A Memetic Algorithm for the Facility Location Problem[*]

Renata Sarmet Smiderle Mendes
Mário César San Felice
Universidade Federal de São Carlos
Rod. Washington Luiz, km 235, São Carlos - SP, Brazil
renatassmendes@gmail.com
felice@ufscar.br

Pedro Henrique Del Bianco Hokama
Universidade Federal de Itajubá
Av. BPS, 1303 - Pinheirinho, Itajubá - MG, Brazil
hokama@unifei.edu.br

Regina Berretta
Pablo Moscato
The University of Newcastle
University Dr, Callaghan NSW 2308, Australia
regina.berretta@newcastle.edu.au
pablo.moscato@newcastle.edu.au

ABSTRACT

The Facility Location Problem seeks to decide how many and which facilities to open in order to serve the demands of a set of clients. It is a very relevant problem due to its theoretical interest, being an NP-hard problem widely studied, and for which numerous approximation algorithms and metaheuristics are proposed. It is also relevant for being motivated by practical applications, modeling problems such as plant positioning, construction of computer networks and information clustering.

In this work we present a memetic algorithm (MA) and a late acceptance algorithm for the uncapacitated facility location problem (UFLP). The computational experiments show that the MA achieves better solutions when compared with approximation algorithms and other metaheuristics.

KEYWORDS. Uncapacitated Facility Location Problem. Memetic Algorithms. Late Acceptance Algorithm.

Paper topics (Combinatorial Optimization; Metaheuristics)

[*]Supported by FAPESP 2018/13083-5, FAPESP 2019/16276-1 and CNPq 435617/2018-4.

1. Introduction

In the Uncapacitated Facility Location Problem (UFL), there is a set of clients D and a set of facilities F. For each j in D and i in F, there is a cost c_{ij} to assign client j to facility i, which we can interpret as the distance between them. Besides that, there is an opening cost f_i for each facility i in F. The goal of this problem is to choose a subset of facilities $F' \subseteq F$ that minimizes the total cost of opening the facilities in F' plus the cost of assigning each client j in D to the closest open facility. In other words, we want to find F' which minimizes

$$\sum_{i \in F'} f_i + \sum_{j \in D} \min_{i \in F'} c_{ij} \quad .$$

The first part is the facility cost and the second part is the assignment or connection cost. At the end, it is said that the facilities in F' are open.

The UFL is an NP-hard problem, which means that it cannot be solved in polynomial time, unless P $=$ NP. One way to get around this difficulty is to use approximation algorithms and another way is to use metaheuristics. The UFL is important in logistics and can be applied in several scenarios, such as to define the location of schools, supermarkets, hospitals, among other facilities, according to the distribution of the users of these services. The UFL can also be used in other scenarios, such as to model the construction of a wired computer network, in which the personal computers are the clients, the servers are the facilities and, therefore, the connection cost increases linearly with distance, while the cost of the facilities corresponds to the price of the servers.

The metric version of the UFL (MUFL), in which the connection costs respect the triangular inequality, is particularly interesting because, while it still is relevant in practice, it has approximation algorithms with much better guarantees than the general case. In fact, while the UFL has no constant approximation algorithm, there are several constant ratio approximation algorithms for the MUFL, such as the algorithm based on the deterministic rounding of linear programming by Shmoys et al. [1997], the algorithm that uses the primal-dual method by Jain and Vazirani [2001], the greedy algorithm proposed by Jain et al. [2003], an algorithm based on randomized rounding by Chudak and Shmoys [2003], and a local search algorithm by Charikar and Guha [2005].

Several metaheuristics were proposed for the UFL, including the genetic algorithm by Kratica et al. [2001], simulated annealing by Aydin and Fogarty [2004], tabu search by Sun [2006], discrete particle swarm optimization by Guner and Sevkli [2008], artificial bee colony optimization by Tunçbilek et al. [2012] and Watanabe et al. [2015], unconscious search by Ardjmand et al. [2014], ant colony optimization by Kole et al. [2014], firefly algorithm by Tsuya et al. [2017] and the monkey algorithm by Atta et al. [2018].

Our contributions. In this work we implemented a late acceptance algorithm for the UFL, based on the framework of Burke and Bykov [2017]. Moreover, we designed a memetic algorithm for the UFL, inspired by Harris et al. [2015]. In particular, this latter algorithm uses both approximation algorithms and search heuristics to, respectively, find initial solutions and improve the solutions found during its generations.

