ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

Engineering and Architecture Department

Master Degree in Computer Engineering

Project Work in Data Mining M

Presented by
Alessandro Franca

Professor
Claudio Sartori

1. Introduction

Outlier detection in large data sets is an active research field in data mining that has many applications in all those domains that can lead to illegal or abnormal behavior, such as fraud detection, network intrusion detection, insurance fraud, medical diagnosis, marketing, or customer segmentation.
Many supervised approaches to outlier mining first learn a model over example data already labeled as exceptional or not, and then evaluate a given input as normal or outlier depending on how well it fits the model. *Unsupervised* methods, instead, have the task of discriminating each datum as normal or exceptional when the training examples are not labeled.
Among the unsupervised approaches, *distance-based outlier detection* methods distinguish an object as an *outlier* on the base of the distance to its nearest neighbors.
The research work exposed in "Fabrizio Angiulli, Stefano Basta, and Clara Pizzuti - Distance-Based Detection and Prediction of Outliers" (2006) proposes a distance-based outlier detection method that finds the top outliers in an unlabeled data set and provides a subset of it, called *Outlier Detection Solving Set*, that can be used to predict the outlierness of new unseen objects. The solving set includes a sufficient number of points that permits the detection of the top outliers by considering only a subset of all the pairwise distances from the data set.

This project work aims to implement the proposed *SolvingSet* algorithm using Julia, a general purpose programming language, designed for high computation purposes: an essential peculiarity for calculating distances on large scale datasets.

2. Julia Programming Language

Julia is a *high-level*, *high-performance*, *dynamic* programming language. While it is a general-purpose language and can be used to write any application, many of its features are well suited for numerical analysis and computational science.

Distinctive aspects of Julia's design include a type system with parametric polymorphism in a dynamic programming language; with *multiple dispatch* as its core programming paradigm. Julia supports *concurrent*, *parallel* and *distributed computing*, and direct calling of C and Fortran libraries without glue code.

2.1 High-performance

In computing, just-in-time compilation, also known as dynamic translation, is compilation done during execution of a program or at run time rather than before execution. Most often this consists of translation to machine code, which is then executed directly but can also refer to translation to another format.

Julia has an LLVM (Low-Level Virtual Machine) based just-in-time (JIT) compiler combined with the language's design to allow it to approach and match the great performance of C language.

Julia provides ease and expressiveness for high-level numerical computing, in the same way as languages such as R, MATLAB, and Python, but also supports general programming. To achieve this, Julia builds upon the lineage of mathematical programming languages, but also borrows much from popular dynamic languages, including Lisp, Perl, Python, Lua, and Ruby.

2.2 Multiple Dispatch

While the casual programmer need not explicitly use *types* or *multiple dispatch*, they are the core unifying features of Julia: functions are defined on different combinations of argument types, and applied by dispatching to the most specific matching definition.

This model is a good fit for mathematical programming, where it is unnatural for the first argument to "own" an operation as in traditional object-oriented dispatch. Operators are just functions with special notation.

Julia aims to create an unprecedented combination of ease-of-use, power, and efficiency in a single language. In addition to the above, some advantages of Julia over comparable systems include:

- User-defined types are as fast and compact as built-ins
- No need to vectorize code for performance; devectorized code is fast
- Lightweight "green" threading (coroutines)
- Elegant and extensible conversions and promotions for numeric and other types
- Efficient support for Unicode, including but not limited to UTF-8
- Call C functions directly (no wrappers or special APIs needed)
- Powerful shell-like capabilities for managing other processes

3. The SolvingSet algorithm

The description provided in this chapter is largely taken from *Angiulli, Basta, Lodi, and Sartori (2016)* , which is also a descendant, along a different research thread, of *Angiulli et al. (2006)*.

In this section, we briefly recall the key concepts of the SolvingSet-based approach, which was used in *Angiulli et al. (2006)* to predict novel outliers.
In the following, we assume a dataset D of objects is given, which is a finite subset of a certain metric space.

Definition 3.1 (Outlier weight). Given an object $p \in D$, the weight $w_k(p,D)$ of p in D is the sum of the distances from p to its k nearest neighbours in D.

