A Multi-Period Facility Location Problem

Carlos Sández García

Mathematics Master Thesis

Universidad de Zaragoza

Advisors:

Herminia I. Calvete and Pedro M. Mateo

September 2018

Prologue

In this work we introduce a novel evolutionary algorithm for the Multi-Period Incremental Service Facility Location Problem. A computational study is carried out to showcase the good results produced by such an heuristic method. The significance is that, to the authors' knowledge, no research has been carried out on similar multi-period problems with genetic or evolutionary algorithms.

The structure of the work is as follows:

Chapter 1 introduces the field of Location Science along with examples of Facility Location problems. This culminates with the Multi-Period Incremental Service Facility Location Problem, a time-bound model for the incremental coverage of demand of non-essential services.

Chapter 2 deals with genetic and evolutionary algorithms, reviewing the mechanisms available in the literature for single-period facility location problems.

Chapter 3 presents the evolutionary algorithm developed to solve the Multi-Period Incremental Service Facility Location Problem. The operators that make up the algorithm are explained and some examples given. The pseudocode is also made available.

Chapter 4 summarizes the results from two computational studies, showing the performance of the algorithm in a wide array of problems. Comparisons to the exact solving and an artificially decoupled model are also reported.

Appendix A contains the code of the algorithm implemented in R.

Resumen

En este trabajo se presenta un novedoso algoritmo evolutivo para resolver el Problema Multi-Periodo de Localización de Instalaciones con Oferta Incremental (Multi-Period Incremental Service Facility Location Problem). Tras un estudio computacional detallado, se demuestra que la calidad de los resultados es suficiente para justificar el uso de este método heurístico. Esto es significativo porque no parece haber artículos sobre la resolución de problemas multi-periodo similares con algoritmos genéticos o evolutivos.

La estructura del trabajo se muestra a continuación:

En el capítulo 1 se introduce el campo de la Ciencia de Localización (Location Science) con ejemplos de problemas de localización de instalaciones. Esto culmina con el Problema Multi-Periodo de Localización de Instalaciones con Oferta Incremental, un modelo dependiente del tiempo para cubrir incrementalmente la demanda de servicios no esenciales.

El capítulo 2 trata sobre los algoritmos genéticos y evolutivos. Se repasan los mecanismos para problemas de localización de instalaciones de un solo periodo que aparecen en la literatura.

En el capítulo 3 se presenta el algoritmo evolutivo desarrollado para resolver el Problema Multi-Periodo de Localización de Instalaciones con Oferta Incremental. Se explican los operadores que forman el algoritmo y se dan algunos ejemplos. También está disponible el pseudocódigo.

En el capítulo 4 se resumen los resultados de dos estudios computacionales, mostrando la eficacia del algoritmo en un abanico de problemas. También se comparan con la resolución exacta del modelo y el obtenido al optimizar periodo a periodo.

En el apéndice A está el código del algoritmo implementado en R.

Contents

Chapter 1

Introduction to Facility Location Problems

Location Science is a discipline concerned with finding the optimal spatial arrangement of facilities that supply some demand. Facility location problems (FLP) are very much part of this area. In FLP, facilities have to be located in space. Facility is a generic term that can represent anything from logistic and distribution centers to hospitals or kindergartens. The point is that a service is provided by the facility, which is used by some demand nodes. The distribution of the facilities should be optimal in the sense that a certain goal must be reached. Maybe a certain supply level is needed, or we want to minimise the sum of the distances between demand nodes and facilities. It is clear how grounded in reality this problem is, and the many applications in fields such as economics or logistics that it has. We are concerned with a specific instance of FLP, but will nonetheless introduce the problem progressively, motivating all the increases in complexity and understanding what has come before. This introduction will follow that of the book by Nickel and da Gama [21, §11].

1.1. Origins of Location Science

The first Location Science problem as it is known today is that of finding the point in Euclidean space that minimises the sum of the distances to three other points. This has been attributed to Pierre de Fermat and thus setting the beginning of Location Science in the 17th Century. Using the modern terminology, there are three demand nodes and one facility which can be located anywhere on $\mathbb{R}^2$ but must minimise the sum of distances. Fermat's problem was solved geometrically in [16] for all cases.

A more general approach and indicative of the gradual complexity to come in modern Location Science was due to Carl Friedrich Launhardt. In [22] he added weights to the demand nodes of the original problem. With this generalisation, the problem is known as a 3-node Weber problem. Alfred

Weber would later on study these kind of problems [36] and thus lead to the modern terminology.

1.2. The p-median Problem

The concept started in the Weber problem was generalised in the form of the p-median problem. Median because the sum of the distances between demand nodes and their respective closest facility must be minimised. The p-median problem was presented in the paper by Cooper [8]. However, it was Hakimi in [14] who proved that there exists an optimal location of facilities over the demand nodes. This reduces the location space from the continuous considered in the Weber problem to a discrete one. Thanks to the location space being now discrete, a mixed-integer linear programming (MILP) formulation for the p-median was introduced in [29]. Kariv and Hakimi further proved in [20] that the p-median problem is NP-hard, that is, there is no known non-deterministic polynomial-time algorithm that solves the p-median problem.

A formulation for the p-median problem can be as follows. Let I be the set of possible locations of the facilities and J be the set of demand nodes. Let d_j be the demand of customer j, c_{ij} be the unit cost of satisfying customer j from facility i, p be the number of facilities to be located, x_{ij} be the fraction of demand that facility i supplies to j and y_i be a binary variable with value 1 when the facility i is located.

$$\min \quad \sum_{i\in I}\sum_{j\in J} d_j c_{ij} x_{ij} \tag{1.1}$$

$$\text{subject to} \quad \sum_{i\in I} x_{ij} = 1, \quad \forall j \in J \tag{1.2}$$

$$\sum_{i\in I} y_i = p \tag{1.3}$$

$$x_{ij} \leq y_i, \quad \forall i \in I, j \in J \tag{1.4}$$

$$y_i \in \{0,1\}, \quad \forall i \in I \tag{1.5}$$

$$x_{ij} \geq 0, \quad \forall i \in I, j \in J \tag{1.6}$$

There is another kind of FLP where the objective is to locate facilities so as to minimise the maximum distance between any demand node and its closest facility. These are called p-center problems for the instance where p facilities must be located. These kinds of problems are tied to the supply of emergency services, where the worst case scenario needs to be alleviated. The concept of center was first introduced in [14]. We will focus on median-based problems.

1.3. Dynamic Facility Location Problems

The first problem to consider dynamic constraints was by Wesolowsky [37]. This Multi-Period Weber problem extends the classical Weber so as to encompass changes in the demand distribution along some time periods. The difference between this model and the previous one is that moving the facility incurs a relocation cost, and thus cannot be understood as several different Weber problems.

After this problem, an additional constraint is imposed in [7], where at most one new facility can be located in each time period. This was done for a special case of FLP where the location space is a graph, but the idea is analogous to the Weber or discrete formulation.

Finally, in [10] a Multi-Period Weber problem where exactly one new facility must be located each time period was introduced. Notice how, in this case, the p-median problem must assign the facilities throughout the p time periods. This problem is the predecessor of the incremental service FLP that will be considered later on.

1.4. Multi-Period Facility Location Problems

The class of Multi-Period Facility Location problems (MPFLP) is a generalisation of the Facility Location problem where a time dependency is incorporated in the form of a multi-period setting. As it has been seen in the previous section, considering dynamic FLP changes the problems fundamentally. The inclusion of time as another dimension for the model affects the structure of the problem, requiring a global overview to solve it. Time-dependent models appear naturally when considering some dynamic decisions, such as inventory management, opening and closing of facilities and capacity changes, as well as other questions arising from budgetary constraints [21].

In FLP, the time frame in which the decisions need to be carried out is the *planning horizon*. Within this planning horizon, many strategic and tactical choices have to be carried out, with long-lasting consequences. The planning horizon can be either finite or infinite, but most of the research is focused on finite planning horizons and ours will too. Even though the time can be considered as continuous, as noted in [21], optimal control is a better alternative to solving continuous-time instances. Thus, the time setting considered in this study will be a finite series of discrete time periods, ranging from the initial one to the planning horizon. As for the spatial setting where the facilities and customers are allocated, three distinct approaches appear in the literature. We will give a basic introduction of the first two, and focus on the third one, which shall be chosen.

Solving for the decoupled model is generally an incorrect approach, as finding the consecutive best arrangement for the individual time periods misses the larger picture and forces the structure to mistakes that could have been foreseen. The qualitatively different structures surfacing from these two

approaches has been studied in [2], concluding that decoupled models tend to give good objectives for the first periods, with solutions deteriorating quickly as periods pass.

1.4.1. Introducing Multi-periodicity

Some measures of introducing multi-periodicity in a given formulation of FLP have already been given. Setting relocation costs, not allowing for facilities to close or setting opening costs all transform an initially decoupled system into a time-bound one. These changes in formulation appear naturally, and are even a necessity when modeling the operation of large companies.

Extensions on the constraints about opening and closing facilities include opening facilities for a minimum period of time, opening a minimum number of facilities each period [2] and only allowing for a subset of facilities to close [30]. Changes in operating capacity can also be implemented, either by reducing or increasing the capacity, which was done for the MPFLP in [34]. These are usually accompanied by economies of scale, where bulk discounts are given based on production volume. Other extensions consider multiple products and multiple stages of production, as shown in [17].

Introducing time dependencies can have a big impact on a problem. An approach for quantifying the influence of time in any MPFLP model is to measure the difference between the exact solution and the time-invariant solution from the static model. The static model has to be artificially constructed, unifying the time-dependent constraints into others that do not depend on time but still yield feasible solutions for the original model. The specific way to do it depends on the MPFLP formulation, but can be contructed by averaging demands. The *value of the multi-period solution*, as defined in [4], is given by the difference between the time-invariant *weak* solution and the *strong* multi-period solution.

1.4.2. Continuous Problems

Continuous problems come from the use of a continuous location space. The following multi-period extension of the Weber problem was proposed in [37] and allowed for the change in the set of nodes in every time period, giving a relocation cost for the factory changing position. This problem can be formulated as follows:

$$\min \quad \sum_{t\in T}\sum_{j\in J_t} c_{tj}(x_t,y_t) + \sum_{t=2}^{|T|} f_t z_t \tag{1.7}$$

$$\text{subject to} \quad z_t = 0 \text{ if } d_{t-1,t} = 0, \quad t \in T \setminus \{1\} \tag{1.8}$$

$$z_t \in \{0,1\}, \quad t \in T \tag{1.9}$$

The objective (1.7) minimizes both the allocation and relocation costs, denoted by c_{tj} and f_t, respectively. The variables used are (x_t, y_t), the euclidean coordinates of a facility in that location in period t;

z_t, a binary variable that indicates whether the facility has changed location; and $d_{t',t}$, the distance by which it moves. The sets T and J_t are the time periods and demand nodes in that period.

Methods for solving this sort of problems started out as dynamic programming and quickly became exact ones [35].

Other multi-period, multi-facility approaches are proposed in [11], which considered the optimal time periods to take as well, the recursive installation of a new facility every period in [33], and an extension of the planar p-median problem in [10].

1.4.3. Network Problems

Network problems encompass those where the facilities are located on a path or tree, at least one per period. The first extension of the network p-median problem considers a weighted network that varies in time with assignment and relocation costs [15]. This paper also studied the 1-center multi-period problem on a network. Some theoretical results hold, which reduce the location space of these kind of problems. However, not much research is still being carried out in this topic since these problems can be reduced to discrete ones [21].

1.4.4. Discrete Problems

Discrete problems are the predominant choice for location space [15]. The p-median problem in a discrete location space is as follows.

Let c_{ijt} be the allocation cost of demand node $j \in J$ to facility $i \in I$ in time period $t \in T$ and x_{ijt} be a binary variable equal to 1 if in time period t facility i is allocated to demand node j. Here, it is considered that facilities must be located on demand nodes, thus $I \subseteq J$. Also, the demand of a customer can only be satisfied by one facility.

$$\min \quad \sum_{t\in T}\sum_{i\in I}\sum_{j\in J} c_{ijt}x_{ijt} \tag{1.10}$$

$$\text{subject to} \quad \sum_{i\in I} x_{ijt} = 1, \quad t \in T, j \in J \tag{1.11}$$

$$\sum_{j\in J} x_{ijt} \leq |J|x_{iit}, \quad t \in T, i \in I \tag{1.12}$$

$$\sum_{i\in I} x_{iit} = p, \quad t \in T \tag{1.13}$$

$$x_{ijt} \in \{0,1\}, \quad i \in I, j \in J, t \in T \tag{1.14}$$

The position of demand nodes is fixed, whereas in 1.4.2 the weights change represent physical displacement of the demand nodes. This amounts to a fixed set of locations for the facilities, which are such that

$I \subseteq J$. No generality is lost, however, since the cost for allocating demand nodes c_{ijt} from facility i to node j in period t can be set as some penalized value where the original j was $j \notin J_t$.

