

INFO-F424 - Combinatorial Optimization

Project - The p -Center Problem

Erica BERGHMAN
Charles HAMESSE

École Polytechnique de Bruxelles

May 2017

Abstract

The purpose of this project is to implement two formulations of the same combinatorial optimization problem in Julia, using the JuMP package. We will start by describing the mathematical aspects of both formulations, then we will explain our implementations and discuss their performance.

Contents

1	Introduction	2
1.1	Implementation	2
1.2	Compiling and running	2
2	Formulations	2
2.1	Daskin (1995)	2
2.2	Calik and Tansel (2013)	3
3	Implementation	4
4	Optimization	4
4.1	Hot starts	4
4.1.1	Random initialization	4
4.1.2	2-approximation algorithm	5
4.1.3	Gurobi Optimizer	5
4.1.4	Comparison of the heuristics	5
4.2	Number of variables	5
4.2.1	Dual problem	5
4.3	Valid inequalities	6
4.3.1	Chvátal-Gomory procedure	6
5	Results	6
6	Conclusion	8
6.1	Comparison of the two formulations	8
6.2	Wrap-up	8

1 Introduction

1.1 Implementation

1.2 Compiling and running

As Julia is a scripting language, nothing needs to be compiled. However, one might have to install the following packages to ensure the correction functioning of our program:

- **ArgParse**: used to parse command-line arguments
- **Cbc**: a simple LP solver
- **Gurobi**: a more complex, commercial LP solver

To launch the program or use it in a bash script, one only needs to `cd` to the root directory of our archive and enter:

```
julia main.jl
```

We implemented a simple yet complete human-machine interface which is well described using `-h` or `--help`:

```
>>> julia main.jl -h
usage: main.jl -i INPUT -f FORMULATION -s SOLVER [-o OUTPUT]
               [-d DIVISOR] [-c INITIAL-CANDIDATE] [-v VERBOSE] [-h]

optional arguments:
  -i, --input, --instance INPUT
                        Path to the instance file.
  -f, --formulation FORMULATION
                        P1 or P3.
  -s, --solver SOLVER   Cbc or Gurobi.
  -o, --output OUTPUT   Output file path. (default: "stdout")
  -d, --divisor DIVISOR
                        Divisor for valid inequality in P1. If set to
                        -1, the new valid inequality is not
                        considered. (default: "-1")
  -c, --initial-candidate INITIAL-CANDIDATE
                        Heuristic for hot start. Use default, random
                        or 2approx. (default: "default")
  -v, --verbose VERBOSE
                        Enable verbose mode. 0 or 1. (default: "0")
  -h, --help
                        show this help message and exit
```

2 Formulations

2.1 Daskin (1995)

This is the formulation referred to as (P1) in the original paper.

According to the usual canvas, the mathematical formulation is given as follows.

Parameters

- p = maximum number of centers;
- N = number of vertices of the instance.

Variables Two variables are used in this formulation:

$$y_j = \begin{cases} 1 & \text{if vertex } j \text{ is a center} \\ 0 & \text{otherwise} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{if vertex } i \text{ assigns to a center in vertex } j \\ 0 & \text{otherwise} \end{cases}$$

Both indices i and j have a range of $[1, N]$.

Objective function

$$\min z \tag{1}$$

$$\text{s.t. } \sum_{j \in N} d_{ij} x_{ij} \leq z \tag{2}$$

These two expressions ensure that the objective value is no less than the maximum vertex-to-center distance, which we want to minimize. Note that (2) is actually implemented as a constraint but shown here for the sake of readability.

Constraints

$$\sum_{j \in N} x_{ij} = 1 \quad \forall i \in N \tag{3}$$

$$x_{ij} \leq y_j \quad \forall i, j \in N \tag{4}$$

$$\sum_{j \in N} y_j \leq p \tag{5}$$

$$y_j \in \{0, 1\} \quad \forall j \in N \tag{6}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in N \tag{7}$$

Constraint (3) assigns each vertex to exactly one center. Constraint (4) ensures that no vertex assigns to v_j unless there is a center at v_j . Constraint (5) restricts the number of centers to p . Constraints (6) and (7) are the binary restrictions for variables x and y .

