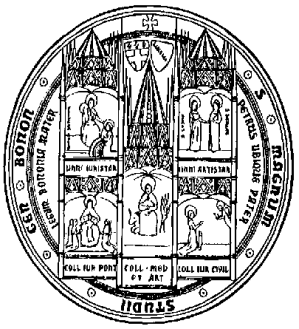


Alma Mater Studiorum

Università degli Studi di Bologna
DEIS

EasyGenetic: A Template Metaprogramming Framework for Genetic Master-Slave Algorithms

Stefano Benedettini Andrea Roli
Luca Di Gaspero

May 20, 2009

DEIS Technical Report no. DEIS-LIA-005-09 *LIA Series no. 95*

EasyGenetic: A Template Metaprogramming Framework for Genetic Master-Slave Algorithms

Stefano Benedettini [1] Andrea Roli [1] Luca Di Gaspero [2]

[1] *DIEGM, Università di Udine*
via delle Scienze 208, I-33100, Udine, Italy
l.digaspero@uniud.it
[2] *DEIS, Campus of Cesena*
Alma Mater Studiorum Università di Bologna
via Venezia 52, I-47023 Cesena, Italy
s.benedettini,andrea.roli@unibo.it

May 20, 2009

Abstract. In this work we present `EasyGenetic`, a genetic solver based on *template metaprogramming*, that enables the user to configure the solver via templates. The framework allows to combine flexibility with efficiency. The framework is designed to be applied to problems for which a master-slave solution strategy can be defined. In the realm of combinatorial optimization, such problems can be those for which a parametrized constructive procedure is available and the solver search the parameter space. We present two successful applications of `EasyGenetic` to hard optimization problems, namely the Haplotype Inference Problem and the Capacitated Vehicle Routing Problem.

Keywords: *Genetic algorithms, metaheuristics, template metaprogramming*

Contents

1 Introduction

Genetic algorithms (GAs) are applied since several decades to problem solving and a plethora of successful cases demonstrates the effectiveness of this paradigm as a tool for building "automatic problem solvers". One of the strengths of GAs is that their machinery is general and it works independently of the problem under consideration. As a consequence, once the user has defined the problem model, a fitness function and he/she has set the algorithm parameters, a GA based solver is ready to be run.

Among other domains, GAs are particularly suitable for design problems, in which the construction of an artifact depends on a set of design choices and parameter values. The designer's goal is to to choose among the set of options and the values of parameters in such a way that the artifact produced has the required characteristics. From an abstract point of view, this problem can be seen as a composite master-slave task: a low-level (slave) task consists of building the artifact on the basis of a parametrized constructive procedure which is instructed by the parameter setting found by a high-level (master) task.

In this work we adopt this very perspective and present `EasyGenetic`, a tool that enables the algorithms designer to implement a genetic solver by combining basic components and to tackle combinatorial optimization problems for which a parametric constructive procedure is available. `EasyGenetic` is a framework for implementing genetic solvers based on the master-slave decomposition of the problem which is developed employing *template metaprogramming* that allows to combine flexibility with efficiency.

Available tools for implementing evolutionary computation solvers are numerous and range from combinatorial optimization solvers (e.g., ParadisEO [7]), to generic evolutionary algorithm tools (such as ECJ [10]). However, to the best of our knowledge, none of them employs template metaprogramming[1], nor they are based on the master-slave metaphor.

The remainder of this paper is structured as follows. In Section 2 we illustrate the architecture of the genetic master-slave solver, whose architecture and implementation is detailed in Section 3. In Section 4 we briefly present two case studies in which `EasyGenetic` has been successfully applied and we conclude in Section 5 with an outlook to future work.

2 The Prototypical Genetic Master-Slave Algorithm

The general idea of the genetic master-slave algorithm we propose is based on the hypothesis that it will be possible to split solution construction in two phases: in the first phase, the parameters of a constructive procedure are set by a *master* solver

[1] ParadisEO makes use of design patterns based on *template instantiation*, but it does not exploit metaprogramming techniques.

and in the second phase the solution is actually built by a *slave* solver. For instance, the constructive procedure can be based on a sequence of decisions whose order is defined by the master solver. Many problems can be decomposed in this way, such as planning or assignment problems.

Algorithm 1 Master-slave high-level framework

Procedure master

1: $\mathcal{P} \leftarrow$ buildInitialPopulation(n)
2: evaluate($\mathcal{P}$)
3: **while** terminating conditions not met **do**
4: $\quad \mathcal{P}' \leftarrow$ applyGeneticOperators($\mathcal{P}$)
5: $\quad$ evaluate($\mathcal{P}'$, slave)
6: $\quad \mathcal{P} \leftarrow$ bestOf(n, $\mathcal{P}$, $\mathcal{P}'$)
7: **end while**
8: **return** min($\mathcal{P}$)

Procedure slave

1: **Input:** population $\mathcal{P}$
2: **Output:** evaluation of individuals of $\mathcal{P}$
3: **for all** $p \in \mathcal{P}$ **do**
4: $\quad s \leftarrow$ buildSolution(p)
5: $\quad$ fitness[p] $\leftarrow$ eval(s)
6: **end for**

In `EasyGenetic`, the master is a genetic algorithm (GA), while the slave algorithm can be, in general, any deterministic constructive procedure that accepts an initial set of parameters that completely define solution construction. For example, for combinatorial problems there exist constructive procedures based on the following parameters: the sequence of objects to be included in the solution and/or the decisions to be taken, the set of preassigned variables or the set of hard constraints to be fulfilled. In a sense, the master explores the search space of "parameter settings", employing the solution returned by the slave as the evaluation of those search space points.