Since no reallocation costs are considered, the p-median problem can be decoupled. A simple modification of the objective function introduces the multi-periodicity. In [38], opening and closing costs are introduced, making the decoupled problem not optimal in general. Further extensions and methods for solving this problem have been developed, since the original dynamic programming approach is only usable for small problems.

When no capacity constraints are considered, that is, there is no limit to the amount of demand a given facility can satisfy, the problem is called *uncapacitated*. In the Uncapacitated Facility Location Problem (UFLP), the demand is allowed to be divided among several facilities. Let x_{ijt} be the fraction of demand supplied to customer j by facility i in period t, and y_{it} be a binary variable that indicates whether facility i is operating at time t or not. Additionally, let f_{it} be the cost of operating facility i in period t.

$$\min \quad \sum_{t\in T}\sum_{i\in I}\left(f_{it}y_{it} + \sum_{j\in J} c_{ijt}x_{ijt}\right) \tag{1.15}$$

$$\text{subject to} \quad \sum_{i\in I} x_{ijt} = 1, \quad t \in T, j \in J \tag{1.16}$$

$$\sum_{j\in J} x_{ijt} \leq |J| y_{it}, \quad t \in T, i \in I \tag{1.17}$$

$$x_{ijt} \geq 0, \quad i \in I, j \in J, t \in T \tag{1.18}$$

$$y_{ijt} \in \{0,1\}, \quad i \in I, j \in J, t \in T \tag{1.19}$$

Notice how the capacity of a given factory is bounded by $|J|$ when that factory is operative. This implies that the problem is indeed uncapacitated.

Again, the objective is to minimize the running and assignment costs. The model can be decoupled, since there are no constraints that make it time-dependent. Introducing a natural relocation cost, or forcing facilities to stay open takes care of that.

Models based on facilities which cannot be closed have been successfully modeled using step variables, that indicate whether a new facility opens in a given period. This, along with further supply restrictions, has been studied in [6].

1.5. The Multi-Period Incremental Service Facility Location Problem

The Multi-Period Incremental Service Facility Location problem (MISFLP) was first introduced by Albareda-Sambola et al. in [2] and is concerned with the operation of non-essential services. In most of the previous literature, the facilities considered supplied an essential service that had to cover the available demand. If, instead, a private or non-essential service is considered, the supply can be incrementally increased by opening new facilities each period and thus increasing the demand coverage until its completion in the planning horizon. The examples provided in the original paper include libraries, nursing homes, kindergartens, parking lots, supermarkets and banks. The MISFLP is the subject of this work, and is presented in depth below.

Let J be the set of customers, I be the set of possible facility locations and T be the set of periods. Every period, a minimum of n^t customers have to be served and at least p^t new facilities must open. Once opened, facilities cannot close before the end of the planning horizon. Also, customers that have their demand satisfied on a given period must also receive the supply every period after. Partial coverage of a customer's demand is not allowed.

The costs are as follows. Let c_{ij}^t be the allocation cost of assigning customer j to facility i at time period t and f_i^t be the cost for opening facility i in time period t. The cost f_i^t includes the initial opening cost in period t and the maintenance cost until the end of the planning horizon. The facilities that supply a given customer's demand may change over time at no extra cost.

Even though full coverage of demand is expected at the end of the planning horizon, partial coverage can be implemented by introducing a dummy facility with infinite costs in all periods except for the last. Unsatisfied demand penalties could also be introduced but are not considered in this model.

In [1], several ways of formulating the MISFLP were proposed and compared to find the strongest one. The mathematical formulation given below is the one with the best results as per [1] and not the one presented originally in [2]. It will, however, not incorporate the modifications in the problem that [1] has over [2], namely customer demand in select periods, service costs and penalties for unfulfilled demand. Let us define

$$x_{ij}^t = \begin{cases} 1 & \text{if at period } t \text{ customer } j \text{ is assigned to facility } i \\ 0 & \text{otherwise} \end{cases}$$

$$\bar{y}_i^t = \begin{cases} 1 & \text{if by period } t \text{ the facility } i \text{ has already opened} \\ 0 & \text{otherwise} \end{cases}$$

With a slight abuse in notation, $\bar{y}_i^{-1} = 0$.

The formulation of the MISFLP is as follows:

$$\min \quad \sum_{i\in I}\sum_{t\in T}\left(f_i^t(\vec{y}_i^t-\vec{y}_i^{t-1})+\sum_{j\in J}c_{ij}^t x_{ij}^t\right) \tag{1.20}$$

$$\text{subject to} \quad \sum_{i\in I}\sum_{j\in J}x_{ij}^t \geq n^t, \quad t\in T \tag{1.21}$$

$$\sum_{i\in I}x_{ij}^t \leq 1, \quad t\in T, j\in J \tag{1.22}$$

$$\sum_{i\in I}x_{ij}^t \geq \sum_{i\in I}x_{ij}^{t-1}, \quad j\in J, t\in T\setminus\{0\} \tag{1.23}$$

$$\sum_{i\in I}x_{ij}^{|T|} = 1, \quad \forall j\in J \tag{1.24}$$

$$x_{ij}^t \leq \vec{y}_i^t, \quad \forall i\in I, j\in J, t\in T \tag{1.25}$$

$$\sum_{i\in I}(\vec{y}_i^t-\vec{y}_i^{t-1}) \geq p^t, \quad \forall t\in T \tag{1.26}$$

$$\vec{y}_i^{t-1} \leq \vec{y}_i^t, \quad \forall i\in I, t\in T \tag{1.27}$$

$$x_{ij}^t, \vec{y}_i^t \in \{0,1\}, \quad i\in I, j\in J, t\in T \tag{1.28}$$

The objective function (1.20) minimizes the setup and maintenance costs of the facilities, as well as the assignment costs. The minimum customers served and facilities opened per period come from the constraints (1.21) and (1.26), respectively. Each customer is assigned to at most 1 facility in (1.22) and once fulfilled, keeps on having its demand satisfied by (1.23). The full demand coverage at the end of the planning horizon is ensured by (1.24). Customers are assigned to open facilities by (1.25). Constraint (1.27) forces opened facilities to stay open until the end of the planning horizon.

One absence from these constraints is any kind of capacity constraints. For all it matters, the model could assign all production to a given facility while opening and not using the others, as long as it is economically viable. The issue with constraints in production capacity is that they greatly increase the difficulty of the problem, and are only considered if required.

For $|T| = 1$, the MISFLP is equivalent to the p-median problem, which was proven in [20] to be NP-hard. This means that the MISFLP is NP-hard as well, and heuristic methods are required to solve large instances of the problem.

Chapter 2

Evolutionary Algorithms

2.1. Introduction to Evolutionary Algorithms

Evolutionary Algorithms (EA) are a kind of metaheuristic that takes inspiration from nature to achieve near-optimal solutions relatively quickly on a wide variety of problems. The resemblance to natural evolution comes in the form of an evolving population of solutions to the studied problem. Each solution is encoded in such a way that it corresponds to an individual with some activated genes. The individuals of the population have their quality measured by a fitness function, the objective being to optimize the fitness so as to find the best possible individual. The population will be subject to several crossover and mutation operators and their results will be selected to be part of the population again based on a selection procedure. This is carried out until a stop criterion is met.

Evolutionary processes were described for the first time in the articles by Friedberg in 1958 [12]. The precursors of Evolutionary Algorithms, Genetic Algorithms (GA), were initially developed by Holland in the 1960s as seen in [18]. They were applied for the first time to the p-median problem in the paper by Hosage [19]. Up until then, p-median problems had been mainly solved by exact methods. However, as the complexity of the studied instances increases, exact methods have an exponentially harder time solving them. This is a direct consequence of the p-median problem being NP-hard [20].

An heuristic for such problems is then desirable, because it might obtain good enough solutions much more quickly. With heuristics, the tradeoff is the loss of certainty of optimality. No longer we know that the obtained solution is the best one. However, for many purposes, the time requirements of exact methods are prohibitive and a close enough solution suffices. Common variants of operators used in evolutionary algorithms applied to the p-median problem are presented in this section

For a review on metaheuristic procedures for the p-median problem, the reader is pointed to the paper by Mladenovic [27].

2.2. Encoding

The individuals in GA are traditionally represented by a fixed-length binary string. This is the encoding chosen in [19]. However, this first application didn't achieve very good results in the p-median problem, as noted by Alp et al. [3]. The issue is that, with the binary representation, most of the possible configurations are unfeasible since more or less than p facilities are located. Because of this, the search space of the algorithm is huge and not representative of the problem. GA with binary encoding must constantly fight this tendency to unfeasibility and requires methods such as restricted search [31]. There are some instances where specifically choosing a binary encoding can be helpful, such as in [26], but most of the research in GA for p-median problems uses the encoding proposed by Alp et al. [3] and we will too.

This encoding assumes that the facilities are assigned to some index, so that $\{1,\dots,m\}$ represents all the facilities. Out of the possible m facilities, p have to be chosen. The natural expression of which are chosen is a tuple of length p with increasing elements in $\{1,\dots,m\}$. The strength of this encoding is that all possible assignments of facilities have a one-to-one relation with all such possible encodings. No longer we have to worry about unfeasibility, and operations can follow much more quickly. This also reduces the space of all possible individuals from 2^m to $\binom{m}{p}$.

2.3. Initialization

The starting population has to be constructed before the algorithm can proceed. A common method is by random generation. An individual would be created by selecting a random subset of p element from the indices. Since the random distribution used is uniform, for sufficiently large population sizes, this would be a quick way of ensuring population diversity.

A more comprehensive approach proposed by Alp et al. [3] is sequential initialization. The individuals will be constructed initially by going over the indices in increments of 1 until they run out and continuing in increments of 2 until all indices are chosen. This continues until the increments are some pre-specified $k < m$. This list is then separated in strings of consecutive p facilities. The example given in [3] is $(m,p,k) = (12,4,2)$, which yields the population

$$(1,2,3,4),(5,6,7,8),(9,10,11,12),(1,3,5,7),(2,4,9,11),(6,8,10,12).$$

Another option is to find some good solutions via local search methods or other quick heuristics and start the population with those. The issue here is that diversity is not enforced. To circumvent this, in [23] the initial good solutions are added to a greedy-generated population which is diversified. This *seeding* of good solutions does require additional heuristics, but speeds up computation times.

2.4. Crossover

The crossover emulates the reproduction process in nature and the exchange of genetic information. This operator has been adapted to the p-median problem in several ways. All of them have to deal with the fact that feasibility should be preserved. Otherwise, operators may result in individuals with less facilities than necessary or repeated facilities. Operators that work with these situations are called *messy*, but are not commonly used [13].

One intuitive crossover operator is the one proposed by Correa et al. in [9]. After individuals have been chosen for the crossover phase, two *exchange vectors* are constructed so that they contain the distinct facilities of the individuals. A random number k between 1 and the length of the exchange vectors minus 1 is chosen. After randomly choosing k facilities from each exchange vector, those facilities are swapped in the individuals. Notice how feasibility is preserved, since the length is kept at p and no repeated facilities are present in an individual. The idea of exchange vectors motivated the mechanic of permuting facilities, the process by which facilities opened on different periods switch places in the individual.

Another operator that is computationally more expensive but gains some sense of optimality direction via greediness is the one presented in [3]. This merge-drop algorithm is more closely related to greedy heuristics than classical GA crossover. The facilities in the chosen parents are all considered. The result is in general not feasible, since the union will likely be greater than p. Facilities are excluded one by one and the fitness evaluated. The subset with one less facility that has the lowest fitness is chosen, and the omitted facility is dropped. This is repeated until enough facilities have been removed and the individual is feasible. The cost is that many evaluations of fitness have to be done. Moreover, the fitness function must allow for the evaluation of non-feasible solutions. Good results are reported using this method, and it is recommended for inexpensive fitness functions.

A crossover operator where the parents have different roles is presented in [28]. The operator takes into account the closeness of the facilities in the individuals and substitutes the selected facilities in the first one by the nearest facility from the other one. Assigning different roles to the parents was also used in the final version of the EA.

Three more classical crossover operators are presented in [5]. An interesting case of operators being dynamically chosen based on population diversity to avoid local optima is sketched in [24]. This paper also extends on the greedy crossover of [3] by presenting a modified increasing vector where the distinct facilities that increase the fitness the least are progressively added. In [25], a dynamic crossover based on the diversity of the population is proposed, which allows for the increasing or decreasing of the population size throughout the iterations.

