EASYMETA: A FRAMEWORK OF METAHEURISTICS FOR MONO-OBJECTIVE OPTIMIZATION PROBLEMS

Rafael Augusto Ferreira do Carmo
Laboratório de Computação Natural e Inteligente (LACONI)
Universidade Estadual do Ceará (UECE)
Av. Paranjana, 1700 – 60.740-903 – Fortaleza - CE
carmorafer@gmail.com

Gustavo Augusto Lima de Campos
Laboratório de Computação Natural e Inteligente (LACONI)
Universidade Estadual do Ceará (UECE)
Av. Paranjana, 1700 – 60.740-903 – Fortaleza - CE
gustavo@larces.uece.br

Jerffeson Teixeira de Souza
Laboratório de Computação Natural e Inteligente (LACONI)
Universidade Estadual do Ceará (UECE)
Av. Paranjana, 1700 – 60.740-903 – Fortaleza - CE
jeff@larces.uece.br

ABSTRACT

The use of metaheuristics as a tool for solving optimization problems has become vastly spread in the most diverse computer science fields. This paper proposes a framework of metaheuristics, named EasyMeta, intended to solve mono-objective optimization problems, and focused on simplicity and usability. The framework is validated through a case study for the optimization version of the well-known Subset Sum problem.

KEYWORDS. Metaheuristics, Framework, Mono-objective optimization problems.

Metaheuristics
1. Introduction

1.1 Optimization Problems

Optimization Problems deal with the maximization or minimization of variable functions in a given set of restrictions on these variables (Raupp, 2003). The function to be maximized or minimized is called objective function, and may be linear or non-linear. Optimization is the process of seeking the best solution, also known as the optimum solution, from a set of feasible solutions to the instance of the problem. More formally, an optimization problem can be defined as:

- A set of variables \( X = x_1, x_2, \ldots, x_n \);
- An objective function \( f(X) \) to be maximize or minimized;
- A set of restrictions on the variables, where the equalities represent relations between these variables and the inequalities the limits of these variables.

An optimal solution for the optimization problem would be a set of values assigned to the variable \( x_1, x_2, \ldots, x_n \), which meets all restrictions and such that the value of the objective function \( f(X) \) is the best possible, i.e., the lowest or the highest value, depending on the type of problem. For minimization problems, a solution, \( s^* \), with the minimum value of the objective function should be sought, that is, \( f(s^*) \leq f(s), \forall s \in S \), where \( S \) is the set of all feasible solutions. In this case, \( s^* \) is called the global optimum solution of \( (S, f) \). A local minimum solution, similarly, inside a neighborhood \( N \), is a solution \( s^* \), such that \( f(s^*) \leq f(s), \forall s \in N(s^*) \), where \( N(s^*) \) contains the solutions in the neighborhood of \( s^* \).

Optimization problems may be mono or multiobjective. Mono-objective problems are those that focus on a single function that ought to be minimize or maximized. In the case of multiobjective problems, there are at least two functions to be simultaneously optimized. Since mono-objective problems are commonly easier to tackle, often the several functions are transformed in a single one so that these multiobjective problems can be treated through mono-objective solvers.

As examples of optimization problems, one can cite: projects in electricity distribution systems, positioning satellites, projects of computers and VLSI chips, vehicle routing, allocation task for workers or machinery, packing boxes in containers, cutting boards and bars, gene and DNA sequencing, classification of plants and animals, and so on. An extensive list of optimization problems can be found in (Crescenzi and Kann, 2005).

