Homework 3 : Simulated Annealing and Genetic Algorithm

University of Nantes
Master 1 Optimization in operational research
Class of Metaheuristics
Professor : Xavier Gandibleux

UNIVERSITÉ DE NANTES

Léon Ferrari
University of Nantes
Master 1 Optimization in operational research

Written with LaTeX

November 17, 2017

Contents

1 Introduction to the Set Packing Problem

The Set Packing Problem with a finite size n of item and m of constraints.
It can be define as follow, with :

$$\begin{array}{llll} Max\,Z & = & \sum_{i=1}^{n} x_i \times c_i & \\ s.c. & \sum_{i=1}^{m}\sum_{j=1}^{n} x_{i,j} \times x_i & \leq & 1 \\ x_i \in \{0,1\} & & & \end{array}$$

With :

$$x_i \quad where \left\{ \begin{array}{ll} 1 & if\, x_i \in solution. \\ 0 & otherwise \end{array} \right\}$$

$$x_{i,j} \quad where \left\{ \begin{array}{ll} 1 & if\, x_{i,j} \in constraint\, i. \\ 0 & otherwise \end{array} \right\}$$

$i \in [1, n], \quad j \in [1, m]$
x_i a variable of the solution.
c_i the cost of variable i.
$x_{i,j}$ an element of the constraint matrix.
This type of problem can be illustrated in several ways, I will introduce you to some of them.

Illustration 1 :
You are moving out of your apartment and you need to pack your different item.
Obviously some of theses objects are more important for you or more expensive.
But you do not have enough boxes so you will have to choose which one you will protect from a possible break.
Here the coefficient of the objective value will be the value of the different items and the constraints will represent the boxes that you own.

Illustration 2 :
You are a manager and you have to create a sport team for your company. In order to select them you send them to pass different test.
You also asked them with who they don't want be if they are in the sport team.
Here the coefficient of the objective value will be the result of the test and the constraints will represent the people they don't want to be with.

1.1 JuMP modelisation for the set packing problem

The modelling of the problem have been done with the language Julia and the package JuMP (Julia for mathematical programming). We used the GLPKSolverMIP , the GLPK solver for mixed-integer problem. Once the data describing the problem have been extract from the file, we can create a model with JuMP and send it to GLPK in order to solve it with the following instructions :

Listing 1: JuMP and GLPK for SPP

```
#= This is a code modelling and solving a SPP problem with JuMP and GLPK)
model   = Model(solver=GLPKSolverMIP())
```

```
@variable( model ,  x[1:n], Bin)
@objective( model , Max, sum( Variables[j] * x[j] for j=1:n ) )
@constraint( model , cte[i=1:m], sum(Constraints[i,j]* x[j] for j=1:n)
<= 1 )
solve(model)
```

1.2 Ten instances of the SPP

Problem Name	Number of Variables	Number of constraints	Best known value
100rnd0100.dat	100	500	372
100rnd0200.dat	100	500	34
100rnd0300.dat	100	500	203
100rnd0400.dat	100	500	16
100rnd0500.dat	100	100	639
100rnd0600.dat	100	100	64
100rnd0700.dat	100	100	503
100rnd0800.dat	100	100	39
100rnd0900.dat	100	300	463
100rnd1000.dat	100	300	40

2 Genetic algorithm

During this second homework we had to choose an algorithm between ACO,GA and Simulated Annealing. I wanted to develop a genetic algorithm because of the many features that can be included and adjusted to fit the problem. For example there are many way of operating a crossover or a mutation. You can add some random individual into your population to diversify it and maybe avoid a local optimum etc.. All the tuning possible concerning the different crossover method, the probability of mutation etc. look interesting.

I will now describe all the different component of my algorithm, beginning by the structure of an individual also called solution.

2.1 Data structure

2.1.1 Problem data structure

To store the problem we will use two different type, a sparse matrix for the constraints and a vector for the coefficient of each variable.

I tried to use the advantage of a sparse matrix, I noticed a significant improvement in time and complexity. The SPP is highly constrained problem but because of its structure where only a few variable are present in each constraint, that is why using a sparse matrix is useful.

2.1.2 Chromosome data structure

A chromosome is set of variables that define a solution. The population of a Genetic algorithm is composed of chromosome.

In my algorithm the chromosome is a data-structure composed of a vector, the solution, and an integer, the fitness value of the solution. The fitness value here is just the objective value of the problem.

It's commonly noticed that the computation of the fitness value is the most expensive function in term of complexity. However the the structure of the set packing problem allow us to never really compute it because here we don't need the *f(x)* being only a set of simple addition. So we just need to add or subtract the coefficient of a variable every time it is add or remove from the solution.