The master-slave scheme is detailed in Algorithm 1. It can be observed that the structure is the typical one of a steady-state genetic algorithm, with the difference that the evaluation of an individual is performed first by asking the slave to build a solution, as a function of the information provided by the individual (line 4), and then evaluating the corresponding solution (line 5). The main advantage of partitioning the problem into master and slave is a clearer separation of concerns, which helps in designing a more extensible solver and allows a simple and neat implementation.

3 The `EasyGenetic` Framework

As already mentioned, `EasyGenetic` is based on template metaprogramming and generic programming techniques. In this section we give a brief description of these techniques and we introduce the notions of *Concept* and *Policy Classes*, which are the fundamental abstractions exploited in the design phase of `EasyGenetic`. Afterwards,

```
T x
T x(y) // or equivalently T x = y
x = y
x == y
```

Figure 1: The required expressions for the *ValueSemantics* concept.

we provide an overview of the architecture of `EasyGenetic` along with the description of the entities involved in the framework.

3.1 Template Metaprogramming

Template metaprogramming is a programming technique used to generate source code at compile time. The main applications of this technique include compile time generation of classes, compile time optimizations (such as implementation selection and loop unrolling), and generic programming. In this sense, templates can be regarded as a Turing-complete purely functional sub-language embedded in the C++ language, which allows compile time computation on the space of types. Prominent examples of applications of these techniques are the C++ Standard Template Library [12] and the Boost Libraries [6].

3.1.1 Generic Programming and Concepts.

Generic programming is a programming style that focuses on building algorithms applicable to the widest possible variety of types. To do so, the programmer has to identify the minimal set of requirements on the types involved in an algorithm and to ensure that any type conforming to those constraints can be used with the algorithm.

Concepts play a central role in generic programming because they enable a programmer to express the required operations on a type in a concise and effective way[2]. Specifically, in C++ a *concept* embodies a set of requirements on a template type parameter coded in terms of valid expressions that involve that type. More precisely, a concept is a structural interface in the sense that any type that syntactically match the required expressions is a model of that concept. As an example, let us consider the *ValueSemantics* concept illustrated in Figure 1. This is a concept on a generic type `T` that requires the definition of the expressions depicted in the figure (`x` and `y` are variable of type `T`).

Therefore, every type conforming to the *ValueSemantics* concept must be default constructible, copy-constructible, assignable and equality-comparable. Conversely,

[2] A very similar language feature can be found in Haskell Type Classes.

every type that satisfies these constraints is automatically a model of the *ValueSemantics* concept, with no need of an explicit account of this fact. As a consequence, every type that models the *ValueSemantics* concept has the same behaviour as a scalar type (e.g., like `int`) and it can be treated similarly also from a syntactic point of view. Finally, the possibility of manipulating types by means of a uniform syntax and the C++ feature to overload language operators (e.g., arithmetic operators such as `+` or `-`) help to increase the conciseness and expressiveness of programs that make use of concepts.

It is worth to notice that, differently from those languages that implement nominal subtyping by means of *interfaces* (e.g., Java), concepts give rise to a form of design-by-contract that is more versatile and further promotes separation of concerns. This is due mainly because of the nature of the subtype relation, which is structural opposed to nominal, and, as in the case of C++, more efficient because it does not involve virtual function call, allowing compilers to perform code optimization.

3.1.2 Policy Classes.

Concepts can be used to implement a programming idiom called *policy-based design*. Policy classes are purely behavioral units of code (actually template classes). In the design of a library, sometimes it is useful to allow library users to customize some behavioral aspects of a component. In order to do this, a component can be parametrized with a number of (template) parameters each corresponding to a specific behavioral aspect. Moreover, each parameter is required to be a model of a concept and the library user has to instantiate those parameters with an actual type conforming to the specific concept. In this scenario, the parametrized component is called a host-class while the actual type parameters are called policy classes. Refer to [1] for a thorough explanation of Policy Classes and a set of real-world examples.

In `EasyGenetic`, the policy classes idiom is used to implement orthogonal aspects such as: population update and initialization and the selection mechanism.

3.2 Architecture of the System

An overview of the architecture of `EasyGenetic` is depicted in Figure 2 in form of an UML class diagram. Each component in the diagram represents either a concrete class or a concept; in particular, classes representing concepts are denoted by the $\ddot{i}\acute{\iota}\frac{1}{2}Concept\ddot{i}\acute{\iota}\frac{1}{2}$ stereotype. For clarity sake, actual types will be typeset in `monospaced` text, while concepts and type variables will be typeset in *italics*.