Definition 3.2 (Top-n outliers). Let *Top* be a subset of D having size n. If there not exist objects $x \in Top$ and y in $(D \setminus Top)$ such that $w_k(y,D) > w_k(x,D)$, then *Top* is said to be the set of the top n outliers in D . In such a case, $w_* = min_{x \in Top}\, w_k(x,D)$ is said to be the weight of the top n-th outlier, and the objects in *Top* are said to be the top n outliers in D .

3.1 The SolvingSet-based approach

Now we recall the notion of *solving set* and the *SolvingSet* algorithm.

Definition 3.3 (Outlier Detection Solving Set). An outlier detection solving set S is a subset S of D such that, for each $y \in D \setminus S$, it holds that $w_k(y,S) \leq w_*$, where w_* is the weight of the top n-th outlier in D.

A solving set S always contains the set *Top* of the *top-n* outliers in D . Furthermore, a solving set can be used to predict novel outliers *(Angiulli et al., 2006)*. Our goal is to compute both a solving set S and the set *Top* .
The *SolvingSet algorithm* *(Angiulli et al., 2006)* is shown in *Fig. 1* and its working logic is described below.
At each iteration (let us denote by j the generic iteration number), the SolvingSet algorithm compares all dataset objects with a selected small subset of the overall dataset, called C_j (for candidate objects), and stores their k nearest neighbours with respect to the set $C_1 \cup \ldots \cup C_j$. From these stored neighbours, an upper bound to the true weight of each data set object can thus be obtained. Moreover, since the candidate objects have been compared with all the dataset objects, their true weights are known. The objects having weight upper bound lower than the n-th greatest weight associated with a candidate object, are called *non-active* (since these objects cannot belong to the *top-n* outliers), while the others are called *active* . At the beginning, C_1 contains m randomly selected objects from D, while, at each subsequent iteration j, C_j is built by selecting, among the active objects of the dataset not already inserted in C_1 , . . . , C_{j-1} during the previous iterations, the m objects having the maximum current weight upper bounds. During the computation, if an object becomes

non-active, then it will not be considered anymore for insertion into the set of candidate points, because it cannot be an outlier.

As the algorithm processes new objects, more accurate weights are computed and the number of non active objects increases. The algorithm stops when no more objects have to be examined, i.e. when all the objects not yet selected as candidate points are non active, and thus C_j becomes empty. The solving set is the union of the sets C_j computed during each iteration.

Input: Dataset D, a distance function $dist(\cdot,\cdot)$, integer number n of outliers, integer number k of nearest neighbours, integer number m of candidate points.
Output: Solving set of D, set of the top-n outliers of D.

```
(1)  SolvingSet(D, dist, n, k, m) {
(2)    PointSet SolvSet = new PointSet();
(3)    PointSet C = new PointSet(m);
(4)    MinHeap Top = new MinHeap(n);
(5)    MinHeap NextC = new MinHeap(m);
(6)    for i = 1 to D.length
(7)        D.get(i).NN = new MaxHeap(k);
(8)    C.set(D.RandomSelect(m));
(9)    while C.length ≠ 0 {
(10)     SolvSet.append(C);
(11)     D.drop(C);
(12)     for i = 1 to C.length {
(13)         p = C.get(i);
(14)         for j = 1 to C.length {
(15)             q = C.get(j);
(16)             d = dist(p, q);
(17)             p.NN.updateMin(d);
(18)             if i ≠ j then q.NN.updateMin(d);
(19)         }
(20)     }
(21)     for i = 1 to D.length {
(22)       p = D.get(i);
(23)       for j = 1 to C.length {
(24)         q = C.get(j);
(25)         if max(p.NN.weight(), q.NN.weight()) ≥ Top.min() {
(26)           d = dist(p, q);
(27)           p.NN.updateMin(d);
(28)           q.NN.updateMin(d);
(29)         }
(30)       }
(31)       if p.NN.weight() ≥ Top.min() then NextC.updateMax(p, p.NN.weight());
(32)     }
(33)     for i = 1 to C.length {
(34)         q = C.get(i);
(35)         Top.updateMax(q, q.NN.weight());
(36)     }
(37)     C.set(NextC.get());
(38)   }
(39)   return(⟨SolvSet, Top.getElements()⟩);
(40) }
```

Figure 1 - The SolvingSet algorithm.

