Programming project – INF431

Evolutionary algorithms

***

Sébastien DUBOIS- Florian FEPPON

Table des matières

Introduction

This project deals with evolutionary algorithms which are a kind of optimization algorithms inspired by biological evolution. The potential solutions are here individuals that evolve: they go through mutations, crossovers and even a selection step. The selection is done by a fitness function that determinates the quality of the potential solutions.
More precisely, the goal of this project is to on a specific evolutionary algorithm (Algorithm 1), as described in the subject:

Algorithm 1: The $(1+(\lambda,\lambda))$ GA with offspring population size λ, mutation probability p, and crossover probability c. Mutation here means first generating a number ℓ according to a binomial distribution with parameters n and p and then generating each offspring independently by flipping exactly ℓ bits in the parent individual. Crossover consists of taking each bit independently with probability c from the second argument x', otherwise from the first argument x. Experience showed that $p = \lambda/n$ and $c = 1/\lambda$ lead to good results.

Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and compute $f(x)$;
Optimization: **for** $t = 1, 2, 3, \ldots$ **do**
  Mutation phase: Sample ℓ from $\mathrm{Bin}(n, p)$;
  for $i = 1, \ldots, \lambda$ **do**
    Sample $x^{(i)} \leftarrow \mathrm{mut}_\ell(x)$ and compute $f(x^{(i)})$;
  Choose $x' \in \{x^{(1)}, \ldots, x^{(\lambda)}\}$ randomly with $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\lambda)})\}$;
  Crossover phase: **for** $i = 1, \ldots, \lambda$ **do**
    Sample $y^{(i)} \leftarrow \mathrm{cross}_c(x, x')$ and compute $f(y^{(i)})$;
  Choose $y \in \{y^{(1)}, \ldots, y^{(\lambda)}\}$ randomly with $f(y) = \max\{f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$;
  Selection step: **if** $f(y) \geq f(x)$ **then** $x \leftarrow y$;

We decline our work in three parts. First we explain our choices regarding the implementation of genetic algorithms. In particular, we attempted to obtain a sufficiently modular code to use the same classes to perform optimizations regardless the nature of the arguments of the fitness function (implementation of optimizer objects).
Then we detail how we applied this generic organization to solve different problems and how we implemented the objects on which we do the optimization.
Finally we present our results and comments about the self-adaptive strategy.

I. Genetic algorithms implementation and organization of the code

Regarding the fact one iteration of the genetic algorithm does not depend on the objects whose fitness function is optimized, we have chosen to implement a global interface for the genetic algorithm.

The organization of the code is the following:

Package Optimizables: all the objects on which the genetic algorithm 1+(λ, λ) GA (self-adaptative or not) should be able to apply. These objects typically dispose of a fitness evaluation function, which can mutate and be crossed over with another object of the same class. We described this specification inside the Optimizable abstract class.

```
package Optimizables;
public abstract class Optimizable {
	public Optimizable mutation(double lambda){return null;}//Perform mutation phase
	public Optimizable crossover(Optimizable xprim, double lambda){return null;};//Perform crossover phase
	public abstract int fitness();//Compute the fitness of the function
	abstract public int lastvalue();//return the last value computed by the fitness function
	abstract public void println();//display the Optimizable
	abstract public void init();//initialization
}
```

Note that we did not manage to specify that the crossover function should apply on some fixed subclass of Optimizable and return the same subclass. Optimizable Subclasses have to check this specification themselves.

Thus this class should dispose of initialization, mutation, crossover and fitness method. Note that in order to dispose of a good adaptability of the code, the mutation and crossover methods compute the final values of mutation and crossover phase (without modifying the object calling the method). Thus, when we apply 1+(λ,λ)GA to binary numbers as described in **Algorithm 1** in the subject, the mutation method will compute l given the value of λ, then perform λ elementary mutations of parameter l, and finally return the best candidate among the new population.
Thus we impose subclass of Optimizable implement the elementary mutations and the choice of the parameters of the probability laws involved in the genetic steps.

Note that we create two additional methods : a *lastvalue()* method which should ease the access to the last value computed by the fitness function without doing once again the computation, and a pretty-printing function *println()*.

Given this generic class Optimizable we implemented the Optimizer class which contains the generic behavior described by **Algorithm 1.** The current value computed by the Optimizer is saved as *current_value* and the best candidate computed since the first iteration is *x*. One iteration of the **Algorithm 1** is contained in the method *one_iteration()*.

We have added a counter in the Optimizer class which counts the number of call to the fitness functions according to the article [1]: mutation phases and crossover phases induce respectively λ calls and the fitness is computed each time a truly better candidate has been found.

Implementing 1+(λ,λ)GA (GAOptimizer) and the self-adaptive version (SAOptimizer) of the algorithm has been done easily by inheriting from the Optimizer class (see GAOptimizer.java and SAOptimizer.java). GAOptimizer just changes the value of λ and SAOptimizer overload the selection method in order to change the value of λ at the end of iteration.