1.2 Metaheuristics

Reeves (1993) defines a metaheuristics as a technique intended to search for good solutions (near optimum solution) at a reasonable computing cost, thus ensuring the viability or the degree of optimality of this solution in a reasonable time. Osman and Laporte (1996) claim that a metaheuristic is a process of iterative generation which guides an underlying heuristic, in order to intelligently combine different concepts to investigate and explore the search space, using a strategy of memorization to structure the information, in order to obtain efficient and near-optimum solutions. In this work, we will adopt the following definition: “A metaheuristic is a general algorithmic structure for solving optimization problems that can be modified to adapt its behavior to the treatment of a specific problem”. Generally, metaheuristics are characterized as strategies that address the searching process, as non-deterministic approximate algorithms, which are designed to explore efficiently the search space to find optimum solutions (or near-optimum solutions) and that use randomization methods with heuristic information. Within this context, several metaheuristics have already been proposed.
BLS is an extremely simple local search algorithm. It is frequently called “Iterative Improvement” or “Hill Climbing”, since each movement is done only if the resulting solution is better than the current one. The algorithm stops when it finds a local minimum. ILS is a metaheuristic based on multiple paths, which means, fundamentally, that the algorithm is based on the perception that a local optimum can be improved through agitation, generating new starting solutions to be considered. The Tabu Search metaheuristic (Glover and Laguna, 1997) is a method of optimization that belongs to the class of local search techniques and that uses the concept of short and long-term memories. Its main feature is the use of a structure called tabu list, related to short-term memory, which stores attributes “banned” from solutions, aimed both to avoid local optimum solutions and the occurrence of cycles. In addition to the concepts of memories and tabu list, this metaheuristic comprise other concepts, such as an aspiration criterion, the duration of a tabu, and use of dynamic tabu lists. Genetic Algorithm is a type of Evolutionary Algorithm which has been employed extensively to solve optimization problems (Holland, 1975). In this approach, an initial population of solutions - in our case a set of test suites - is randomly generated. Next, new populations are generated iteratively based on the previous population until a stopping criterion is reached. The evolution from one to the next population is performed via several genetic operators, including: operations of selection, i.e., the biased choice of which individuals of the current population will reproduce to generate individuals for the new population. This selection prioritizes individuals with high fitness value, which represents how good this solution is. The other two evolutionary operators are: crossover, i.e., the combination of individuals to produce the offspring, and mutation, which randomly changes a particular individual. Scatter Search (Glover and Laguna, 1997) can be seen as an evolutionary metaheuristic which builds solutions through a combination of other solutions. It has its origin in the strategies originally proposed in the 60’s to combine of decision-making rules and restrictions. The Simulated Annealing algorithm, proposed by Kirkpatrick et al. (1983), is a metaheuristic that accepts movements which make the current solution worse as a strategy to escape from local optimum regions. This metaheuristic was inspired by the annealing process of physical systems, that is, based on an analogy with a thermodynamics process to simulate the slow cooling of a heated set of atoms. The Simulated Annealing algorithm is based on the Metropolis procedure (Metropolis et al., 1953), in which one can determine and view the different states (configurations) through which the number of atoms of a material passes until reaching a equilibrium state of balance at a given temperature. The Variable Neighborhood Search (VNS) metaheuristic (Hansen and Mladenovic, 1997) consists in systematically varying the neighborhood of a solution and applying a local search algorithm to search for good solutions.

In this paper we propose a framework of metaheuristics for solving mono-objective optimization problems. The main purpose of the framework is to facilitate the application of metaheuristics for solving problems in diverse knowledge areas and make it possible, as a consequence, for non-specialists to benefit from these techniques.

2. Related Works

In the literature, there are several framework projects that employ metaheuristics for the solution of a great number of problems. The following list presents a set of frameworks of metaheuristics:

**ParadisEO** (Cahon et al., 2004) is a mature framework with a well-developed API. It is implemented in C++ and focused on the deployment of parallel and distributed metaheuristics to solve combinatorial multiobjective optimization problems (CMOOP).

**JMetal** (Durillo, 2006) is a framework developed in Java that is focused on the development, testing and study of metaheuristics for solving CMOOPs. It deals more strongly with evolutionary and populational algorithms.
Open Metaheuristics (Dréo, 2008) focuses on creating tools to allow rigorous statistical studies of the behavior of metaheuristics.

OpenBEAGLE (Gagné and Parizeau, 2008) is another framework developed in C++ focusing on the development of evolutionary algorithms.