2.5. Parent Selection

Now the measures to select parents subject to the crossover operator will be discussed. On many occasions they are just chosen randomly and that produces good results, but other procedures exist. For instance, a simple elitist selection was studied in [5]. Parents that had better fitness were more likely to be chosen, but the choice was still probabilistic. This skewness was formulated in two different versions. The first one accounted for the fitness function, and a normalized value was assigned as weights for the skewed random selection. The other one took the fitness into account only indirectly, since it was the ranking that formed the weights. The population was ranked by fitness, and higher-ranking individuals were more likely to be chosen. The issue with this approach was that similarly-fit individuals may have very different rankings. It might help distinguish between very flat fitness functions, but better results were observed with the first approach, as noted in [5]. This fitness-based elitist selection was also used in the paper which proposes the first GA for solving the p-median problem [19].

A slightly more complex version of this kind of parent selection is the one presented in [32]. The fitness function values were stretched so as to emphasize the difference between similar individuals but without going all-out on a ranking system. After this, the individuals were grouped by fitness. Individual in the same group were equally likely to be selected.

Partially due to the unique encoding chosen for the bionomic algorithm in [26], a unique parent selection technique was chosen. It is more deterministic than the other random-based approaches that we have seen. The idea is to select pairs of parents that are as different from each other as possible. This is measured by maximal independent sets. Essentially, it supercharges exploration of the search space and avoids many local optima traps. Some local search is necessary on the later stages of GA and the authors solved this problem with an additional maturation operator based on greedy heuristics.

From the wide variety of methods available in the literature, it is clear that there is no catch-all approach and that a thorough study of the problems to solve is necessary when dealing with GA [3, §3].

2.6. Mutation

As for the mutation, the usual procedure is to remove some facilities and substitute them for unused ones. A hypermutation is introduced in [9], which is equivalent to a partial maturation. A maturation is a local search on the solution space and has been used in [26] too. Maturation stages are computationally expensive, and are not applied on each iteration. It was also proposed that a maturation stage based on the Interchange Heuristic could be more beneficial to the quality of the solutions in [27].

Another way to introduce diversity is through invasion, a procedure that introduces new individuals in the population. As presented in [5] or [32], invaders are added to keep the diversity high and

strengthen the capabilities of the other operators. Higher invasion rates were found to be more effective.

2.7. Selection

As for the selection of which individuals will go on the next iteration of the GA, most papers apply an elitist selection, which consists on keeping the individuals with the best fitness. An interesting addition to the plain selection comes from [28]. The procedure is called *fight* and every newly created child goes through the fight stage, in which it is pitted against the individual with the worst fitness. However, where an elitist procedure would solve the fight as a win for the child if it had better fitness, *fight* also takes into account the diversity change in the population when including the new children. This way, if a child is better than the worst individual but very similar or even equal to other existing individuals, it will not win the fight. As noted by Perez at al [28], this process can supply a useful local-minima-avoiding technique similar to the parent selection in the bionomic algorithm but without slowing down the algorithm or hindering fine searching. This technique inspired the selection procedure based on the Hamming distance that will be used for the EA proposed in this work. A weaker version of the fight procedure is an elitist one that does not allow repeated individuals.

2.8. Fitness

Most GA opt for a fitness function that corresponds exactly with the objective function of the original optimization problem. In the paper by Salhi and Gamal [32], the fitness function is monotonically distorted to distinguish between more fine increments of fitness. For the p-median problem particularly, the objective function is the cost of allocating the demand nodes to the facilities and their operating costs. Other costs may appear, such as maintenance or relocation costs.

It is also possible to not consider the fitness as the objective function. In [28] the diversity that the individual brings into the population is also considered as part of its fitness, yielding a population-based fitness.

It is important to keep the evaluation of the fitness as quick as possible, since it is the basis for much of the computing overhead in GA for p-median problems. A faster way of preserving diversity, an the one inspiring part of the proposed algorithm, comes from the fight selection.

When choosing a fitness different from the objective function, some precision is lost and any heuristic chosen to substitute it must be as realistic as possible while being much quicker to evaluate. Such a fitness is introduced in [25], considering two separate fitness functions in what is called a *bi-objective fitness*.

Chapter 3

An Evolutionary Algorithm for Solving the MISFLP

In this section we detail the specific procedures by which the proposed EA for the MISFLP solves such problems. The inner structure of the EA is that of a messy evolutionary algorithm. The encoding is index-based and clearly distinguishes between periods. Since the representation is in variable length strings, the algorithm is messy and the encoding is not binary, so the algorithm is evolutionary. The fitness is computed after finding the optimal allocation of customers to the open facilities, which implies that the EA is technically a hybrid EA. All in all, the developed algorithm is a messy hybrid EA. The crossover exchanges facilities between periods and moves the rest to others, favoring the parent with the better fitness in the child creation. The mutation also has some random exchanges of facilities between periods, and includes outright removal of facilities. The children are selected with a protected procedure, so that the diversity is not decreased too sharply.

3.1. Encoding and Fitness

The population of the EA is formed by individuals, each one representing the facilities that open for the first time on that period. Thus, the encoding is as follows. Each individual has $|T|$ disjoint sets of indices in I, the set of facility locations. The indices in the t-th set represent the indices of the facilities which are opened for the first time in time period t. The novelty here is that there can be more than the minimum of facilities being allocated each period, so a variable-length encoding is required.

Given an individual, the facilities that open on each period are given. In order to construct the final solution of the MISFLP it is still required to find the optimal assignment between demand nodes and facilities that minimises the allocation costs.

Let $d_j^t = \sum_{k=t}^{|T|} \left(\min_{i \in I^k} c_{ij}^k \right)$, I^k be the open facilities at time k and z_j^t be a binary variable of value 1

if customer j is newly allocated in period t. Then, the following is the allocation subproblem:

$$\min \quad \sum_{t\in T}\sum_{j\in J} d_j^t z_j^t \tag{3.1}$$

$$\text{subject to} \quad \sum_{t\in T} z_j^t = 1, \quad \forall j \in J \tag{3.2}$$

$$\sum_{j\in J} z_j^t = n^t - n^{t-1}, \quad \forall t \in T \tag{3.3}$$

$$z_j^t \geq 0, \quad \forall j \in J, \forall t \in T \tag{3.4}$$

The encoded individual, together with the optimum of this subproblem, becomes a feasible solution of the MISFLP. The customer allocation subproblem has the same constraints for all possible individuals and those constraints are preloaded in a CPLEX model to speed up the algorithm. The fitness requires solving an optimization problem, so the EA is actually a hybrid EA.

The MISFLP is uncapacitated, and the optimal allocation of customers to facilities is that which assigns the lowest customer-facility pairs every period.

The assignment cost of customer j that has demand coverage starting from period t is d_j^t, since is is allocated to the cheapest available facility on each time period following period t and demand cannot stop. The global cost is the sum of all assignment costs, which means that the objective function takes the sum over customers and starting time periods. Further constraints will take care that enough customers are allocated demand each time period, that there is only one starting period where demand is received for each customer and that at the planning horizon all demand is covered.

The fitness function, coinciding with the optimum in Equation (1.20), requires solving the allocation subproblem each time that it has to be computed. The facility opening and maintenance costs are added to the optimum of this subproblem, resulting in the fitness for the individual. The objective is to find the individual with the lowest possible fitness.

To achieve this, several basic operations are carried out: crossover of two parents, mutation of one parent and selection of children.

An element of the individual denoting the initial opening of some facility in some period will be called a *gene*. The set of all genes, and thus, all encoded information for a certain individual will be the *genome* for that individual. The multi-periodicity could be included in this biological notation by denoting the different sets as *chromosomes*.

3.2. Initialization

The initialization is done in a random manner, meaning that, for each chromosome, the number of facilities that open each period and which ones are chosen is carried out randomly. The only diversity

that comes from this initialization is in the uniform distribution of the random process used. The sequential distribution [3] and population seeding [23], as explained in Section 2.3, cannot be readily applied here. The issue is that since more than the minimum of facilities could open on a given period, in order to ensure the appropriate coverage, the population size should be really high. To give an idea of why this is so, we analyse the search space of a problem with $|T| = 12$ periods and $|I| = 30$ facility locations. The search space of the EA is the space of all possible individuals and its cardinal is computed for the MISFLP with the following formula.

$$(|I|)! \sum_{s_1=p_1}^{|I|-\mathbf{S}_1-\mathbf{P}_1} \frac{1}{s_1!} \sum_{s_2=p_2}^{|I|-\mathbf{S}_2-\mathbf{P}_2} \frac{1}{s_2!} \cdots \sum_{s_{|T|}=p_{|T|}}^{|I|-\mathbf{S}_{|T|}-\mathbf{P}_{|T|}} \frac{1}{s_{|T|}!(|I|-s_1-\cdots-s_{|T|})!} \tag{3.5}$$

In the formula, $p = (p_1, \ldots, p_{|T|})$ is the vector with minimum facilities that can open each period. In order to simplify the expression, we use the notation $\mathbf{S}_1 = 0$ and $\mathbf{S}_i = \sum_{j=1}^{i-1} s_j$, $\forall i = 2, \ldots, |T|$, and $\mathbf{P}_i = \sum_{j=i+1}^{|T|} p_j$, $\forall i = 1, \ldots, |T|-1$ and $\mathbf{P}_{|T|} = 0$.

For $p = (1,1,1,2,1,1,1,1,1,1,1,1)$, the cardinal of the search space is of the order 10^{32}. If we compare that to the problem without multi-periodicity, the possible individuals become $\binom{30}{13}$, which is of the order 10^8. For this specific example, an individual using the minimum possible facilities could be

$$(\{12\},\{8\},\{26\},\{10,27\},\{4\},\{20\},\{2\},\{18\},\{15\},\{22\},\{1\},\{6\})$$

Notice that every individual initialized is a point in the search space, ensuring its feasibility. This means that there are no individuals with periods where the number of opened facilities is lower than the required by p.

The specific construction of initialized individuals is as follows. First, choose a random number of facilities to open between $\sum_{t=1}^{|T|} p^t$ and $|I|$. This is the step that ensures feasibility. After that, assign to periods all available excess slots, that is, those over $\sum_{t=1}^{|T|} p^t$. Then, unused facilities are randomly chosen to fill all unassigned genes, updating the used facilities in the process to avoid repetitions. This creates a new, feasible individual, and the procedure can be carried out as many times as necessary to generate the whole population.

In pilot runs, it was observed that the optimal individuals had no excess facilities, that is, the number of facilities opening each period corresponded exactly with the minimum allowed. Because of this, a second computational study was carried out where half of the initial population was generated with a left-skewed Beta$(1,5)$ distribution in the excess facilities. Individuals initialized by this procedure were more likely to have less opening facilities thus reducing computational time without sacrificing too much diversification. Everything else was kept the same.

Each individual in the generated population is evaluated for fitness as explained previously. After this, the main loop of the algorithm begins. It will continue until the stop criteria is met. In our case, a running time criterion is used.

3.3. Crossover

In every iteration of the loop, there is one crossover and sometimes one mutation, according to some probability of mutation. This is unusual for EA, since usually many children are generated on each iteration. However, the selection operator from the paper by Perez et al. [28] motivated the current selection operator. The significance of it is that premature convergence to local minima, a long-time crux of GA and EA can be significantly overcome through this method. Since it compares the child with the main individual that generated it, multiple children generation is not easily implemented. The crossover works as follows.

First, two random distinct parents are chosen among those in the current population. These two parents are recombinated by the crossover operator. The parents are ordered by fitness, and the one with the lower fitness is called parent α while the other one is parent β. The child is initialized as parent α, to try to retain some of the good properties of α while enriching the gene pool with the genome from β. Once initialized the child, a random period t is selected.

The information from period t in β is translated to α in the following manner. Only the genes that are different in the period t between α (α^t) and β (β^t) are used for the crossover. We consider $\alpha^t \setminus (\alpha^t \cup \beta^t)$ and $\beta^t \setminus (\alpha^t \cup \beta^t)$. If both of these sets are nonempty, random $i \in \alpha^t \setminus (\alpha^t \cup \beta^t)$ and $i' \in \beta^t \setminus (\alpha^t \cup \beta^t)$ are selected. Then, i is substituted by i' in the child and the other position (if it exists) i' is substituted by i. If only i was substituted this time, the genome of the child could contain two instances of the same facility i', which is an unfeasible individual.

Facilities can be exchanged as in the process above, which will be denoted as facility permutation. After the substitution, the choices for facility permutation are updated and the procedure repeated until one of the sets is empty. If it is $\beta^t \setminus (\alpha^t \cup \beta^t)$ which empties first, that is, $|\alpha^t \setminus (\alpha^t \cup \beta^t)| > |\beta^t \setminus (\alpha^t \cup \beta^t)|$, the remaining facilities are removed from the period t of the child and appended to other random periods. Note that all operations preserve feasibility.