It is useful to notice the importance of changing the potential candidate *x* even if the current candidate *y* does have the same fitness than *x* at the end of an iteration: it will help the algorithm to explore more possibilities and thus save computation time (it justifies that *x*<-*y* if fitness(*y*)>=fitness(*x*) and not if fitness(*y*)>fitness(*x*) at the end of the selection process).

The Self-adaptive constructor can take different arguments: given any instance *x* of the Optimizable class :

```
SAOptimizer(Optimizable x);//SAOptimizer with no boundary for lambda and F=1.5 and  a=4 and lambda is multiplied by F^(1/a) at each iteration.
SAOptimizer(Optimizable x,int a);//SAOptimizer with no boundary for lambda and F=1.5 and  lambda is multiplied by F^(1/a) at each iteration.
SAOptimizer o=new SAOptimizer(x,int a,int max);//SAOptimizer with the boundary max for lambda and F=1.5 and lambda is multiplied by F^(1/a) at each iteration.
```

Thanks to this modularity, it is easy to test different optimization algorithms on a given Optimizable object *x*. Given an Optimizable object *x*, we can perform *n* iterations of the **Algorithm1** by writing:

```
//Choice of the optimization strategy
//SAOptimizer o= new SAOptimizer(x,3,n);//Selfadaptive with bounded population
//GAOptimizer o= new GAOptimizer(x,4);//Genetic Algorithm with lambda=4
Optimizer o=new Optimizer(x);//Simple EA algorithm
for(int i=0; I < n ; i++){
        o.one_iteration();
}
```

Given these classes, we have implemented different Optimizable class and tested the algorithm on different problems.

II Optimization problems implementation

1. *OneMax function*

We intend to optimize here the function counting the number of bytes of an arbitrary binary number of a given size *n*. The optimal solution is obviously the binary number of size *n* having *n* ones.
In order to implement this problem, we created a class BinaryNumber extending the Optimizable class (see BinaryNumber.java) and thus overriding fitness, mutation and crossover function. In order to ease the adaptability and its own inheritance, we created a method *copy()* which is needed by the mutation and crossover function. The methods *mutate(l)* and *cross(c,xprim)* compute one elementary mutation or crossover. The methods mutation and crossover computes λ candidate and the parameters l and c according to the value given in the subject. The bytes of the BinaryNumber are saved in an array named *x*.

It is easy then to proceed to the optimization. The following code applied the 1+(4,4)GA algorithm and count the number of fitness evaluations in order to obtain the optimal solution.

```
BinaryNumber x= new BinaryNumber(n);
GAOptimizer o= new GAOptimizer(x,4);
o.counter=0;
o.init();
while (o.current_fitness()!=n){
        o.one_iteration();
}
System.out.println(o.counter);
```

If we wanted to test the self-adaptive strategy, we would just have to change the second line in:
`SAOptimizer o= new SAOptimizer(x);`

2. Maximum Matchings

We intend here to find a maximum number of vertex matchings (set of pairwise disjoint set of edges) in a given undirected graph.
In order to implement graphs and being able to display them graphically, we got inspired from the TD9 (Skyscrapers, see [2]) which gave an implementation of directed graphs and tools to display them. The provided graph class saves vertices in a list and edges in a LinkedHashMap mapping vertices (integers) to vertices.

We updated this class by adding a field LinkedHashMap *edgesMap* which maps vertices to their depending edges (represented by a unique integer id *i* lower than the number of edges) and a field LinkedHashMap *edges* which maps integer id *i* to edges. Edges are represented by a pair of integer (*a*,*b*) when *a* and *b* are two connected vertices, and by convention, *a* is always lower than *b* (the graph is undirected, so that pairs have a unique representation).
We also provide a method `public static Graph randUndirectedGraph(int n)` to generate random graphs according to the procedure described in paragraph 2.2.2. of the subject (each vertex finds three distinct "friends" and then we match all "friends" together) and `public static Graph ring(int n)` which generates the ring with *n* edges.

A set of edges is then represented by a binary number with the bytes *i* equals to 1 if the set contains the edge *i*. In order to do this properly, we create the class BinaryNumberMatching which inherits the class BinaryNumber. The methods mutation and crossover are not changed.
We add a field `Graph g` pointing to the graph of interest. Then we create a method *degree(Integer v)* computing the number of edges depending on a vertex *v* inside the subgraph constituted only by the edges given in the set represented by the binary number.

It is easy then to compute the fitness according to the definition given in the subject:

```
public int fitness() {
        int val=0;
        for(Integer v:g.allVertices()){
                int temp=degree(v);
                if(temp>=1){
                        val+=temp-1;
                }
        }
        lastvalue = numberOfOnes-x.length*val;
        return lastvalue;
}
```

Then we can proceed to the optimization by the very same way as the OneMax function: see the *randgraphtest()* code in Projet.java.

