2 Implementation

The different sorting methods we tried are the following:

- Initial Sorting : We use the initial sorting of the data sets (ordered by ID).
- Random Sorting : Lectures are ordered at random
- Sorting by students : Lectures are ordered by decreasing number of students. Such a sorting seems promising since it fills the crowded courses first, leaving the small courses for the end (that can be put in any remaining room)
- Sorting by students inverted: reverse order of Sorting by students
- Sorting by lectures : Courses are ordered by the number of lectures : A course with a high number of lectures is easier to add to the schedule in the beginning
- Sorting by lectures inverted : Reverse order of Sorting by lectures
- Sorting by curriculum : the lectures that belong to several curricula are put first, a course that does not belong to any curriculum can be added very easily at the end.

We tested each of these method, the results are presented in the next section. Two methods showed to be more efficient than the others: Sorting by students and random sorting. For that reason, we came with a combination of these two methods:

- Sorting by students + random : The lectures are initially sorted by students, then a random number of permutations in this list is done. This allows to benefit from the two best methods simultaneously.

4.3 Tuning parameters

In order to select the best sorting strategy, we do a similar analysis as we did in the previous section. We measure the gap using formula 1 for all our sorting methods on the first 7 data sets, during 5 minutes with an average on 5 trials. The detailed results are presented in Figure 2.

Sorting method	Average gap	Deviation
Initial Sorting	79.30	7.91
Random Sorting	80.71	11.55
Sorting by students	48.57	6.86
Sorting by students inverted	238.27	7.22
Sorting by lectures	98.96	9.35
Sorting by lectures inverted	101.36	9.20
Sorting by Curriculum	107.28	13.16
Sorting by students + random	34.11	8.03

Table 2: Average gap and deviation for *Lecture-GRASP* tuning with different sorting methods

As it has been said before, either the random sorting or the sorting by students give the best results, and the combination of both of these methods: Sorting by students + random, is clearly a good choice for ordering the lectures as we observe in Figure 2. This is confirmed by the average gap values which are given in table 2.

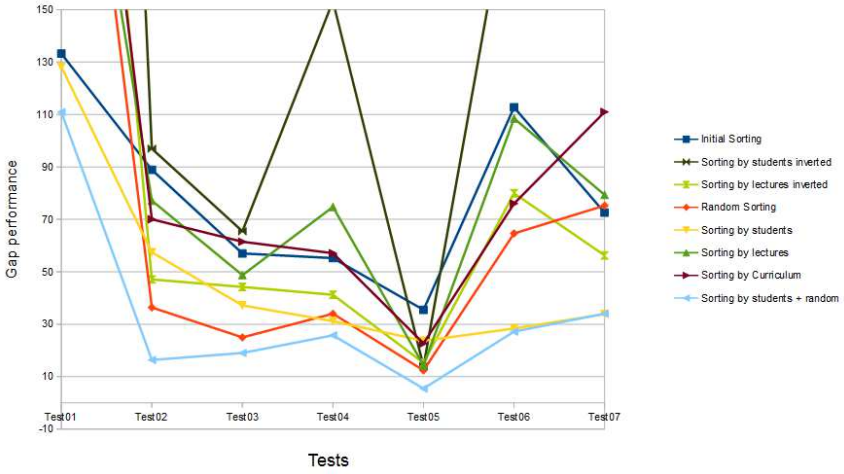

Figure 2: Gap performance for different sorting methods

From now on, we are considering Sorting by students + random as our sorting method. We now need to tune the window size of the *Lecture GRASP* algorithm. The performance given by the gap measure is shown in Figure 3 and the average of the gap in table 3.