Hence, if parent α is $(\{9,1\},\{2\},\{7,3,12\},\{4\})$, parent β is $(\{4,6,10\},\{11\},\{9,7\},\{5\})$ and $p = (2,1,1,1)$ a possible crossover could be as follows. Initially, the child is the same as parent α. After randomly selecting period $t = 3$ to carry out the crossover, the exchange vectors $\alpha^t \setminus (\alpha^t \cup \beta^t)$ and $\beta^t \setminus (\alpha^t \cup \beta^t)$ are computed. They are $(3,12)$ and (9), respectively. We randomly select $i = 12$ and $i' = 9$. The child, which started as $(\{9,1\},\{2\},\{7,3,12\},\{4\})$ is now $(\{12,1\},\{2\},\{7,3,9\},\{4\})$. Notice that if i' had been an unused facility in α, for instance $i' = 1$, the result would have been $(\{9,1\},\{2\},\{7,3,11\},\{4\})$. After this, the exchange vector of α is still nonempty, so the remaining facility (3) moves to another random period $\tilde{t} = 2$. Finally, the child is $(\{12,1\},\{2,3\},\{7,9\},\{4\})$. The pseudocode of the procedure can be seen in Algorithm 1.

Data: dataFrame, parents, $|T|$
Result: child
order parents;
initialize child as parent alpha;
randomly choose period t to do crossover;
find out exchange vectors parAlphaDist, parBetaDist;
set lenA as ex. vector of alpha, lenB respectively;
if min*(lenA,lenB)>0* **then**
  for *k in 1:*min*(lenA,lenB)* **do**
    $i \leftarrow$ random facility in parAlphaDist;
    $j \leftarrow$ random facility in parBetaDist;
    child's $i \leftarrow j$;
    for *all other periods* $\tilde{t}$ **do**
      if j *in child's* $\tilde{t}$ *period* **then**
        child's j in $\tilde{t} \leftarrow i$;
      end
    end
    update exchange vectors
  end
end
if *length(parAlphaDist)>0* **then**
  for *k in parAlphaDist* **do**
    remove facility k from child's period t;
    randomly choose period $\tilde{t}$;
    append k to child's period $\tilde{t}$;
  end
end

Algorithm 1: Pseudocode for the Crossover operator

3.4. Selection

After the crossover, the fitness of the child is evaluated and goes through the selection procedure. In the selection, the child is pitted against parent α. Only if the fitness of the child is better than that of the parent will it be placed in the population. The individual that it will substitute depends on the Hamming distance between the parent and child (set-wise by periods). This distance is the sum per periods of the largest amount of facilities opened between the child and the parent minus the amount of the common facilities. For example, the Hamming distance of $(1,2,3)$ and $(3,4)$ is $3-1=2$.

This avoids premature convergence at local minima. If the obtained distance is lower than a bound minDistance, the child substitutes the parent in the population. If the obtained distance is higher, the child substitutes the individual with worst fitness in the population. In the end, the population is re-ordered by fitness.

3.5. Mutation

The mutation is carried out in each iteration with a fixed probability of prMutate. Again, it receives a randomly-selected parent from the current population which will be mutated. This current population includes the result of the selection after the crossover, so it could be possible to choose the selected crossover child as the individual to mutate. The individual to mutate is started as equal to this parent, and will have some number of genes changed. This number is randomly chosen between a lower and an upper bound. The lower bound is the maximum of 1 and mutRate $-$ 1, mutRate being a fixed parameter. This ensures that some mutation is carried out. The upper bound is the minimum of $\sum_{t=1}^{|T|} p^t$ and mutRate $+ 1$. Since feasible individuals can have as little as $\sum_{t=1}^{|T|} p^t$ genes, it is necessary to set the bound so as to avoid redundant operations.

For each gene changed, we first set the permutation as impossible if there is only one period in which facilities open. This is an implementation issue and really has no effect on the usefulness of the algorithm, since we are focusing on the multi-period instances. Then, a random period t with open facilities is chosen, as well as a facility i in the set m^t of facilities newly opened at period t of individual m.

With probability $\frac{p^t}{|m^t|}$, a permutation of facilities is carried out. The permutation is with respect to a random unused facility with a probability of $\frac{1}{|T|}$, meaning that a new facility will substitute i. This is the only way that facilities not present in the original population can be introduced. If there are no unused facilities, the procedure is repeated for the next gene to mutate. Otherwise, the permutation is with respect to another randomly-chosen period $\tilde{t}$ and $i' \in m^{\tilde{t}}$. In this case, i is substituted by i' and vice versa. If the facilities opening in that period are close to the minimum, it is more likely that this permutation is carried out.

When there is no permutation, a random $i \in m^t$ is removed from that period. With probability $1 - \frac{1}{|T|}$, i is appended to another random period. This accounts for the facility being removed from the individual completely. Notice that the probabilities are set up in such a way that no matter the random outcome, the mutated individual will be feasible.

Another mechanic introduced is that which allows facilities to be completely removed from a period, without permutation. The removal, combined with the moving of excess facilities, takes advantage of the structure of the problem, allowing for more rich exploration of the search space than simple permutation.

After this, the periods with facilities are updated and the loop is repeated for the remaining number of genes to mutate.

The mutated individual is compared to its respective parent and goes through the selection operator, as after the crossover. The pseudocode of the mutation procedure can be seen in Algorithm 2.

Data: parent, mutRate, p, facilities, $|T|$
Result: mutated
initialize mutated individual as parent;
mutRate $\leftarrow$ random between (max(mutRate-1,1) and min(mutRate+1,sum(p)));
mutRate $\leftarrow$ min(mutRate, number of facilities opened in excess of p);
for *k in 1:mutRate* **do**
  $t \leftarrow$ random period in which to mutate a gene;
  $i \leftarrow$ facility in period t;
  if $\mathscr{U}(1)$<*minimum facilities to open in t/facilities open in t* **then**
    if $\mathscr{U}(1)<1/|T|$ **then**
      $i \leftarrow$ random unused facility;
    else
      $\tilde{t} \leftarrow$ other random period;
      $j \leftarrow$ random facility in period $\tilde{t}$;
      $i \longleftrightarrow j$ (permutation in respective periods);
    end
  else
    remove i from original period;
    if $\mathscr{U}(1)>1/|T|$ **then**
      $\tilde{t} \leftarrow$ other random period;
      append i to $\tilde{t}$;
    end
  end
end

Algorithm 2: Pseudocode for the Mutation operator

Chapter 4

Computational Study

The algorithm presented in Chapter 3 has been subject to a thorough computational study. On this section, these results will be presented and put in context.

4.1. Methodology

The algorithm has been implemented in RStudio 1.1.456 with R 3.4.2 and the package cplexAPI 1.3.3 to interface with ILOG CPLEX Callable Library 12.8.0. CPLEX uses all four available cores for the computation whereas RStudio only uses one processor core. When calling CPLEX through the package cplexAPI, the amount of cores used is one as well. The discrepancy in number of cores used has not been accounted for in the results presented since downgrading the execution of the optimizer to one core would be illogical and the implementation of the algorithm does not allow for parallelization in an obvious fashion. The programming language used is good enough for a proof-of-concept algorithm, and it is assumed that, had it been programmed in a compiled language such as C, running times and result quality would have increased accordingly.

Two main computational studies have been carried out on an Intel Core i7-870 processor with 2.93 GHz of base clock speed and 8GB of 1066 MHz DDR3 RAM. The pilot tests for the setting of parameters and graph generation were carried out on a Mac running an Intel Core i5-6267U Processor at 3.1 GHz clock speed and 16 GB of 2133 MHz LPDDR3 RAM. The RAM capacity was not a bottleneck for either configuration, since the test instances took up much less than that.

The set of instances considered is the same as the one used in [2], and they are used for the sake of comparison. The set consists of 195 instances with parameters ranging in $|T| \in \{4,5,6,7,8,10,12\}$, $|I| = \{8,10,12,15,20,30\}$ and $|J| = \{50,100,150,200,500\}$. The first instances generated in such a manner are the ones chosen. For more details on the random distributions used to generate the instances, check [2]. The optimal solution of the MISFLP instances was calculated with CPLEX, keeping track of

the running time. The quality of the results from the EA was then based on these exact solutions.

4.2. Parameter Choice

The experiments are carried out as follows. A pilot study is responsible for fine tuning the parameters used in the whole computational study. For the size of the population to initialize, it is determined that 100 is enough to reach sufficient convergence both in big and small instances. Lower values resulted in the EA being stuck in local minima at small instances. Higher populations took too long to compute without achieving significantly better results. It was observed that the crossover operator struggled to bring out the good individuals, and the reliance was too much on mutation.

The probability of mutation is set at 30% and has been corrected afterwards to 60%. Such high probabilities of mutation are not very common at all in the EA research, but because of the unique choice of the selection operator, the proposed EA has an order of magnitude higher mutation. The reason is because the selection protects the mutation, discarding the mutated individual if its fitness is not appropriate. In this way, it behaves like a maturation or local search. The amount of mutation, upon which the amount of genes to mutate is chosen, is set at 1.

The minimum Hamming distance between two individuals is 3, so individuals with distance 3 or lower to their respective parent α substitute the parent and not the one with the worst fitness.

The first computational study used the EA with only random initialization of the population. This means that the whole 100 individuals initialized were random both in the amount of opened facilities and which ones were chosen.

Two sets of experiments were carried out. On both, the 195 chosen instances are solved 5 times each with set seeds for the random variables. The termination criteria are different depending on the size of the instances. For the smallest instances ($|T| = 4, |I| \leq 20$ and $|J| \leq 150$), the algorithm is stopped after $3 * 10^3$ iterations without a new individual with best fitness. For instances such that $|T| \leq 7, |I| \leq 12$ and $|J| \leq 150$, the algorithm is stopped after 10^4 iterations without a new individual with best fitness. On the remaining instances, the algorithm is allowed to run for 10 minutes.

The ordering by size of the instances first considers the periods, then the facilities available and then the customers. So an instance with $|T| = 5, |I| = 10$ and $|J| = 150$ is considered bigger than one with $|T| = 4, |I| = 30$ and $|J| = 500$.

These criteria were set after comparing the obtained solutions with the optimal solution in pilot studies. However, it is not necessary to solve the optimisation problem, and similar results can be obtained by studying the evolution graphs of instances, such as the ones in Figures 4.1 and 4.2.

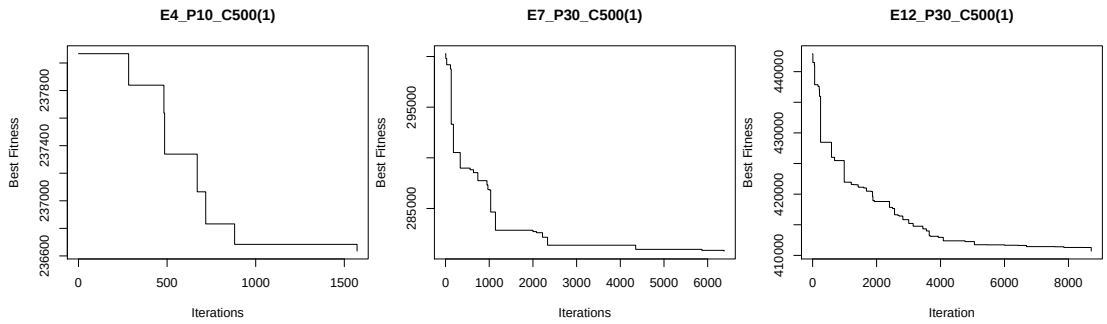

Figure 4.1: Evolution graphs of the fitness function in small, medium and big instances, respectively

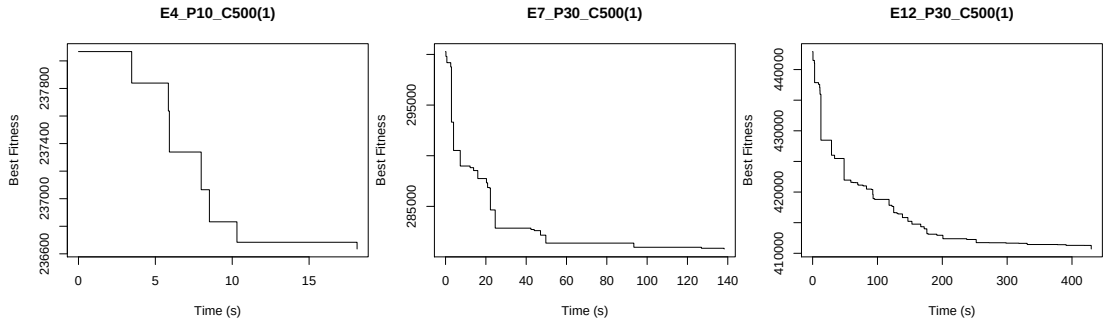

Figure 4.2: Evolution graphs with respect to running time of the fitness function in small, medium and big instances, respectively

4.3. Improving the Decoupled Model

The computational study in [2] includes the solution of the instances with CPLEX in a decoupled manner, i.e. period by period. The optimal location of facilities considering the information from just the first period is found. Then, given these facilities, the optimal location for the following period is obtained. Proceeding in this manner until all periods are considered results in a solution of the MISFLP that is not optimal. However, this decoupled problem is much faster to solve than the original MISFLP, a speed increase of the order of 10^2. For instances of average size, it is likely faster to solve than with the proposed EA as well. The shortfall of the method is that the non-optimal solution cannot be improved further and its structure is usually not desirable. In fact, this procedure yields better optima in the initial periods but worsens as unforeseen costs crop up. The final cost obtained with this artificial decoupling is higher than the actual minimum cost. For the studied instances in [2], the average percent deviation is 2.6%. It is important, then, to set proper termination criteria for the EA such that the GAP reached is lower than the artificially decoupled optimum. Otherwise, the decoupled problem is much faster than the EA and the benefits of using the heuristic decrease.

