A Java Library for Genetic Algorithms Addressing Memory and Time Issues

Luigi Troiano
University of Sannio
Department of Engineering
RCOST – Viale Traiano
82100 Benevento – Italy
troiano@unisannio.it

Davide De Pasquale
University of Sannio
CISE Laboratory
RCOST – Viale Traiano
82100 Benevento – Italy
davide.depasquale@ciselab.org

PREPRINT

***Abstract*—In this paper we introduce a Java library for developing and testing genetic algorithms. The software architecture is aimed at addressing common issues regarding system memory and garbage collection in Java programming. In order to validate our solution, a comparison with other frameworks is provided.**

I. Introduction

Java is a very common language used for implementing genetic algorithms frameworks. However, this type of frameworks heavily use system memory and garbage collection, and this, jointly to Java Bytecode running on virtual machines, can seriously affect performances. Examples of frameworks are ECJ, JDEAL and JGAP. In general they perform worse than native solutions, such as GALib. Performances play a relevant role, as genetic algorithms are resource demanding in facing real world problems. Therefore in order to make Java solutions able to scale problem complexity, a major attention must be paid in devising the framework architecture and implementation, still keeping a highly reusable programming model as expected using a library.

In this paper we present JENES, "yet another" Java genetic algorithm framework, this time devoted to address memory and garbage collection issues. Main features for this purpose are: (i) reuse of objects instead of collecting them for garbage; (ii) parameterized classes in order to strengthen type checking and avoid the need of casting instances at run-time; (iii) fitness evaluated only when strictly required; (iv) finer access control of algorithm methods and events aimed at avoid unnecessary operations. In order to make the framework friendly to be specified and suitable for different problems, algorithm structure is highly configurable by means of interchangeable blocks.

The reminder of this paper is organized as follows: Section 2 outlines the architecture and programming model; Section 3 provides a brief overview of related works and solutions; in Section 4 experimental results are reported; Section 5 discusses conclusions and future directions.

II. Architecture and Programming Model

Programming JENES is easy and intuitive. Solutions in a search space are represented by *individuals*. They are made of a *chromosome* and *fitness value*. Chromosomes encode problem solutions. They are regarded as an array of *genes*. The default implementation provides the following basic types:

- `BitwiseChromosome`, composed by bits; genome contains values coded accorded to the specified bit coding
- `BooleanChromosome`, composed by boolean values; each gene can be true or false
- `DoubleChromosome`, composed by double values; each value in $[lowerBound, upperBound]$, specified at the instantiation time
- `IntegerChromosome`, composed by integer values; with values in $[lowerBound, upperBound]$, still specified at the instantiation time
- `ObjectChromosome`, represents a chromosome whose genes contain to `Objects` references
- `PermutationChromosome`, modelling permutations in problems such as the *travel-salesman problem*.

All classes above implement the `Chromosome` interface. By default these objects have a fixed number of genes, but it is possible to vary their length during the execution, in order to implement a length-variable solution coding. `Individuals` are typed with respect to a particular chromosome at compile-time. A `Population` is a typed collection of compatible individuals. An individual can be *legal* or not, according to the solution admissibility. By default, each individual is legal, but it is possible to invalidate it during execution.

The genetic algorithm is executed by invoking the method `evolve()`. JENES genetic algorithms run as depicted in Fig.1. The algorithm execution is split in three main phases: (i) *start* creates and scores the initial population, then initializes the algorithm breeding stages; (ii) *run* evolves the population applying the just initialized stages until a termination guard is verified; (iii) *stop* ends the algorithm, disposing the stages. A set of events are generated at different execution points. These events are captured by user defined *listeners* or algorithm callback methods.

