Algorithmic Methods for Mathematical Models - COURSE PROJECT -

Marc Díaz Calderón (marc.diaz.calderon@estudiantat.upc.edu)
Pau Adell (pau.adell@estudiantat.upc.edu)

December 2024

Formalization of the problem

The objective of this problem is to form a committee from the faculty members of the School of Computer Science at Barceburg. Each of the N faculty members belongs to one of the D departments, and the committee must include a specific number of members from each department. However, not all members work well together. The challenge lies in selecting members who are compatible, based on a given compatibility matrix m. Members with zero compatibility cannot be in the committee together, and pairs with low compatibility must have a highly compatible intermediary. The goal is to maximize the average compatibility of the committee while meeting all departmental and compatibility constraints.

Integer linear model

Input Variables

The inputs of the problem are the following:

- $N \in \mathbb{N}^+$: The number of faculty members.
- $D \in \mathbb{N}^+$: The number of departments in the faculty.
- A vector n of size D, where, $n[d] \in [1, N]$ represents the number of members of the department $d \in [1, D]$ that must attend the committee. The sum of members of n can not be greater than the total faculty members N.
- A vector d of size N, where $d[i] \in [1, D]$ indicates the department of individual $i, \forall i \in [1, N]$.
- A symmetric compatibility matrix m of size $N \times N$ where the coefficients $m_{ij} \in [0, 1] \subset \mathbb{R}$ of the matrix represent the compatibility between the individual i and j. A higher value indicates better compatibility.

Auxiliary Variables

To avoid potentially costly recalculations in the objective function, we define two auxiliary variables:

- T: This variable represents the total number of individuals that will form the committee. It is computed as:
$$T = \sum_{d \in D} n_d$$
- G: This variable represents the total number of unique pairwise comparisons that can be made from a set of T individuals. It is computed using the Gauss sum formula:
$$G = \frac{(T-1) \cdot T}{2}$$

Decision Variables

- x_i: Variable will be True if the individual i is selected for the committee for $1 \leq i \leq N$.
- y_{ij}: Variable that will be True if the individual i AND the individual j are selected for the committee for $1 \leq i, j \leq N$.
- z: Variable that will contain the average compatibility of a given committee. The goal is to maximize it.

Output

We have two main outputs:

- A value that indicates the average compatibility of the selected committee in order to know the quality of the result.
- A selection of $K = sum(n)$ individuals that will form the committee. The committee must maximize the average compatibility among all selected members while satisfying the following constraints.

Constraints

1. **Department Participation:** Each department $d \in [1, D]$ must have exactly n_d participants, as specified by the input vector n. This can be expressed as:
$$\forall d \in [1, D], \quad \sum_{i=1}^{N} (\delta_{d[i],d} \cdot x_i) = n_d \quad \text{where } \delta_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases} \tag{1}$$
2. **Incompatibility Constraint:** No two individuals i and j whose compatibility $m_{ij} = 0$ can be selected simultaneously:
$$x_i + x_j \leq 1, \quad \forall i, j \in [1, N] \text{ with } m[i][j] = 0 \tag{2}$$
3. **Middleman Friend Constraint:** We also need to ensure that in the case the committee is formed by two individuals whose compatibility is less than 0.15, there must be also in the committee a "middleman friend" that has compatibility higher than 0.85 with both members. That is:
$$\sum_{k \in [1,N] \setminus \{i,j\}} \delta_{m_{ik} > 0.85} \cdot \delta_{m_{kj} > 0.85} \cdot x_k \geq x_i + x_j - 1, \quad \forall i, j \in [1, N] \text{ with } 0 < m_{ij} < 0.15 \tag{3}$$

4. **Pairwise Selection Constraint:** For each pair of individuals i and j, the variable y_{ij} must be True only if both individuals are selected:

$$y_{ij} = 1 \Leftrightarrow x_i = 1 \wedge x_j = 1, \quad \forall i, j \in [1, N] \tag{4}$$

The last constraint, will be the one in charge of maximizing the average compatibility, but this constraint is stated in the following Objective subsection.