2.2 Calik and Tansel (2013)

This is the formulation referred to as (P3) in the original paper. This method uses the fact that the distance d_{ij} are the only possible values for $r_p(F)$. It is thus possible to jump from one d_{ij} to another.

In this formulation, we define the set $R = \{\rho_1, \rho_2, \dots, \rho_K\}$ where $\rho_1 < \rho_2 < \dots < \rho_K$ is an ordering of the distinct distance values of the matrix of distances d_{ij} . One of these values determines the value of $r_p(F)$.

Parameters

$$p = \text{maximum number of centers;}$$

$$N = \text{number of vertices of the instance;}$$

$$K = \text{number of distinct distance values in the instance.}$$

Variables Three variables are used in this formulation:

$$\begin{aligned} a_{ijk} &= \begin{cases} 1 & \text{if } d_{ij} \leq \rho_k \\ 0 & \text{otherwise} \end{cases} \\ y_j &= \begin{cases} 1 & \text{if vertex } j \text{ is a center} \\ 0 & \text{otherwise} \end{cases} \\ z_k &= \begin{cases} 1 & \text{if } r_p(F) = \rho_k \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Both indices i and j have a range of $[1, N]$.

Objective function

$$\min \sum_{k \in T} \rho_k z_k \quad (8)$$

The objective function determines the value of $r_p(F)$ as the corresponding value ρ_k .

Constraints

$$\sum_{j \in M} a_{ijk} y_j \geq z_k \quad \forall i \in N, \forall k \in T \quad (9)$$

$$\sum_{j \in M} y_j \leq p \quad (10)$$

$$\sum_{k \in T} z_k = 1 \quad (11)$$

$$y_j \in \{0, 1\} \quad \forall j \in M \quad (12)$$

$$z_k \in \{0, 1\} \quad \forall k \in T \quad (13)$$

Constraint (9) ensures that each vertex is covered within the selected radius by at least one center. Constraint (10) restricts the number of center to at most p centers. Constraint (11) ensures that exactly one of the variables z_k is selected. Constraints (12) and (13) are the binary restrictions for variables y and z .

3 Implementation

4 Optimization

4.1 Hot starts

Three hot starts have been tested: a random initialization, the 2-approximation algorithms and the initialization implemented in the Gurobi Optimizer.

4.1.1 Random initialization

The random initialization simply consists on choosing p points to be the centers.

As this algorithm is not demanding in computational resources, this initialization is done 1000 times and the best solution is kept as actual initial solution for the program linear.

4.1.2 2-approximation algorithm

The 2-approximation algorithm, also called the farthest-first traversal, is an heuristic that guaranties the solution will not be further away than 2 times the optimal value of the objective function if the triangular inequalities is respected.

The procedure is described in the pseudo-code here after. The first center is chosen randomly among the n points. The minimal distance between each of the points left and the already chosen centers is computed: this gives us for each point the smallest distance it is located from any of the centers. This distance is then maximized in order to chose the next center. This procedure is repeated until the number of centers chosen is equal to p .

2-approximation (π) :

```
input:    problem instance  $\pi$ 
           the number of points  $n$ 
           the number of centers  $p$ 
output:   solution  $sol$ 

centers[p] = randomNumber(1,n)
centersToFind = p - 1

while (centersToFind)
    outer_max = inf
    centerIdx = -1
    for i = 1:n and i not in centers :
        inner_min = findMinDistance(centers , instance)
        if inner_min > outer_max
            outer_max = inner_min
            centerIdx = i
        end
    end
    centers[centersToFind--] = centerIdx
end
sol = zeros(n)
sol[centers] = 1
return sol
```

end 2-approximation

In this case, it is not clear if the distance metric considered has the triangular inequality, this is why no assumptions on the optimality of this solution can be done. The name farthest-first traversal will thus be preferred.

4.1.3 Gurobi Optimizer

This one was not implemented but was still part of the one explored as it is the default one when use the Gurobi Optimizer.

