A FRAMEWORK FOR MEMETIC ALGORITHMS

Phan Anh Tuan, Duong Tuan Anh
University of Technology, VNU-HCM

ABSTRACT: *Memetic algorithm, a combination of genetic algorithm with local search, is one of the most successful metaheuristics to solve complex combinatorial optimization problems. In this paper, we will introduce an object-oriented framework which allows the construction of memetic algorithms with a maximum reuse. This framework has been developed in Java using design patterns to allow its easy extension and utilization in different problem domains. Our framework has been experimented through the development of a memetic algorithm for solving set covering problems.*

Keywords: *memetic algorithm, genetic algorithm, local search, object-oriented framework, set-covering problem.*

1. INTRODUCTION

Memetic algorithms (MAs), introduced by P. Moscato ([7],[8]), are genetic algorithms that apply a separate local search process to refine individuals (e.g. improve their fitness by hill climbing). MAs represent one of the most successful emerging metaheuristics in the ongoing research effort to solve effectively NP-complete combinatorial optimization problems ([1], [6], [8]). From an optimization point of view, MAs are hybrid genetic algorithms that combine global and local search by using a genetic algorithm to perform exploration while the local search method performs exploitation.

In this paper, we introduce an object-oriented framework for developing MA applications. Such a framework is very useful and effective in designing and implementing a memetic algorithm rapidly, since it aims to achieve a high degree of design reuse. The framework is a set of interrelated classes that embody an abstract design for solutions to combinatorial optimization problems using MA. The framework establishes a reference application architecture ("skeleton"), providing not only reusable software elements but also some type of reuse of architecture and design patterns, which may simplify application development considerably. It is not the main objective of the framework to provide with a fast "running system" but with a fast "design and implementation system". This framework will serve object-oriented programmers and evolutional computation developers that want to reuse not only at the code level but also at the design level.

The basic idea of the framework is to capture the essential features of most MA techniques and their possible compositions. The user's application is obtained by writing derived classes for the selected subset of the framework classes. Such user-defined classes contain only the specific problem description, but no control information for the algorithms.

The organization of the paper is as follows. Section 2 describes the framework for MAs, including the details of the components in the framework architecture. Section 3 introduces briefly an actual application of the framework: the development of a MA for solving the set covering problem. Section 4 gives some discussion on related works. Conclusion and future works are given in Section 5.

2. FRAMEWORK FOR MEMETIC ALGORITHMS

2.1 Memetic Algorithm

A general schema of a memetic algorithm is given in Figure 1.

```
procedure MA
begin
  for j = 1 to popsize do  // popsize: size of population P
    i = generateSolution();  // an initial solution
    i = LocalSearch(i); add individual i to P;
  endfor;
  repeat
    for j = 1 to #recombinations do
       select two parents $i_a$, $i_b$ $\in$ P randomly;
       ic = Recombine($i_a$, $i_b$);
       ic = LocalSearch($i_c$); add individual ic to P;
    endfor;
    for j = 1 to #mutations do
       select an individual i $\in$ P randomly;
       $i_m$ = Mutate(i);
       $i_m$ = LocalSearch($i_m$);  add individual $i_m$ to P;
    endfor;
    P = select(P);
  until terminate = true;
end;
```

Figure 1. General schema of a memetic algorithm

2.2 Background on Design Patterns and Object-Oriented Framework

Design patterns are abstract structures of classes that are commonly presented in object-oriented applications and that have been precisely identified and classified. Design patterns 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 designer can reuse not only "computer code" but also basically "designs". A software framework is a reusable architecture, which provides the skeleton and basic behavior of a certain kind of software product. In general, a framework is a collection of tightly related classes whose interrelation is usually given by the design patterns that the framework implements. The user needs only to define suitable derived classes, which implement the operations of the abstract classes in the framework. The framework relies extensively on the three following design patterns from Gamma et al. 's book [4]:

- **Abstract Factory** which provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Strategy** defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- **Template Method** defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithms.

2.3 Architecture of the Framework for Memetic Algorithms

In Figure. 2 we show the main components of the framework, using UML notation. If a class B is connected to class A by a line with an open triangle pointing toward A, then B is derived from A. If class A has a filled arrow to class B, then an instance of class A "has" one or more instances of an object of class B. The main design patterns used in this architecture are Template Method, Abstract Factory, and Strategy.