Objective function

We want to maximize the average compatibility between all the members of the committee. As stated in the decision variables then, the objective function is expressed as:

$$\mathit{maximize}\,(z) \tag{5}$$

And this z will be upper bounded by the average calculation. Hence, by maximizing Z, we are rising the upper bound as much as possible. Then to compute z, we will just iterate without repetition, as the compatibility that individual i has with individual j is symmetrical, and divide by the total given by the Gauss sum. Then, z is defined as:

$$z \leq \frac{\sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} m_{ij} \cdot y_{ij}}{G} \tag{6}$$

1 Meta-heuristics

Here, we will describe the development of the meta-heuristic algorithms used to solve the problem. We have chosen Python as the programming language because the class-provided code is written in Python, which allows quick and straightforward development as well as easy handling of text file input and output. However, we are fully aware that optimization problems could benefit from the speed of a language like C++. The code structure consists of the three different solvers inheriting from a base class which contains helper functions common to the three solvers.

1.a Greedy Constructive algorithm

The core of our greedy algorithm can be seen in Algorithm 1. It consists of an initial empty solution and a set of all possible candidates. So, while we still haven't assigned all the required members to the committee, we will do as follows:

First, we will update our set of candidates with the `Feasibility Function` that can be seen in Algorithm 2. We then check if there are any candidates left to choose from, and choose one by selecting the one with the best score (cost), which we value according to the `CostFunction` shown in Algorithm 3. We will then assign this candidate to the solution, remove it from the set of candidates, and update the constraints and number of assigned members.

Our feasibility function is straightforward. It starts with the set of all possible candidates (in the first iteration, this includes everyone) and filters them based on three criteria. First, that the candidate belongs to a department that still needs participants (`n[d[c]]>0`). Also, that the candidate is compatible with the current partial solution (i.e., no compatibility value is 0.0) in the call to `CandidateIncompatible`. Finally, if the candidate has a compatibility

Algorithm 1: Greedy Algorithm

```
1  $S \leftarrow [-1, -1, \ldots, -1]$ ;
2  $C \leftarrow \{0, 1, \ldots, N-1\}$;
3  $count \leftarrow 0$;
4  while $count < sum(n)$ do
5      $C \leftarrow \{$FeasibilityFunction($C, n, S$)$\}$;
6      if $C$ is empty then
7          break
8      end
9      $c_{best} \leftarrow \arg\max_{c \in C}$ CostFunction($c$, $C$, $S$, $count$);
10     $S[count] \leftarrow c_{best}$;
11     $C \leftarrow C \setminus \{c_{best}\}$;
12     $n[d_{c_{best}}] \leftarrow n[d_{c_{best}}] - 1$;
13     $count \leftarrow count + 1$;
14 end
15 return $< f(S), S >$;
```

Algorithm 2: Feasibility Function

```
Input  : Set of candidates $C$, Current partial solution $S$
Output : $C'$ – Set of feasible candidates
1 $C' \leftarrow \{\}$;
2 foreach $c \in C$ do
3     if $n[d[c]] > 0$ and not CandidateIncompatible($c, S$) and not NeedsMiddlemanAndNotFound($c, C, S$) then
4         $C' \leftarrow C' \cup \{c\}$
5     end
6 end
7 return $C'$
```

below 0.15 with any member of the current solution, the function ensures that there is at least one individual, either in the list of candidates or already in the solution, with a compatibility greater than 0.85 to resolve this incompatibility with the call to `NeedsMiddlemanAndNotFound`.