The initial population is created by cloning a sample individual, the latter serving as *prototpye* [1] for a given number of copies. These copies are then randomly altered in order to get genetic diversity of the initial population. The *randomization rate* (by default 100%) defines the number of copies to be

978-1-4244-5612-3/09/$26.00 ©2009 IEEE

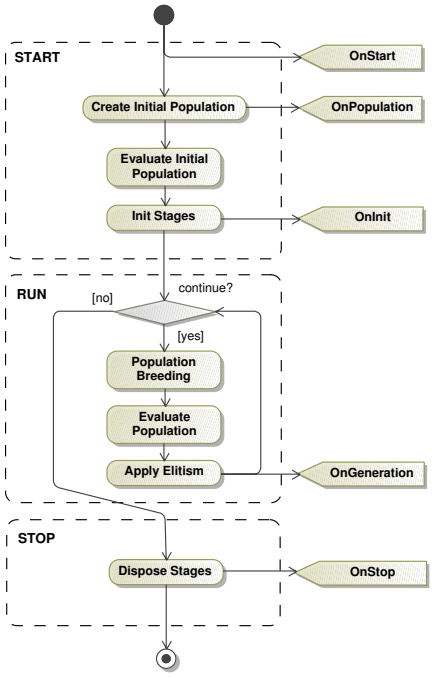

Fig. 1. Algorithm flow in JENES

altered.

Particular attention has been paid to code optimizations in JENES. The most important regards the memory usage. Objects are continuously recycled. This permits to reduce the overhead provided by the garbage collection. If the population size is fixed along the algorithm evolution, the memory occupation grows at the beginning and does not change after. This allows to reduce the overall software footprint and time for building and collecting objects. The chromosome footprint is reduced, such as in the case the `BitwiseChromosome` able to work at the level of single bits, thus outperforming other solutions. Moreover, individuals already evaluated are not re-evaluated in the following iterations. During the algorithm evolution, past populations are buffered in the algorithm's history. At each generation, the eldest population and its individuals are reused. Past populations, instead of being deallocated, are kept in memory and reused. This technique avoids to allocate in memory new populations, thus limiting the use of the garbage collector.

JENES data structures are strongly typed, so population and individuals can only work with compatible types. This allows to reduce the number of checks at runtime, as the use of parametric classes assures the correct data type at compile time. This also entails that there is no need of casting objects, resulting in better software reliability and speed. This feature is due to generics available since Java 1.5.

An algorithm in JENES is structured in composable components, termed `Stages`. The population passes through the stages and there it is transformed. Each stage receives an input population and produces an output population. It is important to consider that the output population is pre-initialized with recycled individuals taken from history. Therefore JENES does not allocate new `Individuals` in memory if population size does not change, but only set the *Chromosome* of available elements. The reference to input and output populations can be respectively obtained by methods `getCurrentPopulation()` and `getNextPopulation()`.

All stages are interchangeable. This makes possible to structure an algorithm that best fits the problem characteristics and user needs, just deciding how to organize the flow trough the stages. Stages regarding the structure are `Sequence` and `Parallel`, whilst `Operators` are elementary stages performing tasks such as *selection*, *crossover* and *mutation*. The user can assembly its own algorithm deciding the structure and which operators to use, such as choosing between the *roulette wheel* or *tournament* selection, or between a single point or two points crossover, and so on. `SimpleGeneticAlgorithm` provides a facade, structuring a canonical algorithm with the sequence of stages:

- `Selector`: `RouletteWheelSelector` or `TournamentSelector`
- `Crossover`: `OnePointCrossover` or `TwoPointsCrossover`
- `Mutator`: `SimpleMutator`

More in general, a genetic algorithm can be implemented by subclassing `GeneticAlgorithm` and implementing the abstract method `evaluate(Individual)`. This method serves to evaluate the fitness of each individual, therefore it is specifically related to the problem to solve.

JENES provides support to elitism, assuring best individuals to the next generation, whose number is set by `setElitism(int)`. Individuals are substituted according to two strategies:

- *random*: individuals are randomly selected and substituted
- *worst*: individuals with worst fitness are substituted

The first strategy is more efficient as it does not require to sort the population. The drawback is that individuals with a good fitness could be substituted. The second strategy is slower, but assures that only worst individuals are substituted.