4. SolvingSet Implementation in Julia

As anticipated, the goal of this project work was to provide an implementation of the *SolvingSet algorithm* (Chapter 2) and therefore, to solve the *ODP* problem, using the Julia programming language.
First, it has been necessary to fully understand the concept of *multiple-dispatch*. Therefore a first version of the algorithm has been developed as a script, with the unique objective of becoming familiar with the language. We will not go into this beta version.

The final version represents the *centralized SolvingSet algorithm*. An object oriented programming style has been followed, and a Master/Slave architecture has been chosen, thinking about scalability and future implementations.

A detailed explanation is provided below. The figure will be taken as a reference.

4.1 Data Structures and main Functions

Julia does not have classes in the object-oriented sense as it is not an object-oriented language. The main paradigm of Julia which replaces object-oriented programming is *multiple dispatch*.

Taking Python as reference language, the closest Julia analogy to a Python `class` is a `mutable struct`. The biggest difference between them, is that in almost all use cases the latter do not “own” methods, only data. These are initialized with constructors which are analogous with the Python `init` method.

To define what in an object oriented style would have been a method, one possibility is to use functions that take as arguments an instance of the desired data structure.

4.1.1 MasterInstance and SlaveInstance

*MasterInstance.jl*contains the struct `MasterInstance`, representing an instance of the Master side problem.
It contains the configuration parameters (`n,m,k,DistanceFunction`), two sets representing the Solution of the ODP problem: *solvingSet* and the *solution* (“*Top*” in *Fig. 1*), and other management parameters.

Factory method:

+ `createMasterInstance(cfg::MasterConfiguration)`
  Factory method that creates a `MasterInstance` from a configuration

SlaveInstance.jl contains the struct `SlaveInstance`, representing an instance of the Slave.
It contains the configuration parameters (taken from Master), and other management parameters regarding the dataset, which will be local to the slave slave.

This data structure is associated with the following self-explanatory functions:

+ `getDatasetName(s::SlaveInstance)`
+ `getDatasetDimension(s::SlaveInstance)`
+ `getPointDimension(s::SlaveInstance)`
+ `loadDataset(s::SlaveInstance)`

Factory method:

+ `createSlaveInstance(path::String)`
  Factory method that creates a `SlaveInstance` from a string representing the dataset's path.

4.1.2 Heap

Heap.jl contains two data structures: one representing the nearest neighbours of a Point, and the other the Solution set of the ODP problem.

Both extend the abstract type `Heap` - *(Abstract Factory pattern).*

`NN`: data structure representing the heap of the nearest neighbours of a Point.
Composed by: an ordered tree-like data structure, used to represent in a decreasing order the distances of the neighbours of the Point; a field representing the maximum capacity of the heap, and a field representing the weight of the point, as the sum of the distances between the point and his neighbourhood.

`MinHeap`: data structure representing the *Solution* ("*Top*" in *Fig. 1*).
Composed by: an ordered tree-like data structure, sorted by increasing point weight; and a field representing the minimum capacity of the heap ("*m*" in *Fig. 1*).

Associated methods:

+ `updateMin(h::NN,d::Float)`

Insert the input distance into `NN` if the heap is not full, or if this distance is lower than the first element of the tree. The total weight is updated accordingly.

+ `updateMax(h::MinHeap,el::Element)`

Insert the input element into `MinHeap`, if the heap is not full, or if the element's weight is higher than the first element of the tree.

4.1.3 Element

Element.jl contains the definition of the `Point` data structure, represented by: point id, coordinates, `NN` heap, and a boolean to check if the point is active or not.
It extends the abstract type `Element` - *(Abstract Factory pattern).*

4.1.4 ODPMasterAlgorithm

ODPMasterAlgorithm.jl contains a data structure representing an instance of the algorithm Master side. Main fields: a `MasterInstance`, and other variables to store the algorithm solution.

Main associated methods:

+ `runODPAlgorithm(this::ODPMasterAlgorithm, filepath::String)`
  Runs the SolvingSet algorithm. Input: an instance of `ODPMasterAlgorithm`.

Factory method:

+ `getMasterAlgorithm(cfg::MasterConfiguration)`
  Factory method to create an ODPSlaveInstance from a configuration.

4.1.5 ODPSlaveAlgorithm

ODPSlaveAlgorithm.jl contains a data structure representing an instance of the algorithm Server side. Felds: a `SlaveInstance`, and other management variables.