Since a generic programming approach naturally encourages a design methodology that follows the separation of concerns principle, each entity in the system architecture provides only a restricted non-overlapping set of features required by the genetic algorithm. The main component in the architecture is the `Solver` class

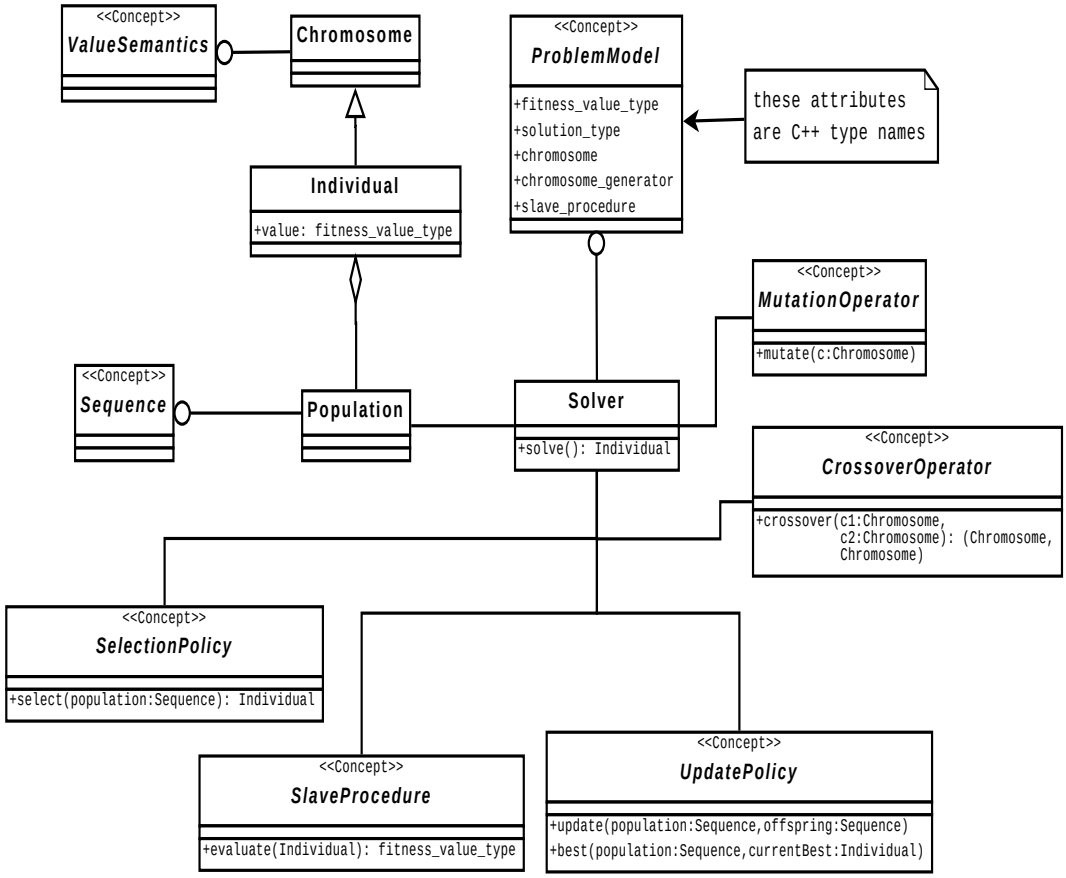


Figure 2: Architecture of `EasyGenetic`

that contains the actual skeleton of the generic master-slave genetic algorithm (Figure 3). Following the principles of generic programming, the `Solver` is a class template whose type parameters are actual types conforming to the relative concepts. One of the advantages of this approach is that we obtain a solver that is configurable in every aspect at compile time by providing the desired type parameters. In this way we avoid tedious boilerplate code and to change the components of the system in an almost declarative way.

3.2.1 ProblemModel.

The central entity of the architecture is the *ProblemModel* concept, whose specification is given below.

```
T::fitness_value_type
T::solution_type
```

```
class Solver {
  Individual solve() {
    std::vector<Individual> pop(pop_size);
    for (uint i = 0; i < pop_size; ++i) {
        Chromosome c = ChromosomeGenerator::generate(model);
        fitness_value_type v = SlaveProcedure::evaluate(model, c);
        pop[i] = Individual(c, v); // Individual is-a Chromosome
    }
    std::sort(pop.begin(), pop.end());
    Individual best = pop.back();
    for (/* termination conditions not met */) {
        std::vector<Individual> offspring(offspring_size);
        while(/* offspring is not full */){
      //select chromosome with SelectionPolicy::select(pop)
      //apply crossover and mutation operators
        for (uint i = 0; i < offspring_size; ++i)
            offspring[i].value = SlaveProcedure::evaluate(model, offspring[i]);
        UpdatePolicy::update(pop, offspring);
        UpdatePolicy::best(pop, offspring, best);
    }
    return best;
  }
}
```

Figure 3: `Solver` simplified main method.

```
  T::chromosome
  T::chromosome_generator
  T::slave_procedure
```

The purpose of *ProblemModel* is to provide the `Solver` with the actual problem-specific type information about various key entities of the system. Each member of the *ProblemModel* concept represents an actual type (such as scalar C++ type like `int` or `double`, or user-defined classes), and some of them must conform to specific concepts.

Starting from the top, we have the following requirements: *fitness_value_type* is a scalar C++ type of the fitness value, while *solution_type* is the actual type of a solution to the problem. *chromosome* is the type of the *Chromosome* which comprises the genetic information of a single *Individual*. *Chromosome* type is obviously problem-specific, therefore it has to be defined by the user. It represents the actual input to the slave procedure and must be model of the *ValueSemantics* concept (see Figure 1). *chromosome_generator* and *slave_procedure* are the actual types that model the *ChromosomeGenerator* and *SlaveProcedure* concepts.

3.2.2 Individual.

An `Individual` is an actual class that models a population individual and encapsulates its genetic material, i.e. the representation of a chromosome, and its fitness function value, which is computed by the slave procedure.