2. Late Acceptance

In this subsection we present our late acceptance algorithm for the UFL problem, based on the late acceptance hill-climbing heuristic, proposed by Burke and Bykov [2017]. Late Acceptance (LA) is a metaheuristic derived from Local Search (LS). In LA, y_j defines whether the facility j is open or not. A move is defined to change this value, i.e, $y_j \leftarrow 1 - y_j$. Thus, a move can close a facility that is currently open or can open a facility that is currently closed. The only restriction on a candidate move is that facility j cannot be closed if it is the only one open. In both LS and LA, given an initial solution, we consider its neighborhood, which is the set of solutions that diverge from it by only one move, and a move is applied to go to a new solution within the neighborhood. However, unlike LS, the LA does not stop when it is out of moves that improve the current solution. In order to escape from local minima, LA accepts non-improving moves when the cost of the candidate solution is better than that of the solution considered a number of iterations (lh) before. The number lh is an input parameter which impacts the total processing time of the search procedure.

Let Δz_j^k represent the resulting solution cost change of the move switching the status of facility j at iteration k. The lower the value of Δz_j^k, the more attractive is the move. The value of Δz_j^k is computed or updated for each $j \in F$ before the move is selected. The details of the computation or update of Δz_j^k's are described by Sun [2006]. In order to accept moves whose cost is better than that of a solution considered a number of iterations before, LA maintains a list, called f_a, with a fixed length (lh) with the values of previous solutions. At each iteration, we select a facility that minimizes the solution cost change of the move Δz_{j_k}. We always accept non-worsening moves and moves that are better than the total cost of the solution lh iterations before, which is stored at f_a.

A difference from our implementation to the algorithm proposed by Burke and Bykov [2017] is that we associate a flag to each facility, marking those which do not satisfy any of the previous accepting criteria, and we cannot choose moves that are flagged. We did this in order to avoid falling in a loop by always choosing the same move, since we are using the best fit technique to choose it. When a move satisfies the accepting criteria, we make this move, update the current solution and check if it is better than the best one found so far, updating it, if it is the case. We also update each Δz_{j_k} and mark all facilities as unflagged to restart the choice of the move. Then, we check we update the current total cost at the f_a list.

We continue this search until two stopping criteria are satisfied: maximum number of iterations in which the current solution may not improve, and number of iterations without improving the best solution found so far.

3. Memetic Algorithm

In this subsection we present a memetic algorithm (MA) for the UFL problem, inspired by the algorithm proposed by Harris et al. [2015] for the Quadratic Assignment Problem. The MA procedure is presented after its components are discussed.

Similar to the Genetic Algorithm (GA), MA is a population-based metaheuristic. Basically, it is an evolutionary algorithm that consists of an initial population composed of chromosomes (individuals), each one representing a viable solution to the problem. At each iteration, the algorithm generates new individuals using crossover and mutation operators. Crossover is the operation that generates a new individual (called offspring), through the exchange of genes from two individuals, now called parents. Mutation is the operation that modifies the status of a facility, i.e., it changes the status between open and closed.

Unlike GA, however, MA uses extra knowledge to improve the solutions in the population, as explained by Neri et al. [2011]. For instance, it can use different techniques, as heuristics, other metaheuristics or exact methods after mutation and/or crossover to improve some members of the population.

3.1. Greedy Algorithms and Local Search

In this work, we use several heuristics to improve the solutions in the population of our MA. Following, we cite them:

- Local Search (LS), our implementation of the 2.414-approximation algorithm proposed by Charikar and Guha [2005];
- Late Acceptance (LA), described in Section 2;
- Map Greedy, a random procedure which generates solutions through the following steps. First, we choose a subset F' of the facilities using one of two random approaches: sampled or shuffled. In the former each facility is sampled uniformly at random with a fixed probability. In the latter we build a permutation of the facilities using Knuth shuffle, which chooses each permutation evenly, and we use a subsequence of this permutation on each solution, to ensure that each facility is considered at least once. This decision must be established before starting. Then, we connect each client to its closest facility in F', and we remove from F' any unused facility. Finally, an instance is built only with the facilities remaining in F' and the Greedy Algorithm (Gr), by Jain et al. [2003], runs on this instance to produce our solution;
- Tabu Search (TS), a simplification of the one proposed by Sun [2006]. In TS we use only the short-term memory, i.e, it stores a list of forbidden moves (called tabu list). If no better solution is found in the neighborhood and if the new solution is not forbid by the tabu list, it may perform moves for neighbor solutions which are worse than the current solution.
- Map Tabu Search, similar to Map Greedy which uses TS instead of Gr;

3.2. Population Structure

An individual corresponds to an array of booleans, called $open\_facilities$, with size equal to $|F|$, representing whether each facility is open or not.