JENES uses the Mersenne Twister randomizer [2] for the generation of pseudo-random numbers and values. This provides a fast generation of high-quality random numbers.

Capturing events is useful to collect statistics and perform analyses. Algorithm events can be captured by implementing interfaces `AlgorithmEventListener` and `GenerationEventListener`. The presence of two listener classes is due to performance considerations. Indeed, the latter is only aimed at capturing `onGeneration` events, whilst the first class is aimed at capturing remaining `onStart`, `onPopulation`, `onInit` and `onStop` events. Events can also be captured by overriding methods `onStart(long)`, `onPopulation`, `onInit(long)`, `onGeneration(long)`, and `onStop(long)`. Time in milliseconds is passed to event call back methods. By deafult,

PREPRINT

the body of these methods is empty as they do nothing. An `AlgorithmEventListener` can be registered (removed) by invoking the method `addAlgorithmEventListener` (`removeAlgorithmEventListener`). Similarly, a `GenerationEventListener` is registered by `addGenerationEventListener`, and removed by `removeGenerationEventListener`.

During evolution, JENES collects statistics regarding:

- *Execution*. `GeneticAlgorithm.Statistics` is responsible for storing statistics about the time spent by the whole evolutionary process and the number of generations performed.
- *Population*. `Population.Statistics` collects statistics about a population, regarding highest, lowest, mean and standard deviation of fitness for both legal and illegal individuals.
- *Operators*. Specific classes are responsible for storing statistics and time spent about operators.

More information and detailed documentation are available at `http://jenes.ciselab.org/`.

A. A simple example

The simplest way for setting-up a genetic algorithm in JENES is by subclassing `SimpleGA`. It is possible to subclass `SimpleGA` by defining a specific subclass or more simply by an anonymous subclass as described below.

```
33    SimpleGA<BooleanChromosome> sga =
34      new SimpleGA<BooleanChromosome>(pop,
            GENERATION_LIMIT) {
35    @Override
36    protected void evaluateIndividual(Individual<
          BooleanChromosome> individual) {
37      // Make your evaluation here ...
38    }
39  };

41  sga.setElitism(10);
42  sga.setBiggerIsBetter(false);
43  sga.evolve();
```

`SimpleGA` is a subclass of `GeneticAlgorithm`. The main feature of `SimpleGA` is that the breeding sequence is predefined and made of a `Selector`, a `Crossover`, and a `Mutator`. Which selection method (i.e. Tournament or Roulette Wheel) or crossover method (i.e. One Point or Two Points) to adopt, and algorithm parameters can be decided at construction time.

Both classes are **`abstract`**, requiring to implement the method `evaluateIndividual`. This method provides the fitness of a given individual depending on the particular problem. As an example let us consider the problem of finding a vector of boolean that is completely true (or false). Therefore we will consider as fitness the number of true values in a chromosome, we wish to maximize (or minimize). Let us take into account an implementation made by subclassing `GeneticAlgorithm` directly. The first thing to do is to define the (anonymous) subclass providing an implementation of `evaluateIndividual`.

```
63    GeneticAlgorithm<BooleanChromosome> ga = new
          GeneticAlgorithm<BooleanChromosome>(pop,
          GENERATION_LIMIT) {

65      @Override
66      protected void evaluateIndividual(Individual<
            BooleanChromosome> individual) {

68        BooleanChromosome chrom = individual.
              getChromosome();
69        int count = 0;
70        int length=chrom.length();

72        for(int i=0;i<length;i++)
73          if(chrom.getValue(i))
74            count++;

76        individual.setScore(count);
77      }
78    };
```

This algorithm works with `Individuals` and `Population` of `BooleanChromosomes`. Therefore we first create a sample individual used as prototype for instancing the other population elements.

```
60    Individual<BooleanChromosome> sample = new
          Individual <BooleanChromosome>(new
          BooleanChromosome(CHROMOSOME_LENGTH));
61    Population<BooleanChromosome> pop = new Population <
          BooleanChromosome>(sample, POPULATION_SIZE);
```