Window size	Average gap	Deviation
1	3.42	9.00
2	11.70	12.68
3	18.88	8.78
4	23.65	10.21
5	34.11	7.29
6	64.95	6.86

Table 3: Average gap and deviation for tuning *Lecture-GRASP* window size

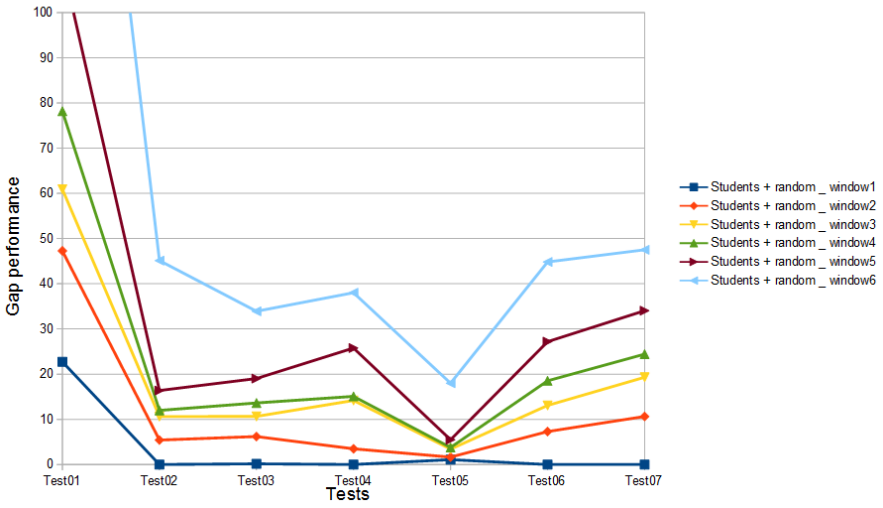

Figure 3: Gap performance for different window sizes

We can conclude from these results that for this algorithm, a pure greedy selection method with a window size of 1 is the best one. Although we have seen in the classic GRASP from the last section that a pure greedy method leads to the same final solution, it is not the case here, since there is some randomness in the choice of the lectures. This randomness is enough to provide a neighbourhood to explore, which allows a pure greedy method to be efficient.

4.4 Comparison between GRASP and Lecture-GRASP

The main point of this Lecture-GRASP algorithm is to reduce the computational time for one iteration. In order to verify this, we plot the time for one iteration depending on the number of lectures of the test files. The corresponding plot

in log-log axis is displayed in Figure 4. We can observe that one iteration of Lecture-GRASP has one order of complexity lower than the classic GRASP algorithm. For the data sets we are considering, this means that we can perform at least 100 more iterations. This allows to go through much more solutions, and therefore, to hope for a better final result if both algorithms are run for the same amount of time.

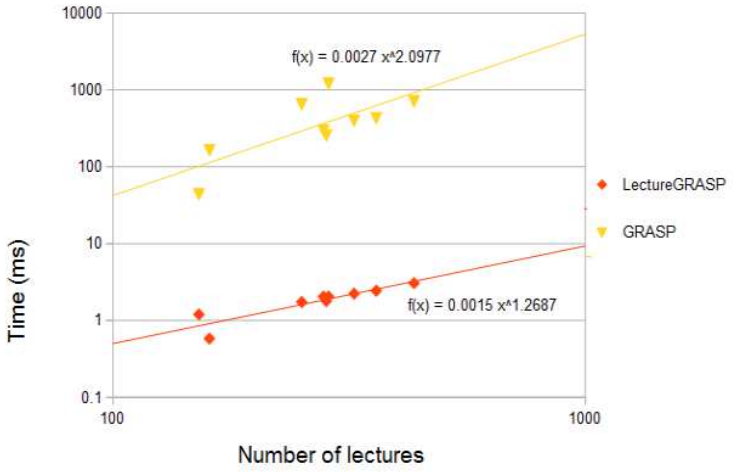

Figure 4: Computational time comparison between GRASP and Lecture-GRASP

5 LNS

5.1 Algorithm theory

The biggest inconvenience of *GRASP* is that it can be difficult to escape from a local optimum, since GRASP is always applied on a same starting point. In order to overcome this problem, *LNS* which stands for "Large Neighbourhood Search", was created. This method consists of destroying part of the solution and reconstructing it with a greedy algorithm. This allows to search through larger neighborhoods and avoids finding the same solution over and over again.

Another issue of GRASP is that the solution is deleted at each iteration, loosing all the progress that has been made. LNS allows to keep working on the existing solution, which prevents this waste of work.