4.4. First Experiment

The first experiment has a randomly initialized population of size 100, 30% probability of mutation, 1 as the amount of mutation and minimum Hamming distance 3.

The quality of the results from the first experiment is shown in Table 4.1. Specifically, the average percent GAP and the maximum GAP of the 5 iterations is shown. The GAP is a measure of the quality of the achieved solution, calculated as follows:

$$\frac{best_{EA} - opt}{opt} * 100$$

The percentage GAP is where opt is the optimum value of the objective function, as calculated by CPLEX and $best_{EA}$ the best individual in the population at the termination of the evolutionary algorithm. Lower GAPs are better, and GAPs equal to 0 mean that the optimum is reached. These cases are marked with an asterisk.

Since heuristic methods should produce quick solutions, the computational times between the EA and the CPLEX optimisation are compared. Specifically, t_{EA}, the time in seconds that it takes the EA to obtain the individual with the best fitness before termination of the EA is compared to $time_{opt}$, the time to reach the optimum by CPLEX.

The ratio *fast* shown in Table 4.2 is calculated as follows:

$$\frac{t_{opt}}{t_{EA}}$$

Values of *fast* close to 1 mean that no discernible speed difference exists between the EA and CPLEX for solving a given instance. Values higher than 1 correspond to instances where the EA is faster than CPLEX. These cases have been prepended with a dot to help distinguish them. In Table 4.2, the average t_{EA} are also shown.

A quick summary of the results obtained from the first experiment is the following. In 88% of the cases, the optimal solution is reached. Overall, the solution quality is quite good, with an average GAP of $1.03 * 10^{-4}$. The worst instance has GAP $3.60 * 10^{-3}$, which is more than an order of magnitude lower than the expected GAP from a decoupled model. This validates the used termination criteria.

In terms of speed, on average the EA is half the speed of CPLEX. However, this is due to the instances not being big enough. Large instances cannot be easily optimised and thus are the goal of heuristic methods. If we only consider the instances where the EA is faster, we find that for 29 of the bigger instances, the EA is 7.9 times faster on average. As for the biggest instance available ($|T| = 12$, $|I| = 30$ and $|J| = 500$), the EA is up to 39 times faster.

Out of the 195 attempted instances, only in 8 of them none of the 5 attempts reach the optimum. For these instances, the average GAP is 10^{-3} and the average minimum GAP, $5 * 10^{-4}$. They are also solved efficiently, with the EA being 5.3 times faster on average.

		$\|I\|$											
		8		10		12		15		20		30	
$\|T\|$	$\|J\|$	av.	max	av.	max	av.	max	av.	max	av.	max	av.	max
4	50	*	*	*	*	*	*	*	*	0.1	0.4	0.12	0.34
	100	*	*	*	*	*	*	*	*	*	*	0.02	0.04
	150	*	*	*	*	*	*	*	*	*	*	0.04	0.12
	200	*	*	*	*	*	*	*	*	*	*	0.01	0.02
	500	*	*	*	*	*	*	*	*	*	*	*	*
5	50	*	*	*	*	*	*	*	*	*	*	0.13	0.36
	100	*	*	*	*	*	*	*	*	*	*	*	*
	150	*	*	*	*	*	*	*	*	0.9	0.15	0.01	0.03
	200	*	*	*	*	*	*	*	*	*	*	0.02	0.07
	500	*	*	*	*	*	*	*	*	*	*	0.05	0.22
6	50	*	*	*	*	*	*	*	*	*	*	0.10	0.17
	100	*	*	*	*	*	*	*	*	*	*	0.03	0.17
	150	*	*	*	*	*	*	*	*	*	*	0.12	0.23
	200	*	*	*	*	*	*	*	*	*	*	0.01	0.05
	500	*	*	*	*	*	*	*	*	*	*	0.05	0.13
7	50	*	*	*	*	*	*	*	*	*	*	*	*
	100	*	*	*	*	*	*	*	*	*	*	0.01	0.01
	150	*	*	*	*	*	*	*	*	0.01	0.02	*	*
	200	*	*	*	*	*	*	*	*	*	0.02	0.02	0.07
	500	*	*	*	*	*	*	*	*	*	*	*	*
8	50	*	*	*	*	*	*	*	*	*	*	0.05	0.06
	100	*	*	*	*	*	*	*	*	*	*	*	*
	150	*	*	*	*	*	*	*	*	*	*	0.06	0.16
	200	*	*	*	*	*	*	*	*	*	*	0.02	0.05
	500	*	*	*	*	*	*	*	*	*	*	0.01	0.02
10	50			*	*	*	*	*	*	0.01	0.07	0.09	0.19
	100			*	*	*	*	*	*	*	*	0.02	0.02
	150			*	*	*	*	*	*	*	*	0.01	0.06
	200			*	*	*	*	*	*	*	*	0.03	0.13
	500			*	*	*	*	*	*	0.14	0.21	0.21	0.28
12	50					*	*	0.02	0.05	*	*	*	0.01
	100					*	*	*	*	*	*	0.04	0.05
	150					*	*	*	*	0.04	0.10	0.02	0.07
	200					*	*	0.01	0.04	*	*	0.02	0.04
	500					0.01	0.07	0.06	0.17	0.07	0.18	0.20	0.32

Table 4.1: Average and maximum GAP

		$\|I\|$											
		8		10		12		15		20		30	
$\|T\|$	$\|J\|$	t_{EA} (s)	fast	t_{EA} (s)	fast	t_{EA} (s)	fast	t_{EA} (s)	fast	t_{EA} (s)	fast	t_{EA} (s)	fast
4	50	1.28	0.18	7.69	0.03	2.62	0.07	8.52	0.03	15.66	0.01	51.23	0.01
	100	1.12	•1.03	3.84	0.26	6.33	0.08	9.25	0.04	19.55	0.02	55.77	0.01
	150	2.74	0.13	17.00	0.11	6.80	0.11	10.50	0.10	14.21	0.15	40.96	0.04
	200	4.82	0.15	7.03	0.13	16.90	0.10	12.78	0.23	52.56	0.14	27.56	0.11
	500	17.36	0.26	28.37	•1.07	13.39	0.17	57.73	0.29	70.66	0.09	90.48	0.33
5	50	4.06	0.05	2.07	0.12	7.46	0.04	15.27	0.01	19.81	0.01	50.36	0.01
	100	4.89	0.33	8.60	0.14	12.21	0.05	8.39	0.08	37.47	0.03	34.09	0.11
	150	17.29	0.05	33.45	0.09	35.75	0.04	31.17	0.10	63.72	0.04	52.63	0.03
	200	9.19	0.09	32.57	0.05	54.57	0.03	44.74	0.33	27.21	•1.01	88.86	0.05
	500	97.86	0.15	70.75	0.24	84.49	0.09	151.99	0.21	141.58	0.76	242.42	0.05
6	50	3.43	0.20	5.23	0.11	15.41	0.06	12.27	0.10	41.46	0.05	59.90	0.01
	100	10.07	0.12	10.37	0.13	10.70	0.06	25.13	0.29	39.02	0.02	50.40	0.43
	150	10.88	0.43	38.62	0.04	24.80	0.26	13.35	0.10	36.86	0.67	106.83	0.02
	200	32.90	0.13	26.55	0.24	46.99	0.06	21.33	0.11	69.37	0.17	74.43	0.87
	500	27.68	0.35	57.60	0.10	65.74	0.12	124.43	0.17	261.79	0.07	333.93	0.07
7	50	5.28	0.12	8.47	0.17	12.86	0.04	191.58	0.01	38.56	0.02	75.26	0.03
	100	16.77	0.09	23.68	0.35	21.11	0.46	45.80	0.49	146.78	0.15	253.10	0.02
	150	15.20	0.15	47.18	0.36	15.17	0.06	196.22	0.56	55.04	0.06	104.88	0.84
	200	38.07	0.26	52.37	0.10	90.54	0.16	100.45	0.50	102.04	0.07	142.47	0.05
	500	53.20	•1.48	97.12	0.46	68.04	0.34	107.40	0.43	275.63	0.41	281.56	•6.28
8	50	12.61	0.04	51.50	0.02	20.34	0.04	28.62	0.02	76.28	0.02	256.10	0.01
	100	15.77	0.14	65.29	0.12	152.78	0.12	50.33	0.09	241.05	0.01	149.30	0.60
	150	116.14	0.04	94.34	0.09	71.43	0.72	234.35	0.15	213.84	0.78	306.26	0.06
	200	242.97	0.02	278.68	0.02	165.77	0.13	303.25	0.18	203.47	0.19	371.19	0.64
	500	34.64	•1.69	90.08	•1.45	274.49	0.65	280.44	0.72	367.50	0.07	529.49	0.29
10	50			24.14	0.09	42.24	0.15	97.70	0.10	251.48	0.02	332.29	0.01
	100			120.29	0.02	29.60	0.53	193.74	0.27	283.23	0.48	227.68	0.02
	150			123.75	0.23	322.71	0.18	73.44	0.60	343.99	0.49	304.13	0.38
	200			80.26	0.22	54.92	0.37	145.60	0.78	105.95	0.12	261.61	0.39
	500			141.68	•1.02	191.44	•1.65	402.21	0.33	533.68	•3.07	550.80	0.65
12	50					50.23	0.21	155.95	0.14	219.74	0.20	327.33	0.05
	100					147.34	0.06	69.52	0.29	234.09	0.45	289.90	0.16
	150					92.27	0.26	245.08	0.83	189.86	0.25	308.20	0.53
	200					165.79	0.50	115.42	•2.26	295.07	0.96	318.30	0.25
	500					316.58	•1.66	448.89	•1.29	564.10	•2.16	491.38	•34.84

Table 4.2: Average times of EA and comparison with CPLEX optimum

4.5. Second Experiment

An additional experiment has been carried out with mixed population initialization and higher mutation probability at 60%. The mixed strategy for initializing the population consists of a random initialization of half of the population and a skewed initialization of the other half. With the chosen parameters, 50 individuals are random and the next 50 are skewed towards opening less facilities. As noted in Section 3.2, the skewed distribution used is a left-skewed Beta$(1,5)$ scaled so that the bounds for minimum and maximum possible facilities match those of the instance. The probability of mutation was also adjusted, since pilot studies showed good results with this mutation rate.

As before, the quality of the solutions achieved is shown in Table 4.3 and the time performance in Table 4.4. This experiment shows lower GAP with more optima being reached and at slightly faster speed.

A more in-depth summary is the following. More optima are reached, with 90% of the cases achieving the optimum. The solution quality is better, with an average GAP of $8.86 * 10^{-5}$. The worst instance also has lower GAP, at $3.44 * 10^{-3}$.

In terms of speed, the EA is again half the speed of CPLEX on average. The restriction to the 27 instances where the EA is faster results in improvements of speeds as well, with the EA being 8.5 times faster on average. For the biggest instance available, the EA is up to 44 times faster.

Only 6 instances are not solved optimally in any of the 5 attempts, for which the average GAP is $1.3 * 10^{-3}$ and the average minimum GAP, $6.3 * 10^{-4}$. Since there are 2 fewer instances not solved optimally, now the EA is 6.7 times faster on average.

