Locations, Cities and Centres

Applying Integer Linear Programming and metaheuristic procedures for combinatorial optimisation problem solving

AMMM

Course Project

Lluís Alemany Puig

13th June 2017 - Q2

This report is a slight modification of the one delivered as part of the course's project.

Contents

1 Formal Statement

An Internet retail company wants to build several logistic centres in order to operate in a new country. Its goal is to spend the minimum amount of money while making sure that customers receive their products quickly enough.

The company has a set of L locations $\mathcal{L} = \{\mathbb{l}_1, \cdots, \mathbb{l}_L\}$ where logistic centres can be installed in, and a set of C cities $\mathcal{C} = \{\mathbb{c}_1, \cdots, \mathbb{c}_C\}$ that need to be served. For each location we know its coordinates $u_l = (u_{l,x}, u_{l,y}) \in \mathbb{R}^2$ $(\forall l : 1 \leq l \leq L)$, and for every city we know its coordinates $v_c = (v_{c,x}, v_{c,y}) \in \mathbb{R}^2$ and its population $p_c \in \mathbb{N}$ $(\forall c : 1 \leq c \leq C)$. We have available a set of T logistic centre types $\mathcal{T} = \{\mathbb{t}_1, \cdots, \mathbb{t}_T\}$. Each type is characterised by a working distance $\omega_t \in \mathbb{R}$, capacity $s_t \in \mathbb{N}$ and installation cost $i_t \in \mathbb{R}$ $(\forall t : 1 \leq t \leq T)$.

The goal is to know the type of the centres to be installed in the different locations in order to serve all the different cities as quickly and as cheaply as possible. Not all logistic centres types need to be used, and not all locations have to have one centre installed. If a location requires a centre to be installed in it then it can be at most one centre of one type but many centres of the same type can be installed in more than one location (that is, different locations may have a centre of the same type installed in them). However, centres can only be installed in those locations that are at least at a distance of D to the rest of locations that have a centre installed in them.

The installed centre of type $\mathbb{t}_t$ in a location $\mathbb{l}_l$ may be acting as a primary or as a secondary centre when serving the different cities $\mathbb{c}_c$, and not necessarily as primary or as secondary all the time: it may serve as a primary centre to some of the cities and as a secondary centre to the rest. All cities need to be served exactly by a primary centre and a secondary centre. From now on, when a location $\mathbb{l}_l$ has a centre of type $\mathbb{t}_t$ installed in it that serves some cities as a primary centre and some other cities as a secondary centre we will say that that location is serving those cities with a primary and with a secondary role, respectively. Now, a location may serve many cities - at least one if it has a centre installed and none if no centre is installed in it - but it can never serve the same city with a primary and a secondary role at the same time. Besides, a location $\mathbb{l}_l$ with a centre of type $\mathbb{t}_t$ installed in it can only serve cities $\mathbb{c}_c$ with a primary role if the distance between the cities and the location is less than the working distance of the centre installed: $d(\mathbb{l}_l, \mathbb{c}_c) \leq \omega_t$. But, if the location is serving cities with a secondary role then the distance between the location and the cities can be at most three times the working distance of that centre: $d(\mathbb{l}_l, \mathbb{c}_c) \leq 3\omega_t$. One last constraint is that the sum of the population of those cities $\mathbb{c}_c$ served by a location $\mathbb{l}_l$ (that has a centre of type $\mathbb{t}_t$ installed in it) with a primary role plus 10% of the sum of the population of those cities served by the same location with a secondary role can not exceed the capacity of the centre s_t.

2 Integer Linear Programming formulation

In this section is described all the information regarding the ILP formulation of the problem: the variables, objective function, the constraints to model the problem, and the implementation in OPL.

2.1 Input data

The input data that makes up an instance of the problem, extracted from the statement, is referenced in the OPL code using the following names:

- The ranges $1, \cdots, L$, $1, \cdots, C$ and $1, \cdots, T$, where L, C and T are the number of locations, cities and centre types respectively:

```
range L = 1..nLocations;
range C = 1..nCities;
range T = 1..nCentres;
```

- Location coordinates u_l

```
float loc_x[l in L];
float loc_y[l in L];
```

- City coordinates v_c:

```
float city_x[c in C];
float city_y[c in C];
```

- City population p_c:

```
int city_pop[c in C];
```

- Centre's working distance ω_t:

```
float work_dist[t in T];
```

- Centre's capacity s_t:

```
int centre_cap[t in T];
```

- Centre's installation cost i_t:

```
float instal_cost[t in T];
```

- Distances between locations, and between locations and cities.

```
float dist_ll[l1 in L][l2 in L];
float dist_lc[l in L][c in C];
```

The values of each of these variables are read from the input with the only exception of the last two. These are calculated in a preprocessing block and are meant to make the code more readable. Here is the preprocessing block:

```
execute {
    for (var l1 = 1; l1 <= nLocations; ++l1) {
        for (var l2 = 1; l2 <= nLocations; ++l2) {
            dist_ll[l1][l2] = Math.sqrt(
                Math.pow(loc_x[l1] - loc_x[l2], 2) +
                Math.pow(loc_y[l1] - loc_y[l2], 2)
            );
        }
```

```
        for (var c = 1; c <= nCities; ++c) {
            dist_lc[l1][c] = Math.sqrt(
                Math.pow(loc_x[l1] - city_x[c], 2) +
                Math.pow(loc_y[l1] - city_y[c], 2)
            );
        }
    }
}
```

2.2 Decision variables

For the ILP formulation of this problem the following decision variables have been defined. The name each of these decision variables is given and the OPL code is also specified.

1. $I_{l,t}$: "centre of type $\mathbb{t}_t$ is installed in location $\mathbb{l}_l$" - ($\mathbb{B}$, $\forall l, t : 1 \leq l \leq L, 1 \leq t \leq T$)

   ```
   dvar boolean location_centre[l in L][t in T];
   ```

2. $LP_{l,c}$: "location $\mathbb{l}_l$ serves city $\mathbb{c}_c$ with a primary role" - ($\mathbb{B}$, $\forall l, c : 1 \leq l \leq L,\ 1 \leq c \leq C$)

   ```
   dvar boolean location_pcity[l in L][c in C];
   ```

3. $LS_{l,c}$: "location $\mathbb{l}_l$ serves city $\mathbb{c}_c$ with a secondary role" - ($\mathbb{B}$, $\forall l, c : 1 \leq l \leq L,\ 1 \leq c \leq C$)

   ```
   dvar boolean location_scity[l in L][c in C];
   ```

2.3 Objective function

The value to minimise is the cost of centre installation. Therefore, we will be using the following objective function:

$$\text{minimise} \sum_{t=1}^{T} \left(i_t \cdot \sum_{l=1}^{L} I_{l,t} \right) \tag{1}$$

that involves the installed centre types $I_{l,t}$ and the corresponding installation cost i_t.

The implementation in OPL is the following:

```
minimize sum (t in T) instal_cost[t]*( sum (l in L) location_centre[l][t] );
```

2.4 Constraints

The problem is solved using the following constraints.