Template Method

The Template Method patterns provide with a general-purpose algorithm which can be easily tailored to different situations. A component of the framework using this design pattern is illustrated in Figure 3. In the figure, *AbstractMemeticAlgorithm* is an abstract class providing the general memetic algorithm with several necessary operations: *localSearch(...)* for local search step, *mutate(...)* for mutation operator, *crossover(...)* for crossover operator, *parentSelection(...)* for parent selection strategy, *selectNextGeneration(...)* for survivor selection strategy.

The derived class *AbstractMA* inherits the class *AbstractMemeticAlgorithm* will hook into the process of memetic algorithm by overriding all these above-mentioned operations. *AbstractMA* overrides all the methods of the class *AbstractMemeticAlgorithm*: *localSearch(...), mutate(...), crossover(...), parentSelection(...), selectNextGeneration(...), initializePopulation(...)*.

AbstractMA is still an abstract class since it has not implemented the *stoppingCondition()* method.

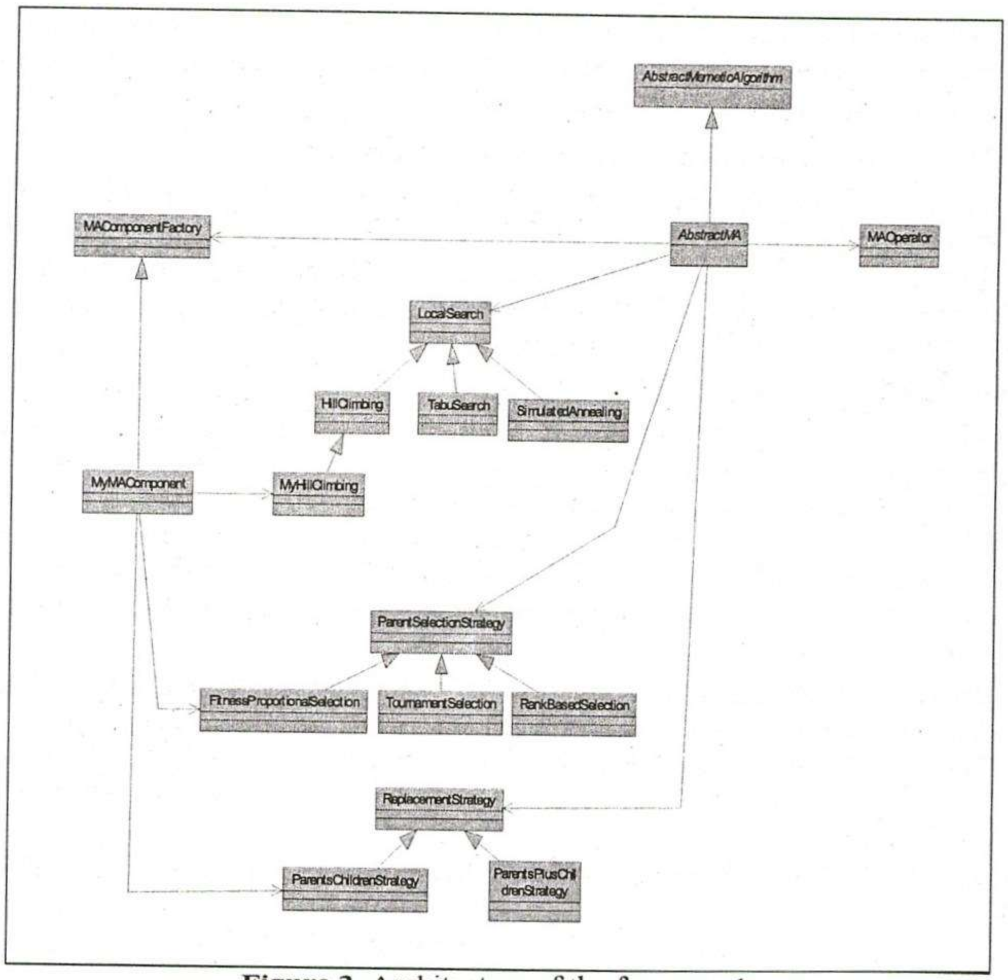

Figure 2. Architecture of the framework

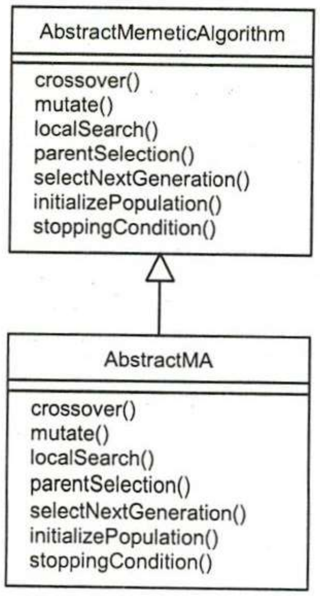

Figure 3. A component in the framework using Template Method pattern

Abstract Factory

The class *MAComponentFactory* implements an Abstract Factory pattern. It defines an interface to create objects in the three classes: *LocalSearch*, *ParentSelectionStrategy*, and *ReplacementStrategy*.

The classes *LocalSearch*, *ParentSelectionStrategy* and *ReplacementStrategy*, which are abstract, define an interface for local search step in the memetic algorithm, an interface for parent selection strategy and an interface for survivor selection strategy, respectively.

There are several sorts of local search objects. *HillClimbing, TabuSearch,* and *SimulatedAnnealing* are the subclasses derived from the *LocalSearch* class and each of these objects will know how to implement a specific local search technique selected by the user. The specific local search method can be hill climbing, Tabu search, or simulated annealing. *MyHillClimbing* is a subclass that is derived from the *HillClimbing* class and implements the hill climbing local search method. This class will be completed by the user and it overrides the necessary methods.

There exist subclasses of the abstract class *ParentSelectionStrategy* that will create the appropriate instance of a parent selection object: *FitnessPropotionalSelection, TournamentSelection* and *RankBasedSelection.*

