Algorithmic Methods For Matematical Models

Final Project

Sergio Rodríguez Guasch

Contents

1 Problem statement

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

The company has a set of locations L where logistic centers could be installed, and a set of cities C that need to be served. For each location $l \in L$ we know its coordinates (l_x, l_y), and for every city $c \in C$ we know its coordinates (c_x, c_y) and its population p_c. We have available a set T of logistic center types. Each type $t \in T$ represents a logistic center with capacity cap_t, working distance d_city_t, and installation cost $cost_t$.

Each city must be served by exactly one primary and one secondary center, which must of course be different. Logistic centers should be placed so that the distance between any two of them is at least d_center. The capacity of a center type t enforces that the sum of the populations of the cities it serves as a primary center plus 10% the sum of the populations of the cities it serves as a secondary center cannot exceed cap_t. With respect to its working distance, a center of type t cannot be neither the primary center of any city at distance more than d_city_t, nor the secondary center of any city at distance more than $3 \times d\_city_t$.

The goal of this project is to decide where to install logistic centers, determine of which type each center will be and to which primary and secondary center each city should be connected to in order to minimize the total installation cost.

2 Linear programming formulation

2.1 Variables

The decision variables (with their meanings) are the following:

$X_{l,t}$ = Location $l \in L$ has a center of type $t \in T$
$P_{c,l}$ = City $c \in C$ has the center at location $l \in L$ as a primary center
$S_{c,l}$ = City $c \in C$ has the center at location $l \in L$ as a secondary center

All variables are going to be boolean. This means that our problem is formulated as an Integer Linear Programming (ILP) problem.

Apart from the decision variables stated above, the following constants are going to be used:

pop_c = Population of the city $c \in C$
pop_all = Sum of populations of all cities $c \in C$

$dcl_{c,l}$ = Distance between the city $c \in C$ and the location $l \in L$
d_center = The minimum required distance between two centers
dll_{l_a,l_b} = One if the distance between locations l_a and l_b for all $l_a, l_b \in L$ is at least d_center, zero otherwise.
$cost_t$ = The cost to install a center of type $t \in T$
cap_t = The capacity of a center of type $t \in T$
d_city_t = The working distance of a logistic center of type $t \in T$

2.2 Objective function

$$\text{minimize} \sum_{l \in L} \sum_{t \in T} cost_t \times X_{l,t} \tag{1}$$

The objective function is literally taken from the project statement:

The goal of this project is to decide where to install logistic centers, determine of which type each center will be and to which primary and secondary center each city should be connected to in order to ***minimize the total installation cost***

2.3 Constraints

The statement also introduces some constraints that define how a solution should look and what satisfy. These constraints are going to be extracted from the statement and formalized as linear constraints.

The very first constraint to impose is that in each location at most one logistic center should go in. Given the definition of $X_{l,t}$, this is straightforward to implement:

$$\sum_{t \in T} X_{l,t} \leq 1 \quad \forall l \in L \tag{2}$$

Each city must be served by exactly one primary and one secondary center, which must of course be different

This text introduces two different constraints:

1. *Each city must be served by exactly one primary and one secondary center...*
2. *...which must of course be different*

We can encode the first constraint as follows:

$$\sum_{l \in L} P_{c,l} = 1 \quad \forall c \in C \tag{3}$$

$$\sum_{l \in L} S_{c,l} = 1 \quad \forall c \in C \tag{4}$$

The second constraint can be encoded as something that is very similar to a logical implication:

$$P_{c,l} + S_{c,l} \leq 1 \quad \forall c \in C, l \in L \tag{5}$$

Equation 5 can be derived as follows: from the second constraint we can deduce that $P_{c,l} \implies \neg S_{c,l}$ and that $S_{c,l} \implies \neg P_{c,l}$. Both implications can be written as $\neg P_{c,l} \vee \neg S_{c,l}$, which is equivalent to say that $P_{c,l}$ and $C_{c,l}$ cannot be true at the same time (so, at most one of them can be true).