EasyLocal++ (Gaspero and Schaefer, 2003) and METSlib (Maischberger and Cumming, 2008) are frameworks developed in C++ focusing on the development of local search algorithms. EasyLocal++ implements three metaheuristics (Hill Climbing, Tabu Search and Simulated Annealing), METSlib implements five (Random Restart Local Search, ILS, VNS, Simulated Annealing and Tabu Search).

The main disadvantage of the approaches described above is the complexity of applying them, since all of these frameworks have large and complex APIs, which makes their understanding, and consequent application, a complex task. Their use is basically restricted for those who already have a good knowledge regarding metaheuristics. This fact was the primary motivator for the development the proposal outlined in this paper.

3. The Framework EasyMeta

3.1. Framework Requirements

In order to enable a broader use of this framework for optimization problems, we will present and discuss in this section the set of requirements that are prioritized by the proposal described next.

Simplicity. the framework instantiation process should be simple enough so that, even without a strong knowledge about metaheuristics, the framework can be employed to solve optimization problems in the most diverse areas.

Usability. the access to the framework’s features should be simplified so as to facilitate the invocation calls, manipulation of parameters and treatments of results.

Extensibility. the inclusion of new metaheuristics should be facilitated.

Portability. the framework should be portable to different hardware and software platforms, thus facilitating the expansion of its use.

Given these requirements, the framework described below proposes a simple solution, with good usability and high extensibility and portability.

3.2. Framework Architecture

Since simplicity is a key requirement, the architecture of the framework EasyMeta was designed in such a way the users have a minimum amount of effort to instantiate the framework in order to solve their particular problems, or to develop and insert new heuristics. Next, we describe and discuss the proposed architecture for this framework.

Figure 1 describes the class diagram for the EasyMeta framework. One can easily see that the core of the project is formed by abstract classes Metaheuristics, Problem and Solution. This occurs because all communication responsible for managing the problem solving process is performed via the methods required by these classes. The class Explorer is responsible for interfacing with the user and for the instantiation of the metaheuristics, when requested by the user. Following, we list and detail the responsibilities of each one of these classes:
Metaheuristics: defines the basic behavior of all metaheuristics in the framework. It has two methods. The solve() method must return an object of type Solution. This returned object Solution is a valid solution to the instance of a problem (previously passed as parameter, through the constructor of the Metaheuristics subclasses). The method listParameters() is responsible for listing all parameters required by the metaheuristic (for example, ‘size of the neighbourhood(k)’ for Simulated Annealing). This method is important since it allows the user to correctly parameterize his metaheuristic. A common feature of all metaheuristics is that the constructor must receive the same parameters as a HashMap containing the parameters that are required by such metaheuristic and an object of the class Problem. This HashMap is built by the Explorer class.

Problem: the subclasses of this class, unlike the metaheuristics, are necessarily built by the user. In it, the structures that shape the problem and the methods required by the superclass must be implemented, which will work directly with the subclass of Solution, which also must be implemented by the user. An important method is generateNeighborSolution(), where given a base solution and a ‘degree’ of distance, the method should return a valid solution that is within this ‘degree’ of distance from the base solution. The methods combine() and mutate() are only needed when new evolutionary metaheuristics are to be implemented.
Solution: the subclasses of this class do not require any specific method, only the `toString()` and `clone()` methods. It can just be used to encapsulate the data structures that represent the solutions to the given problem. Methods may be added to enrich its implementation.

Metaheuristics subclasses: they are the metaheuristics already implemented within the framework that are ready to use. The name of class directly refers to the metaheuristic that is implemented. In order to add new methods as a subclass of `Metaheuristics`, the only method that should be implemented is `solve()`, which solves the problem it received as an argument and returns a valid solution for it.

