POLITECNICO DI TORINO

Master's Degree in Engineering and Management

Master's degree thesis

Analysis and algorithms to solve the bilevel assignment problem

Supervisors:
Federico Della Croce di Dojola
Rosario Scatamacchia

Candidate:
Francisco Brogiolo

July 2021

Table of Contents

1. Introduction

1.1. Description of the bilevel assignment problem

In bilevel optimization problems, there are two collaborative or conflicting decision makers, named the leader who acts first, and the follower that reacts depending on the leader's choice. On its general form, each decision maker has its own decision variables and objective functions, but there are common constraints. There has been an increasing interest in bilevel programming on the scientific community since the 2000s, due to its capability to model complex problems in which more than one party are involved, e.g., the government (leader) controlling policy variables (such as tax rates) to maximize employment or minimize the use of a resource, and the industry (follower) being regulated and trying to maximize net incomes.

The assignment problem (AP) is well known in the literature and the typical problem consists of assigning n origins i to n destinations j, such that, given a costs matrix of size n *x* n in which each value corresponds to each cost c_{ij} (of assigning origin i to destination j) the total cost obtained as a linear sum is minimized. Each origin and each destination must be assigned once. The very first algorithm that solves this problem efficiently was proposed by H.W. Kuhn in 1955 and is known as *Hungarian Method* [1], which by exploiting structural properties of the problem is able to solve it in $O(n^3)$ time.

In this thesis, we consider the bilevel assignment problem (BAP), in which the leader is given n origins and n destinations, and he must select k origins and k destinations (with $k<n$). The follower proceeds to solve the assignment problem of the corresponding selected nodes, that form a costs matrix of size k *x* k. The leader wants to carry out the selection in a way that the optimal solution of the follower is maximized. There exist some papers that work on the BAP. For instance, in [2] exact solutions approaches are developed, but the problem faced in this thesis has some particularities: the edges (costs) are common to both decision makers, and the objective functions of the leader and the follower are not aligned, since the former wants to maximize the solution of the latter, who in turn seeks to solve the minimization version of the assignment problem.

1.2. Objective and structure of the thesis

The aim of this thesis is to solve the considered BAP, by developing exact and relaxed models, metaheuristic and matheuristic approaches. It consists of eight chapters.

In the second one, general theory is presented about optimization problems, linear programming, and computational complexity. In the third one, specific concepts of the assignment problem and of bilevel programming are explained. The next three present the resolution approaches and algorithms.

In chapter four, we first present the mathematical formulation of the BAP, together with the corresponding dual formulation that is required to implement the model in the MIP solver. Then, we test several variations and a relaxation of the model, and we evaluate them by considering the achieved results.

Chapter five focuses on metaheuristic approaches to solve the BAP. We first develop greedy algorithms, followed by local search and iterated local search algorithms. Finally, we show that a final enhancement of the neighborhood search, based on an iterative construction of upper bounds to the optimal solution of the assignment problem, leads to better performances.

In chapter six, we first present some theory about matheuristics. Then, we develop and implement a matheuristic algorithm, with the aim of improving the results obtained by the heuristic approaches.

Results are presented and compared in chapter seven, together with the final conclusions of the thesis. Finally, the references are listed in chapter eight.

2. Optimization problems and computational complexity

2.1. Introduction

Throughout our lives as human beings, but in nature itself as well, countless decisions are taken every day. Many of them contain several alternatives which produce different results. Being aware of it or not, we continuously improve our decisions, until we are satisfied enough with the obtained output or because there is no way to enhance it anymore. After that, we just tend to repeat it, as it is considered the one that produces the "optimal" outcome. Determining which route to take from home to work so that it is the shortest or the fastest, scheduling (even implicitly) the work to be done to cook in the lesser time possible or the timetable to be able to perform the multiple desired activities every week without overlapping each other, or without overload any particular day, are a few of the endless list of examples.

Moreover, as a society, the importance of appropriately solving complex optimization problems have huge impacts on diverse areas. Consider for example, the energy and cost wastes reached by a manufacturing plant if the related production line is not studied and optimized (i.e. the machines need to stay more time turned on to produce a single unit, times the x units that the plant produces over a certain period of time); the traffic problems on a big city where public transport and streets are not appropriately planned; or food shop's delivery costs if the deliveryman does not follow the shortest path to reach its clients.

2.2. Formal definition and classification

Let us define an optimization problem as any problem in which the aim is to find the best solution, based on an expected or desired output, among all feasible solutions. If one wants to analyze these decisions in an organizational level, their complexity grows enormously, and we start to explicitly define the objective function, the constraints and the set of parameters that will produce the output. Formally, recalling [3], we need to find x to:

$$\begin{aligned} &minimize\ f(x) \\ &subject\ to\ (s.t.)\ g_i(x) \geq 0 \quad i = 1, \dots, m \\ &\qquad\qquad h_j(x) = 0 \quad j = 1, \dots, p \end{aligned} \tag{2.1}$$

Where the problem is classified as a *nonlinear programming problem* for the case in which f, g_i and h_j are general functions of the parameter $x \in \mathbb{R}^n$. When f is convex, g_i concave, and h_j linear, then the problem is called a *convex programming problem*.

If all those functions are linear, the problem is further categorized as a *linear programming problem*.

Definition [3]

An *instance of an optimization problem* is a pair (F, c) where F is any set, the domain of feasible solutions[1]; c is the cost function, a mapping:

$$c: F \rightarrow \mathbb{R}^1$$

The problem is to find an $f \in F$ for which:

$$c(f) \leq c(y) \; for \; all \; y \in F$$

Such a point f is called a *globally optimal* solution to the given instance, or, when no confusion can arise, simply an *optimal solution.* □

Definition [3]

An *optimization problem* is a set I of instances of an optimization problem. □

Informally speaking, an *instance* corresponds to a set of data such that we have enough information to obtain a solution, while a *problem* is a set of instances, usually generated in the same way.

Moreover, optimization problems can be divided into those with continuous variables and those with discrete variables that are known as combinatorial problems. In the first ones, we are looking for a set of real numbers or a function; in the second ones for an object from a finite set. The optimization of a combinatorial problem is the field called Combinatorial Optimization.

2.3. Linear programming

In the context of this thesis, linear programming is relevant as it is one way to solve the assignment problem. Moreover, for the BAP formulation we will take advantage of duality theory, so a brief introduction to linear programming and duality theory is presented.

2.3.1. Linear programming and the standard form of a linear program

Linear programming is a field in mathematics, developed in the twentieth century, which presents many applications in engineering, economics, logistics, chemistry, etc. As defined in section 2.2, a linear program is a problem which objective function and constraints present only linear functions. A widely use way to solve this kind of problems is to apply the SIMPLEX method (see chapters 4 and 6 of [4]).

[1] A feasible solution is a solution that satisfies all the constraints.

The definition of the *standard form* of a linear program varies in literature (e.g., in [5] it is defined as a minimization problem with equality constraints), but let us present the *standard maximization form* as defined in [6]:

$$\begin{aligned} &maximize\ c^T x \\ &s.t.\ Ax \leq b \\ &x \geq 0 \end{aligned} \tag{2.2}$$

Where:

The vector $c = \begin{pmatrix} c_1 \\ \cdots \\ c_n \end{pmatrix} \in \mathbb{R}^n$ is the column vector of coefficients of the objective function, c^T is the transpose matrix of c, $x = \begin{pmatrix} x_1 \\ \cdots \\ x_n \end{pmatrix}$ is the column vector of variables (so the product $c^T x$ gives a value as a result), A is the $n\ x\ m$ matrix of coefficients of the left-hand side of the inequalities and b is the m-dimensional vector of the right-hand side of the inequalities.

2.3.2. Linear programming duality

Given a linear program in the considered *standard form*, called the primal, then it can be converted into its dual, which is a *minimization linear program* able to give an upper bound to the primal and it is defined as:

$$\begin{aligned} &minimize\ b^T y \\ &s.t.\ A^T y \geq c \\ &y \geq 0 \end{aligned} \tag{2.3}$$

We may apply the following rules to obtain the dual. If the primal has a maximization objective function, the dual will have a minimization objective function. For every constraint (excluding the restrictions in sign for vector x) of the dual, we define one variable y_i, that is, we have m variables y_i. Moreover, we define as many dual constraints (excluding the restriction in sign constraints for vector y) as the number of primal variables. We take the transpose of the matrix A of coefficients on the left-hand side of the inequality. Finally, we switch the coefficients vector in the objective function by the right-hand-side vector of the inequalities.

We proceed to recall the following theorems of weak and strong duality.

Weak duality theorem [6]

If LP1 is a linear program in *maximization standard form*, LP2 is a linear program in *minimization standard form*, and LP1 and LP2 are duals of each other, then:

- If LP1 is unbounded then LP2 is unfeasible
- If LP2 is unbounded then LP1 is unfeasible
- If LP1 and LP2 are both feasible and bounded, then:

$$opt(LP1) \leq opt(LP2)$$

Proof: suppose we have a maximization linear program in *standard form*:

$$\begin{aligned} &maximize\ c_1x_1 + \cdots + c_nx_n \\ &s.t.\ \ a_{1,1}x_1 + \cdots + a_{1,n}x_n \leq b_1 \\ &\quad\vdots \\ &\quad a_{m,1}x_1 + \cdots + a_{m,n}x_n \leq b_m \qquad (2.4) \\ &\quad x_1 \geq 0 \\ &\quad\vdots \\ &\quad x_n \geq 0 \end{aligned}$$

For every choice of non-negative scaling factors $y_1, \ldots, y_m$, we can derive the inequality:

$$\begin{aligned} &y_1 \cdot (a_{1,1}x_1 + \cdots + a_{1,n}x_n) \\ &\qquad + \cdots \\ &+ y_m \cdot (a_{m,1}x_1 + \cdots + a_{m,n}x_n) \\ &\leq y_1b_1 + \cdots + y_mb_m \end{aligned}$$

Which is true for every feasible solution $(x_1, \ldots, x_n)$ to the linear program (2.4). We can rewrite the inequality as:

$$\begin{aligned} &(a_{1,1}\,y_1 + \cdots + a_{m,1}\,y_m)x_1 \\ &\qquad + \cdots \\ &+ (a_{1,n}\,y_1 + \cdots + a_{m,n}\,y_m)x_n \\ &\leq y_1b_1 + \cdots + y_mb_m \end{aligned}$$

A certain linear function of the x_i is always at most a certain value, for every feasible $(x_1, \ldots, x_n)$. Then we choose the y_i so that the linear function of the x_i for which we get an upper bound is, in turn, an upper bound to the cost function of $(x_1, \ldots, x_n)$. We can reach this by choosing y_i such that:

$$\begin{aligned} &c_1 \leq a_{1,1}\,y_1 + \cdots + a_{m,1}\,y_m \\ &\quad\vdots \qquad (2.5) \end{aligned}$$

$$c_n \leq a_{1,n}\, y_1 + \cdots + \ a_{m,n}\, y_m$$

Now we see that for every non-negative $(y_1, \ldots, y_m)$ that satisfies (2.5), and for every $(x_1, \ldots, x_n)$ that is feasible for (2.4):

$$\begin{aligned} & c_1 x_1 + \cdots + \ c_n x_n \\ & \leq \left(a_{1,1}\, y_1 + \cdots + \ a_{m,1}\, y_m\right) x_1 \\ & \qquad + \cdots \\ & + \left(a_{1,n}\, y_1 + \cdots + \ a_{m,1}\, y_m\right) x_n \\ & \leq y_1 b_1 + \cdots + \ y_m b_m \end{aligned}$$

Clearly, we want to find the non-negative values $y_1, \ldots, y_m$ such that the above upper bound is as strong as possible, then we want to:

$$\begin{aligned} & \textit{minimize } b_1 y_1 + \cdots + b_m y_m \\ & s.t. \ \ a_{1,1} y_1 + \ \cdots + a_{m,1} y_m \geq c_1 \\ & \qquad \vdots \\ & \qquad a_{n,1} y_1 + \ \cdots + a_{m,n} y_m \geq c_n \\ & \qquad y_1 \geq 0 \\ & \qquad \vdots \\ & \qquad y_m \geq 0 \end{aligned}$$

Therefore, if we want to find the scaling factors that give us the best possible upper bound to the optimum of a linear program in *standard maximization form*, we end up with a new linear program in *standard minimization form*, called the dual, proving the third statement.

Finally, by observing the third statement we realize that it is also saying that if LP1 and LP2 are both feasible, then they have to both be bounded, because every feasible solution to LP2 gives a finite upper bound to the optimum of LP1 (which cannot then be $+\infty$) and every feasible to LP1 gives a finite lower bound to the optimum of LP2 (which cannot then be $+\infty$), proving the first and second statement. □

The strong duality theorem states that if either the primal or the dual are feasible, then the two local optima are equal to each other. The proof of this theorem can be found in [7].

We may define the *duality gap* as the difference between the *primal* and the *dual* solutions. Given an optimal solution of the primal p^* and the optimal dual solution d^*, the *duality gap* is equal to $d^* - p^*$. The gap is 0 only when the strong duality

theorem holds. Otherwise, if the primal is a maximization problem, then the gap is always strictly greater than 0.

2.4. Computational complexity

Computational complexity focuses on the computational tasks' complexity. Its aim is to determine what computers cannot do, drawing the line between what is possible and what is not. As explained in [8] there exist two fundamental computational tasks in the context of complexity theory known as search problems and decision problems. The formers consist of a specification of a set of all valid solutions, possibly an empty one, for each possible instance. The aim is to find the corresponding solution. The assignment problem and the travel-salesman problem are included in this category. In the latters, one is required to determine whether the instance is in the specified set. These are problems whose answer is yes or no.

A computation is a process that is able to iteratively modify an environment by applying certain rules. In each application, the rule modifies and depends on only a portion of the environment, known as the active zone. Our aim is to design computational rules or algorithms that are able to produce a desired output by altering the environment. This leads to the mapping from input *x* to output *y* reached by the computation. In this way, the algorithm determines a function, which is precisely the mapping of inputs to outputs.

2.4.1. Time complexity and space complexity

The complexity of an algorithm refers to the resources that it needs to find the solution of a problem. Commonly studied resources are time complexity and space complexity. The first is related to the execution time, which is the number of elementary operations that are needed to solve the problem. It is studied as a function of the input size of the problem, meaning that *execution time = f(n).* Time complexity is the worst-case execution time of an algorithm, and it is denoted by a capital "o". In mathematical terms:

$$O\big(f(n)\big): \exists\ a\ constant\ c: execution\ time \leq f(n) * c$$

This means that the time complexity of the algorithm is defined by the maximum number of steps needed to solve an instance of that specific size. *Efficient algorithms* are defined as polynomial-time-algorithms, meaning that the upper-bound time complexity is a polynomial function of the input size ($O(n^k)$ for $k \in \mathbb{N}$). Moreover, the time complexity of a given problem is the complexity of the fastest algorithm that solves that problem.

Space complexity can be defined as the amount of temporary memory consumed by the computation for storing intermediate results of the computation, i.e., without considering the memory required to store the input and output information. It is also a function of the input size and the worst-case is considered.

2.4.2. Classes P and NP

We can define the Class P as the set of problems that in the worst case are solvable by means of an efficient (polynomial time) algorithm. The assignment problem falls in this category.

The class NP is the set of decision problems solvable by a non-deterministic polynomial time algorithm. That is, if the answer for a given decision problem is "yes", checking that this hypothesis is correct can be done in polynomial time. An example of a problem belonging to this class is the Sudoku: given a complete grid (a solution), verifying if it is correct or not is doable in polynomial time. However, current algorithms take exponential time to find a solution for difficult instances.

Moreover, a problem X is in the NP-complete class if it satisfies two conditions:

1. X is in the class NP.
2. Each problem in the class is polynomially reducible to X. [2]

One of the biggest questions that has not been answered by the scientific community is whether P=NP. It refers to whether or not all the problems belonging to NP class (in which verifying if a solution is correct or not takes polynomial-time) can be solved efficiently (a solution can be found in polynomial time). Explained in a different way, its aim is to determine whether or not finding solutions is harder than checking their correctness. Proving that a problem is in NP-Complete implies that the problem is not in P unless P=NP.

Finally, a problem X is in NP-Hard if there exists an NP-Complete problem that reduces to it, but it has not been proven that $X \in$ NP.

2.4.3. Computational complexity of the BAP

In [9], a bilevel assignment problem is presented, in which the edges controlled by the leader are different to the ones controlled by the follower and the objective functions of both decision makers seek to minimize a linear sum or bottleneck functions. The authors proved that when the pessimistic rule is applied (an explanation of the pessimistic and optimistic positions is presented in section 3.2.1), all variants arising if the leader's and follower's functions are sum or bottleneck functions are NP-hard. If the optimistic rule is applied, they proved that the problem is NP-hard if at least one decision maker has a sum objective function.

However, the bilevel assignment problem considered in this thesis, in which the edges controlled by the leader and the follower are the same, and the leader's objective function seeks to maximize the optimal solution of the minimization

[2] A problem is reducible to another if it is possible to efficiently solve the former provided an algorithm that efficiently solves the latter. For more information, the reader may refer to Section 2.2.1 of [8].

function of the follower, remains open with respect to its computational complexity, since it has not been proved it is NP-hard.

3. Assignment problem and bilevel programming

3.1. The assignment problem

In chapter 5, a heuristic approach is proposed to solve the BAP, in which the resolution of the assignment problem can be considered as the "elementary operation" of the algorithm. Because of that, this section focuses on the main aspects of the AP: definition, variations, resolution approaches and applications.

3.1.1. Definition of the assignment problem

In order to define the assignment problem, let us first introduce some concepts of graph theory. A graph G is a pair G = (V, E) where V is a finite set of nodes or vertices, and E has as elements subsets of V of cardinality two, called edges. Given a graph B= (W, E), if the set of vertices W can be partitioned in two sets, U and V, and each edge in E has one vertex in U and one vertex in V, then B is called a bipartite graph and it is usually denoted by B = (U, V, E). A matching M of a graph G consists of a subset of edges with the property that no two edges of M share the same node, and it is a perfect matching if it covers every vertex of the graph.

Then, consider a bipartite graph B = (U, V, E) with real-valued weights on its edges, and suppose that B is balanced, with $|U| = |V|$. The assignment problem asks for a perfect matching in B of minimum total weight.

From now on, we will stick to the minimization version of the assignment problem. If we would want to maximize the total weight, we should just negate all the weights, and solve the minimization problem. Please consider that the terms row and origin (i_s) are equivalent, and so the terms column and destination (j_s).

Variables c_{ij} represent the weights or costs of each edge, variables x_{ij} represent whether an edge is selected or not, and let us consider $|U| = |V| = n$. Then, the assignment problem can be formulated as follows:

$$min \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} x_{ij} \tag{3.1}$$

$$s.t \sum_{j=1}^{n} x_{ij} = 1 \qquad \forall j \tag{3.2}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall j \tag{3.3}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \tag{3.4}$$

Where (3.1) is the objective function, constraints (3.2) and (3.3) enforce that exactly one edge must be selected for every origin i and destination j, and (3.4) enforces that variables x_{ij} are binary. If $x_{ij} = 1$, the edge linking origin i and destination j is selected, otherwise, it is not selected.

We can notice that this is an integer linear program. However, the problem can be solved as a continuous program by replacing constraint (3.4) by (3.5):

$$x_{ij} \geq 0 \qquad \forall\, i,j \tag{3.5}$$

This change allows in theory fractional variables, but in practice the optimal solution takes only integer variables as the constraint matrix is totally unimodular[3] [10].

3.1.2. Some variations of the assignment problem

In the classic AP, $|U| = |V|$, so the problem is called balanced. In the case that this condition is not satisfied, let us say $|U| = n$ and $|V| = r$, with $n > r$, the problem is unbalanced. To solve it, one possibility is to add $n - r$ dummy vertices in U and their corresponding $(n - r)*n$ edges with large positive costs, and then solve the problem as if it were balanced. We mention that the unbalanced problem can be reduced to a balanced one by applying a more efficient method called *standard doubling technique* and that the unbalanced problem can also be solved directly by taking an algorithm for the balanced case and trying to generalize it to handle the unbalanced case (for more details about these two methods, please refer to [11]).

If the objective function seeks to minimize the maximum over all the selected costs:

$$min\, max\, (c_{ij}x_{ij})\ \forall\, i, \forall\, j$$

Then the problem is called linear bottleneck assignment problem and can also be solved in polynomial time. One algorithm is presented in [12].

Finally, we may introduce the quadratic assignment problem, a variation of the AP that is not only NP-hard and hard to approximate but it is also a practically intractable problem [13]. Explained as a facility location problem, it consists of n facilities that are to be assigned to n locations. For each pair of facilities, a weight or flow is specified (matrix $A = a_{ij}$) and for each location a distance is specified (matrix $B = b_{ij}$). The cost of allocating facility $\pi(i)$ to location i and facility $\pi(j)$ to location j is given by $a_{\pi(i)\pi(j)}b_{ij}$. The problem is to find the assignment π of locations to facilities such that the total cost obtained as the sum of the product between distances and flows is minimized:

[3] A matrix is called totally unimodular if each of its sub determinants is 0, 1, or −1. An integral matrix A is totally unimodular if and only if for each integral vector b, the set $\{x : Ax \leq b, x \geq 0\}$ is an integer polyhedron [31].

$$min \sum_{j=1}^{n} \sum_{i=1}^{n} a_{\pi(i)\pi(j)} b_{ij}$$

3.1.3. Approaches to solve the assignment problem

There exist different ways to solve the assignment problem. The easy though extremely computational expensive enumeration method consists of simply going over all feasible solutions, calculate its corresponding value and choose the best one. Simplex method can also be applied to the continuous program presented in section 3.1.1. Moreover, as the assignment problem is a special case of the transportation problem in which the flows are either 0 or 1, the transportation method (see [14]) is able to solve the AP as well. However, the most efficient method to solve the assignment problem, was developed in 1957 and is known as *The Hungarian Method*, that solves the problem in polynomial time.