The population passes through a sequence of `Stages` for being processed. We create stages and we add them to the algorithm body.

```
80    AbstractStage<BooleanChromosome> selection = new
          TournamentSelector<BooleanChromosome>(3);
81    AbstractStage<BooleanChromosome> crossover = new
          OnePointCrossover<BooleanChromosome>(0.8);
82    AbstractStage<BooleanChromosome> mutation = new
          SimpleMutator<BooleanChromosome>(0.2);
83    ga.addStage(selection);
84    ga.addStage(crossover);
85    ga.addStage(mutation);
```

As we decide to have elitism 1, we specify

```
87    ga.setElitism(1);
```

If we are interested to find a solution containing all false values, we consider the problem as minimization by setting

```
89    ga.setBiggerIsBetter(false);
```

Finally, we run the algorithm

```
90    ga.evolve();
```

When evolution is finished, we can collect algorithm and population statistics in order to get the execution time required to solve the problem and the optimal solution

```
92    Population.Statistics stats = ga.
          getCurrentPopulation().getStatistics();
93    GeneticAlgorithm.Statistics algostats = ga.
          getStatistics();

95    System.out.println("Objective: " + (ga.
          isBiggerBetter() ? "Max! (All true)" : "Min! (
          None true)"));
```

PREPRINT

```
96      System.out.println();

98      Individual solution = ga.isBiggerBetter() ? stats.
            getLegalHighestIndividual() : stats.
            getLegalLowestIndividual();

100     System.out.println("Solution: ");
101     System.out.println( solution.getChromosome() );
102     System.out.println( solution );
103     System.out.format("found in %d ms.\n", algostats.
            getExecutionTime() );
```

More examples and details can be found online.

III. Related work

There exist several publically-available libraries enabling genetic algorithms in Java. For the meaning of comparison, we considered ECJ [3] and JDEAL [4]. ECJ is a research project, developed at George Mason University's ECLab Evolutionary Computation Laboratory, designed to be highly flexible, with algorithms dynamically configured at runtime by a user-provided parameter file. JDEAL is an object-oriented library of Evolutionary Algorithms, with both local and distributed algorithms, for the Java language. JDEAL design is addressed to easily extend and integrate specific operators, chromosomes and algorithms, reusing existing components. The computational load can be distributed through multiple machines, so that idle CPU time can be used to speed up calculations.

In order to make a more exhaustive comparison, we also considered GAlib [5], a C++ native library of genetic algorithms. The library includes tools for using genetic algorithms to do optimization in any C++ program. GAlib includes many different representations, genetic operators, genetic algorithms, stopping criteria, scaling methods, selection methods, and evaluation schemes. It can be used with PVM (parallel virtual machine) to evolve populations and/or individuals in parallel on multiple CPUs.

There are many design choices JENES adopted in order to improve performances. The main feature regards the object pooling as a way for facing garbage collection issues.

Researchers have studied garbage collection for a long time (see for example references [6], [7], [8], [9]), and benefits of garbage collection over explicit memory management are widely accepted, but this imposes trade-off to performances. Blackburn et al. [10] highlight how architectural trends are making this advantage more evident, as standard explicit memory management is unable to exploit the locality advantages of contiguous allocation. It is therefore possible that garbage collection presents a performance advantage over explicit memory management on current or future architectures.

However, the evidence that garbage collection can affect performances is known since long time (see for example [11]). Performance issues become relevant in Java applications, as code is executed by virtual machine and there is a wide usage of garbage collection.

