UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Algorithmic Methods for Mathematical Models

Department of Computer Science

Project AMMM: Selecting the best committee

Authors:
Alex Herrero
Lluna Clavera

Professors:
Enric Rodríguez-Carbonell
Luís Velasco

December 9, 2024

1 Formal definition of the problem

The problem we need to solve consists in creating a committee of professors satisfying certain conditions. For this problem we will have a set of professors, each of them assigned to a specific department, as well as information about the compatibility between said professors. The goal of this problem will be to produce the "best" possible committee, that is, the committee that maximizes the average compatibility between its members with certain restrictions about compatibilities between them. We assume that compatibility is symmetric and compatibility with oneself is maximum.

Variable	Meaning	Range	Type
N	Number of members of faculty	Integer	Integer
D	Number of departments in the faculty	Integer	Integer
d_i	Department of professor i	$1 \leq i \leq N$	Integer
n_p	Number of people needed from department p	$1 \leq p \leq D$	Integer
m_{ij}	Compatibility between professor i and j	$0 \leq m_{ij} \leq 1,\ 1 \leq i, j \leq N$	Real

Table 1: Input from the problem

More formally given the input from **Table 1** we define each professor with an integer value i with $1 \leq i \leq N$. We consider a solution S as a subset $S \subseteq \{1, \ldots, N\}$ such that S holds the following conditions:

1. $|\{i \in S \,|d_i = p\}| = n_p$ for every $1 \leq p \leq D$ (We require a fixed number of professors of each department)
2. $\forall i, j \in S$, $m_{ij} \neq 0$ (We can not have two professors with 0 compatibility in the solution)
3. $\exists i, j \in S$, $m_{ij} < 0.15 \implies \exists k \in S$, $m_{ik} > 0.85$ and $m_{jk} > 0.85$ (When two professors in the solution have very low compatibility, there must exist a third professor in the solution with high compatibility with both)

Given a function $f(S) = \frac{2}{|S| \cdot (|S|-1)} \sum_{i \in S} \sum_{\substack{j \in S \\ i<j}} m_{ij}$, which calculates the average compatibility of the committee, the objective of this problem is to find a solution S^* such that $f(S^*) \geq f(S)$ for all possible solutions S.

2 Integer Linear Programming formulation

For our ILP formulation we will define the following auxiliary set of indices:

$$A_d = \{i \in \{1, \ldots, N\} : d_i = d\}$$
$$W_{ij} = \{k \in \{1, \ldots, N\} : m_{ik} > 0.85 \wedge m_{jk} > 0.85\}$$
$$N_{ij} = \{(i, j) \in \mathbb{N} \times \mathbb{N} : 1 \leq i < j \leq N \wedge m_{ij} < 0.15\}$$

Were A_d defines all professors that belongs to a certain department d, W_{ij} defines all professors that have high trust with professors i and j and N_{ij} defines all professors with low compatibility ($m_{ij} < 0.15$). **Table 2** provides information about the decision variables that we decided to use for this problem.

Variable	Meaning	Range	Type
x_i	Professor i goes to the committee	$1 \leq i \leq N$	Boolean
y_{ij}	Professor i and j go to the committee	$1 \leq i, j \leq N$	Boolean

Table 2: Decision variables for the problem

And with that we will provide the *Integer Linear Programming* formulation where we consider $n = \sum_{p=1}^{D} n_p$:

Objective Function:

$$\text{Maximize:} \quad \frac{2}{n \cdot (n-1)} \sum_{i=1}^{N} \sum_{j=i+1}^{N} m_{ij} \cdot y_{ij}$$

Subject to:

$$\text{Number of participants per department:} \quad \sum_{i \in A_p} x_i = n_p \text{ for every } 1 \leq p \leq D$$

$$\text{No two professors with 0 compatibility:} \quad \lceil m_{ij} \rceil \geq x_i + x_j - 1 \text{ for every } 1 \leq i < j \leq N$$

$$\text{Fit another professor in case that not enough trust:} \quad \sum_{k \in W_{ij}} x_k \geq x_i + x_j - 1 \text{ for every } (i,j) \in N_{ij}$$

$$\text{Correlation between } y_{ij} \text{ and } x_i, x_j: \quad \begin{cases} x_i \geq y_{ij} \\ x_j \geq y_{ij} \end{cases} \text{ for every } 1 \leq i < j \leq N$$

Regarding the objective function, all possible pair of professors can be calculated as $\frac{n \cdot (n-1)}{2}$. In our case we want to get the average of all possible pairs, that is why we put in front of the summation $\frac{1}{\frac{n \cdot (n-1)}{2}}$. We also added the decision variable y_{ij} because we want to count this compatibility only if both professors go to the committee. If this was non-linear programming we would use the product of two decision variables x_i, x_j, but it is not.