The most important decision in our greedy algorithm comes in how we evaluate each candidate to determine whether it should be included in the solution. Our evaluation function consists of two main parts:

Part 1: Penalized Affinity Function

We calculate a penalized affinity value that measures the compatibility of one candidate with another. Then, given the compatibility value from the matrix `m[i][j]` we apply the following transformations:

- **Strong penalty for low compatibility:** If the compatibility value < 0.15, we strongly penalize it using an exponential decay function, which will provide a score ranging from -1.35 for compatibility 0 to -1.

$$score = -e^{2\times(0.15-value)}$$

- **Positive boost for high compatibility:** As opposed to the penality, if the compatibility value $>= 0.85$ we boost it using an exponential growth function which will provide a score ranging from $+1.85$ for compatibility 0.85 to $+2.35$ for compatibility 1.

$$score = value + e^{2\times(value-0.85)}$$

- **Neutral case:** For the rest the value remains the same as the compatibility score, hence ranging from $+0.15$ for compatibility 0.15 to $+0.85$.

$$score = value$$

Part 2: Weighted Evaluation

Using the penalized affinity function, we compute two aggregate scores for each candidate. First, the sum of scores with members already included in the partial solution and the sum of scores with all other candidates still available for selection. The final score for a candidate is a weighted sum of these two values:

$$score = W_1 \cdot \textit{Affinity with Solution} + W_2 \cdot \textit{Affinity with Candidates} \tag{7}$$

The progress ratio (amount of members already in the solution) determines the weights dynamically. Early in the process, we emphasize compatibility with all candidates to avoid future conflicts (W_2). Later, as fewer candidates remain, we prioritize compatibility with the existing solution to improve and maximize the final average compatibility (W_1).

Algorithm 3: Cost Function (GreedyCostFunction)

```
Input : Candidate to evaluate $c$, Set of candidates $C$, Current partial solution $S$ and number
        of already assigned candidates $count$
1  Function PenalizedAffinity($value$):
2      if $value < 0.15$ then
3          return $-e^{2\times(0.15-value)}$ // Strong penalty for poor affinity
4      else if $value \geq 0.85$ then
5          return $value + e^{2\times(value-0.85)}$ // Boost for strong affinity
6      return $value$
7  $sum\_solution \leftarrow \sum_{i\in S, i\neq -1}$ PenalizedAffinity($m[candidate][i]$);
8  $sum\_candidates \leftarrow \sum_{i\in C, i\neq c}$ PenalizedAffinity($m[candidate][i]$);
9  $progress\_ratio \leftarrow count \div sum(n)$;
10 $W_1 \leftarrow progress\_ratio$;
11 $W_2 \leftarrow 1 - progress\_ratio$;
12 return $W_1 \cdot sum\_solution + W_2 \cdot sum\_candidates$
```

1.b Local Search procedure

The Greedy algorithm, while efficient, may overlook better solutions due to its inherent limitations. To address this, we use a Local Search algorithm to refine the solution by exploring its neighbourhood for improvements.

Starting with an initial solution, the algorithm generates potential neighbours by swapping candidates from the current solution with **valid** candidates from the pool of not already assigned individuals from the same department. This is done in the call to `GenerateNeightbors` that can be seen in Algorithm 5. Our algorithm uses a **best-improving strategy**, where at each iteration only the swap that gives the maximum improvement is accepted. The search continues until one of the following conditions is met; a predefined maximum number of iterations is reached, the allowed time budget is exceeded or, no improvement is found in a complete iteration of swaps. The process is detailed in Algorithm 4.

Algorithm 4: Local Search
Best-improving strategy

```
Input  : Initial Solution $S_0$
1  $S \leftarrow S_0$;
2  for $it \leftarrow 1$ to $max\_iterations$ do
3      if $time.now() - start\_time \geq max\_time$ then
4          break
5      end
6      $N \leftarrow$ GenerateNeighbors($S$);
7      if $N = \emptyset$ then
8          break
9      end
10     $S' \leftarrow \arg\max_{n \in N} f(n)$;
11     if $f(S') > f(S)$ then
12         $S \leftarrow S'$;
13     end
14     else
15         break // No improvement
16     end
17 end
18 return $< f(S), S >$;
```