2.1.3 Population data structure

The population will be a vector of chromosome ordered by decreasing order of objective value. And every time a new "chromosome" will be created he will be added in this array in a way to keep the order and the last element of the array will be deleted.

And if we keep the population ordered we do not need store the minimum or maximum fitness value of the population.

2.2 Building a feasible solution

2.2.1 Greedy randomized adaptive search procedure (GRASP)

In a genetic algorithm you handle a population and apply different method on this group of solution in order to increase it. You try to mix solutions in order to build a better one. Because a GA use this strategies in order to crate new individual you want your base population to be diverse. And GRASP is perfectly adapted to build this kind of solution.

It will allow us to have a diverse, good and feasible population. In order to build different good solution we will use the construction algorithm called GRASP from Mauricio G. C. Resende. The population for the genetic algorithm will be generated using GRASP. It is an easy to program and understand algorithm. The aim is too pick randomly the candidate which will be set to one through a random candidate list. The major interest of this algorithm is the technique used create the random candidate list therefore a parameter, α, is used to built it. α is a coefficient used in the creation of the random candidate list, it will be used to compute a lower bound limit for the selection of the candidate variable.

Once we have compute the utility for each variables, in the case of the SPP problem the so called "utility" is :

$$U_i = \frac{c_i}{\sum_{j=1}^{m} x_{j,i}}$$

In the following steps *Utility* will be considered as a Vector of size n containing the utility of each variable. The lower bound limit for the utility will be compute as follow :

$$\textit{Limit} = \min(\textit{Utility}) + \alpha \times (\max(\textit{Utility}) - \min(\textit{Utility}))$$

And now the construction algorithm :

```
Data: Alpha,V,C
Result: Solution
/* C the constraint matrix, V the variables cost */
/* Sort the array in decreasing order based on the second row */
Utility = $\forall v \in V, U_v = \frac{Cost_v}{\sum_{j=1}^{m} X_{j,v}}$
/* Sort by decreasing order */
sort(Utility)
while NotEmpty(RCL) do
    Limit = Min(Utility) + Alpha * (Max(Utility)- Min(Utility))
    RCL = $\emptyset$
    for i := 1 to N do
        if Utility[i] >= Limit then
            RCL = add(RCL,i)
        else
            break
        end
    end
    Candidate = RandomSelect(RCL)
    UpdateUtility(Utility)
    addToSolution(Solution,Candidate)
end
return Solution
```

Algorithm 1: GRASP Construction

2.3 Parent selection

A roulette selection is used to find the parents that will produce the new element of the population. The roulette selection is simple it will randomly select two parents X_1 and X_2 with the probability :

$$p_{xi} = \frac{f_i(x) - f_{min}(x)}{\sum_{j=1}^{n} (f_j(x) - f_{min}(x))}$$

In my algorithm four solution will be selected using the roulette selection and then a binary tournament will select the two future parents. A binary tournament is simple :

```
Data: $S_1, S_2, S_3, S_4$
Result: $P_1, P_2$
/* $S_i$ a solution $P_i$ a possible parent                */
if $S_1 > S_2$ then
    if $S_3 > S_4$ then
        $P_1 = S_1$ $P_2 = S_3$
    else
        $P_1 = S_1$ $P_2 = S_4$
    end
else
    if $S_3 > S_4$ then
        $P_1 = S_2$ $P_2 = S_3$
    else
        $P_1 = S_2$ $P_2 = S_4$
    end
end
return $P_1$,$P_2$
```

Algorithm 2: Binary tournament

2.4 Crossover method

The crossover method is the most important part of the algorithm. It will allow us to create new individual/solution from the other by crossing them. Multiple methods exist to do a crossover between to solution using a binary mask for example Each crossover in my algorithm will produce two children combination of the parents. Because the method use for the crossover have a high chance to make the new solution infeasible we will have to repair the solution after. It is a called the two-fusion operator[1] Here is the crossover algorithm :

```
1. if $P_{1,i} == P_{2,i}$
       set $C_{1,i} = C_{2,i} = P_{1,i}$
2. else
       set randomly k et k' $(k \neq k')$ a value in 1,2
       set $C_{1,j} = p_{k,j}$ with probability $p_k = f_k^f / f_k^f + f_{k'}^f$
         and $C_{1,j} = p_{k',j}$ with probability $1 - p_k$
       set $C_{2,j} = p_{k',j}$ with probability $p_{k'} = f_{k'}^f / f_k^f + f_{k'}^f$
         and $C_{2,j} = p_{k',j}$ with probability $1 - p_{k'}$
```