Lastly we need to be sure that $y_{ij} = 1$ if and only if $x_i = 1$ and $x_j = 1$. Constraints $x_i \geq y_{ij}$ and $x_j \geq y_{ij}$ will ensure that, if $y_{ij} = 1$ then you must have $x_i = x_j = 1$. In contrast, there is no explicit constraint forcing y_{ij} to be 1 when both x_i and x_j are 1. This situation is not really a problem due to the fact that we are in a maximization problem. The more addends you have in the objective function the higher it will be, so setting y_{ij} will always be preferred.

3 Solution using meta-heuristics

We suspect that this problem is, indeed, an NP-Complete problem so heuristics should be applied in order to get an approximation of the optimal solution. For that, we will use three different approaches: A Greedy approach, a greedy approach that will be improved using local search and, finally, using the GRASP method.

3.1 Greedy algorithm

A Greedy algorithm consist in, given a *greedy function* and a set of candidates, evaluating each candidate and, after that, choosing the one that scores the best according to said *greedy function*.

In our case we will define our greedy function as follows: Let S be a partial solution of the greedy algorithm (i.e. S is a set of selected candidates on an arbitrary iteration) and let U be all possible feasible candidates given a partial solution S. We define the greedy function $q(\cdot)$ of an element $u \in U$ as follows:

$$q(u) = \sum_{s \in S} m_{us}$$

With this idea in mind, we want to maximize, as much as possible, the objective function of our final solution, so we are going to pick the value that maximizes our *greedy function* at each iteration.

It is important to consider that the very first choice of the algorithm is somewhat special, as every partial solution S will be the empty set at the beginning and every candidate will be evaluated to the same value. For that reason, we decided that the first element of our solution will be the one that has the highest average compatibility with all other professors. After that, we will make use of the greedy function previously defined.

An implementation of it can be found in **Algorithm 1** as follows:

Algorithm 1 Greedy Algorithm

```
function GreedySolve(N, D, d, n, m)
    remaining ← sum(n)                                  ▷ Get total Committee size
    S ← ∅
    S ← S ∪ {bestCompat(N, m)}          ▷ Feasible Professor with highest average compatibility
    remaining ← remaining − 1
    while remaining > 0 do
        U ← feasibleProfs(N, D, d, n, m, S)
        if U = ∅ then
            return null                                 ▷ No feasible solution was found
        else
            p ← arg max_{u∈U} {q(u)}   ▷ Find the best feasible professor according to the greedy function
            S ← S ∪ {p}
            n_{d_p} ← n_{d_p} − 1                       ▷ Decrease the needed professors of p's department
            remaining ← remaining − 1
        end if
    end while
    return S
end function
```

3.2 Local Search

A Local Search procedure is a heuristic method to improve non-optimal solutions (if they were optimal there would be nothing to improve!) of a determinate problem. To make these improvements, a local search procedure tries changing some values of the solution to obtain several new solutions. To change one solution to another, we use a function called *operator* that transforms elements of the solution space into other elements of the same solution space.

The Local Search procedure will receive one single solutions, and then it will get multiple new solutions applying the same operator (or, sometimes, multiple operators) with different parameters. All solutions that can be reached from an initial solution S is called the *Neighborhood of* S or $\mathcal{N}(S)$.

The goal of a local search procedure is to explore several solutions of $\mathcal{N}(S)$ and then pick one that improves the initial solution (we can choose between picking the first solution that shows an improvement or exploring all solutions and then picking the best one). After picking a new solution S' we will repeat the process of searching in $\mathcal{N}(S')$ until we reach a local optimal solution (that is, a solution S^* such as $f(S^*) \geq f(s)$ for all $s \in \mathcal{N}(S^*)$).

