MAFRA: A Java Memetic Algorithms Framework

Natalio Krasnogor
Intelligent Computer System Centre
University of the West of England
Bristol, United Kingdom
Natalio2.Krasnogor@uwe.ac.uk
www.csm.uwe.ac.uk/~ n2krasno

Jim Smith
Intelligent Computer System Centre
University of the West of England
Bristol, United Kingdom
James.Smith@uwe.ac.uk

Abstract

In this paper we will introduce the **M**emetic **A**lgorithms **FRA**mework, a general purpose evolutionary computation framework. MAFRA allows the construction of complex evolutionary systems with a maximum of reuse between different instantiations of the framework. MAFRA has been developed in java using design patterns to allow for its easy extension and utilization in different problem domains. MAFRA is an open source code project. The files composing the Memetic Algorithm Framework can be seen in Fig. 1.

1 Introduction

It is now well established that a combination of Genetic Algorithms with local search are amongst the most powerful metaheuristcs to search complex continuous or combinatorial spaces [5, 8]. GAs combined with local search (LS) were named "Memetic Algorithms"(MAs) in [10]. In the literature, MAs have also been named Hybrid Genetic Algorithms, Genetic Local Searchers, Lamarckian Genetic Algorithms, Baldwinian Genetic Algorithms and even Parallel Genetic Algorithms.

The goal of this paper is to introduce the reader to a set of interrelated classes. These clases constitute an object oriented framework that allows for a rapid construction of novel MAs applications and experimental settings. It is not the primary goal of MAFRA to provide with a fast "running system" but with a fast "design and test system". This framework will serve object oriented (OO) programers and evolutionary computation developers that want to reuse not only at the code level but also at the design level. However, non OO programmers, or developers, that alredy have thousands of lines of code (i.e. Fortran, C, Pascal) might find little benefit in using MAFRA. It is important to note that this framework serves also to develop pure evolutionary algorithms where no local search is involved.

Even though many "general purpose" libraries are available in the web for constructing evolutionary applications, just few of them are of real value if developing time and effort is considered, the reason is that they do not exploit the possibilities of an object oriented design, let aside the use of design patterns. Quite recently several papers appeared showing the benefits of an object oriented design of such a framework[9],[1],[11],[2] and even a hierarchical and polytyping functional programming approach has been developed [6](see references herein).

2 Brief Introduction to Design Patterns and Object Oriented Frameworks

Before going into the details of MAFRA's architectural design and use we will briefly introduce the concepts of design patterns and frameworks. The roots of design patterns are due to the contemporary architect Christopher Alexander [1] who wrote several books related to urban planning and architectural design. Patterns (and Pattern Languages) are used to capture experiences in the design of solutions to difficult but ubiquitous problems and to describe best practices in such a way that other designers can reuse not only "computer code" but fundamentally "designs". A software framework is intimately related to design patterns. A framework is a reusable architecture (i.e. a collection of classes), seldom a complete application, that provides the skeleton and basic behavior of a certain kind of software product. It explicitly defines a

[1] http://www.math.utsa.edu/sphere/salingar/Chris.text.html

contextual background of "memes", metaphors, ideas, etc which collaborate and interact in a given problem domain. The skeleton provided by a framework is usually "filled" or completed with specific plug-ins that are connected through some hot-spots or plug-points. These plug-points are usually implemented by delegation, call-backs or polymorphism [3]. The reader should not confuse a framework with a programming library. There is an important difference between the two. When the former is used, the user needs to implement just a few call back functions that will be called from within the framework. It is the framework the responsible for doing almost all the work. A programming library is just a collection of methods and constants that need not show any cohesion at all. In general a framework is a collection of tightly related classes whose interrelation is usually given by the design patterns that the framework implements.