Given a balanced assignment problem, the corresponding steps to apply the *Hungarian Method* are the following ones. If the problem is unbalanced, we can apply one of the methods mentioned in the previous section to reduce it to balanced.

1. From the cost's matrix find the smallest value of each row and subtract it from each element of the corresponding row.
2. On the matrix obtained above, find the minimum value for each column, and subtract it from each element of that column. In this way now there is at least one 0 in each row and column.
3. Look over the rows until a row with only one 0 is found. Mark with a circle that 0 and mark the rest of the elements of that row with a cross. Once all rows have been inspected, apply this procedure to all the columns.
   If a row and/or column has two or more zeros, and one cannot be chosen by inspection then assign arbitrary any one of these zeros and cross off all other zeros of that row or column. Repeat this procedure until all elements are assigned either a circle or a cross.
4. If the number of circles is equal to n, then the optimal solution was found and is to select the circled costs. Otherwise go to step 5
5. Draw the minimum number of horizontal and/or vertical lines to cover all the zeros of the reduced matrix.
6. Find the smallest element of the matrix that is not covered by any line. Rewrite a new cost matrix by subtracting that value from all uncovered elements and adding it to the elements covered by two lines.
7. Go to step 4 and repeat until n circled elements are found.

A small example is presented below. Let us define the following *4x4* cost's matrix:

	j = 1	j = 2	j = 3	j = 4
i = 1	10	6	5	8
i = 2	8	3	4	7
i = 3	9	6	9	12
i = 4	10	5	8	10

Step 1) Minimum elements of each row = [5,3,6,5]. After subtracting them to each element of the corresponding row, the new matrix is:

	j = 1	j = 2	j = 3	j = 4
i = 1	5	1	**0**	**3**
i = 2	5	**0**	1	4
i = 3	**3**	0	3	6
i = 4	5	0	3	5

Step 2) Minimum elements of each column = [3,0,0,3]. New matrix:

	j = 1	j = 2	j = 3	j = 4
i = 1	2	1	0	0
i = 2	2	0	1	1
i = 3	0	0	3	3
i = 4	2	0	3	2

Step 3) Mark zeros with a circle and the rest of the row/column with a cross (only one circle per column/row):

	j = 1	j = 2	j = 3	j = 4
i = 1	~~2~~	~~1~~	(0)	~~3~~
i = 2	~~2~~	(0)	~~1~~	~~1~~
i = 3	(0)	~~0~~	~~3~~	~~3~~
i = 4	~~2~~	~~0~~	~~3~~	~~2~~

Step 4) There are 3 circles but $n = 4$. Go to step 5.

Step 5) Draw the minimum number of horizontal/vertical lines to cover all zeros.

	j = 1	j = 2	j = 3	j = 4
i = 1	~~2~~	~~1~~	~~0~~	~~0~~
i = 2	2	~~0~~	1	1
i = 3	~~0~~	~~0~~	~~3~~	~~3~~
i = 4	2	~~0~~	3	2

Step 6) Smallest value not covered by any line = 1. Subtract it from every uncovered element and add it to elements covered by two lines. Mark the zeros with a circle again.

	j = 1	j = 2	j = 3	j = 4
i = 1	2	2	0	(0)
i = 2	1	0	(0)	0
i = 3	(0)	1	3	3
i = 4	1	(0)	2	1

Step 7) We find 4 circled elements, so the solution that consists of assigning $i = 1$ to $j = 4$, $i = 2$ to $j = 3$, $i = 3$ to $j = 1$ and $i = 4$ to $j = 2$ is optimal, with value 26.

3.1.4. Applications of the assignment problem

One typical example of a practical application of the AP is the assignment of taxi drivers to clients in such a way that the total distance to reach all the clients is minimized. Other examples in which AP is useful in real-word applications, recalling [15], are:

- In assigning machines to factory orders.
- In assigning people/employees to perform certain jobs.
- In assigning contracts to bidders by systematic bid-evaluation.
- In assigning teachers to classes.
- In assigning accountants to clients' accounts.
- In assigning taxi drivers to passengers.

3.2. Bilevel optimization

3.2.1. Definition of bilevel optimization

Bilevel optimization is defined as a mathematical program, where an optimization problem contains another optimization problem as a constraint [16]. This kind of optimization problems presents two hierarchical decision makers. Each one tries to optimize its own objective function and the outcome of every decision of the upper-level authority (leader), will depend on the optimal solution found by the lower-level authority (follower). The leader has control over the upper-level variables, and knows beforehand the follower's interest and constraints, while this last reacts in consequence of the leader's decision and is only aware of his own objective function and constraints. General bilevel programming is known to be strongly NP-Hard [17], and we mention that some problems are even harder to solve in the polynomial-time hierarchy (a generalization of the complexity classes introduced in section 2.4), as proved in [18].

Definition [16]

For the upper-level objective function $F: \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and the lower-level objective function $f: \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ the bilevel problem is given by:

$$\underset{x_u \in X_U, x_l \in X_L}{"min"} F(x_u, x_l)$$

$$subject\ to\ x_l \in \underset{x_l \in X_L}{argmin}\{f(x_u, x_l): g_j(x_u, x_l) \leq 0, j = 1, \dots, J\}$$

$$G_k(x_u, x_l) \leq 0, k = 1, \dots, K$$

Where $G_k: \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$, $k = 1, \dots, K$ denote the upper-level constraints and $g_j: \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ represent the lower-level constraints, respectively. Equality constraints may also exist and they have been avoided for brevity. The sets $X_U \in \mathbb{R}^n$ and $X_L \in \mathbb{R}^m$ in the definition may denote additional restrictions like integrality. It is common to assume these to be sets of reals, unless mentioned otherwise. □

The quotes in the upper-level minimization function denotes that there may exist uncertainty in which value is to be taken when the lower-level problem presents multiple optimal solutions. This ambiguity is usually resolved by taking two different positions:

- **Optimistic position:** it occurs when the follower, in presence of multiple optimal solutions, chooses the one that favors the leader the most.
- **Pessimistic position:** in this situation, contrarily to the optimistic one, given multiple optimal solutions the follower selects the one that will result in the worst solution value for the leader.

3.2.2. Applications of bilevel programming and interest over time

Bilevel programming has several real-word applications, and its interest has been increasing in the last two decades. For example, in a chemical industry bilevel problems can model a complex reaction, in which the upper-level problem wants to maximize the reaction's output, and the lower-level problem appears as an equilibrium condition, which is an entropy functional minimization problem [19]. Let us enumerate some other relevant applications of bilevel programming:

- **Environmental Economics**: where an authority wants to tax an organization or individual that is polluting the environment through its operations. The authority wants to find the optimal level of tax, considering both revenues and pollution.
- **Optimal design:** in cases of structural optimization or optimal shape design. One example is the kind of problems in which the leader wants to minimize the cost or the weight of a structure, while the follower solves the potential energy minimization problem.
- **Facility location:** this kind of problems can be formulated with bilevel programming if while deciding the location the organization considers its competitor's reaction i.e., the competitors adapt their facilities in order to make them more attractive.
- **Machine learning:** while applying this optimization technique, there are a number of parameters to be defined, that are normally tuned by using brute forces strategies such as random search or grid search. Bilevel optimization can achieve a more efficient search for those parameters.
- **Principal-agent problem:** when the problem involves a principal (leader) that sub-contracts an agent (follower) to act in his behalf. Usually, the agent tries to act in favor of his own interest, so the bilevel problem involves the creation of the contract that is optimal for the principal.

It is relevant to add that in [16], an extensive study was carried out and proved that the interest over time of bilevel programming has grown significantly since the middle of the previous decade. These results are presented in Figure 3.1. Moreover, the topics that presented the biggest growth are supply chain, telecommunication, facility location and railway applications.

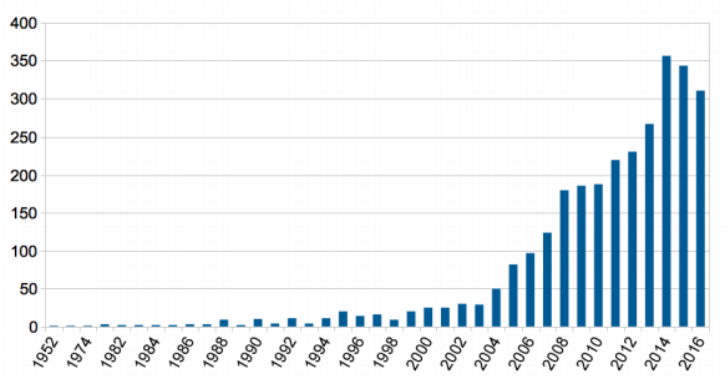

Figure 3.1 Number of publications on bilevel programming every year, extracted from [16]

4. Exact and approximate solutions of the bilevel assignment problem

In this Chapter, an exact approach is proposed to solve the BAP, starting from the mathematical formulation, and it is later improved by means of a relaxation and a simplification of the model.

4.1. Mathematical formulation of the BAP

Let us recall that in this BAP, the leader has to select k over n (with $k<n$) origins and destinations, in such a way that the solution for the corresponding AP of size k, that represents the follower's reaction, is maximized.

Therefore, if we wish to take the AP formulation given by (3.1) to (3.3) and (3.5) as a starting point, we have on the one hand, to define new variables that represent the leader's selection. Let us achieve this by adding binary variables o_i (for origins) and d_j (for destinations), for $i, j = 1, 2, \ldots, n$. If the origin or destination is selected, o_i or d_j takes value 1, and otherwise 0.

On the other hand, we have to add the leader's objective function, that wants to maximize the solution of the AP. In this way, we can formulate the BAP as follows:

$$g_l = Max(g_f) \tag{4.1}$$

$$s.t. \sum_{i \in I} o_i = k \tag{4.2}$$

$$\sum_{j \in J} d_j = k \tag{4.3}$$

$$g_f = min \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} x_{ij} \tag{4.4}$$

$$s.t \sum_{j=1}^{n} x_{ij} = o_i \qquad \forall \text{ i} \tag{4.5}$$

$$\sum_{j=1}^{n} x_{ij} = d_j \qquad \forall \text{ j} \tag{4.6}$$

$$x_{ij} \geq 0 \qquad \forall \text{ i, j} \tag{4.7}$$

$$o_i, d_j \in \{0,1\} \qquad \forall \text{ i, j} \tag{4.8}$$

Where:

- Constraints (4.1) and (4.4) are the objective functions of the leader and the follower.
- Constraints (4.2) and (4.3) enforce that the leader must select k origins and k destinations.
- Constraints (4.5) and (4.6) represent the typical AP constraints. Notice that the right-hand side is 1 in the case the origin or destination is selected and 0 otherwise.
- Constraints (4.7) and (4.8) are the variable's constraints.

Clearly a MILP solver cannot handle the presence of two nested problems. At the same time, we manage to derive a single linear model to solve the BAP. We reformulate constraints (4.4) to (4.7) on its dual form, leading to an aligned objective function in maximization form and allowing to formulate the problem in the solver with only one model.

By duality theory (see section 2.3.2), the dual model will have a maximization objective function, variables u_i and v_j (constraints 4.5 and 4.6) which will be unrestricted in sign, constraints of the form $u_i + v_j \leq c_{ij}$ and constraints 4.2 and 4.3 remain unchanged. Moreover, we need to add some big-M constraints. The resulting model is the following:

$$\max \sum_{i=1}^{n} o_i u_i + \sum_{j=1}^{n} d_j v_j \tag{4.9.a}$$

$$s.t \quad u_i \leq o_i M \qquad \forall \text{ i} \tag{4.10}$$

$$v_j \leq d_j M \qquad \forall \text{ j} \tag{4.11}$$

$$u_i + v_j \leq c_{ij} + M\left(2 - o_i - d_j\right) \quad \forall \text{ i, j} \tag{4.12}$$

$$\sum_{i \in I} o_i = k \tag{4.13}$$

$$\sum_{j \in J} d_j = k \tag{4.14}$$

$$o_i, d_j \in \{0,1\} \qquad \forall \text{ i, j} \tag{4.15}$$

$$u_i,\ v_j \quad \text{unrestricted in sign } \forall \text{ i, j} \tag{4.16}$$

The big-M method ensures that constraint (4.12) is only applied to the selected origins and destinations. We can notice that the value for M must be at least the maximum(c_{ij}). In section 4.2.1 an analysis of the optimum value of M is presented.

This formulation presents a non-linear objective function of the form (binary*continuous) variables. Although there exist solvers, like CPLEX, that

manage to solve this kind of problems, it is not the best approach because a non-linear program is harder to solve than a linear program.

As an example, Figure 4.1 presents the Engine Log results from CPLEX Studio IDE 20.1 (next times referred as CPLEX), while using this non-linear formulation to solve instance 1 (a detailed explanation of how the instances were generated is presented in section 4.3), for *k=2*, *n=10*, a time limit of 60 seconds and M = 100.

```
! ------------------------------------------------------------------------
! Search terminated by limit, 293 solutions found.
! Best objective         : 182 (gap is 943,4%)
! Best bound             : 1.899
! ------------------------------------------------------------------------
! Number of branches     : 658.793
! Number of fails        : 300.629
! Total memory usage     : 8,9 MB (8,4 MB CP Optimizer + 0,5 MB Concert)
! Time spent in solve    : 60,02s (60,00s engine + 0,03s extraction)
! Search speed (br. / s) : 10.981,5
! ------------------------------------------------------------------------
```

Figure 4.1: CPLEX Engine Log results for the non-linear model, k = 2 and n = 10

The optimal solution was found, but optimality was not proven, as the best bound value is 1.899. This seems an ineffective approach, considering that it could not solve to optimality even this small size instance, with a relatively high time limit. Therefore, in the following section the objective function is linearized.

4.2. Linearization of the objective function

In this section, the linearization of the objective function for the two following alternatives is performed, named Model a) and Model b):

- Model a): u_i and v_j unrestricted in sign.
- Model b): u_i and $v_j \geq 0$ (this additional constraint will be further analyzed in section 4.4).

To do so, let us consider variables a_i, b_j $\forall$ i, j to be added, such that:

$$a_i = o_i * u_i \text{ and } b_j = d_j * v_j$$

This can be reached, as suggested at the end of section 11.2 in [4], by adding the following linear constraints, where L and U are the lower and upper bounds for a_i and b_j. Let us define U = maximum (c_{ij}) satisfying constraint (4.12), and L = -U:

Model a):

$$L \leq a_i \leq U \quad \forall \text{ i} \tag{4.17.a}$$

$$L * o_i \leq a_i \leq U * o_i \quad \forall \text{ i} \tag{4.17.b}$$

$$u_i - (1 - o_i) * U \leq a_i \leq u_i - (1 - o_i) * L \quad \forall \text{ i} \tag{4.17.c}$$

$$a_i \leq u_i + (1 - o_i) * U \quad \forall \text{ i} \tag{4.17.d}$$

Model b)

$$a_i \leq U * o_i \quad \forall \text{ i} \tag{4.17.a}$$

$$a_i \leq u_i \quad \forall \text{ i} \tag{4.17.b}$$

$$a_i \geq u_i - (1 - o_i) * U \quad \forall \text{ i} \tag{4.17.c}$$

$$a_i \geq 0 \quad \forall \text{ i} \tag{4.17.d}$$

In a similar way we define constraints (4.18.a) to (4.18.d) replacing a_i, o_i and u_i by b_j, d_j and v_j.

After adding the corresponding constraints (4.17) and (4.18), we can replace our objective function (4.9.a) by:

$$max \sum_{i=1}^{n} a_i + \sum_{j=1}^{n} b_j \tag{4.9.b}$$

In this way, we reached a Mixed Integer Linear Programming formulation. Considering the example solved with the non-linear model (instance 1, $k = 2$ and $n = 10$), we observe major improvements in terms of time:

Model a) Takes 0.64s to find and check the optimal solution.

Model b) Takes 0.36s to find and check the optimal solution.

4.2.1. Determining the optimal value for M

As explained before, the minimum value of M that ensures the satisfaction of constraint (4.12) is the maximum $c_{ij} = 99$ for the studied instances. M was set to 100, 10^3, 10^6 and 10^9 to analyze its impact in the performance of the model. The algorithm was run for instances of size 10 (in the next section a detailed explanation of the instances), for different values of k and for Models a), and b). Obtained results and times did not change for different values of M, so we can conclude that it is not a relevant parameter on the results and set M = 100.

4.3. Empirical comparison between Model a) and Model b)

In this section, the results of Model a) and Model b) are compared in terms of quality of the solution and the time the solver needs to find the optimal solution. The models were implemented on CPLEX using Python 3.7, and the open-source Python API called *docplex*.

This analysis was carried out for three input problem sizes, 10 instances for each size. Values of the instances were randomly generated following a uniform distribution of integers between 10 and 99. The values of n and k are determined as follows:

- n = 10: k from 2 to 9 (with step 1)
- n = 30: k from 3 to 27 with step 3
- n = 50: k from 5 to 45 with step 5

Instances of size 10 are evaluated, and after a refinement of the model, the bigger sizes are studied.

For n = 10, the time limit was set to 60 seconds. Both models always reached optimality within the time limit, so relative gaps were 0%. The solution values were the different in 5 out of 80 cases, in which Model a) outperformed Model b). With respect to the time required to reach optimality, Model b) substantially outperformed Model a), as we see in Figure 4.2 where for every k the values represent the average of the 10 instances.

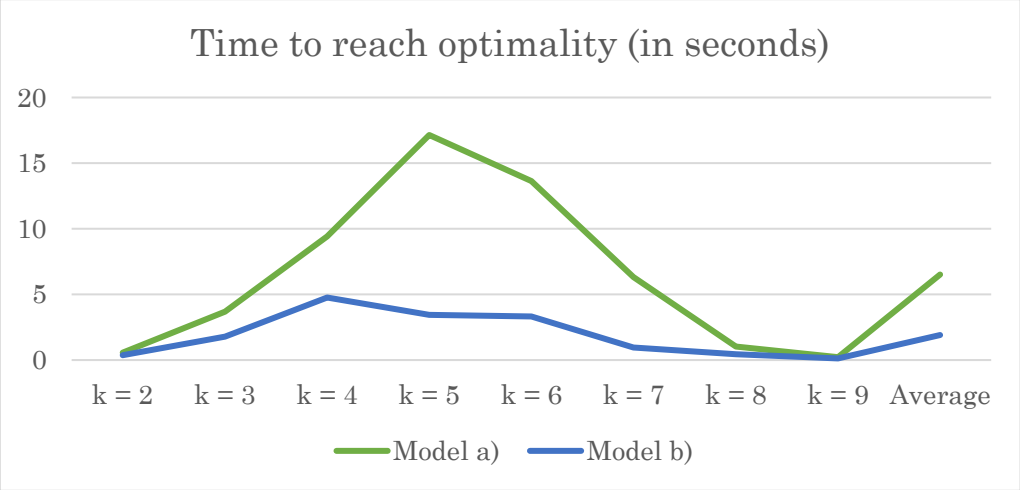

Figure 4.2: Time to reach optimality Models a) and b), n = 10

Although Model b) presents a higher performance in terms of time, as stated above, the reported optimal solution was worse than the one of Model a) in five cases. This could suggest that the assumption $u_i, v_j \geq 0$ that was added as a constraint in Model b), is not correct. Let us analyze it in detail.

4.4. Analysis of the assumption $u_i, v_j >= 0$

Results are not always the same for Model a) and Model b). In Table 4.1, in orange are marked the values for which this difference is present.

Instance	k = 2	k = 3	k = 4	k = 5	k = 6	k = 7	k = 8	k = 9
1	182	240	256	266	268	275	262	243
2	179	238	239	245	245	233	235	205
3	173	210	233	245	259	253	242	221
4	180	227	238	250	251	233	236	234
5	181	247	299	314	321	314	301	277
6	177	212	233	248	256	249	223	197
7	181	227	254	267	282	278	269	252
8	191	247	316	345	354	356	338	319
9	172	220	247	264	277	277	260	251
10	184	239	270	260	264	261	254	234

Table 4.1: Optimal solutions for n = 10

Analyzing these five cases, the leader's selections were identical for three of them (instances 1 and 10 for k = 9 and instance 6 for k = 8), meaning that the solution of the AP was wrong for Model b). On the other two cases, even the selected origins and destinations were different.