Dieckmann and Hoelzle [12] present an analysis of the memory usage for six of the Java programs in the SPECjvm98 benchmark suite. They found that non-pointer data usually represents more than 50% of the allocated space for instance objects, that Java objects tend to live longer than objects in Smalltalk or ML, and that they are fairly small. Although generational garbage collection[1] largely mitigated the limitation of initial Java garbage collector, the programmer is left to control the creation but not the disposal of objects as pointed out by McManis colums at JavaWorld [14]. Recently Xian [15] conducted an experiment observing that degradation behavior of a widely-used Java application server running a standardized benchmark is due to garbage collection policies.

Reusing objects is a strategy, known as Pooling Design Pattern among researchers and practitioners, in order to boost Java performances [16], [17]. Although the main criticism [13] in pooling objects is that object allocation is very efficient in modern languages and objects pool can keep a large number of unused objects in memory, we share the idea with others [18] that in case of data structures whose size is mostly constant over the run, object pools provides relevant benefits as (i) pooled objects are mostly used and (ii) collectible objects are not left in memory (causing a space overhead) before garbage collector is invoked.

IV. Experimental results

In order to evaluate performances, we tested the framework against a set of standard benchmarking problems, namely:

- De Jong's test functions
- Royal Road Problem
- Travel-Salesman Problem

De Jong's test functions [19] outline 5 optimization landscapes with different complexity. They are defined as:

Each problem is solved by adopting a `BitwiseChromosome`, instead of `BooleanChromosome` , in order to save memory and performing genetic operations faster. Indeed, we employ an array of **`int`**, each holding 32 bits, instead of an array of **`boolean`**s. The chromosome is split in octets each representing an integer value between 0 and 255. This value is after converted to a floating point number assigned to variable x_i in the functions.

The second benchmark we considered is the Royal Road Problem as described in [20]. In this case, the goal is to find a particular pattern of bits, maximizing the function

$$f(x) = \sum_{s \in S} c_s \sigma s(x) \tag{1}$$

where c_s is a value assigned to a schema, x is a bit string and $\sigma_s(x)$ is 1 if x is an instance of the schema s, 0 otherwise.

The last benchmark is the well known Travel-Salesman Problem (TSP), aimed at finding the best Hamiltonian route in a weighted undirected graph. As admissible solutions are permutations, we implemented a particular class of chromosomes named `PermutationChromosome`, which preserves solution admissibility during genetic operations.

[1] A generational garbage collector divides the heap into multiple generations; most JVMs use two generations, a "young" and an "old" generation. Objects are allocated in the young generation; if they survive past a certain number of garbage collections, they are considered "long lived" and get promoted into the old generation. HotSpot offers a choice of three young-generation collectors (serial copying, parallel copying, and parallel scavenge). *Excerpt from [13]*

PREPRINT

Just presented problems have been implemented as well as in ECJ, JDEAL and GALib frameworks, adopting a coding as much as possible similar in order to make more reliable the experimental results. In particular the coding we adopted is:

- De Jong Functions: bitwise coding in JENES, JDEAL and GALib, boolean coding in JDEAL and ECJ.[2]
- Royal Road Problem: bitwise coding in JENES, JDEAL and GALib, boolean coding in ECJ.
- Travel-Salesman Problem: `PermutationChromosome` in JENES, an integer vector in JDEAL, ECJ and GALib.

In this experimentation, we focused only to memory usage and execution time ignoring convergence and result quality. Therefore we considered different populations size (made of 100, 250, 500, 750, 1000, 2500, 5000 individuals). Each test consisted of 10 runs on a Pentium IV 2.4 GHz with 1 MB L2 cache and 1 GB Ram. Time spent by simulation is reported in Fig.2.

	min	max
JENES	13s	1h 18m
ECJ	52s	9h 57m
JDEAL	58s	13h 9m
JDEAL BW	52s	1h 54m
GALIB	1,3s	7m 44s

Fig. 2. Time spent by simulation.

Overall simulation time is able to offer a preliminary overview. GALib performed faster than the other frameworks, as expected being a native solution. Among Java frameworks, JENES generally performed better than ECJ and JDEAL, also when comparison is between bitwise implementations. Adopting boolean coding heavily increased the time for running tests. More evidences can be found analyzing test results in details.