2.1 The Design Patterns Used in MAFRA

It is out of the scope of this paper the detailed explanation of what is and how to use a design pattern. A concise book on the topic is [3]. In MAFRA the following design patterns from Gamma et al's book are extensively used:

- **Abstract Factory** which provides an interface for creating families of related objects without specifying their concrete implementations. In this way one can guarantee that the system is independent on how the specific objects are defined, created or manipulated. See 4.1 and 4.2.4.
- **Factory Method** which defines an interface for creating an object, but lets subclasses decide which class to instantiate. This pattern allows a class to defer instantiation to subclasses. See 4.2.4.
- **Strategy** defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. See 4.2.3.
- **Template Method** defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps or aspects of an algorithm without changing the algorithm. See 4.3.
- **Visitor** represents an operation to be performed on the elements of an object structure. Visitor lets you define new operations without changing the classes of the elements on which it will operates. See 4.4.

For java implementations of these patterns the reader is referred to [4].

3 MAFRA Architecture

In Fig. 2 we show a class association diagram of classes *EvolutivePlan*, *Problem*,*Plan* and *GA*. *EvolutivePlan* class is the class that holds the three main parts of an evolutionary system: a GA, a problem to be solved and a plan. The plan will use the GA to solve the given problem. The *Problem* class provides the basic mechanisms by which a problem is defined. It specifies operations to read a problem instance, to create and check the feasibility of a solution, compute its fitness, etc.

Plan class, which is abstract, implements a Template Method Pattern. In essence it is the responsible of the interaction between the parts of the evolutionary system. Its subclasses redefine all or some of the DoName(...) (see 4.3) methods as a way of defining a concrete and complete behaviour for an evolutive plan.

A *GA* instance interacts with several different strategies that specify the way to perform mutations, local search, crossover, selection,etc. This strategies represent algorithms that might be interchanged at the application level without affecting any other part of the application. In Fig. 3 it is possible to see this relations.

4 MAFRA's Patterns and Use Case

We will shortly explain how we use the design patterns mentioned before in the construction of the framework. A use case based on the *Counting Ones* problems will clarify the concepts.

4.1 Abstract Factory Pattern

We will explain the Abstract Factory Pattern by its use in the definition of the *CrossOverFactory* class. In general the designer of the framework can not anticipate the kind of encodings that the user will utilize to define the individuals of a population. However, he must provide a way to perform a crossover between two or more solutions. To accomplish this we can use the design shown in figure 4.

In this figure we define a *CrossOverFactory class* which prescribes an interface to create crossover objects. There will be several sorts of crossover objects: one-point, two-point, uniform, etc. Each one of these objects will know how to mate 2,3 or more individuals using the appropriate number of 'crossing over' points.

According to the specific encoding of a solution (i.e. a String encoding, binary encoding, etc) there will exist subclasses of *CrossOverFactory* class that will create the appropriate instance of a crossover object. In the figure mentioned, two subclasses appear: *BitSetEncodingXFactory* and *StrinEncodingXFactory*. The user application will use the appropriate class according to the representation needed. Each one of those subclasses co-operate with the *AbstractCrossOver* hierarchy to actually return a crossover instance. In this way it is very easy to prototype a system with a given encoding an later, just changing the appropriate factory it is possible to use a, i.e., more efficient encoding.

4.2 The Counting Ones Problem: A Use Case

In this section we describe a use case of MAFRA using the well known *COnes* which is defined by *Counting Ones Problem*
Instance: An integer $k > 0$.
Solution: A binary string S, such that $|S| = k$. .
Measure: The number of 1's (ones) present in S