3.2.3 SlaveProcedure.

The *SlaveProcedure* concept is another core concept of the architecture. It defines the interface to which the slave component must comply. A model of this concept must provide a static function which accepts an *Individual* as its argument and returns the related fitness function value.

```
fitness_value_type T::evaluate(ProblemModel& model, Individual& individual)
```

3.2.4 ChromosomeGenerator.

The *ChromosomeGenerator* concept specifies how to generate a new chromosome. The chromosomes generated by means of the procedure related to this concept are used in the initialization step of the algorithm.

```
Chromosome T::generate(ProblemModel& model)
```

3.2.5 UpdatePolicy.

The *UpdatePolicy* concept specifies the interface for updating the population for the next generation of the genetic algorithm.

```
T::update(Sequence& population, Sequence& offspring)
T::best(Sequence& pop, Sequence& offspring, Individual& incumbentSol)
```

The *update* procedure simply selects the individuals which constitute the population of the next iteration according to a specific criterion. The framework provides already two possible built-in implementations: a steady state update, which selects the best individuals from either the current population and the current offspring, and a replacement update, which simply discards the current population and carries the whole offspring in the next iteration. This procedure does not operate on a concrete population type, but rather it accepts a generic object which models a C++ *Sequence*. That is, it provides a pair of iterators to the beginning and the end of the sequence, whose elements are comparable by a less-than relation. The latter requirement is naturally fulfilled by the `Individual` type since two individuals can easily be ordered by considering their fitness values.

The *best* procedure selects the fittest individual among the current population and the offspring and it possibly updates the incumbent solution. This function

belongs to this concept because logically only the actual type implementation of *UpdatePolicy* is aware of the arrangement of the individuals in the sequences, so it can find the best individual in the most efficient way.

3.2.6 SelectionPolicy.

The *SelectionPolicy* concept provides the interface for components that implement a selection procedure.

```
Chromosome T::select(Sequence& population)
```

The *select* function is mainly invoked during the offspring construction phase in which genetic operators are applied on the fittest individuals. As for *UpdatePolicy* procedures, *select* accepts an object conforming to the *Sequence* concept. In the current version of `EasyGenetic`, roulette-wheel and k-way tournament selections are already provided as built-in selection rules.

3.3 CrossoverOperator and MutationOperator.

Finally, there are the *CrossoverOperator* and *MutationOperator* concepts.

```
std::pair<Chromosome, Chromosome>
  T::crossover(Chromosome& parent1, Chromosome& parent2) // crossover
T::mutation(Chromosome& c) // in-place mutation
```

These interfaces are particularly difficult to define because of the great variety of genetic operators that have been defined. While dealing with different chromosome types is not an issue and it is fully resolved by generic programming, the different genetic operators feature also various calling conventions, making a generic approach more problematic. As an example, the classical crossover operator produces a new chromosome from two parents, but it is also possible to define a three-way crossover that spawns two new descendants. If we model these two crossover operations through method calls, the first would map to a method with two arguments that returns a single value, while the second would be a method with three arguments that returns a pair of values. In this situation, generic programming hardly help because it only deals with types. In order to avoid over-generalization and to maintain the principle of keeping things simple and efficient, during the design of `EasyGenetic` we choose to take a radical approach, that is we allow only two-parents crossovers returning a pair of descendants and only mutations on a single chromosome.

4 Case-Studies

4.1 Haplotype inference

Haplotype Inference is a challenging problem in bioinformatics that consists in inferring the basic genetic constitution of diploid organisms on the basis of their genotypes, thus labelling which genes are inherited along the maternal or paternal line. This piece of information allows to perform association studies for the genetic variants involved in multifactorial diseases and the individual responses to therapeutic agents. A notable approach to the problem is to encode it as a combinatorial problem (under certain hypotheses, such as the *pure parsimony* criterion) and to solve it using combinatorial optimization techniques.

We employ `EasyGenetic` to rapidly implement an effective solver for the Haplotype Inference Problem based on a well-known constructive procedure. More details on the problem and on different solution techniques can be found in [9, 5]; moreover, in [4], an embryonal version of the principles of `EasyGenetic` has been first presented.

In the Haplotype Inference Problem we deal with *genotypes*, i.e., strings of length m that correspond to chromosomes with m sites. Each value in the string belongs to the alphabet $\{0, 1, 2\}$. A position in the genotype is associated with a site of interest on the chromosome and it has value 0 (wild type) or 1 (mutant) if the corresponding chromosome site is a homozygous site (i.e., it has that state on both copies) or the value 2 if the chromosome site is heterozygous. A *haplotype* is a string of length m that corresponds to only one copy of the chromosome (in diploid organisms) and whose positions can assume the symbols 0 or 1. Given a chromosome g, we say that the unordered pair $\langle h, k\rangle$ *resolves* g, and we write $\langle h, k\rangle \triangleright g$ if the following conditions hold (for $j = 1, \ldots, m$):

$$g[j] = 0 \Rightarrow \quad h[j] = 0 \wedge k[j] = 0 \tag{1}$$

$$g[j] = 1 \Rightarrow \quad h[j] = 1 \wedge k[j] = 1 \tag{2}$$

$$g[j] = 2 \Rightarrow \quad (h[j] = 0 \wedge k[j] = 1) \vee (h[j] = 1 \wedge k[j] = 0) \tag{3}$$

Observe that, according to the definition, for a single genotype string the haplotype values at a given site are predetermined in the case of homozygous sites, whereas there is a freedom to choose between two possibilities at heterozygous places. This means that for a genotype string with l heterozygous sites there are 2^{l-1} possible pairs of haplotypes that resolve it.