We also provide BinaryNumberMatching with a method `public void displayWindow()` which shows the graph *g* with the set of edges represented by the BinaryNumberMatching enlightened.Then we can display the result of an optimization by an Optimizer *o* by `((BinaryNumberMatching) o.x).displayWindow();`

3. Eulerian Cycles

As recommended in the subject, we encoded this problem considering pairings among the edges incident with a vertex.
We decided to represent individuals using a matrix int[i][j] where *i* is the number of vertices and *j* the number of edges. This matrix is the field pairing of the object Couplage.

```
public class Couplage extends Optimizable {
  int nbV ;  //number of vertices
  int nbE ;  // number of edges
  private int lastvalue;
  UndirectedGraph g;
  int[][] pairing ;
…
}
```

Precisely:
- pairing[v][e] equals -1 if and only if edge *e* is not incident with vertex *v*
- pairing[v][e] equals *e* means edge *e* is not matched yet (the matching is not perfect)
- pairing[v][e] equals *f* means (*e*,*f*) is a pairing for vertex *v*.

So with this representation the matching for vertex *v* is approximately the column *pairing[v]* (we should not take care of -1s).

As said in the subject, we do not want to solve half of the problem before evolutionary algorithm has started, so at the beginning pairing[*v*][*e*] equals -1 or *e* - it means there is not any pairing.

For the fitness function, we decided to consider the length of the cycles. Because taking care of cycles is relevant - or even possible- only if we have a perfect matching, we need two steps to compute the fitness. Firstly we check if every edge is matched with another edge - this means pairing[*v*][*e*] == *e* for none of edges and vertices, and means *numberOfEdgesAlone()* equals zero . If not, we do not look at the cycles and we give a penalty for each edge which is not matched. If all the edges are matched, the matching is perfect and we can compute the lengths $\ell_1, \ldots, \ell_k$ of the *k* cycles induced by pairing and we take $f := \text{sum}(\ell^2)$ (because $\text{sum}(\ell)$ = number of edges regardless of the number of cycles).

Finally, we created a method **public void** *makeEulerian()* which simply adds edges to a random graph so that every vertex has even degree.

4. SAT Problem

We got interested then in attempting to test these optimization algorithms to NP hard problems and apply 1+(λ,λ) in situations where no efficient and deterministic algorithm already exists.

We consider formula given in their conjunctive normal form. We represent them with three classes (implemented in the package SAT): Clause, Formule, Litteral (see Clause.java, Formule.java, Litteral.java). A Formule contains a set (HashSet) of objects of class Clause. A Clause contains a set of objects of class Litteral. And a litteral is represented by an index *index* and a boolean *negation* which represent the literal x_{index} if *negation* is false and $\neg x_{\text{index}}$ if *negation* is true.

We provided the Formule class with two methods generating random formulas :
public static Formule generate(**int** numberOfLitterals,**int** sizeOfClauses,**int** numberOfClauses);//generate random formula
public static Formule generate2(**int** numberOfLitterals,**int** sizeOfClauses,**int** numberOfClauses);//generate random satisfiable formulas. Can produce more litterals than numberOfLitterals as litteral *i* is always present in clause *i*

Note that these two methods produce Formula with correctly indexed literals (if there are n literals, then it is ensured that literals are indexed from 0 to n-1). This property can be ensured to any formula by the function rename of the Formule class. generate2 can create

A valuation is represented by an object of class Valuation (a LinkedHashMap mapping integer to boolean values) or simply by an array int[] *x* such that the litteral x_i is set to be true if and only if *x*[i] exists and is equals to 1. A valuation x satisfies a formula if x maps any literal to a boolean value and if all the clauses are satisfied. A clause is satisfied if one of its literals is satisfied.

The satisfiability problem is to determine whether a given formula can be satisfied or not. We decided here to consider this problem as an optimization problem : Optimizable objects are valuations (which can easily be represented by BinaryNumbers). The fitness function simply counts the number of Clauses that are satisfied.
We can count them by the method **public int** numberOfSatisfiedClauses(**int**[] v); given in the Formule class.

We created a BinaryNumberValuation class which extends BinaryNumber. Its attribute int[] *x* represents a valuation as said in the previous paragraphs. Note that this class needs correctly indexed Formulas to avoid working on partial valuations and getting the exception “Erreur: la valuation est partielle”.

5. Hamiltonian Cycles

Another NP problem which caught our interest is the problem of finding Hamiltonian cycles in a graph. Our first motivation was the travelling salesman problem but it seemed to us that it would be very difficult to know when we should stop the algorithm and that working on Hamiltonian cycles was a nice start.
We can remind that a Hamiltonian cycle in a graph is a cyclic path that visits each vertex exactly once. Although this problem seems to be close to the Eulerian cycle problem, determining whether such a path exists in a graph is a NP-hard problem.