Algorithm 4 *LNS* algorithm

```
1: min = +infinity
2: Current Solution ← feasible solution
3: while time < timelimit do
4:     Destroy current solution partially
5:     Repair destroyed solution
6:     Update current solution value with delta evaluations functions
7:     if current solution value < min then
8:         min ← current solution value
9:         Best solution ← current solution
10:    end if
11: end while
```

5.2 Implementation

The destroy and repair functions are crucial since they determine the neighbourhood in ALNS. For this reason we consider several destroy functions. We can think the destroy function as a chance to reschedule the timetable. It sounds natural, for example, to remove a room with high capacity that is being used by a course with a low number of students. In general, it is reasonable to remove any partial solution that is taking the place of a potentially better solution. In this sense, all destroying functions that releases a lecture, a room, a time slot or a curriculum are worth trying. Furthermore, a random choice of these destruction possibilities at each iteration. The different destruction methods are given below:

- *DESTROY COURSE* Removes all lectures from the schedule for a given Course chosen randomly and un-books the rooms those lectures were using at the time slots they took place.

- *DESTROY ROOM* Releases a given room and un-schedules all lectures that were using that room.
- *DESTROY TIME SLOT* Removes all lectures that were scheduled at the given time slot and sets free the rooms that they were using at that time.
- *DESTROY CURRICULUM* Un-schedules all lectures of all courses that belong to a given curriculum. It also releases the rooms those lectures were using at the time they were scheduled.
- *DESTROY RANDOM* It chooses randomly among the destroying functions explained above.

Not only what we destroy is important, but also how big the part of the solution that we destroy is, that is how many lectures, rooms, time slots or curricula respectively, we decide to release. The destroying methods that we consider have a destroying size which corresponds to the number of times the method is iterated.

For the repair function, we decide to test the tuned *GRASP* and *Lecture-GRASP* since they have given good results in the previous sections.

- *REPAIR GRASP* with window length of 1.

  Notice that this method was not given good results in the previous section due to lack of randomness, however now randomness is introduced with the DESTROY function, then we reconsider this method again.
- *REPAIR GRASP* with window length of 2
- *REPAIR LECTURE-GRASP* with window length of 1

We combine all these parameters and tune them in the following section.

5.3 Tuning Parameters

Since this is hard to estimate beforehand, we start with destroying size of 1 and we keep studying higher destroying sizes until we observe the results get worse.

In a similar fashion as how we tuned in the previous methods, we calculate the average gap for the different parameters. Due to the size of all the possible combinations, a graphical presentation would be very difficult to analyze. However the average gap values are presented in different tables for the different repair functions (Tables 4, 5 and 6).

Destroy size	Destroy Function: Course	Curriculum	Room	Time slot	Random
1	(107.1, 32.0)	(117.6, 7.9)	(54.6, 7.0)	(84.0, 26.1)	(62.4, 7.8)
2	(107.2, 21.2)	(116.9, 7.0)	(54.8, 7.4)	(89.9, 27.8)	(60.9, 6.7)
3	(46.4, 6.4)	(140.8, 8.6)	(81.9, 6.7)	(24.8, 6.3)	
4	(51.7, 5.9)			(35.9, 7.5)	
5	(57.2, 6.5)			(45.6, 9.0)	

Table 4: (Average gap, deviation) for *LNS* tuning with repair function *GRASP* window size 1

Destroy size	Destroy Function: Course	Curriculum	Room	Time slot	Random
1	(103.8, 10.3)	(145.8, 8.1)	(78.7, 6.6)	(33.3, 13.5)	(86.8, 5.8)
2	(101.6, 12.2)	(146.1, 6.9)	(78.8, 7.2)	(30.9, 17.4)	(85.1, 7.1)
3	(92.3, 10.3)	(161.5, 9.5)	(96.8, 6.9)	(53.9, 10.2)	
4	(90.7, 5.9)			(59.6, 10.9)	
5	(89.5, 7.1)			(67.6, 10.1)	