It must be noted that if some city is using the center placed at location l as a primary or secondary center then the location l must have at least one center. These constraints can be stated in a logical way as follows:

$$P_{c,l} \implies \sum_{t \in T} X_{l,t} \geq 1 \quad \forall c \in C, l \in L$$

$$S_{c,l} \implies \sum_{t \in T} X_{l,t} \geq 1 \quad \forall c \in C, l \in L$$

These logical implications can be encoded as follows:

$$\sum_{t \in T} X_{l,t} \geq P_{c,l} + S_{c,l} \quad \forall c \in C, l \in L \tag{6}$$

Equation 5 guarantees that at most one of $P_{c,l}$ and $S_{c,l}$ will be true, letting us to write these two logical implications in only one constraint instead of two. Consider equation 6. If $P_{c,l}$=$S_{c,l}$=0 then this equation does not impose any constraint because $X_{l,t}$ are boolean variables. However, if ($S_{c,l}$) $P_{c,l}$=1 then this equation is imposing that $\sum_{t \in T} X_{l,t} \geq 1$ (in fact, to be exactly one if we take into account equation 2).

Logistic centers should be placed so that the distance between any two of them is at least d_center

Since centers are going to be built at fixed locations $L = \{l_1, ..., l_{|L|}\}$ this can also be stated as *if two locations $l_a, l_b \in L$ are separated by a distance which is strictly less than d_center, then we can build a logistic center in at most one of them.*

Written as a logical implication we get the following statement:

$$\text{dist}(l_a, l_b) < d\_center \implies \sum_{t \in T} X_{l_a,t} + \sum_{t \in T} X_{l_b,t} \leq 1 \quad \forall l_a, l_b \in L$$

Since the centers locations and *d_center* are constants we can precompute a $|L| \times |L|$ matrix *dll* where dll_{l_a,l_b} is one if $\text{dist}(l_a, l_b) \geq d\_center$, and zero otherwise. Using this matrix, we can write the following constraint:

$$\sum_{t \in T} X_{l_a,t} + X_{l_b,t} \leq 1 \quad \forall l_a, l_b \in L : \quad \text{index}(l_a) < \text{index}(l_b) \wedge dll_{l_a,l_b} = 0 \tag{7}$$

Note that if $l_a = l_b$ then this constraint should not be applied because it would prevent the problem to have solutions. Another observation is that euclidean distance is symmetric in the sense that $d(x, y) = d(y, x)$, so one can save $\binom{|L|}{2}$ constraints if only pairs (l_a, l_b) such that $\text{index}(l_a) < \text{index}(l_b)$ are considered.

The capacity of a center type t enforces that the sum of the populations of the cities it serves as a primary center plus 10% the sum of the populations of the cities it serves as a secondary center cannot exceed cap_t

In order to encode this constraint we are going to use an approach that is similar to the Big M method:

$$(\sum_{c \in C} P_{c,l} \times pop_c) + (0.1 \sum_{c \in C} S_{c,l} \times pop_c) \leq cap_t + (1 - X_{l,t}) \times pop\_all \quad \forall l \in L, t \in T \tag{8}$$

Let's suppose that $X_{l,t}$=0. Given that equation 5 imposes that no pair $(P_{c,l}, S_{c,l})$ can be true at the same time we can affirm that the maximum value that the left part of equation 8 can reach is *pop_all*, so the inequality will always hold. However, if $X_{l,t}$=1 then equation 8 imposes an upper bound (probably) lower than the total population, which is precisely the capacity of a center of type t.

With respect to its working distance, a center of type t cannot be neither the primary center of any city at distance more than d_city_t*, nor the secondary center of any city at distance more than* $3 \times d\_city_t$

This text is equivalent to saying that if a location $l \in L$ has a center of type $t \in T$ then no city $c \in C$ can use this center as a primary center if $\text{dist}(c, l) > d\_city_t$. In a similar way, a city c cannot use this center as a secondary center if $\text{dist}(c, l) > 3 \times d\_city_t$. Following the same technique to implement implications as in equation 5, we get the two following constraints:

$$X_{l,t} + P_{c,l} \leq 1 \quad \forall c \in C, l \in L, t \in T : \quad dcl_{c,l} > d\_city_t \tag{9}$$

$$X_{l,t} + S_{c,l} \leq 1 \quad \forall c \in C, l \in L, t \in T : \quad dcl_{c,l} > 3 \times d\_city_t \tag{10}$$

3 OPL model

This section implements the model described before. Some of the constraints are written in a more compact form than the equations shown at the previous section.