3.3. Using the Framework

The `Explorer` class deserves special attention, therefore it is discussed separately. As described earlier, this class is responsible for interfacing with the user. Consequently, in order to apply the framework, the user must run this class - usually via the command “java Explorer”. This call will make the program enter an idle state, waiting for the user to type a valid entry to be executed. The valid entries are (We use the following acronyms: MH for Metaheuristic, PB for Problem and AC for Configuration File):

- **help MH**: This command calls the method `listParameters()` from `Metaheuristics` to return the list of parameters that should be passed as argument.
- **MH CF\_mh PB CF\_pb**: With this call, the user invokes the implementation of the metaheuristic MH to solve the problem PB, both parameterized by the configuration files `CF\_mh` and `CF\_pb` respectively. After this process, the program will print out a summary of the performed execution and the solution to the particular instance of the problem.
- **compare MH\_1 AC\_MH\_1 MH\_n AC\_MH\_n PB AC\_PB**: This entry compares the performance of the selected metaheuristics with a unique problem. One can compare both the same metaheuristic with different parameters, as well as different metaheuristics. After the execution of each metaheuristic, the program prints out a summary of the execution of the metaheuristic and, at the end of all executions, the summary of the metaheuristic which achieved the best performance.

After the treatment of each entry, the program returns to an idle state, waiting for new calls or until the command “quit” is entered, which will stop its execution.

It is important to mention that the class `Explorer` handles the configuration files of the metaheuristics and the problems differently. In the configuration files of metaheuristics, the user must insert, in each line, the name of the parameter (previously defined in the implementation of the metaheuristic) and its desired value, separated by spaces. This is required so that the class `Explorer` can build the HashMap and pass it as a parameter to the constructor of desired metaheuristic – the other parameter to this constructor is an instance of the `Problem` class. On the other hand, the configuration files of problems have no defined format, since the implementation of the extraction of these values is done by the user. The only specification is that the constructor of subclasses of `Problem` should receive a String, which refers to the name of the configuration file.

One important fact is that the size of the solution neighborhood is treated as an integer, so the user needs to adapt to this design decision. Another important characteristic is that the framework only works to minimize the value of the objective function. If the problem is to maximize, the evaluation value of a solution must be multiplied by -1.

Table 1 lists the parameterization of all metaheuristics already implemented in the framework and explains their purpose.
To facilitate the use of the framework, it contains the implementation of a method of local search already implemented. With that, in spite of the information inherent problem of not being taken into consideration, you do not have to worry about creating a method of search, aparting your job from this issue. It is made making a call for the implementation of Simulated Annealing parametrizing it so that only movements of improvement are accepted and that the distance from the neighborhood of solutions is very small. Doing this we exchange performance and quality by usability.

<table>
<thead>
<tr>
<th>Metaheuristic</th>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLS</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>ILS</td>
<td>n_iterations</td>
<td>Number of iterations</td>
</tr>
<tr>
<td>Genetic Algorithm</td>
<td>n_iterations</td>
<td>Number of iterations</td>
</tr>
<tr>
<td></td>
<td>population_size</td>
<td>Size of the population</td>
</tr>
<tr>
<td></td>
<td>n_mutants</td>
<td>Percentage of individuals to be mutated</td>
</tr>
<tr>
<td>Scatter Search</td>
<td>n_iterations</td>
<td>Number of iterations</td>
</tr>
<tr>
<td></td>
<td>reference_set_size</td>
<td>Size of the reference set</td>
</tr>
<tr>
<td>Tabu Search</td>
<td>n_iterations</td>
<td>Number of iterations</td>
</tr>
<tr>
<td></td>
<td>tabu_tenure</td>
<td>Lifetime of a tabu item</td>
</tr>
<tr>
<td>Simulated Annealing</td>
<td>neighborhood_size</td>
<td>Size of the neighborhood</td>
</tr>
<tr>
<td>VNS</td>
<td>k_max</td>
<td>Maximum size of neighborhood</td>
</tr>
<tr>
<td></td>
<td>n_iterations</td>
<td>Number of iterations</td>
</tr>
</tbody>
</table>

It is worth mentioning that the framework is implemented in Java to meet the portability requirement and to facilitate its dissemination. The other requirements are met by the simplicity of the API, which makes the framework easy to understand and instantiate.