Table 5: (Average gap, deviation) for *LNS* tuning with repair function *GRASP* window size 2

Destroy size	Destroy Function: Course	Curriculum	Room	Time slot	Random
1	(112.6, 25.8)	(166.3, 9.8)	(88.5, 7.1)	(28.4, 13.2)	(94.4, 5.6)
2	(107.5, 32.7)	(160.0, 9.9)	(90.7, 6.1)	(37.2, 37.9)	(97.1, 5.9)
3	(63.7, 7.9)	(168.6, 7.7)	(109.4, 7.4)	(63.9, 10.3)	

Table 6: (Average gap, deviation) for *LNS* tuning with repair function *Lecture-GRASP* window size 1

We observe that destroying part of the solution by using *DESTROY TIME SLOT* gives in general good results. Nevertheless, other destroying functions give good results for certain repairing functions and window lengths, for example, destroying course with destruction size of 3, works well with repairing function *GRASP* with window size of 1; and with *Lecture-GRASP* with window size of 1. Destroying room with destruction size of 3, works well with repairing function

GRASP with window sizes of 1 and 2. The random method does not give very good results, which means that there must be another way of combining the methods in order to obtain the best from all of the above destroy functions. This is the motive for introducing a clever way of getting the best from all the methods: ALNS, which is presented in the next section.

5.4 Time comparison between GRASP and LNS

As it has been said before, GRASP and Lecture-GRASP throw away their solution to start again from the beginning at each iteration. This is not the case for LNS, where the same solution is kept and improved. The destroy and repair method only change a small part of the solution, and is therefore less computationally expensive than the GRASP methods. Furthermore, LNS allows to try new neighbourhoods at each iteration. For these reasons, we expect LNS to perform better than GRASP and Lecture-GRASP.

In Figure 5, we can see that, as expected, one iteration of LNS with Destroy Timeslot of destroying size 3 and GRASP repair takes 10 times less time than an iteration of Lecture-GRASP. We can expect to cover much more possible solutions with LNS, and therefore, expect a better final solution.

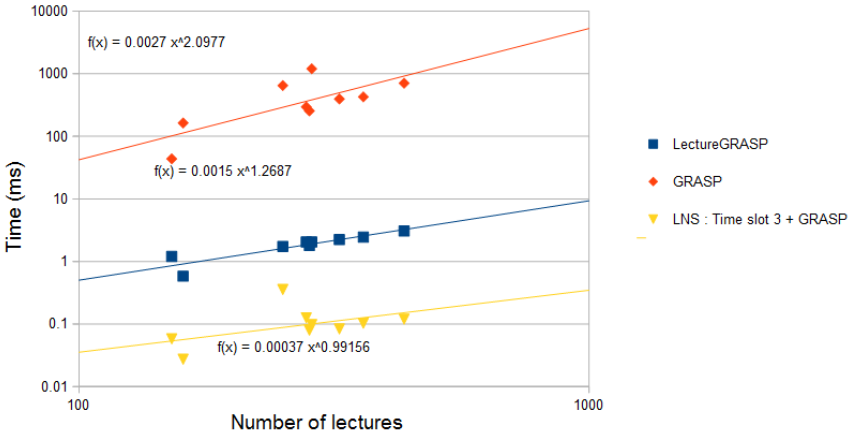

Figure 5: Time per iteration for GRASP, Lecture-GRASP and LNS

6 ALNS

6.1 Algorithm theory

In the last section, we noticed that different destruction and reconstruction methods could lead to a good solution. However, for our LNS algorithm, we can only choose one. This is harmful, since these methods could complement each other. For example, destroying a time slot allows to reorganize the rooms, but destroying a room allows to reschedule courses without having to worry about the room stability or room capacity. Furthermore, some of these methods are more efficient for certain data sets than for other ones. This is why we decided to test an adaptative LNS (ALNS).

The idea of ALNS is to let the algorithm use several construction and destruction methods, in the hope of getting benefit from the combination of the methods we used in the last section. However, we need to make sure that the efficient methods will be used more often, hence a reward system is added to the algorithm. This reward system will increase the probability of selecting the methods that have given a good result increasing the quality of the solution. By doing that, we hope the algorithm to adapt to the specifics of every data set.