[1]A Genetic Algorithm-Based Heuristic for Solving the Weighted Maximum Independent Set and Some Equivalent Problems written by Mhand Hifi

```
**Data:** $P_1, P_2$
**Result:** $C_1, C_2$
/* $C_i$ a children, $P_i$ a parent                                    */
for $i := 1$ to $N$ do
    if $P_{1,i} == P_{2,i}$ then
        $C_{1,i} = P_{1,i}$
        $C_{2,i} = P_{1,i}$
    else
        $K_1 = \text{rand}(1 \vee 2)$
        if $K_1 == 1$ then
            $k_2 = 2$
        else
            $k_2 = 1$
        end
        $p_{k1} = f_{k1}(x)/f_{k1}(x) + f_{k2}(x)$
        $p_{k2} = f_{k2}(x)/f_{k1}(x) + f_{k2}(x)$
        if $rand() < p_{k1}$ then
            $C_{1,i} = P_{k1,i}$
        else
            $C_{1,i} = P_{k2,i}$
        end
        if $rand() < p_{k2}$ then
            $C_{2,i} = P_{k2,i}$
        else
            $C_{2,i} = P_{k1,i}$
        end
    end
end
return $C_1$,$C_2$
```

Algorithm 3: Crossover

2.5 Mutation

2.5.1 Repairing the solution

We explain that the crossover method may create infeasible solution. So we have to repair the solution, we use a simple method. Here is the algorithm :

```
Data: $S_1, C$
Result: $S_1$
/* $S_i$ a solution,$C$ the constraints                                  */
for $i$ := 1 to $N$ do
    if $S_{1,i} == 1$ then
        for $j$ := 1 to $M$ do
            if $C_{j,i} == 1$ then
                for $\forall S_{1,k} \in C_j$ do
                    if $S_{1,k} \neq S_{1,i}$ and $S_{1,k} = 1$ then
                        $S_{1,k} = 0$
                    end
                end
            end
        end
    end
end
return $S_1$
```

Algorithm 4: Repairing a solution

This the simple implementation of *RepairSolution*,I did it differently using sparse matrix characteristics. This look time consuming but it is implemented using only one for loop.

2.6 Improving a solution

It's a greedy algorithm of descent, where he go through the variable and if a variable is free he set it to one. I just implemented a greedy local search adding all possible variables than can be added to the solution while maintaining the feasibility of the solution.

1. $\forall V \notin Solution$

   if $V \cup Solution$ is a feasible solution

   then $Solution = V \cup Solution$

2.7 Parameters & stop conditions

Our algorithm don't have yet a mutation method because the crossover imply to repair and do LS on the solution, so it implicitly execute a mutation.

But we have a probability of crossover of 30%, knowing that two children are produced at each crossover it is a really high probability of crossover.

And our algorithm have two stop condition, when the number of generation is over the limit or when the population converge in value, we consider that an algorithm converge in value when $Stop_i$ is below 0.01 :

$$Stop = \frac{f_{max}(x) - f_{min}(x)}{f_{avg}(x)}$$

I set the number of generation to 30 but the algorithm will mainly stop because of the stop condition than because he reached the 30th generation. He will only reach the 30th generation for big sized problem.

3 Results and performances

In this chapter we will compare the result obtained by our previous "heuristics" and the genetic algorithm. I will also introduce the environment the algorithm is running on.

3.1 System and environment

We solved the problem on the following system :

OS	Centos 7 64 bits
Processor	i7 4720HQ
CPU Freq	2.6GHz
Physical Core CPU	4
RAM	8GB
Julia	Version 0.6.0
Solver	GLPK

3.2 Results

Below the heuristic construction go with the LS, the generation of the population is included in the time of the GA. We obtained the following result :

Problem	Grasp construction		Local search		Genetic algorithm	
Data	$\tilde{Z}_{avg}$	Time(s)	$\tilde{Z}_{avg}$	Time(s)	$\tilde{Z}_{avg}$	Time(s)
100rnd0100	338.755	0.0001	352.01	0.05	369.12	0.18
100rnd0200	29.675	0.0001	33.985	0.06	32.0	0.1
100rnd0300	181.265	0.0555	185.81	0.06	197.84	0.18
100rnd0400	13.295	0.0606	15.31	0.07	15.16	0.1
100rnd0500	617.24	0.0001	621.435	0.02	631.48	0.04
100rnd0600	61.5	0.0235	64	0.0067	64.0	0.04
100rnd0700	491.715	0.0001025	497.7	0.02	495.3	0.03
100rnd0800	35.99	0.0437	38.32	0.02	37.76	0.03
100rnd0900	431.85	0.0001	446.84	0.04	453.98	0.07
100rnd1000	36.17	0.0001	39.525	0.04	38.5	0.074

Here we can observe that the genetic algorithm is slower but find an higher average value in general.