Algorithm 5: Generate Neighbors

```
Input  : $S$ – Current solution (list of assigned candidates)
Output : $N$ – List of valid neighboring solutions
1  $N \leftarrow \{\}$ ;
2  foreach $i \in S$ do
3      $C \leftarrow \{c \mid d[c] = d[i]$ and $c \notin S\}$;
4      foreach $c \in C$ do
5          $neighbor \leftarrow S$;
6          $neighbor.swap(i, c)$;
7          if SolutionIsValid($neighbor$) then
8              $N \leftarrow N \cup \{neighbor\}$;
9          end
10     end
11 end
12 return $N$
```

1.c Greedy Randomized Adaptive Search Procedure (GRASP)

The Greedy Randomized Adaptive Search Procedure (GRASP) is an improved version of the greedy algorithm that incorporates randomness to prevent getting stuck in local optima. The algorithm starts by generating an initial solution with some randomness in the `DoConstructionPhase`, shown in Algorithm 7, followed by a local search to improve it using `DoLocalSearch`, as explained in the previous subsection. This process continues until a stopping condition (such as a maximum number of iterations or time limit) is reached, with the best solution being kept throughout the iterations. Algorithm 6 shows the core process of our GRASP algorithm, following this procedure.

The interesting part of the algorithm comes in the generation of the solutions to which we will apply the local search procedure (i.e., `DoConstructionPhase`). The first step in generating a solution, similar to the greedy algorithm, is to create a list of possible candidates that could be assigned to the solution, using the same `FeasibilityFunction` depicted in Algorithm 2. Then, using the same cost function as in the greedy approach (explained in Algorithm 3), the candidates are ranked from best to worst. The key difference between the greedy algorithm and GRASP is that instead of selecting the best candidate, we construct a **quality based Restricted Candidate List** (RCL) and select one candidate at random from this list.

We would like to have in this list those candidates that have cost values close to the best candidate, how close will depend on the parameter α (between 0 and 1). Then, given the best and worst candidates (q_{max} and q_{min} in the equation) and the α value, we will define a threshold as follows:

$$threshold = q_{max} - \alpha \cdot (q_{max} - q_{min}) \tag{8}$$

Algorithm 6: GRASP Solver

```
1  $S \leftarrow \{\}$;
2  for $it \leftarrow 1$ to $max\_iterations$ do
3      if $time.now() - start\_time \geq max\_time$ then
4          break
5      end
6      $S' \leftarrow$ DoConstructionPhase();
7      if $S' = \emptyset$ then
8          continue
9      end
10     $S'' \leftarrow$ DoLocalSearch($S'$);
11     if $f(S'') > f(S)$ then
12         $S \leftarrow S''$;
13     end
14 end
15 return $< f(S), S >$;
```

Then, all those candidates whose cost is more or equal to the threshold we stablish will go inside the RCL. A large α (= 1) value favours randomness (more candidates in the RCL), as:

$$threshold = q_{max} - 1 \cdot (q_{max} - q_{min}) = \cancel{q_{max}} - \cancel{q_{max}} + q_{min} = q_{min}$$

And all the scores are greater or equal than the worst. On the other hand, smaller α (= 0) value favours greediness (fewer candidates in the RCL) as:

$$threshold = q_{max} - 0 \cdot (q_{max} - q_{min}) = q_{max} - \cancel{0 \cdot (q_{max} - q_{min})} = q_{max}$$

And there might be just one candidate whose value is greater or equal than the best, itself.

Algorithm 7: Construction Phase

```
1  $S \leftarrow [-1, -1, \ldots, -1]$ ;
2  $C \leftarrow \{0, 1, \ldots, N-1\}$;
3  $count \leftarrow 0$;
4  while $count < sum(n)$ do
5      $C \leftarrow \{$FeasibilityFunction($C, n, S$)$\}$;
6      if $C$ is empty then
7          break
8      end
9      $q_{max} \leftarrow \max\{$CostFunction($c, C, S, count$) $\mid c \in C\}$;
10     $q_{min} \leftarrow \min\{$CostFunction($c, C, S, count$) $\mid c \in C\}$;
11     $threshold \leftarrow q_{max} - \alpha \cdot (q_{max} - q_{min})$;
12     $RCL_{max} \leftarrow \{c \in C \mid$ CostFunction($c, C, S, count$) $\geq threshold\}$ ;
13     Select $candidate \in RCL_{max}$ at random;
14     $S[count] \leftarrow candidate$;
15     $C \leftarrow C \setminus \{candidate\}$;
16     $n[d_{candidate}] \leftarrow n[d_{candidate}] - 1$;
17     $count \leftarrow count + 1$;
18 end
19 return $S$;
```