In our case, we have used a *Swap Operator* that swaps elements (professors from the same department in our case) outside our solution with elements inside of it. We also used a *first improvement* strategy, that is, we jump to the first new solution that shows an improvement. We also considered the *best improvement* strategy, but we started testing the *first improvement* and we found that the results were pretty good (and fast) already (see **Subsection 4.3**), so we decided to stick with this strategy.

Algorithm 2 shows the exact implementation of our local search method and **Algorithm 3** describes the implementation of our operator.

Algorithm 2 Local Search

```
function LocalSearch(N, D, d, n, m, S)
    (i, j) ← findFeasibleSwap(N, D, d, n, m, S)
    while (i, j) is not null do
        S ← (S \ {i}) ∪ {j}
        (i, j) ← findFeasibleSwap(N, D, d, n, m, S)
    end while
    return S
end function
```

Algorithm 3 Local Search Operator

```
function findFeasibleSwap(N,D,d,n,m,S)
    for i = 1,...,N do
        for j = i+1,...,N do
            if validSwap(N, D, d, n, m, S, i, j) then        ▷ Feasible swap between elements
                outside ← from i,j the one that is not in S
                inside ← from i,j the one that is in S
                out ← 0
                in ← 0
                for k = 1,...,N do                           ▷ See if swap would improve our solution
                    if k ∈ S and k ≠ inside then
                        out ← out + m_{outside,k}
                        in ← in + m_{inside,k}
                    end if
                end for
                if out > in then
                    return (inside, outside)
                end if
            end if
        end for
    end for
    return null
end function
```

3.3 GRASP

A GRASP algorithm can be seen as a *randomized greedy* algorithm followed by a local search in order to improve the previous result. When we talk about a *randomized greedy* we are referring to the following: Given a greedy function $q(\cdot)$ and a candidate set U, a greedy algorithm will choose the element from U that scores the best (that is, the one with maximum or minimum value depending on the optimization). In a *randomized greedy* (or a *constructive greedy*) we do the following: Given a parameter $0 \leq \alpha \leq 1$ we construct a *Restricted Candidate List* (RCL) such that this list contains all the elements from U that are not more than $\alpha\%$ far away from the best candidate. From this list we then select, randomly, an element.

Because we added randomness, every execution of a constructive greedy may lead to a different result, that will be improved by using *local search*. For this reason, we will execute the whole process (*constructive greedy* + *local search*) various times in order to obtain the best solution that we could find. **Algorithm 4** defines our GRASP procedure for this particular problem and **Algorithm 5** defines the constructive Greedy. It is worthwhile mentioning that we have not implemented a special selection for the first element, unlike in our greedy algorithm, so the first element will always be selected at random. With high enough iterations, this is not a problem, as the algorithm will end up trying all possible combinations for the first element (including elements of the optimal solution!).

Algorithm 4 GRASP Procedure

function GRASP(N, D, d, n, m, α, $maxIter$)
  $S \leftarrow \emptyset$
  for $i = 1, \ldots, maxIter$ **do**
    $U \leftarrow constructiveGreedy(N, D, d, n, m, \alpha)$
    if $U \neq \emptyset$ **then**
      $U \leftarrow localSearch(N, D, d, n, m, U)$
    end if
    if $f(U) > f(S)$ **then** ▷ Check average compatibility
      $S \leftarrow U$
    end if
  end for
  return S
end function

Algorithm 5 Constructive Greedy

function CONSTRUCTIVEGREEDY(N, D, d, n, m, α)
  remaining $\leftarrow sum(n)$
  $S \leftarrow \emptyset$
  while remaining > 0 **do**
    $U \leftarrow feasibleProfs(N, D, d, n, m, S)$
    if $U = \emptyset$ **then**
      return $\emptyset$
    end if
    $q_{max} \leftarrow \max_{u \in U}\{q(u)\}$
    $q_{min} \leftarrow \min_{u \in U}\{q(u)\}$
    rcl $\leftarrow \{u \in U \mid q(u) \geq q_{max} - \alpha \cdot (q_{max} - q_{min})\}$
    $p \leftarrow$ get random element from rcl
    $S \leftarrow S \cup \{p\}$
    $n_{d_p} \leftarrow n_{d_p} - 1$
    $remaining \leftarrow remaining - 1$
  end while
  return S
end function

4 Experimentation and results

For the following subsections we used the samples given with the project plus 92 extra samples that were generated randomly from $N = 9$ professors up to $N = 100$. For GRASP we decided to execute 30000 iterations for every sample and run each sample three times just to be more likely to find a solution.