2. A location can have at most one centre installed.

$$\sum_{t=1}^{T} I_{l,t} \leq 1 \qquad \forall l : 1 \leq l \leq L \tag{2}$$

```
forall (l in L) {
    sum (t in T) location_centre[l][t] <= 1;
}
```

3. If a location is serving one or more cities then it must have a centre installed.

$$C \cdot \sum_{t=1}^{T} I_{l,t} \geq \sum_{c=1}^{C} (LP_{l,c} + LS_{l,c}) \qquad \forall l : 1 \leq l \leq L \tag{3}$$

```
forall (l in L) {
    nCities*( (sum (t in T) location_centre[l][t]) )
    >=
    (sum (c in C) (location_pcity[l][c] + location_scity[l][c]));
}
```

This constraint is basically modelling the following implication:

$$\sum_{c=1}^{C} LP_{l,c} + LS_{l,c} \geq 1 \Longrightarrow I_{l,t} = 1$$

4. Each city has to be served exactly by two locations: one with a primary role and another with a secondary role.

$$\sum_{l=1}^{L} LP_{l,c} = 1, \quad \sum_{l=1}^{L} LS_{l,c} = 1 \qquad \forall c : 1 \leq c \leq C \tag{4}$$

```
forall (c in C) {
    sum (l in L) location_pcity[l][c] == 1;
    sum (l in L) location_scity[l][c] == 1;
}
```

5. A location can not serve a city as both primary and secondary centre.

$$LP_{l,c} + LS_{l,c} \leq 1 \qquad \forall l, c : 1 \leq l \leq L,\ 1 \leq c \leq C \tag{5}$$

```
forall (l in L, c in C) {
    location_pcity[l][c] + location_scity[l][c] <= 1;
}
```

This is modelling the following implications:

$$LP_{l,c} = 1 \iff LS_{l,c} = 0, \qquad LP_{l,c} = 0 \iff LS_{l,c} = 1$$

6. The sum of the population of those cities a location serves with a primary centre plus 10% of the sum of the population of those cities it serves with a secondary centre can not exceed its capacity.

$$\sum_{c=1}^{C} LP_{l,c} \cdot p_c + 0.10 \cdot \sum_{c=1}^{C} LS_{l,c} \cdot p_c \leq \sum_{t=1}^{T} I_{l,t} \cdot s_t \qquad \forall l : 1 \leq l \leq L \tag{6}$$

```
forall (l in L) {
    sum (c in C) location_pcity[l][c]*city_pop[c]
    +
    0.10*( sum (c in C) location_scity[l][c]*city_pop[c] )
    <=
    sum (t in T) location_centre[l][t]*centre_cap[t];
}
```

Assume P_l and S_l to be the sets of cities a location $l \in \mathcal{L}$ serves with a primary and a secondary role respectively. Formally, using the previously defined boolean variables:

$$P_l = \{c \in \mathcal{C} \mid LP_{l,c} = 1\}, S_l = \{c \in \mathcal{C} \mid LS_{l,c} = 1\}$$

Constraint 6 is modelling the following implication:

$$I_{l,t} = 1 \implies \sum_{c \in P_l} p_c + 0.10 \cdot \sum_{c \in S_l} p_c \leq \omega_t$$

$\forall l, t : 1 \leq l \leq L,\ 1 \leq t \leq T.$

7. Centres cannot be installed in locations that are at a distance smaller than D.

$$\sum_{t=1}^{T} I_{l_1,t} + I_{l_2,t} \leq 1 \qquad \forall l_1, l_2 : 1 \leq l_1 < l_2 \leq L, :\ d_{l_1,l_2} < D \tag{7}$$

where $d^2_{l_1,l_2} = (u_{l_1,x} - u_{l_2,x})^2 + (u_{l_1,y} - u_{l_2,y})^2$ is the squared distance between the locations l_1 and l_2.

```
forall (l1 in L, l2 in L : l1 < l2) {
    if (dist_ll[l1][l2] < D) {
        sum (t in T) (location_centre[l1][t] + location_centre[l2][t]) <= 1;
    }
}
```

This is modelling the following implication:

$$d_{l_1,l_2} < D \implies (I_{l_1,t} = 0 \wedge I_{l_2,t} = 1) \vee (I_{l_1,t} = 1 \wedge I_{l_2,t} = 0)$$

$\forall l_1, l_2, t : 1 \leq l_1 < l_2 \leq L,\ 1 \leq t \leq T.$

8. The distance between the location and the cities it serves with a primary role can not be greater than the working distance of that centre.

$$LP_{l,c} \cdot d_{l,c} \leq \sum_{t=1}^{T} I_{l,t} \cdot \omega_t \qquad \forall l, c : 1 \leq l \leq L,\ 1 \leq c \leq C \tag{8}$$

where $d^2_{l,c} = (u_{l,x} - v_{c,x})^2 + (u_{l,y} - v_{c,y})^2$ is the squared distance between location l and city c.

```
forall (l in L, c in C) {
    location_pcity[l][c]*dist_lc[l][c]
    <=
    sum (t in T) location_centre[l][t]*work_dist[t];
}
```

This constraint is modelling the following implication:

$$I_{l,t} = 1 \wedge LP_{l,c} = 1 \implies d_{l,c} \leq \omega_t$$

$\forall l, c, t : 1 \leq l \leq L,\ 1 \leq c \leq C,\ 1 \leq t \leq T.$

9. The distance between the centre and the cities it serves with a secondary role can not be greater than three times the working distance of that centre.

$$LS_{l,c} \cdot d_{l,c} \leq \sum_{t=1}^{T} I_{l,t} \cdot 3\omega_t \qquad \forall l, c : 1 \leq l \leq L,\ 1 \leq c \leq C \tag{9}$$

where $d_{l,c}^2 = (u_{l,x} - v_{c,x})^2 + (u_{l,y} - v_{c,y})^2$ is the distance between the location l and the city c.

```
forall (l in L, c in C) {
    location_scity[l][c]*dist_lc[l][c]
    <=
    sum (t in T) location_centre[l][t]*3*work_dist[t];
}
```

This constraint is modelling the following implication:

$$I_{l,t} = 1 \wedge LS_{l,c} = 1 \Longrightarrow d_{l,c} \leq 3\omega_t$$

$\forall l, c, t : 1 \leq l \leq L,\ 1 \leq c \leq C,\ 1 \leq t \leq T$.

3 Metaheuristics

In the following subsections is described the code implemented to solve the problem using heuristics and metaheuristics, that is, the description of the code that actually implements the heuristic framework, and the different heuristics used applied in the Local Search, GRASP and BRKGA metaheuristics.

3.1 Code

In order to be able to implement, apply and compare different heuristics, random constructors and chromosome decoders easily, an abstract class named *problem* was implemented and given a series of virtual methods. These methods will be implemented in subclasses and each of them has a particular purpose: solve the problem using local search only, with or without a random constructor or using genetic algorithms, ... This class will be used by the other classes that implement the different metaheuristic algorithms, described in subsection 3.1.3, and has no attributes.

3.1.1 Random number generator

Since part of this project requires randomly generated values for the GRASP and the BRKGA metaheuristics, a simple interface was designed to test different random number generators. This interface is the abstract class *random_generator*.

This is implemented using the C++ 11's ¡random¿ header. It is a template that takes two parameters: the random engine, and the type of the numbers to be generated. This is a virtual class from which two other classes inherit its public and protected members. These two classes are specialised for generating discrete random values and continuous random values respectively.

3.1.2 The *problem* abstract class

The different functions that make up the *problem* class can be classified into the following categories.

Constructing a solution

```
// Constructs an empty solution.
virtual problem *empty() const = 0;

// Constructs a solution from scratch. It uses a greedy algorithm.
// Returns the evaluation of the solution.
virtual double greedy_construct()
    throw(infeasible_exception) = 0;

// Explores this solution's neighbourhood and stores:
// - the best neighbour if BI is true (best improvement)
// - the first best neighbour if BI is false (first improvement)
// Best means a neighbour that maximizes the evaluate() function.
virtual void best_neighbour(pair<problem *, double>& best_neighbour,
    const local_search_policy& p = Best_Improvement) = 0;

// Constructs a randomized solution using a restricted candidate
// list (RCL) sorted using the parameter alpha.
// Returns the evaluation of the solution.
virtual double random_construct(random_number_generator *rng, double alpha)
    throw(infeasible_exception) = 0;

// Constructs a solution from a given chromosome.
// Returns the evaluation of the solution.
virtual double decode(const chromosome& c)
    throw(infeasible_exception) = 0;
```

Evaluating a solution

```
// Evaluates the instance of this problem returning a scalar value
// representing its cost.
virtual double evaluate() const = 0;
```

Debugging a solution

```
// Writes into the output stream the instance of this problem.
virtual void print(const string& tab = "", ostream& os = cout) const = 0;

// Checks all constraints regarding solution feasibility. Returns
// true if the solution is feasible. Returns false otherwise.
virtual bool sanity_check(const string& tab = "", ostream& os = cerr) const = 0;
```

Memory handling functions

```
// Creates a copy of the instance of this problem.
virtual problem *clone() const = 0;

// Creates a copy of the instance of the problem passed as parameter.
virtual void copy(const problem *p) = 0;

// Clears the memory used by the instance of this problem.
// Everything is reset so that the method construct would create
// a solution to the problem if it were to be called.
virtual void clear() = 0;
```

More details can be found in the documentation.