Given a population of n genotypes, the *Haplotype Inference Problem* is the problem of finding a set of n pairs of (not necessarily distinct) haplotypes $\phi = \{\langle h_1, k_1\rangle, \ldots, \langle h_n, k_n\rangle\}$, so that $\langle h_i, k_i\rangle \triangleright g_i, i = 1, \ldots, n$. From the mathematical point of view, there are many possibilities for building the set of haplotypes, since

there is an exponential number of possible haplotypes for each genotype. Therefore, a criterion has to be added to the model for evaluating the solution quality. One natural model of the Haplotype Inference Problem is the *pure parsimony* approach that consists in searching for a solution that minimizes the total number of distinct haplotypes (the problem is APX-hard [11]).

4.1.1 The slave constructive procedure.

The Haplotype Inference Problem definition makes constructive procedures very appealing. Indeed, a constructive procedure can incrementally build a set H of haplotypes which, taken in pairs, resolve the genotypes. Such a procedure can start from an empty set and add one or two haplotypes at a time, while it scans the set of genotypes G. The objective is to build H as small as possible, i.e., to find a minimal cardinality set of haplotypes that composes the phasing. To this aim, new haplotypes should be added to H only when necessary, i.e., when no pair of haplotypes already in H resolves the current genotype g. In principle, an optimal solution could be found if an oracle were to indicate the right order of visiting the genotypes and the right starting pair of haplotypes, along with properly defined criteria for choosing the values to assign. This is in general not possible, but an iterative and adaptive search strategy could be very effective in exploring these possibilities.

In the context of this problem, the slave algorithm can be, in general, any constructive procedure that is fed with an initial set of resolving haplotypes and with some criteria to complete the solution. The procedure we will describe is variations of a constructive procedure known as *Clark's rule* [8]. Clark's inference rule exploits the property of complementarity[3] between a genotype and a haplotype. The procedure works as follows:

1. Let $G' \subseteq G$ be a subset of genotypes with zero or one ambiguous site only. From G', an initial set of haplotypes H explaining the genotypes in G' can be inferred by complementarity. Be $G \leftarrow G'$.

2. Choose a pair (g, h) $g \in G, h \in H \,|\, h$ resolves g. If such pair exists, set $H \leftarrow H \cup \{h' | h'$ is the complement of h with respect to $g\}$ and $G \leftarrow G \setminus \{g\}$.

3. Iterate step 2 until:

   - $G = \emptyset$: in that case H is a solution to the problem;
   - Cannot find a pair (g, h): in that case the algorithm fails.

Although the procedure is fast and simple, it has some drawbacks. First of all, it needs an initial haplotype set to "bootstrap", which might be impossible to obtain. In

[3]It is possible to show that given a genotype g and a haplotype h resolving g there exists a unique complementary haplotype h' such that $\langle h, h' \rangle \triangleright g$

fact, in non-trivial instances the presence of non-ambiguous genotypes or containing one heterozygous site is quite unlikely.

Secondly, the algorithm must make an arbitrary choice in step 2 because here there are two sources of non-determinism:

1. The set of genotypes that can be solved by a haplotype in the current set H can have cardinality grater than one;

2. The genotype chosen could be solvable by more than one haplotype in H.

As a consequence, the quality of the solution returned is heavily dependent on the order in which genotypes are explained and haplotypes selected at each step. Nevertheless, the application of Clark's rule to Haplotype Inference Problem is appealing because it naturally tries to re-use haplotypes in the current partial solution to explain remaining genotypes. It is reasonable to think that some haplotypes in an optimal solution resolve many genotypes. The main idea is thus to employ a learning procedure, in the form of a population-based metaheuristic, to guide the non-deterministic choices made in step 2; therefore, such a procedure has to learn an optimal genotype resolution order and some criteria to choose the most promising haplotype when asked to resolve a genotype.

We implemented a slightly modified Clark's rule, since in its original form it may not find a solution at all given a genotype ordering. Each time the algorithm cannot find a compatible haplotype for the current genotype g, it randomly select a haplotype pair for g and then continues. This way we ensure to find a complete solution to the problem. The returned result is the best out of a set number of independent runs.

4.1.2 Results.

We compared the modified Clark rule, 400 iterations, (labelled `rule`) against the genetic master-slave (labelled `ga`) with the following parameters: population and offspring size of 100 individuals, 500 maximum iterations, 100 idle iterations; every new descendant has been mutated exactly once.

In Figure 4.1.2 we plot the results of the two algorithms. Each sub-figure refers to a single instance set. We run each algorithm 10 times on each instance and report on the y-axis the sum of the cardinalities found by the algorithm over all the instances of a particular instance set. The results clearly show that the master-slave algorithm outperforms the simple Clark rule.

4.2 Capacitated Symmetric Vehicle Routing Problem

As a second case study, we show the development of a solver for the Capacitated Symmetric Vehicle Routing Problem (CVRP). This problem is highly relevant in

operations research and logistics and it is known to be a hard combinatorial optimization problem [3]. The VRP requires finding a set of routes, each of which assigned to a vehicle, such that each customer is visited and a cost function minimized (e.g., the total travel length). In the capacitated version of the problem a constraint on the maximum capacity of each vehicle is enforced. Usually, besides a cost function, also the number of vehicles composing the fleet is minimized.

