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**

2 Formulations **2**
2.1 Daskin (1995) . . . 2
2.2 Calik and Tansel (2013) . . . 3

3 Developer documentation **4**
3.1 Helper functions . . . 5

4 Optimization **5**
4.1 Hot starts . . . 5
4.1.1 Random initialization . . . 5
4.1.2 2-approximation algorithm . . . 6
4.1.3 Gurobi Optimizer . . . 6
4.1.4 Comparison of the heuristics . . . 6
4.2 Number of variables . . . 6
4.2.1 Dual problem . . . 6
4.3 Valid inequalities . . . 7

5 Results **8**

6 Conclusion **9**
6.1 Comparison of the two formulations . . . 9
6.2 Wrap-up . . . 10

1 Introduction

This project is implemented in Julia, a high-level, high-performance dynamic programming language for numerical computing.

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

- `ArgParse`: used to parse command-line arguments
- `Cbc`: a simple LP solver
- `Gurobi`: a more complex, commercial LP solver. Only used if requested in `P1.jl` and `P3.jl`.

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

$$\begin{aligned} p &= \text{maximum number of centers;} \\ N &= \text{number of vertices of the instance.} \end{aligned}$$

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

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

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

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

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

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

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

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

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

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 $R = \{\rho_1, \rho_2, ..., \rho_K\}$ where $\rho_1 < \rho_2 < ... < \rho_K$ 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 = maximum number of centers;
N = number of vertices of the instance;
K = number of distinct distance values in the instance.

Variables Three variables are used in this formulation:

$$a_{ijk} = \begin{cases} 1 & \text{if } d_{ij} \leq \rho_k \\ 0 & \text{otherwise} \end{cases}$$

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

$$z_k = \begin{cases} 1 & \text{if } r_p(F) = \rho_k \\ 0 & \text{otherwise} \end{cases}$$

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

Objective function

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

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

Constraints

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

$$\sum_{j \in M} y_j \leq p \tag{10}$$

$$\sum_{k \in T} z_k = 1 \tag{11}$$

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

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

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 Developer documentation

- `main.jl` : Entry point of the program
- `P1.jl` : Formulation P_1
- `P3.jl` : Formulation P_3
- `run_all_gurobi_P1.sh` : Run all instances with formulation P_1 with Gurobi as solver
- `run_all_gurobi_P3.sh` : Run all instances with formulation P_3 with Gurobi as solver
- `chvatal-gomory-divisor-search.sh` and `chvatal-gomor-divisor-search-2.sh` : Bash loops implementing a grid search to try and find the best divisor value (divisor is explained later on).
- `README.md` : Read me containing primary informations on the project and how to compile
- `instances/` : All the instances to test
- `out/` : Example outputs of some executions
- `report/` : Report (in LaTeX and in PDF)
- `resources/` : Paper and project formulation

3.1 Helper functions

PCInstance.jl Type PCInstance, contains 5 attributes: n, p, d, K, rho.

read_instance.jl Input: path, Output: instance::PCInstance

Parse an instance (given in path) and returns a PCInstance

Result.jl Type Result, contains 2 attributes: score, time.

twoapprox_heuristic.jl Input: instance::PCInstance, Output: solution::ArrayUInt8,n

Function returning the 2-approximation heuristic of an instance

random_heuristic.jl Input: instance::PCInstance, Output: solution::ArrayUInt8,n

Function returning a random solution of an instance

bestHeuristicRandom.jl Input: instance::PCInstance, Output: solution::ArrayUInt8,n

Returns the best solution out of 1000 random solutions

bestHeuristicTwoApprox.jl Input: instance::PCInstance, Output: solution::ArrayUInt8,n

Select the best solution out of 10 solutions found by the 2-approx heuristic

bestHeuristicMix.jl Input: instance::PCInstance, Output: solution::ArrayUInt8,n

Select the best solution out of 10 solutions found by the 2-approximation heuristic and 1000 random solutions.

print_solution.jl Input: instance::PCInstance, solution::ArrayUInt8,n

Print the position of the centers.

avg-divisor-search.py Python file to realize the grid search on the parameter "divisor"

4 Optimization

4.1 Hot starts

Three hot starts have been tested: a random initialization, the 2-approximation algorithms and the default initialization implemented in the Gurobi Optimizer or Cbc, depending on which is used in the current execution.

4.1.1 Random initialization

The random initialization simply consists of 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 the actual initial solution for the program.

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 π
             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 whether the distance metric considered has the triangular inequality property, this is why no assumptions on the optimally 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 using the Gurobi Optimizer and it yields good results.

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 $K \in N$, we have that #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. The solution of this inequality 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

We used the Chvátal-Gomory procedure to add valid inequalities to the set of constraints.