4.2.1 Code Example

Let us analyze a piece of java code needed to run an application based on MAFRA to solve *COnes*. The code will make explicit the use of the hot-spots. Please note that the line numbers to the left of each code line are not part of the source code but only serve to facilitate the following explanations.

```
1 import java.util.*;
2 public class Test{

3 public static void  main(String []argv) throws CloneNotSupportedException
4   {
5     /* General Evolutionary Plan Variables */
6     EvolutionaryPlan                       theEvolutionaryPlan;
7     GA                                     myGa;
8     SimpleGAPlan                           myPlan;

9     /* CrossOver Stage Variables */
10     MatingStrategy                         myMatingStrategy;
11     BitSetEncodingXFactory                 myCrossOverFactory;
12     TournamentSelectionMethod              myMatingSelectionMethod;

13     /* Mutation Stage Variables */
14     MutationStrategy                       myMutationStrategy;
15     BitSetEncodingMFactory                 myMutationFactory;

16     /* Selection Stage Variables */
17     SelectionStrategy                      mySelectionStrategy;
18     MuPlusLambdaSelectionStrategyExecutor myMPLSExecutor;

19     /* Problem and Individual Variables */
20     OneMaxProblem                          myProblem;
21     IndividualBitSetFactory                myFactory;

22     /* Individual and Population Variables */
23     Population                             myPopulation;
24     Population                             myOffsprings;
25     BitSetIndividual                       anIndividual;
26     BitSet                                 aChromosome;

27     /* Visitors Variables */
28     FitnessVisitor                         aFitnessVisitor;
29     DisplayVisitor                         aDisplayVisitor;
30     SortingVisitor                         aSortingVisitor;

31     /* General Variables */
32     Hashtable                              args;
33     long i,j,gene;

34     /* General Initialization */
35     MAFRA_Random.initialize(27131411);
36     aSortingVisitor = new SortingVisitor();
37     aDisplayVisitor = new DisplayVisitor();
38     myProblem       = new OneMaxProblem(100);
39     aFitnessVisitor = new FitnessVisitor(myProblem);

40     /* Populations Initialization*/
41     myFactory           = new IndividualBitSetFactory(myProblem);
42     myPopulation        = new Population(20,myFactory);
43     myPopulation.setName("Parents");
44     myPopulation.acceptVisitor(aDisplayVisitor);
45     myOffsprings = new Population();
46     myPopulation.copyTo(myOffsprings,20);
47     myOffsprings.setName("Offsprings");
48     myOffsprings.acceptVisitor(aDisplayVisitor);

49     /* Mutation Stage Initalization */
50     myMutationFactory   = new BitSetEncodingMFactory();
51     myMutationStrategy  = new MutationStrategy(myMutationFactory);
52     args = new Hashtable();
53     args.put("Population",myOffsprings);
54     args.put("ProbabilityPerIndividual", new Double(0.0));
55     myMutationStrategy.setArguments(args);

56     /* CrossOver Stage Initialization */
57     myCrossOverFactory      = new BitSetEncodingXFactory();
58     myMatingSelectionMethod = new TournamentSelectionMethod();
59     myMatingStrategy        =
          new MatingStrategy(myCrossOverFactory,myMatingSelectionMethod);
60     args = new Hashtable();
61     args.put("offspringsPopulation",myOffsprings);
62     args.put("parentsPopulation",myPopulation);
63     args.put("matingProbability",new Double(1.0));
64     args.put("lambda",new Long(20));
65     args.put("matingPoolSize", new Long(100));
66     myMatingStrategy.setArguments(args);

67     /* Selection Stage Initialization */
68     myMPLSExecutor      =
  new MuPlusLambdaSelectionStrategyExecutor(myPopulation,myOffsprings,20,20);
69     mySelectionStrategy = new SelectionStrategy(myMPLSExecutor);

70     /* GA setting */
71     myGa = new GA();
72     myGa.addMatingStrategy(myMatingStrategy);
73     myGa.addMutationStrategy(myMutationStrategy);
74     myGa.addSelectionStrategy(mySelectionStrategy);
75     myGa.addVisitor("sortingVisitor",aSortingVisitor);
76     myGa.addVisitor("displayVisitor",aDisplayVisitor);
77     myGa.addVisitor("fitnessVisitor",aFitnessVisitor);
78     myGa.addPopulation("parentsPopulation",myPopulation);
79     myGa.addPopulation("offspringsPopulation",myOffsprings);
80     myGa.addParameter("maxGenerationsNumber",new Long(50));

81     myPlan = new SimpleGAPlan(myGa);
82     theEvolutionaryPlan =
          new EvolutionaryPlan(myGa,myProblem,myPlan,null,null,null);
83     theEvolutionaryPlan.run();
84   }
85 }
```