4.1.4 Comparison of the heuristics

4.2 Number of variables

4.2.1 Dual problem

The first idea to reduce the number of variables is to solve the dual problem. Indeed if a problem has l variables and m constraints, its dual problem has m variables and l constraints. If $l > m$, the dual problem does reduce the number of variables. Let's have a look at the formulations and their number of constraints and variables.

Formulation	# variables	# constraints
P_1	$n^2 + n + 1$	$n^2 + 2n + 2$
P_3	$n + K$	$nK + 2$

Considering that n and $K \in \mathbb{N}$, the $\#variables < \#constraints$. For P_1 this is obvious. For P_2 , we have to solve the inequality $n + K > nK + 2$, which would indicate that taking the dual would reduce the number of variables. Using WolframAlpha, the solution of this inequation for $n > 1$ is $K < \frac{n-2}{n-1}$ which is always smaller than 1. As K is an integer, this gives $K \leq 0$ which does not lead to any solution as $K > 0$ by definition.

4.3 Valid inequalities

4.3.1 Chvátal-Gomory procedure

Procedure to construct a valid inequality for the set $X = \{x \in \mathbb{R}_+^n : Ax \leq b\} \cap \mathbb{Z}^n$, where A is a $m \times n$ matrix with columns $\{a_j, j = 1, \dots, n\}$:

$$\sum_{j=1}^n \lfloor ua_j \rfloor x_j \leq ub \text{ where } u \in \mathbb{R}_+^m \text{ and } x \geq 0 \text{ is valid for } P.$$

This inequation is used in constraint (2) of P_1 with a factor called *divisor*:

$$\sum_{j \in N} \lfloor d_{ij}/divisor \rfloor x_{ij} \leq z/divisor \quad (14)$$

The divisor parameter has been optimized blablabla.

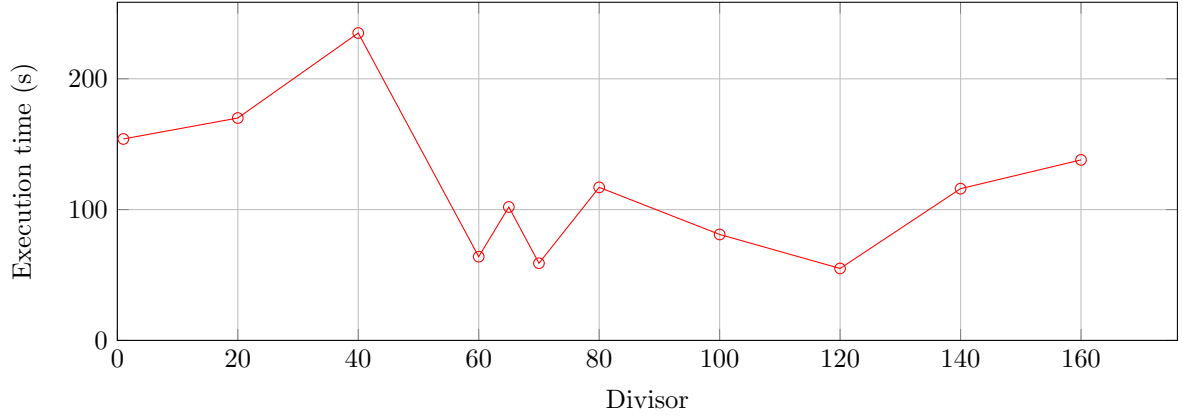


Figure 1: Qualified run-time distributions for qualities of 5% (in red), 2% (in green), 1% (in purple) and 0.5% (in cyan).

Multiple divisors have been used to add multiple ensemble of constraints.

With Gurobi there is not much improvement. However with another solver as Cbc we can see an improvement in the computational time.