Inspired by the representation recommended in the subject for the Eulerian cycle problem, we decided to implement this problem by considering pairings of edges. In contrary to the Eulerian cycle problem, there is here only one pairing for each vertex, which is stored in *pairing*[*vertex*].
We created a class CouplageH which inherits from the class Optimzable:

```
public class CouplageH extends Optimizable {
  Graph g ;
  int nbV ;  //number of vertices
  int nbE ;  // number of edges
  private int lastvalue ;
  IntegerPair[] pairing ;
…
}
```

As said before, a pairing is completely described by the knowledge of exactly two edges for each vertex *v*. In our representation, this pairing is pairing[*v*].
At the beginning of the algorithm, pairing[*v*] equals (-1,-1) for each vertex *v*.
An elementary mutation consists of the following steps : *(i)* Choose a random edge *e*; *(ii)* Choose randomly one of its two vertices ; *(iii)* insert e in *v*'s pairing : if the pairing is not complete (this means at least one of the two integers in pairing[*v*] equals -1), replace -1 with *e* ; otherwise choose randomly the one to replace. As in the Eulerian cycle problem, a full mutation step is the result of l elementary mutations, where l is obtained from taking a Poisson distribution expecting λ and then adding one.
For the crossover operator, we simply copy the couple of edges matched for each vertex. Again, the offspring $cross_c(x,x')$ take a matching from x with probability 1-c and from x' with probability c.

In contrary to Eulerian cycles, we do not have a simple condition to know if a graph is Hamiltonian or not. So we decided to use special graphs as test instances, which we know they have a Hamiltonian cycle. These special graphs are simply built as random graph beyond a ring (the ring is obviously a Hamiltonian cycle).

The last thing we had to choose was how to compute the fitness.
We thought it was natural to consider the paths in the graph which were possible with the pairings given. Thus, we compute the fitness by looking for the longest path and by using only the vertices which have a complete pairing (without *-1*).
We also thought about another approach. As a matter of fact, we can highlight that if all pairings are complete and if the number of edges used equals to the number of vertices, this matching gives a union of cycles. Hence this is a Hamiltonian cycle if and only if there is just one cycle. However we will see in the results this approach was less efficient.

III. Results

1. OneMax functions

We reproduced the experiments given in the article [1]. We tested different evolutionary algorithms for BinaryNumber of size *n* from 50 to 1000 with a step equals to 50. We esteemed the number of fitness evaluation needed to obtain the optimum by computing the average number of evaluations over 100 tests for each *n*. We observe clearly the decrease in evaluations of fitness function while passing from the 1+1EA algorithm to the 1+(λ,λ)GA algorithm, and from the 1+(λ,λ)GA algorithm to the self-adaptive version. Regarding time evaluation, it appears that the self-adaptive version seems to induce some extra cost in time (due to more variables manipulations): the most efficient algorithm in time seems to remain the 1+(8,8)GA among the tested strategies. Time evaluation remains quite difficult and is not really reliable to produce results with weak variance.

Results are shown in annex.

2. Stable Matchings

The different algorithms give coherent results for low value of n, the number of edges of the graph. We tested it on the ring problem, where we know explicitly the best solution (take edges alternatively around the ring).

We are compelled to consider the cases where *n* is odd and where *n* is even separately. Indeed, the number of configurations inducing the optimum is bigger in the odd case (two arbitrary adjacent edges must not be in the optimum set): the number of fitness evaluations needed to solve the odd case is then significantly lower (by a constant ratio) than to solve the even case.

Then we do optimizations for increasing values of *n*, from 2 to 40. Surprisingly we get no good result with the adaptive or genetic algorithm. The 1+1EA (no crossover phase, λ=1) method remains the best among the previous algorithms tested. By plotting the curves obtained and doing a regression we find that the complexity seems to be in $O(x^4)$ as predicted in the subject.
It seems that the crossover chosen does not help to find truly better solution (at least for the ring problem): very good solutions does not really resemble to good solutions.

3. Eulerian cycles

For our tests, we start with a random graph with n vertices and then we modify it – if necessary – so that each of its vertices has even degree. So the exact graph on which the algorithm works does not necessarily have exactly *n* vertices. Moreover the crucial parameter for the Eulerian cycle problem is the number of edges and not really the number of vertices.
For each *n* given, we compute 10 tests on 10 different graphs and return the average. Thus, the little incoherence highlighted above does not really appear in the results.

As recommended in the subject, we first made tests with a Poisson distribution with expectation one (and then adding one) for the mutation step.
Here again we sorry to see that the 1+1EA remains better than the 1+(λ,λ) GA and the self-adaptive one. On the other hand, we did not see a real difference in the results given by genetic algorithm and the self-adaptive one.

However we tried to compute the mutation step by taking a Poisson distribution with expectation λ (and then adding one) and there results were completely different.
Firstly the genetic algorithm becomes more efficient than with the previous implementation, but most of all it becomes more efficient than the 1+1 EA. Among the genetic algorithms, it seems that it becomes more and more efficient when λ increases. Finally the self-adaptive algorithm gives the best results.

Despite everything, all the algorithms described here give - approximately - linear results (when counting the number of fitness evaluations), which is already good. Of course, we can just judge from tests we managed to run (*n* lower than 250) and so we cannot guarantee a linear complexity for big values.

As we start the algorithm with zero matching, it seems that much of the algorithm consists in doing mutations in order to obtain a perfect matching. That is what could explain why taking a Poisson distribution with expectation λ reduce the number of iterations. But in terms of complexity, that does not make the algorithm really better.

Finally, we tested a different fitness function which computes the maximum of the cycles' length (whereas the first fitness function computes the sum) and we did not see a real difference on the results, for any of the algorithms tested (EA, GA and self-adaptive).