3.1 Source code

```
int numCities=...;
range C=1..numCities;

int numTypes=...;
range T=1..numTypes;

int numLocations=...;
range L=1..numLocations;

float cityCoordinates[c in C][_ in 1..2] = ...;
float locationCoordinates[l in L][_ in 1..2] = ...;

int cityPopulation[c in C]=...;
int totalPopulation = sum(c in C) cityPopulation[c];

float d_center=...;

int typeCapacity[t in T] = ...;
float typeDistance[t in T] = ...;
float typeCost[t in T] = ...;

float qll[a in L][b in L];
int dll[a in L][b in L];
float dcl[c in C][l in L];


execute {
  function dis(p, q) {
    return Math.sqrt((p[1]-q[1])*(p[1]-q[1]) + (p[2]-q[2])*(p[2]-q
    [2]));
  }

  for(var i=1; i<=numLocations; ++i) {
    for(var j=1; j<=numLocations; ++j) {
      qll[i][j] = dis(locationCoordinates[i], locationCoordinates[j
    ]);
      if(qll[i][j] >= d_center) {
        dll[i][j] = 1;
      }
      else {
        dll[i][j] = 0;
      }
    }
  }
  for(var i=1; i<=numCities; ++i) {
    for(var j=1; j<=numLocations; ++j) {
      dcl[i][j] = dis(cityCoordinates[i], locationCoordinates[j]);
    }
  }
};

dvar boolean X[l in L][t in T];
dvar boolean P[c in C][l in L];
dvar boolean S[c in C][l in L];

```

```
// EQUATION 1
minimize sum(l in L, t in T) typeCost[t]*X[l,t];

subject to {
  // EQUATION 2
  forall(l in L)
    sum(t in T) X[l, t] <= 1;
  // EQUATION 3
  forall(c in C)
    sum(l in L) P[c, l] == 1;
  // EQUATION 4
  forall(c in C)
    sum(l in L) S[c, l] == 1;
  // EQUATION 5
  forall(c in C, l in L)
    P[c, l] + S[c, l] <= 1;
  // EQUATION 6
  forall(c in C, l in L)
    sum(t in T) X[l, t] >= P[c, l] + S[c, l];
  // EQUATION 7
  forall(la in L, lb in L : la < lb && dll[la, lb] == 0)
    sum(t in T) (X[la, t] + X[lb, t]) <= 1;
  // EQUATION 8
  forall(l in L, t in T)
    sum(c in C) cityPopulation[c]*(P[c, l] + 0.1*S[c, l])
    <= typeCapacity[t] + (1 - X[l, t])*(totalPopulation);
  // EQUATION 9
  forall(l in L, t in T, c in C : dcl[c][l] > typeDistance[t])
    X[l, t] + P[c, l] <= 1;
  // EQUATION 10
  forall(l in L, t in T, c in C : dcl[c][l] > 3.0*typeDistance[t])
    X[l, t] + S[c, l] <= 1;
}
```

4 Meta-heuristics

4.1 Previous comments and observations

Both meta-heuristics have been implemented taking into account the following fact: if we have chosen a primary and a secondary location for all cities, then we can compute the optimal center placement (for this configuration) quickly. For a partially constructed solution (that is, a solution that has some city with no locations assigned) we can compute the *potential* cost by only taking into account the assigned cities. Also, we can check if a partially constructed solution is necessarily unfeasible or not. These observations led us to consider only the centers assigned to cities as decision variables in our meta-heuristics. In both meta-heuristics we will use the procedure exposed in algorithm 1 to compute the optimal center placement in a partially constructed solution. Also, whenever two solutions are compared (for example, when sorted) it must be assumed that this comparison is done with algorithm 2. The intuition of the tie breaker rule is to give more chances to solutions that are *more likely to change*.