The population is structured as a complete ternary tree with 13 nodes, as shown in Figure 1. Each node is composed of two individuals, the *pocket* and the *current*. A specific individual ind is identified in the tree by $nodes[i][j]$, where i is the node it resides at and j indicates whether it is stored in the pocket (if $j = 0$) or in the current (if $j = 1$). Within the tree, there are also subpopulations, i.e, groups of 4 nodes, one parent and three offspring. The subpopulations are the nodes $\{0, 1, 2, 3\}$, $\{1, 4, 5, 6\}$, $\{2, 7, 8, 9\}$ and $\{3, 10, 11, 12\}$.

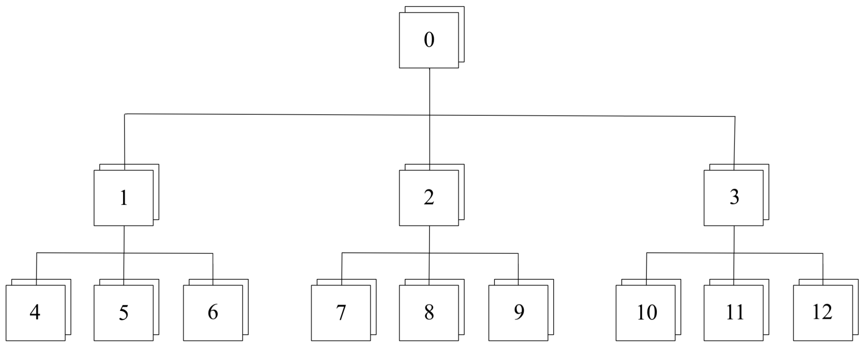

Figure 1: The 3-layered ternary tree, with the nodes labelled. At each node there are 2 individuals, which represent 1 pocket and 1 current.

3.3. Updating population

There is a pre-established order in the tree structure that respects the following two rules:

- In every node, the pocket solution never costs more than the current one of this same node;
- The pocket of the parent node never has a solution cost greater than the pocket one of its respective offspring.

To maintain this order, each time an operation that modifies the individuals is performed, firstly we do a check within each node to ensure that the pocket is better than the current one, swapping otherwise. Then, the nodes of each subtree are compared, starting from the bottom and raising the best solutions, until reaching the root of the tree, which should always keep in its pocket the best solution present in the population. To find even better solutions, we run the TS when the pocket in node 0 is updated.

3.4. Population Initialization

At the beginning of the algorithm, the pocket and current of every node must be initialized with an initial solution. This is done as follows:

- The pocket of node 1 is initialized with a solution generated by the 1.61-approximation greedy algorithm (Gr), proposed by Jain et al. [2003];
- The pocket of node 0 (root) is initialized with a solution generated by the LS, using the pocket of node 1 as initial solution;
- Every other pocket and current are initialized using the random procedure Map Greedy.

After the generation of the initial population, we run the function that updates the population, establishing the order in which the best solutions are above in the tree.

3.5. Crossover and Mutation

The crossover and the mutation operations are done using information from the boolean array ($open\_facilities$), which indicates which facilities are open. Four different crossovers were implemented in order to make comparisons between them: the Uniform Crossover, the One Point Crossover, the Groups Crossover and the Union Crossover. The crossover to be used must be established before the algorithm starts.

- The **Uniform Crossover** copies for the offspring what is the same in both parents and draws what is different, with a 50% chance of opening or not the facility;
- The **One Point Crossover** draws uniformly at random a number n (from 0 to $|F|$) and then copies the information from one parent on the first n facilities and the rest from the other parent;
- The **Groups Crossover** identifies clients which are connected to a common facility in both parents and defines them as a group. If a group was connected to the same facility in both parents, this facility is open in the offspring. Otherwise, we find the facility which minimizes the cost of connecting that group to and open it in the offspring;
- The **Union Crossover** copies all open facilities of each parent to the offspring, keeping the others closed;

In order to restore lost or unexplored genetic material into the population, trying to prevent the premature convergence of the MA to suboptimal solutions, we modify the status of a small percentage of the facilities through the mutation operator.