2 Instance generation

When generating the instances, we first decided to follow an approach similar to the generations we did in the laboratory. We implemented a script in which, given a number of instances and a range of departments and members, it would generate as many datasets as instances specified in a balanced way. With this approach, we thought that, if at the time of generating datasets, these had dynamic values, we would obtain instances with greater variability.

It generates the necessary members for each department and assigns a number of them to the committee so that there are no capacity problems, and all conditions are met (which are not at all trivial), for this, we decided to make a combination between minimum standards and random choices.

Once generated the datasets, we encountered a problem that we did not expect. When we tried our datasets, we saw that most of them had no solution. Analysing as an example the datasets given, we saw that the compatibilities were not really random. All the datasets use the same matrix but with few values changed to generate different solutions and, they all have the same number of departments.

To fix this issue, we concluded that our solution would be completely random but, if in the random process, it generated a compatibility less than 0.15, our next compatibility would be higher than 0.85 in order to try to stabilize this condition. This does not mean that all instances have solutions, but in practice, most of them have. In addition, we end up choosing to always select the same number of groups (2), for all the datasets (although we also allow modifying this in the script if needed). The reason for this decision is that there is a relationship between the number of people you can assign to a group and the groups there are. If there are few groups and many people, as there are many possible options in each department, it should be more expensive to find the perfect ones.

3 Results

Once the datasets are generated, we can proceed to optimize our hyperparameter α and test the solvers to compare their execution time, and the solution they provide.

Alpha Tuning

To tune our hyperparameter α, we temporarily modified the GRASP solver to just execute the *construction phase*. To compare the alpha values, we executed 200 iterations of the GRASP solver with alpha values ranging from $\alpha = [0.0, 1.0]$ with 0.1 steps. The fitness of the solver, shown as $f(s)$ in the equation, is compared with the *real* best solution provided by CPLEX. The fitness of this CPLEX solution is shown as $\mathcal{F}(s)$. Then, we define the metric `Relative Gap` as follows.

$$\text{Relative Gap (\%)} = \frac{\mathcal{F}(s) - f(s)}{\mathcal{F}(s)} * 100 \tag{9}$$

Then we selected the datasets. We decided not to use a small number of members but also not too large, as if we had to compute $\mathcal{F}(s)$ for all of them, CPLEX would take a lot of time. The results of the Relative Gap vs the alpha value can be seen in Figures 1 and 2 for datasets of size 50 and 55 members respectively.