We will compare now two algorithm the genetic algorithm and an algorithm based on multiple grasp construction where we apply a simulated annealing.

We will compare the two algorithm on these various sized problem :

Problem	N	M	Best Z
100rnd0100.dat	100	500	372
200rnd0300.dat	200	1000	731
500rnd0100.dat	500	2500	323
1000rnd0300.dat	1000	5000	661

Here are the different parameters used :

1. GRASP construction

   $\alpha = [0.5, 0.6, 0.75, 0.90]$

   $p_\alpha = [0.25, 0.25, 0.25, 0.25]$

2. Genetic algorithm

   Size of the population : 100.

   Generation : 30.

   Probability of crossover : 30%.

3. Simulated annealing

   20 construction

   Cooling coefficient : 0.95

   Init temperature : 1000

   Step size : 100

   Stop temperature : 1.0

For the parameter of population, I think It should be sized according to the problem but I don't know how I correctly tune it. For example we can have a bigger population for small sized (100 or 200 variables) problem as it take a small amount of time.

But for bigger we could may be reduce it to be faster however it is not really logical as "more" solution will exist so a bigger population will be needed to express the diversity of possible solution.

So for the problem *500rnd0100.dat* and *1000rnd0300.dat* we will run the genetic algorithm with different population size and stop conditions.

Problem	Simulated annealing			
	$\tilde{Z}_{min}$	$\tilde{Z}_{avg}$	$\tilde{Z}_{max}$	Time(s)
100rnd0100.dat	336	349.5	363	0.64
200rnd0300.dat	645	685.2	708	1.01
500rnd0100.dat	239	269.75	288	6.18
1000rnd0300.dat	457	524.15	573	12.38

Problem	Genetic algorithm							
	Parameter		f(x)			Time		
	Population	Limit	$\tilde{Z}_{min}$	$\tilde{Z}_{avg}$	$\tilde{Z}_{max}$	Min	Avg	Max
100rnd0100.dat	50	0.01	365	370.15	372	0.14	0.2	0.32
100rnd0100.dat	50	0.05	361	367.65	372	0.08	0.12	0.24
100rnd0100.dat	100	0.01	367	371.55	372	0.22	0.37	0.45
100rnd0100.dat	100	0.05	365	367.15	372	0.18	0.23	0.31
200rnd0300.dat	50	0.01	696	706	712	0.58	0.73	0.98
200rnd0300.dat	50	0.05	687	697.6	708	0.25	0.32	0.41
200rnd0300.dat	100	0.01	696	707.85	715	1.21	1.49	2.16
200rnd0300.dat	100	0.05	687	697.45	709	0.46	0.61	0.89
500rnd0100.dat	50	0.01	289	307.75	319	15.1	21.13	29.5
500rnd0100.dat	50	0.05	289	298.5	311	7.16	14.41	25.01
500rnd0100.dat	100	0.01	299	309.3	320	34.5	48.27	65.5
500rnd0100.dat	100	0.05	289	304.85	319	17	38.83	59.03
1000rnd0300.dat	50	0.01	568	589.4	634	68.06	110.04	141.96
1000rnd0300.dat	50	0.05	561	578.75	603	56	76.73	108
1000rnd0300.dat	100	0.01	575	596	611	194	276	329
1000rnd0300.dat	100	0.05	567	593.6	628	179.2	230.5	322.54

We can see that the GA has generally better result and if it is fast for small problem it is long for medium/big sized problem.

We can see that increasing the value of the limit or lowering the size of the population drastically change the execution time at the cost of the global score of the popullation.

But we are here interested only in the best solution found so far and not the "global value" of the population. If the population converging on a value can be a stop condition it does imply that all the population have the same value.

Lowering the value of the limit could be done for any problem even if it could lead to be blocked in a local optimum.

The couple GRASP + CS can found a solution in a really short amount of time compare to the GA but it is solution of lower quality. Also maybe if we modified some parameter to let it run as long as the GA it might found a better solution.

I have tried, and it didn't, the solution are always worst or equal to the average solution of the GA.

3.3 Performance

I collected the time spend in each function for the problem *500rnd0100.dat* :

Function Name	Role	Time
BinaryTournament	Select Parents	0.000010
CrossoverMethod	Create new childrens	0.000049
RepairSolution	Repair a children	0.017485
AugmentIndividual	Improve a solution by a LS	0.159890
InsertAndReplaces	Insert child in ordered array	0.000009

AugmentIndividual and *RepairSolution* are the most time expensive function and they executed for every new children created. The only improvement I found was to test if a children is equal to one of his parents, if it is none of this two function will be executed (Because the children is already a feasible and not improvable solution). I measured the time used by

AugmentIndividual and *RepairSolution* and the percent of time used in the alorithm was around 90% *AugmentIndividual* for and 7% for*RepairSolution*.