4. Hamiltonian cycles

As said before, we tested our algorithms by starting with a random graph built on a ring. All results in the annex are the average on 10 tests made on 10 different graphs for *n* given.

Firstly, we can discuss about the fitness function.
Our first approach was to consider the length of paths in the graph (using only selected edges). As with the Eulerian cycle problem we computed the fitness function on the one hand by taking care of the longest path and on the other hand by computing the sum of paths' length squares. In contrary to the Eulerian cycle problem, we observed a huge difference in the results: the first solution is far better.
Our second approach is the one described in the first part: the fitness function is approximately the opposite of the number of edges used for the matching, plus a bonus when there is only one cycle – which means we have found a Hamiltonian cycle. Results can be compared to those obtained with the fitness function which computes the sum of paths' length squares.
According to this, we kept the best fitness function and made all the detailed tests in annex with it.

Regarding the expectation we take for the Poisson distribution in an elementary mutation, it seems that it does not make much difference, but despite everything we could assume that taking an expectation of one (and then adding one) is a little bit better.

The comparison we made of the different algorithms - 1+1EA, 1+(λ,λ)GA self-adaptive or not- let us conclude that 1+1 EA remains the best one. The 1+1 EA gives a complexity which seems to increase like n^k ($3 < k < 4$) in terms of fitness evaluations, where *n* is the number of vertices, as shown in annex. Considering that the Hamiltonian cycle problem is NP-hard and so, that we could only build exponential determinist algorithms (if $P \neq NP$) to find Hamiltonian cycles, these results can be taken positively.
Between 1+(λ,λ) GA and the self-adaptive one, it seems that the first one wins and that it becomes less and less efficient when λ increases. Unfortunately we did not manage to perform many tests as they took very much time, even for low values of *n*.

5. SAT Problem

Testing SAT problem is rather difficult as these algorithms will not possibly tell us if a given formula is satisfiable. We considered formula with three distinct literals by clause. We could count the number of iterations needed to satisfy 7/8 of the clauses, as we know that this is reachable for any formula.
Nevertheless it seems this number is constant after performing tests (this is related to the fact that 7/8 is the average of the number of satisfied clauses for a uniformly distributed random valuation.
The difficulty lies also in the fact that a SAT problem can be described by three distinct parameters: length of a clause, number of clauses, number of variables.

We decided to perform tests on formula with *n* literals and *n* clauses of length 3: we ensure that the *i*-th contains the variable *i*, so that the formula is automatically satisfiable.
Once again, the 1+1EA strategy seems surprisingly to be the best: the crossover phase does not bring any help.

Conclusion

It appears that evolutionary strategies seem to be interesting provided some conditions are gathered:

- The problem is discrete. In the contrary case continuity properties should help to find faster algorithms.

- We dispose of a criterion to stop the iterations: stop when a satisfying and reachable fitness has been found. In particular we do not seem to have any convergence property on the individuals tested.

- We have some discrete “continuity” property: genotype must have an influence on the phenotype, that is to say individuals with optimal fitness evaluation should resemble to individuals with good fitness evaluation. It imposes to define a good structure to implement the problem: appropriate mutations, crossover and representation of individuals. This seems to be crucial regarding the self-adaptive strategy: we obtained good result with the self-adaptive strategy for the OneMax problems and for the problems with intelligent graph representation.

- The choice of the fitness function must be intelligent and appropriate to the problem considered as shown in the Hamiltonian cycle problem.

References

[1] B. Doerr, C. Doerr, and F. Ebel, “Lessons from the black-box: Fast crossover-based genetic algorithms” in Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2013. ACM, 2013, pp. 781{788.

[2] https://www.enseignement.polytechnique.fr/informatique/INF431/X12-2013-2014/TD/TD9/INF431-td_9-1.php

User guide

The main function is written in Projet.java and allow to launch several optimization problems by commenting or uncommenting.

nonselfadaptivetest() - Test One max function with GA(lambda,lambda) algorithm for several lambda and n. Averages over 100 tests are done.

selfadaptivetest() - The same with the self-adaptive strategy (SA algorithm), with several values for n where the population increases by 1.5 at each truly better solution found.

randgraphtest() - generates a random undirected graph of size 5 and perform 100 iterations of the SA algorithm over the maximum matching problem and then display the graph and the selected matchings.

ringtest() - performs tests over the maximum matching problem with ring of size n and compute the average number of evaluation.

eulerianCycleTest() - generates a random Eulerian graph and compute the average number of fitness evaluations for several sizes, regarding the problem of finding an Eulerian cycle.

hamiltonienCycleTest() - generates a random Hamiltonian graph and compute the average number of fitness evaluations for several sizes, regarding the problem of finding a Hamiltonian cycle.

satTest() - generates random solvable formulas with n clauses of size 3 and n literals and apply the algorithms to solve the satisfiability problem. Average number of fitness evaluation is displayed for several values of n.