4.2.2 Understanding The Code

In order to integrate MAFRA into a running application the user must define an evolutionary plan. The evolutionary plan, given by an instance of the class EvolutionaryPlan is defined by a GA, a plan, a problem, a statistician, a log and a display object. See figure 2. For each one of those objects an appropriate class in MAFRA exists. The variable definitions between lines 5 and 30 shows the classes involved. Once all the objects are defined and initialized it is possible to create a new evolutionary plan and to run it. This is accomplished in lines 81 to 83. Note that the class SimpleGAPlan is a subclass of Plan. The object myGa,

which is an instance of class GA, is defined and set in lines 70 to 80. As the reader can see in the UML documentation, an instance of a GA can be initialized with instances of the following classes: *FinalizationStrategy*, *InitializePopStrategy*, *RestartPopStrategy*, *MutationStrategy*, *SelectionStrategy*, *MatingStrategy* and *LocalSearchStrategy*. In our case, because we are using a simple plan composed by mating, selection and mutation phases alone, only the associated instances are used. The above mentioned classes implements the Strategy Pattern. An example of its use is explained in 4.2.3. The GA is also given the visitors, populations and miscellaneous parameters that define its behavior.

Due to the fact that different instantiations of MAFRA might need a variety of parameters to initialize the strategies used, they are given using a hash table. Each entry to the argument's hash table is an association composed by a key (the name of the parameter) and a value. See the javadoc documentation files for the reserved parameters' names.

4.2.3 Specifying a Strategy

The classes with names of the form XXXXStrategy implement a Strategy pattern. These are subclasses of Strategy which implement the Executor interface. In essence, any instance of Strategy or one of its subclasses implements the execute() method. Each class that implements a strategy holds an object that will perform a specific algorithm. The body of the execute() method might have additional code (default code). Some strategies receive during creation time an instance of a factory that creates the appropriate executor. See UML documentation, javadoc files and source code for details. As an example consider lines 67 to 69 where a selection strategy is created with an executor object as parameter. In this case the executor is a particular instantiation of a $(\mu + \lambda)$ selection. It receives as parameters, the populations to work with (parents and offsprings), μ and λ.

In [7] several memetic algorithms were compared on the *TSP* and the *Protein Folding* problem domains. In the experiments described therein, different MAs were obtained just by changing the following line in the application main code:

```
myLocalSearchStrategy = new
 ElitistLinearAnnealingLocalSearchStrategy(myLocalSearchFactory);

 to

myLocalSearchStrategy = new
 ElitistHillClimberLocalSearchStrategy(myLocalSearchFactory);

 to

myLocalSearchStrategy = new
 ElitistLocalSearchStrategy(myLocalSearchFactory);

etc.
```

The application employed different local search strategies which in turns used a local search factory. The factory provides with a set of basic 'move' operators. An important feature to bear in mind is that MAFRA allows a dynamic loading and unloading of strategies, hence, allowing to change (i.e.) the local search strategy on-line.

4.2.4 Using Factories

Two patterns related with factories are described bellow.

The Abstract Factory Pattern:

The strategies mentioned in the previous section make extensive use of the Abstract Factory pattern to achieve a reusable design. Consider for example lines 56 to 66 where the mating strategy is initialized by receiving a TournamentSelectionMethod instance and a factory for AbstractCrossOver instances. In this case the factory is BitSetEncodingXFactory. The user must note that if we decide to change the encoding of our problem from BitSet to String we only need to instantiate a StringEncodingXFactory and pass it to the MatingStrategy (see line 59). The new factory will be responsible of providing the MatingStrategy instance with a crossover object that "knows" how to perform (i.e.) a two parents one point crossover under a string encoding.