Fig.3 and Fig 4 outline the behavior of frameworks when facing the first De Jong's function, with a population of 100 individuals, where the mean of memory usage and the mean of generation execution time are plotted along 100 generations, over 10 runs. Both JDEAL and ECJ show a saw shaped profile for memory, growing due to unused objects left in memory before they are garbage collected on regular basis. The overhead spent by the virtual machine to sweep memory affects time required to process generations as shown by time peaks in figure. On the contrary, JENES performance is characterized by a flat memory profile and by an almost constant execution time, entailing an efficient memory management. In this sense, JENES seems to be similar to GALib, the most efficient implementation for this benchmark. This behavior proved to be consistent over the 10 runs.

In order to understand how frameworks behave in stressed operational conditions, we present results as obtained by testing the same benchmark with populations made of 5000 individuals each. In Fig.5, JENES again performed better than ECJ and JDEAL. The saw-shaped profile is still visible, although becoming very irregular in the case of JDEAL-bitstring, as memory goes under stress due to physical limits. Also execution time increases, abnormally in the latter case. Again, we can notice peaks when garbage collection operates on memory. The overall experimental results are summarized in Fig.6.

[2]JENES and JDEAL support both bitwise and boolean codings.

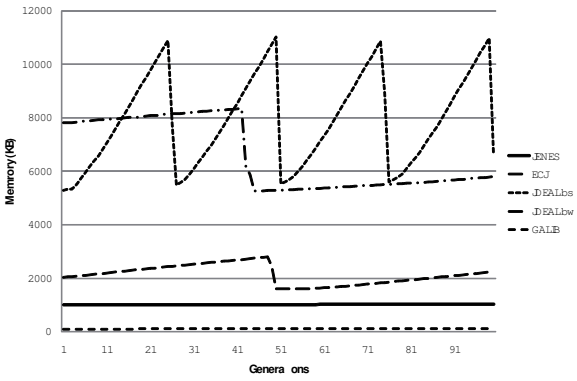

Fig. 3. Memory usage, first De Jong's function, 100 individuals.

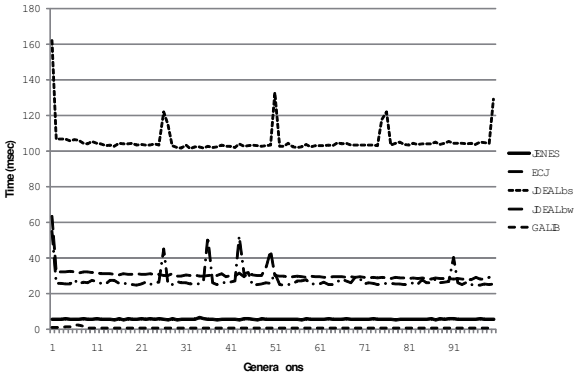

Fig. 4. Execution time, first De Jong's function, 100 individuals.

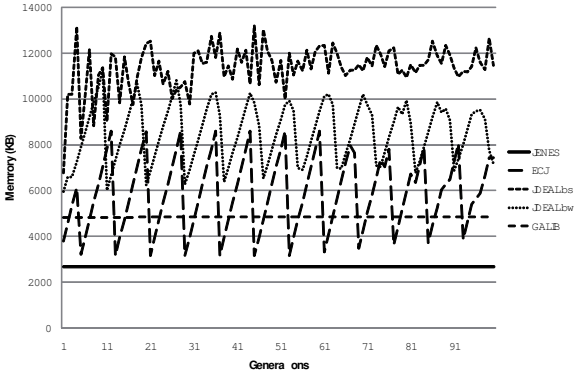

Fig. 5. Memory usage, first De Jong's function, 5000 individuals.