3.4. The Instantiation Process

The framework instantiation process is a very simple procedure, in such a way the user does not need to have a deep knowledge about the functioning of the metaheuristics, requiring only the knowledge about the structure of his particular problem. In short, one can list the steps of the instantiation process as follows:

1. Implementation of the subclasses of Problem and Solution;
2. Compilation of these new classes;
3. Insertion of the generated classes in the folder where the framework is located.

After these steps, the problem can be solved by any metaheuristic implemented in the framework. If one wants to add a new metaheuristic to the framework, the new metaheuristic class should inherit from Metaheuristics.

4. Case Study - Subset Sum Problem
In this case study, we will work with the optimization version of the subsets sum problem. It is important to mention that the decision version of this problem is a well-known NP-complete problem.

This problem can be formally defined as follows: Given a set \( C = \{c_1, c_2, \ldots, c_n\} \) of numbers and one value \( x \), a solver must return a subset \( C' \subseteq C \) of items where the sum of the elements in \( C' \) is as near as possible to \( x \) (less than or equal to it).

Following the instantiation process described in the previous section, the classes to describe the problem and the solution should be implemented initially, in this case, \texttt{SubsetSumProblem} and \texttt{SubsetSumSolution}. In this paper, we will only show the most important parts of each of these implementations, since some parts of the code are obvious or unnecessary at the time (as treatment of errors).

```java
import java.util.BitSet;

public class SubsetSumSolution extends Solution{
    BitSet bitset;
    Problem problem;

    public SubsetSumSolution(int length, Problem problem){
        bitset = new BitSet(length);
        this.problem = problem;
    }

    public String toString(){
        return problem.evaluate(this) + "";
    }

    public SubsetSumSolution clone(){
        SubsetSumSolution clone = new SubsetSumSolution(bitset.size(), problem);
        clone.bitset = (BitSet) bitset.clone();
        return clone;
    }
}
```

Figure 2. SubsetSumSolution source-code

Figures 2 and 3 exemplify the source-codes of classes that make the framework instantiable. As one can see, the constructor of the \texttt{SubsetSumProblem} class receives a string as the parameter from the problem configuration file. This file is free of format, and, in this case study, is a text file containing the following sequence of elements: \( x \mid C \mid c_1c_2c_3\ldots c_n \), elements which are passed to the variable \( x \) and vector \texttt{elements}, which has size \(| C |\).
import java.util.*;

public class SubsetSumProblem extends Problem{
    double x, elements[];

    public SubsetSumProblem(String conf_file, Problem problem) throws Exception{
        Scanner sc = new Scanner(new java.io.File(conf_file));
        x = sc.nextDouble();
        elements = new double[sc.nextInt()];
        for(int i = 0; i < elements.length; i++)
            elements[i] = sc.nextDouble();
    }

    public double evaluate(Solution solution){
        SubsetSumSolution sol = (SubsetSumSolution) solution;
        double eval = 0d;
        for(int i = 0; i < elements.length; i++)
            eval += sol.bitset.get(i) ? elements[i] : 0d;
        return eval;
    }

    public boolean isValid(Solution solution){
        return evaluate(solution) <= x;
    }

    public generateRandomSolution(){
        SubsetSumSolution sol = new SubsetSumSolution(elements.length, this);
        Random rd = new Random();
        do{
            for(int i = 0; i < elements.length; i++)
                sol.bitset.set(i, rd.nextBoolean());
        }while(!isValid(sol));
        return sol;
    }

    public Solution generateNeighborSolution(Solution bs, int min_d, int max_d){
        SubsetSumSolution sol = new (SubsetSumSolution) bs, new_sol;
        Random rd = new Random();
        do{
            new_sol = (SubsetSumSolution) sol.clone();
            int dist = rd.nextInt(max_d - min_d) + min_d + 1;
            for(int i = 0; i < dist; i++)
                new_sol.bitset.flip(rd.nextInt(elements.length));
        }while(!isValid(new_sol));
        return new_sol;
    }
}

Figure 3 SubsetSumProblem source-code

In the SubsetSumSolution class, the method clone() was subscribed to facilitate the creation of the neighbor solutions. Table 2 shows examples of configuration files to an instance of the problem in question and three metaheuristics that can solve it.