The Factory Method:

The Factory Method pattern is used by ProblemFactory and IndividualFactory classes. Both of this classes are intimately related to Population and Individual classes. Between lines 38 and 42 a ProblemFactory subclass instance, in this case OneMaxProblem (*COnes*), is instantiated and passed onto a specific factory for individuals in the BitSetEncoding. In turns, this instance will be used by the population to create and initialize new individuals under a specific problem and encoding.

4.3 The Template Method Pattern

The Template Method Patterns provide with a general purpose algorithms which can be easily tailored to different situations. The class *Plan* (see UML design) implements this pattern and is used in line 81. SimpleGAPlan overrides just a few methods from its superclass, this methods are DoReproduce(), DoMutate() and DoUpdatePopulation(). When a plan is requested to execute, the run method in the superclass is launched. This method has a fixed skeleton:

```
1      /** Executes the plan.*/
2      public void run()
3      {

4       DoInitPopulation();
5       while (!DoCheckTermination())
6       {

7        DoReproduce();
8        DoFineGrainSchedulerX();
9        DoMutate();
10       DoFineGrainSchedulerM();
11       DoCoarseGrainScheduler();
12       DoUpdatePopulation();
13       DoRestartPopulation();
14       DoMetaScheduler();
15       generationsNow++;
16       showPopulation();
17      }
18     }
```

where the default implementation of all the DoXXXX() methods is "{}" (an empty body). Hence subclasses, by redefining those methods, can implement a plethora of different plans.

4.4 The Visitor Pattern

In this use case we used three different visitors, namely SortVisitor, FitnessVisitor, and DisplayVisitor. If the user wants to change how and when the fitness is calculated, the only thing he needs to do is to redefine the FitnessVisitor. If a different way of displaying the solutions is required, changing just the DisplayVisitor will do the job. In the same way, if a new kind of operation or measurement needs to be applied to the population or its individuals it can be defined using a new visitor, in such a way that it is unnecessary to redefine any other class in the application or in MAFRA.

5 Conclusions and Further Work

Due to space limitation we were only able to introduce MAFRA. We have been working with it on several problems, *TSP*,*NK-Landscapes*,*Max-SAT*, *COnes*, etc and it was straight forward to adapt to new encodings or evolutionary scenarios. Its actual version supports not only MAs but also co-evolutionary schemes where several populations evolve at once. We plan to add multi-criteria functionality in the immediate future. An Important feature that will be added to the framework is a top-level GUI that will allow an interactive construction of evolutionary applications. It will represent each strategy that defines an application by a box that will be filled with the appropriate object instances taken from a toolbox of available operators. This top-level design tool will make MAFRA much more easy to use by non OO programers and evolutionary computation scientist in general.

References

[1] M. Eldred, W. Hart, W. Bohnohoff, V. Romero, S. Hutchinson, and A. Salinger. Utilizing object-oriented design to build advanced optimization strategies with generic implementation. *American Institute of Aeronautics and Astronautics.*

[2] C. Fleurent and J. Ferland. Object oriented implementation of heuristic search methods for graph coloring, maximum clique and satisfiability. In *DIMACS Series in Discrete MAthematics and Theoretical Computer Science.* American Mathematical Society, 1994.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Desing Patterns, Elements of Reusable Object-Oriented Sofware.* Addison-Wesley, 1995.

[4] M. Gran. *Patterns in Java, A catalog of reusable design patterns illustrated with UML.* Wiley, 1998.

[5] W. E. Hart. Adaptive global optimization with local search. *Ph.D. Thesis, University of California, San Diego*, 1994.

[6] N. Krasnogor, P. Mocciola, D. Pelta, G. Ruiz, and W. Russo. A runnable functional memetic algorithm framework. In *Proceedings of the Congreso Argentino de Ciencias de la Computacion, Vol. I*, pages 525–536, Universidad Nacional del Comahue, Argentina, 1998.