3.1.3 Metaheuristics framework

The metaheuristics framework has been implemented in C++. A simple UML diagram describing the inheritance relationships between classes can be seen in figure 1. This framework was implemented this way so as to allow more flexibility in future projects. Each class uses the *problem* abstract class to model as generally as possible the different algorithms.

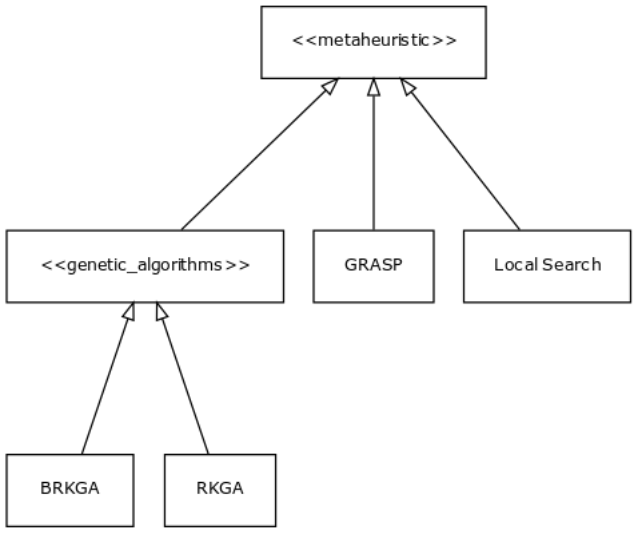


Figure 1: Simplified UML diagram describing the metaheuristics framework.

3.2 Preliminaries

In this section are described few notations and concepts for the sake of understandability of this text:

- Sets of cities served by location l: P_l and S_l. These two sets are defined in section 2.4, when defining the constraint 6 and represent the cities served by l with a primary and a secondary role respectively.
- Occupied capacity of a location: $K(l)$: this can be formally defined using the usual formula:

$$K(l) = \sum_{c \in P_l} p_c + 0.10 \cdot \sum_{c \in S_l} p_c \tag{10}$$

- Added capacity of a city c to a location l with a role r: $K_a(c, r)$. This measures the increment of the capacity in location l ($K(l)$) when making location l serve city c with a role r. This is formally defined as follows:

$$K_a(c, r) = \begin{cases} p_c, & r = \text{ primary} \\ 0.10 \cdot p_c, & r = \text{ secondary} \end{cases} \tag{11}$$

  Therefore:

$$K(l) = \sum_{c \in P_l} (K_a(c, primary) + \sum_{c \in S_l} K_a(c, secondary)$$

- Capacity gap in a location l with a centre installed when assigning a city c with a role r: G. This simply measures how much capacity is left in location l after it has been assigned to city c with role r, and can only be measured if that location has a centre t installed. G is formally defined as follows:

$$G(l, c, t, r) = c_t - (K(l) + K_a(c, r)) \tag{12}$$

  Notice that for the assignation of l to c with role r to be valid, G must be equal or greater than 0 ($G(l, c, t, r) \geq 0$).

Remember that p_c denotes the population of city c.

3.3 Greedy algorithms

In this section are described the different heuristics implemented to solve the problem using Local Search. This includes the greedy construction algorithm and the neighbourhood exploration algorithm.

3.3.1 Greedy construction algorithm

The greedy construction algorithm follows this basic procedure: while there are still cities lacking a primary or a secondary location, consider all possible candidates for assignation and take the one that minimises the cost of the greedy function. When the candidate that minimises this cost is found what remains is to add it to the solution being constructed. This step will be repeated until there are no more candidates left. A candidate for this algorithm is the tuple $< c, l, r >$, where $c \in \mathcal{C}$ is a city, $l \in \mathcal{L}$ is a location and r represents represents the role that location l will have when serving city c (that is either "primary" or "secondary"). It is worth being mentioned that we do not consider as valid candidates those that contain a location that is not far enough from the other locations with a centre installed and neither those with a location that is already serving the city in the candidate. Therefore, the candidates are "filtered" using the distance criterion and checking already assigned roles. Then they are evaluated using the usual capacity and distance (between city and location) constraints. The greedy constructor algorithm is, therefore, as follows:

Algorithm 1: Greedy construction

Input: *Problem* the specification of the problem
Output: S a possibly feasible solution to *Problem*

```
1  Function GreedyConstruct(Problem) is
2      S := ∅ empty solution
3      U := ∅ the set of locations used
4      k := 1, K := 2|C| the number of assignations to be made
5      while k ≤ K do
6          F := ∅ the candidate list
7          for < c, l > : c ∈ C, l ∈ L do
8              d := min{d(l, l') : l' ∈ L ∧ l' ≠ l ∧ l' has a centre installed}
9              if ¬(c ∈ P_l ∨ c ∈ S_l) ∧ d ≥ D then
10                 if c ∉ P_l then F := F ∪ {< c, l, primary >}
11                 if c ∉ S_l then F := F ∪ {< c, l, secondary >}
12         if F = ∅ then return Infeasible
13         m := MinCandidate(F, □)
14         S ← Assign(m_c, m_l, m_r)
15         U := U ∪ {m_l}, k := k + 1
16     S ← FindAndAssignCentres(U)
17     return S
```

The function in line 16 simply loops over all locations used l in U and finds the cheapest centre that can serve the cities assigned to each location. Even though the procedure is designed to terminate by returning infeasibility when a candidate can not be found (line 12), the existence of at least one centre type available for all locations when line 16 is reached is not guaranteed.

The symbol $\square$ denotes any greedy function. Several greedy cost functions were defined:

$$f_1(< c, l, r >) = \begin{cases} d(c,l), & \text{case 1} \\ +\infty, & \text{case 2} \\ d(c,l) \cdot i_t, & \text{case 3} \\ +\infty, & \text{otherwise} \end{cases} \tag{13}$$

$$f_2(< c, l, r >) = \begin{cases} K(l) + K_a(c,r), & \text{case 1} \\ +\infty, & \text{case 2} \\ (K(l) + K_a(c,r)) \cdot i_t, & \text{case 3} \\ +\infty, & \text{otherwise} \end{cases} \tag{14}$$

$$f_3(< c, l, r >) = \begin{cases} K(l) + K_a(c,r) + d(c,l), & \text{case 1} \\ +\infty, & \text{case 2} \\ (K(l) + K_a(c,r)) + d(c,l) \cdot i_t, & \text{case 3} \\ +\infty, & \text{otherwise} \end{cases} \tag{15}$$

Case 1 is found when location l is already serving some cities and can also serve city c with role r, case 2 is found when location l can not serve city c, whether it is already serving other cities or not, and case 3 when location l is not serving any other city and can serve city c with centre t installed in it. In case 3, centre t is the cheapest centre that can serve city c. $d(c,l)$ denotes the distance between city c and location l. Obviously, each of these functions lead to different in terms of the solution's cost, but it is only the last function (see greedy function 15) that makes this algorithm behave the best: the other two functions made it impossible for the algorithm to find centres available for some of the locations used (remember line 16) whereas the last seems to guarantee that, if the construction turns

out to be infeasible, it is because no feasible candidate was found. All experiments will be carried out using this function.