There exist subclasses of the abstract class *ReplacementStrategy* that will create the appropriate instance of replacement selection object: *ParentsChildrenStrategy* and *ParentsPlusChildrenStrategy.* These classes can implement one of the following survivor selection strategies. (1) *ParentsChildrenStrategy* implements the survivor selection strategy (η, λ) in which η parents will be replaced by λ best children. (2) *ParentsPlusChildrenStrategy* implement the survivor selection strategy ($\eta+\lambda$) in which η best individuals will be selected from the population consisting of η parents and λ children.

AbstractMA plays the role of a client who uses the interfaces *MAComponentFactory*, *LocalSearch*, *ParentSelectionStrategy*, and *ReplacementStrategy*. These interfaces will be used in implementing the methods *localSearch(...), mutate(...), crossover(...), parentSelection(...), selectNextGeneration(...)* and *initializePopulation(...)* of the class *AbstractMA*.

To develop a memetic algorithm, the user has to define a concrete class *MyMA* derived from the class *AbstractMA* (or the class *AbstractMemeticAlgorithm*) and implement the necessary methods. For example, if *MyMA* inherits from the class *AbstractMA*, the user needs to implement the method *stoppingCondition(...)* to specify the terminating condition of the memetic algorithm and the method *createMAAbstractFactory(...)* to create an *AbstractFactory* object of the class *MAComponentFactory.*

MAOperator provides the interface for the class *AbstractMA*. It implements the operations *mutate(...), crossover(...)*, etc. It provides the interface for implementing the basic genetic operators: one-point crossover, two-point crossover, uniform crossover and mutation.

Strategy

Some examples of the use of *Strategy* pattern in our framework are described as follows. The classes *HillClimbing, TabuSearch,* and *SimulatedAnnealing* have been factored into the class *LocalSearch,* as already described in the above subsection Abstract Factory. The classes *FitnessProportionalSelection, TournamentSelection*, and *RankBasedSelection* have been factored into the class *ParentSelectionStrategy.*

The main data representations used in the framework are *population* and *individual*. The class *Population* represents the population and the class *Individual* represents the individuals in a population of the memetic algorithm (see Figure 4 and Figure 5).

The class *Individual* provides the interface for implementing an individual. The class consists of only one abstract operation *objectiveFunction(...)* that will be completed by the user later. The *objectiveFunction(...)* is the objective function defined in the problem under consideration. The instantiation of an *Individual* object is through an *IndividualFactory* object which plays the role of an *AbstractFactory*. For example, in Figure 7 the user defines the class *MyIndividualFactory*, which is derived from the class *IndividualFactory*, to instantiate a *MyIndividual* object. The class *MyIndividual* is a concrete class derived from the class *Individual*.