Let us first analyze why the AP solution of Model b) was wrong. Recalling the classical AP formulation:

$$min \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} x_{ij} \tag{4.19}$$

$$s.t. \sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, \dots, n \tag{4.20}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad j = 1, \dots, n \tag{4.21}$$

$$x_{ij} \geq 0 \qquad i, j = 1, \dots, n \tag{4.22}$$

And its dual:

$$min \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \tag{4.23}$$

$$s.t.\ u_i + v_j \leq c_{ij} \qquad \forall \text{ i, j} \tag{4.24}$$

Notice that adding constraint:

$$u_i, v_j \geq 0 \qquad \forall \text{ i, j} \tag{4.25}$$

In the dual formulation, is equivalent (from duality theory) to modifying the = sign in constraints (4.20) and (4.21) by a ≥ sign in the primal formulation. Model a) uses the assignment problem (dual) formulation (let us call it just AP), while Model b)

adds constraint (4.25) (let us call it AP AC, for additional constraint). Since the primal model has a minimization objective function, this change in constraints (4.20) and (4.21) represents a relaxation with respect to the original formulation, meaning that Model b) is not exact but approximate. Notice that more than one edge will be selected in some row or column only if it reduces the solution value, and the following inequality is always satisfied:

$$AP\ AC\ Opt.value \leq\ AP\ Opt.value \tag{4.26}$$

Initially, one can think that this change will never modify the optimum value of the AP (by intuition selecting more edges implies increasing the solution value). This is the most usual behavior between the two models but it is not always true, as previously showed in Table 4.1. The following example represents the costs matrix that produces the optimal solution of the leader for n = 10, k = 9, and instance number 1:

	0	1	2	3	4	5	6	7	8
0	53	72	84	86	88	82	97	75	45
1	26	26	67	95	31	65	16	38	58
2	95	70	93	91	51	99	28	53	32
3	67	94	53	10	49	13	29	27	38
4	68	56	42	33	81	17	23	58	45
5	23	65	12	99	58	20	95	49	15
6	24	37	72	43	21	77	16	55	61
7	24	78	39	82	60	32	18	52	47
8	60	87	65	49	94	33	94	72	59

	0	1	2	3	4	5	6	7	8
0	53	72	84	86	88	82	97	75	45
1	26	26	67	95	31	65	16	38	58
2	95	70	93	91	51	99	28	53	32
3	67	94	53	10	49	13	29	27	38
4	68	56	42	33	81	17	23	58	45
5	23	65	12	99	58	20	95	49	15
6	24	37	72	43	21	77	16	55	61
7	24	78	39	82	60	32	18	52	47
8	60	87	65	49	94	33	94	72	59

Table 4.2: a) The left-hand side matrix; b) The right-hand side matrix

Both costs matrixes contain the same values, but in Table 4.2 a), in orange are marked the optimal edges for the AP AC, while in Table 4.2 b) we see in green the optimal selection for the AP. The optimum cost of AP is 247, while for the AP AC it is 243. We notice that in the second case, there are 10 selected nodes instead of 9, which actually leads to a decrease in the objective function by taking advantage of the relaxed constraint.

Moreover, from the BAP formulation, given a leader's selection we have:

$$AP\ Opt.value = \ Follower's\ reaction = Leader's\ OF\ value$$

Therefore, if we use the AP AC to calculate the solution for the BAP, we will have:

$$AP\ AC\ Opt.value \leq\ Leader's\ OF\ value \tag{4.27}$$

Which means that this modification leads to a lower bound for the actual solution of the BAP. Of course, using the AP AC to calculate the leader's solution may lead to non-feasible solutions for the original model, as shown in the previous example. However, we can use it to find the optimal selection of o_i and d_j, and then calculate the optimal solution of the $AP\ Opt.value$, being sure that it will always be greater than or equal to the one obtained by the AP AC, by (4.27). This is summarized in

Relaxed Model Algorithm, which will be used to refine the Model b) optimal solutions after they are found.

Relaxed Model Algorithm

Step 1. Solve the BAP by using Model b) as described in section 4.2

Step 2. Using the selected origins and destinations, create the optimal matrix P.

Step 3. Solve the AP of P and return the optimum value.

For simplicity, and as no change is directly applied to Model b), but we only add extra steps to possibly correct its solutions, let us use the same name Model b) while referring to the Relaxed Model Algorithm.

In this way, we can take advantage of the fact that the relaxed model runs faster. For $n=10$, it reduced the average required time to find the optimal solution for all instances and k_s to less than one third (from 6.50 to 1.90 seconds). As a drawback, the optimal selection for the leader is not found in 2 out of 80 cases (the other 3 "problematic" examples are corrected by Steps 2 and 3 of the Relaxed Model Algorithm). This happens because the AP AC solution for the optimal selection gives a value that is lower than the one conceived as optimal by the relaxed model. To illustrate this particular case let us analyze the BAP for $k=9$ for instance 5. Its optimal solution is 285, but if we solve the AP AC for the selected origins and destinations, this value is reduced to 272, and then, from the relaxed model point of view it is better to select different origins and destinations that lead to a value of 277.

Going on with the analysis, being faster in smaller instances implies finding better results for bigger ones. Let us check if this is the case by comparing the performances of both models for the 10 instances of size 30 using a time limit of 60 seconds. First of all, the solver always reached the time limit without reporting optimality (relative gaps always > 0%). On the other hand, globally speaking, the total average of the 90 results of Model a) was 415.0, 2.5% lower than the one reached by Model b) that was 425.6. Additionally, a more detailed analysis by comparing one by one the results reached by both models is presented in Figure 4.3, and complementarily relative gaps are presented in Figure 4.4.

We can observe that the global win-lose-draw rate is clearly on favor of Model b), and this pattern is repeated for every k. Additionally, relative gaps of Model b) were tighter than Model a).

While analyzing the instances of size 50, a similar behavior between both models exists: the win-lose-rate of Model b) against Model a) is 81%-14%-5%, and also this pattern repeats for every k. We can therefore conclude that Model b) outperforms Model a).

It is worth to mention that while running Model b), the solution of the AP for the selected matrix (obtained on Steps 3) was always the same as the result obtained by the AP AC (Step 1), meaning that the relaxation produced by constraint (4.25), was actually not affecting the calculation of the optimal solution, while it did lead to improved performances.

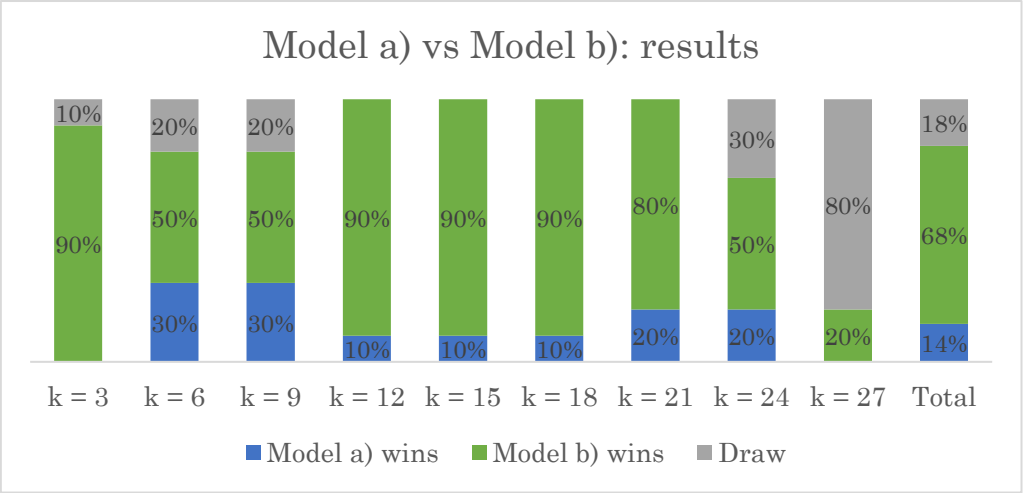

Figure 4.3: Comparison of Models a) and b): results

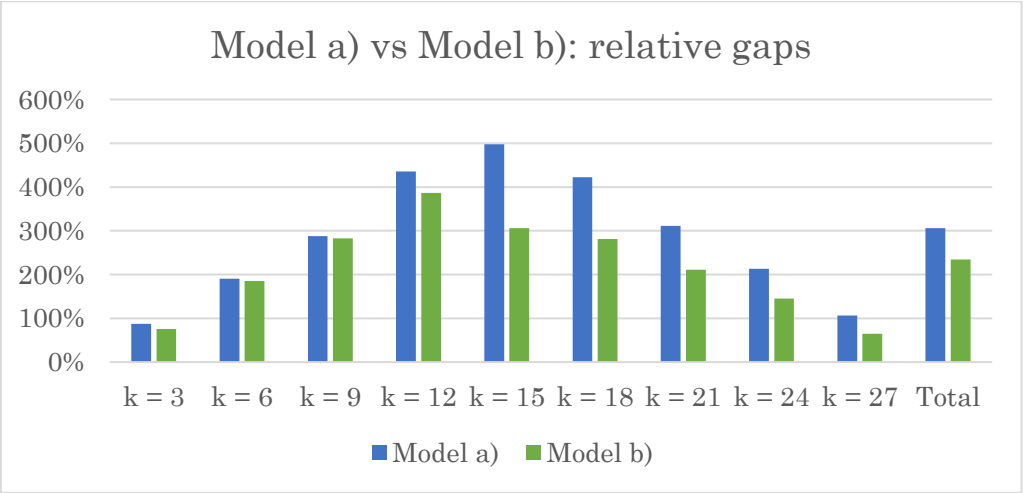

Figure 4.4: Comparison of Models a) and b): relative gaps

4.5. An alternative formulation only with binary variables

An alternative model can be considered having only binary variables of the form u_{hi} and v_{lj} that replace the continuous variables u_i and v_j, avoiding the non-linear objective function of the dual model presented in section 4.1.

By making an appropriate use of the constraint $u_i, v_j \geq 0$, such as showed on the previous section, we can define h binary variables u_{hi} for every u_i, where h is an integer that can vary between 0 (in the case that the node is not selected) and h_{imax}. Analogously we define l binary variables v_{lj} for every v_j, where l is an integer varying from 0 to l_{jmax}. Notice that h_{imax} and l_{jmax} represent the maximum possible value that u_i and v_j can take on the previous model, which depends on constraint (4.12)

that is applied only to the k selected origins and destinations. Variables $u_{0i} = 1$ and $v_{0j} = 1$ imply that the origin i and the destination j is not selected. Then we have:

$$h_{imax} = Min(c_{ij} \,\forall \text{ j: } v_{0j} = 0) \forall \text{ i: } u_{0i} = 0 \text{ and } l_{jmax} = Min(c_{ij} \,\forall \text{ i: } u_{0i} = 0) \forall \text{ j: } v_{0j} = 0$$

However, we cannot use these formulas to define the values of $h_{imax} \,\forall$ i and $l_{jmax} \,\forall$ j (that are actually required to complete the model formulation), because we do not know beforehand the selected i_s and j_s and then we do not know which v_{0j} and u_{0i} are 0. Hence, let us define them as the maximum of each specific row and column, meaning that, every h_{imax} and l_{jmax} will be different based on the edges of origin i and destination j. In this way, they can be used as input constants in the model, and at the same time we do not set them to be unnecessarily large (e.g. $Max(c_{ij})$), which would increase the number of variables:

$$h_{imax} = Max(c_{ij} \,\forall \text{ j}) \,\forall \text{ i} \text{ and } l_{jmax} = Max(c_{ij} \,\forall \text{ i}) \,\forall \text{j}$$

Having defined the values for h_{imax} and l_{jmax}, we can formulate the BAP as follows:

$$max \sum_{i=1}^{n} \sum_{h=1}^{h_{imax}} hu_{hi} + \sum_{j=1}^{n} \sum_{l=1}^{l_{jmax}} lv_{lj} \tag{4.28}$$

$$s.t. \sum_{h=0}^{h_{imax}} u_{hi} = 1 \;\forall \text{ i} \tag{4.29}$$

$$\sum_{l=0}^{l_{jmax}} v_{lj} = 1 \quad \forall \text{ j} \tag{4.30}$$

$$\sum_{h=1}^{h_{imax}} hu_{hi} + \sum_{l=1}^{l_{jmax}} lv_{lj} \leq c_{ij} + M\left(u_{0i} + v_{0j}\right) \forall \text{ i, j} \tag{4.31}$$

$$\sum_{i=1}^{n} \sum_{h=1}^{h_{imax}} u_{hi} = k \tag{4.32}$$

$$\sum_{j=1}^{n} \sum_{l=1}^{l_{jmax}} v_{lj} = k \tag{4.33}$$

$$u_{hi}, v_{lj} \in \{0,1\} \quad \forall h, i \;\; \forall l, j \tag{4.34}$$

Observe that we avoid two big-M constraints. Moreover, constraints (4.29) and (4.30) state that u_i and v_j can take only one value between 0 and h_{imax} and l_{jmax}, correspondingly. Finally, constraints (4.31), (4.32) and (4.33) are equivalent to (4.12), (4.13) and (4.14) but formulated with binary variables.

As it was previously mentioned, this model does not present neither a non-linear objective function nor non-linear constraints, so it can be implemented without any intermediate procedure. With the aim of the comparison, let us name this formulation Model c), to be compared with Model b) of the previous section, since this last outperforms Model a).

For n = 10, both models were tested for the same 10 instances and k_s as before, and a time limit of 60 seconds. First of all, the results achieved were the same, with only

1 exception in which Model b) outperforms Model c). On the other side, with respect to the time required to reach optimality, we see that Model b) considerably outperforms Model c), as we can see in Figure 4.5.

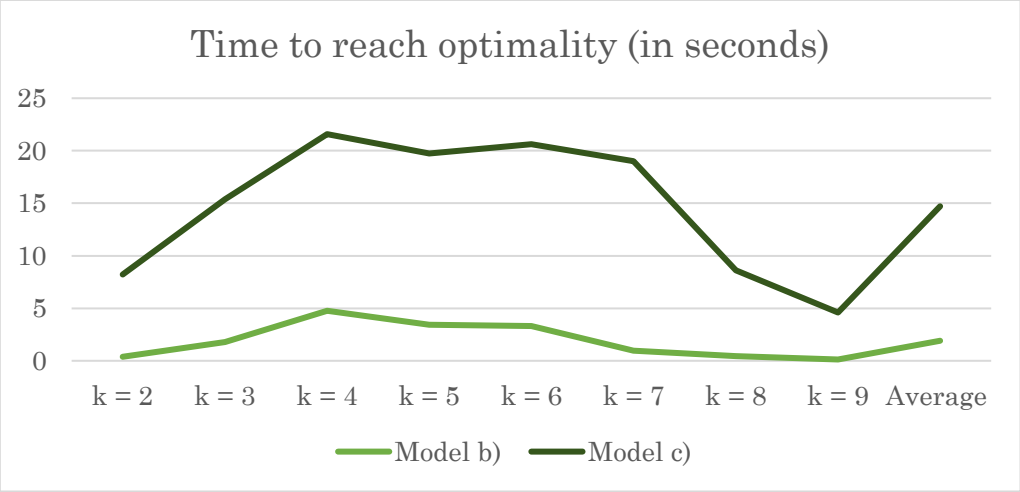

Figure 4.5: Time to reach optimality Models b) and c), n = 10

For n = 30, the time limit of 60 seconds was always reached. Comparing one by one the results of both models, just as done in section 4.4, we observe that Model b) strongly outperforms Model c), as shown in Figure 4.6. The author considers that no more analyses on the matter need to be done before concluding that Model b) is better than c).

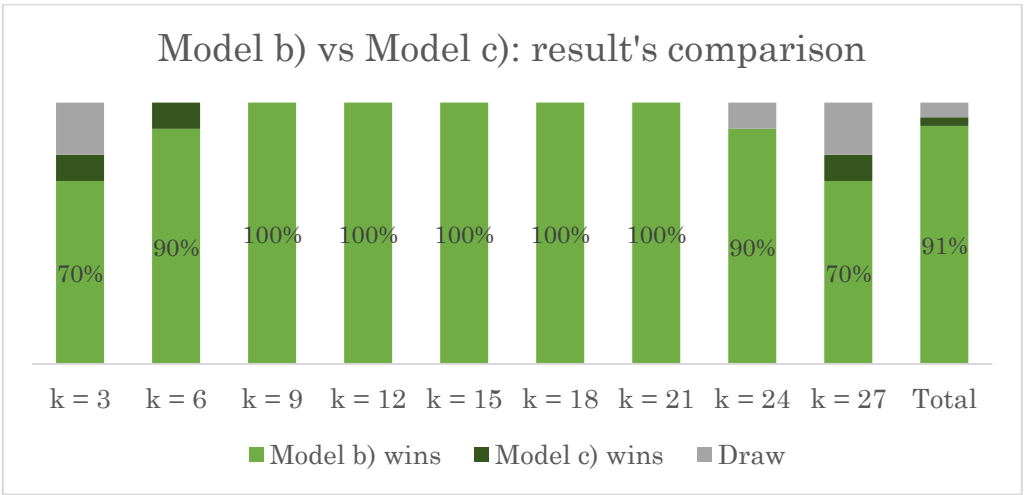

Figure 4.6: Comparison of the results for n = 30 between Models b) and c)

4.6. A final improvement of the model formulation

In this section, an improvement on the previous formulation is proposed. Recalling the dual formulation of the BAP, our objective function was:

$$max \sum_{i=1}^{n} o_i u_i + \sum_{j=1}^{n} d_j v_j \qquad (4.9.a)$$

The improvement is to exclude variables o_i and d_j from (4.9.a). Let us explain why this change is correct. Let us consider variable o_1. If it is 0, then original the term $o_1 u_1$ of the objective function will be 0 while the new term will take the value of u_1. However, as u_1 is affected by constraint (4.10):

$$u_i \leq o_i M \quad \forall \text{ i} \quad \Rightarrow \quad u_1 \leq o_1 M \quad \Rightarrow \quad u_1 \leq 0$$

Then $u_1$cannot add value to the solution, but only be null or negative. Since the objective function wants to maximize the result, the solver will set $u_1$to 0. Moreover, if $o_1 = 1$, then $o_1 u_1 = u_1$. In this way, we can conclude that the exclusion of o_1 will have no effect while computing the objective function but will simplify the model.

As the example was developed for a generic o_i, this logic can be extended to the rest of the o_{is}. Similarly, by considering constraint (4.11), the above explanation is valid for every d_j, and we can conclude that the modification is correct. Hence, we can replace our objective function (4.9.a) by:

$$max \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \qquad (4.9.c)$$

This replacement of (9.a) by (9.c) allow us to exclude constraints (4.17) and (4.18) because (4.9.c) is already linear. Let us call this new formulation Model d). Finally, let us add constraint (4.25) and follow the Relaxed Model Algorithm for the implementation (of course, replacing Model b) by Model d) in Step 1), since this relaxation showed that it improves the performances of the exact models.

Let us compare Model d) with Model b). Experimental results suggest that the performance of both models is similar for small size instances. As an example, Figure 4.7 presents the output given by the solver (in the python environment) while running both models, for n = 10, instance 1 and k = 2. Notice that the selected o_{is}, d_{js}, and the objective function value is the same:

```
Original model linearized (Model b)
solution for: BAP
objective: 182.000
o_0 = 1
o_9 = 1
d_5 = 1
d_7 = 1
u_0 = 3.000
v_5 = 85.000
v_7 = 94.000
a_0 = 3.000
b_5 = 85.000
b_7 = 94.000
None
Solution:  182
Optimal
Time  0.3
Relative gap =  0.0
Iterations 4119
```

```
New improved model (Model d)
solution for: BAP
objective: 182.000
o_0 = 1
o_9 = 1
d_5 = 1
d_7 = 1
u_0 = 88.000
u_9 = 85.000
v_7 = 9.000
None
Solution:  182
Optimal
Time  0.28
Relative gap =  0.0
Iterations 3611
```

Figure 4.7: Output of CPLEX run in python. Instance 1, n = 10, k = 2. Models b) and d)

We do notice some differences in u_i and v_j, but they do not affect the optimal selection and value. Moreover, the time required to find the optimal solution and the number of iterations were similar, with slightly less iterations and time achieved by the improved model.

After testing the models with the instances of size 10 and the corresponding k_s, both, of course, always reached the same results. With respect to the time required to reach optimality, five runs for each model were done. They demonstrated to be equally fast, as the total average to find an optimal solution was 1,830 seconds for Model b) and 1,825 for Model d).

For n = 30, Figure 4.8 presents the win-lose-draw rate obtained by comparing the results one by one, for the corresponding instances and k_s defined in section 4.3, within a time limit of 60 seconds. Since the performances are very similar, an identical analysis but for instances of size 50 is presented in Figure 4.9. Again, we do not notice huge differences between the performance of both models, but we can affirm that Model d) is slightly better than Model b).