6.2 Implementation

The ALNS algorithm is very similar to the classic LNS, the difference lies in the choice of the destruction and reconstruction method. At the beginning, each method can be selected with an equal probability.

Once methods are chosen, the classic LNS algorithm runs. Depending on the result of the new solution, the corresponding probability is updated with a reward. The four different rewards are the following by decreasing value:

- ω_1 New best solution : the solution found is the new global maximum
- ω_2 Better solution than current : the new solution is better than the previous one, but not the global best
- ω_3 Solution accepted : the new solution is worse, but below a given threshold of the previous solution.
- ω_4 Solution rejected : the new solution is above the threshold.

The formula to update the probabilities is:

$$\Phi \leftarrow (1 - \lambda)\Phi + \lambda\omega \tag{2}$$

The pseudo code for the ALNS is given below.

Algorithm 5 *ALNS* algorithm

```
min = +infinity
Current Solution ← feasible solution
while time < timelimit do
    Choose a destroy method from a probability array DestroyProba
    Destroy current solution partially
    Choose a repair method from a probability array RepairProba
    Repair destroyed solution
    if new solution value < Selection-Threshold · old solution value then
        Update current solution value with delta evaluations functions
        if current solution value < min then
            min ← current solution value
            Best solution ← current solution
            Update DestroyProba and RepairProba with reward ω1
        else if current solution value < old solution value then
            Update DestroyProba and RepairProba with reward ω2
        else (current solution value > old solution value)
            Update DestroyProba and RepairProba with reward ω3
        end if
    else
        Do NOT update the solution
        Update DestroyProba and RepairProba with reward ω4
    end if
end while
```

6.3 Tuning parameters

Tuning ALNS is a difficult task given the high number of different parameters given below:

- Selection of the different destruction and repair methods that can be selected.
- Reward ω_1, ω_2, ω_2, ω_4
- Damping factor λ
- Threshold to accept the new solution

Since it is impossible to tune our algorithm on all these parameters, we used a *hill-climbing* approach by tuning these parameters separately. We also do the

following assumption: $\omega_1 = 100S^2$, $\omega_2 = 100S$, $\omega_3 = 100$, $\omega_4 = 0$ with the variable $S \in \mathbb{R}^+$. This allows to reduce the number of variables from 4 to 1.

6.3.1 Selection of the destruction and repair methods

From the previous section, we know that some methods work better than others, and some destruction methods work well with a given repair method. Therefore, there are different ways of choosing the methods we want our algorithm to use.

- ALNS-Full : All the methods are possible, that is, Destroy: Time Slot, Course, or Room, for destroying size from 1 to 3; Repair *GRASP* and *LectureGRASP* with a window size from 1 to 3.
- ALNS-Select: Only the methods that gave the best results in the last section are kept, that is, Destroy Course 3, Destroy Room 2, Destroy Time Slot from 1 to 3; Repair *GRASP* with a window size from 1 to 3 and *LectureGRASP*.
- ALNS-combined: In that case, the destroy and repair method are not chosen independently from each other, but chosen together. The list of possible choices is made of the best methods from the previous section.

While ALNS-Full may allow inefficient methods to be used in the algorithm, ALNS-combined on the contrary may show not enough diversity. The results of the tuning for tests 1 to 5 with a running time of 5 minutes and averaged over 5 different trials is displayed in table 7 and in Figure 6.