To construct a valid inequality for the set $X = \{x \in R^n_+ : Ax \leq b\} \cap Z^n$, where A is a $m \times n$ matrix with columns $\{a_j, \text{j} = 1, \ldots, \text{n}\}$, we write:

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

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

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

We tried looking for a value of the divisor which would improve the performance as much as possible. It is clear that the strength of the inequality depends on how the floor operation *cuts* the coefficients. So both the values in the distance matrix and the divisor have an important impact. Let us run our program using Cbc on the first instance to have some insight on how the divisor influences the execution time:

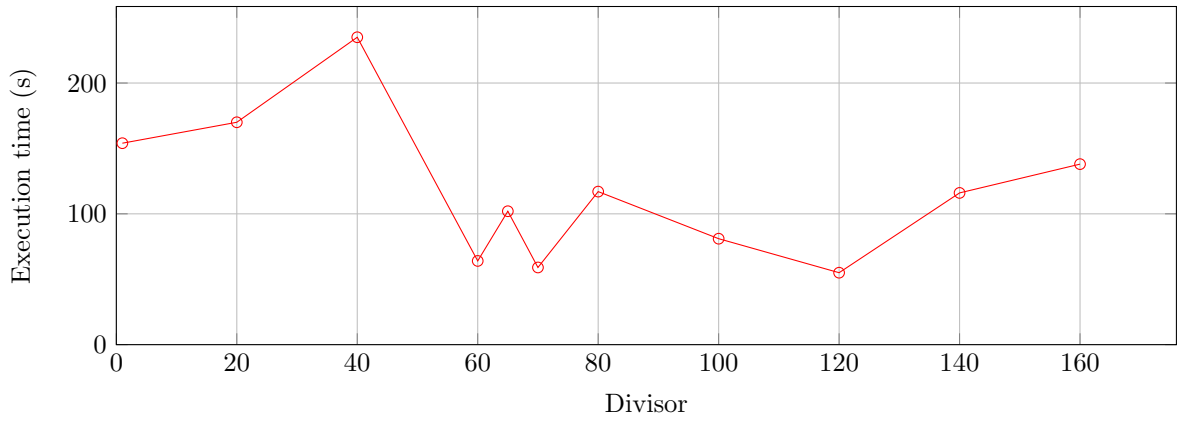

Figure 1: Example executions on the first instance. The search was first carried out systematically on every 20 values, then refined around what seemed to be a minimum, around 60-70.

We also tried using multiple divisors and add an ensemble of constraints. Again, this led to a great variety of results.

Let's however remark that using Gurobi, adding these valid inequalities does not have much effect: Gurobi itself already does implement operations (and actually, many more), so our valid inequality probably gets drowned into the pool of optimizations made by Gurobi.

With Cbc, which is much more rudimentary, we could see astonishing improvements in the computational time (sometimes leading to more than 200% speed-ups). Here is a plot of various divisor values used on the first three instances:

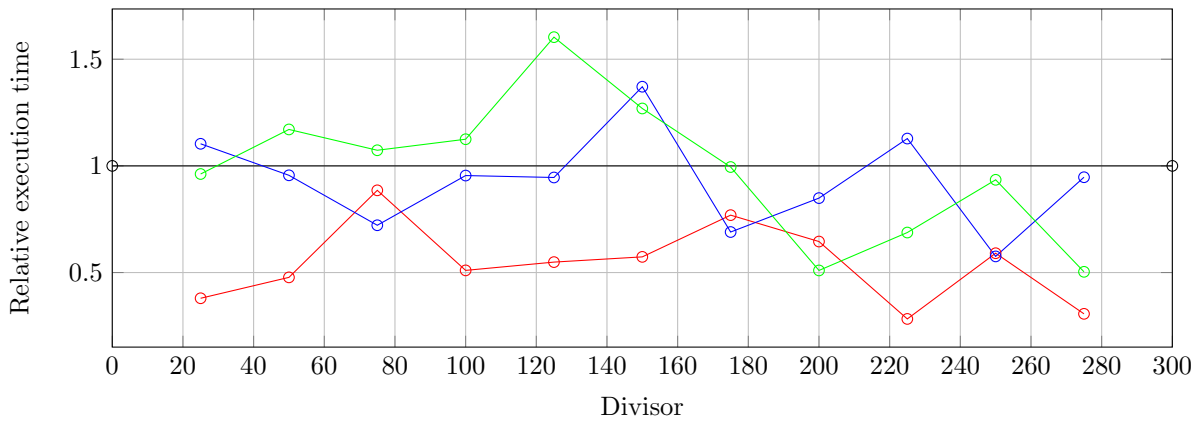

Figure 2: Attempt at finding the best divisor for instances of size $N = 100$. Straight lines are results without adding the VI. The relative time is expressed as the ratio of the execution time with a given divisor over the reference execution time, without the VI. Dots represent the performance with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

There does not seem to be a global optimum for all instances, even though they have roughly the same distribution of distances. On instance 1, it seems to work great. On the two others, we have a curious behaviour: we have been asking ourselves why we could get times greater than before when adding a VI. We couldn't come up with a certain answer: perhaps our valid inequality is actually not valid, or it somehow makes Cbc shy away from the optimal solution, or again some other aspect we don't know of is involved. This is left as an open question.