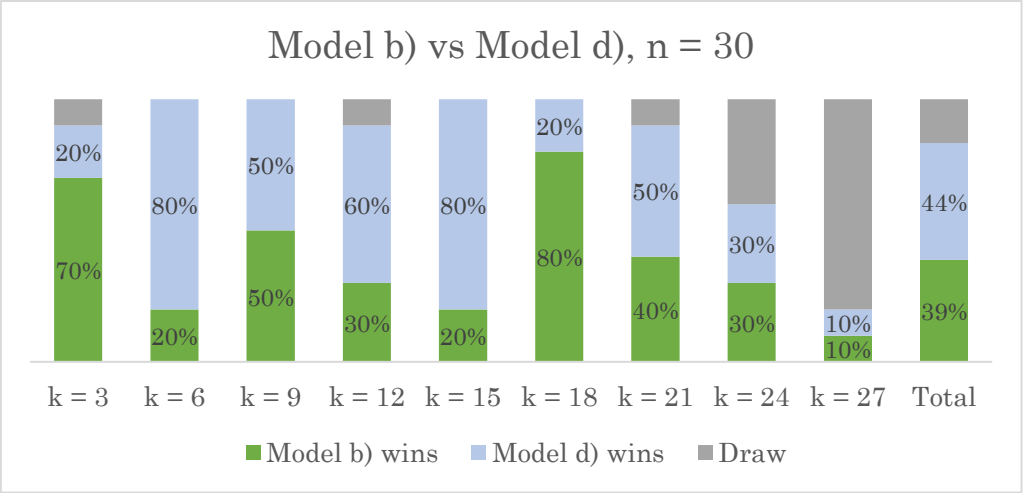

Figure 4.8: Comparison of the results for n = 30 between Models b) and d)

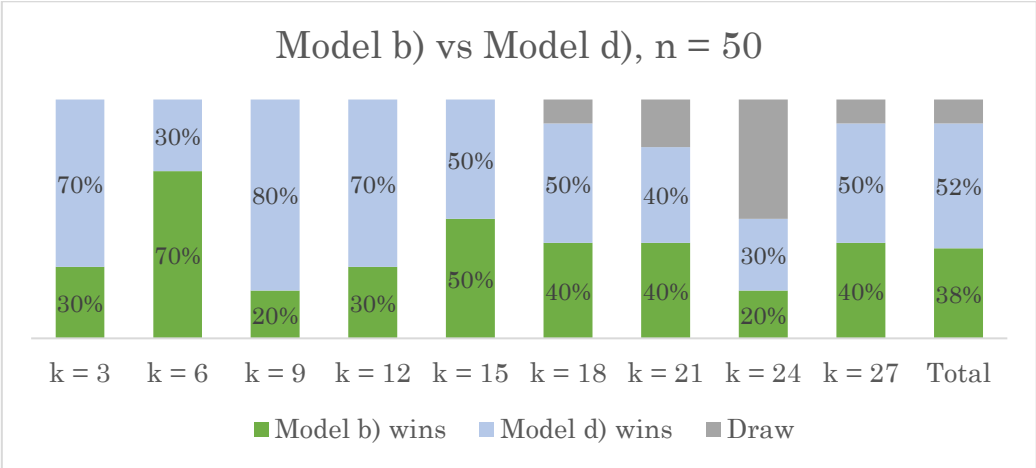

Figure 4.9: Comparison of the results for n = 50 between Models b) and d)

As a conclusion of this chapter, four models were implemented, and we can establish the following preference relationships between them:

- Model b) ≻ Model a).
- Model b) ≻ Model c).
- Model d) ≽ Model b).

Meaning that the best model to solve the BAP by means of a MILP solver is Model d).

5. A heuristic approach

5.1. Introduction

A heuristic is a widely used technique for problem solving that consists of a set of rules whose aim is to find good solutions for the given problem at a reasonable time (i.e., computational cost). It is approximate since it does not guarantee optimality but provides good solutions for little effort [20]. Constructive algorithms, local search, and improved heuristics are some of the main heuristic techniques.

A *greedy algorithm* is a constructive procedure that starts at a candidate set (that may be an empty set), from which the solution is iteratively built up by fixing the value of some variables one at a time. At each step, the decision is taken by considering only the information at hand, ignoring the effect that it may have in the future. The algorithm stops when a complete and feasible solution is finally constructed. These types of algorithms are easy to invent and implement, and in some specific problems they return the optimal solution. For instance, given an edge-weighted, undirected, and connected graph G = (V, E) [4] the *Kruskal's algorithm* and the *Prim's algorithm* are greedy algorithms that construct the minimum spanning tree[5] [21]. Moreover, given a collection of production items and two machines where: each item must pass first for machine one and then for machine two, each machine can handle only one item at a time and each item is associated with two arbitrary positive numbers that represent the set up plus the work time for that item to pass through each machine, the *Johnson's rule* leads to the optimal machine's scheduling that minimizes the total elapsed time [22]. However, there are many problems that cannot be solved to global optimality by the greedy approach [23].

Another well-known heuristic approach is *local search* (LS), which is explained in detail in section 5.3. Additionally, there exist several meta-heuristics based on local search, for instance *Iterated Local Search* (ILS), *Variable Neighborhood Search* (VLS) and *Tabu Search*. The first one consists of iteratively applying intensification (*local search*) and perturbation (modification of the current best solution) phases and will be explained in detail in section 5.5. The second one, was proposed in 1997 by Mladenović and Hansen [24], and its basic idea is to systematically change the neighborhood (generally by increasing its size) both within a descent phase to find a local optimum and in a perturbation phase to get out of the corresponding valley. It is based on the fact that a local optimum with respect to one neighborhood structure is not necessarily so for another. Finally, a *Tabu Search* heuristic allows some moves that do not improve the solution after a local optimum is reached, but then forbids to re-do these moves with the aim of avoiding the initial local optimum. We mention that there exist many other meta-heuristic techniques, such as Simulated Annealing, Genetic Algorithms, Ant Colony Optimization, Greedy Randomized

[4] A graph is undirected if the two endpoints if its edges are not distinguished form each other, and it is connected if there is a path (a sequence of consecutive edges) connecting every pair of vertices [30].

[5] A spanning tree consists of a subset $T \subseteq E$ of edges, where T does not contain cycles and for every pair $v, w \in V$ of vertices, T should include a path between v and w.

Adaptive Search Procedures, etc. Please refer to [25] for more information about these methods.

Finally, a matheuristic is an improved heuristic that brings together meta-heuristics and mathematical programming (exact approaches). In section 6.1, the reader can find an extensive explanation of this technique.

5.2. Greedy algorithms to solve the BAP

As it is known in literature [26], a *local search* can be enhanced by the use of a *greedy algorithm* to produce the initial feasible solution. This generally leads to overall better local optima with less computational cost than starting by a randomly generated solution. For this reason, several greedy algorithms are developed and tested in this section with the aim of generating the best possible initial solution. Please remember that the terms row and origin are equivalent, and so the terms column and destination.

By observing the essence of the BAP, we notice that the leader will try to select the origins and destinations in such a way that the corresponding costs matrix of size k x k are as high as possible. A matrix with mainly small values will lead to a low-value solution for the AP, while one that contains mainly high values will tend to produce the opposite. However, this "rule of thumb" may be misleading. Let us consider the following two examples of leader's selections:

$$\begin{bmatrix} 90 & 84 & \mathbf{21} \\ \mathbf{40} & 59 & 76 \\ 68 & \mathbf{15} & 99 \end{bmatrix} \qquad \begin{bmatrix} \mathbf{28} & 25 & 38 \\ 54 & 40 & \mathbf{37} \\ 20 & \mathbf{13} & 11 \end{bmatrix}$$

Matrix 1. Solution value: 76 *Matrix 2. Solution value: 78*

We may think that Matrix 1 is better than Matrix 2 from the leader's perspective, because it contains in general higher values, but actually it is the other way around (in bold the optimum selection of the follower). Said that, we can state that while constructing a greedy algorithm it is not enough to achieve as much high values as possible. Instead, it makes more sense to avoid the lowest costs.

Considering these preliminaries, five different greedy algorithms are presented. *Greedy Algorithm 1 (Greedy 1)* is based on the fact that the follower will generally select the edge of minimum cost for each given origin and destination. In this way, the leader deletes those values, so that the follower cannot choose them:

Greedy Algorithm 1

Step 1. **Find** the minimum value for each origin and destination.

Step 2. **Delete** the *(n-k)* rows and columns that have the lowest minimum values.

The time complexity of *Greedy 1* is given by $O(n * 2n) = O(n^2)$, where $O(n)$ is the complexity of finding the minimum value over a list of n elements, and $O(2n)$ represents the number of times that we find the minimum.

We notice that in this first attempt, we start from the n x n matrix and in one step we reduce it to a size of k x k. To further improve the results, the next greedy algorithms are iterative and delete only one row or one column at each step, by considering the average value α of the x lowest elements of each row and column and deleting the array that corresponds to the minimum α. Let us define:

$$\alpha = Average(x\ lowest\ costs)$$

What varies between *Greedy Algorithms 2* to *5* is how x is calculated, so we first present the common steps of these algorithms.

Greedy Algorithms 2 to 5 (generic steps)

Step 1. **Calculate** α for each row and column.

Step 2. **Delete** the row or column with the lowest α. If there are k remaining rows/columns, delete the column/row with the lowest α.

Step 3. If the number of remaining rows and columns is equal to k, **stop.** Otherwise, go to Step 1.

On *Greedy Algorithm 2 (Greedy 2),* we set $x = k$, taking into consideration that the follower will be given only k values per row and column. On *Greedy Algorithms 3 to 5*, the value of x is variable depending on the number of remaining rows and columns.

For instance, on *Greedy Algorithm 3 (Greedy 3)*, all the remaining elements of each row and column are considered at each step. We define two values of x, one for rows and other for columns and α is calculated by considering the corresponding x:

$$x_{row} = len(row);\ x_{col} = len(col)$$

Where the function len(a) returns the number of elements of the list a.

Moreover, *Greedy Algorithm 4 (Greedy 4)* considers the minimum *(n-k)* costs to calculate α, leaving the rows and columns with the maximum mean for the k selected origins and destinations:

$$x_{row} = min\big((n-k), len(row)\big);\ x_{col} = min\ ((n-k), len(col))$$

Finally, *Greedy Algorithm 5 (Greedy 5)* calculates x considering half of the values of each array. As the number of rows and columns decrease, we reduce x in such a way that α is not biased by a cost that the follower will probably not consider:

$$x_{row} = max\ (2, floor(\frac{len(row)}{2}));\ x_{col} = max\ (2, floor(\frac{len(column)}{2}))$$

Where the function max(a) returns the maximum value over all the elements of list a. The function floor(b) returns the immediate lower integer value of b (if b is integer, it just returns b).

Note that the complexity of *Greedy Algorithms 2, 3 ,4* and *5* is the same, as the main structure of the algorithm is similar, and the only difference is the formula to calculate x. Therefore, let us define their time complexity by using *Greedy Algorithm 5* as an example:

$$O(2 * (n-k) * 2n * (nlogn + n/2)) = O((n-k) * n^2 logn)$$

Where:

- $O(2(n-k))$: is the number of iterations, equal to the total deleted origins and destinations.
- $O(2n)$: is the number of times that we sort and calculate the average at each iteration.
- $O(nlogn)$: is the time complexity to sort an array.
- $O(n/2)$: is the time complexity to calculate the average for the $n/2$ lowest values.

We notice a higher time complexity compared to the one of *Greedy 1*. However, this difference will be found irrelevant in the overall complexity of the Local Search algorithm.

Now, let us compare the *Greedy Algorithms'* performances, based on the quality of their solutions. The same 10 instances for each size of 10, 30 and 50 developed in section 4.3 will be used, as well as the same values of k.

Results of the five *Greedy Algorithms*, a randomly generated solution, and a final alternative that consists of taking the maximum of *Greedy 4* and *Greedy 5* (named *Greedy 4_5*) are presented as the average value obtained for the 10 instances, for every k. Figures 5.1 and 5.2 present the results for the instances sizes of 10 and 30, while the graph for size 50 is not added because it is similar to these two figures.

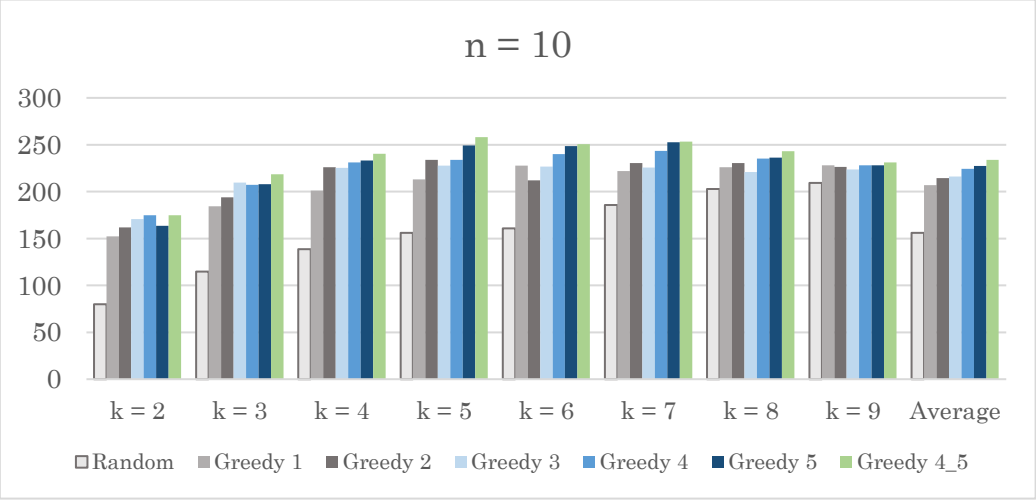

Figure 5.1: Results of the Greedy Algorithms for n = 10

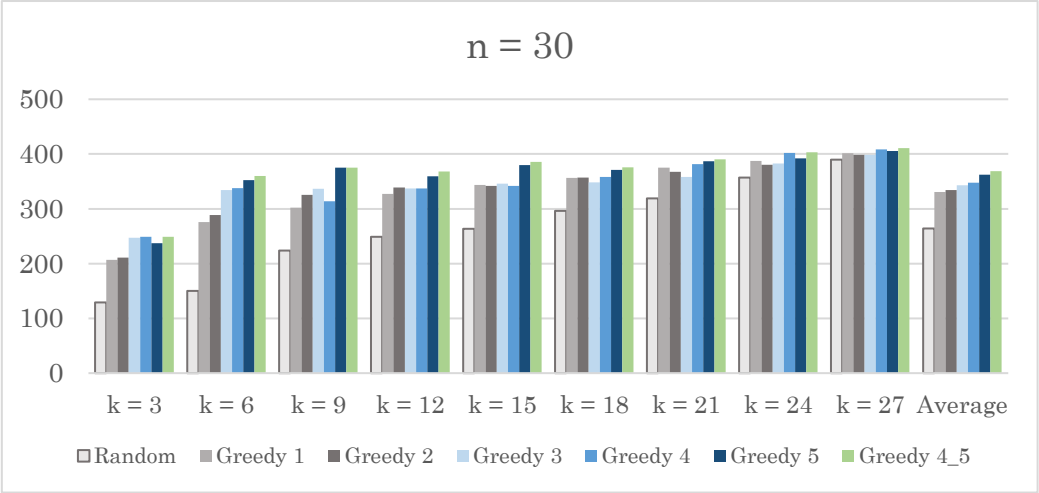

Figure 5.2: Results of the Greedy Algorithms for n = 30

We can conclude that the best results are individually obtained by *Greedy 5*, and they can be improved by using *Greedy 4_5*, without increasing the time complexity. If we compute the maximum of all the greedy algorithms the improvement is insignificant.

5.3. Local search: some theory

5.3.1. Neighborhood

Given a feasible point $f \in F$ of a particular problem, the *neighborhood* is defined as the set $N(f)$ of points that are "close" in some sense to the point f.

Definition [3]

Given an optimization problem with instances (F, c), a *neighborhood* is a mapping:

$$N: F \rightarrow 2^F$$

defined for each instance.[6] □

If $F = \mathbb{R}^n$, the set of points within a fixed Euclidean distance provides a natural neighborhood.

Different types of moves can be selected to generate a neighborhood. Some basic examples consist of a swap and/or insertion move. Let us consider a basic scheduling problem, for which we want to minimize the sum of the completion times, there is only one machine and 4 jobs with durations [2,5,4,6] for jobs 1 to 4 correspondingly. Although the optimal solution is almost trivial, let us illustrate the swap and insertion moves. Suppose that f = [2,5,4,6], with a total completion time of 37.

[6] For any set F, we denote by 2^F the set of all subsets of F (i.e., $2^F = \{F': F' \subseteq F\}$).

If we create the neighborhood by performing just one swap, the neighbors will be generated by swapping 2 with 5, 5 with 4, and 4 with 6, that is:

$$N_{simple\ swap} = \{[5,2,4,6],[2,4,5,6],[2,5,6,4]\}$$

With its corresponding costs of:

$$c_{simple\ swap} = \{40, 36, 39\}$$

Let us consider then an insertion-move to generate a neighborhood, with a distance of insertion of two. This means that, for example, the 1st element is inserted into the 3rd position, and jobs 2 and 3 go to the 1st and 2nd position correspondingly, and so on:

$$N_{insertion\ of\ 2} = \{[5,4,2,6],[2,4,6,5],[4,2,5,6],[2,6,5,4]\}$$

$$c_{insertion\ of\ 2} = \{42,37,38,40\}$$

5.3.2. Local and global optima

For some problems, it is extremely difficult to find the best global solution, but it is possible to find the so-called local optima, that may not be very distant from the global optimum.

Definition [3]

Given an instance (F, c) of an optimization problem and a neighborhood N, a feasible solution $f \in F$ is called *locally optimal with respect to N,* or simply *locally optimal* whenever N is understood by context if:

$$c(f) \leq c(g)\ for\ all\ g \in N(f) \quad \square$$

Meaning that the term "local" depends on the definition of the neighborhood and it means that for that neighborhood the solution cannot be further improved.

5.3.3. Definition and general algorithm

Local search is a heuristic method able to produce locally optimal solutions. It is based on a trial-and-error approach, and its algorithm can be described as follows. Given an instance (F, c) of an optimization problem, where F is the feasible set and c is the cost function, we start at an initial solution $n_0 \in F$, and cost c_0, which is in turn defined as the current best solution s^*. Then, we select a neighborhood rule N and generate the neighborhood $N(s^*)$ containing p neighbors $n_1 \ldots, n_p$. We start to evaluate n_1 considering an *acceptance criterion*. The simplest and most widely used acceptance test is based on whether the neighbor improves or not upon the current best solution. In the case of a minimization problem, this means that n_1 is accepted if $c(n_1) < c(s^*)$. Otherwise, we continue to evaluate one by one $n_2 \ldots, n_p$. If we apply the *First Improvement Strategy* to perform the search into the neighborhood, as soon as an improvement is found in n_i, we stop the evaluation phase and set $s^* = n_i$. If

we choose to work with the *Steepest Descent Strategy*, the evaluation phase does not stop until the p neighbors are evaluated and the best of them is selected. The next iteration generates the neighborhood $N(n_i)$ and we repeat this procedure until a s*topping test* is satisfied. A time limit or the lack of neighbors that improve the current best solution after evaluating all the neighborhood (point in which we can say that a local optimum s^* was reached) are commonly used stopping tests.

We can summarize this procedure, considering the *First Improvement Strategy*, as follows:

Step 1. Generate an initial solution n_0, and set $s^* = n_0$.

Step 2. Generate the neighborhood $N(s^*) = n_1 \dots, n_p$.

Step 3. Evaluate $n_1 \dots, n_p$ according to the *acceptance criterion*. Whenever n_i is accepted, stop, and set $s^* = n_i$.

Step 4. *Stopping test*: if no neighbor is accepted, s^* is a Local Optimum: stop. If there is a time limit and it has been reached, stop (output: current s^*). Otherwise go to Step 2.

5.4. A local search algorithm to solve the BAP

In this section, a Local Search algorithm is designed and implemented. First of all, we need to produce an initial solution n_0. In order to improve the local search performance, we will not start from a random solution, but instead from the solution obtained by the best greedy algorithm: *Greedy 4_5*, explained in section 5.2.

The next step is to determine the neighborhood. Let us define a neighborhood that consists of swapping every selected origin by every non-selected origin; and swapping every selected destination by every non-selected destination, always performing just one move at a time. This produces a number of neighbors generated at each iteration:

$$Number\ of\ neigbors\ of\ N(n_0) = 2 * k * (n - k) = 2 * (nk - k^2)$$

Let us consider a toy example, with n = 4 and k = 2 to illustrate the neighborhood. On the current solution n_0, let us suppose that the selected [[origins],[destinations]]= [[1,3], [2.3]]. Then the neighborhood of $n_0$will be:

$$N(n_0) = \{ [[2,3],[2,3]], [[3,4],[2,3]], [[1,2],[2,3]], [[1,4],[2,3]] \dots$$

$$N(n_0) = \dots [[1,3],[1,3]], [[1,3],[3,4]], [[1,3],[1,2]], [[1,3],[2,4]]\}$$

Now, let us to define the acceptance criterion to be applied to determine the adequacy of every neighbor n_i: if there is an improvement with respect to the current best solution, then the neighbor is accepted, otherwise it is rejected. This means that we need to calculate the solution of the AP for every neighbor. As a consequence, the time complexity of the acceptance criterion is $O(k^3)$, where k is the follower's problem size. In this way, let us compute the complexity of the local search as:

$$O(It.* (2 * (nk - k^2) * k^3)) = O(It.* k^4 * (n - k))$$

Where It. is the total number of iterations required to find a local optimum, $2 * (nk - k^2)$ is the number of neighbors and k^3 is the time needed to evaluate each neighbor.