		$\|I\|$											
		8		10		12		15		20		30	
$\|T\|$	$\|J\|$	av.	max	av.	max	av.	max	av.	max	av.	max	av.	max
4	50	*	*	*	*	*	*	*	*	*	*	0.14	0.21
	100	*	*	*	*	*	*	*	*	*	*	0.01	0.04
	150	*	*	*	*	*	*	*	*	*	*	0.02	0.04
	200	*	*	*	*	*	*	*	*	*	*	0.01	0.03
	500	*	*	*	*	*	*	*	*	*	*	*	*
5	50	*	*	*	*	*	*	*	*	*	*	0.05	0.1
	100	*	*	*	*	*	*	*	*	*	*	*	*
	150	*	*	*	*	*	*	*	*	*	*	*	*
	200	*	*	*	*	*	*	*	*	*	*	0.02	0.07
	500	*	*	*	*	*	*	*	*	*	*	0.01	0.02
6	50	*	*	*	*	*	*	*	*	*	*	0.13	0.33
	100	*	*	*	*	*	*	*	*	*	*	0.01	0.04
	150	*	*	*	*	*	*	*	*	*	*	0.03	0.17
	200	*	*	*	*	*	*	*	*	*	*	0.03	0.15
	500	*	*	*	*	*	*	*	*	0.03	0.16	*	*
7	50	*	*	*	*	*	*	*	*	*	*	*	*
	100	*	*	*	*	*	*	*	*	*	*	0.01	0.01
	150	*	*	*	*	*	*	*	*	*	0.02	*	*
	200	*	*	*	*	*	*	*	*	*	*	0.01	0.03
	500	*	*	*	*	*	*	*	*	*	*	*	*
8	50	*	*	*	*	*	*	*	*	0.04	0.18	0.01	0.05
	100	*	*	*	*	*	*	*	*	*	*	*	*
	150	*	*	*	*	*	*	*	*	*	*	0.07	0.21
	200	*	*	*	*	*	*	*	*	*	*	0.03	0.05
	500	*	*	*	*	*	*	*	*	*	*	*	0.01
10	50			*	*	*	*	*	*	0.01	0.07	0.13	0.19
	100			*	*	*	*	*	*	*	*	0.02	0.05
	150			*	*	*	*	*	*	0.01	0.04	*	*
	200			*	*	*	*	*	*	*	*	0.02	0.12
	500			*	*	*	*	*	*	0.22	0.28	0.1	0.2
12	50					*	0.01	0.01	0.05	*	*	0.01	0.03
	100					*	*	*	*	*	*	0.02	0.05
	150					*	*	*	0.02	0.05	0.09	0.02	0.07
	200					*	*	0.01	0.04	*	*	0.03	0.07
	500					0.03	0.05	*	*	0.12	0.28	0.25	0.34

Table 4.3: Average and maximum GAP

		$\|I\|$											
		8		10		12		15		20		30	
$\|T\|$	$\|J\|$	t_{EA} (s)	fast	t_{EA} (s)	fast	t_{EA} (s)	fast	t_{EA} (s)	fast	t_{EA} (s)	fast	t_{EA} (s)	fast
4	50	1.79	0.26	7.53	0.07	2.11	0.17	9.23	0.05	16.75	0.02	55.09	0.01
	100	1.28	0.22	3.38	0.36	9.12	0.07	8.64	0.06	21.45	0.04	39.33	0.02
	150	2.78	0.16	13.16	0.51	7.13	0.16	16.43	0.12	13.65	0.18	42.92	0.04
	200	6.13	0.09	8.22	0.15	16.74	0.22	16.05	0.25	35.93	0.21	56.36	0.10
	500	22.48	0.44	23.98	•1.72	13.36	0.21	35.98	0.91	65.89	0.10	54.09	0.79
5	50	5.04	0.03	3.22	0.07	6.43	0.06	19.08	0.02	20.13	0.02	55.31	0.01
	100	5.84	0.67	12.45	0.16	25.91	0.06	13.2	0.07	48.4	0.04	26.53	0.26
	150	21.15	0.05	16.19	0.16	43.96	0.04	28.84	0.16	49.52	0.11	77.68	0.03
	200	13.27	0.11	38.17	0.06	58.75	0.04	29.97	0.48	24.82	•1.25	112.58	0.07
	500	132.18	0.31	108.01	0.26	87.68	0.09	173.88	0.46	149.5	0.89	259.39	0.07
6	50	4.35	0.21	4.61	0.14	11.31	0.12	11.1	0.14	31.34	0.09	71.56	0.01
	100	15.07	0.21	10.12	0.17	13.09	0.06	25.17	0.43	27.02	0.05	61.64	0.62
	150	14.74	0.37	44.59	0.08	51.94	0.54	16.87	0.10	41.54	0.64	122.56	0.02
	200	28.84	0.21	35.88	0.28	38.57	0.12	22.41	0.15	110.81	0.25	102.22	0.88
	500	16.41	0.75	44.46	0.19	74.31	0.15	75.53	0.95	213.78	0.11	297.41	0.09
7	50	4.17	0.21	7.17	0.37	15.36	0.06	10.8	0.05	33.1	0.03	85.83	0.04
	100	59.19	0.02	14.3	0.85	22.77	0.50	24.75	•1.21	99.17	0.29	176.11	0.04
	150	25.96	0.19	53.38	0.39	14.77	0.09	269.44	0.65	51.95	0.08	79.55	•1.71
	200	69.44	0.27	78.34	0.10	166.85	0.16	249.16	0.65	91.58	0.08	129.89	0.08
	500	46.49	•2.36	96.44	0.79	72.65	0.57	98.97	0.50	333.44	0.51	197.15	•12.64
8	50	9.78	0.13	68.26	0.02	18.79	0.06	31.54	0.02	136.27	0.02	280.74	0.01
	100	19.35	0.18	237.4	0.19	191.6	0.20	27.56	0.11	100.54	0.04	82.17	•1.59
	150	77.07	0.09	144.51	0.17	72.47	0.80	99.85	0.68	171.55	0.81	245.79	0.19
	200	264.85	0.04	268.56	0.13	336.02	0.52	230.44	0.62	239.32	0.29	333.32	0.82
	500	51.43	•1.28	64.26	•2.41	228.65	0.93	199.04	•1.66	349.66	0.09	494.19	0.39
10	50			26.17	0.12	47.24	0.18	49.33	0.14	175.17	0.03	216.13	0.02
	100			196.71	0.02	38.59	0.49	102.88	0.69	284.2	0.73	250.9	0.04
	150			97.12	0.36	223.27	0.27	91.04	•1.20	222.49	0.75	296.85	0.62
	200			100.26	0.23	63.52	0.38	130.67	•1.01	105.34	0.13	348.25	0.38
	500			124.64	•1.41	246.87	•1.48	333.17	0.43	561.1	•3.40	537.98	0.92
12	50					43.69	0.24	87.8	0.34	139.42	0.23	267.39	0.08
	100					116.8	0.12	97.07	0.25	244.42	0.52	391.5	0.16
	150					128.59	0.40	192.57	•1.09	241.66	0.45	342.36	0.61
	200					331.34	0.67	183.25	•2.46	270.61	•1.47	427.5	0.31
	500					315.78	•2.80	473.88	•1.75	512.07	•2.90	512.66	•45.25

Table 4.4: Average times of EA and comparison with CPLEX optimum

Chapter 5

Conclusions

In this work, an evolutionary algorithm for the Multi-Period Facility Location Problem is proposed. The objective is to achieve good results quickly for instances where the linear solver fails to achieve optimality in a reasonable time. The initial part of this work focuses on introducing the MISFLP and the existing available research on evolutionary algorithms applied to the p-median problem. We develop an evolutionary algorithm that is able to solve a multi-period problem. The computational study shows enticing results that validate the proof-of-concept implementation in R. Both the quality and the speed of the algorithm are satisfactory and justify the use of this heuristic over the decoupled model or the MISFLP model stated in Section 1.5 for bigger instances.

To the authors' knowledge, this is the first case of an evolutionary algorithm applied to a Multi-Period FLP. The developed operators and successful results encourage further research into these kinds of problems by means of evolutionary-based heuristics.

Many extensions of the base algorithm were considered, but they exceeded the scope of the project. The running time termination criteria could be modeled so as to be dynamic and depend on the size of the studied problem. A more detailed computational study focusing on the benefits of different population sizes and strategies for choosing the proper amount is also warranted. As for the specific algorithm, several extensions that show promise and may be considered in the future are the following: adding a proper maturation stage with a Local Interchange heuristic, dynamically adjusting the algorithm parameters based on the diversity of the population and finding a faster fitness, since it takes up most of the computation time. Similarly, a refactoring of the code or a translation to C would likely improve the efficiency. The MISFLP decoupled model could also be programmed and solved for the analysed instances, in order to compare the GAP instance by instance.

Acknowledgements

This research has been supported by the Vice-Rectorate of Scientific Policy of the University of Zaragoza (PEX-18-004). This support is gratefully acknowledged.

Appendix A

Code of the Evolutionary Algorithm

Code/main.R

```
#Main file

library('cplexAPI')                #Used for solving the allocation subproblems
library('gtools')                  #Used for the natural sorting of filenames

source('parse.R')
source('genetic.R')
source('init.R')
source('allocate.R')
source('crossover.R')
source('mutation.R')
source('selection.R')


#Parameters:

dataSeed          <- 45004
solveSeeds        <- c(61420, 45498, 47284, 86094, 44162, 11854, 82696, 78401, 74861, 82818, 42284, 44249, 94029, 22427,
                       57635, 57043, 69665, 58334, 11994, 36237, 27954, 96102, 22073, 13746, 27740, 83262, 31764, 50826,
                       79943, 76675)
filepaths         <- list.files(path = '../datosMISFLP/')[-1]
filepaths         <- paste('../datosMISFLP/', filepaths, sep = '')
filepaths         <- mixedsort(filepaths)
numIter           <- 5             #Less than 30
chosenProblems    <- filepaths[c(1:195)*10-9]    #Only solving the first instance of each problem
bigWithGap        <- chosenProblems[c(113,117,119,141,146,148,149,150,166,167,168,171,172,174,175,
```

```
27                                          180,181,184,188,190,191,192,193,194,195)
                                              ]
28
29  sizePopulation    <- 100            #Number of individuals
30  prMutate          <- 0.6            #Probability to enter the mutation
        stage each iteration
31  mutRate           <- 1              #Amount of genes to mutate
32  maxNoBestIter     <- 3000           #Maximum iterations allowed
       without a new best fitness
33  maxTime           <- 600            #Maximum time allowed
34  minDiff           <- 3              #Minimum difference between kept
       father and child
35  autoMaxNoBestIter<- TRUE            #Automatically select
       maxNoBestIter based on preset
36
37  results           <- 'results.txt'
38
39  for(problem in chosenProblems){
40    cat(sprintf('Creating problem from file...\n'))
41
42    data <- parse(problem)
43
44    sizePeriods      <- data$sizePeriods[1]
45    sizeFacAvailable <- data$sizeFacAvailable[1]
46    sizeCustomers    <- data$sizeCustomers[1]
47    n                <- data$n        #Difference in minimum number of
         customers to start receiving product from period to period
48    p                <- data$p        #Vector with minimum of facilities
          to open per period
49    c                <- data$c        #Assignment cost of customers to
         facilities by period [[t]][[j]][[i]]
50    f                <- data$f        #Cost of running facility from
         period [[t]][[i]]
51
52    if(sum(p) > sizeFacAvailable){
53      print('Insufficient Facilities')
54      next()
55    }
56
57    if(sum(p) < mutRate){
58      print('Attempting to mutate more facilities than possible')
59      next()
60    }
61
62    if(length(p[p != 0]) < 2){
63      print('Near-empty individuals cause problems with mutation')
64      next()
65    }
66
67    if(sizePeriods < 3){
```

```
68      print('Periods too small, can cause problems with sample
            generation')
69      next()
70    }
71
72    if(autoMaxNoBestIter){
73      if(problem %in% chosenProblems[1:23]){maxNoBestIter <- 3000}
74      if(problem %in% chosenProblems[24:103]){maxNoBestIter <- 10000}
75      if(problem %in% chosenProblems[104:195]){maxNoBestIter <-
            10000000}
76    }
77
78    cat(sprintf('Problem construction finished\n'))
79
80    for(i in 1:numIter){
81      cat(sprintf('Evolving new population...\n\n'))
82      time       <- system.time(lst <- genetic(solveSeeds[i],
            sizePopulation, p, n, sizeFacAvailable,
83                                    c, f, prMutate, mutRate,
                                          maxNoBestIter, minDiff,
                                          maxTime))
84      elapsed    <- unname(time[3])
85
86      df <- lst[[1]]
87      bestTime  <- lst[[2]]
88
89      fileConn  <- file(results,open="at")
90      writeLines(sprintf('%s\t%s\t%s\t%s\t%s\t%6f\t%2f\t%2f',dataSeed,
            solveSeeds[i], maxNoBestIter,sizePopulation,
91                         problem, df$fit[[1]],bestTime[[1]],elapsed),
                               fileConn)
92      close(fileConn)
93      rm(df)
94    }
95  }
```