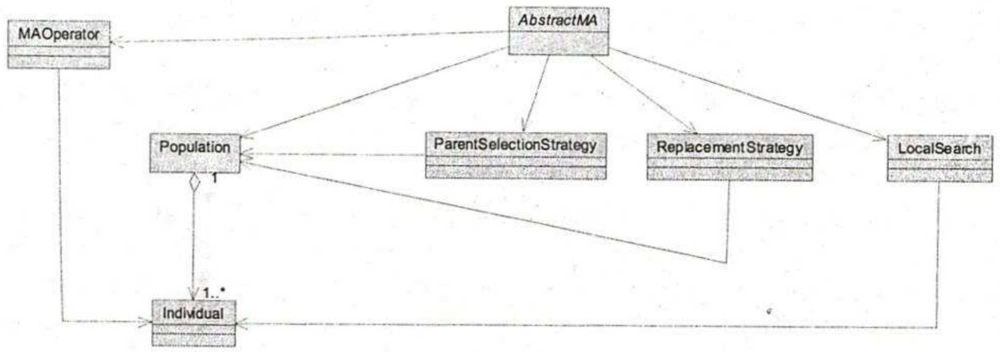

Figure 4. Some other components of the framework

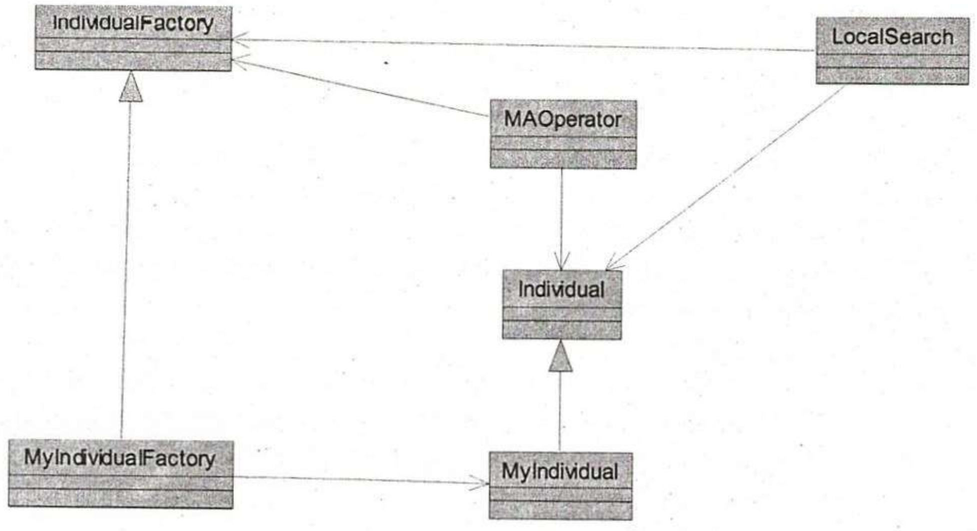

Figure 5. Some other components of the framework

3. AN APPLICATION OF THE FRAMEWORK: THE SET COVERING PROBLEMS

In this section, we present the application of the framework for developing a MA to solve one well-known combinational optimization problem: set covering problem.

The set covering problem (SCP) is the problem of covering the rows of a m row, n column, zero-one matrix (a_{ij}), by a subset of the columns at minimum cost. Defining $x_j = 1$ if column j (with cost $c_j > 0$) is in the solution and $x_j = 0$ otherwise, the SCP is

$$\text{Minimize} \quad \sum_{j=1}^{n} c_j x_j \tag{1}$$

$$\text{Subject to} \quad \sum_{j=1}^{n} a_{ij} x_j \geq 1, \qquad i = 1, \ldots, m \tag{2}$$

$$x_j \in \{0,1\} \qquad j = 1, \ldots, n \tag{3}$$

Equation (2) ensures that each row is covered by at least one column and equation (3) is the integrality constraint. If all the cost coefficients c_j are equal, the problem is called the unicost SCP. The SCP is NP-complete. It has many practical applications including crew scheduling, location of emergency facilities, assembly line balancing and boolean expression simplification.