I used in a really stupid way the sparse matrix that's why it was so long. A sparse matrix here is used to represent the constraint, the basic datastructure of a sparse matrix in Julia is :

Variable	1	1	1	2	2	3	3	4	4	4
Constraint	2	3	5	3	4	1	2	1	4	5
Values	1	1	1	1	1	1	1	1	1	1

It have to be read by column, the first(1) variable is present in the constraint number 2,3 and 5 with values 1, 1 and 1.Instead of handling an N*M matrix you can use this representation. It often use less space and allow more efficient algorithm to be designed. It is perfect for the bin packing problem since the constraint contains a few elements by constraints.

At the end, I did change my function and now the balance change from 90% to 7 % for the *AugmentIndividual*. And now *RepairSolution* used around 90% of the time but it does not use more time than before.

It is now way faster. Here is the mean time for each problem used in example :

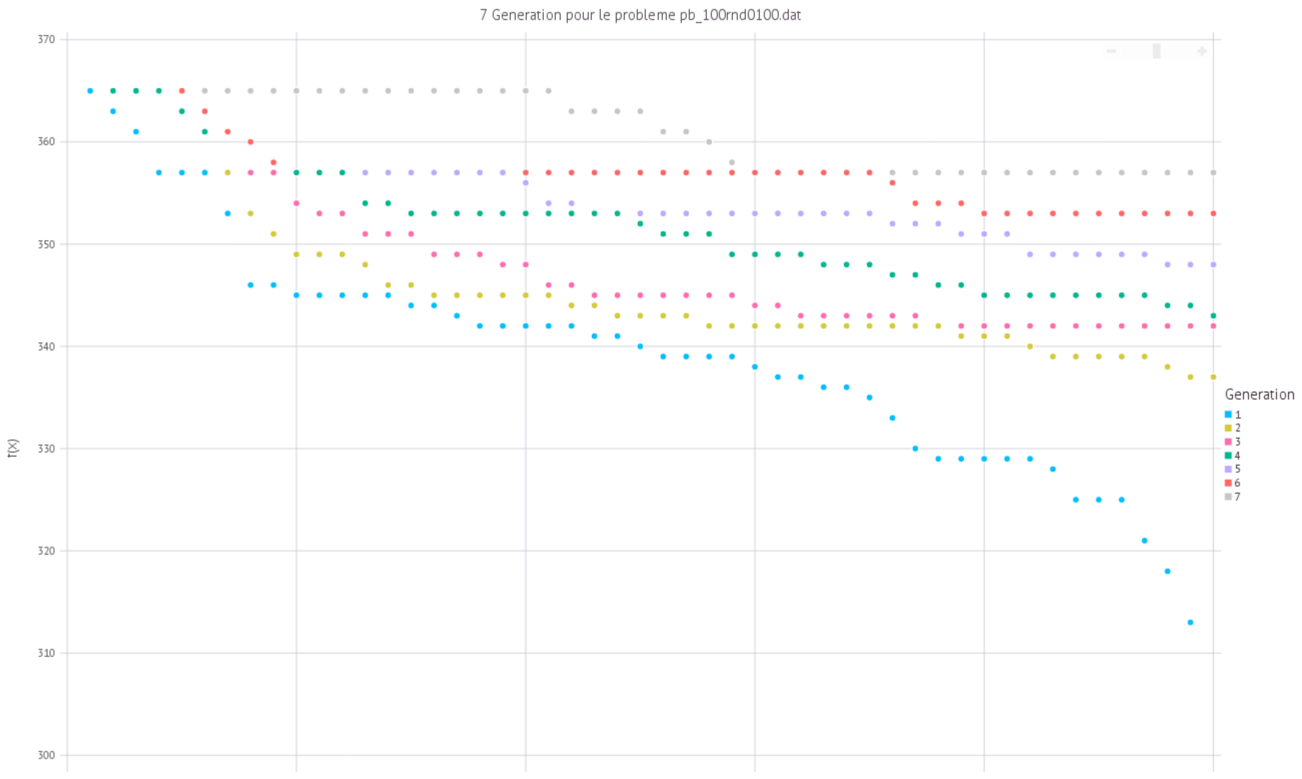
Problem	Genetic algorithm				
	Parameter		Time		
	Population	Limit	Min	Avg	Max
100rnd0100.dat	50	0.01	0.10	0.11	0.13
100rnd0100.dat	50	0.05	0.08	0.09	0.09
100rnd0100.dat	100	0.01	0.15	0.16	0.18
100rnd0100.dat	100	0.05	0.13	0.15	0.16
200rnd0300.dat	50	0.01	0.45	0.5	0.57
200rnd0300.dat	50	0.05	0.35	0.35	0.35
200rnd0300.dat	100	0.01	0.87	1.04	1.20
200rnd0300.dat	100	0.05	0.41	0.49	0.53
500rnd0100.dat	50	0.01	1.19	1.64	1.85
500rnd0100.dat	50	0.05	1.02	1.14	1.20
500rnd0100.dat	100	0.01	3.90	4.1	4.49
500rnd0100.dat	100	0.05	2.99	3.78	4.85
1000rnd0300.dat	50	0.01	6.35	7.94	9.09
1000rnd0300.dat	50	0.05	3.44	4.09	5.21
1000rnd0300.dat	100	0.01	18.37	26.03	34.04
1000rnd0300.dat	100	0.05	14.48	19.22	21.54