Performing some optimization :

Define an object x which inherits the **abstract** Optimizable **class**.
For example :
BinaryNumber x=**new** BinaryNumber(10);

Define the Optimization strategy by giving the Optimizable object as a parameter :
For example :
SAOptimizer o=**new** SAOptimizer(x);//Self Adaptive algorithm with no boundary on the population
Optimizer o=**new** Optimizer(x);//1+1EA Algorithm
GAOptimizer o=**new** GAOptimizer(x,4);//1+(4,4)GA Algorithm

Initializing the algorithm :
o.init();//Define the first candidate

Perform iterations :
o.one_iteration();

Counting the number of fitness evaluation:
o.counter;

Getting the best found value after the last iteration and its individual :
o.current_value;
o.x;

Optimizable objects:

BinaryNumber x=**new** BinaryNumber(**int** n);//Creates a binary number of length n.
BinaryNumberMatching x=**new** BinaryNumberMatching(UndirectedGraph g);//Create a binary number representing a set of edge of the undirected graph g
BinaryNumberValuation x=**new** BinaryNumberValuation(Formule form);//Creates a binary number representing a valuation over the formula form. The litterals of the formula need to be correctly indexed (from 0 to n-1 where n is the number of variables). This can be achieved by the rename() method of the Formule class.

Couplage x=**new** Couplage(UndirectedGraph g);//creates a Couplage with zero matching, for Eulerian cycles
public int mesureCylce() ;// compute the sum of cycles' length square
public int mesureCylce2() ;// compute the length of the longest cycles

CouplageH x=**new** CouplageH(UndirectedGraph g);//creates a Couplage with zero pairing, for Hamiltonian cycles
public int lenghtPaths() ; // compute the sum of paths' length square, used by fitness1
public int lenghtPaths2() ; // compute the length of the longest path, used by fitness3

Other structures:

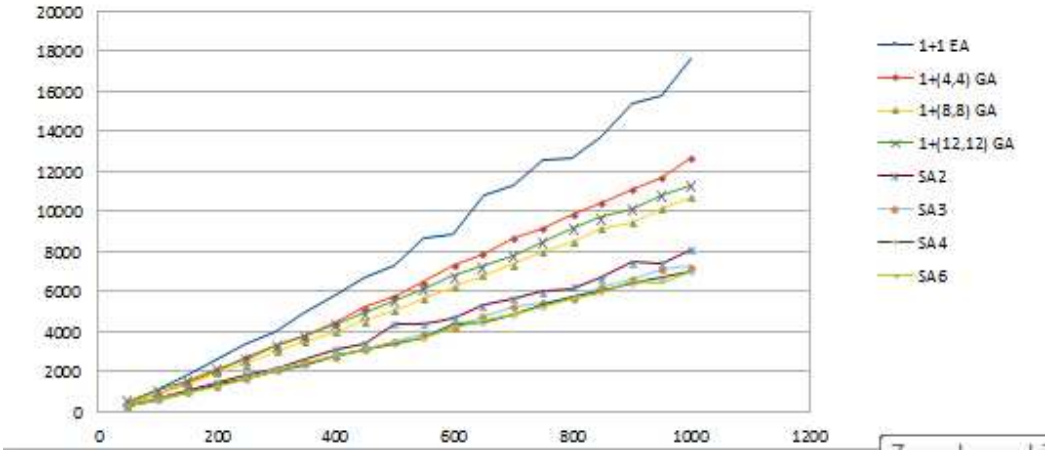
UndirectedGraph g = **new** UndirectedGraph(); Creates a **new** undirected graph with edge numbering. Each edge has a unique identifier to be handled by the BinaryNumberMatching **class**.
g.addVertex(i);//Creates the vertex i
g.addSuccessor(i,j);//Link i and j together (without orientation)
g.getEdgeNumbers(i);//Return the list of the edges adjacent to i. An edge is represented by an IntegerPair (couple of edges (a,b)).
g.getEdge(i);//Return the edge numbered i

```
Formule f=new Formule();//Creates an empty formula (Logical AND of several clauses)
Litteral l=new Litteral(int i,boolean b);//Creates the litteral x_i if b is false and non(x_i) if b is true. b is optional and of value false by default
Clause c=new Clause();//Create an empty clause (Logical OR of several litterals)
c.addLitteral(l);//Add the litteral l to the clause c
f.addClause(c);//Add the clause c to the formula f
f.evaluate(int[] v);//Attempts to evaluate the formula f with the valuation v. v is an array of integer with v[i]=1 if and only if the variable x_i in the formula f is set to true. f throws an exception when v is partial.
f.renameLitterals();//Rename all the litterals from 0 to n-1 where n is the number of variables
Formule f=Formule.generate(int numberOfLitterals,int sizeOfClauses,int numberOfClauses);
//Generate a random formula f with a determined number of litterals, clauses and size of clauses. It is not garanteed that f is satisfiable.
Formule f=Formule.generate2(int numberOfLitterals,int sizeOfClauses,int numberOfClauses);
//The same but the formula asserts that the clause i contains the variable xi. Thus litterals can be added and the formula is automatically satisfiable.
```