First, the *best improvement strategy* (see section 5.3.3) was considered. However, it was not efficient with respect to the total number of neighbors evaluated, which is equal to the number of Hungarian Methods applied. For this reason, other four alternatives were developed to reduce it. Let us explain the five neighbor's selection rules.

Neighborhood 1 (N1): apply the *best improvement strategy*.

Neighborhood 2 (N2): apply the *first improvement strategy*, but considering one improvement swapping rows, and one swapping columns. Take the best neighbor of those two.

Neighborhood 3 (N3): start analyzing the rows. If an improvement is found by swapping rows, replacing, let us say, row x by row y, complete the analysis of all the k neighbors generated by replacing the k selected rows by row y. Then, select the best neighbor and stop. Otherwise, apply the same procedure for the columns. If an improvement is found replacing a row, the next iteration starts scanning columns and vice versa.

Neighborhood 4 (N4): consider the nodes selected by the follower for the current best solution and sort them in a non-descending order. Start by swapping all the rows following the order of the sorted list. If no improvement is found, repeat the same procedure for all the columns. When an improvement is found by deleting for example row x, complete the replacement of all the n-k non-selected rows. If other improvements are found, take the best one, otherwise take the only one that improves the solution and stop.

Neighborhood 5 (N5): it is similar to *N4,* but the following difference is present. After the sorted list of the follower's nodes is generated, instead of scanning all the rows and then (if no improvement is found) all the columns, start by swapping the row or column that corresponds to the node having the lowest value. If no improvement is found, go for the second node, and so on. Analyze node by node and at each node swap first the row and then the column (this order first rows then columns is not fixed but depends on where the previous improvement was found). When an improvement is found, continue just as described in *N4.*

Let us summarize the *Local Search Algorithm* using *N5.*

Local Search Algorithm using N5

Step 1. *Initial solution generation.*

Generate an initial solution n_0 using *Greedy 4_5*. Set:

$s^* = n_0$

leader_selection = origins and destinations selected by the leader

follower_nodes = nodes selected by the follower sorted in a non-decreasing order by the value of each c_{ij}. It contains the i, j coordinates and the corresponding c_{ij}.

Step 2. *Generation and evaluation of the neighborhood* following $N5$

For *i* in *follower_nodes [i coordinates]*:

For new_row in non_selected_rows:[7]

new_selection = *leader_selection. Swap(row[i], new_row)*

new_selection_value = Hungarian_Method(new_selection)

if *new_selection_value* improves s^*:

accept the neighbor.

If one or more neighbors were accepted:

Select the best of them.

Set the new s^*, *leader_selection* and *follower_nodes* variables.

Go to Step 3.

Else:

Apply the previous 9 lines to the columns.

Step 3. *Stopping criterion*

If no improvement was found in Step 2 or time_limit reached:

Stop

Return s^*

Else:

Go to Step 2

The same instances as in previous sections are used to test the five different neighborhood rules. Additionally, results of the best greedy algorithm are presented, to show the effectiveness of LS on improving the greedy solution. Figures 5.3 and 5.4 present the results for $n = 10$ and $n = 30$, which values represent the average of the 10 instances, for every k.

As it was expected, LS is able to improve the solution given by the greedy algorithm, and the amount of improvement is bigger the closer k is to $n/2$. Comparing the performances of the different neighborhood rules, we notice insignificant differences in the solution values. For $n = 30$, the best total average result is obtained by N2: 417.6, while the worsts by N4 and N5 with 416.6. Moreover, the performances do not seem to be affected by k, since the results were similar for all k_s.

[7] Note that in this case we start with the rows, however, the algorithm starts with the rows only in the case that the previous improvement was found by swapping a column, and vice versa.

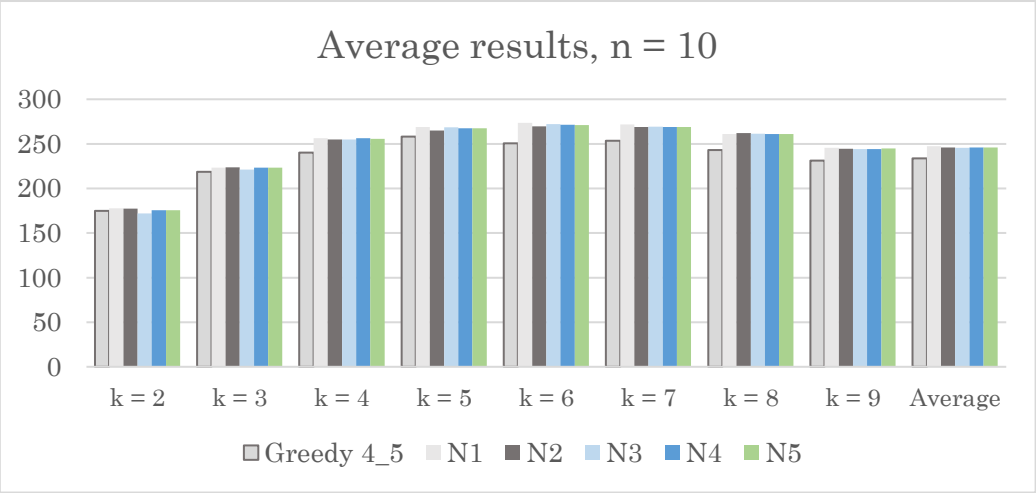

Figure 5.3: Results for the different Neighborhood rules, n = 10

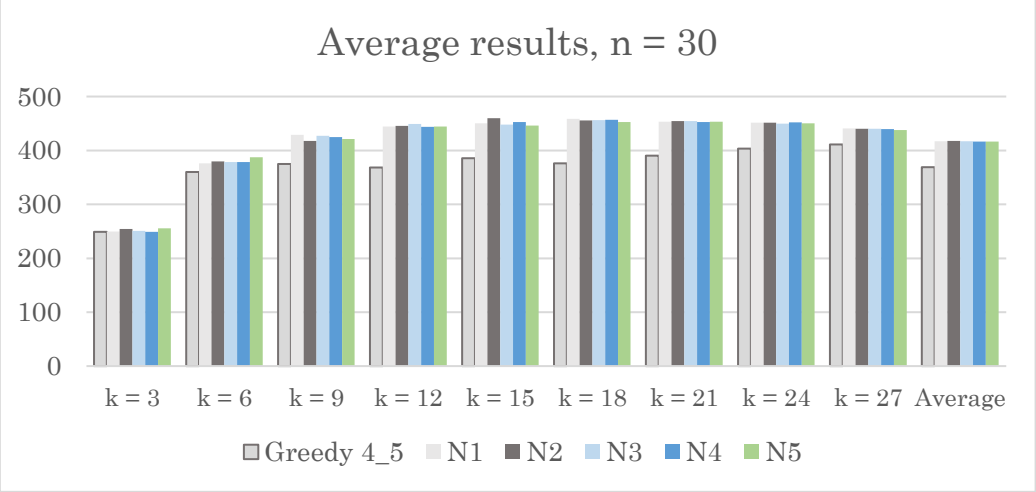

Figure 5.4: Results for the different Neighborhood rules, n = 30

Additionally, it is necessary to compare their performances with respect to a time metric. This can be done by considering the number of Hungarian Methods that every candidate algorithm applies before reaching a local optimum. Figures 5.5 and 5.6 present these results for n = 10 and n = 30. Again, the values represent the average for the 10 instances, for every k.

While comparing the time performances, we observe considerable differences. Roughly speaking, we observe that N4 and N5 need one third of the time required by N1 and N2, while N3 is in the middle. This shows that, considering the optimal selection (assignments) of the follower in such a way that the origins and destinations corresponding to the lowest costs are swapped first, allows to significant reductions of the computational time.

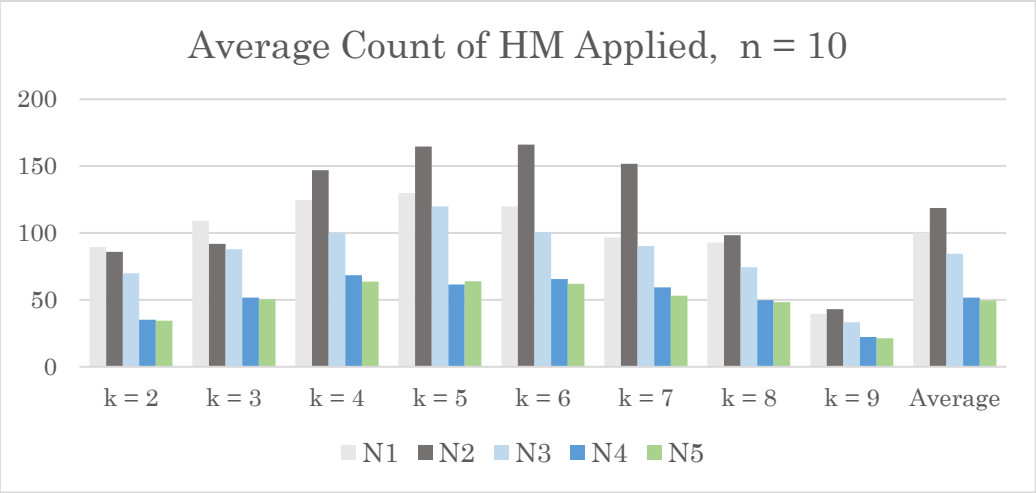

Figure 5.5: Number of Hungarian Method needed to reach local optimality, n = 10

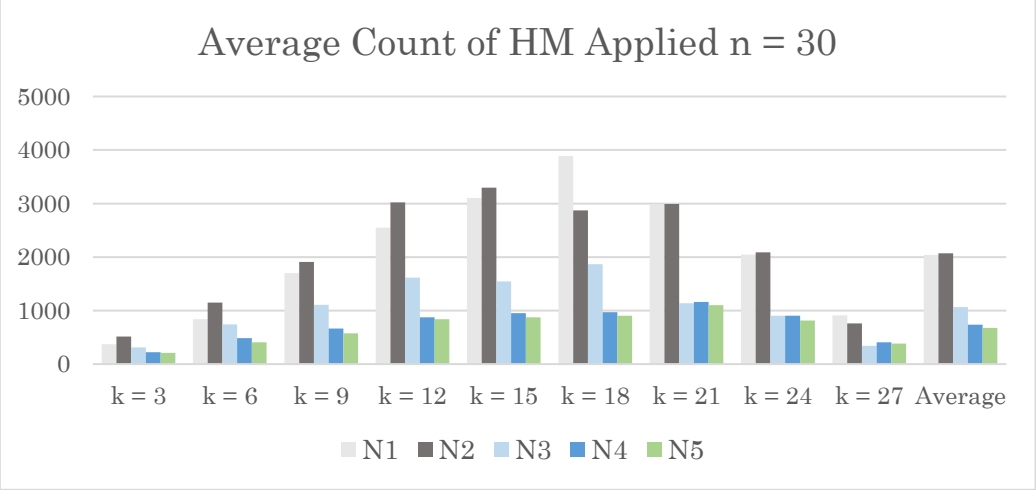

Figure 5.6: Number of Hungarian Method needed to reach local optimality, n = 30

Due to the fact that N5 is the fastest rule and that its quality of the solution is similar to the one of the other neighborhood rules, we can state that N5 is preferred over the other alternatives. Consequently, it will be used later in the Iterated Local Search section.

Let us add that, for n = 10, 14 out of 80 cases (17%) the global optimum was found by the *Greedy 4_5* and after local search is applied, this number increases to 49 (61%).

5.4.1. Effectiveness of the greedy algorithm in local search

As mentioned before, local search starts by a solution generated by a greedy algorithm. Here, the improvements reached by this non-random start are analyzed, for n = 30. First of all, the total average of the results of the 10 instances for all the

previously defined values of k, improves by 3% if we start local search by a greedy solution instead of by a random solution, as showed in Figure 5.7. Moreover, with respect to the average time (again, measured by comparing the number of Hungarian methods applied until local optimality is reached) was reduced by 23% by starting from the non-random solution.

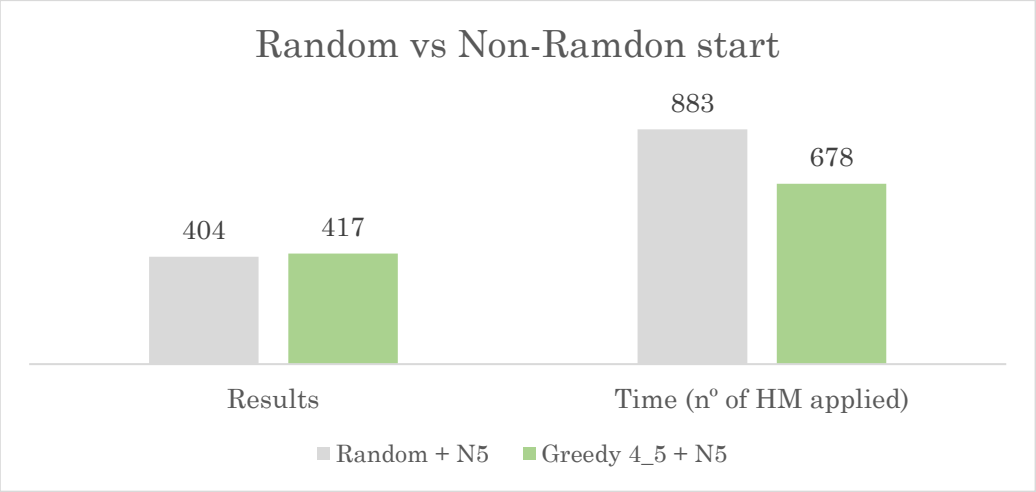

Figure 5.7: Random against non-random start Local Search, n = 30

Similar results were obtained for n = 10 with a solution improvement of 3.2% and a time reduction of 29%. In this way, we showed that the use of a greedy algorithm improves local search results and time.

5.5. Iterated local search: some theory

5.5.1. Introduction

One well known algorithm to improve the local search results is iterated local search (ILS). It is a conceptually simple metaheuristic but nevertheless has led to the state-of-the-art algorithms for many computationally hard problems, and the idea is to go beyond the first local optimum by applying perturbation and intensification (local search) cycles. It does not focus on all the solution space, but in the candidates returned by the local search algorithm. It builds a sequence of solutions that are better than the ones that would be created by a repeated random trial heuristic.

5.5.2. The general algorithm

The generic algorithm of ILS can be described as follows. We start by applying *local search* method. After the first local optimum s^* is found, we perform a *perturbation*

that leads to an intermediate solution s', with a worse solution value than s^*. A *perturbation* consists of a number of changes done in a solution, generally performed randomly, with the aim of achieving a good starting point to repeat the local search. Then, we apply the local search algorithm to this perturbed solution, reaching a new local optimum $s^{*\prime}$. If this new solution satisfies a certain *acceptance criterion*, then $s^* = s^{*\prime}$. Otherwise, we go back to the original s^* and apply the perturbation-local search process again. This is repeated until the *stopping test* is reached, generally a time limit. Figure 5.8, recalling [26] illustrates this sequence.

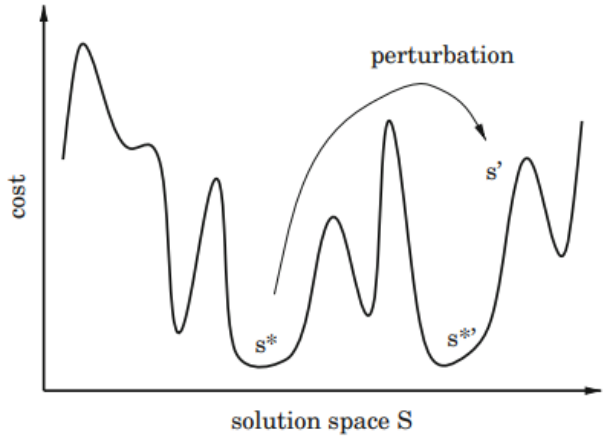

Figure 5.8: Graphic description of ILS applied to a minimization problem [26]

5.5.3. Enhancing iterated local search

While applying ILS, there are four modules that have to be considered: initial solution, local search, perturbation, and acceptance criterion. The overall performance of ILS will strongly depend on the success of each module, so one should try to optimize them to develop the best possible ILS. These modules are:

- **Initial solution generation**: it can be a random start or a constructive solution. It has been shown in section 5.4.1 that a constructive solution approach, after local search is applied, gives better results and in less time than a random start solution.
- **Local Search**: in section 5.4, a great effort was done to highly improve this module by taking advantage of the specific structure of the problem.
- **Perturbation**: this module is going to be explained and improved in the next section.
- **Acceptance criterion**: an improvement in the solution is generally applied, and in this case this criterion will be used as well.

5.6. Iterated local search to solve the BAP

In this section, ILS is implemented and optimized, and an analysis of how it improves LS is presented. The first step consists of applying the *Local Search Algorithm* using N5, obtaining the first local optimum s^*. Secondly, a *perturbation* is done to s^*: a number μ of selected origins and destinations is randomly swapped with the non-selected ones (chosen randomly as well). Consider that μ should be sufficiently high not to be trapped in the same local optimum, but also low enough not to fall into a random start approach. Therefore, we cannot define μ as a constant nor for instances of different size, neither for distinct values of k given a fixed n.

Let us define three different formulas to calculate μ depending on n and k, namely μ_1, μ_2 and μ_3, each one representing a different level of perturbation. The smallest perturbation is produced by μ_1, and it gradually increases for μ_2 and μ_3. In Figure 5.9 are presented, as an example, the values for μ, for n = 30, while their corresponding formulas, that allow to compute μ for every n and k are the following.

$$\mu_1 = Max\left(1; floor\left(\frac{min(k; n-k)}{6}\right)\right)$$

$$\mu_2 = ceil\left(\frac{min(k; n-k)}{4}\right)$$

$$\mu_3 = ceil\left(\frac{min(k; n-k)}{2}\right)$$

The function ceil(b) does the exact opposite of floor(b) and returns the immediate higher integer value of b (if b is integer, it just returns b). We can observe that μ_1 perturbs only a small portion of s^*, while μ_2 and μ_3 around 25% and 50% respectively.

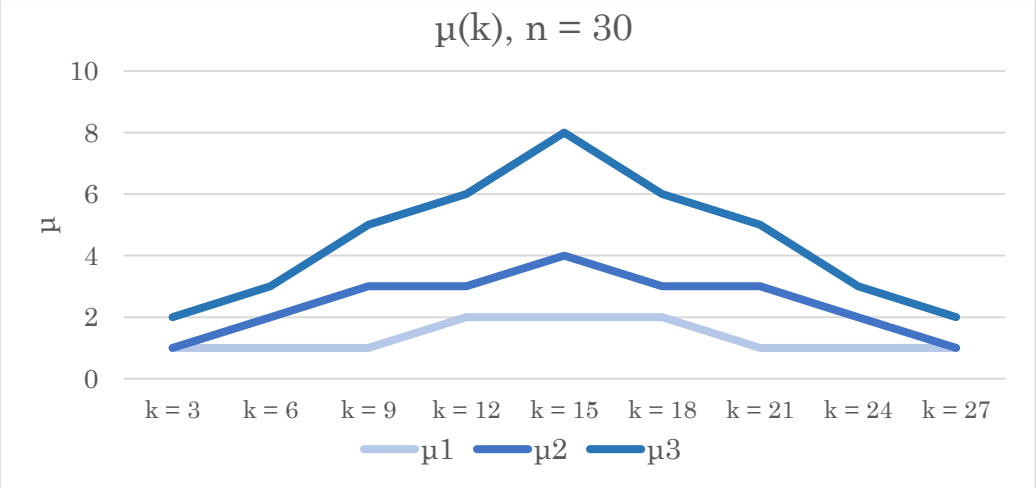

Figure 5.9: Graphical illustration of μ_1, μ_2 and μ_3 as a function of k, for n = 30

To evaluate the different μ_s, the instances of size 10 are not suitable, because as showed at the end of section 5.4, after applying LS, the results achieved are all optimal or near optimal. Hence, the analysis is carried out for n = 30, and a time limit of 60 seconds. Average results for the 10 instances and the corresponding

values of k (the same as used before) are presented at the left-hand side of Figure 5.10.

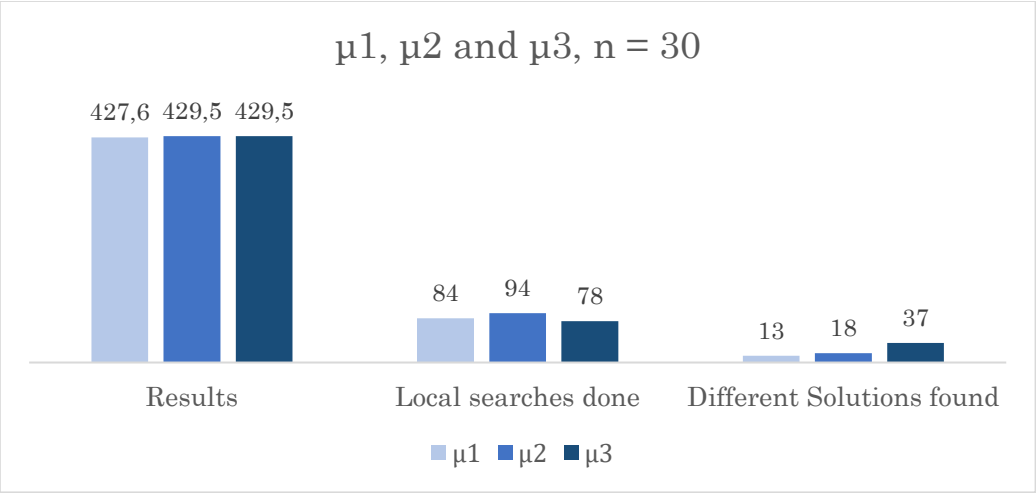

Figure 5.10: Results, Number of Local searches done, and different solutions found for every μ

We can observe that ILS performed slightly better using μ_2 and μ_3 compared to μ_1, but differences are not considerable. Moreover, in the same Figure the global average of the total local searches done and of the number of different solutions found are shown. These last metrics allow us to evaluate the effectiveness of each alternative to escape from the current local optimum. In this way, we can realize that the algorithm falls many times in an already found solution, especially for alternatives with less perturbation μ_1 and μ_2.