4.1 Selecting α parameter on GRASP

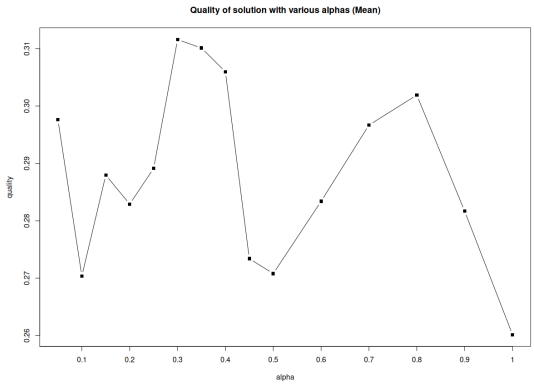
After developing a GRASP implementation, we need to tune the α parameter. For that, we are going to execute the following experiment: We are going to run all 100 samples that we have for each possible value of α that we are going to test ($\alpha = 0.05$ to $\alpha = 0.5$ in 0.05 increments and $\alpha = 0.5$ to $\alpha = 1.0$ in 0.1 increments) three times. Then, we are going to compute the mean objective function of all 300 executions (3 times, 100 samples) for each α and we are going to test what value of α provides us with the best mean value. It may seem weird to test the solution with different increments, but we decided to do the test this way because lower values of α seemed to provide better results. For that reason, we wanted to use a higher resolution on the values that provided better results.

After executing the experiment, results were quite interesting. Regarding **Figure 1**, we observe that $\alpha = 0.3$, $\alpha = 0.35$ and $\alpha = 0.4$ yield the best results in terms of mean (**Figure 1a**) and mean of maximum [1](**Figure 1b**) values, respectively.

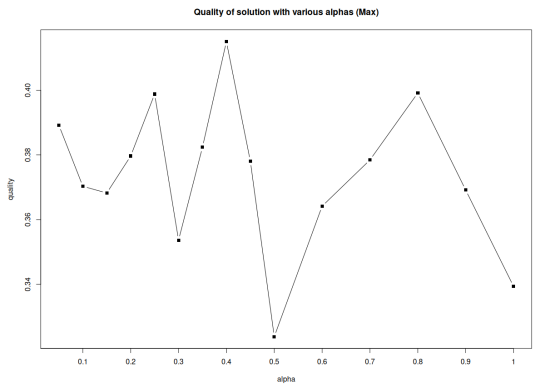
From **Figure 2**, we see that when considering only the solved samples (**Figure 2a**), $\alpha = 0.3$ achieves a significant improvement compared to $\alpha = 0.4$. However, it is noteworthy that **Figure 2b** shows $\alpha = 0.4$ solves more samples than $\alpha = 0.3$.

This indicates that while $\alpha = 0.4$ finds more solutions, solutions obtained with $\alpha = 0.3$ are of higher quality overall.

Both values have their pros and cons. However, we decided to use $\alpha = 0.3$, as we believe it is more meaningful to compare solutions when they exist rather than focusing solely on finding all possible solutions. There is also the possibility to use $\alpha = 0.35$ which is in the middle of both values mentioned. However, we decided that we wanted to focus mainly on the quality of the solution.



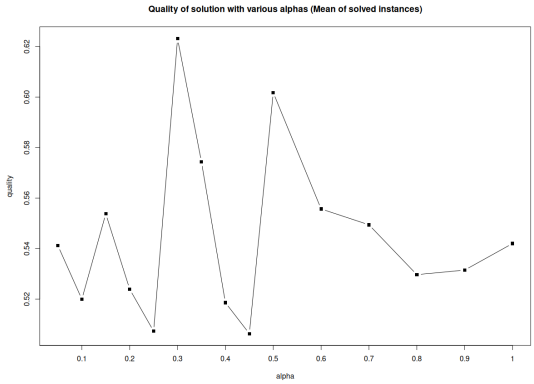
(a) Mean of objective functions for each alpha