3.1 Memetic Algorithm for Set Covering Problems

Representation and Fitness Function

We use a n-bit binary string as the chromosome structure where n is the number of columns in the SCP. A value of 1 for the i-th bit implies that the column i is in the solution. The fitness of an individual is directly related to its objective function value. With the binary representation, the fitness f_i of an individual i is calculated simply by

$$f_i = \sum_{j=1}^{n} c_j s_{ij} \tag{4}$$

where s_{ij} is the value of the j-th bit (column) in the string corresponding to the i-th individual and c_j is the cost of bit (column) j.

Parent Selection Technique

We choose the *tournament selection* method. It works by forming two pools of individuals, each consisting of T individuals drawn from the population randomly. Two individuals with the best fitness, each taken from one of the two tournament pools, are chosen for mating. Here we apply the *binary* tournament selection (i.e. $T = 2$) as the method for parent selection.

Population Replacement model

Once a new feasible child has been generated, the child will replace a randomly chosen member in the population. This type of replacement method is called *incremental replacement* or *steady-state replacement*.

Variable Mutation Rate

We select to utilize a *variable mutation rate* rather than a fixed, small mutation rate and this mutation rate should depend on the rate at which the MA converged.

Heuristic Feasibility Operator

The solutions generated by the genetic operators may violate the problem constraints (i.e. some rows may not be covered). To make all solution feasible, one additional operator is needed. We apply the heuristic feasibility operator proposed by Beasley et al. [3]. This operator involves the identification of all uncovered rows and the addition of columns such that all rows are covered. The steps are listed as follows:

Let

I = the set of all rows

J = the ser of all columns

α_i = the set of columns that covers row i, $i \in I$

β_j = the set of rows covered by column j, $j \in J$

S = the set of columns in a solution

U = the set of uncovered rows

w_i = the number of columns that cover row i, $i \in I$ in S

```
1. Initialize $w_i = | S \cap \alpha_i |$, $\forall\ i \in I$.
2. Initialize $U = \{ i \mid w_i = 0, \forall\ i \in I\}$.
3. For each row $i$ in $U$ (in increasing order of $i$ }
    (a) find the first column $j$ (in increasing order of $j$) in that minimizes
        $c_j/|U \cap \beta_j|$,
    (b) add $j$ to $S$ and set $w_i = w_i + 1$, $\forall i \in \beta_j$. Set $U = U - \beta_j$.
```

Step 1 and 2 identify the uncovered rows. Step 3 is a heuristic in the sense that columns with low cost-ratios are being considered first.

Hill-Climbing Operator

We use a hill climbing operator as a local search technique in our MA for SCP. The hill climbing operator is as follows:

For each column j in S (sorted in decreasing column cost), if $w_i \geq 2$, $\forall i \in \beta_j$ then set $S = S - j$ and set $w_i = w_i - 1$, $\forall i \in \beta_j$. Now S is a feasible solution that contains no redundant columns.

Generating Initial Population

To generate initial population, we also use a method proposed by Beasley et al. [3]. Each of the initial solutions is generated randomly and then it is made feasible by eliminating redundant columns by a method similar to that used in the heuristic feasibility operator except that the redundant columns are dropped in a random manner rather than by cost.

To summarize our memetic algorithm for the SCP, the following steps are used:

1 – Generate randomly an initial population with a given size N.

2 – Select two parents P_1 and P_2 from the population using the binary tournament selection.

3 – Recombine (crossover) the parents P_1 and P_2 to produce an offspring C.

4 – Mutate k genes randomly selected from an offspring C where k is determined by a variable mutation schedule.

5 – Perform local search on the offspring C to improve its fitness.

6 – Repeat the steps 2-5 until the fixed number of offspring is reached.

7 – Select individuals for the next generation using $(\eta+\lambda)$ replacement strategy.

8 – If the population incurs premature convergence, restart another initial population.

9 – Repeat the steps 2-5 until the termination condition is satisfied. The best solution is the solution with the largest fitness value in the population.

3.2 Code Example

Let us look at a piece of code needed to run an application of the framework to solve the SCP using MA.