Code/genetic.R

```
1  #This is the main genetic function, whith calls to the different
       functions.
2  #The algorithm initializes the population and does crossover and
       mutation in each iteration.
3  #It stops after the required stop criteria are met.
4
5  #The parameters are:
6  # chosenSeed:                Seed for the random calls in this
       problem
7  # sizePopulation:            Size of the population to initialize
8  # p:                         Vector with minimum of facilities to
       open per period
9  # n:                         Difference in minimum number of
       customers to start receiving product from period to period
```

```
# sizeFacAvailable:          Number of available facilities by index
# c:                         Assignment cost of customers to facilities by period [[t]][[j]][[i]]
# f:                         Cost of running facility from period [[t]][[i]]
# prMutate:                  Probability of mutation
# mutRate:                   Amount of genes to mutate
# maxNoBestIter:             Maximum iterations allowed without a new best fitness
# minDiff:                   Minimum difference between kept father and child

genetic <- function(chosenSeed, sizePopulation, p, n, sizeFacAvailable, c, f, prMutate = 0.1, mutRate = 1,
                    maxNoBestIter = 10000, minDiff = 3, maxTime = 120)
{
  set.seed(chosenSeed)

  sizePeriods <- length(p)
  sizeCustomers <- sum(n)

  cat(sprintf('The chosen seed is          %5d\n', chosenSeed))
  cat(sprintf('The population size is      %s\n', sizePopulation))
  cat(sprintf('The number of periods is    %s\n', sizePeriods))
  cat(sprintf('The number of facilities is %s\n', sizeFacAvailable))
  cat(sprintf('The number of customers is  %s\n', sizeCustomers))

  #Set up allocation subproblem with CPLEX

  env   <- openEnvCPLEX()                          #Open CPLEX environment
  prob  <- initProbCPLEX(env)                      #Assign problem
  nc    <- sizePeriods*sizeCustomers               #Number of variables
  nr    <- sizePeriods + sizeCustomers             #Number of constraints
  rhs   <- c(rep(1,sizeCustomers), n)              #Allocation subproblem restriction rhs
  sense <- rep('E', nr)                            #All restrictions are equalities
  lb    <- rep(0, nc)                              #Nonnegative variables
  ub    <- rep(CPX_INFBOUND, nc)                   #Variables are not bounded on the upper side
  beg   <- 2*c(0:(nc-1))                           #Begin indices of rows
  cnt   <- rep(2, nc)                              #Number of non-zero elements per row
  val   <- rep(1, 2*nc)                            #Non-zero elements
  ind   <- c()                                     #Initialize column
```

```
    indices
for(i in c(0:(nc-1))){
  ind <- append(ind, i%%sizeCustomers)
  ind <- append(ind, sizeCustomers + floor(i/sizeCustomers))
}
obj   <- rep(0, nc)                                  #Initialize empty
   objective function

copyLpCPLEX(env, prob, nc, nr, CPX_MIN, obj, rhs, sense, beg, cnt,
    ind, val, lb, ub, NULL)
presolveCPLEX(env, prob, CPX_ALG_AUTOMATIC)          #Presolve
   objective-less problem

#Start problem

population <- init(sizePopulation, p, f, sizeFacAvailable,
   sizePeriods)
allo1 <- allocate(env, prob, population, c, f, sizeCustomers,
   sizePeriods)
fitness <- allo1[[1]]
# allocateElapsed <- allo1[[2]]

df <- data.frame(popu=population, fit=fitness)  #Load initialized
   and evaluated population
df <- df[order(unlist(df$fit)),]                     #Order population
   in ascending fitness
dfchild <- dfmutated <- df[1,]                       #Load dummy
   individuals for children and mutations

iter          <- 0
noBestIter    <- 0
bestFit       <- df$fit[1][[1]]
start         <- Sys.time()

while((difftime(Sys.time(), start, units = "secs") < maxTime) && (
   noBestIter < maxNoBestIter)){

  #Crossover section

  parentsIndex <- sample(1:sizePopulation, 2)                    #
     Randomly select parents
  child <- crossover(df, parentsIndex, sizePeriods)              #
     Create child by crossover of the parents
  allo2 <- allocate(env, prob, child, c, f, sizeCustomers,
                    sizePeriods, allocateElapsed)                #
                       Evaluate children fitness
  childFit <- allo2[[1]]
  # allocateElapsed <- allo2[[2]]
  dfchild <- data.frame(popu=child, fit=childFit)
  df <- selection(df, dfchild, parentIndex = parentsIndex[order(
     unlist(df$fit[parentsIndex]))[1]],
```

```
83                      minDiff, sizePeriods, sizePopulation)       #
                           Proceed with selection of child
84
85      #Mutation section
86
87      if(runif(1) < prMutate){
88        parentIndex <- sample(1:sizePopulation, 1)                #
             Randomly select individual to mutate
89        mutated <- mutation(df[parentIndex,1][[1]], mutRate, p,
90                            c(1:sizeFacAvailable), sizePeriods)   #
                               Create child by mutation of selected
                               individual
91        allo3 <- allocate(env, prob, mutated, c, f, sizeCustomers,
92                          sizePeriods,allocateElapsed)            #
                             Evaluate fitness of mutated child
93        mutatedFit <- allo3[[1]]
94        # allocateElapsed <- allo3[[2]]
95        dfmutated <- data.frame(popu=mutated, fit=mutatedFit)
96        df <- selection(df, dfmutated, parentIndex, minDiff,
97                        sizePeriods, sizePopulation)              #
                           Proceed with selection of mutated child
98      }
99
100     #Check if new best fitness achieved
101
102     if(bestFit > df$fit[[1]]){
103       bestFit <- df$fit[[1]]
104       noBestIter <- 0
105       bestTime <- difftime(Sys.time(), start, units = "secs")
106       cat(sprintf('In iteration %5d a new optimal minimum of %2f was
              achieved in %2f seconds\n',
107                   iter, bestFit, bestTime[[1]]))
108       # print('allocate')
109       # print(allocateElapsed)
110       # print('crossover')
111       # print(crossoverElapsed)
112       # print('mutate')
113       # print(mutateElapsed)
114       # print('select')
115       # print(selectElapsed)
116       # allocateElapsed <- 0
117       # start <- Sys.time()
118     }
119     else{
120       noBestIter <- noBestIter + 1
121     }
122     iter <- iter + 1
123   }
124   cat(sprintf('Stop citeria reached in %s iterations\n\n', iter))
125   return(list(df, bestTime))
126 }
```

Code/init.R

```
#This function initializes the population.
#The total number of facilities to open is randomly chosen between the min and max possible.

#The parameters are:
# sizePopulation:           Size of the population to initialize
# p:                        Vector with minimum of facilities to open per period
# f:                        Cost of running facility from period [[t]][[i]]
# sizeFacAvailable:         Number of available facilities by index
# sizePeriods:              Number of periods

init <- function(sizePopulation, p, f, sizeFacAvailable, sizePeriods){

  population <- I(vector("list", sizePopulation))

  for(i in 1:sizePopulation){

    population[[i]] <- I(vector("list", sizePeriods))

    if(i<sizePopulation/2){
                                                    #Total number of facilities that open
      totalFacOpen     <- sample(c(sum(p):sizeFacAvailable),1)
    }
    else{
      totalFacOpen     <- floor(sum(p)+(sizeFacAvailable-sum(p))*rbeta(1,1,5))
    }
    facOpenPer       <- p
                                                    #Facilities that open each period
    facNotUsed       <- c(1:sizeFacAvailable)

    if(totalFacOpen > sum(p)){
      for(k in 1:(totalFacOpen-sum(p))){
        moreFacInPer <- sample(1:sizePeriods,1)
        facOpenPer[moreFacInPer] <- facOpenPer[moreFacInPer] + 1
                #Randomly choose periods in which to open
      }

       #more than the minimum of facilities
    }
    for(t in 1:sizePeriods){
      if(length(facNotUsed) == 1){
                                                    #Catch necessary because sample(c(3),1) could be 2
        chosenFacInPer <- facNotUsed
```

```
37        }
38        else{
39          chosenFacInPer <- sample(facNotUsed,facOpenPer[t])
                        #Randomly choose unused facility to open in
                    period t
40        }
41        population[[i]][[t]] <- chosenFacInPer
42        facNotUsed <- facNotUsed[!facNotUsed %in% chosenFacInPer]
                  #Update used facilities
43      }
44    }
45
46    return(population)
47  }
```

Code/allocate.R

```
1  #This gives the optimal allocations of demand of the sets of
       facilities in the participants
2
3  #The parameters are:
4  # env:                      CPLEX environment
5  # prob:                     CPLEX problem preloaded with data
6  # participants:             Individuals to be allocated (assumed
       same dimensions)
7  # c:                        Assignment cost of customers to
       facilities by period [[t]][[j]][[i]]
8  # f:                        Cost of running facility from period [[t
       ]][[i]]
9  # sizePeriods:              Number of periods
10 # sizeCustomers:            Number of customers
11
12 allocate <- function(env, prob, participants, c, f, sizeCustomers,
       sizePeriods, allocateElapsed=0){
13
14   sizeAllocated <- length(participants)
15   nc            <- sizePeriods*sizeCustomers
16
17   fitness       <- I(vector("list", sizeAllocated))     #Initialize
       fitness vector
18   dAst <- d     <- I(vector("list", sizePeriods))       #Initialize
       objective and dummy objective vector
19
20   for(p in 1:sizeAllocated){
21
22     participant <- participants[[p]]
23
24     #The problem is uncapacitated and without reallocation costs
25     #The objective then is the sum of the cheapest facilities for
         customer j at each period from t to the end
26
27     for(t in 1:sizePeriods){
```

```
28        openFacilities <- unlist(participant[c(1:t)])
29
30        #Figure out cheapest opened facility at any given period for
              the customers
31
32        d[[t]] <- lapply(lapply(seq_len(ncol(c[[t]])), function(i) c[[
              t]][openFacilities,i]),min)
33      }
34
35      for(t in 1:sizePeriods){
36        dAst[[t]] <- rep(0,sizeCustomers)
37        for(per in t:sizePeriods){
38
39          #Find out assignation cost opening facility in period t (
                stays open to the end)
40
41          dAst[[t]] <- dAst[[t]] + unlist(d[[per]])
42        }
43      }
44
45      #allocateElapsedStart <- Sys.time()
46
47      obj <- unlist(dAst)
48
49      chgObjCPLEX(env, prob, nc, c(0:(nc-1)), obj)       #Change dummy
            objective to the calculated
50
51      lpoptCPLEX(env, prob)                              #Optimize
            subproblem with CPLEX
52      result <- solutionCPLEX(env, prob)
53
54      fitness[[p]] <- result$objval
55
56      #Add running facility cost to fitness (independent from
            allocation)
57
58      for(t in 1:sizePeriods){
59        if(length(participant[[t]]) > 0){
60          fitness[[p]] <- fitness[[p]] + sum(unlist(f[[t]][participant
                [[t]]]))
61        }
62      }
63      #allocateElapsed <- allocateElapsed + Sys.time() -
            allocateElapsedStart
64    }
65
66    solution      <- I(vector("list", 2))
67    solution[[1]] <- fitness
68
69    #solution[[2]] <- allocateElapsed
70    return(solution)
```

```
}
```

Code/crossover.R

```
#This inputs two random parents and creates a child.

#The parameters are:
# df:                       Dataframe with evaluated population
# parents:                  Indices of two individuals of the
   population with fitness
# sizePeriods:              Number of periods

crossover <- function(df, parents, sizePeriods){
  orderPar <- df[parents,][order(unlist(df[parents,]$fit)),]
     #Order parents to find alpha and beta

  parAlpha  <- orderPar[1,1][[1]]
  parBeta   <- orderPar[2,1][[1]]

  child     <- parAlpha
     #Child is initialized as alpha parent

  t         <- sample(1:sizePeriods, 1)
     #Randomly choose one period to do crossover

  inter     <- intersect(parAlpha[[t]],parBeta[[t]])
     #Find out equal facilities in chosen period

  parAlphaDist  <- parAlpha[[t]][!parAlpha[[t]] %in% inter]
     #Find out alpha's distinct facilities
  parBetaDist   <- parBeta[[t]][!parBeta[[t]] %in% inter]
     #Find out beta's distinct facilities

  lenA <- length(parAlphaDist)
  lenB <- length(parBetaDist)

  if(min(lenA,lenB) > 0){
     #Permutation of same-period facilities is only
    for(k in 1:min(lenA,lenB)){
       #possible if both parents have distinct facilities
      i <- parAlphaDist[sample(1:length(parAlphaDist),1)]
         #Find random distinct facility of alpha i
      j <- parBetaDist[sample(1:length(parBetaDist),1)]
         #Find random distinct facility of beta j
      child[[t]][match(i, child[[t]])] <- j
         #Substitute child's i by j

      for(tprime in c(1:sizePeriods)[-t]){
         #Look for j in all of other periods
        if((j %in% child[[tprime]])){
          child[[tprime]][match(j, child[[tprime]])] <- i
             #Substitute found child's j by i
```

```
35          }
36        }
37
38        parAlphaDist  <- parAlphaDist[!parAlphaDist %in% i]
            #Update choices for facility permutation
39        parBetaDist   <- parBetaDist[!parBetaDist %in% j]
40      }
41    }
42
43    if(length(parAlphaDist) > 0){
        #All remaining distinct facilities in alpha
44      for(k in 1:length(parAlphaDist)){
          #are moved to other periods
45        child[[t]]  <- child[[t]][-match(parAlphaDist[k],child[[t]])]
            #Remove facility from t
46        tprime      <- sample(c(1:sizePeriods)[-t], 1)
            #Randomly choose period to move facility to
47        child[[tprime]] <- append(child[[tprime]], parAlphaDist[k])
            #Append facility to new period
48      }
49    }
50    #if(length(unlist(child))!=length(unique(unlist(child)))){browser
        ()}
51    childAst      <- I(vector("list", 1))
52    childAst[[1]] <- child
53
54    return(childAst)
55  }
```