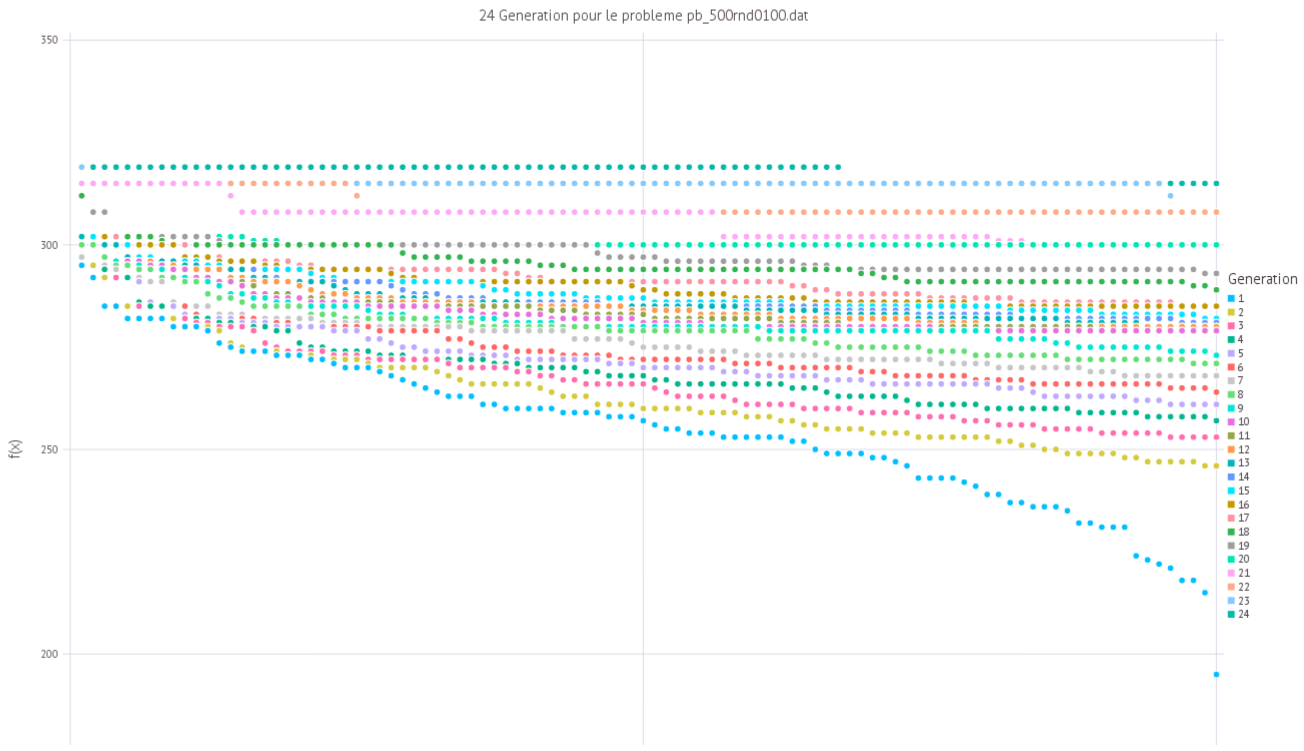
So i am pretty proud now, even if I still have some idea on how to improve different function. The average solution found are the same, just the execution time have been reduced. I also improved *RepairSolution* to use efficiently sparse matrix. It now run in less than three second for a problem with two thousand variables.