The framework has implemented for us all the main parts of the MA. Therefore we do not need to code the whole MA for SCP from the scratch. We have to implement just some concrete classes derived from the abstract classes necessary for the algorithm, including the class *HillClimbingFSCP* which implements the concrete local search step in the MA for SCP. Here, the concrete classes consists of the following:

- The class *IndividualSCP*, derived from the abstract class *Individual*, represents concrete individuals in the MA for SCP.
- The class *MASCPFactory*, derived from the abstract class *MAComponentsFactory*, helps to create the *LocalSearch*, *ParentSelectionStrategy*, and *ReplacementStrategy* objects.
- The class *Individual_SCP_Factory*, derived from the abstract class *IndividualFactory* helps to create the *Individual* objects.
- The class *MA_SCP*, derived from the abstract class *AbstractMA*, includes the code which is modified to incorporate the heuristic used for generating only feasible initial solutions and some heuristics used for the mutation operator in the MA for SCP.

The main Java code for running the main program is given as follows:

```
import java.util;
public class Test {
  public static void main(String []argv)
  {
int numbersOfCol = 1000;
int iNumbersOfChildrenAtEachGeneration = 100;
int sizeOfPopulation = 100;
IndividualFactory individualFactory = new
Individual_SCP_Factory(numbersOfCol);
AbstractMA  maSCP = new MA_SCP(sizeOfPopulation,
iNumbersOfChildrenAtEachGeneration, individualFactory);
maSCP.setMutationProb(mutationProb);
    maSCP.setMutationRate(mutationRate);
    maSCP.setRestartMutationProb(mutationProbRestart);
    maSCP.setRestartMutationRate(mutationRateRestart);
    maSCP.start();
    Individual indiv = maSCP.getOptimalIndividual();
  }
}
```

Some variables used in the above code are explained as follows. Let *numbersOfCol* denote the number of columns in the SCP under consideration, *sizeOfPopulation* the population size and *iNumbersOfChildrenAtEachGeneration* the number of offspring generated in each generation, respectively.

Chromosome encoding is implemented through an *individualFactory* object, which is instantiated at line 8. The memetic algorithm for SCP is executed through the *maSCP* object belonging to the abstract class *AbstractMA*. The concrete class corresponding to the abstract class is *MA_SCP*.

Lines 10, 11 set the mutation probability, the mutation rate for the memetic algorithm when the population has not converged. Lines 12, 13 reset the mutation probability for the memetic algorithm when the population has already converged. Here we apply the variable mutation rate technique as mentioned in subsection 3.1.

3.3 Experiments

The experiments of the MA for set covering problems using the framework have been done on the PC Pentium III 700Mhz, 256 MRAM. All the benchmark problems for SCP were obtained electronically from the OR-Library [2], available at the website: http://people.brunel.ac.uk/~mastjjb/jeb/jeb.html.

We have tested 15 benchmark problems downloaded from the website, namely *scp41*, *scp42, scp43, scp44, scp45, scp46, scp47, scp48, scp49, scp61, scp62, scp63, scp64*, and *scp65*. For each problem, we have performed 10 runs. Runs are terminated whenever the expected number of generations has reached. The experiments show that the MA can find near optimal solutions for the benchmark SCP instances in not more than 200 generations. This number of generations is much less than that required by pure genetic algorithms in order to obtain the same high quality solutions of the same SCPs. (For more details about experimental results, readers can refer to [9]). Note that this framework serves also to develop pure genetic algorithms where no local search is involved.

We have developed two versions of the MA algorithm for SCP: the MA implementation for SCP without using the framework and the one that utilizes the framework. The former is composed of 674 lines of Java code whereas the latter consists of only 123 lines. This confirms that the user can save a lot of effort when using the framework to develop a MA application in comparison to what is required for design and implementing a MA application from the scratch. The framework exhibits several advantages, not only in terms of code reuse by also in methodology and conceptual clarity. However, the framework approach still has one potential drawback: the performance of the MA implementation which applies the framework may be slower than that of the MA developed from scratch.

4. DISCUSSION

A few of object-oriented frameworks for MA applications have been already developed and are described in literature, notably in [5] and [10].

The framework developed by Wu [10] is a C++ framework for MA algorithms. This framework is based on only two design patterns: Template Method and Strategy. It does not include the important classes that support parent selection strategy, population replacement strategy, and genetic operators and it focuses mainly to the logic of local search and memetic algorithm. In our framework, we use more design patterns: Abstract Factory, Template Method and, Strategy and offer more abstract classes in order that it requires less effort from the user to develop a MA application. In short, our framework aims at a higher level of abstraction than the Wu's framework.