PREPRINT

MEMORY (KB)

	POP SIZE	1F	2F	3F	4F	5F	ROYAL	TSP
JENES	100	1017	1014	1021	1077	1140	1007	2642
	250	1080	1068	1088	1216	1378	1074	2641
	500	1187	1176	1208	1451	1785	1181	2644
	750	1319	1291	1343	1706	1827	1316	2640
	1000	1410	1373	1441	1921	1861	1403	2640
	2500	2004	1919	2094	2981	2797	2006	2643
	5000	2670	2479	2882	5217	4264	2314	2643
ECJ	100	2135	2128	2288	3208	2083	2059	3144
	250	3063	2576	2890	4133	2780	2790	3337
	500	3058	2917	3095	4440	3109	2923	3629
	750	3102	3071	3384	5869	3071	3112	3924
	1000	3248	3090	3765	5745	3071	3318	4376
	2500	4397	4149	4538	10831	3896	4224	6164
	5000	5837	4592	6469	18205	4583	6166	6500
JDEAL (Bitstring)	100	7869	7888	7793	7890	7914	5830	8092
	250	7935	7877	7804	8105	7902	6834	8107
	500	8023	7984	7965	8465	7959	7968	8097
	750	8160	8111	8151	9074	8108	8146	8095
	1000	8357	8188	8223	9607	8225	8144	8094
	2500	8615	6469	8588	12795	6897	8313	8106
	5000	11383	10674	10837	20491	11124	8956	8098
JDEAL (Bitwise)	100	6574	6631	6550	6363	6540	-	-
	250	6740	6680	6732	7686	6813	-	-
	500	8101	7941	8007	7893	7971	-	-
	750	7972	7876	8022	8278	8059	-	-
	1000	8196	8135	8097	8140	8081	-	-
	2500	8273	8295	8342	8527	8350	-	-
	5000	8539	8496	8632	10762	8989	-	-
GALIB	100	108	108	121	244	108	64	161
	250	253	253	284	590	253	146	388
	500	494	494	557	1167	494	283	766
	750	735	735	829	1744	735	420	1144
	1000	976	976	1101	2322	976	557	1522
	2500	2424	2424	2736	5785	2424	1377	3789
	5000	4836	4836	5461	11556	4836	2744	7568

TIME (msec)

	POP SIZE	1F	2F	3F	4F	5F	ROYAL	TSP
JENES	100	1317	1267	1500	9470	5982	7541	54022
	250	3204	3077	3713	23669	14898	18781	57894
	500	6333	6122	7287	47014	29850	37433	54019
	750	9562	9017	11138	70404	45066	57341	58368
	1000	12689	11906	14623	93871	59428	75066	54580
	2500	31881	30190	37048	236852	149820	190516	54167
	5000	64239	61463	74532	469777	295945	378948	54135
ECJ	100	6911	5213	11706	71843	10892	7294	96273
	250	17138	12489	29188	182949	27072	18112	233247
	500	32727	25873	59019	365139	53652	36115	482598
	750	49440	38397	89006	543518	81419	54443	532332
	1000	65272	53032	117457	718287	109441	72885	468156
	2500	161392	132313	295952	1800109	270312	184545	1160974
	5000	325906	262992	584704	3581958	533069	363093	2317770
JDEAL (Bitstring)	100	7869	7888	7793	7890	7914	5830	8092
	250	63645	25948	93481	233970	64718	60695	174122
	500	73528	53535	112231	478674	73221	124510	174314
	750	110905	79990	168982	728515	111381	190352	172918
	1000	148334	107712	224278	938990	149381	154174	173580
	2500	379117	277128	569541	2351405	390756	397995	173465
	5000	771369	559474	1159741	4733739	777174	824435	173705
JDEAL (Bitwise)	100	6580	5162	6120	9318	11188	-	-
	250	15200	13064	15737	23276	28176	-	-
	500	32528	27273	34135	46638	59055	-	-
	750	51797	42016	52974	71438	91090	-	-
	1000	69415	56688	71751	95089	122439	-	-
	2500	194161	153019	201924	244862	326539	-	-
	5000	418057	321617	435669	500045	682664	-	-
GALIB	100	220	220	182	1274	2869	130	852
	250	531	531	451	2974	6715	330	2015
	500	934	934	959	6231	12947	656	4385
	750	1169	1169	1173	10093	19015	1025	6390
	1000	1951	1951	1833	14725	25385	1171	8577
	2500	7771	7771	6129	27402	46410	5307	21144
	5000	18726	18726	17766	19342	35791	15009	26450

Fig. 6. Overall experimental data regarding memory usage and execution time.