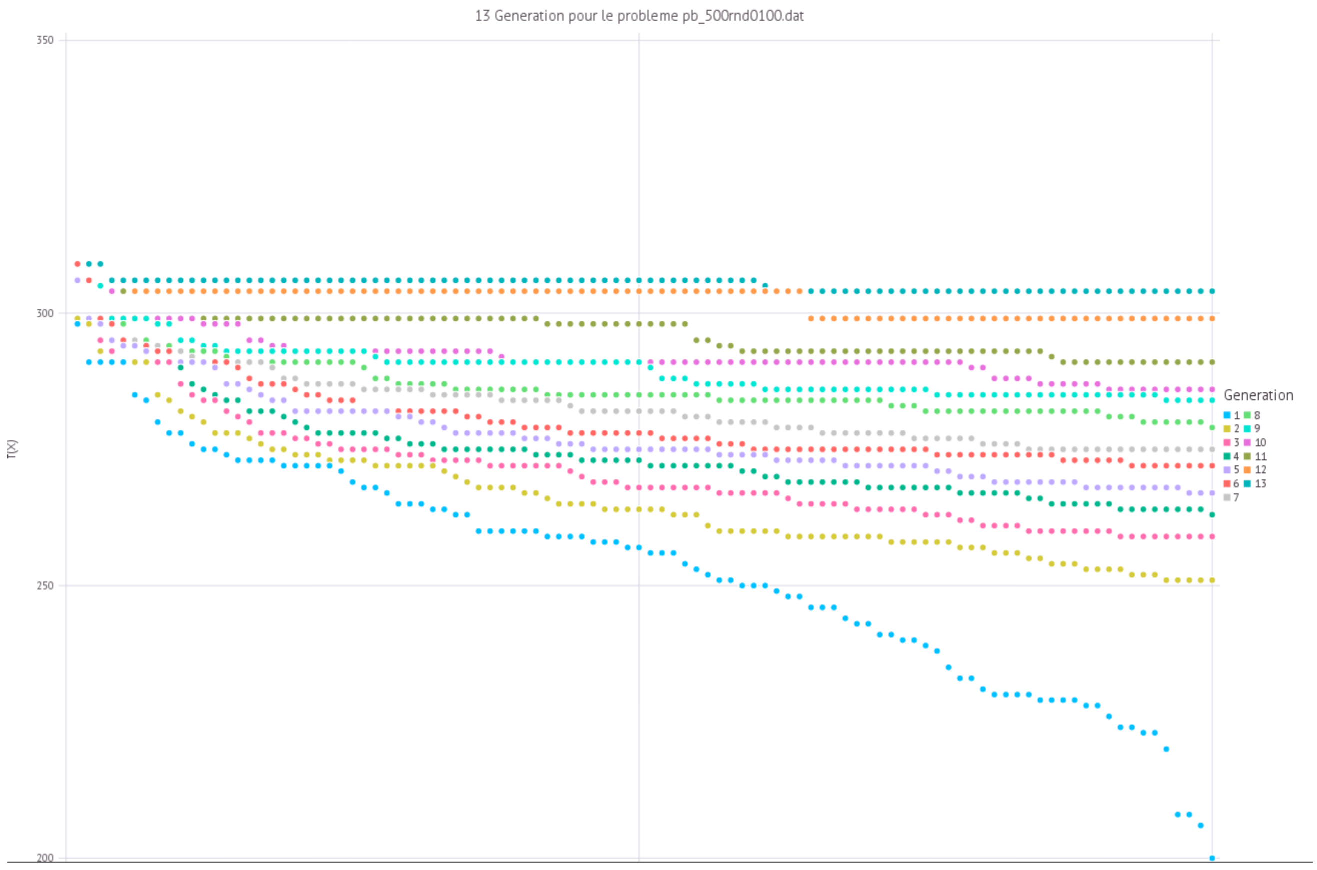
Now the weak point is the GRASP construction that is the slowest part of the program.