Main associated methods:

+ `setParam(this::ODPSlaveAlgorithm, n::Int64, k::Int64, m::Int64, dist::String):`
  Used to set the configuration parameters sent by the Master

+ `computeIteration(this::ODPSlaveAlgorithm, C::Set{Point}, lb::Float64):`
  Execute an algorithm iteration. Input: Heap of candidate Points, solvingSet lower bound.

Factory method:

+ `getSlaveAlgorithm(path::String)`
  Factory method to create an `ODPSlaveInstance` starting from a string representing the dataset's path.

4.1.6 IterationResult

IterationResult.jl contains `IterationResult` data structure, used to store the temporary results of each iteration between Master and Slave, such as the set of candidate points *C* for the next iteration , the number of calculated distances, and the number of points still *active* after the current iteration.

4.1 Algorithm Explanation

For the implementation of the algorithm itself, it has been fully complied with what described in *Chapter 2*. Therefore, for a detailed explanation of the algorithm, we recommend reading *Angiulli et al. (2006)*. and Angiulli F,, Basta S., Lodi S., Sartori C. (2020).

It is assumed that the dataset is local to the Slave. Therefore the Master has to send the algorithm parameters to the Slave, since the latter is responsible for running those parts of the algorithm that operate directly on the dataset.

4.1.2 Master:

Once the configuration parameters for the algorithm are set, the algorithm execution starts.

The termination conditions are defined, and the variables to contain the *Solution (Top)* and the *Solving Set* are initialized.
An instance of `IterationResult` is created.

The communication with the Slave can begin.
In order, the Master:

1. Sends the configuration parameters to the Slave.
2. Requires information about the dataset, such as: dataset name, dataset size, point dimension.
3. Asks the slave a set of *m* random points from the dataset, to be used as a set of *candidate points C* for the first iteration.
4. Runs a loop in which, while the termination conditions are not met, the Slave is asked to compute an iteration of the algorithm.
5. Awaits the answer and, once received, the number of calculated *distances*, the number of *active* points left, and the set of *candidate points (*to be analyzed at the next iteration) are updated.
6. Once the termination conditions are met, the *Solution* set and the *Solving Set* have been successfully filled.

4.1.3 Slave:

Once retrieved the configuration parameters from the Master, the Slave is ready to execute its part of the algorithm.

Its main tasks are:

1. Provide the Master with a set of *m candidate* points, chosen at random from the local *dataset*.
2. Compute an iteration of the algorithm. This phase consists of:
    a. Comparison of the *candidate set C* with itself in order to establish the weight of each *candidate* point.

    b. Comparison of the (*Dataset* \ *C*) set, with the candidate set *C*. In this step the weights of each point of the dataset are calculated, and the heap *NextCand* (candidate points to be given in input to the next iteration) is filled.
3. Once the previous steps have been completed, an instance of *IterationResult* is returned to the Master, including: number of calculated distances, the number of *active* points left, and *nextCand* heap.

5 Tests

The experiments have been conducted on a personal computer, equipped with an Intel(R) Core(TM) i7-4500U CPU 1.40GHz and 8GB of RAM.

The following tests were performed on 2d point datasets, in order to easily verify if the algorithm was implemented correctly.

5.1 Test 1

Fig. 2a shows the full data set *D* of 788 points in R^2.
In Fig. 2b, a SolvingSet *S* for `ODP<D, dist, 10, 6, 6>` where "dist" is the Euclidean distance, is depicted (the orange points represent the Solution set).
The algorithm seems to correctly identify the 10 outliers.

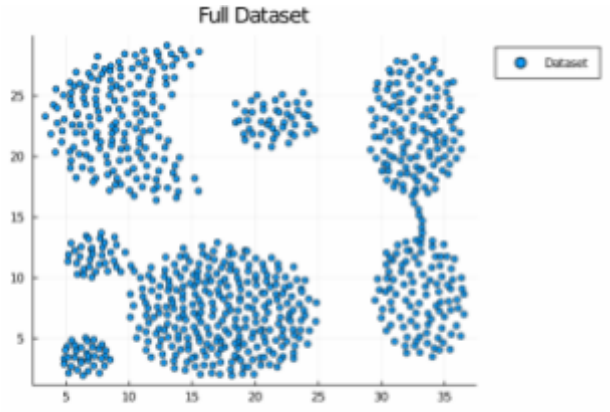

Figure 2a - Full dataset.