For this reason, a new *adaptive* function for μ is developed with the aim of avoiding both a search in a fraction of the solution space that do not present any improvement with respect to the current best solution and to be trapped in the same local optimum. Let us call these new perturbations μ_1', μ_2' and μ_3'.

In these *adaptive* perturbations, the initial μ is the same as before, and it grows to $\mu + 1$ for the origins if no improvement is found after five consecutive local searches. The same is applied for the destinations after presenting other five iterations without any progress in the solution. Moreover, if the local search falls on an already found local optimum three times, then $\mu = \mu + 1$, first for the destinations and secondly (after other three times of finding a repeated local optimum) for the origins. In this way, we alternately increase the number of origins and destinations randomly swapped, thus increasing the perturbation, and hopefully getting away of the current local optimum by an appropriate use of the *memory*. When an improvement to the current best solution is found, μ is reset to its original value, and the counter for repeated solutions and no improvements is reset to 0. This idea is summarized in ILS *Adaptive perturbation Algorithm*, using μ_2'.

Adaptive perturbation Algorithm

Step 1: *Local Search and parameters definition*

Implement the *Local Search Algorithm* using N5, obtaining s^*.

Set $\mu_2(n,k)$; no_improvements_counter = 0; repeated_sol_counter = 0;
solutions_list= [s^*]

Step 2. *Perturbation.*

If no_improvements_counter is multiple of 5:
    $\mu_{2origins} = \mu_{2origins} + 1$
Elif no_improvements_counter is multiple of 10:
    $\mu_{2destinations} = \mu_{2destinations} + 1$
If repeated_sol_counter is multiple of 3:
    $\mu_{2destinations} = \mu_{2destinations} + 1$
Elif repeated_sol_counter is multiple of 6:
    $\mu_{2origins} = \mu_{2origins} + 1$
$s' = s^*$.random.swap($\mu_{2origins}$ and $\mu_{2destinations}$)

Step 3. *New Local Search and acceptance test*

While stopping criterion is not met **do**:

    $s^{*\prime}$= LocalSearch(s')

    If $s^{*\prime}$ improves s^*:
        $s^* = s^{*\prime}$
        Reset μ_2, no_improvements_counter, repeated_sol_counter

    Else:
        no_improvments counter += 1

    If $s^{*\prime}$ in solutions_list:
        repeated_sol_counter += 1

    Else:
        Add $s^{*\prime}$ to solutions_list

    s' = Perturbation ($s^{*\prime}$)

Results for the *adaptive perturbation*, number of local searches performed, and different solutions found for n = 30 are presented in Figure 5.11, as average of all instances and k_s.

We can observe that although the average results are only slightly better for every μ' with respect to the corresponding μ, the investigated solution space is much larger, especially for μ_1 and μ_2.

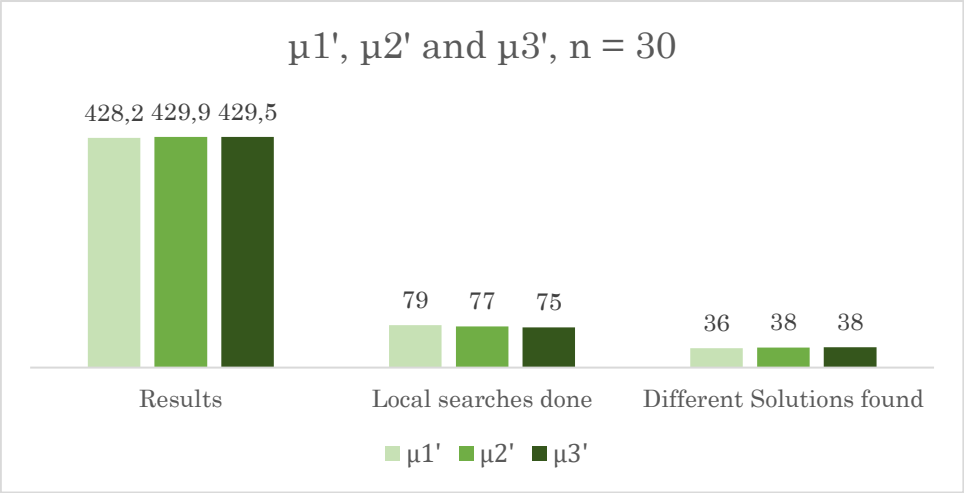

Figure 5.11: Results, Number of Local searches done, and different solutions found for every μ'

In addition to that, let us compare for the same instances of size 30 the performance of the different μ_s considering the percentage of improved solutions with respect to LS, and by a pairwise comparison. The first indicator is just pointing out that the *adaptive* perturbation allows a slightly better performance, as showed in Table 5.1.

	μ1	μ1'	μ2	μ2'	μ3	μ3'
% of solutions improved	66%	74%	69%	74%	70%	74%

Table 5.1: Percentage of solutions of LS that are improved by ILS, for every μ, n = 30

Moreover, as the percentage of solutions improved is similar, let us consider a pairwise comparison between the six alternative perturbations (by comparing the solution values for every k and instance). It is not straightforward the selection of one of them, however, we notice that μ_2' is the only one that always wins, even if it is sometimes by just a small difference, as showed in Table 5.2.

	μ1	μ1'	μ2	μ3	μ3'
μ2' wins	47%	23%	26%	29%	32%
Other μ wins	11%	20%	18%	23%	27%
Draw	42%	57%	57%	48%	41%

Table 5.2: Results of μ_2' against the other five types of perturbations

As a conclusion, the *adaptive* perturbation rule is useful, and the optimum function to calculate the number of perturbed rows and columns, considering all these analyses, is μ_2'.

5.6.1. Improvements achieved by ILS compared to LS

In Table 5.1, we notice that around 74% of the solutions given by LS are improved by ILS. However, let us analyze in detail these improvements to evaluate the effectiveness of ILS.

For n = 10, the available window for improvement was small, because, as previously explained, in 61% of the studied instances and k_s the optimal was found by LS. However, for a time limit of 2 seconds, ILS does improve the solutions, leading to a 98% of global optimal solutions found.

With respect to n = 30, ILS was able to increase the solution value of LS, on average, in 66,5/90 = 74% of the cases, after running ILS 5 times for the 10 instances, for the corresponding k_s and a time limit of 60 seconds. Below an example of the LS and ILS results is presented, for a run of the algorithm in which the improvement rate was 73%. Table 5.3 contains the results obtained by LS, while Table 5.4, by ILS, and a green value indicates an improvement.

Local Search results										
n = 30	k									
Instance	3	6	9	12	15	18	21	24	27	Avg. Inst.
1	266	392	451	487	490	488	489	477	464	445
2	257	356	418	462	467	455	417	428	406	407
3	248	373	399	429	412	430	458	436	426	401
4	239	381	407	417	393	388	410	407	403	383
5	254	391	430	414	450	464	473	465	447	421
6	258	374	431	451	441	475	445	463	456	422
7	258	410	399	404	412	425	455	455	427	405
8	259	394	457	496	439	460	454	447	438	427
9	256	401	414	444	469	488	463	453	456	427
10	262	402	406	437	488	456	470	475	457	428
Avg. k	256	387	421	444	446	453	453	451	438	417

Table 5.3: Results of Local Search

Iterated Local Search results										
n = 30	k									
Instance	3	6	9	12	15	18	21	24	27	Avg. Inst.
1	269	413	453	488	490	496	490	477	464	449
2	268	405	448	471	469	468	433	428	407	422
3	265	382	425	433	456	470	458	441	429	418
4	269	383	419	421	420	424	410	408	403	395
5	271	393	438	442	461	467	473	465	455	429
6	267	400	461	490	477	481	472	468	456	441
7	273	410	443	449	457	457	455	455	444	427
8	265	394	464	496	484	460	454	448	438	434
9	272	401	444	465	490	488	463	461	456	438
10	270	409	456	477	488	498	492	475	467	448
Avg. k	269	399	445	463	469	471	460	453	442	430

Table 5.4: Results of Iterated Local Search

We can observe that ILS is usually able to improve the results. However, we also notice different behaviors depending on the value of k. For $k \leq 18$, ILS reached, on average of 5 runs, an 86% rate of improvement, while for $k > 18$, this value decreases to 51%. This induces to think that there could be an opportunity of enhancing the perturbation phase for those values of k.

Analyzing Table 5.5, we notice that for $k > 18$ the number of LS the algorithm can perform within the time limit of one minute is small. Additionally, the adaptive part of the perturbation phase has a negligible impact for these cases, since it needs at least 5 iterations without improvements, or 3 repeated local optima found before starting to make stronger perturbations, and we can observe that the number of local searches completed is between 3.5 and 6.5. Finally, we notice that the percentage of repeated local optima found is relatively large.

	k = 21	k = 24	k = 27
Average number of LS completed	3,6	3,7	6,2
Average different local optima	2,6	2,3	2,8
% of repeated local optima	27,8%	37,8%	54,8%

Table 5.5: Original Perturbation

	k = 21	k = 24	k = 27
Average number of LS	3,4	3,1	5,3
Average different local optima	3,2	3,0	3,2
% of repeated local optima	5,9%	3,2%	39,6%

Table 5.6: New Perturbation

With that being said, let us consider a new perturbation rule that is stronger (even stronger than μ_3') by increasing the number of swapped rows and columns:

$$New\ perturbation = Initial\ perturbation + \left(10 * \frac{n-k}{n}\ \ if\ \frac{k}{n} \geq 0.7\right)$$

The new perturbation was tested for k = 21, 24 and 27, for the 10 instances. It was able to reduce the percentage of repeated local optima found, as shown in Table 5.6, but, when compared to LS, results were a bit worse than for the original

perturbation, reaching a 40% of rate of improvement. In conclusion, we refute the hypothesis that a stronger initial perturbation can lead to better results.

The smaller improvement rate reached for bigger k_s may be caused by the fact that having greater k_s implies fewer local searches done within the time limit, since the complexity of the algorithm considerably grows with k (see section 5.4). This idea is summarized in Figure 5.12.

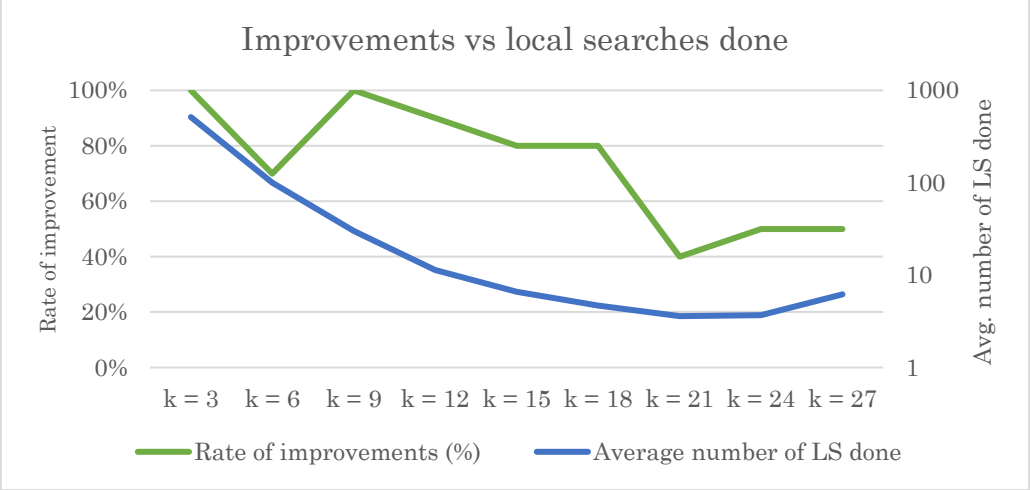

Figure 5.12: Rate of improvements (%) compared to the number of iterations (LS)

Finally, for $n = 50$, we set a time limit of 300 seconds, and we use the values of k defined in section 4.3. We realize an overall improvement rate of 57%, that can be subdivided in two, depending on k. That is, 80% for $k \leq 25$, and 25% for $k > 25$. Again, trying to modify the perturbation does not make sense, since the reason of the smaller improvement rate for bigger k_s is simply that the number of local searches done within the established time limit (shown in Table 5.7) is not enough to allow ILS to always improve the LS results, even though we are not trapped in the same local optima.

	k = 30	k = 35	k = 40	k = 45
Average number of LS	1,9	1,7	1,9	3,5
Average different local optima	1,9	1,7	1,7	2,5
% of repeated local optima	0%	0%	10,5%	28,6%

Table 5.7: Number of LS, percentage of repeated local optima, n = 50

To sum up, ILS improves the LS results, especially for intermediate and low values of k. For bigger k_s, fewer iterations of local search can be performed within the time limit, and this leads to fewer solutions ameliorated.

5.7. A final improvement of the neighborhood search

The neighborhood search rule N5 (see section 5.4) performed a fast search, by swapping first the origins and destinations that are more likely to induce an improvement in the leader's objective function. However, the previously defined acceptance criterion to verify the adequacy of each neighbor consisted of solving the AP and checking if there was an improvement with respect to the current best solution. So, the time complexity to evaluate each neighbor was $O(k^3)$ (remember that k is the follower's problem size).

In this section, we add an intermediate step in the acceptance criterion, where we iteratively construct upper bounds to the optimal solution of the AP for each neighbor, and which time complexity is $O(k^2)$. This is advantageous because whenever an upper bound is lower than or equal to the leader's current best solution, we can skip the resolution of the AP, thus reducing the computational time of the acceptance test of that neighbor.

Let $C_{complete}$ be the original cost's matrix of size $n\ x\ n$ and $C_{current}$ be the cost's matrix of size $k\ x\ k$ that corresponds to the selection associated to the current solution of the leader:

$$C_{complete} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix};\ C_{current} = \begin{bmatrix} a_{\mathrm{I\,I}} & a_{\mathrm{I\,II}} & \cdots & a_{\mathrm{I}\,k} \\ a_{\mathrm{II\,I}} & a_{\mathrm{II\,II}} & \cdots & a_{\mathrm{II}k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k\,\mathrm{I}} & a_{k\,\mathrm{II}} & \cdots & a_{kk} \end{bmatrix}$$

Latin number's indexes are used for $C_{complete}$ while roman number's indexes denote that the origin or destination is selected in the current solution of the leader and are used for $C_{current}$. Let l and m be the indexes for generic rows and columns of $C_{current}$, where each $a_{lm} \in C_{current}$ is equal to the corresponding cost $c_{ij} \in C_{complete}$, but considering that the indexes are not equivalent, i.e. a_{11} is not necessarily equal to c_{11}, but only if origin 1 and destination 1 are selected in the current solution.

Let F_s be a vector of size k that contains the costs of all the assignments that correspond to the optimal follower's solution:

$$F_s = [b_{\mathrm{I}}, b_{\mathrm{II}}, \ldots, b_k]$$

Where we can define b as:

$$b_{row\_l} = selected(a_{lm})\ \forall\ l \text{ or } b_{col\_m} = selected(a_{lm})\ \forall\ m$$

Notice that the values of F_s will be the same either if they are constructed following b_{rows} or $b_{columns}$ definitions, however, with the only difference that they will be sorted by rows or by columns order, correspondingly.

Now let us consider that a neighbor of $C_{current}$ is generated by swapping row x that is in the solution with row y that is not in the solution and let us call it $C_{swap\_rows(x,y)}$. This means that on $C_{current}$, the values $[a_{x\,\mathrm{I}}, a_{x\,\mathrm{II}}, \ldots, a_{xk}]$ are replaced by $[c_{y\,\mathrm{I}}, c_{y\,\mathrm{II}}, \ldots, c_{yk}]$.

In addition to that, let us suppose that on the AP solution of $C_{current}$ the follower has made the following assignments:

1. Origin x to destination m. Meaning that $a_{xm} = b_{row\_x} = b_{col\_m}$
2. Origin p to destination q. Meaning that $a_{pq} = b_{row\_p} = b_{col\_q}$

3. Origin r to destination s. Meaning that $a_{rs} = b_{row\_r} = b_{col\_s}$

In this way, if we consider $II < x < p < r < k$ we can define the F_s vector of $C_{current}$ sorted by rows order as:

$$F_{s\_current} = [b_{\mathrm{I}}, b_{\mathrm{II}}, \dots, a_{xm}, \dots, a_{pq}, \dots, a_{rs}, \dots, b_k]$$

Hypothesis. If

$$c_{yq} + a_{ps} + a_{rm} \leq a_{xm} + a_{pq} + a_{rs} \tag{5.1}$$

Then the neighbor $C_{swap\_rows(x,y)}$ will never improve the leader's current best solution.

Proof. Let us consider the solution value of $C_{current}$, as the sum of the vector $F_{s\_current}$:

$$C_{current\ value} = \sum_{\mathrm{I}}^{k} b_l = b_{\mathrm{I}} + b_{\mathrm{II}} + \dots + a_{xm} + \dots + a_{pq} + \dots + a_{rs} + \dots b_k$$

Let us construct a feasible solution for the AP of $C_{swap\_rows(x,y)}$ that contains c_{yq}, a_{ps} and a_{rm}, by making some changes to a copy of $F_{s\_current}$ (since the original vector cannot be modified), let us say $F_{s\_current\_copy}$. First of all, notice that as row x is not present in this neighbor, the cost a_{xm} cannot be present in $F_{s\_current\_copy}$, leaving the incumbent row y and column m unassigned. Now let us follow these steps:

1. Assign origin y to destination q, so $c_{yq} = b_{row\_y} = b'_{col\_q}$. This completes the assignment of origin y but implies that the previously assigned cost a_{pq} must be deleted from the $F_{s\_current\_copy}$ vector, otherwise we would select two edges for destination q and we would not satisfy constraint (3.3) of the AP. Therefore, an assignment is to be done for row p.
2. Assign origin p to destination s, so $a_{ps} = b'_{row\_p} = b'_{col\_s}$. As a consequence, we complete the assignment of row p, but we must delete the cost a_{rs} from vector $F_{s\_current\_copy}$, leaving row r unassigned.
3. Assign row r to column m, so $a_{rm} = b'_{row\_r} = b'_{col\_m}$.

In this way, we completed the assignments satisfying all the AP constraints (see section 3.1.1), so we constructed a feasible solution that, in turn, represents an upper bound to the optimal solution of the AP of $C_{swap\_rows(x,y)}$. As we made only three changes to the vector $F_{s\_current\_copy}$, we can compute the value of the upper bound as:

$$C_{swap(x,y)\ upper\ bound\ value} = C_{current\ value} + \delta$$

Where:

$$\delta = -b_{row_x} - b_{row_p} - b_{row_r} + b_{row_y} + b'_{row_p} + b'_{row_r} =$$

$$\delta = -a_{xm} - a_{pq} - a_{rs} + c_{yq} + a_{ps} + a_{rm}$$

Therefore, if:

$$c_{yq} + a_{ps} + a_{rm} \leq a_{xm} + a_{pq} + a_{rs} \text{ (hypothesis)}$$

Then $\delta \leq 0$, meaning that:

$$C_{swap(x,y)\ upper\ bound\ value} \leq C_{current\ value}$$

Of course, the solution of the AP of $C_{swap(x,y)}$ will satisfy:

$$C_{swap(x,y)\ value} \leq C_{swap(x,y)\ upper\ bound\ value}$$

And finally:

$$C_{swap(x,y)\ value} \leq C_{current\ value}$$

We have found that the follower's solution for $C_{swap(x,y)}$ will always be lower than or equal to the value of $C_{current}$, meaning that it is impossible that the leader realizes an improvement by swapping row x and row y (since he wants to increase the value of his current solution), and the hypothesis is proved. □

Notice that the construction of the upper bound was done by changing only three values in the vector $F_{s\_current\_copy}$, meaning that constant time was required. Therefore, we can reformulate the neighbor's acceptance criterion by adding an intermediate step in which we generate at most k^2 upper bounds. If at any point the inequality (5.1) is satisfied, we skip the resolution of the AP and reject the neighbor, thus reducing the acceptance criterion's computational time for the corresponding neighbor from $O(k^3)$ to $O(k^2)$. Otherwise, we must solve the AP, reaching a similar time complexity compared to the one of the previous acceptance criterion, since $O(k^3 + k^2) = O(k^3)$.

The *Improved Acceptance Criterion Algorithm* describes the upper bounds generation for a neighbor created by swapping rows x (in the current best solution) and y (not in the current best solution), but it can be analogously applied for the other rows swaps and columns swaps. The input of this algorithm is $C_{current}$, $F_{s\_current}$ and row y.