3.3.2 Neighbourhood exploration

Since we are dealing with a minimisation problem, a neighbour of a solution could be a very similar instance but with a centre installed less, or with a cheaper centre. That is, obtaining a new neighbour from a given solution consists on finding a location $l \in \mathcal{L}$, with a centre installed, removing it if possible and reassigning the cities it served to other locations with a centre installed. In case it is not possible to remove it, we will try to replace it with a cheaper one. Notice that it would not make much sense to try replacing a centre in the first place since removing it leads to a much better solution. And, only when it can not be removed does it make sense to try and replace it. The pseudocode of the algorithm is as follows:

Algorithm 2: Finding a neighbour

Input: S feasible solution, l location with a centre installed
Output: B a feasible solution with a cost equal to or smaller than S's

```
1  Function FindNeighbour(S, l) is
2      B := copy(S)
3      C_l := cities served by l
4      N := CanRemoveCentre(C_l, l)
5      if N ≠ ∅ then
6          for c ∈ C_l do
7              B ← Remove(c, l)
8              if c ∈ P_l then B ← Assign(c, N_c, primary)
9              else if c ∈ S_l then B ← Assign(c, N_c, secondary)
10     else
11         I_l := installed centre in location l
12         S := sort centres by installation cost
13         for t ∈ S do
14             if CanServeCities(t, P_l, S_l) ∧ i_t < i_{I_l} then
15                 B ← ReplaceCentre(I_l, t)
16                 Break
17     return B
```

We find in this algorithm some functions that need some clarification. To begin with, the function we find in line 4 (CanRemoveCentre) takes the cities location l is serving along with the appropriate role $\mathcal{C}_l$, and the location l itself. With these two parameters, this function tries to assign each of the cities in $\mathcal{C}_l$ to other locations with a centre already installed. These locations are then returned in a structure where, for each city, we find the new location that will serve it. If no locations are returned (line 5) then the algorithm wll try to replace the centre.

The typical constraints of distance and capacity have to be satisfied in order to choose a location for city c but the location chosen is, among those that satisfy these constraints, the closest in distance to the city and the one that maximises the capacity gap. Therefore, the new assignations are done greedily: that is, they are done in increasing order of added capacity.

Algorithm 3: Removing a centre

Input: $\mathcal{C}_l$ cities served by location l, l the location with the centre we are trying to remove
Output: N a set of new locations that can serve the cities in $\mathcal{C}_l$

```
1  Function CanRemoveCentre(C_l, l) is
2      N := ∅
3      Γ_l := sort cities in C_l by added capacity K_a
4      for c ∈ Γ_l do
5          Q := ∅
6          for l' ∈ L : l' has a centre installed do
7              if location l' satisfies all constraints then Q := Q ∪ {l'}
8          if Q ≠ ∅ then
9              m_l := MinDistGap(Q)
10             N := N ∪ {< c, m_l >}
11     return N
```

The functions Remove, Assign and ReplaceCentre found in lines 7, 8 and 9, and 15 of algorithm 2 respectively, are quite self-explanatory. They make a location stop serving a city, updating all data structures, assign a location to serve a city, also updating all necessary data structures, and replacing one centre installed for another, respectively.

The actual neighbourhood exploration algorithm is as follows: given an initial solution, for every pair of different locations with a centre installed in them - in the initial solution - try to apply the procedure described above. The returned value is called a neighbour. Depending on the policy of the exploration (either First-Improvement or Best-Improvement) the procedure will stop when it finds the first neighbour with lower cost than the initial solution or will explore all possible neighbours and will return the one with lowest cost.

Algorithm 4: Exploring the neighbourhood

Input: S feasible solution, *Policy* the neighbourhood exploration policy
Output: B a feasible solution with a cost equal to or smaller than S

```
1  Function NeighbourhoodExploration(S, Policy) is
2      B := copy(S)
3      for l ∈ L do
4          N := FindNeighbour(S, l)
5          if cost(N) < cost(B) then
6              B := N
7              if Policy = First-Improvement then return B
8      return B
```

3.4 GRASP - Randomised construction

In this section is described the random constructor that will be used by the GRASP metaheuristic [1]. This constructor basically follows the same procedure in the greedy constructor algorithm 1: it will construct the candidate list by using the same definition of what a candidate is and using the greedy cost function 15. Then, by using the parameter α it will construct the Restricted Candidate List, choose one of the candidates at random and add it to the solution. The following pseudocode models the usual procedure for doing this.

Algorithm 5: Randomised construction

Input: *Problem* the specification of the problem
Output: S a possibly feasible solution to *Problem*

Function RandomConstructor(*Problem*) **is**
  $S := \varnothing$ empty solution
  $U := \varnothing$ the set of locations used
  $k := 1$
  $K := 2|\mathcal{C}|$ the number of assignations to be made
  while $k \leq K$ **do**
    $F :=$ Build candidate list as in algorithm 1 with greedy function $\square$
    if $F = \varnothing$ **then return** Infeasible
    $s :=$ Min$(F, \square)$
    $S :=$ Max$(F, \square)$
    $RCL := \varnothing$ the Restricted Candidate List
    for $f \in F$ **do**
      if $\square_f \leq s + \alpha \cdot (S - s)$ **then** $RCL := RCL \cup \{f\}$
    $m :=$ select one element in the RCL at random
    $S \leftarrow$ Assign(m_c, m_l, m_r)
    $U := U \cup \{m_l\}$
    $k := k + 1$
  $S \leftarrow$ FindAndAssignCentres(U)
  return S

Again, the symbol $\square$ denotes any greedy function. The actual GRASP procedure is as follows:

Algorithm 6: Greedy Randomised Search Procedure

Input: *Problem* the specification of the problem, $I \in \mathbb{N}$ a number of iterations, *Policy* the policy of the local search procedure
Output: B a feasible solution to *Problem*

Function GRASP(*Problem*, I, *Policy*) **is**
  $B := \varnothing$ empty solution
  $i := 1$
  while $i \leq I$ **do**
    $R :=$RandomConstructor(*Problem*)
    $L :=$NeighbourhoodExploration$(R,$ *Policy*$)$
    if *cost*$(L) <$ *cost*(B) **then** $B := L$
    $i := i + 1$
  return B

We can consider that the cost of an empty solution is $+\infty$.

3.5 BRKGA - Chromosome decoder

In this section is described a simple algorithm that will be used by the BRKGA procedure [2]. BRKGA stands for Biased Random-Key Genetic Algorithm and is a procedure that, roughly, consists on finding the fittest individuals within a population and "crossing" them with not-so-fit individuals of the same population to produce a great variety of individuals, some of which are quite likely to be better than its parents, and, after repeating this step several times, find the fittest individual. To do that, individuals are represented with chromosomes, a set of genes, that need to be decoded into our solution in order to evaluate their fitness. This decoder algorithm is described later in this section in algorithm 9. The

BRKGA procedure works as follows:

Algorithm 7: Biased Random-Key Genetic Algorithm

Input: *Problem* the specification of the problem, $G \in \mathbb{N}$ a number of generations, s the size of each individual's chromosome, p the size of the population, p_e the size of the elite population, p_m the size of the mutant population, ρ_e the inheritance probability

Output: A feasible solution to *Problem*

```
1  Function BRKGA(Problem, G, s, p, p_e, p_m, ρ_e) is
2      P := InitializePopulation(p, s)
3      E := TrackEliteIndividuals(P, p_e)
4      g := 1
5      while g ≤ G do
6          M := MutantPopulation(p_m)
7          R := Crossover(E, P − E, p − p_e − p_m, ρ_e)
8          P := E ∪ M ∪ R
9          E := TrackEliteIndividuals(P)
10         g := g + 1
11     return FittestIndividual(E)
```

The procedure starts by initialising the population set, with a number p of individuals, each represented by a chromosome of size s, and tracks the p_e fittest individuals $\mathcal{E}$, (lines 2 and 3). Then, the procedure will proceed to produce the G generations of individuals, one depending on the previous one. A generation is produced by creating a set of p_m random individuals (the mutant set $\mathcal{M}$, line 6) and then cross some of them with a randomly chosen individual from $\mathcal{E}$ to produce the crossover population $\mathcal{R}$ (line 7). The new population set is then defined as the previous elite set of individuals, together with the mutant and the crossover populations (line 8). Since there are new individuals in this population, there may be new elite individuals that need to be kept track of. That is were we redefine the set $\mathcal{E}$, in line 9. Finally, the procedure returns the fittest individual of the elite set, that is also the fittest individual of the whole population.