4.2.1 The slave constructive procedure.

Our master-slave algorithm is a generalization of the Clarke-Wright saving heuristic proposed in [2]. The main components of a Clarke-Wright heuristic are a list of pairs of customers, called *saving list*, and a deterministic constructive procedure which, starting from an infeasible solution made of one route for each client node, iteratively picks the first pair of customers from the list and merges two routes if the selected client nodes are the endpoints of two different routes. The key idea is thus to order the saving list according to a heuristic function, called *saving function*, on the pairs of customers. The saving function is crucial because it determines the ordering of the elements in the saving list. In [2] has been proposed a function which takes into account three different components weighted by three coefficients in the cube $[0, 2]^3$. An optimal parameter configuration has been found through repeated evaluation of the algorithm.

It is easy now to recognize the main ingredients for a master-slave algorithm. From a this perspective, a saving list is but an input to a deterministic slave procedure which returns a set of routes.

Our algorithm starts from a initial population generated in the following manner: for each individual we sampled uniformly at random a point in the cube $[0, 2]^3$ and computed a saving list using those parameters. The slave procedure is *the same* as in [2]. Results show that using a master-slave approach produces sizable improvements over Altinel and Öncan's heuristic.

4.2.2 Results.

We compare two instances of our algorithm against the heuristic proposed in [2] on the same set of problem instances. The two instances differ only in the parameters configuration and have been selected among a number of variations after a thorough statistical analysis whose result we omit because lack of space.[4] Both instances use a population and offspring size of 200, while mutation rate has been set to 20% for `CVRP-20` and 30% for `CVRP-30`.

[4] Complete results are available on authors' website.

4.3 `EasyGenetic` solver instantiation

We now show the flexibility of `EasyGenetic` framework by showing how easily is to write instantiation code.

First we instantiate, in a problem specific, way a class conforming to *ProblemModel* concept. `Params` represents a type that hold genetic algorithm parameters, such population size, and is typically initialized by parsing the command line arguments.

```
typedef /*...*/ problem_model;
typedef /*...*/ Params;
problem_model model(/*...*/);
```

Then we setup the actual solver by specifying its template parameters; we refer to (3.1.2) for an explanation of the Policy Class idiom. Here we specify in order: what is the problem model, the selection procedure, the termination condition, the update procedure. Finally, we instantiate the solver supplying the model and the parameters.

```
typedef Solver<problem_model,
  roulette_wheel, // choose which selection scheme you prefer
  either<max_iterations<2000>, // state your termination condition
  max_idle_iterations<2000> >, // maxIterations < 2000 OR maxIdleIterations < 2000
  SteadyState // update procedure
  > Solver;
Solver solver(model, Params(/*...*/));
```

To run the solver and produce the best `Individual`, we invoke the following method.

```
Individual result = solver.solve();
```

And finally we transform the best individual in a solution using a problem-specific way, in fact *computeSolution* method is not part of *ProblemModel* concept.

```
problem_model::solution_type sol = model.computeSolution(result);
```

It is worth noting that, thanks to generic programming, the only changes to make in order to switch to a different problem are confined in the definition of `problem_model` type in the first typedef and the invocation of *computeSolution* that translate an `Individual` in an actual solution.

5 Conclusion and future work

In this paper, we have presented `EasyGenetic`, a genetic master-slave framework based on metaprogramming. The usage of templates makes it possible to combine flexibility —as the user can easily configure the solver— and efficiency —as compiled

code is optimized. These properties have been showed in two case- studies, in which `EasyGenetic` has been used to solve Haplotype Inference Problem and Capacitated Vehicle Routing Problem respectively.

The framework is in its early development stage and more features can be added. For example, the implementation of more genetic operators can be added. Furthermore, `EasyGenetic` can be extended with general features such as *observers*, which can be used to trace and monitor the execution of the algorithm. Moreover, an integration with the pre-existing frameworks for developing and analyzing stochastic local search algorithms, EasyLocal++ and EasyAnalyzer, is planned.

Acknowledgements

We thank Maria Battarra for her help, support and collaboration on the application of `EasyGenetic` to the CVRP.

References

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, February 2001.

[2] I.K. Altinel and T. Öncan. A new enhancement of the Clarke and Wright savings heuristic for the capacitated vehicle routing problem. *Journal of the Operational Research Society*, 56:954–961(8), August 2005.

[3] M. Battarra, B. Golden, and D. Vigo. Tuning a parametric Clarke–Wright heuristic via a genetic algorithm. *Journal of the Operations Research Society*, 59:1568–1572, 2008.

[4] S. Benedettini, L. Di Gaspero, and A. Roli. Genetic master-slave algorithm for haplotype inference by parsimony. Technical Report DEIS-LIA-09-003, University of Bologna (Italy), January 2009. LIA Series no. 93.

[5] S. Benedettini, A. Roli, and L. Di Gaspero. Two-level ACO for haplotype inference under pure parsimony. In *Ant Colony Optimization and Swarm Intelligence, 6th International Workshop, ANTS 2008*, volume 5217 of *Lecture Notes in Computer Science*. Springer–Verlag, 2008.

[6] Boost C++ libraries. `http://www.boost.org/`. Viewed: April 2009.