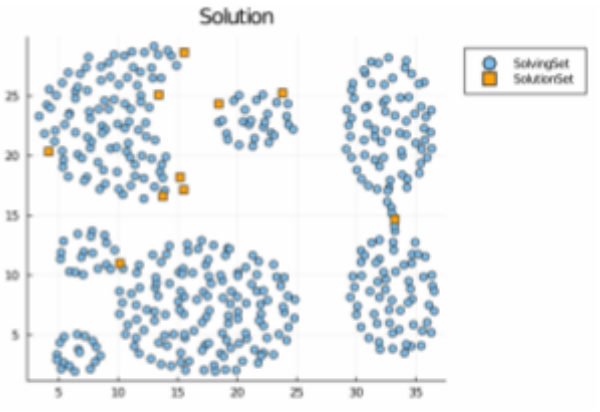

Figure 2b - SolvingSet and Solution set

Fig. 3 shows an example of execution of the algorithm *SolvingSet* on the data set of Fig. 2a. It reports the iterations 2 to 5 of the algorithm for n = 10, k = m= 6.
Green squares represent the points of the candidate set *C* during the current iteration, while orange circles represent the points composing the set *SolvingSet* at the beginning of the current iteration.

During the first iteration, the algorithm randomly selects six points, and then the six points to be inserted in *C* during the next iteration are computed (green squares in the top-left figure).

Notice that the points to be inserted in *C* during the next iteration are always those more distant from the current solving set *SolvingSet*,

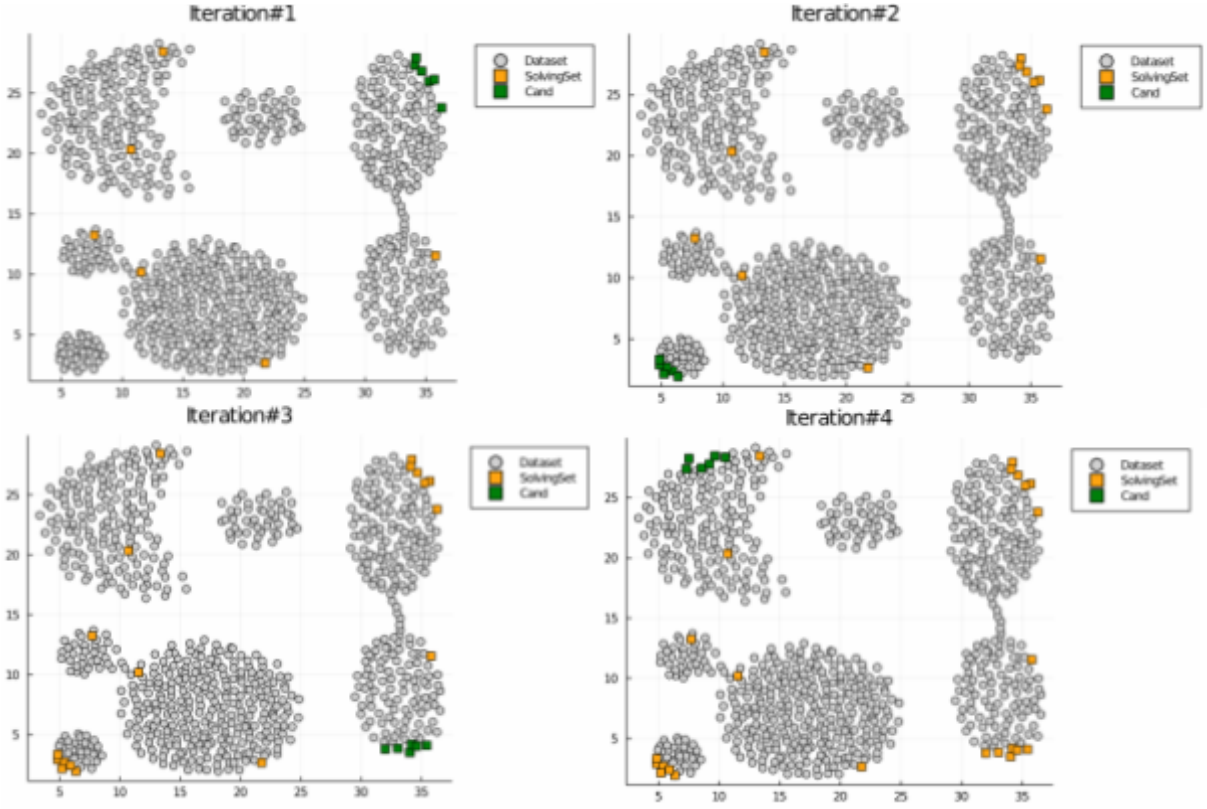

Figure 3 - First 4 algorithm iterations.