The framework MAFRA by Krasnogor et al. [5] is a Java framework for MA algorithms. This framework is based on five design patterns: Abstract Factory, Factory, Template Method, Strategy, and Visitor and therefore offers more abstract classes than our framework does.

However, examining closer at the architecture of MAFRA, we feel difficult to know where and how a particular design pattern is utilized in the framework. In term of this aspect, our framework is clearer, more recognizable, and therefore easier to use.

5. CONCLUSIONS

In this paper, an object-oriented framework was presented to be used as a general tool for the development and implementation of MAs in Java. The basic idea of framework is to capture the essential features of most MA techniques and their possible compositions. The user's application is obtained by writing derived classes for a selected subset of the framework classes. Such user-defined classes contain only the specific problem description, but no control information for the algorithms.

As an example of actual use of the framework, we present the development of the MA for solving a well-known combinatorial optimization problem: set covering problem. The development of this application shows that it is very useful and effective to use the framework for solving a specific problem by MA.

One direction of our future works is to apply the framework in developing MA applications for several other problems such as Traveling Salesman Problem, Vertex Cover, Bin Packing, Set Partitioning and some typical timetabing problems.

MỘT KHUNG THỨC CHO GIẢI THUẬT MEMETIC

Phan Anh Tuấn, Dương Tuấn Anh
Trường Đại Học Bách Khoa, ĐHQG-HCM

TÓM TẮT: *Giải thuật Memetic, sự kết hợp giải thuật di truyền với tìm kiếm cục bộ, là một trong những siêu-heuristic khá mạnh để giải những bài toán tối ưu tổ hợp phức tạp. Trong bài báo này, chúng tôi giới thiệu một khung thức hướng đối tượng mà hỗ trợ cho việc xây dựng những giải thuật memetic với khả năng tái sử dụng tối đa. Khung thức này được phát triển bằng Java, sử dụng những mẫu thiết kế cho phép mở rộng dễ dàng và tiện dụng trong nhiều lãnh vực ứng dụng. Khung thức này đã được thử nghiệm qua việc xây dựng một giải thuật memetic để giải bài toán phủ tập.*

Từ khóa: *giải thuật memetic, giải thuật di truyền, tìm kiếm cục bộ, khung thức hướng đối tượng, bài toán phủ tập.*

REFERENCES

[1]. A. Alkan, E. Ozcan, Memetic Algorithms for Timetabling, *Proc. of IEEE Congress on Evolutionary Computation*, (2003), 1796-1802.

[2]. J.E.Beasley, OR-Library: distributing test problems by electronic mail, *Journal of the Operational Research Society*, 41:1069 – 1072, (1990).

[3]. J.E.Beasley, P. C. Chu, A Genetic Algorithm for the Set Covering Problems, *European Journal of Operational Research*, 94:392-404, (1996).

[4]. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements Of Reusable Object-Oriented Software, Addison-Wesley, (1995).

[5]. N. Krasnogor and J. Smith, MAFRA : A Java Memetic Algorithms Framework, Technical Report, Intelligent Computer System Centre, University of the West of England, Bristol, Untited Kingdom, (2000).

[6]. N. Krasnogor and J. Smith, A Memetic Algorithm With Self-Adaptive Local Search: TSP as a case study, Technical Report, Intelligent Computer System Centre, University of the West of England, Bristol, United Kingdom, (2000).

[7]. P. Moscato, On Evolution, Search, Optimization, Genetic Algorithms, and Martial Arts: Towards Memetic Algorithms, Technical Report 790, Caltech Concurrent Computation Program, California Institute of Technology, Pasadena, California, USA, (1989).

[8]. P. Moscato and C. Cotta, A Gentle Introduction to Memetic Algorithms, in F. Glover and G. A. Kochenberger (Eds.) *Handbook of Metaheuristics*, Kluwer Academic Publishers, (2003), 105 - 144.

[9]. P. A. Tuan, A Framework for Memetic Algorithms and an Application in Solving Set Covering Problems, Master Thesis, Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, (2007).

[10]. F. Wu, A Framework for Memetic Algorithms, Master of Science Thesis, University of Auckland, (2001).