[7] S. Cahon, N. Melab, and E-G. Talbi. ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

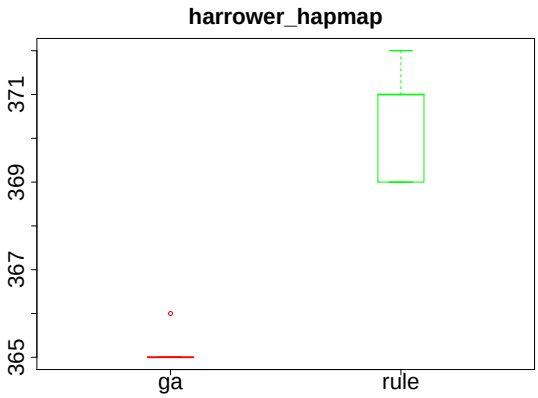
[8] A.G. Clark. Inference of haplotypes from PCR-amplified samples of diploid populations. *Molecular Biology and Evolution*, 7:111–122, 1990.

[9] L. Di Gaspero and A. Roli. Stochastic local search for large-scale instances of the haplotype inference problem by pure parsimony. *Journal of Algorithms: Algorithms in Logic, Informatics and Cognition*, 63(1–3), 2008.

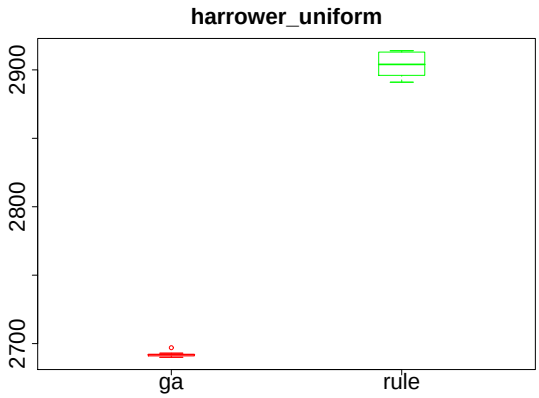
[10] EClab. `http://cs.gmu.edu/~eclab/`. Viewed: April 2009.

[11] G. Lancia, M.C. Pinotti, and R. Rizzi. Haplotyping populations by pure parsimony: Complexity of exact and approximation algorithms. *INFORMS Journal on Computing*, 16(4):348–359, 2004.

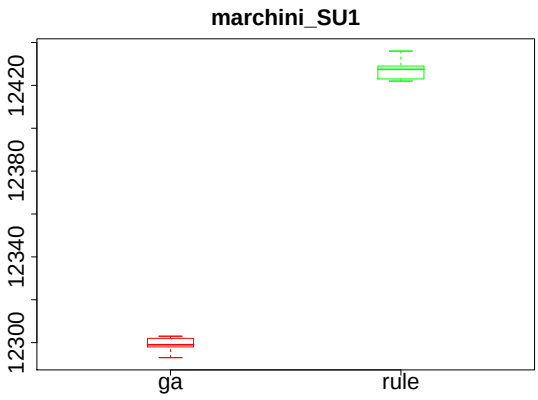
[12] Standard Template Library. `http://www.sgi.com/tech/stl/`. Viewed: April 2009.