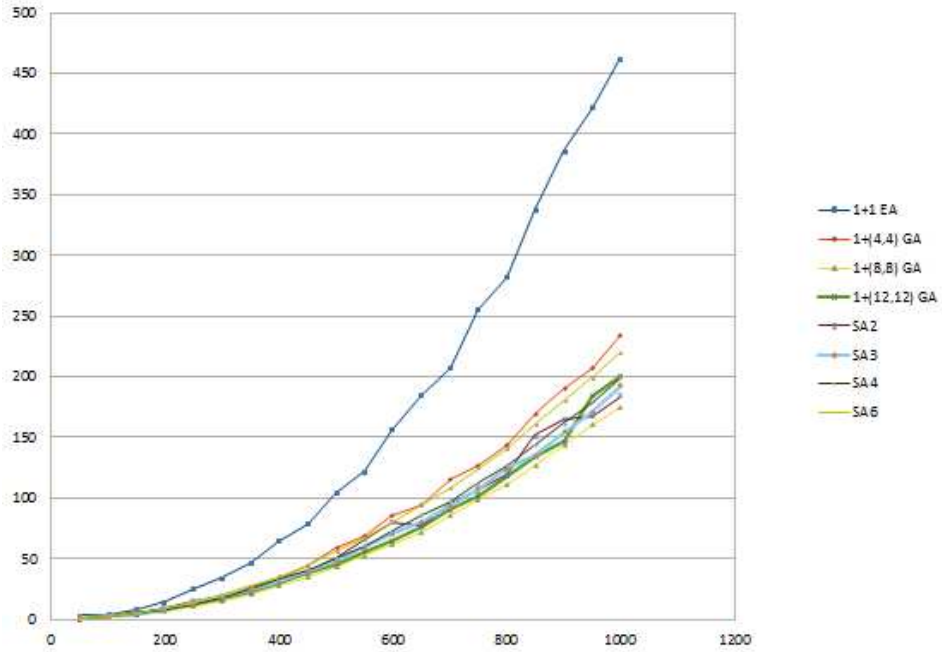
Annex

You can find all row data with java files (named projet_data1 and projet_data2).

OneMax

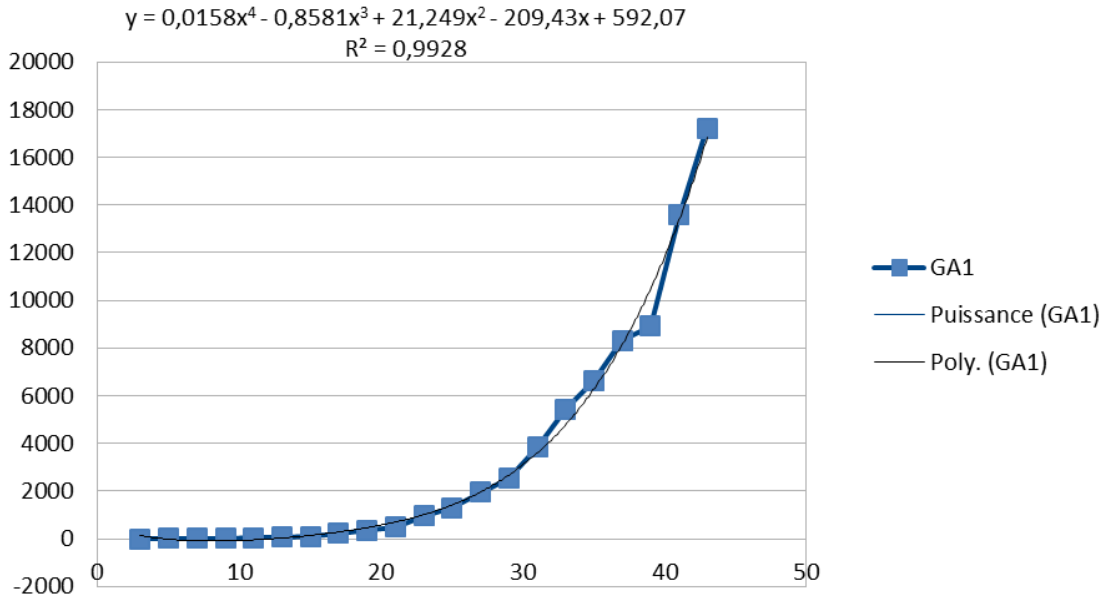


OneMax Number of fitness evaluation (100 tests for each)