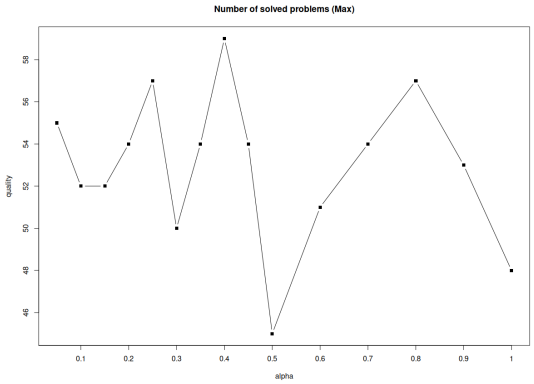
(b) Mean of max of objective functions for each alpha

Figure 1: Quality of solution according to max and mean of samples

[1]Mean of maximum is computed taking the best value of the objective function over the three executions of each sample, then taking the mean over those 100 values



(a) Mean of objective functions for each alpha using only solved instances



(b) Number of instances solved by different values of alpha

Figure 2: Number of solved instances and mean of it

4.2 Successes and misses

Nevertheless, it is interesting to look how often our heuristics find a solution whenever it exists. Looking at **Table 3** we see that both Greedy and Local Search fail 50% of the time (failing means that it does not find a solution when it exists). Unfortunately, unless we consider changing our *greedy function*, this is the best that we are able to achieve.

When it comes to GRASP, randomness starts playing an important role, as we see that every execution gets different failures. Although they are close to 50 fails, we normally tend to get better results than with the previous deterministic methods.

To interpret **Table 3**, GRASP (Max) is computed taking the best result of each sample among the three iterations, while GRASP (Mean) is computed as the mean of hits/misses among all three iterations.

Algorithm	Success	No solution	Fail
CPLEX	86	14	0
Greedy	36	64	50
Local Search	36	64	50
GRASP First Execution	42	58	44
GRASP Second Execution	43	57	43
GRASP Third Execution	35	65	51
GRASP (Max)	52	48	34
GRASP (Mean)	40	60	46

Table 3: Success and Failures from different methods

4.3 Quality of heuristics

After executing each heuristic, it is interesting to see how well the solution was with respect to the optimal one.

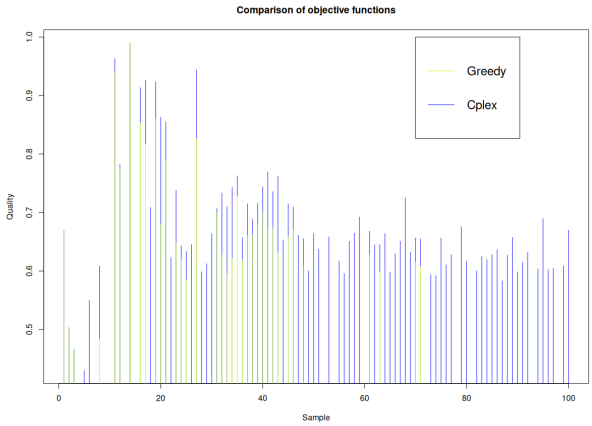
Starting with the greedy approach, Figure 3a illustrates the increasing difficulty of obtaining a solution as the problem size grows. Although this is the initial approach for solving the problem, the results are quite interesting, as we observe that, in general, the solutions found by the greedy algorithm are *close* to the optimal solutions (see also **Table 4** which shows how many samples achieve certain qualities).

Building on this, we introduce a Local Search phase following the greedy solution to further improve

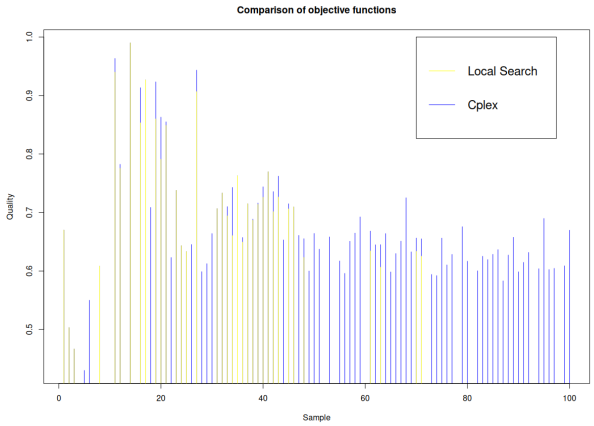
the results. As shown in Figure 3b, the solutions obtained with Local Search are indeed closer to the global optimum (and in some cases achieve it!) than those obtained using the greedy approach alone.