Improved Acceptance Criterion Algorithm

Step 1. Create the upper bounds and check if condition (5.1) is satisfied.

Solve_AP = True

$a_{xm} = b_{row\_x} = b_{col\_m}$, store m

For q in [1, 2, ..., k]: (excluding m)

    $a_{pq} = b_{col\_q} = b_{row\_p}$, store p

    For s in [1, 2, ..., k]: (excluding m and q)

        $a_{rs} = b_{col\_s} = b_{row\_r}$, store s

        If $c_{yq} + a_{ps} + a_{rm} \leq a_{xm} + a_{pq} + a_{rs}$:

            Solve_AP = False

            Stop both for loops.

Return Solve_AP

Step 2. Evaluate the adequacy of the $C_{swap\_rows(x,y)}$ neighbor.

If Solve_AP = True:

Set $C_{swap\_rows(x,y)} = C_{current}$.swap(row x, row y)

Solve the AP to check the adequacy of neighbor $C_{swap\_rows(x,y)}$

Else:

Go to the next neighbor.

Notice that by applying the *Improved Acceptance Criterion Algorithm*, the computational complexity of the *Local Search Algorithm* is not reduced (since we may never satisfy condition (5.1)), but we achieve a considerably faster search by reducing the computational time of the acceptance test of a fraction of the neighborhood, as it is showed in the next section.

5.7.1. Implementation of the improved acceptance criterion

We will now analyze the improvements reached by this new acceptance criterion with respect to the previous one. Recalling section 5.4, the optimal neighborhood rule was *N5*, so let us add to it this last improvement and call it *Improved N5*.

First of all, considering only local search, let us evaluate the number of neighbors that *Improved N5* is able to refute without solving the AP, let us say *skip,* over all the neighbors checked by *N5*, until the local optimum is reached. We can define:

$$Percentage\ of\ skipped\ neigbors\ = \frac{Number\ of\ neighbors\ skipped\ by\ Improved\ N5}{Number\ of\ neighbors\ checked\ with\ N5}$$

Figure 5.13 summarizes this metric for n = 10, 30 and 50, where each value represents the average of the 10 corresponding instances, and k is presented as a function of n (data labels are left for n = 50).

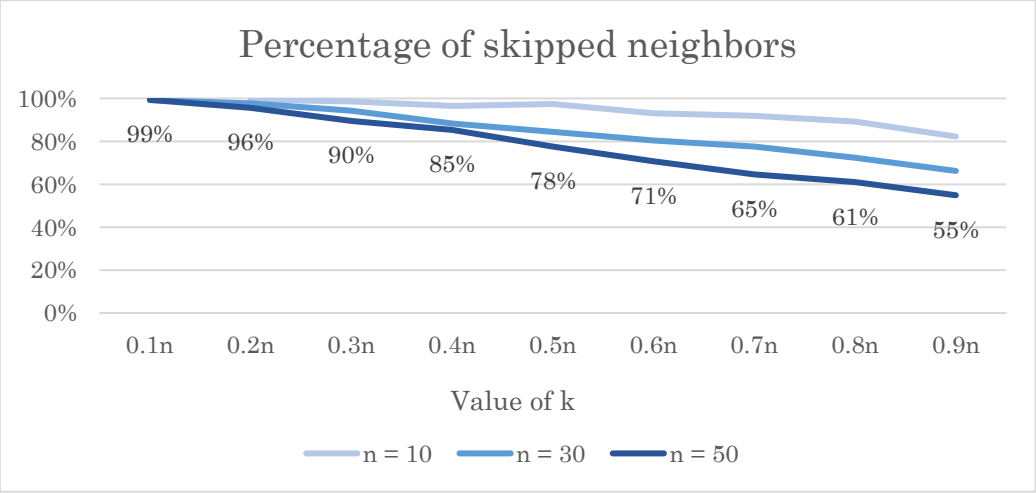

Figure 5.13: Percentage of neighbors skipped by the use of the improved acceptance criterion in LS

Results show that a considerable fraction of the total evaluated neighbors (i.e., more than a 50% even for the worst case) are *skipped*. Moreover, we notice that this improvement is more significant the lower the value of k with respect to n, and the smaller the instance size.

We additionally present a numerical example of the remarkable reduction reached. Figure 5.14 shows the number of neighbors in which the AP is solved while using *N5* and *Improved N5,* as the average value for the ten instances, for $n = 50$.

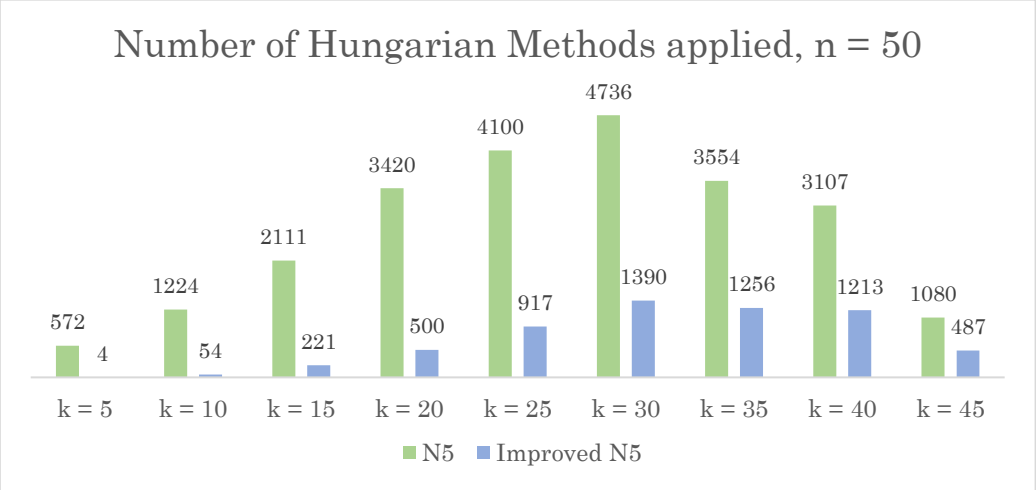

Figure 5.14: Number of Hungarian Method needed to reach local optimality, n = 50

Let us then compare these two approaches with respect to the results of ILS. Table 5.8 presents the overall win rate of ILS performed with *N5* against ILS performed with *Improved N5*, for time limits of 60 and 300 seconds for $n = 30$ and $n = 50$. We can observe that *Improved N5* outperforms *N5*. The reason for these better performances is that the number of LS done within the time limit increased due to the fact that *Improved N5* runs faster.

	N5 wins	*Improved N5* wins	Draw
n = 30	10%	42%	48%
n = 50	6%	58%	37%

Table 5.8: Comparison between ILS implementing N5 and Improved N5

We realize a small percentage in which *N5* wins, and the reason is that the random nature of the perturbation phase can lead to better results in less local searches done.

6. A matheuristic approach

6.1. Matheuristics: definitions and classification

Both exact and heuristic approaches have their own advantages and disadvantages. Mathematical programming can guarantee optimality, but for big instances of the problem the required CPU time can be so large that it is practically unsolvable. On the other side, heuristic techniques provide fast and good solutions, but cannot guarantee its optimality. In the recent years, a lot of attention was devoted to the combination of these two while solving hard combinatorial optimization problems, leading to hybrid approaches, also known as matheuristics.

There exist various ways in which exact and heuristic methods have been brought together to improve the overall performance of an algorithm. On the one hand, there exist *Collaborative Combinations*, in which the methods are not part of each other but they only exchange information. In this approach, one option is that exact and heuristic methods are executed *sequentially*, meaning that either the heuristic is executed as a pre-processing step of the exact method or vice-versa. One can also solve to optimality the LP relaxation of the problem and after a rounding procedure the solution is given as input to the heuristic approach. Moreover, instead of sequentially both methods may be executed *intertwined or in parallel*, please refer to [27] for more information on the matter.

On the other hand, there are the *Integrative Combinations*, in which one algorithm is embedded into the other. One is the "master" and the other acts as a "slave". Either the heuristic is the high-level algorithm and controls the calls to the exact approach or vice-versa. When incorporating exact methods into heuristics, we can use the LP relaxation as mentioned above, but in addition to use it as an initial solution generator, it can be used to heuristically guide the neighborhood search.

Moreover, exact methods can be used to perform a *Large Neighborhood Search*, as mentioned in [28]. The main point is to model the large neighborhood search problem, solve it exactly, and the optimal solution is then selected as the next neighbor. To model this neighborhood problem, there exist the full or partial neighborhood exploration. In the first one, each feasible solution to the search problem induces a move on the local search. In the second one, at each iteration a part of the solution is left fixed and the rest free, and the exact method is called to optimize the free variables.

Finally, the incorporation of metaheuristics to enhance the performance of mathematical programming is also possible, as explained in [29]. While solving combinatorial optimization problems exactly, usually tree search is applied. The benefits from metaheuristics are reached by giving an initial feasible solution which induces a good lower bound and reduces the search space. Additionally, throughout the tree search, metaheuristic might be called to provide hopefully improved solutions that allow to further prune the search tree. Finally, we mention that metaheuristic can be also used by exact methods on *column and cut generation.*

6.2. A matheuristic algorithm to improve ILS

In this section, a matheuristic algorithm is defined with the aim of further improving the ILS results. As summarized in *Matheuristic Algorithm*, the first step is done by the metaheuristic. We use *Improved N5* in this step, since it showed to be the best option to implement ILS. Once it reaches the time limit, the best solution $s^* = s_0$ is given as input to the Step 2, in which the *partial neighborhood exploration* approach is applied. Following the above explanation, a *partial neighborhood search problem (PNSP)* is generated, in which a fraction α of the selected origins and destinations of s^*, are set to be in the solution by adding constraints $o_i, d_j = 1$ and a fraction $0 < \alpha < 1$ of the non-selected ones are set to be out of the solution by adding constraints $o_l, d_m = 0$. Each *PNSP* with its corresponding fixed variables represents a neighbor n_i of the neighborhood $N(s^*)$. The selection of the fixed variables is randomly done, with the exception of one variation that will be explained later. In this step, the exact method solves to optimality relatively fast (depending on the number of variables that are fixed) this constrained model, producing a solution s_i. Then, the *first improvement strategy* is applied: if an improvement is found on a neighbor n_i, then we move to this new solution by setting $s^* = s_i$, and finally we perform a new search of the neighborhood $N(n_i) = N(s^*)$. The algorithm stops when stopping criteria is met.

Matheuristic Algorithm:

Step 1: *Initial solution.*

Perform ILS until the time limit is reached to obtain s_0 and set $s^* = s_0$.

Step 2: *Matheuristic*

While stopping criteria is not met **do:**

> From s^*, define a *PNSP* by randomly fixing a number $\alpha * k$ of selected o_i, d_j to be in the solution and randomly fixing a number $\alpha * (n - k)$ of the non-selected o_l, d_m to be out of the solution.
>
> Solve to optimality the *PNSP,* obtaining s_i
>
> **If** s_i improves s^*, **then** $s^* = s_i$

end do

Step 3: *Return* s^*.

The matheuristic procedure described in Step 2 belongs to the group of *integrative combinations*. Additionally, we could also say that *Matheuristic Algorithm* as a whole, acts as a *collaborative combination* matheuristic, since we first apply a metaheuristic and then further optimize it with an algorithm that takes advantage of exact methods.

6.3. Determining the value of α

The success of the matheuristic algorithm depends on a precise definition of α. Note that if $\alpha = 1$, the solution of the *PNSP* will be instantly given and exactly equal to the initial s_0. The closer α is to 1, the less chances to escape from the current local optimum. Moreover, if $\alpha = 0$, we were simply applying an exact model algorithm and the closer it is to 0, the more time is required to solve the problem to optimality, reducing the efficiency of the algorithm. By these reasons, we may start testing the *Matheuristic Algorithm* setting $\alpha = 0.5$ for Step 2.

Let us first test the algorithm for n=30 for practical reasons: the time limit of 60 seconds defined for Step 1 and Step 2 for this size is much less than the one for bigger instances (300 seconds for n=50), so more analysis can be made in less time. After defining the optimum α, we are going to test it for n=50. For smaller instances, optimality was reached by previous methods, so there is no opportunity to improve the solution.

Let Matheuristic A (Math. A) be the *Matheuristic Algorithm* in which $\alpha = 0.5$. By running the algorithm, the main disadvantage that it presented, was that for different values of k, the average number of iterations (calculated considering the 10 instances) that can be done within the time limit were pretty dissimilar. As showed in the blue line in Figure 6.1, for k = 15 this average is 13 (for some instances only 6 and 7 iterations were done) while for other k_s it was greater than 50.

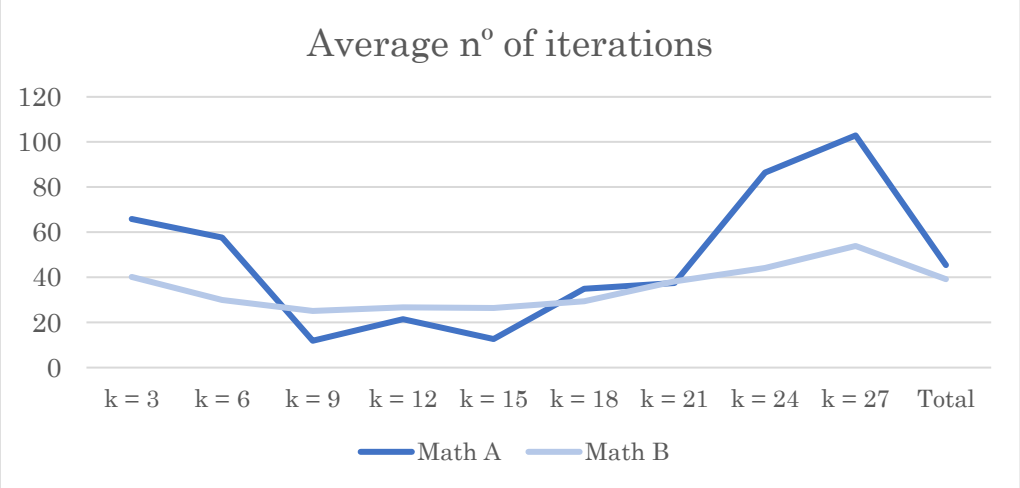

Figure 6.1: Average number of iterations done for each value of k, for Math. A and Math. B

However, this does not imply that defining a bigger α, for example, $\alpha = 0.7$ is better: although the number of iterations will increase, the chances to escape from the local optimum will decrease. For this reason, an adaptive α is proposed and is used in Matheuristic B (Math. B), represented by the light-blue line in Figure 6.1. This consists of starting with $\alpha = 0.7$ (for which the solver takes less than a second to solve the corresponding *PNSP*), and if the algorithm is not able to improve the solution within 5 iterations, then $\alpha = \alpha - 0.05$. This logic is repeated every 5 iterations without an improvement. When we manage to increase the value of the

solution, we restart $\alpha = 0.7$. In this way, we reached a more equilibrated number of iterations for different values of k, and at the same time we ensure that if for any instance it is difficult to improve the solution for high values of α, the algorithm will gradually reduce it, reaching at some cases an improvement only for $\alpha = 0.2$. A more detailed analysis on the matter is presented in section 6.4.2.

In addition to that, a third α is proposed. While evaluating the output of the algorithm as it run, many times a portion of the randomly fixed variables repeated themself one iteration after another. This is a waste of time because if the $PNSP_i$ is similar to the $PNSP_{i-1}$ the exact method is not likely to reach a different local optimum. In this way, let us define the Matheuristic C (Math. C) as the *Matheuristic Algorithm* such that it uses the adaptive α of Math. B and apart from that, the selection of the fixed variables is not always random. It is random at iteration i, but at iteration $i+1$ we add the constraint that we must select as many variables that were not selected in the previous iteration as possible. We can summarize this idea as follows.

Matheuristic C: selection of fixed variables

Iteration i: *Random selection*

Select randomly all the fixed variables.

Iteration $i+1$: *Fix as many variables that were not fixed in Iteration i as possible.*

If $\alpha > 0.5$ **then** select all the non-selected variables at It. i, and the rest of them randomly.

If $\alpha = 0.5$ **then** select all the non-selected variables at It. i.

If $\alpha < 0.5$ **then** randomly select the variables, but only from the list of all the non-selected variables at It. i.

6.4. Implementation

In this section, a comparison between the three alternatives of α is carried out, and also an analysis of the improvements found in function of α and k is presented.

6.4.1. Results for the three matheuristics

First of all, the three variations of α were tested, following the Matheuristic Algorithm. For $n=30$, Step 1 (ILS) run for 60 seconds, and the same time limit was established for Step 2 (matheuristic part), leading to a total duration of 120 seconds. This was repeated for the same 10 instances and k_s as in chapters 4 and 5.

A pairwise comparison between Math. C and the other two approaches is shown in Table 6.1.

Math. A wins	6%
Math. C wins	11%
Draw	83%

Math. B wins	6%
Math. C wins	12%
Draw	82%

Table 6.1: Comparison between the three variations of α

We can observe that there is a very high draw rate, due to the fact that the differences on the algorithms were not so huge. However, we can also affirm that Math. C is slightly better than the other two methods, and therefore is considered as the best one.

Let us then analyze Figure 6.2, that presents how many times the matheuristic part of the Math. C algorithm (Step 2) is able to improve the ILS solution (Step 1) for every value of k (please remember that there are 10 instances). The global improvement rate is 12% (11 solutions). Moreover, there is a pronounced difference between $k \leq 15$, values for which the local optimum reached by the metaheuristic was very difficult to improve by the matheuristic with only a 2% of ameliorated solutions; and $k > 15$, where 25% of the times the solution values were increased.

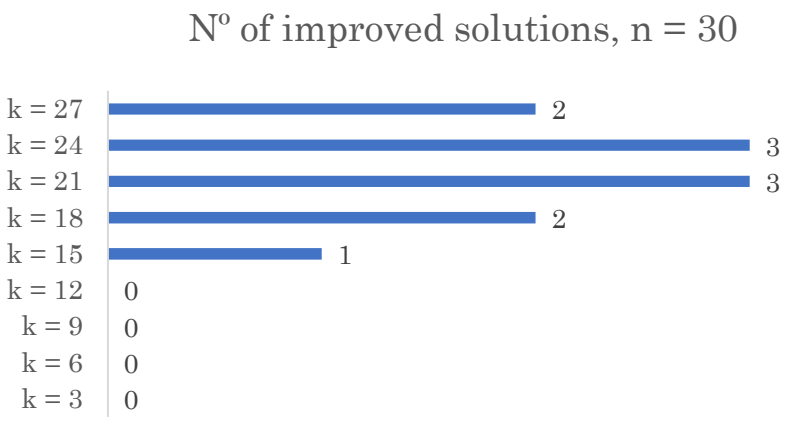

Figure 6.2: Number of solutions of ILS (Step 1) that were improved by Matheuristic C (Step 2)

Moreover, let us analyze the performance of Math. C for the 10 instances of size 50 and the same k_s of the previous chapters. The time limit was set to 300 seconds for Step 1, and the same for Step 2. Given the larger size of the problem, there was logically a bigger available room to increase the value of the local optimum given by ILS than for smaller n_s, and we realized a greater rate of improvement achieved by Step 2, globally reaching 23/90 = 26% ameliorations of the ILS best solutions. Again, it is confirmed in Figure 6.3 that for smaller k_s it is more difficult to enhance the ILS solutions than for larger k_s, reaching an 8% of improvements for $k \leq 15$ and a 48% for $k > 25$.

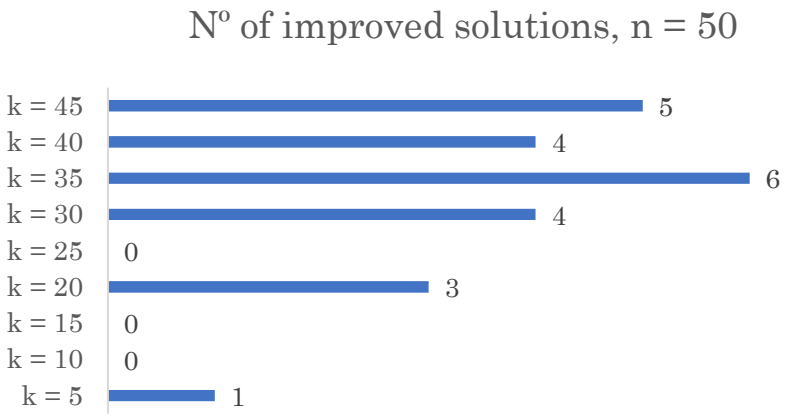

Figure 6.3: Number of solutions of ILS (Step 1) that were improved by Matheuristic C (Step 2)

6.4.2. A complementary analysis of the adaptive α

Let us present an analysis that shows how many improvements of the ILS solution were found for each value of α for every k, while implementing Matheuristic C Algorithm. Here, improvement does not only represent an increase on the value of a solution given by ILS, as analyzed above, but it also includes all the following improvements of the new s^*, reached before the algorithm stops.

This is useful to prove the effectiveness of utilizing an *adaptive* α. In Figure 6.4 we notice that the improvements were mainly found for big and intermediate values of α. However, we also realized three improvements for $\alpha \leq 0.40$ (specifically two for $\alpha = 0.30$ and one for $\alpha = 0.20$) and they were present for the biggest k_s.