[7] N. Krasnogor and J. Smith. A memetic algorithm with self-adaptive local search: Tsp as a case study. In *GECCO-2000: Proceedings of the Genetic and Evolutionary Computation Conference.* Morgan Kaufman, 2000.

[8] M. Land. Evolutionary algorithms with local search for combinatorial optimization. *Ph.D. Thesis, University of California, San Diego*, 1998.

[9] T. Lenaerts and B. Manderick. Building a genetic programming frammework, the added value of design patterns. *Proceedings of EuroGP 98*, 1998.

[10] P. Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms. Technical Report Caltech Concurrent Computation Program, Report. 826, California Institute of Technology, Pasadena, California, USA, 1989.

[11] R. Slootmaekers, H. Wulpen, and W. Joosen. Modelling genetic search agents with a concurrentt object oriented language. *Proceedings of HPCN Europe 1998, Lecture Notes in Computer Science 1401*, 1998.

MAFRA's home page	http://www.csm.uwe.ac.uk/~ n2krasno/MAFRA/mafra.html
This document	http://www.csm.uwe.ac.uk/~ n2krasno/MAFRA/DOCS/mainDocument.html http://www.csm.uwe.ac.uk/~ n2krasno/MAFRA/DOCS/mainDocument.ps
Javadoc documentation	http://www.csm.uwe.ac.uk/~ n2krasno/MAFRA/DOCS/tree.html
UML design	http://www.csm.uwe.ac.uk/~ n2krasno/MAFRA/DOCS/mafra.ps
Source code	http://www.csm.uwe.ac.uk/~ n2krasno/MAFRA/SOURCE/mafra.tar.Z

Figure 1: The MAFRA project

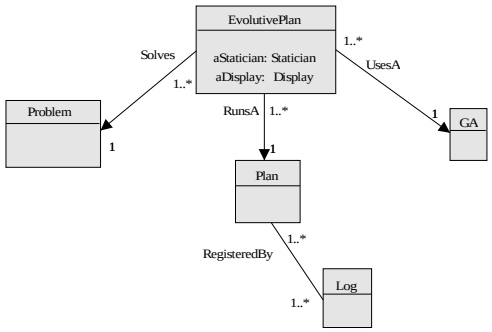

Figure 2: Principal Classes: EvolutivePlan, GA, Plan, Problem

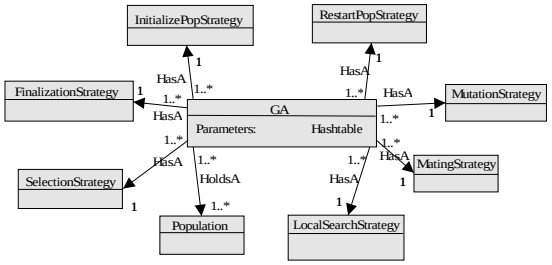

Figure 3: The classes that collaborate whith GA

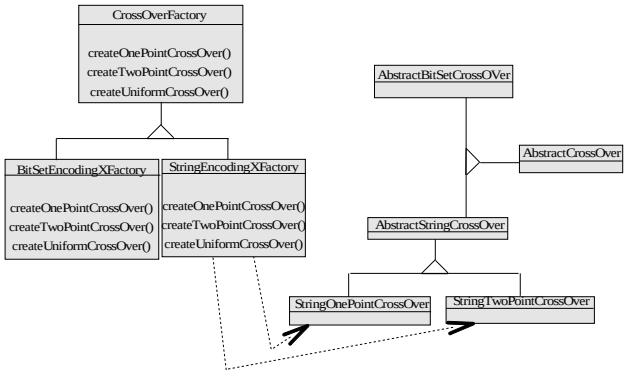

Figure 4: Abstract Factory Design Pattern: a CrossOverFactory example