5.2 Test 2

For the second test, we considered a data set composed by N = 800.000 2D points, shown in Fig. 4 together with the top n = 100 outliers for k=m= 20.

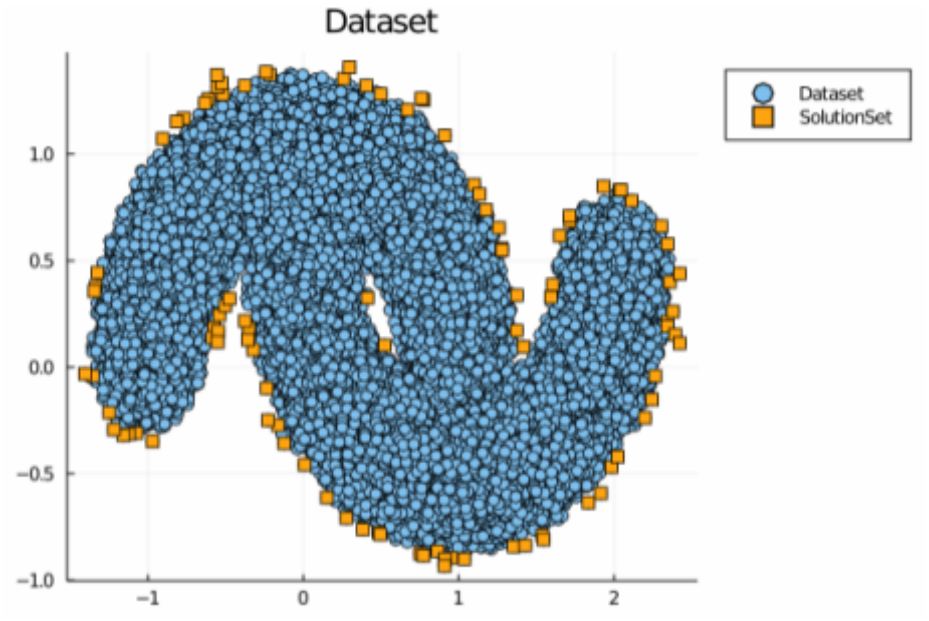

Figure 4 - Full dataset.

Figs. 5a shows the sizes |S| of the solving sets for `ODP<D, dist, 100, 20, 20>` obtained considering several values |D| of the data set size, using the Euclidean distance.
To vary the data set size, we considered in each experiment the data set obtained by picking distinct objects at random from the original data set. It is clear from these figures that the ratio |S|/|D| decreases dramatically with increasing values of |D|.

Fig. 5b depicts the execution times of the algorithm SolvingSet on the data of Fig. 4.
The figures show that the time trend of the algorithms roughly follows the product |S|*|D|.