The crossover is done by using the inheritance probability ρ_e: for $p - p_m - p_e$ non-elite individuals u, an elite individual e is randomly chosen and then they are crossed to create a new individual z. If σ_u, σ_e and σ_z are the chromosomes of the individuals respectively, the k-th gene of the newly generated individual, $\sigma_{z,k}$ is chosen between $\sigma_{u,k}$ and $\sigma_{e,k}$ according to the aforementioned inheritance probability ρ_e: if a randomly generated scalar $\phi \in [0, 1]$ is smaller than ρ_e then $\sigma_{e,k}$ is chosen. If it is greater then $\sigma_{u,k}$ is chosen instead.

Algorithm 8: Generating the crossover population

Input: $\mathcal{E}$ elite set of individuals, $\mathcal{F}$ non-elite set of individuals, p_c number of crossover individuals, ρ_e probability of inheritance

Output: $\mathcal{R}$ set of crossover individuals

```
1  Function BRKGA(E, F, p_c, ρ_e) is
2      R := ∅
3      r := 1
4      while r ≤ p_c do
5          e :=RandomChoice(E)
6          u :=RandomChoice(F)
7          z := ∅ new individual
8          k := 1
9          while k ≤ |σ_x| do
10             φ :=Random(0, 1)
11             if φ ≤ ρ_e then σ_{z,k} := σ_{e,k}
12             else σ_{z,k} := σ_{u,k}
13             k := k + 1
14         R := R ∪ {z}
15     return R
```

As it has been mentioned, finding the fittest individual relies on the fact that a chromosome σ has to be decoded into a feasible solution. The fitness of the individual is, then, the cost of the solution. In this case, the chromosome decoder is also based on a previous algorithm: the greedy constructor. This decoder will sort the chromosome so that the lowest values of the genes are placed at the beginning and the highest at the end. This yields an order of these chromosomes that will be used to construct a solution. The number of genes for each chromosome is equal to the number of cities in the instance $|\mathcal{C}|$. The indexes that represent the order of the genes in the chromosome are then interpreted as city indexes and then, for each city, we find the best location candidate according to some greedy function and assign it to the solution. Once all cities have been assigned a primary and a secondary location, remains only to assign centres to the locations used.

Algorithm 9: Randomised construction

Input: σ a chromosome, $|\sigma| = |\mathcal{C}|$
Output: S a possibly feasible solution obtained from σ

```
1  Function ChromosomeDecoder(σ) is
2      S := ∅ empty solution
3      U := ∅ the set of locations used
4      S_σ := {< g, σ_g > | < i, σ_i >∈ S_σ ∧ < j, σ_j >∈ S_σ ⟺ i < j → σ_i < σ_j} the sorted chromosome
5      k := 1
6      K := 2|C| the number of assignations to be made
7      while k ≤ K do
8          for < c, σ_c >∈ S_σ do
9              F := ∅ the candidate list
10             for l ∈ L do
11                 d := min{d(l, l') : l' ∈ L ∧ l' ≠ l ∧ l' has a centre installed}
12                 if ¬(c ∈ P_l ∨ c ∈ S_l) ∧ d ≥ D then
13                     if c ∉ P_l then F := F ∪ {< c, l, primary >}
14                     if c ∉ S_l then F := F ∪ {< c, l, secondary >}
15             if F = ∅ then return Infeasible
16             m := Min(F, □)
17             S ← Assign(m_c, m_l, m_r)
18             U := U ∪ {m_l}
19             k := k + 1
20     S ← FindAndAssignCentres(U)
21     return S
```

Notice the change of order of the instructions in lines 8 and 9 with respect to the greedy constructor algorithm: in this decoder they are interchanged and the instructions from line 16 to 19 are inside the for loop whereas in the greedy constructor they are outside, just right after it.

4 Comparative results

In this section the different algorithms designed for the metaheuristics will be compared to the ILP model for a few different instances of the problem. But first, we will start by briefly describing the instances used.

4.1 Description of the instances used

All the instances used to compare the different algorithms were purely randomly generated, that is, there was not any change made by hand. There are two sets of different inputs: the *big* set of inputs and the inputs generated by one of the students of the course *********, while the set of instances *big* were generated by the author of this report.

Some of the inputs in the *big* data set were generated following some rules: the locations of all of the inputs were generated with some randomised space between them (as a function of the parameter D) in order to ensure their feasibility for that matter, but only the cities of all the inputs, but two of them, were generated with some randomised space between them (as a function of D). This function is αD, where $\alpha \in \mathbb{R}$ is a uniformly distributed random number in the interval $[0, 1]$. All cities and locations were generated within certain ranges of values of x and y and were different for almost every input. The populations, working distances, installation costs, ..., were also randomly generated within certain ranges, again different for most inputs. One particularity worth being mentioned is that the capacities of the centres were generated slightly above the largest population for all centres. On the one hand, this way of generating the instances does not guarantee their feasibility, it only increases their chances to be so. On the other hand, this does make the instances easy to solve: there is not much variety when it comes to the types of centres used. However, they have been very useful to test the heuristics before switching to bigger and more difficult instances.

4.1.1 Optimal solutions and execution times

In the following tables are presented the optimal values of the objective function of the ILP formulation and the execution time needed to solve them. We also specify the number of cities, locations and centre types.

Instance name	# Locations	# Cities	# Centre types	ILP	Time (s)
$big-16$	32	14	3	1892.960	1.482
$big-17$	37	11	2	1354.040	3.182
$big-20$	49	11	3	536.200	2.095
$big-21$	42	21	8	582.132	2.532
$big-23$	31	18	8	127.876	1.724
$big-24$	40	23	6	3607.100	19.440
$big-25$	47	19	8	319.376	2.527
$big-26$	44	25	14	487.323	3.524
$big-29$	46	26	11	695.350	4.240
$big-30$	37	30	13	1655.110	3.660
$t-5$	22	50	18	88350.300	347.768
$t-6$	22	50	18	52254.000	4.305

Table 1: Size, optimal solution and time to reach it for each instance.

The inputs t-$\{5, 6\}$ are the only instances by ******** that will be used in this project.

4.2 Local Search

The local search procedure consists on two parts: the greedy construction algorithm and the neighbourhood exploration, explained in section 3.3. In the first place, after performing some experiments with instances of several sizes and difficulty, it seems that this construction procedure is good enough to reach optimality for those that are small, but not so much as soon as the size of the instance increases. However, this procedure is startlingly fast, a highly desirable feature for these kind of procedures.

Now, the neighbourhood exploration algorithm, in spite of always using the *Best* policy, does not produce the results we were expecting, that is, results close to the optimum solution. This is due to two main reasons: this procedure, when applied to solutions greedily constructed, does not improve them too much since locations already have the cheapest centres that can be installed hence making it impossible to either remove or replace them. The other reason is that, in spite of actually being capable of improving some instances, the speed at which this algorithm converges is too fast, that is, this algorithm finds very soon a dead-end in the feasible neighbourhood space, even when applied to randomly constructed instances (see section 4.3). Therefore, this procedure is made to stop when no improvement is made or when a maximum of iterations is reached.