	ALNS-Full	ALNS-select	ALNS-combined
Average Gap	45.34	21.46	24.47
Deviation	11.78	12.55	10.77

Table 7: Average gap and deviation for tuning *ALNS*

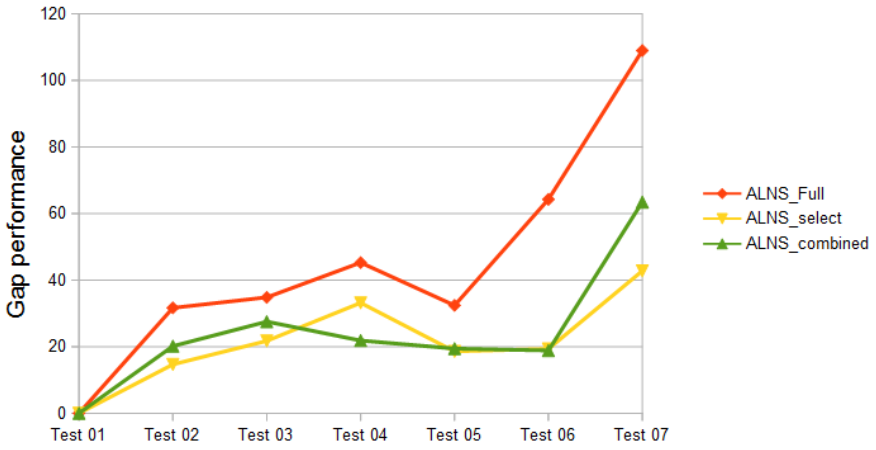

Figure 6: Gap performance for different ALNS methods

It appears that ALNS-Full is not as efficient as the two other ALNS methods, that only include efficient destroy and repair methods. ALNS-Combined yields good results, but is slightly worse than ALNS-select. From now on, we will therefore consider ALNS-select.

6.3.2 Tuning S and λ

We now tune the reward parameter S ($\omega_1 = 100S^2, \omega_2 = 100S, \omega_3 = 100, \omega_4 = 0$) and the damping factor λ for tests 3 to 5 with a running time of 5 minutes averaged over 5 trials in the table 8 below. A dichotomic method has been used, which explains the lack of certain values, which have been replaced by an x. The best value is obtained for $S = 500$ and $\lambda = 2.5 \cdot 10^{-3}$, which is also a "local minimum". We will now consider this values for our algorithm.

	$\lambda = 10^{-2}$	$\lambda = 5 \cdot 10^{-3}$	$\lambda = 2.5 \cdot 10^{-3}$	$\lambda = 10^{-3}$	$\lambda = 5 \cdot 10^{-4}$	$\lambda = 10^{-4}$
S = 10	x	34.3	x	34.0	37.9	36.1
S = 50	x	30.6	x	30.7	32.2	32.1
S=100	x	28.1	x	30.4	28.7	32.0
S=250	26.2	28.9	30.1	x	x	x
S=500	28.1	27.1	25.4	27.0	30.4	34.7
S=750	26.5	29.5	30.3	x	x	x
S=1000	x	30.3	x	39.4	30.3	40.3
S=10000	x	x	x	79.2	x	79.2

Table 8: Average gap and deviation for *ALNS* tuning with different s and λ values

6.3.3 Tuning the selection threshold

When the solution is destroyed and repaired, the objective function can sometimes be degraded. If the difference is low, it should still be added to the solution, since it may help to leave a local minimum. However, if the difference is too large, it may damage the solution too much. This is why a selection threshold is added. This threshold is a percentage of the current solution. The results of the tuning for tests 2 to 5 with a running time of 5 minutes and averaged over 5 different trials is displayed in table 9.

Selection Threshold	Average gap	Average deviation
1	112.8	18.5
1.01	43.1	7.7
1.015	39.0	9.5
1.02	28.6	6.6
1.025	27.8	15.7
1.03	32.9	17.5
1.04	41.3	14.5
1.07	67.6	19.2
1.1	91.6	25.8
1.2	125.4	23.6
1.3	118.3	21.3
1.4	115.8	8.1

Table 9: Average gap and deviation for *ALNS* tuning with different threshold values

We then take our selection threshold equal to 1.025. The final parameters for ALNS are given below

Parameter	value
λ	$2.5 \cdot 10^{-3}$
S	500
Selection threshold	1.025

Table 10: Final *ALNS* parameters after tuning

7 Results

In this section, we use all the algorithms described in this report to solve 6 new data sets which have not been used to train these algorithms. The algorithms have been running for a period of 5 minutes and the results have been averaged on 5 different trials. The algorithms with the tuned parameters are stated below:

- *GRASP*
- *Lecture-GRASP*
- *LNS*:
  Destroy method: Course-3.
  Repair method: *GRASP* with window size 1
- *LNS*:
  Destroy method: Room-2.
  Repair method: *GRASP* with window size 1
- *LNS*:
  Destroy method: TimeSlot-3.
  Repair method: *GRASP* with window size 1
- *ALNS*

The mean solution values are given in figures 7, 8 and 9. The reason for having these values in different plots is the difference in magnitude of the solution value among the different tests.