Code/mutation.R

```
1  #This mutates the received individual.
2
3  #The parameters are:
4  # parent:                 The individual to mutate
5  # mutRate:                The amount of genes to mutate (
     facilities to change)
6  # p:                      Vector with minimum of facilities to
     open per period
7  # facAvailable:           Available facilities by index
8  # sizePeriods:            Number of periods
9
10 mutation <- function(parent, mutRate, p, facAvailable, sizePeriods){
11   mutated <- parent                                          #Start
       mutated individual as parent
12   nonZeroPeriods <- which(lengths(mutated) != 0)             #Find
       out periods where facilities are opened
13
14   mutRate <- sample(c((max(mutRate-1,1):min(mutRate+1,sum(p)))),1)
           #Random  feasible mutation (bounded)
15   mutRate <- min(mutRate, sum(lengths(mutated)[nonZeroPeriods] - p))
         #Avoids unfeasible states
```

```
16
17   for(k in 1:mutRate){
18     t <- sample(nonZeroPeriods, 1)                          #Select
         random period on which to mutate a gene
19
20     iIndex  <- sample(1:length(mutated[[t]]), 1)            #Find
         gene i to mutate (facility to change)
21     i       <- mutated[[t]][iIndex]
22
23     if(runif(1) < p[t]/length(mutated[[t]])){               #
         Permutation more likely with less open excess facilities
24       if((runif(1) < 1/(sizePeriods))){                     #
           Permutation with unused facility
25         notUsed <- facAvailable[-unlist(mutated)]           #A
             little overkill, but global iIndex not calculated
26         if(length(notUsed) == 0){next()}                    #Quicker
             than setting and updating global notUsed
27         if(length(notUsed) == 1){                           #Catch
             necessary because sample(c(3),1) could be 2
28           mutated[[t]][iIndex] <- notUsed
29         }
30         else{
31           mutated[[t]][iIndex] <- sample(notUsed, 1)        #
               Substitute i with randomly chosen unused facility
32         }
33       }
34       else{                                                 #
           Permutation with facility in other period
35         if(length(nonZeroPeriods) == 2){
36           tprime  <- nonZeroPeriods[-t]
37         }
38         else{
39           tprime  <- sample(nonZeroPeriods[-t], 1)          #
               Randomly choose other period
40         }
41
42         jIndex    <- sample(1:length(mutated[[tprime]]),1)  #
             Randomly choose facility j in tprime
43         j         <- mutated[[tprime]][jIndex]
44         mutated[[t]][iIndex]      <- j                      #
             Substitute i by j in original period
45         mutated[[tprime]][jIndex] <- i                      #
             Substitute j by i in the other period
46       }
47     }
48     else{                                                   #Move or
         remove
49       mutated[[t]]        <- mutated[[t]][-iIndex]          #Remove
           i from original period
50
51       if(runif(1) > 1/(sizePeriods)){
```

```
52          tprime              <- sample(c(1:sizePeriods)[-t],1) #
               Randomly choose period to move i to
53          mutated[[tprime]] <- append(mutated[[tprime]], i)    #Append
               i to chosen period
54        }
55
56        nonZeroPeriods      <- which(lengths(mutated) != 0)    #Update
             periods when facilities are opened
57      }
58    }
59    #if(length(unlist(mutated))!=length(unique(unlist(mutated)))){
         browser()}
60    mutatedAst      <- I(vector("list", 1))
61    mutatedAst[[1]] <- mutated
62
63    return(mutatedAst)
64  }
```

Code/selection.R

```
1  #This decides whether to introduce a new individual or not.
2
3  #The parameters are:
4  # df:                       Main population df
5  # dfCandidate:              Database with new candidate
6  # parentIndex:              Parent against which to compare
      candidate (optional)
7  # minDiff:                  Minimum difference between individuals
8  # sizePeriods:              Size of the periods
9  # sizePopulation:           Size of the population to initialize
10
11 selection <- function(df, dfCandidate, parentIndex = 1, minDiff,
      sizePeriods, sizePopulation){
12
13   dist <- 0                                                    #
        Initialize Hamming distance
14
15   for(t in 1:sizePeriods){
16     interLen  <- length(intersect(df[[parentIndex,1]][[t]],
          dfCandidate[[1,1]][[t]]))      #Find number matching
          facilities
17     maxLen    <- max(length(df[[parentIndex,1]][[t]]),length(
          dfCandidate[[1,1]][[t]]))
18     dist      <- dist + maxLen - interLen                      #Since
          vector lengths may be different,
19   }                                                            #
        distance is counted against highest possible
20
21   if(df$fit[[parentIndex]] >= dfCandidate$fit[[1]]){           #
        Candidate is only selected if it surpasses parent
22     if(dist > minDiff){                                        #
          Candidate is significantly different from parent
```

```
23          df[sizePopulation,] <- dfCandidate[1,]                    # Candidate substitutes worst individual
24        }
25        else{                                                       # Candidate is too similar to parent
26          df[parentIndex,] <- dfCandidate[1,]                       # Candidate substitutes parent
27        }
28        df <- df[order(unlist(df$fit)),]                            # Population is reordered
29      }
30
31      return(df)
32    }
```

Bibliography

[1] ALBAREDA-SAMBOLA, M., ALONSO-AYUSO, A., ESCUDERO, L. F., FERNÁNDEZ, E., HINOJOSA, Y., AND PIZARRO-ROMERO, C. A computational comparison of several formulations for the multi-period incremental service facility location problem. *TOP 18*, 1 (2010), 62–80.

[2] ALBAREDA-SAMBOLA, M., FERNÁNDEZ, E., HINOJOSA, Y., AND PUERTO, J. The multi-period incremental service facility location problem. *Computers & Operations Research 36*, 5 (2009), 1356–1375.

[3] ALP, O., ERKUT, E., AND DREZNER, Z. An efficient genetic algorithm for the p-median problem. *Annals of Operations research 122*, 1-4 (2003), 21–42.

[4] ALUMUR, S. A., NICKEL, S., SALDANHA-DA GAMA, F., AND VERTER, V. Multi-period reverse logistics network design. *European Journal of Operational Research 220*, 1 (2012), 67–78.

[5] BOZKAYA, B., ZHANG, J., AND ERKUT, E. An efficient genetic algorithm for the p-median problem. *Facility location: Applications and theory* (2002), 179–205.

[6] CANEL, C., AND KHUMAWALA, B. M. Multi-period international facilities location: An algorithm and application. *International Journal of Production Research 35*, 7 (1997), 1891–1910.

[7] CAVALIER, T. M., AND SHERALI, H. D. Sequential location-allocation problems on chains and trees with probabilistic link demands. *Mathematical programming 32*, 3 (1985), 249–277.

[8] COOPER, L. Location-allocation problems. *Operations research 11*, 3 (1963), 331–343.

[9] CORREA, E. S., STEINER, M. T. A., FREITAS, A. A., AND CARNIERI, C. A genetic algorithm for solving a capacitated p-median problem. *Numerical Algorithms 35*, 2-4 (2004), 373–388.

[10] DREZNER, Z. Dynamic facility location: The progressive p-median problem. *Location Science 3*, 1 (1995), 1–7.

[11] DREZNER, Z., AND WESOLOWSKY, G. Facility location when demand is time dependent. *Naval Research Logistics (NRL) 38*, 5 (1991), 763–777.

[12] FRIEDBERG, R. M. A learning machine: Part i. *IBM Journal of Research and Development 2*, 1 (1958), 2–13.

[13] GOLDBERG, D. E. Messy genetic algorithms: Motivation analysis, and first results. *Complex systems 4* (1989), 415–444.

[14] HAKIMI, S. L. Optimum locations of switching centers and the absolute centers and medians of a graph. *Operations research 12*, 3 (1964), 450–459.

[15] HAKIMI, S. L., LABBÉ, M., AND SCHMEICHEL, E. F. Locations on time-varying networks. *Networks: An International Journal 34*, 4 (1999), 250–257.

[16] HEINEN, F. *Über Systeme von Kraeften, deren Intensitaeten sich wie die n. Potenzen der Entfernungen gebenener Punkte von einem Central-Punkte verhalten: in Beziehung auf Punkte, für welche die Summe der n. Entfernungspotenzen ein Maximum oder Minimum ist*. Baedeker, 1834.

[17] HINOJOSA, Y., PUERTO, J., AND FERNÁNDEZ, F. R. A multiperiod two-echelon multicommodity capacitated plant location problem. *European Journal of Operational Research 123*, 2 (2000), 271–291.

[18] HOLLAND, J. H. *Nonlinear environments permitting efficient adaptation*. New York: Academic, 1967.

[19] HOSAGE, C., AND GOODCHILD, M. Discrete space location-allocation solutions from genetic algorithms. *Annals of Operations Research 6*, 2 (1986), 35–46.

[20] KARIV, O., AND HAKIMI, S. L. An algorithmic approach to network location problems. ii: The p-medians. *SIAM Journal on Applied Mathematics 37*, 3 (1979), 539–560.

[21] LAPORTE, G., NICKEL, S., AND DA GAMA, F. S. *Location science*. Springer, 2016.

[22] LAUNHARDT, W., AND BEWLEY, A. *The Theory of the Trace: Being a Discussion of the Principles of Location*. Lawrence Asylum Press, 1900.

[23] LI, X., XIAO, N., CLARAMUNT, C., AND LIN, H. Initialization strategies to enhancing the performance of genetic algorithms for the p-median problem. *Computers & Industrial Engineering 61*, 4 (2011), 1024–1034.

[24] LIM, A., AND XU, Z. A fixed-length subset genetic algorithm for the p-median problem. In *Genetic and Evolutionary Computation Conference* (2003), Springer, pp. 1596–1597.

[25] LORENA, L. A. N., AND FURTADO, J. C. Constructive genetic algorithm for clustering problems. *Evolutionary Computation 9*, 3 (2001), 309–327.

[26] MANIEZZO, V., MINGOZZI, A., AND BALDACCI, R. A bionomic approach to the capacitated p-median problem. *Journal of Heuristics 4*, 3 (1998), 263–280.

[27] MLADENOVIĆ, N., BRIMBERG, J., HANSEN, P., AND MORENO-PÉREZ, J. A. The p-median problem: A survey of metaheuristic approaches. *European Journal of Operational Research 179*, 3 (2007), 927–939.

[28] PEREZ, J. M., GARCIA, J. R., AND MORENO, M. A parallel genetic algorithm for the discrete p-median problem. *Studies in Locational Analysis 7* (1994), 131–141.

[29] REVELLE, C. S., AND SWAIN, R. W. Central facilities location. *Geographical analysis 2*, 1 (1970), 30–42.

[30] ROODMAN, G. M., AND SCHWARZ, L. B. Extensions of the multi-period facility phase-out model: New procedures and application to a phase-in/phase-out problem. *AIIE Transactions 9*, 1 (1977), 103–107.

[31] SALCEDO-SANZ, S., PORTILLA-FIGUERAS, J. A., ORTIZ-GARCÍA, E. G., PÉREZ-BELLIDO, A. M., THRAVES, C., FERNÁNDEZ-ANTA, A., AND YAO, X. Optimal switch location in mobile communication networks using hybrid genetic algorithms. *Applied Soft Computing 8*, 4 (2008), 1486–1497.

[32] SALHI, S., AND GAMAL, M. A genetic algorithm based approach for the uncapacitated continuous location–allocation problem. *Annals of Operations Research 123*, 1-4 (2003), 203–222.

[33] SCOTT, A. J. Dynamic location-allocation systems: some basic planning strategies. *Environment and Planning A 3*, 1 (1971), 73–82.

[34] SHULMAN, A. An algorithm for solving dynamic capacitated plant location problems with discrete expansion sizes. *Operations research 39*, 3 (1991), 423–436.

[35] SWEENEY, D. J., AND TATHAM, R. L. An improved long-run model for multiple warehouse location. *Management Science 22*, 7 (1976), 748–758.

[36] WEBER, A. *Uber den standort der industrien*. University of Chicago Press, 1929.

[37] WESOLOWSKY, G. O. Dynamic facility location. *Management Science 19*, 11 (1973), 1241–1248.

[38] WESOLOWSKY, G. O., AND TRUSCOTT, W. G. The multiperiod location-allocation problem with relocation of facilities. *Management Science 22*, 1 (1975), 57–65.