<table>
<thead>
<tr>
<th>SubsetSumProblem</th>
<th>Simulated Annealing</th>
<th>VNS</th>
<th>Tabu Search</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 10 7 1 2 9 3 8 4 5 6 10</td>
<td>n_iterations 20</td>
<td>n_iterations 5</td>
<td>tabu_tenure 5</td>
</tr>
</tbody>
</table>
Also due to space restrictions, in this case study we did not use the evolutionary metaheuristic, so the implementations of `mutate()` and `combine()` will not be displayed.

Table 3 Sample Input for the Framework

<table>
<thead>
<tr>
<th>Metaheuristic</th>
<th>Configuration</th>
<th>Problem</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLS</td>
<td>bls.conf</td>
<td>SubsetSumProblem</td>
<td>ssp 100.conf</td>
</tr>
<tr>
<td>TabuSearch</td>
<td>ts.conf</td>
<td>SubsetSumProblem</td>
<td>ssp 100.conf</td>
</tr>
<tr>
<td>VNS</td>
<td>vns.conf</td>
<td>SubsetSumProblem</td>
<td>ssp 100.conf</td>
</tr>
<tr>
<td>GeneticAlgorithms</td>
<td>ga.conf</td>
<td>SubsetSumProblem</td>
<td>ssp 100.conf</td>
</tr>
<tr>
<td>ILS</td>
<td>ils.conf</td>
<td>SubsetSumProblem</td>
<td>ssp 100.conf</td>
</tr>
<tr>
<td>ScatterSearch</td>
<td>ss.conf</td>
<td>SubsetSumProblem</td>
<td>ssp 100.conf</td>
</tr>
<tr>
<td>Quit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Finally, the framework can be executed. Table 3 shows the execution of the framework to solve an instance of the subset sum problem from a group of 100 randomly generated elements and $x = 26345748$. The value of $x$ was calculated as the sum of randomly chosen elements from the set. Table 4 shows the results given by the instantiated metaheuristics.

Table 4 Framework Output

<table>
<thead>
<tr>
<th>Metaheuristic</th>
<th>Problem</th>
<th>Execution time (ms)</th>
<th>Solution found</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLS</td>
<td>SubsetSumProblem</td>
<td>23</td>
<td>26344692.0</td>
</tr>
<tr>
<td>TabuSearch</td>
<td>SubsetSumProblem</td>
<td>119</td>
<td>26345676.0</td>
</tr>
<tr>
<td>VNS</td>
<td>SubsetSumProblem</td>
<td>36</td>
<td>26345466.0</td>
</tr>
<tr>
<td>GeneticAlgorithms</td>
<td>SubsetSumProblem</td>
<td>320</td>
<td>26345713.0</td>
</tr>
<tr>
<td>ILS</td>
<td>SubsetSumProblem</td>
<td>72</td>
<td>26345521.0</td>
</tr>
<tr>
<td>ScatterSearch</td>
<td>SubsetSumProblem</td>
<td>214</td>
<td>26345388.0</td>
</tr>
</tbody>
</table>

Conclusion and Future Works

This paper presents a proposal for a simple framework of metaheuristics, which is intended to facilitate the work of any user who does not have solid knowledge about the functioning of metaheuristics but who wants to employ them as a tool to solve specific optimization problems. As shown, the coding required to solve a problem, like the Subset Sum problem, is small and does not require major programming skills. The effort is completely focused on understanding and solving the problem, and not in the metaheuristics. Moreover, users who want to test their own solutions can use the framework to compare them with those implementations already available.

As future work, we will consider the increment in the number of implemented metaheuristics, as well as revision of the framework architecture to support constructive metaheuristics and also the creation of a graphical interface for it. Another object of study will be the creation of tools to automate the parameterization process.

References


Glover, F. (1977), Heuristics for integer programming using surrogate constraints, Decision Sciences, 8, 156-166.


Holland, J.H. (1975), Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, Univ. of Michigan.