We establish the number of facilities ($qty\_mutation$) to be mutated through a parameter. Thus, a number (that ranges under $|F|$) is chosen uniformly at random and the facility, with index in $open\_facilities$ corresponding to the chosen number has its status inverted, i.e., if it was open it closes, and if it was closed it opens. We repeat this process $qty\_mutation$ times.

The mutation operator is applied after every crossover except the Groups Crossover, since it already had a higher computational cost to determine which facilities will be open, in addition to not copying that information entirely directly from the parents.

After the crossover and the mutation, each client must be connected to the nearest open facility and the facilities that ended up not being connected to anyone should be closed. The final cost must also be updated.

3.6. Next Generation

In order to produce a new individual, first we must apply the crossover operator to a pair of pocket and current individuals, then apply the mutation operator to the offspring, as described in Section 3.5. Finally, we must apply a heuristic, chosen from those presented in Section 3.1, to improve the resulting individual. To produce the next generation we must produce new individuals in a specific order, according to their position in the tree.

Consider a subpopulation, as described in Section 3.2, with a parent (p) and three descendants (c1, c2, c3). Also, let current[i] and pocket[i] be, respectively, the current and the pocket individuals of node i. We produce the next generation's current individuals of a subpopulation as following:

- current[p] = produce(pocket[p], current[c1])
- current[c3] = produce(pocket[c3], current[p])
- current[c1] = produce(pocket[c1], current[c2])
- current[c2] = produce(pocket[c2], current[c3])

We compute the new individuals of each subpopulation from top to bottom. If a produced individual is equal to another one in the tree, then it is replaced by a random individual generated by Map Greedy. After that, the population must be updated, as described in Section 3.3.

In order to decide which heuristic we use to improve each individual, two strategies were established, called S1 and S2. In the first strategy (S1), as LA usually obtains better solutions than Map TS, but also has a longer computational time, it is applied only to the first subpopulation, composed of nodes $\{0, 1, 2, 3\}$, and then Map TS is applied to the other subpopulations from the tree. This is illustrated in Figure 2.

Even better than LA, but also with a longer computational time, we have the TS. Thus, to use it without harming computational time too much, the second strategy (S2) works with probabilities to choose which search algorithm to apply: 60% chance of applying Map TS, 30% chance of applying LA, and 10% chance of applying TS. The other heuristics considered to be used were discarded after extensive tests. The choice of which strategy will be used must also be made before the algorithm starts.

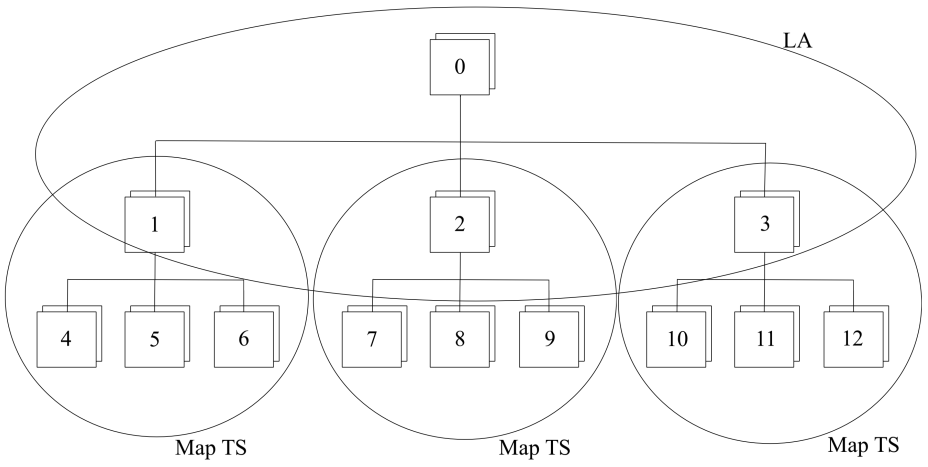

Figure 2: First strategy (S1), LA is applied to the first subtree, while Map TS is applied to the others.

3.7. Overall MA

The algorithm has several input parameters to indicate: (i) the percentage of facilities that will change status after the crossover; (ii) which crossover to use (Uniform, One Point, Union or Groups); (iii) whether it should use strategy S1 or S2; (iv) the probability of opening a facility when generating random solutions; (v) whether it will generate random solutions by sampling or shuffling; (vi) the number of generations without improvement before the root is updated; (vii) the number of times in a row that the root can be updated without improving the best saved solution before the algorithm halts. These two last parameters are our stopping criteria.