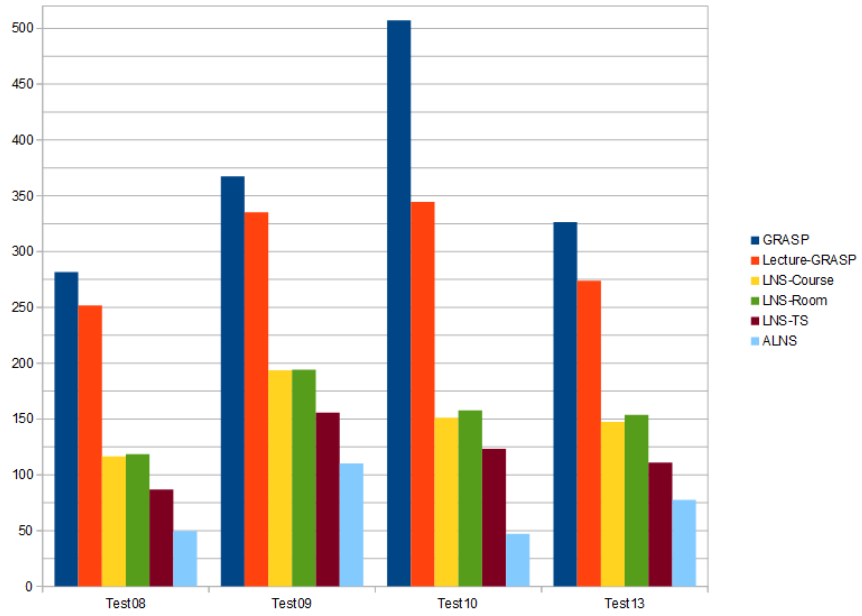

Figure 7: Mean solution value for different algorithms for tests 8, 9, 10, 13

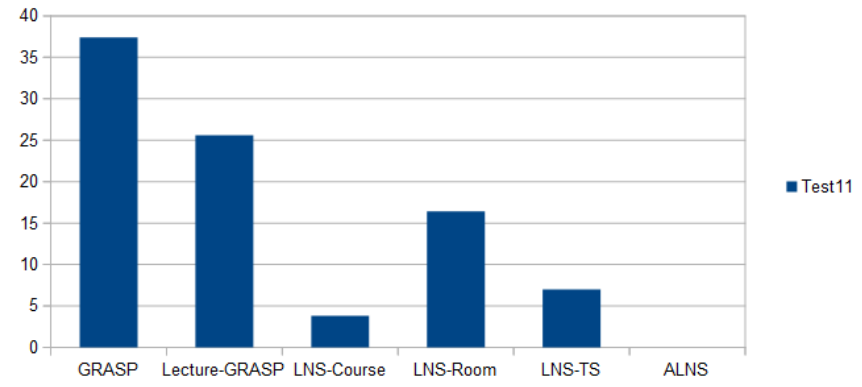

Figure 8: Mean solution value for different algorithms for Test 11

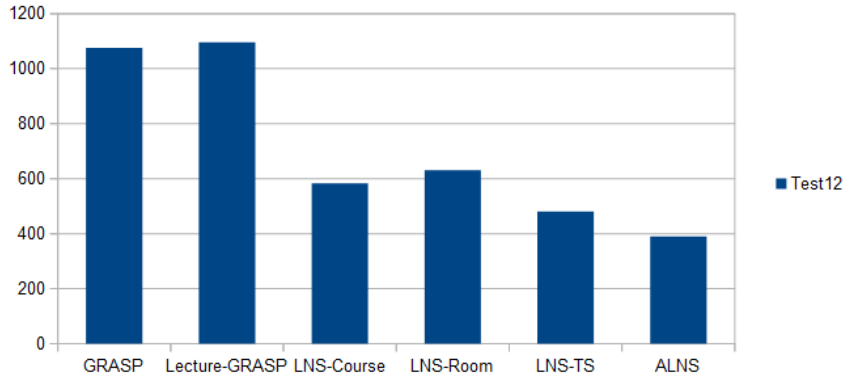

Figure 9: Mean solution value for different algorithms for Test 12

The gap performance is displayed in Figure 10 and the values are given in table 11. Test11 has not been taken into account, since the minimum value reaches 0.

	Average gap	Deviation
GRASP	482.6	16.8
Lecture-GRASP	372.9	8.2
LNS-Course	132.0	6.4
LNS-Room	140.5	5.7
LNS-TS	83.1	4.4
ALNS	7.7	7.4

Table 11: Average gap and deviation for the different algorithms

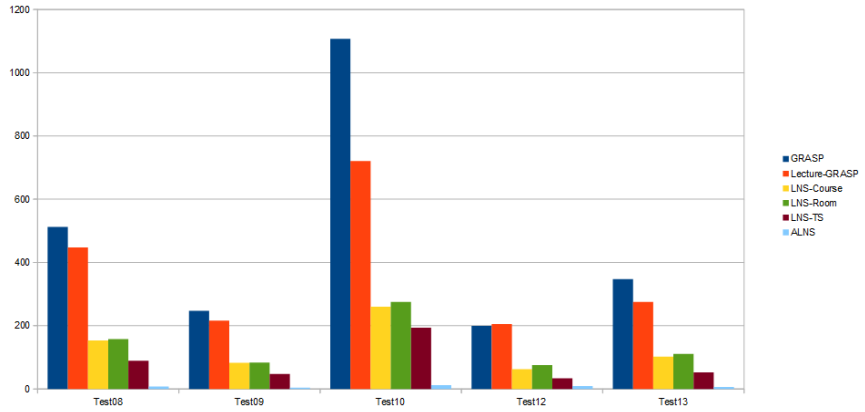

Figure 10: Average gap for the different algorithms

The results are as expected: there is a clear evolution in the solution value of the problem. That is, *GRASP* returns the maximum value with respect to the other algorithms for all the data sets tested. Secondly, we find the results of *Lecture-GRASP* whose difference with the first algorithm is in general an enhancement which depends to some extent on the data. The solution that *LNS* provides is considerably better than the previous ones. The variations of this last algorithm for different destroying functions can be significant but it cannot be generalized since it is data dependent. Finally we find a last abrupt improvement with *ALNS* which shows that this algorithm does combine all the previous methods in a complementary way.

8 Conclusions

Our initial GRASP algorithm yields quite bad results, since each iteration is very expensive and the search is occurring in a very small neighbourhood. For that reason, we tried to reduce the computational time by choosing the courses in the greedy search in Lecture-GRASP, which showed to be slightly better than GRASP. However, these two methods start all over again at each iteration, which is a huge loss of time. LNS was the logical next step : the search is done "horizontally" instead of "vertically". LNS yielded much better results. However, a lot of different destroy and repair methods were efficient, but LNS can only use one. For that reason, we finally implemented ALNS, which combines the best LNS methods. ALNS is clearly the best heuristic we tried. Although it is difficult to really assess the quality of a heuristic since we do not know the value of the optimal solution, we think that greedy methods and especially ALNS are a reliable heuristic in the problem we consider.

References

[1] Edmund K. Burke & Graham Kendall: *Search Methodologies* New York: Springer, 2005.

[2] Lectures and notes provided on campusnet.dtu.dk for course *42137 Optimization using metaheuristics* taught by Thomas Stidsen at DTU 2015.

[3] David Pisinger and Stefan Ropke. Large neighborhood search. link

[4] Feo, T.A., & Resende M. (1994) Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization 6: 109-133, 1995.*