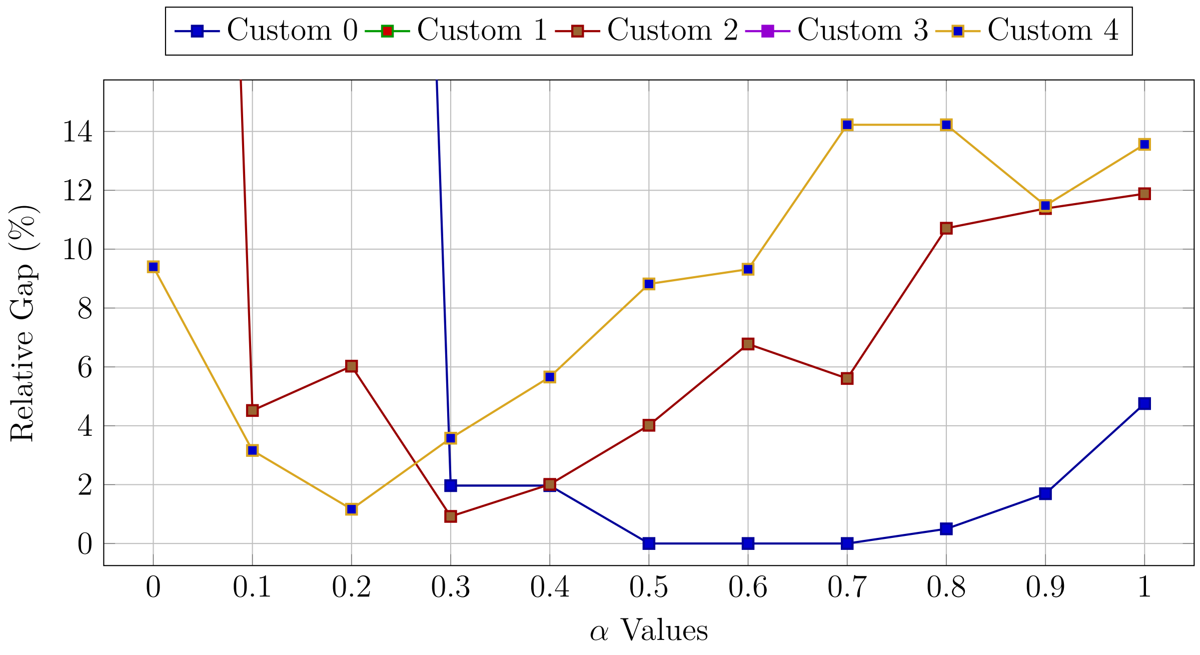

Figure 1: Relative Gap (%) vs different alpha values for different set of random instances of size N = 50

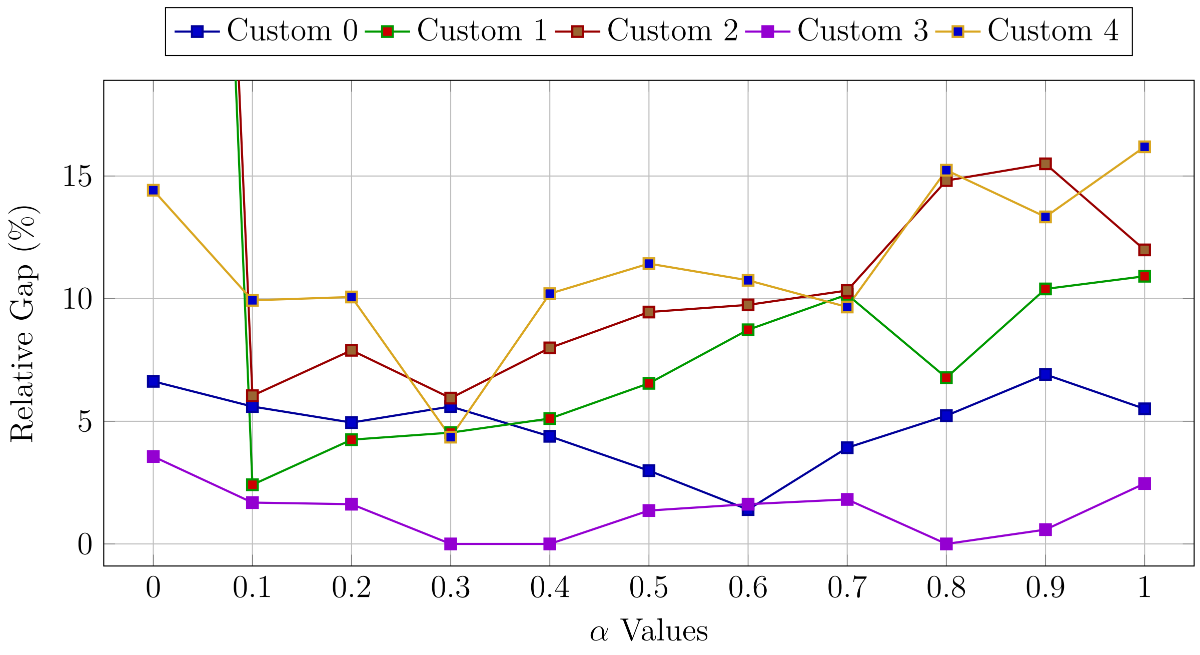

Figure 2: Relative Gap (%) vs different alpha values for different set of random instances of size N = 55