The algorithm initializes the population as described in Section 3.4 and it produces new generations as described in Section 3.6. If we achieve the limit number of generations without improvement before the root is updated, we change the root. To do so, the descendant of the root with the lowest cost value will rise to the root's place and a new random individual is generated using Map Greedy and improved by the application of the LA on it, to replace the space that became empty. After this process, updates to the population must be applied to ensure that the pockets of each node remain better than currents and that the highest lift nodes have better solutions. The algorithm ends when this root exchange process achieves the stopping criteria.

4. Computational Results

We used the 90 instances of the Koerkel-Ghosh symmetric and asymmetric packages. All of these instances were taken from the Max Planck Institut Informatik website[1]. The Koerkel-Ghosh symmetric and Koerkel-Ghosh asymmetric packages have instances with the same number of customers and facilities, ranging from 250, 500 and 750, with the first package having symmetric connection matrices while the second does not. Each group of instances with the same size is divided in three classes: A, B and C. Opening costs are drawn uniformly at random from [100, 200] in class A, from [1000, 2000] in class B and from [10000, 20000] in class C. More details on the instances of each package can be found on the website from which they were taken.

All experiments were performed in a computer with an Intel Core i5 processor, operating at a frequency of 2.4GHz with 8GB RAM, and the codes are implemented in C++. This section describes some of the computational results of these implementations with the mentioned instances.

[1]http://resources.mpi-inf.mpg.de/departments/d1/projects/benchmarks/UflLib/index.html

The complete table with the results of all tests and all codes mentioned in this section can be found on github[2].

For each algorithm, we compare the cost of the solution found with the best upper bound (BUB), from Karapetyan and Goldengorin [2017], and we show the time spent for those instances.

4.1. Late Acceptance

A series of tests were performed with the LA algorithm, varying the parameters over a huge number of configurations. To compare the configurations, the first 5 instances of size 250 and the first 5 instances of size 500 were used, both from the asymmetric package.

At each iteration, to choose which facility we will try to make a move, we consider two techniques: the Best Fit (BF) and the First Feasible (FF). In the BF, we select a facility that has the minimum extra cost Δz_j^k and that is not flagged. In the FF, we select the first facility that has a feasible move, i.e., it cannot be the only one open.

For BF, a cycle was perceived (ranging among few facilities) until the f_a vector of size lh was completely filled in. Then, it became similar to LS ($lh = 1$ is equal to LS from the beginning). So, it was noted that a larger lh is of no use, as it stays in the cycle and generates the same results as a small lh. For FF, it was not possible to conclude which is best, since it varied according to the size of the instance. Thus, we opted to use an lh relative to the respective number of facilities.

All 90 instances were tested with $lh = [0.05, 0.1, 0.5, 1]$ multiplied by the number of facilities for FF and $lh = 10$ fixed for BF. For the stopping criterion parameters, we considered $\alpha_1 = 2.5$ and $limit\_idle = 0.02$. The average results of these tests can be seen in Table 1. The first column is the configuration, the second column is the percentage of the difference between the cost of the solution found and the BUB (best upper bound), and the third column is the average time spent (in seconds) to run the LA algorithm.

Table 1: LA tests comparing configurations

Configuration	**Cost $\%$ BUB**	**Time (s)**
FF, $lh = 0.05$ (relative)	0.103%	1.61
FF, $lh = 0.01$ (relative)	0.101%	2.59
FF, $lh = 0.5$ (relative)	0.099%	7.69
FF, $lh = 1$ (relative)	0.097%	8.18
BF, $lh = 10$ (fixed)	0.082%	2.54

It was concluded, then, that the BF technique with $lh = 10$, stop criterion $\alpha_1 = 2.5$ and $limit\_idle = 0.02$ is the best configuration considering cost benefit and this configuration was adopted for the future uses of this algorithm.

Improvements were made to the code, alternating the programming paradigm, and all the instances with the chosen configuration were tested with the new version of the code. The comparison of the results obtained by it with the BUB can be seen in Figure 3 and the time spent to both generate the initial solution with Gr and to run the LA algorithm can be seen in Figure 4. The average of the results of these tests can be seen in Table 2, where the first column is the group of instances with the same size and symmetry status, the second column is the percentage of the difference between the average of the cost of the solutions in this group and the BUB, and the third column is the average of the time spent (in seconds) on this group to run the LA algorithm.