Algorithm 1 Compute optimal center type for a partially constructed solution

```
1:  procedure ComputeOptimalCenters(S : partially constructed solution)
2:      for i in 1..#Locations do
3:          if locations(i).assignedPopulation ≠ 0 then
4:              t ← the cheapest suitable type for the ith location (if any)
5:              if t is null then
6:                  make S unfeasible
7:                  return S
8:              S.locationType(i) ← t
9:      if S is not valid then
10:         make S not feasible
11:         return S
12:     make S feasible
13:     return S
```

As a remark, a location type is *suitable* if it satisfies the population and distance constraints imposed by the cities center configuration.

4.2 GRASP

In our GRASP implementation we build feasible solutions as follows: for every city c_i compute all the possible pairs of locations (p, s) candidates such that the assignation of p and s as the primary and secondary center of c respectively does not lead to an unfeasible solution, sort these candidates by their potential cost and choose one among the α first ones. Our local search implementation explores the neighbors of a solution such that their (partial) cost is strictly less than the original solution cost. A solution y is considered a neighbor of a

Algorithm 2 Compare two solutions

```
1: procedure CompareSolutions(A, B : (Possibly partially constructed) solutions)
2:     if A is not valid then
3:         return B
4:     if B is not valid then
5:         return A
6:     ca ← cost of A
7:     cb ← cost of B
8:     if A ≠ B then
9:         if ca < cb then
10:            return A
11:        else
12:            return B
13:    ra ← 0
14:    rb ← 0
15:    for 1 in 1..#numLocations do
16:        ra ← ra + A.locationPopulation(i) / A.locationCapacity(i)
17:        rb ← rb + B.locationPopulation(i) / B.locationCapacity(i)
18:    if ra > rb then
19:        return A
20:    else
21:        return B
```

solution a if x can be transformed to y by performing exactly one of the following changes to x:

1. Swap the primary centers of two different cities
2. Swap the secondary centers of two different cities
3. Swap the primary city of a city with the secondary city of a different city
4. Reassign the primary and the secondary center (at least one of them) of a city

As we have mentioned before, the center types are not considered decision variables as they are computed depending on the locations assigned to the cities, so they are not taken into account when considering if a pair of solutions x and y are neighbors or not. Pseudo-code for the GRASP, construction and local search phases can be found in algorithms 3, 4, and 5 respectively.

Algorithm 3 GRASP

```
1: procedure GRASP
2:     bestSolution ← an empty, unfeasible solution
3:     for i in 1..maxGraspIterations do
4:         sol ← generateRandomSolution()
5:         sol ← localSearch(sol)
6:         if sol is better than bestSolution then bestSolution ← bestSolution
7:     return bestSolution
```

Algorithm 4 Random solution generator

```
1: procedure generateRandomSolution
2:     sol ← an empty, unfeasible solution
3:     perm ← a random permutation of 1, 2, ...numCities
4:     for i in 1..numCities do
5:         candidateList ← an empty list
6:         for (p, s) in (1..numLocations × 1..numLocations) do
7:             candidate ← a copy of sol
8:             candidate.cityPrimaryCenter(perm(i)) ← p
9:             candidate.citySecondaryCenter(perm(i)) ← s
10:            if candidate is valid then
11:                candidateList.add(candidate)
12:        n ← length of candidateList
13:        if n = 0 then
14:            make sol unfeasible
15:            return sol
16:        sort candidateList according to the costs of the solutions
17:        sol ← a random member of the αn first members of candidateList
18:    ComputeOptimalCenters(sol)
19:    return sol
```

Algorithm 5 Local search

```
1: procedure localSearch(sol : solution)
2:     bestSolution ← sol
3:     S ← an empty set
4:     S.insert(sol)
5:     while S is not empty do
6:         cur ← the cheapest solution in S
7:         if cur is better than bestSolution then
8:             bestSolution ← cur
9:         S.delete(cur)
10:        neighbors ← all the neighbors of cur that improve cur
11:        for i in 1..#neighbors do
12:            S.insert(neighbors(i))
13:            while S has more than beamSize members do
14:                S.deleteWorstSolution()
15:    return bestSolution
```

4.3 BRKGA

Our chromosome is represented as a vector of numCities reals in the interval $[0, 1)$. The decoding step is described in algorithm 6. As we can see, the decoder behaves almost identically to the GRASP random solution procedure when $\alpha = 0$. We must note that, even the resemblance with the GRASP construction phase, this procedure has no random component, so the decoding step is totally deterministic.