Finally, with the GRASP approach, we observe improvements in both the number of samples solved and the quality of the solutions. Interestingly, as shown in Figure 3c[2], GRASP consistently comes very close to the optimal solution.

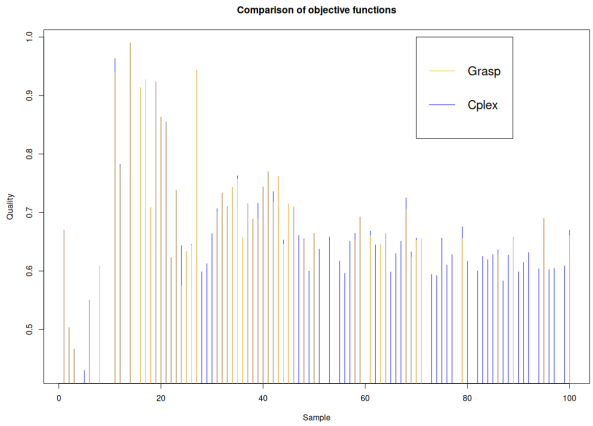
Recall that **Table 4** also provides information about how many samples got within a certain margin from the optimal solution. It is important to note that we discarded all instances where no solution was found to construct the table. As the table shows, the only heuristic that obtains results of less than 85% of the optimum is the greedy approach, and the GRASP[3] approach also ends (usually) really close to the optimum (95%)



(a) Greedy Solution



(b) Local Search Solution



(c) GRASP Solution

Figure 3: Solutions obtained by Greedy, LS and GRASP

[2]In this case we chose the maximum of the iterations instead of the mean. We found that when the GRASP solver founds a solution it usually produces similar results in between operations, but when one execution finds a solution and the others not, the mean will be way lower than usual and so it will not really reflect how close to the solution it can get. The least realistic part about the max is, then, the number of solved samples, but we can trust the quality of the solutions.

[3]We also used max for this test for the same reason as before, the mean would result in extremely low values in some occasions

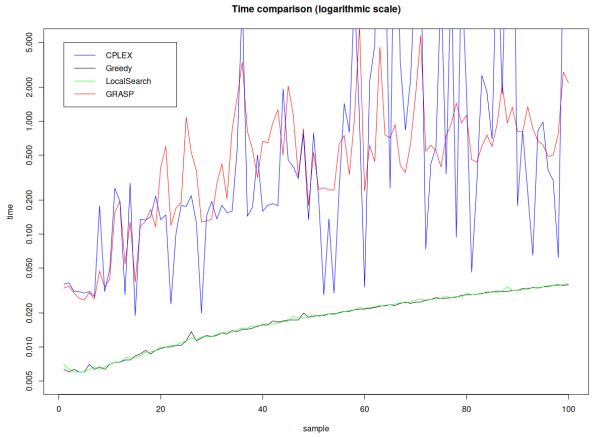
% Close to optimal solution	**Greedy**	**Local Search**	**GRASP**
100%	3	10	20
$> 95\%$	7	20	31
$> 90\%$	16	5	0
$> 85\%$	5	1	1
$< 85\%$	5	0	0

Table 4: Number of samples that reached certain qualities

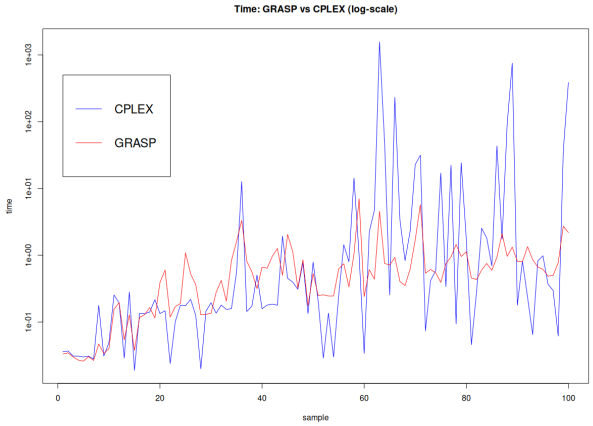
4.4 Execution Time

Regarding the execution time, Figure 4a shows that CPLEX and GRASP were indeed the methods that took the longest to solve. Specifically, when comparing CPLEX and GRASP (Figure 4b), we observe that the execution time for GRASP is more stable than that of CPLEX. Recall that Figure 4 displays all times on a logarithmic scale. Still, the bigger the size of the problem the more common it is for the GRASP to be faster than CPLEX, so with bigger problem sizes, it will be, in general, more advantageous to use the GRASP instead of the CPLEX (in terms of execution time).