For the cases in which no improvement was found, we at least are sure that we have explored a big solution space. On the one hand, by applying many iterations, and on the other hand by decreasing the value of α, up to 0.20 sometimes, without being able to increase the solution value, which means that the local optimum is robust and difficult to improve.

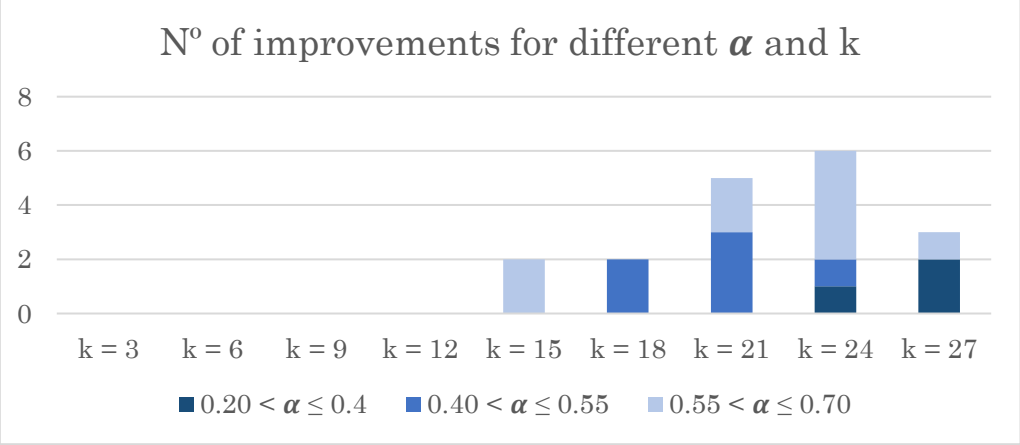

Figure 6.4: Improvements reached by Math. C, depending on k and α

As a conclusion of this chapter, the adaptive α is useful to optimize the implementation of the matheuristic and Matheuristic C is the preferred approach. Moreover, we can affirm that for small values of k the solution of ILS is really hard to be improved, while it can be sometimes improved for big values of k.

7. Results and conclusions

7.1. Considerations

In this chapter, the best of all the proposed alternatives of each of the three approaches developed in chapters 4, 5 and 6 are compared. Consider that Model d) presented the best performances among the MIP models computing an exact or approximate solution, ILS using the *Improved N5* rule to perform the neighborhood search and μ_2' in the perturbation phase showed to be the best metaheuristic approach and finally Math. C was the best of the developed matheuristic approaches. These three specific models and algorithms will be used while referring to MIP, ILS and Math. respectively, on the current chapter.

The three models and algorithms were coded in Python 3 while the CPLEX Optimization Studio IDE 20.1 was called from Python by using the *docplex* library. Some important python libraries that were used and greatly simplified the implementation phase are the following.

- **Docplex:** is the API that connects Python with CPLEX.
- **Munkres:** this library is able to apply the Hungarian Method to the given matrix in $O(n^3)$ time, and at the same time it registers the selected c_{ij} and their corresponding values. It was applied to solve the AP on LS.
- **NumPy:** is a fundamental package for scientific computing on Python. It is extremely useful to work with arrays and was mainly used to generate the neighbors on the LS.
- **Pandas:** is a data analysis module for Python. It was used to export the data into Excel. Combined with a for loop, the code exports the results for the 10 instances and the 9 different values of k in only one run, avoiding the required time to copy and paste the outputs of the code in Excel manually.
- Other libraries as **random**, **operator**, **math**, etc.

A final consideration is that instances and k_s for each problem size are the same as studied in the previous chapters, please refer to section 4.3 for a detailed explanation of the generation of the instances and the corresponding k_s for different n_s.

7.2. Results for n = 30

Differently from the previous section, in this section the time limit is set to 120 seconds for the three approaches, so we are able to compare them under equal conditions.

Table 7.1 presents the corresponding results for n = 30. In bold it is marked the solution that is the maximum value found over the three approaches. First of all, the MIP approach is clearly outperformed by the other approaches, as it only finds the best solution 33 times, against 78 and 76 of ILS and Math (out of 90 problems solved).

Size	k	Instance	MIP	ILS	Math.
30x30	3	1	265	**269**	**269**
		2	267	**268**	**268**
		3	**265**	**265**	**265**
		4	**269**	**269**	**269**
		5	268	**271**	**271**
		6	261	**267**	**267**
		7	**273**	**273**	**273**
		8	259	**265**	**265**
		9	268	**272**	**272**
		10	266	**270**	**270**
	6	1	379	**413**	**413**
		2	366	**405**	402
		3	364	**392**	384
		4	**387**	**387**	**387**
		5	399	398	**401**
		6	385	**400**	**400**
		7	389	**410**	**410**
		8	390	**397**	**397**
		9	**401**	**401**	**401**
		10	388	409	**411**
	9	1	458	**463**	460
		2	414	**456**	448
		3	419	**427**	**427**
		4	432	**441**	**441**
		5	424	**438**	**438**
		6	**461**	**461**	**461**
		7	431	**443**	**443**
		8	459	**464**	**464**
		9	**445**	444	**445**
		10	453	**457**	456
	12	1	492	492	**493**
		2	462	**471**	**471**
		3	447	**456**	**456**
		4	435	**441**	**441**
		5	**451**	**451**	**451**
		6	460	**494**	**494**
		7	455	**456**	**456**
		8	**496**	**496**	**496**
		9	**473**	471	**473**
		10	**487**	**487**	477
	15	1	502	**509**	**509**
		2	**471**	**471**	**471**
		3	460	**461**	456
		4	424	**431**	**431**
		5	456	**465**	**465**
		6	487	**488**	487
		7	**457**	**457**	**457**
		8	448	**482**	480
		9	451	**490**	**490**
		10	486	**488**	**488**
	18	1	494	**508**	**508**
		2	**468**	**468**	**468**
		3	464	**470**	**470**
		4	420	**424**	417
		5	467	**470**	468
		6	**484**	481	482
		7	454	**457**	**457**
		8	450	**475**	**475**
		9	468	**488**	**488**
		10	468	**498**	**498**
	21	1	482	**497**	490
		2	**443**	437	**443**
		3	467	**470**	**470**
		4	410	**413**	411
		5	**473**	**473**	**473**
		6	478	**482**	**482**
		7	455	**456**	**456**
		8	454	454	**458**
		9	470	470	**471**
		10	490	**492**	**492**
	24	1	**488**	**488**	**488**
		2	**428**	**428**	**428**
		3	**454**	449	**454**
		4	407	**408**	**408**
		5	464	**465**	**465**
		6	**469**	468	**469**
		7	**455**	**455**	**455**
		8	**448**	**448**	**448**
		9	461	**462**	**462**
		10	**477**	**477**	**477**
	27	1	**471**	**471**	**471**
		2	406	**407**	**407**
		3	**432**	429	**432**
		4	**403**	**403**	**403**
		5	**455**	**455**	**455**
		6	**456**	**456**	**456**
		7	**444**	**444**	**444**
		8	**438**	**438**	**438**
		9	**456**	**456**	**456**
		10	**467**	**467**	**467**
# Best			33	**78**	76

Table 7.1: Results for n = 30

Therefore, let us focus our attention on Math. and ILS, by performing a one-by-one comparison of every result of both approaches for all instances and k_s. The overall win-lose-draw rate for Math. is 13%-14%-72%. A very high draw rate is achieved, pointing that optimality is probably reached by both methods.

Moreover, before arriving to other conclusion we should remember that, as showed in section 6.4.1, the pure matheuristic part of the Math. algorithm is not able to improve the ILS result for $k \leq 12$. Contradictorily, we can find on Figure 7.1 some cases where $k \leq 12$ and Math. reaches better solutions than ILS. This is due to the fact that those results were actually achieved by the ILS part of the Math. algorithm. Sometimes, the random perturbation allows to find better results in less time while running ILS. Considering this fact, and by appropriately clustering k depending on which algorithm performs better (showed in Figure 7.2), we can conclude that:

- ✓ For $k \leq 18$: ILS performs better than the matheuristic.
- ✓ For $k \geq 21$: matheuristic performs slightly better than ILS.

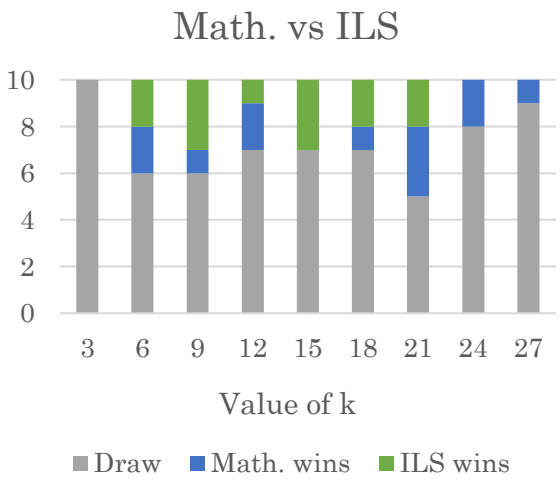

Figure 7.1: Math. against pure ILS, n = 30

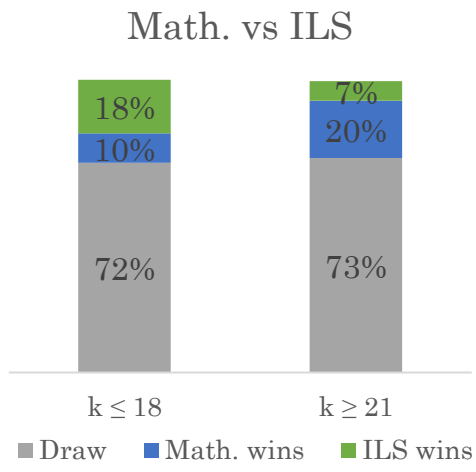

Figure 7.2: Comparison with clustered ks

7.3. Results for n = 50

Again, we set the same time limit for the three approaches, in this case of 600 seconds. Table 7.2 presents the corresponding results for $n = 50$. We can observe that it is confirmed that the MIP approach is outperformed.

Size	k	Instance	MIP	ILS	Math.
50x50	5	1	379	**399**	397
		2	374	402	**403**
		3	382	**399**	**399**
		4	378	**396**	**396**
		5	352	**405**	402
		6	384	**387**	**387**
		7	373	**402**	**402**
		8	395	**399**	**399**
		9	374	**403**	**403**
		10	376	**386**	**386**
	10	1	512	**529**	528
		2	532	**544**	**544**
		3	514	552	**560**
		4	**530**	518	524
		5	507	543	**544**
		6	520	**522**	**522**
		7	**543**	**543**	**543**
		8	558	**575**	**575**
		9	500	**539**	537
		10	502	**528**	**528**
	15	1	528	**570**	569
		2	577	**605**	**605**
		3	556	**595**	590
		4	548	565	**567**
		5	553	**612**	**612**
		6	557	**579**	**579**
		7	550	**560**	559
		8	595	**614**	**614**
		9	559	**589**	586
		10	559	**571**	570
	20	1	580	589	**591**
		2	620	**632**	**632**
		3	613	**614**	613
		4	563	**581**	**581**
		5	622	**646**	**646**
		6	**604**	603	599
		7	552	**566**	549
		8	618	**627**	626
		9	609	604	**611**
		10	587	**591**	590
	25	1	583	**594**	589
		2	622	621	**626**
		3	601	**607**	592
		4	560	**577**	571
		5	608	635	**637**
		6	588	**608**	**608**
		7	572	**578**	576
		8	586	**636**	633
		9	620	**637**	631
		10	**595**	**595**	590
	30	1	606	611	**612**
		2	635	633	**636**
		3	598	614	**615**
		4	577	590	**591**
		5	645	**647**	643
		6	607	**621**	**621**
		7	578	**583**	556
		8	622	639	**644**
		9	631	640	**643**
		10	587	**595**	583
	35	1	617	620	**623**
		2	637	**650**	649
		3	612	626	**627**
		4	598	597	**601**
		5	641	**655**	644
		6	613	**620**	**620**
		7	583	569	**585**
		8	627	**635**	621
		9	643	649	**651**
		10	596	**604**	599
	40	1	621	**622**	**622**
		2	**655**	649	**655**
		3	627	623	**628**
		4	602	**612**	**612**
		5	**659**	655	655
		6	621	621	**622**
		7	578	**590**	576
		8	628	**635**	628
		9	652	**661**	**661**
		10	607	**613**	**613**
	45	1	629	**630**	629
		2	**652**	651	**652**
		3	**631**	629	630
		4	**610**	**610**	**610**
		5	**661**	**661**	**661**
		6	623	623	**624**
		7	**593**	**593**	589
		8	**633**	**633**	**633**
		9	657	**660**	**660**
		10	**625**	623	**625**
# Best			13	**61**	55

Table 7.2: Results for n = 50

Therefore, let us carry out a comparison between Math. and ILS, in the same way as in the previous section. In this case, the overall win-lose-draw rate for the hybrid approach was 30%-36%-34%, a quasi-uniformly distributed result. We notice that the draw rate considerably decreased compared to the one of $n = 30$, probably because optimality is more difficult to achieve for these larger instances.

Moreover, in Figure 7.3 we observe that for $k \leq 25$ Math. outperforms ILS nine (out of fifty) times. However, in only one of those cases the credit can be given to the matheuristic part of Math. algorithm, since the other eight times the final solution of Math. is the one given by the ILS part of that algorithm. Then, we can cluster the values of k just as showed in Figure 7.4, and we can conclude that:

- ✓ For $k \leq 25$: pure ILS performs better than the Math.
- ✓ For $k \geq 30$: the matheuristic approach performs better than ILS.

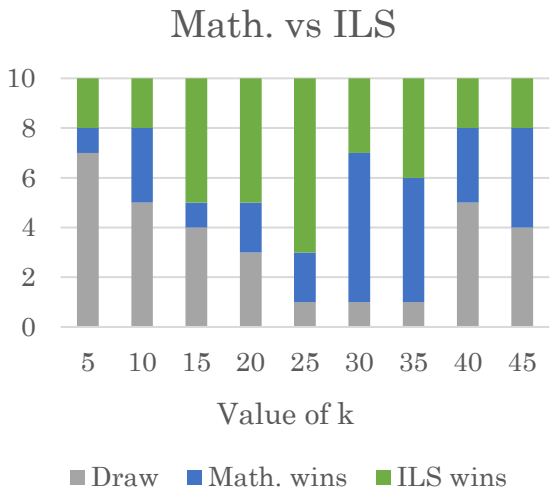

Figure 7.3: Math. against pure ILS

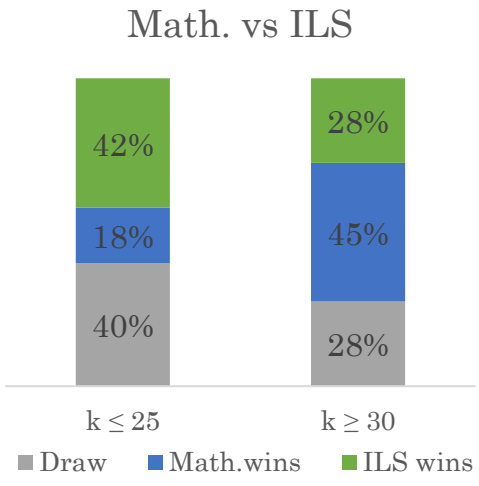

Figure 7.4: Comparison with clustered ks

7.4. Conclusion

To sum up, this thesis presented different approaches to solve the BAP. Each of them was improved as much as possible in the corresponding chapter, by the use of relaxed models, by performing a neighborhood search following a determined order, by constricting upper bounds to the AP solution, by developing adaptive algorithms, etc.

After an extensive comparison that was done in this chapter, we can conclude that the MIP approach demonstrated to have the worst performance. Moreover, we can state that a pure ILS is better for small values of k, while a hybrid approach is preferred for bigger k_s.

8. Bibliography

[1] H. Kuhn, "The Hungarian Method for the Assignment Problem", *Naval Research Logistics Quarterly,* vol. 2, pp. 83-97, 1955.

[2] Behdad Beheshti, Oleg A. Prokopyev, Eduardo L. Pasiliao, "Exact solution approaches for bilevel assignment problems", *Springer Science + Business Media New York,* pp. 215-242, 2015.

[3] Papadimitriou, C. H., K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, New York: Prentice Hall, 1998.

[4] T. C. Hu, Andrew B. Kahng, Linear and Integer Programming Made Easy, San Diego: Springer, 2016.

[5] David G. Luenberger, Yinyu Ye, Linear and non-linear programming, Stanford: Springer, 2016.

[6] L. Trevisan, *Combinatorial Optimization: Exact and Approximate Algorithms,* San Francisco: Standford University, 2011.

[7] Jay-Kumar Sundararajan, "Advanced Algorithms Open Course. Lecture notes no. 14: Duality", Massachusetts Institute of Technology, Cambridge, 2004.

[8] Goldreich Oded, Computational complexity: A Conceptual Perspective, New York: Cambridge University Press, 2008.

[9] Elisabeth Gassner, Bettina Klinz, "The computational complexity of bilevel assignment problems", *Springer-Verlag,* pp. 379-394, 2009.

[10] R. O. Singh, S. H. Takhimiya, C. S. Pimpale, "The Demonstration of Assignment Problem for Undergraduate Engineering Study", *International Journal of advanced research in Computer Science and Management Studies,* vol. 2, no. 8, pp. 30-34, 2014.

[11] Lyle Ramshaw, Robert E. Tarjan, "On minimum-cost assignments in unbalanced bipartite graphs," HP Laboratories, 2012.

[12] Ronald D. Armstrong and Zhiying Jin, "Solving linear bottleneck assignment problems via strong spanning trees", *Operations Research Letters,* vol. 12, no. 3, pp. 179-180, 1992.

[13] Eranda Çela , The Quadratic Assingment Problem: theory and algorithms, Springer-US, 1998.

[14] Colin F. Palmer, Alexander E. Innes, Operational Research by example, London: Macmillan Education UK, 1980, pp. 121-141.

[15] *Operations Research and Management Decision course, Unit 1 Lesson 19 Assignment Problem,* Institute of Public Enterprise, Hyderabad.

[16] Ankur Sinha, Pekka Malo, Kalyanmoy Deb., "A Review on Bilevel Optimization: From Classical to Evolutionary Approaches and Applications", *IEEE,* vol. 22, no. 2, pp. 276-295, 2017.

[17] P. Hansen, B. Jaumard, "New branch-and-bound rules for linear bilevel programming", *SIAM Journal on Scientific and Statistical Computing,* vol. 13, no. 5, p. 1194–1217, 1992.

[18] Xiaotie Deng, "Complexity issues in bilevel linear programming", in *Multilevel Optimization: Algorithms and Applications*, Boston, Springer, 1998, pp. 149-164.

[19] Peter A. Clark, Arthur W. Westerberg, "Bilevel programming for steady-state chemical process design-i. fundamentals and algorithms", *Computers & Chemical Engineering,* vol. 14, no. 1, pp. 87-97, 1990.

[20] S. V. Marco Caserta, "Metaheuristics: Intelligent Problem Solving", in *Matheuristics: Hybridizing Metaheuristics and Mathematical Programming*, Boston, Annals of Information Systems, vol 10. Springer, 2010, pp. 1-38.

[21] Tim Roughgarden, "Part 3: Greedy Algorithms and Dynamic Programming", in *Algorithms Illuminated*, New York, NY, Soundlikeyourself Publishing, LLC, 2019, pp. 57-94.

[22] S. M. Johnson, "Optimal two- and three-stage production schedules with setup times included", *Naval Research Logistics (NRL),* Vols. P-402, pp. 1-10, 1954.

[23] W. Bednorz, Advances in Greedy Algorithms, Vienna: In-Teh, 2008.

[24] N.Mladenović, P. Hansen, "Variable neighborhood search", *Computers and Operations Research,* vol. 24, no. 11, pp. 1097-1100, 1997.

[25] Editors: Michel Gendreau, Jean-Yves Potvin, Handbook of Metaheuristics, Springer, 2019.

[26] H. R. Lourenço, Olivier C. Martin, and Thomas Stützle, "Iterated Local Search: Framework and Applications", in *Handbook of Metaheuristics*, Springer, 2018, pp. 129-168.

[27] J. Puchinger and G.R. Raidl, "Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification", in *First International Work-Conference on the Interplay Between Natural and Artificial Computation*, Berlin, 2005.

[28] Maniezzo Vittorio, Stützle Thomas, Voß Stefan, "Usage of Exact Algorithms to Enhance Stochastic Local Search Algorithms", in *Matheuristics. Annals of Information Systems, vol 10*, Boston, Springer, 2010, pp. 103-134.

[29] Maniezzo Vittorio, Stützle Thomas, Voß Stefan, "MetaBoosting: Enhancing Integer Programming Techniques by Metaheuristics", in *Matheuristics. Annals of Information Systems, vol 10*, Boston, Springer, 2010, pp. 71-102.

[30] C. Caldwell, "Graph Theory Glossary", University Tennessee at Martin, 1995. [Online]. Available: https://primes.utm.edu/graph/glossary.html.

[31] Hoffman, A.J., Kruskal, J, "Introduction to Integral Boundary Points of Convex Polyhedra", in *50 Years of Integer Programming 1958-2008*, Berlin, Springer, 2009, pp. 49-76.