To conclude on this attempt to optimize, we ran several other tests (see the performance comparison in conclusion) and observed that this VI implementation didn't actually help in most cases, considering all instances.

5 Results

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

- `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

We had the same results with both formulations.

6 Conclusion

6.1 Comparison of the two formulations

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

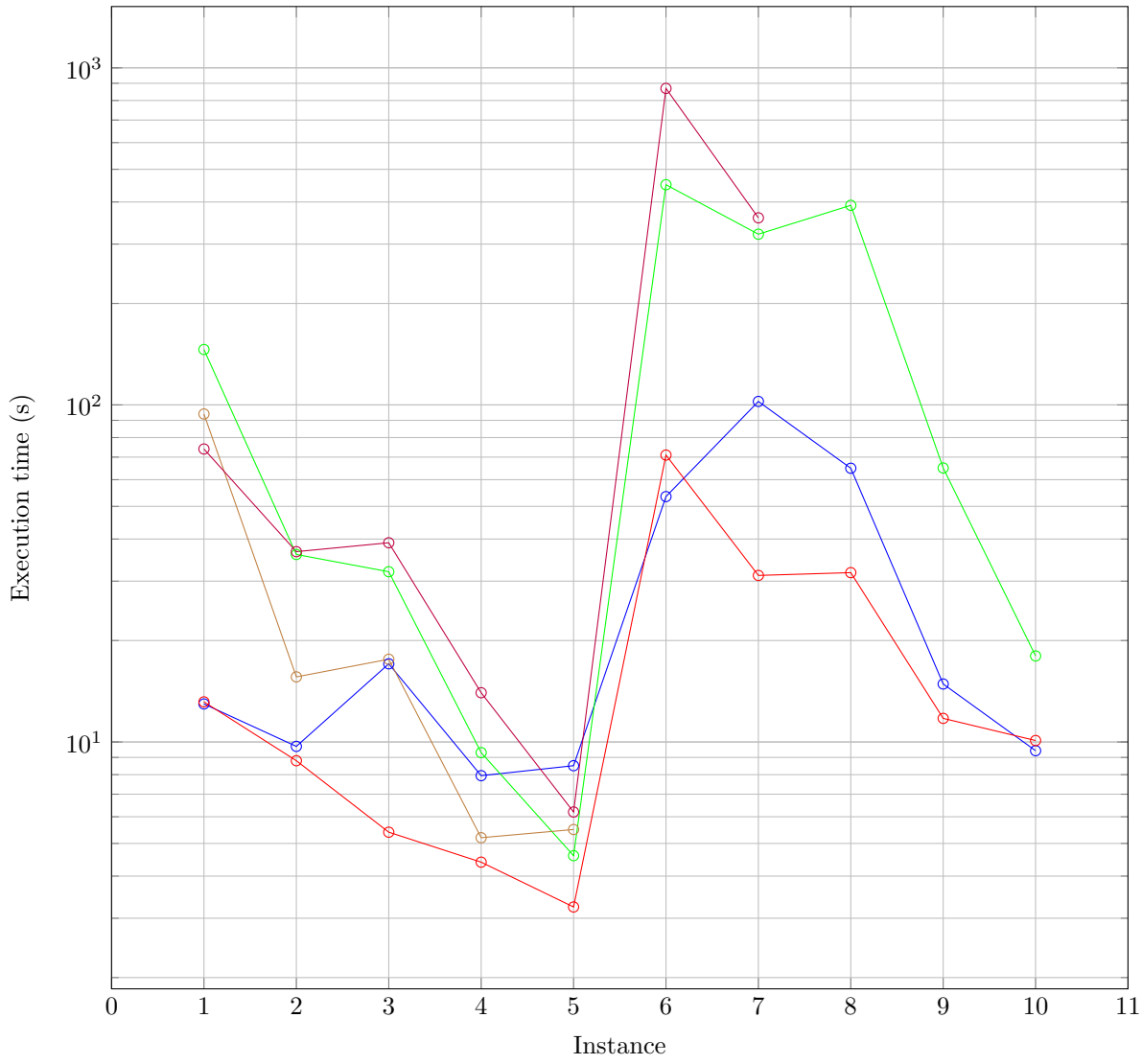

Figure 3: Execution time for Gurobi on all instances with P1 (in red) and P3 (in blue) and for Cbc with P1, without any optimization (in green), using the random heuristic (in brown), using the farthest-first heuristic (in cyan) and using the VI attempt with divisor 100 (in purple). Cbc with P3 would take too much time and we would often not see the end of the computation, it is therefore not relevant to display it here along the others.

Note how we had to use a logarithmic scale for the vertical axis to keep the visualisation relevant. Indeed, Cbc could reach very long times compared to Gurobi.

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.

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.