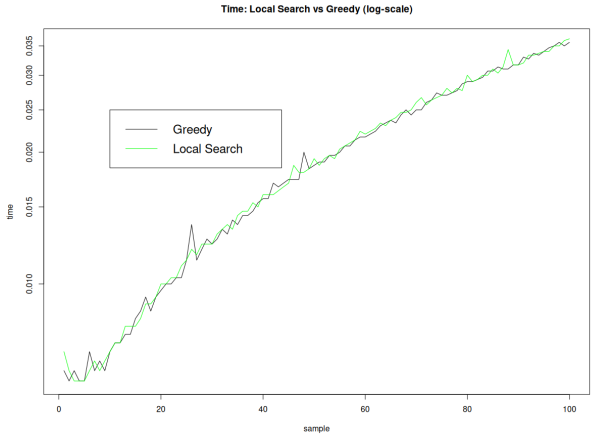
The difference in time between Greedy Search and Local Search is insignificant. Although Local Search is slightly slower, with our first improvement policy, we quickly arrive at a local optimum.



(a) Time among all different approaches



(b) Time GRASP vs CPLEX



(c) Time Greedy vs LS

Figure 4: Logarithmic Scale of time execution

5 Conclusions and Future Work

5.1 Our findings

In this project, we researched different approaches to solve a difficult optimization problem (possibly NP-complete, although the demonstration of it is out of the scope of this project).

The first approach deals with finding the optimal solution of the problem, with an ILP solver. This first approach would technically be the preferred, but with high enough problem size it starts being too slow and may only be useful when you absolutely need the optimal and can wait for hours to obtain that solution. Still, with our samples ($N \leq 100$) we managed to obtain solutions in less than 30 minutes (although some samples took almost 30 minutes).

The second approach is to use heuristic methods to obtain good solutions in much more acceptable time (with our samples the CPLEX took, sometimes, many minutes while the slowest execution of our heuristic methods took less than 10 seconds). For this approach we used three different methods: A Greedy algorithm, the Greedy + Local Search method and, finally, the GRASP meta-heuristic (that also makes use of the local search). Logically, each method provided better results than the previous one as every method improves on the previous one (the local search method is adding a step to the greedy and the GRASP method was similar to the local search but improving the selection of the initial solution).

In conclusion, as we observed in the different experiments, the CPLEX model is really useful when you absolutely need the optimal solution (and even better if the problem size is small) but with samples of big enough size (specially if you need to solve a lot of instances of the problem) it starts being much more convenient to use a heuristic method. More specifically, we found that the GRASP algorithm is usually the best. It can be tuned to reduce the number of iterations (and therefore you could *personalize* the time it takes to complete) and we also found that it usually finds more and better solutions than the other two methods. Still, a simple greedy or a greedy + local search could be easier to implement than the GRASP (and they need no tuning) and they still provide good (and fast) solutions.

5.2 Future work

There are multiple questions left open after our work, these questions could be solved with a further study of the problem, but they are out of the scope of our project. These open questions are the following:

- Each iteration in the GRASP procedure is entirely independent of the others, suggesting that parallelization could be implemented relatively easily.
- As discussed in **Subsection 4.1**, it may be interesting to conduct a more detailed experiment using the values $\alpha = 0.3$, 0.35, and 0.4. The value 0.35 provides a balance for the issue outlined in the referenced subsection.
- Different implementations of the greedy function may increase the number of solved samples, and they may also manage to get closer to optimal solutions.
- Future implementations using different heuristic methods (e.g., BRKGA or Simulated Annealing) also appear promising.
- Comparing the performance of other ILP solvers on this problem (such as Gurobi) against CPLEX is another promising direction to evaluate how both solvers behave.
- Trying different ILP models that solve this problem in order to compare performance between them may also be interesting

- We only used the *first improvement* strategy for our local search, but our project could be extended implementing *best improvement* and comparing the results with the results we obtained. One could even research whether the GRASP algorithm with *best improvement* has different optimal values of α than the ones we found.