5 Results

First, let us show the optimal values we found for each instance:

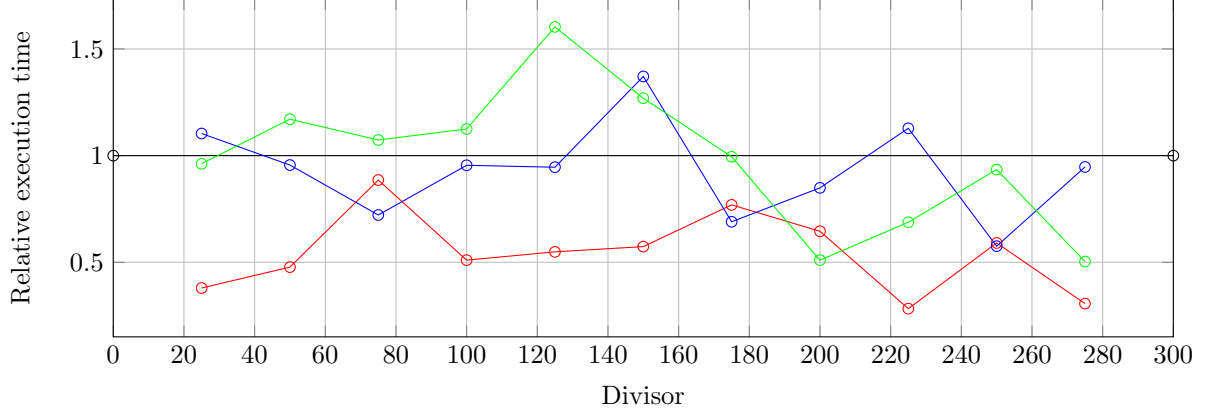


Figure 2: Attempt at finding the best divisor for instances of size $N = 100$. Straight lines are results without adding the VI. The relative time is expressed as the ratio of the execution time with a given divisor over the reference execution time, without the VI. Dots represent the performance with various divisors. Instance 1.out is in red, 2.out in blue, 3.out in green.

- 1.out: 127
- 2.out: 98
- 3.out: 93
- 4.out: 74
- 5.out: 48
- 6.out: 84
- 7.out: 64
- 8.out: 55
- 9.out: 37
- 10.out: 20

Next, let us see the execution time required for all instances:

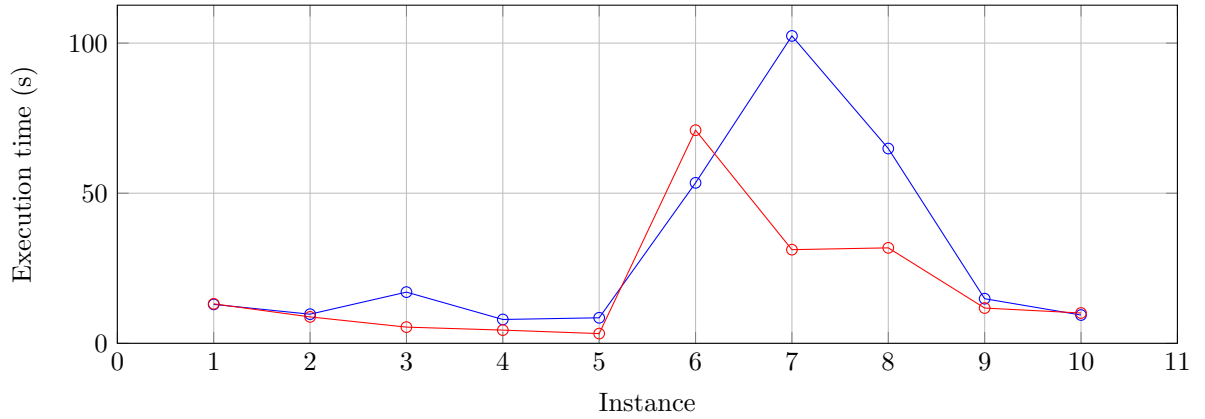


Figure 3: Execution time for Gurobi on all instances with P1 (in red) and P3 (in blue)

We see that the instances requiring the longest execution time are 6.out and 7.out. Not only they have a large number of vertices ($N = 200$), but also a very low p value, respectively 5 and 10. Intuitively, this low p combined with a high N increase the difficulty of the search.

Instance // Objective function // Example of solutions // computational time (mean) //

6 Conclusion

6.1 Comparison of the two formulations

6.2 Wrap-up

This project gave us interesting insight on how to practically use solvers and in a way, how they worked internally. Comparing our own implementations with Gurobi was a challenging and exciting task as we could see our improvements make the execution times required to solve instances decrease and decrease over time.