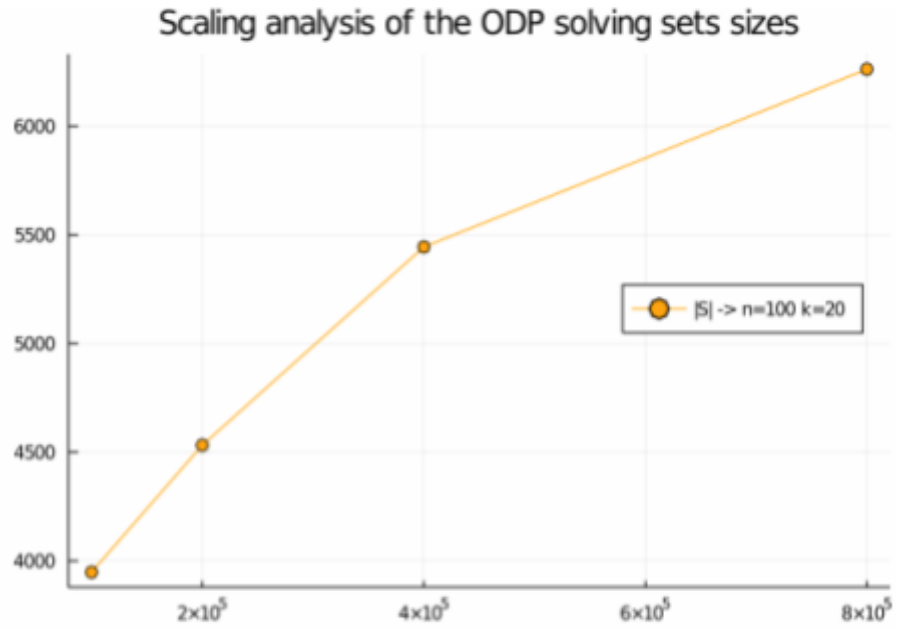

Fig. 5a - Scaling analysis of the ODP solving sets size. The x-axis represents the size of the data set, while the y-axis represents the size of the solving set.

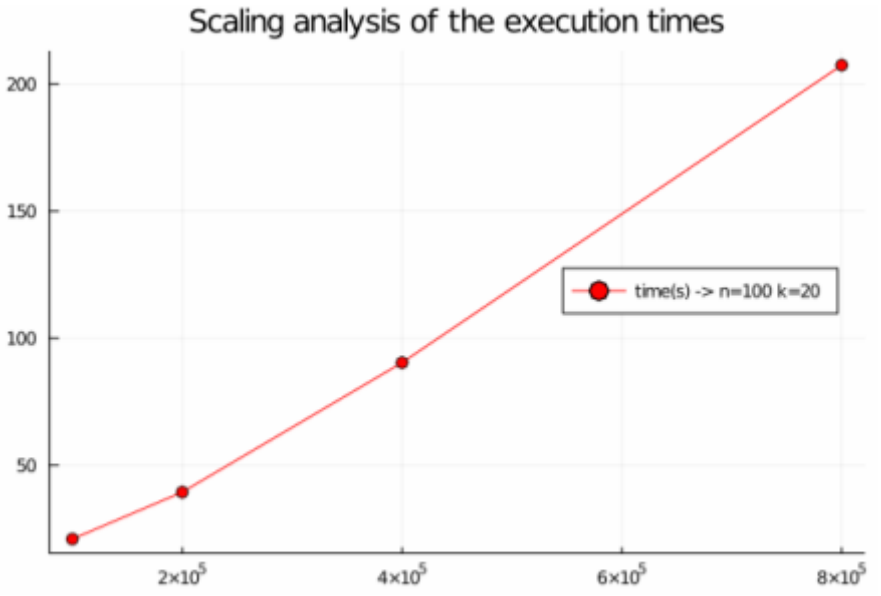

Fig. 5b - Scaling analysis of the execution times of algorithm SolvingSet. The x-axis represents the size of the data set, while the y-axis the execution time in seconds.

6. Conclusions

During this project work, a working version of the centralized SolvingSet algorithm has been developed using Julia. A local Master/Slave model was followed.

Talking about the development experience with the new programming language, some initial difficulties were encountered due to the habit of programming with object-oriented languages.
The main difficulty was therefore to develop the software following an object-oriented programming style, taking advantage of Julia's main feature: multi-dispatch.
Once the operation was understood, it was not difficult to proceed, and get to the expected result.

It is recommended to perform tests with larger data sets, and of a different nature.
As for possible future developments, a version of the distributed SolvingSet algorithm could be implemented.
The code can also be easily integrated with algorithm improvements, such as *FastSolvingSet*.

7. Bibliography

Angiulli, F. , Basta, S. , & Pizzuti C. (2006). Distance-based detection and prediction of outliers. IEEE Transactions on Knowledge and Data Engineering, 18 , 145–160.

Angiulli F., Basta S., Lodi S., Sartori C. Reducing distance computations for distance-based outliers. Expert Systems With Applications 147 (2020) 113215.

Angiulli, F. , Basta, S. , Lodi, S. , & Sartori, C. (2016). GPU strategies for distance-based outlier detection. IEEE Transactions on Parallel and Distributed Systems, 27 , 3256–3268.

The Julia Programming Language.

Julia 1.5 Documentation.