V. Conclusions and future work

JENES is a programming framework for developing genetic algorithms in Java designed to pay particular attention to memory and time usage by reusing individuals and populations, thus limiting the garbage collector overhead. Experimentation confirmed that this strategy can effectively address some performance issues. The complexity of internal design is hidden to the programmers by a set of intuitive and extensible APIs.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.

[2] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[3] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, E. Popovici, K. Sullivan, J. Harrison, J. Bassett, R. Hubley, and A. Chircop., "Ecj 18 - a java-based evolutionary computation research system," 2008. [Online]. Available: http://cs.gmu.edu/ eclab/projects/ecj/

[4] J. Costa, N. Lopes, and P. Silva, "Jdeal - the java distributed evolutionary algorithms library," online, 1999. [Online]. Available: http://laseeb.isr.ist.utl.pt/sw/jdeal/home.html

[5] M. Wall, "GAlib: A C++ library of genetic algorithm components," *Mechanical Engineering Department, Massachusetts Institute of Technology*, 1996. [Online]. Available: http://lancet.mit.edu/galib-2.4

[6] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: An exercise in cooperation," *CACM*, vol. 21, no. 11, pp. 966–975, November 1978.

[7] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software Practice and Experience*, vol. 18, no. 9, pp. 807–820, September 1988.

[8] A. W. Appel, "Simple generational garbage collection and fast allocation," *Software Practice and Experience*, vol. 19, no. 2, pp. 171–183, 1989.

[9] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[10] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: the performance impact of garbage collection," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 25–36, 2004.

[11] B. Zorn, "The measured cost of conservative garbage collection," *Software Practice and Experience*, vol. 23, pp. 733–756, 1993.

[12] S. Dieckmann and U. Hoelzle, "A study of the allocation behavior of the SPECjvm98 java benchmarks," in *Proceedings ECOOP'99*, ser. LCNS 1628. Lisbon, Portugal: Springer-Verlag, Jun. 1999, pp. 92–115.

[13] B. Goetz, "Java theory and practice: Urban performance legends, revisited," *IBM DeveloperWorks*, 2005. [Online]. Available: http://www.ibm.com/developerworks/java/library/j-jtp09275.html

[14] C. Mcmanis, "Not using garbage collection," *JavaWorld.com*, January 1996. [Online]. Available: http://www.javaworld.com/javaworld/jw-09-1996/jw-09-indepth.html

[15] F. Xian, W. S. An, and H. Jiang, "Garbage collection: Java application servers' achilles heel," *Sci. Comput. Program.*, vol. 70, no. 2-3, pp. 89–110, 2008.

[16] R. Klemm, "Practical guidelines for boosting java server performance," in *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*. New York, NY, USA: ACM, 1999, pp. 25–34.

[17] M. Kircher and P. Jain, "Pooling pattern," in *Proc. EuroPLoP*, Kloster Irsee, Germany, 2002.

[18] J. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence, "Java programming for high performance numerical computing," *IBM Systems Journal*, vol. 39, pp. 21–56, 2000.

[19] K. De Jong, "An analysis of the behaviour of a class of genetic adaptive systems." Ph.D. dissertation, University of Michigan, 1975.

[20] M. Mitchell, S. Forrest, and J. H. Holland, "The royal road for genetic algorithms: Fitness landscapes and ga performance," in *Proceedings of the First European Conference on Artificial Life*. MIT Press, 1991, pp. 245–254.

PREPRINT