Most instances have obtained solutions with the cost up to 0.3% from the BUB, with only a few instances slightly larger than that, the maximum being 0.59% from BUB. Analyzing the time,

[2]https://github.com/renatasarmet/ICAlgoritmos

Table 2: Late Acceptance tests

Group Instance	Cost % BUB	Time (s)
ga250	0.112%	1.04
ga500	0.097%	7.51
ga750	0.056%	24.85
gs250	0.077%	1.01
gs500	0.082%	7.51
gs750	0.069%	25.1
Average	0.082%	11.17

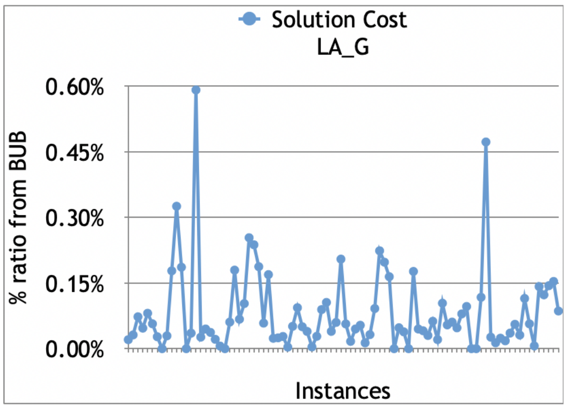

Figure 3: Comparison of the costs achieved by the LA algorithm with the BUB.

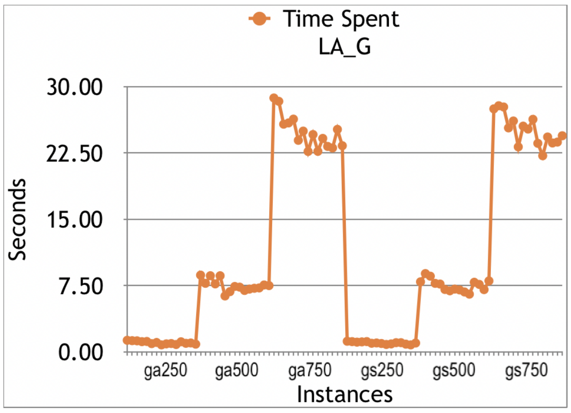

Figure 4: Time to build the initial solution with the Greedy and to run the LA algorithm.

we may see that the instance size is the main factor, while being symmetrical or not does not affect much.

4.2. Memetic Algorithm

In this subsection a summary of the tests performed with the memetic algorithm is described. Seven different instances were chosen to be used in the initial tests, representing both the symmetrical and asymmetric packages, the 3 types of size and also the 3 classes (A, B and C). All parameters were tested in different combinations, including the tree structure, which started with 5 pockets and 1 current, similar to the one proposed by Harris et al. [2015], and ended as described in Section 3.2.

The 4 crossovers described in section 3.5 were tested, all with the generation of the initial population being both sampled and shuffled (in addition to the pockets of the first nodes with greedy and LS, as described in 3.4). After several tests varying each parameter, it has been decided that the percentage of facilities to open would be 10% of the total, mutation rate 1%, maximum 5 generations without improvement to change the root and maximum 2 times in a row that the root has been changed without improvement in the best solution found to stop the search.

For crossovers 1 and 2, we tested the application of LA, TS and the Map TS after the crossover, and for crossovers 3 and 4, LA, TS, Map TS, in addition to the Map Greedy for crossover 3. With that, it was verified that the application of TS after each crossover generates better solutions. However, TS has a much higher cost then the Map TS, which improves the solution just a little. The LA is in the middle, both in cost improvement and in time spent. Thus, we designed strategies S1

Table 3: MA - Crossover 2 and S2

Group Instance	Cost $\%$ BUB	Time (s)
ga250	0.0000%	19.7
ga500	0.0052%	139.3
ga750	0.0008%	425.5
gs250	0.0001%	18.1
gs500	0.0006%	133.7
gs750	0.0011%	435.5
Average	0.0013%	195.1

Table 4: MA - Crossover 1 and S2

Group Instance	Cost $\%$ BUB	Time (s)
ga250	0.0000%	17.4
ga500	0.0002%	129.1
ga750	0.0003%	417.7
gs250	0.0000%	17.9
gs500	0.0000%	124.4
gs750	0.0003%	456.4
Average	0.0001%	193.8

and S2, described in Section 3.6, to combine different search algorithms after each crossover and take advantage of the good things that each one can offer.