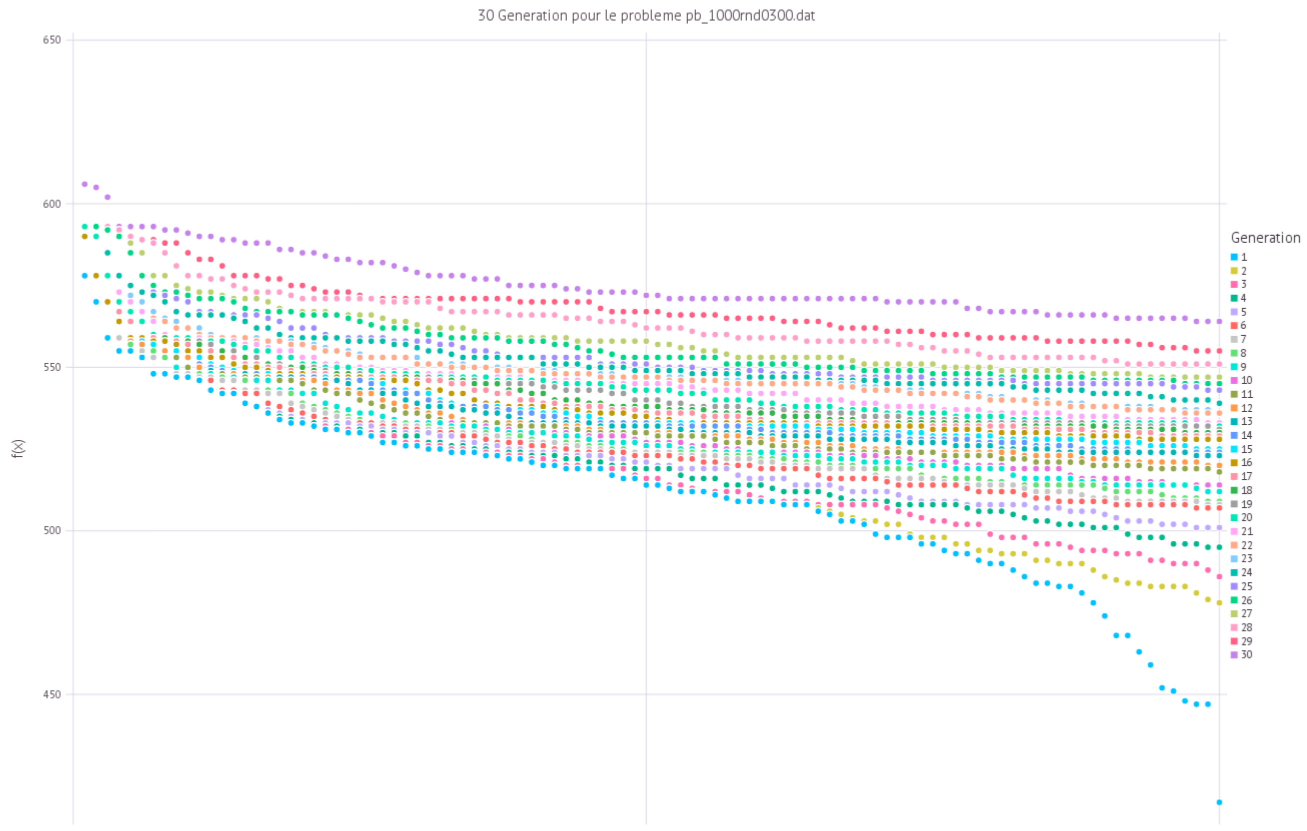
3.4 Interpreting the results

Here are some plot of my GA :









The decreasing f(x) is coming from the fact that the individual are sort in decreasing order by there objective value.

We can see in the second plot that the generation 24 have nearly all here individual at the same level. They probably converge on the same solution.

As generation increase we see the worst value disappear and the diversity of our population decreasing.

We can also noticed on the 3rd and 4th plot that some individual are over the "convergence" bar. The GA may have stopped too soon because it could have used this individual to maybe reach a better solution.

Adding a mutation could be interesting, it will widen the search space.

4 Conclusion

This last homework was interesting thanks to the multiple strategy you could use in a GA and the freedom it offer. I would have like to introduce some other function, like the random add of a new individual or a real mutation. But it didn't looked necessary on the kind of problem we were running our algorithm.

Different problem allowing more complex strategy could be fun and may be learn some tricks that are applicable to some specific type of problem could be interesting.

I've tried to implement and use sparse matrix in the GA because with my previous work it was too slow. I did it successfully but there is still the construction algorithm based on GRASP that use my previous work. I began to work on a construction algorithm made specially for the GA and independent of all my previous work but I didn't had the time to finish it.

I've also been using Julia embedded function as much as possible to make the program faster, but it happen once that a Julia function was slower than my code(Because of a stop condition).

But I actually failed to smartly use sparse matrix, I didn't know how to get through them to search a precise index or variable so I used a function called *find* and *findnext* but I noticed later than more than 90% of the execution time was used by these function in the *AugmentIndividual* function.

I was afraid to use VNS or Path-Relinking as a LS because it is easy, fast and does not need tons of new data to be used.However as it could have lowered the diversity of the population I decided to not.

I don't know if it was possible to just use a crossover and mutation in my GA, as it is a highly constraint problem it looked difficult to rely only on these two method to get a good solution.

Thanks to you I could spend a couple hours more on my GA, and it's now faster than it have ever been.(And correct)