Algorithm 6 BRKGA Chromosome Decoder

```
1:  procedure DECODECHROMOSOME(chr : vector of numCities reals in [0, 1))
2:      perm ← an empty list
3:      for i in 1..numCities do
4:          discr ← chr(i) × cityPopulation(i)
5:          add (discr, i) to perm
6:      sort perm by first pair component (second as tie breaker)
7:      sol ← an empty, unfeasible solution
8:      for i in 1..numCities do
9:          bestCost ← ∞
10:         bestPair ← (-1, -1)
11:         for (p,s) in numLocations × numLocations do
12:             cand ← a copy of sol
13:             cand.cityPrimaryCenter(i) ← p
14:             cand.citySecondaryCenter(i) ← s
15:             cand ← ComputeOptimalCenters(cand)
16:             if cand is valid and cand.solutionCost < bestCost then
17:                 bestCost ← cand.solutionCost
18:                 bestPair ← (p, s)
19:         if bestCost = ∞ then
20:             make sol unfeasible
21:             return sol
22:     return sol
```

5 Analysis and Comparative Results

This sections analyzes and compares the performance of the three optimizations methods developed in this project. The analysis is focused on how the solution quality improves during time on a fixed instance. The instances are generated almost randomly. This is done in purpose for two reasons: it is easier to build random datasets and our previous experience in combinatorial optimization (for example, SAT) shows us that random instances are usually harder than realistic datasets (that is, instances that model some real scenario).

As we can see in figures 1 and 2, GRASP seems to be the worst meta-heuristic,

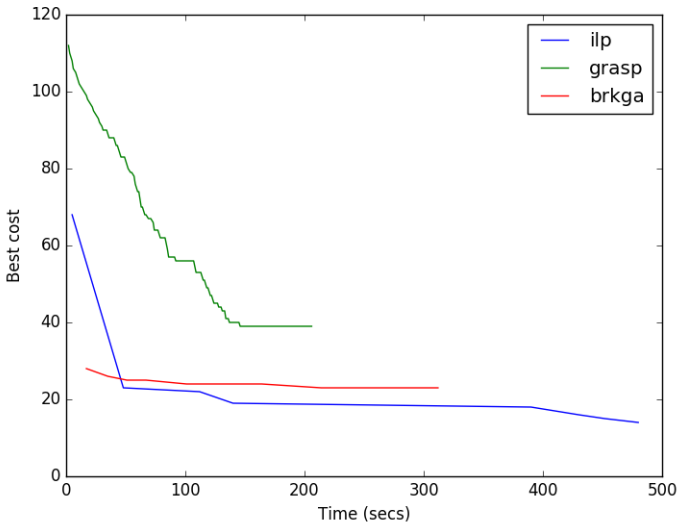

Figure 1: Evolution of solution quality over time for a random instance consisting of 80 cities and 160 locations. This dataset is named random_4.dat and can be found in the data folder

both in solution quality and amount of required time to stabilize. However, BRKGA manages to find good solutions quicker than the ILP model. Also, if we observe the tendency, it seems that the amount of time that we have to wait until the ILP gives a reasonably good solution increases a lot, while BRKGA seems to be more stable. Also, the difference of the best found solution by BRKGA and the actual optimal solution is not very big. These observations may lead us to consider only BRKGA on big instances.