See, for instance, the following tables, where is presented the performance of this procedure for several *big* inputs:

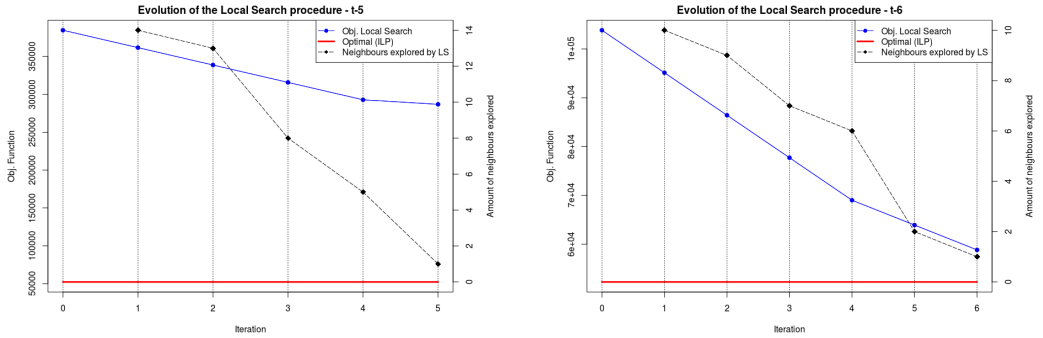
Instance name	Time (s)	Objective Function	Gap	Iteration	# Neighbours explored
$big-16$	0.000	2832.776	939.816	0	0
	0.000	2517.283	624.323	1	1
$big-17$	0.000	1943.056	589.016	0	0
$big-20$	0.000	612.800	76.600	0	0
	0.000	536.200	0.000	1	1
$big-25$	0.000	479.064	159.688	0	0
	0.000	439.142	119.766	1	6
	0.000	399.220	79.844	2	4
$big-30$	0.000	2373.196	718.086	0	0
	0.000	2137.012	481.902	1	4
	0.000	2026.671	371.561	2	3
	0.000	1916.330	261.220	3	1

Table 2: Traces extracted from the execution of the local search procedure. The *Gap* values represent the difference in absolute value between the optimal solution and the solution found using Local Search.

Notice how the previous description of these algorithms fits the data presented in table 2: there is a fast convergence of the algorithm (very few iterations until no improvement is made) and the size of the neighbourhood set is significantly small. In only one of the chosen inputs the optimal solution is reached (instance *big-20*). Moreover, there is one instance where the neighbourhood exploration does not produce any result (instance *big-17*).

Just to confirm what already seems to be true enough, in the following two figures is presented the same information for the instances *t-5* and *t-6*. In blue we find the value of the objective function obtained by the local search procedure in iterations $1, 2, ...$ In the iteration 0 the value corresponds to the solution constructed with the greedy constructor. This is bottomed by a red line with the value

of the optimal solution. The dotted line represents the number of neighbours explored. The execution time is always 0.



(a) Illustration of the trace for the input *t-5*.

(b) Illustration of the trace for the input *t-6*.

Figure 2: Two traces of the Local Search procedure.

Notice that the total amount of neighbours explored is greater when we use input *t-5*, another aspect of that input that may be used to explain its difficulty along with the big amount of time taken by the ILP solver.

For all the executions the policy of the neighbourhood exploration algorithm was *Best-Improvement*.

4.3 GRASP

The GRASP procedure is a simple procedure that, in general terms, constructs an instance randomly, improves it by using, in this case, a local search procedure, and is executed for a fixed number of iterations. The method described in section 3.4 for randomly constructing a solution poses a serious problem when it comes to generate a feasible instance: there is absolutely no guarantee that it is always feasible. When this happens, the method will skip that iteration and try to generate a feasible solution.

In figure 3 we see a short execution of the GRASP procedure:

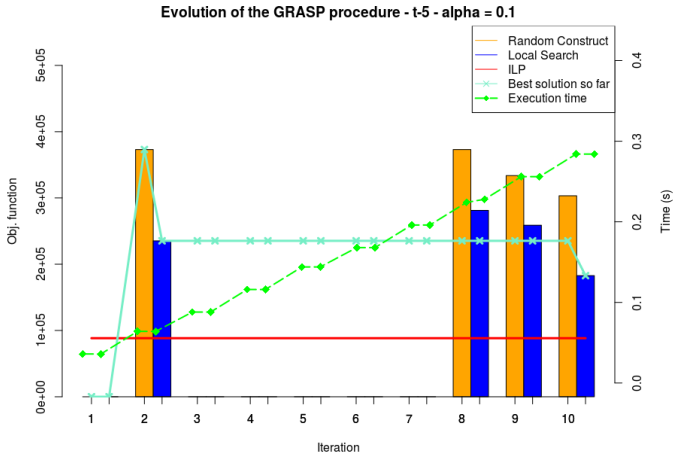
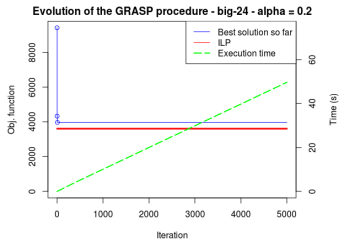


Figure 3: Illustration of the trace for the input *t-5*, $\alpha = 0.1$.

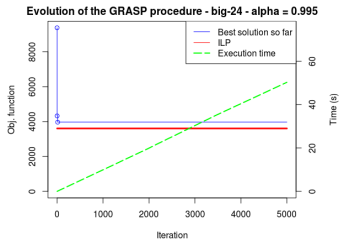
In this figure we find all relevant information we need to know to explain the main features, in the short term, of this procedure. In orange we find the value of the solutions obtained by the random constructor and in blue the value of the solution improved by the local search procedure for 10 iterations. In light blue we find the value of the best solution found until the specified iteration. The red line represents again the value of the optimal solution found by the ILP model, and in green the execution time.

First and foremost, as anyone could notice, there are missing values for some of the iterations. That means that the random constructor produced an infeasible solution. However, the time does not stop and continues growing. But it does so linearly, a highly desirable property for this procedure. This simple graph illustrates the complete behaviour of this procedure. While there are some iterations that do not produce a better solution, or no solution at all, the method always keeps the best solution, the value of which is represented in the light-blue line.

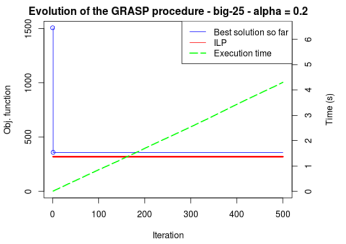
Now, this method, although designed to provide many different solutions by randomly constructing them and improving them later on, does not work the same way for all instances, and for some of them it is clearly useless. Take, for instance, inputs *big-24* and *big-25*. The evolution of this procedure for different values of α shows that, even after waiting for three times as much time as it takes the ILP solver to find the optimum, reaching the optimum is nearly impossible. The figures showing this evolution are the following:



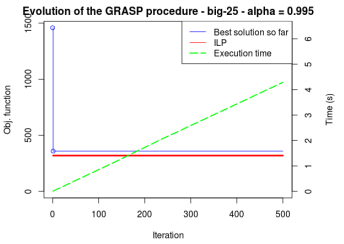
(a) Illustration of the trace for the input *big-24*, $\alpha = 0.2$. Best objective = 3967.81.



(b) Illustration of the trace for the input *big-24*, $\alpha = 0.995$. Best objective = 3967.81.



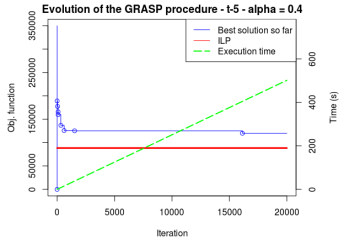
(c) Illustration of the trace for the input *big-25*, $\alpha = 0.2$. Best objective = 359.298.



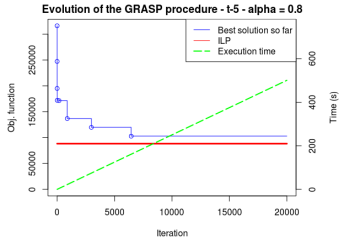
(d) Illustration of the trace for the input *big-25*, $\alpha = 0.995$. Best objective = 359.298.

Figure 4: Traces of the GRASP procedure for two inputs and different values of α.

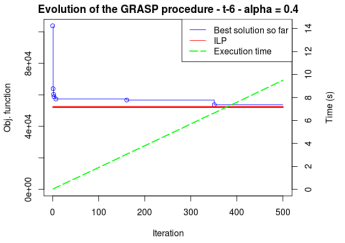
Obviously, this procedure's quality highly depends on the quality of the random constructor and the neighbourhood exploration. These two do not seem to work well with the previous instances due to their low difficulty and size. Although the optimum is never reached in these cases, and the evolution is quite poor, the time increases linearly, something that was already expected when inspecting the behaviour in figure 3. And indeed, the difficulty affects the behaviour of the procedure. See the following figures where we try to solve the instances *t-5* and *t-6*.