In addition, it was noticed that, although crossovers 3 and 4 are more elaborate and more intelligent of the problem, they took more computational time and did not generate better solutions than simple crossovers 1 and 2, when accompanied by search algorithms. Therefore, more tests were determined to be performed. Both crossovers 1 and 2 of the crossover, with the initial solutions both shuffled and random and with the strategies S1 and S2. All of these combinations, in order to reach a conclusion which would be the one to be used.

A table with all these final tests can be found on github[3]. The tests with S1 in general were much faster, but the solutions were worse. Finally, two tests were performed with all instances, one of them being crossover 2, with random initial solutions and strategy S2, and the other with crossover 1, with shuffled initial solutions and strategy S1. The average results of these tests can be seen at Table 3 and Table 4, respectively. The first column is the group of instances with the same size and symmetry status, the second column is the percentage of the difference between the average of the cost of the solutions in this group and the BUB, and the third column is the average of the time spent (in seconds) for this group to run the MA.

It was concluded, then, that the crossover 1 with shuffled initial solutions and strategy S2 is the best configuration considering cost benefit. The comparison of the results obtained by it with the BUB can be seen in Figure 5 and the time spent to run the memetic algorithm can be seen in Figure 6.

For most instances it has obtained solutions with gap really close to 0.0% comparing to the BUB, with only a few instances slightly larger than that, the maximum being 0.0035%. Analyzing the time, instances of size 250 took around 17 seconds, those of size 500 took around 130 seconds and those of size 750 around 440 seconds, maintaining this pattern even though alternating between symmetrical and asymmetrical instances.

4.3. Overall Results

Using all instances of packages Koerkel-Ghosh symmetric and asymmetric, Table 5 contains the average difference in cost obtained by each algorithm and by the Best Upper Bound (BUB), Karapetyan and Goldengorin [2017], as well as the respective time spent.

As we can see, the Memetic Algorithm was the one that found the best solutions, being very close to the BUB, but it also spent much more time than the others. On the other hand, the Late Acceptance spent almost the same time as the LS, but got slightly worse results, probably due to the LS using a more sophisticated neighborhood. And, finally, the Tabu Search spent, on average, just a little more time than the other search heuristics and obtained better solutions than those.

[3]https://github.com/renatasarmet/ICAlgoritmos/blob/master/memetic_tests.pdf

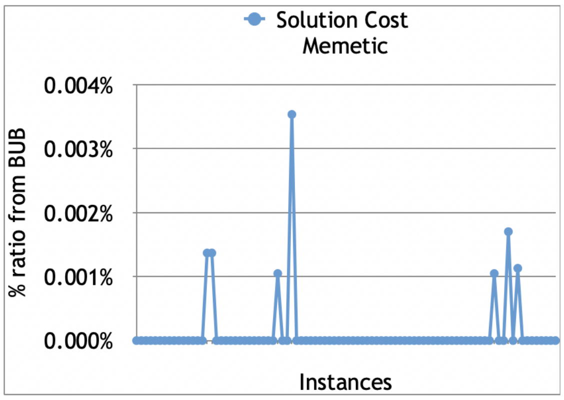

Figure 5: Comparison of the costs achieved by the MA with the BUB.

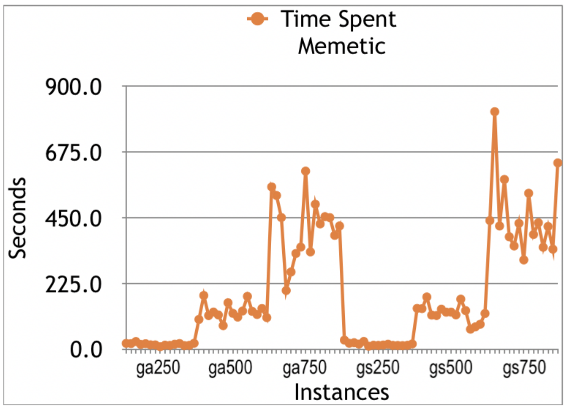

Figure 6: The time to run the Memetic Algorithm.

Table 5: Average test results for each algorithm.