The first thing to note is that in some cases no solution was found, they have values of *Relative Gap* = 100. To make the results clearer, we have adjusted the plot height, so these values are computed but not shown.

As can be seen, for some alpha values very close to 0, i.e. a fully greedy approach, some do not find a solution. This can be explained by the fact that all executions of the GRASP give the same result. Most likely, we have a greedy algorithm that is too demanding or too focused on finding the best solution. This could be considered a bad strategy, because if we want to apply local search later, it would be more important to find a solution, even if it is a bad one. But after testing with different heuristics, the one we chose gave the best average results.

We can also see that as the value of alpha increases, the number of problems solved increases, because by adding randomness we force our algorithm to explore more options. In addition, as the randomisation increases ($\alpha > 0.6$), the solvers produce much worse solutions than for smaller alpha values. In both figures, we can see that values of $0.1 <= \alpha <= 0.4$ give the best results on average. This means that even if our greedy is able to find a solution, a touch of randomness seems to improve the solution. Given these results, we have decided to use $\boldsymbol{\alpha} = \mathbf{0.3}$ as it gives the best results more often.

CPLEX and heuristics

Finally, we compare the solvers both in terms of solution quality and execution time. For this comparison, we selected instances that not only have a feasible solution but can also be solved by our greedy algorithm (which obviously are not all of them).

We have chosen the following values, shown in Table 1, although we have to say that these are very relative as for calculating the execution time, we have found cases in which with 50 members CPLEX can take 30 minutes. The reason is completely dependent on the problem variables.

Members	**CPLEX**	**Greedy**	**Greedy+LocalSearch**	**GRASP**
10	0.10	0.0000	0.00000	0.03501
25	0.21	0.00100	0.00100	0.32870
50	37.12	0.00400	0.01951	3.15251
60	429.00	0.00700	0.04852	5.87972
70	1,800.00	0.00800	0.04426	12.33155
80	1,803.00	0.01351	0.18565	19.59827
90	1,800.00	0.01551	0.11866	38.35215

Table 1: Execution time comparison in seconds for the different solvers with different problem sizes.

As can be seen in the table, the difference in execution time is clear. While CPLEX can take more than half an hour, GRASP does not take more than a minute. The most important thing to note is that the execution time does not increase linearly, but exponentially. We can see that the greedy solver is the faster one, taking no more than 2 seconds for an instance of size 90. The local search to improve this solution takes on average ten times longer. Finally, for datasets of size 70, 80 and 90, we can see that CPLEX takes more than 30 minutes to find a solution.

And with respect to the quality of the solutions that we have achieved with the algorithms, we can see that on average the greedy achieves the worst results, the greedy + local search improves it by a little, and finally the one that achieves almost the best results consistently is the GRASP algorithm. Considering that CPLEX has a search among all possible solutions, should be consistently the one with bests results however, we can see how the GRASP solver, with much less time, has very similar results, being a very good competition.