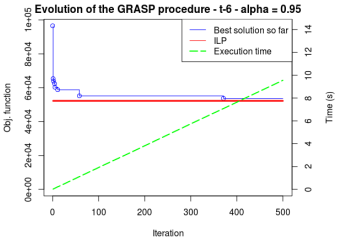
(a) Illustration of the trace for the input *t-5*, $\alpha = 0.4$. Best objective = 119848.



(b) Illustration of the trace for the input *t-5*, $\alpha = 0.8$. Best objective = 102851.



(c) Illustration of the trace for the input *t-6*, $\alpha = 0.4$. Best objective = 53727.



(d) Illustration of the trace for the input *t-6*, $\alpha = 0.95$. Best objective = 53727.

Figure 5: Evolution of the GRASP procedure for inputs *t-5* and *t-6* and different values of α.

With these figures we see that the procedure improves the solution slightly over time. This means that there may be hope to improve the solution even more if we waited enough time. However, in figures 5a and 5b the procedure already took as much time as the ILP solver (actually, slightly more), and in figures 5c and 5d the procedure took twice as much time as the ILP solver. Besides, there is no improvement at all in the solutions for thousands of iterations in the first pair, and hundreds of iterations in the second. Thus, the procedures proposed in section 3.4 are not good enough to solve these instances.

Notice another two aspects, though, that are that the parameter α neither seems to have much effect on the execution time or on the quality of the best solution found at the end of the execution, for the same number of iterations.

4.4 BRKGA

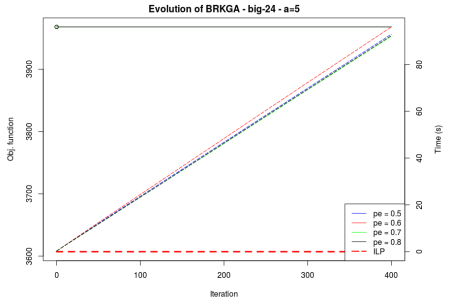
This is the last procedure that will be commented in this report. In section 3.5 is explained how the BRKGA procedure depends on a chromosome decoder. In this section is evaluated the impact of this decoder and the BRKGA's parameters on the quality of the solutions found. To do that, we will follow the recommendations given in the article [2] for the choice of the parameters.

We applied three different values for the total size of the population defined as a function of the parameter $a \in \mathbb{N}$: $p = a \cdot n$, where n is the size of the chromosome, which, according to the description of the decoder, is the amount of cities for each instance. See table 1 in section 4.1.1 to see the description of each instance. With this parameter set, the others were defined as $p_e = 0.175p$, $p_m = 0.2p$, and then four different values for the probability of inheritance were tried to have more diversity in results of experimentation $\rho_e = 0.5, 0.6, 0.7, 0.8$.

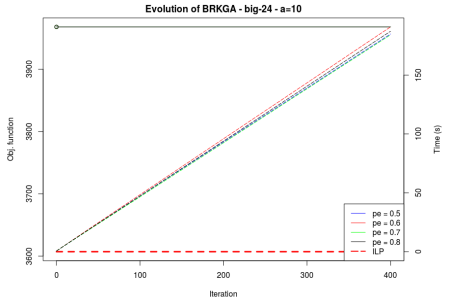
The figures presented have a common format: in four different colours are presented the values of the best solution at every generation of the population, one for each value of ρ_e. The value of the parameter a is specified at the top of each figure. The values of p, p_e and p_m are are also specified,

but in the caption, and in a dotted red line is denoted the value of the optimal solution found with the ILP solver. The other coloured and dotted lines correspond to the execution time in seconds of the procedure. Their corresponding axis is found to the right of the figures.

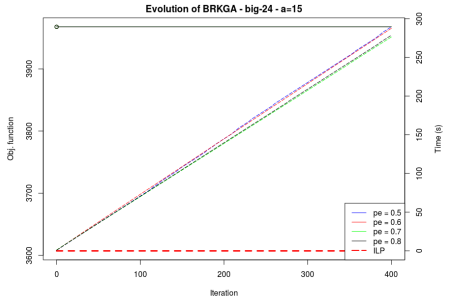
The first results recorded are very disappointing. These are the results of this procedure when applied to the instances *big-24* and *big-25*. We can see in figure 6 how the values of the best solution found throughout the hundreds of generations do not improve a bit (with just one exception) for any value of ρ_e and a and do not seem to be willing to get close to the optimal value (in red). This happened in a time that is a few times larger than the ILP needed to solve the instance, therefore we should not expect the results to improve much more with the chosen configuration of parameters.



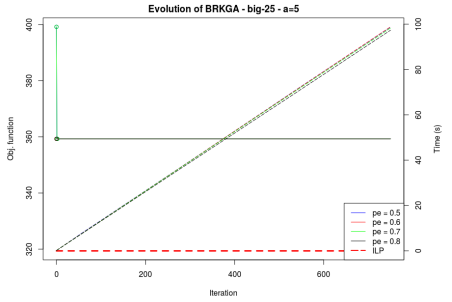
(a) Illustration of the trace for the input *big-24*. $a = 5$, $p = 115$, $p_e = 20$, $p_m = 23$.



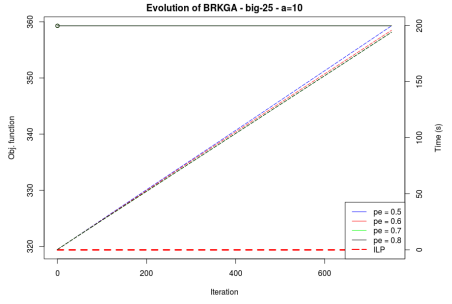
(b) Illustration of the trace for the input *big-24*. $a = 10$, $p = 230$, $p_e = 40$, $p_m = 46$.



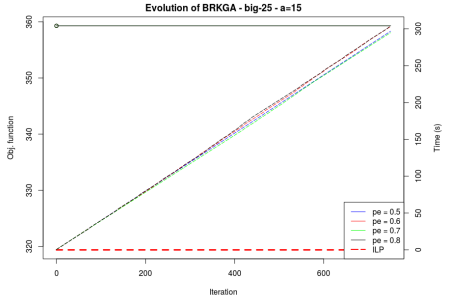
(c) Illustration of the trace for the input *big-24*. $a = 15$, $p = 345$, $p_e = 60$, $p_m = 69$.



(d) Illustration of the trace for the input *big-25*. $a = 5$, $p = 95$, $p_e = 16$, $p_m = 19$.



(e) Illustration of the trace for the input *big-25*. $a = 10$, $p = 190$, $p_e = 33$, $p_m = 38$.

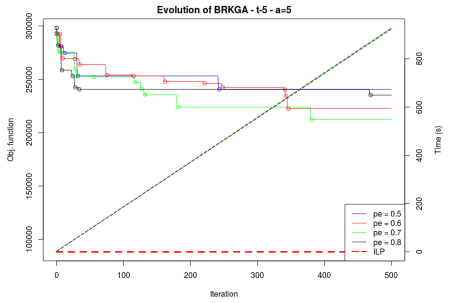


(f) Illustration of the trace for the input *big-25*. $a = 15$, $p = 285$, $p_e = 49$, $p_m = 57$.

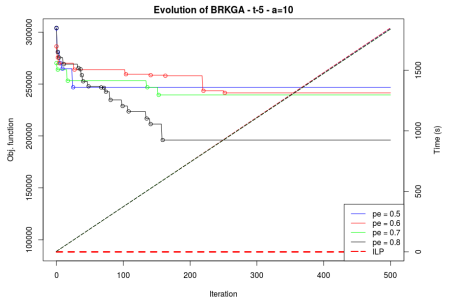
Figure 6: Evolution of the BRKGA procedure for the inputs *big-24* and *big-25* and values of $a = 5, 10, 15$.