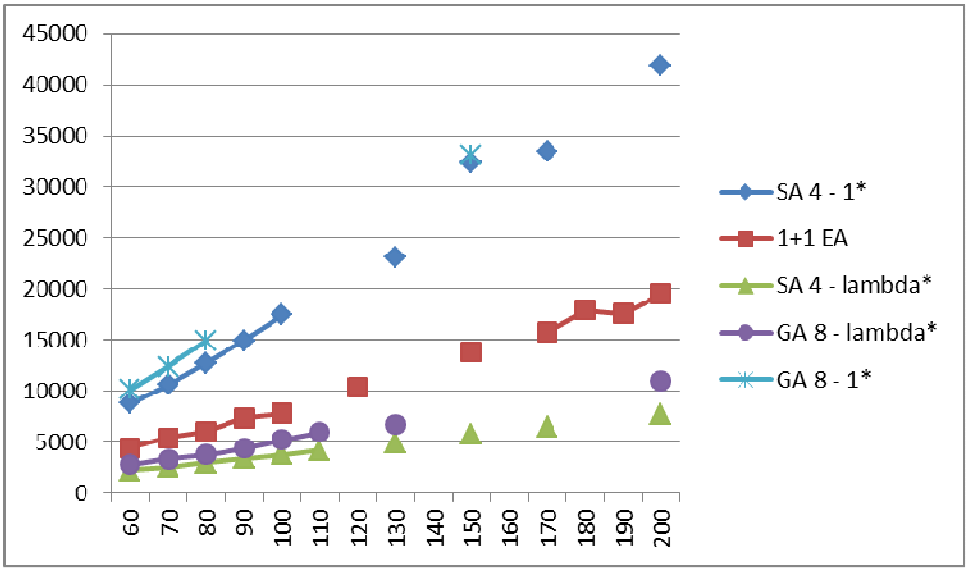


OneMax Time Evaluation (ms)

Maximum Matching



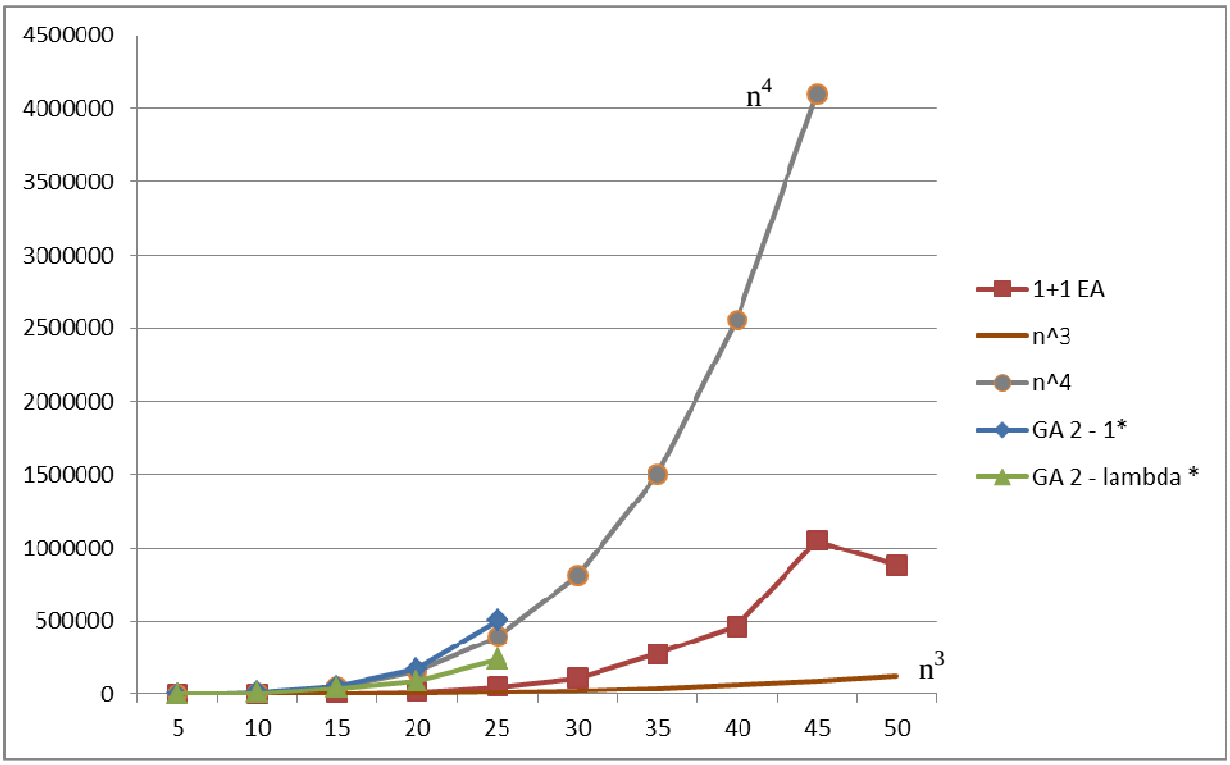
Eulerian cycles



* refers to the expectation of the Poisson distribution for the number of elementary mutations

Running time	n = 200									
1+1 EA		SA 4 -1*		SA 4 - lambda*		GA8 - lambda		GA16 - lambda*	
7113,55	ms	19879,68	ms	3017,38	ms	3565,95	ms	3506.57	ms
19508	counter	41919,95	counter	7671,3	counter	11023,74	counter	8877.09	counter

Hamiltonian cycles



n	1+1 EA	GA 2 - lambda*	GA 2 - 1*	SA 4 - lambda*	SA 4 - 1*
5	152,0775	627,84	982,3	530,72	2000
10	2446,255	10236,49	18512,58	348 701	3 815 331
15	9553,2375	44823,75	54250,71	explose	explose

Satisfiability

	SA4	SA3	GA8	GA4	GA1	GA12
N=200	2306,09	2368,4085	768,54615	655,07385	597,66231	900
N=300		3125	1271,164	1089,006	1027,756	1567,736