On the other hand, we see that CPLEX does not get the best objective value with instances of members 70,80 and 90 and the reason is that we had to add a time limit (of 30 minutes) and

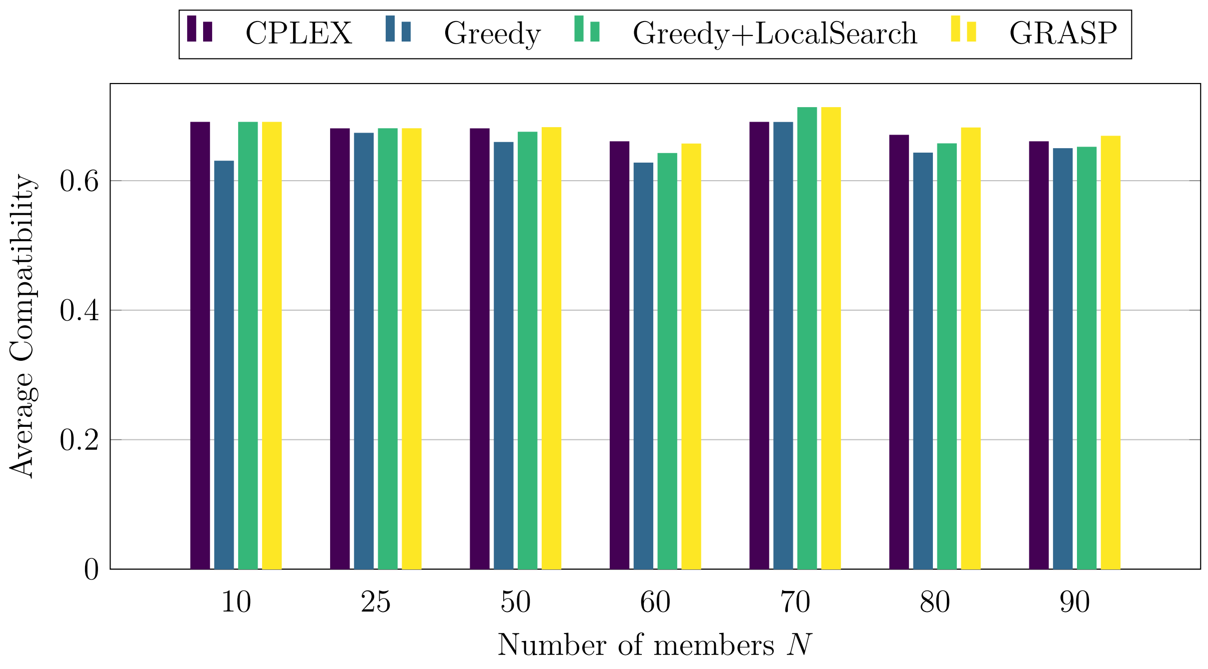

Figure 3: Comparison of the solution's average compatibility value for the 4 solvers for different instance sizes.

we indeed reach it. Meaning that the best solution CPLEX could find in that period of time is in fact lower than the one found by the GRASP method.

In conclusion, after trying the four different solvers, we see that CPLEX is the one that, if the problem instance has a solution, will end up finding the optimal one. However, this exhaustiveness comes at a price. We have seen how it reaches 30 minutes of computation time for instances of size > 70, and in some scenarios this can be far from optimal. The other three solvers offer a trade-off between reaching optimality and computation time to varying degrees. We can see that Greedy, which is objectively the worse in comparison, still achieves a fairly consistent solution in a large number of problem instances. In just over a second, it delivers a solution that is 2% worse than the optimum, making it ideal for real-time applications such as embedded systems or time-sensitive environments. If we have more time, we can use local search to improve this solution. In no more than 10 seconds, it will provide solutions much closer to the optimum in most cases. Finally, the GRASP, although with an execution time of 40 seconds for larger instances, consistently provides the **best** solution.

References

[1] L. Velasco, *Greedy, local search, grasp - heuristics*, UPC, FIB, Lecture slides, AMMM, 2024.

[2] P. Festa, "Grasp: Basic components and enhancements," *Telecommunication Systems*, vol. 46, pp. 253–271, Mar. 2011. DOI: 10.1007/s11235-010-9289-z.

[3] R. Alvarez-Valdes, F. Parreño, and J. Tamarit, "A grasp/path relinking algorithm for two- and three-dimensional multiple bin-size bin packing problems," *Computers & Operations Research*, vol. 40, no. 12, pp. 3081–3090, 2013, ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2012.03.016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054812000779.