However, the results were more promising with the inputs *t-5* and *t-6* where the difficulty of the

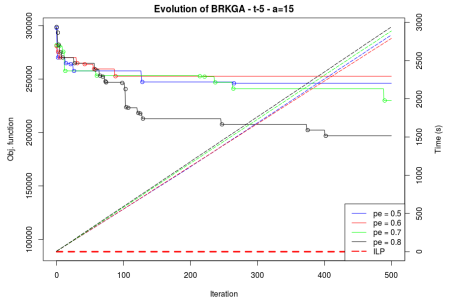
instances is greater. We can see in figure 7 the impact in the behaviour on the procedure the parameters p, p_e, p_m and ρ_e have. In contrast with the previous instances, we can see how by increasing the value of ρ_e the value of the best solution improves bit by bit over time. Unfortunately, the execution time needed to generate each new generation in these instances is really large: it takes about one hour for every value of ρ_e, when $a = 15$ and for instance *t-5*, to generate 500 generations (here we see the impact of a on the execution time). This means that the optimal value is not likely to be reached until a really big amount of time has passed, therefore the code needs to be greatly optimised (by making it parallel, for example). The value of a has also a positive impact on the quality of the solution found: the larger the population is, the more variety it allows for mutant individuals and space for elite individuals which leads to better chances of obtaining fitter individuals at every generation, and this can be seen in figures 7b and 7c - just to mention a few - where the value of the best optimal solution for $\rho_e = 0.8$ improves at a faster rate. However, in spite of almost constantly improving the solution, the best found by this procedure is really far from the optimum.



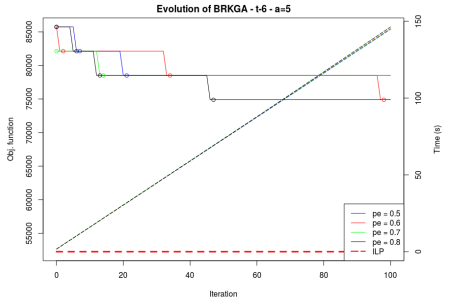
(a) Illustration of the trace for the input *t-5*. $a = 5$, $p = 250$, $p_e = 43$, $p_m = 50$.



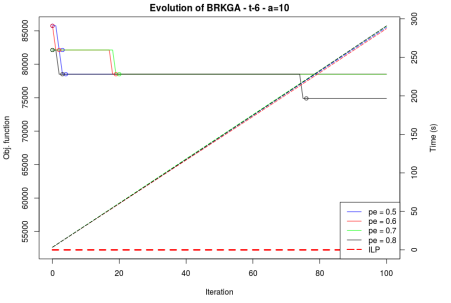
(b) Illustration of the trace for the input *t-5*. $a = 10$, $p = 500$, $p_e = 87$, $p_m = 100$.



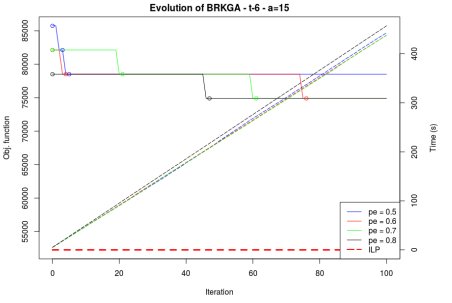
(c) Illustration of the trace for the input *t-5*. $a = 15$, $p = 750$, $p_e = 131$, $p_m = 250$.



(d) Illustration of the trace for the input *t-6*. $a = 5$, $p = 250$, $p_e = 43$, $p_m = 50$.



(e) Illustration of the trace for the input *t-6*. $a = 10$, $p = 500$, $p_e = 87$, $p_m = 100$.



(f) Illustration of the trace for the input *t-6*. $a = 15$, $p = 750$, $p_e = 131$, $p_m = 250$.

Figure 7: Evolution of the BRKGA procedure for the inputs *t-5* and *t-6* and values of $a = 5, 10, 15$.

5 Conclusions

In this project were tested many different techniques for combinatorial optimisation problems: from the computation of the optimal solution using an Integer Linear Programming solver to its approximation using several different heuristics. These methods are clearly different when it comes to their definition and implementation: the former has been easy to formulate and to test whereas the latter had more difficulty in their design, implementation and testing since we were not formally provided with an extendable framework for that purpose. That is why the framework described in section 3.1.3 was implemented. Admittedly, the python code could have been easily modified. However, python is not a sensible choice for a project like this and the execution times shown in figure 7 is a tangible proof.

Now, these two kind of techniques for combinatorial optimisation problems are also different in the amount of time they need to find an optimal solution - or a nearly optimal solution for the heuristics case. The ILP solver should take a lot of time to find the optimal solution in a general case whereas the heuristics, if well designed and implemented, should take a small amount of time (when compared to the ILP solver's) to find, not an optimal solution, but a nearly optimal. That is, if we can not find the optimal because the instance is too large, heuristic methods can help us find a not-as-good solution but, at least, find a solution good enough for our purposes.

However, in this project we haven't been able to do so. Despite our efforts to design good heuristics that could fit in three different procedures (Local Search, see section 3.3, GRASP, see section 3.4 and BRKGA, see section 3.5) we could not manage to obtain any instance that could fit in this description. Instead, all instances could be solved by the ILP solver in a relatively small amount of time and the heuristics produced results far from being the optimal, and in a really huge amount of time as a general case.

Nevertheless, what we managed to do is to show the quality of each procedure. The one that performed the worst for our case is the Local Search procedure. Making this procedure worthwhile relies too much on the initial solution being poor enough so as to be properly improved by making little changes, that is, the neighbourhood exploration. Therefore, Local Search applied to our greedy constructor has shown not to be of good use due to the small exploration of the neighbourhood, shown in table 2 and in figure 2. However, it was useful in the case of the GRASP procedure where randomly constructed solutions could be greatly improved using this technique. Solutions did reach values near the optimum. However, we observed that the behaviour of this procedure was rather erratic: "easy instances" were solved nearly to the optimal value very soon in the procedure and then they were stuck in that best solution for the rest of the execution. This makes us think that, no matter how long we leave that procedure running, we will never reach the optimal solution. Instances that were more "difficult" made this procedure useful due to the continuous improvements that solutions suffered through time. This means that it is quite likely to obtain the optimal solution if one is patient enough for bigger instances.

The BRKGA procedure seems to be the best in behaviour, but not in results. We find a lot more improvements in this kind of metaheuristic than in the others, but it has higher execution times and worse results. On the one hand, as mentioned several times through this report, several improvements found during the execution of the first, say, 30 minutes makes us think that there could be more if we waited more time, hence obtaining a solution closer and closer to the optimal. On the other hand, the results obtained are far worse than those obtained with the GRASP procedure: the difference between the best solution by the BRKGA and the ILP solver is greater than between the best found by the GRASP procedure and the ILP solver.

Arguably, just by looking at the results presented, the GRASP procedure seems to be the first choice to use when trying to solve any instance of this problem (since we tried "easy" and "difficult" instances). However, further improvements need to be made: this procedure is fairly parallel (its efficiency can

be easily improved) and it only needs a better neighbourhood exploration, since a randomised greedy construction can easily lead to extremely bad solutions. However, the results are still more promising when looking at the BRKGA procedure due to its theoretical background: a really big amount of diversity in the solution space and continuous improvements through time. Therefore, as a conclusion, despite the results being worse, the BRKGA is the best procedure that could be used to solve instances of this problem, after having improved its efficiency enough.

References

[1] Thomas A. Feo and Mauricio G. C. Resende. "Greedy Randomized Adaptive Search Procedures". In: *Journal of Global Optimization* 6.2 (Mar. 1995), pp. 109–133. ISSN: 1573-2916. DOI: 10.1007/BF01096763. URL: https://doi.org/10.1007/BF01096763.

[2] José Fernando Gonçalves and Mauricio G. C. Resende. "Biased random-key genetic algorithms for combinatorial optimization". In: *Journal of Heuristics* 17.5 (2011), pp. 487–525. ISSN: 1381-1231. DOI: 10.1007/s10732-010-9143-1. URL: http://link.springer.com/10.1007/s10732-010-9143-1.