A possible explanation of the poor GRASP performance is the big similarity between the greedy function and the actual cost function (even if the tie-breaker is not present on the actual cost function). A very strong indicator of this is that the BRKGA chromosome decoder procedure is identical to the GRASP construct phase when $\alpha = 0$, and BRKGA always gives better solutions that GRASP. In fact, sometimes a full GRASP run is not able to even reach the best initial solution of BRKGA (that is, the best initial random mutant prior to any cross-over and selection).
We also analyzed how *structured* instances influence on the solvers performance and solution quality. For this case, we have created instances of the following kind:

1. There are N points.

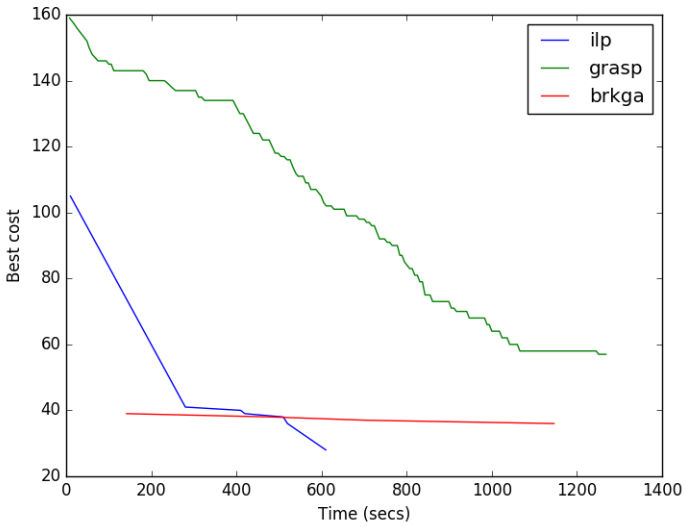

Figure 2: Evolution of solution quality over time for a random instance consisting of 160 cities and 200 locations. This dataset is named random_4.dat and can be found in the data folder

2. Locations are the points that belong to the convex hull, cities are the other points.

3. Distance variables as *d_center* and working distances are determined according to distances between locations and cities.

The intention of this dataset is to model a somehow realistic scenario where cities are concentrated and industrial places surround them. A graphical example of this kind of dataset can be found in figure 3. For the dataset represented in figure 3 the ILP model needs 0.54 seconds to reach the optimal solution, while meta-heuristics sometimes do not even find an initial, feasible, solution.

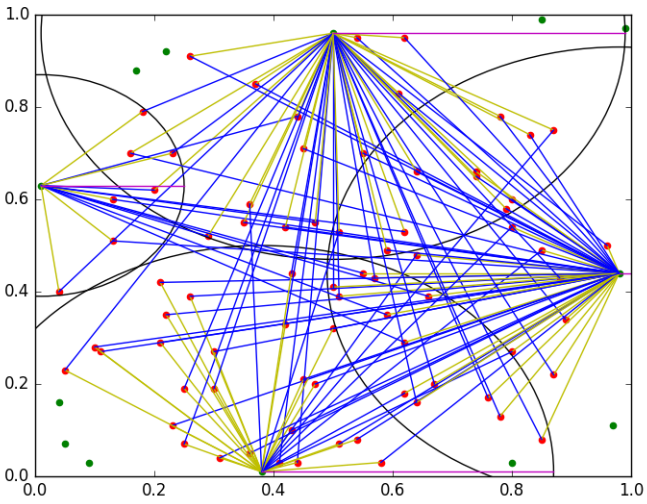

Figure 3: A graphical example of a convex hull dataset and its solution. Green points represent locations, red points cities, yellow lines a primary center relation, and blue lines a secondary center relation. Purple lines are to help to identify the primary working distance of each location. This dataset can be found in data/hull_1.dat.

6 Conclusions

As expected, we have observed that meta-heuristics are useful to find *good enough* solutions quickly, and that they become more useful when the problem size increases due to the performance of the Integer Linear Programming model. Also, we have learned a very valuable lesson: ad-hoc optimization methods must try to take into account the structure of the data. Even if this project did not define how a *typical* dataset should look, it has been shown that it is not hard to build *killer* datasets for our meta-heuristics that are likely to appear in real life. More precisely, it must be taken into account that instances with little valid solutions are hard and challenging for a randomized meta-heuristic.