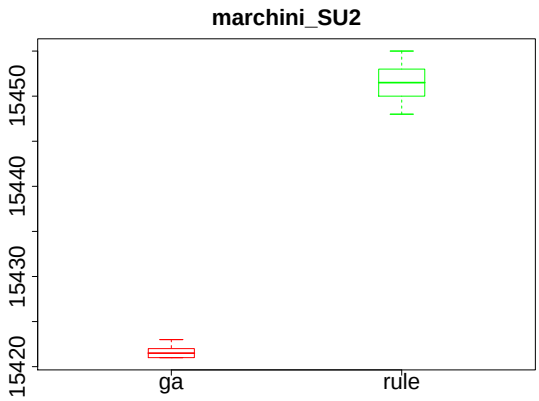
(a) Harrower HapMap



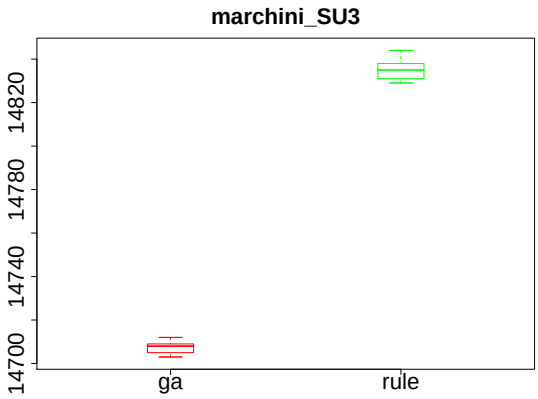
(b) Harrower Unfiorm



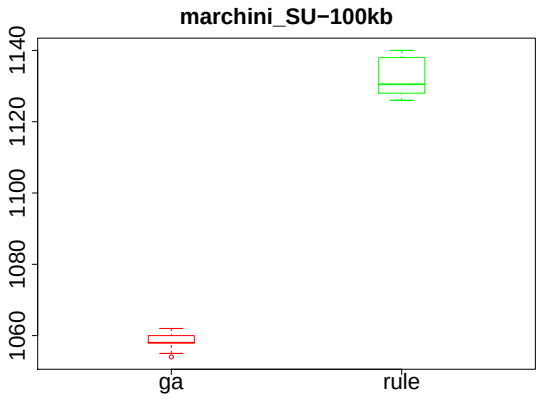
(c) Marchini SU1



(d) Marchini SU2

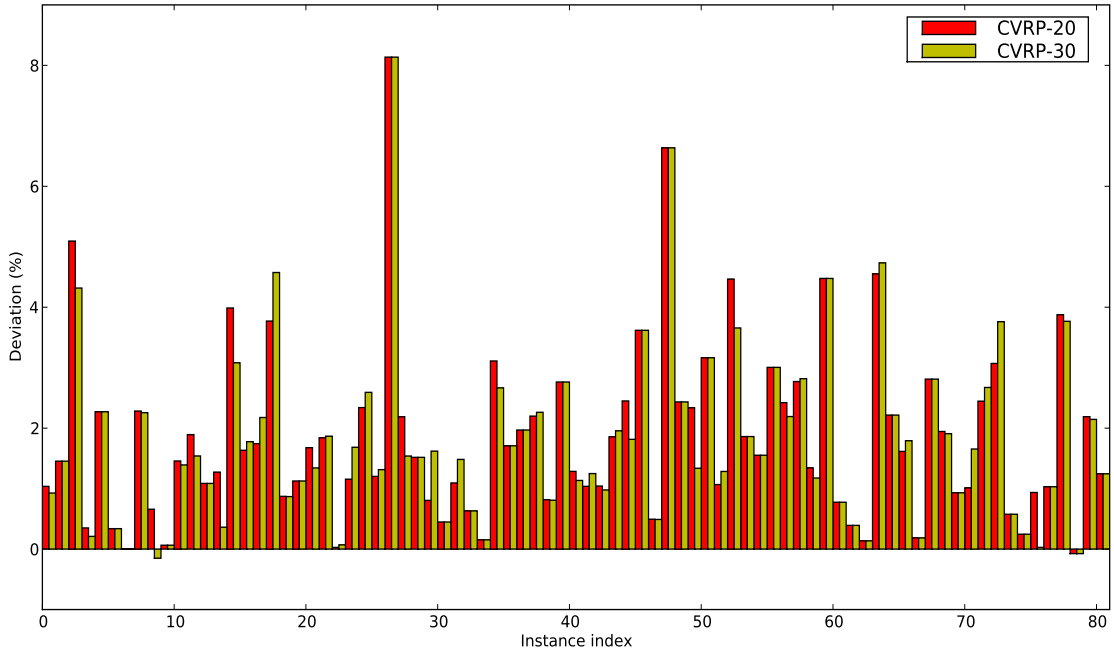


(e) Marchini SU3

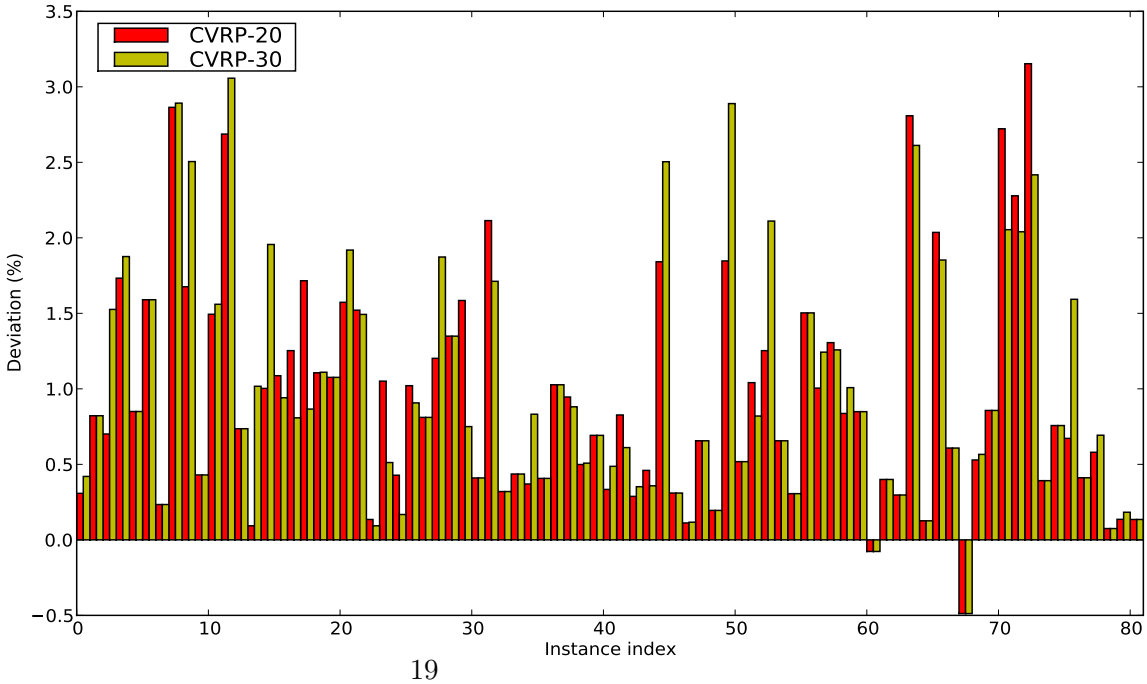


(f) Marchini SU-100kb

Figure 4: Comparison between Clark's rule solver (RULE) and its application into `EasyGenetic` (the resulting algorithm is denoted by GA). Results are shown on various `HIP` benchmarks.



(a) Deviations against Altinel-Öncan



(b) Deviations against best known

Figure 5: Results our genetic master-slave solver on hard `CVRP` common benchmark instances. In Figure (a), the percental deviation between our algorithm and Altinel-Öncan's one is plotted. In Figure (b), a comparison against the best known results is shown.