Algorithm	**Cost % BUB**	**Time (s)**
Local Search	0.0580%	11.28
Tabu Search	0.0245%	12.30
Late Acceptance	0.0822%	11.17
Memetic Algorithm	0.0001%	193.82

5. Conclusion

In this work we implemented a late acceptance algorithm for the UFL, based on the framework of Burke and Bykov [2017]. Moreover, we designed a memetic algorithm for the UFL, inspired by Harris et al. [2015]. In particular, this latter algorithm uses approximation algorithms, search heuristics and randomized algorithms to find initial solutions and improve the solutions found during its generations.

The performance of both algorithms was compared with an implementation of the local search by Charikar and Guha [2005] and a simplification of the tabu search by Sun [2006]. Our proposed memetic algorithm found excellent solutions with costs very close to the Best Upper Bound from Karapetyan and Goldengorin [2017]. For all the 90 instances of the Koerkel-Ghosh symmetric and asymmetric packages used, it obtained results on average 0.0001% from the best upper bound. However, depending on the size of the instance, it also had an expensive computational cost. On the other hand, search heuristics, such as the late acceptance, achieved good results in a more reasonable time.

References

Ardjmand, E., Park, N., Weckman, G., and Amin-Naseri, M. R. (2014). The discrete unconscious search and its application to uncapacitated facility location problem. *Computers & Industrial Engineering*, 73:32 – 40. ISSN 0360-8352.

Atta, S., Sinha Mahapatra, P. R., and Mukhopadhyay, A. (2018). *Solving Uncapacitated Facility Location Problem Using Monkey Algorithm*, p. 71–78. ISBN 978-981-10-7565-0.

Aydin, M. E. and Fogarty, T. C. (2004). A distributed evolutionary simulated annealing algorithm for combinatorial optimisation problems. *Journal of Heuristics*, 10:269–292.

Burke, E. K. and Bykov, Y. (2017). The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70 – 78. ISSN 0377-2217.

Charikar, M. and Guha, S. (2005). Improved combinatorial algorithms for facility location problems. *SIAM Journal on Computing*, 34(4):803–824.

Chudak, F. A. and Shmoys, D. B. (2003). Improved approximation algorithms for the uncapacitated facility location problem. *SIAM Journal on Computing*, 33(1):1–25.

Guner, A. and Sevkli, M. (2008). A discrete particle swarm optimization algorithm for uncapacitated facility location problem. *Journal of Artificial Evolution and Applications*, 2008.

Harris, M., Berretta, R., Inostroza-Ponta, M., and Moscato, P. (2015). A memetic algorithm for the quadratic assignment problem with parallel local search. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, p. 838–845.

Jain, K., Mahdian, M., Markakis, E., Saberi, A., and Vazirani, V. V. (2003). Greedy facility location algorithms analyzed using dual fitting with factor-revealing lp. *J. ACM*, 50(6):795–824.

Jain, K. and Vazirani, V. V. (2001). Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *J. ACM*, 48(2): 274–296.

Karapetyan, D. and Goldengorin, B. (2017). Conditional markov chain search for the simple plant location problem improves upper bounds on twelve körkel-ghosh instances.

Kole, A., Chakrabarti, P., and Bhattacharyya, S. (2014). An ant colony optimization algorithm for uncapacitated facility location problem. *Artificial Intelligence and Applications*, 2014:55–61.

Kratica, J., Tošic, D., Filipović, V., and Ljubić, I. (2001). Solving the simple plant location problem by genetic algorithm. *RAIRO - Operations Research - Recherche Opérationnelle*, 35(1):127–142.

Neri, F., Cotta, C., and Moscato, P. (2011). *Handbook of memetic algorithms*, volume 379. Springer.

Shmoys, D. B., Tardos, E., and Aardal, K. (1997). Approximation algorithms for facility location problems (extended abstract). In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, p. 265–274, New York, NY, USA. ACM.

Sun, M. (2006). Solving the uncapacitated facility location problem using tabu search. *Computers & Operations Research*, p. 2563–2589.

Tsuya, K., Takaya, M., and Yamamura, A. (2017). Application of the firefly algorithm to the uncapacitated facility location problem. *Journal of Intelligent & Fuzzy Systems*, 32:3201–3208.

Tunçbilek, N., Tasgetiren, F., and Esnaf, S. (2012). Artificial bee colony optimization algorithm for uncapacitated facility location problems.

Watanabe, Y., Takaya, M., and Yamamura, A. (2015). Fitness function in abc algorithm for uncapacitated facility location problem.