OptFrame: a computational framework for combinatorial optimization problems

Igor Machado Coelho[1], Sabir Ribas[1], Mário Henrique de Paiva Perché[1]
Pablo Luiz Araújo Munhoz[1], Marcone Jamilson Freitas Souza[2], Luiz Satoru Ochi[1]

[1] Instituto de Computação – Universidade Federal Fluminense (UFF)
Rua Passo da Pátria, 154 – Bloco E, 3º andar – CEP 24.210-240 – Niterói (RJ), Brasil

[2]Departamento de Ciência da Computação – Universidade Federal de Ouro Preto (UFOP)
Campus Universitário, Morro do Cruzeiro, CEP 35.400-000, Ouro Preto (MG), Brasil

imcoelho@ic.uff.br, sribas@ic.uff.br, mperche@ic.uff.br
pmunhoz@ic.uff.br, marcone@iceb.ufop.br, satoru@ic.uff.br

Abstract. *This work presents OptFrame, a computational framework for the development of efficient heuristic based algorithms. The objective is to provide a simple C++ interface for common components of trajectory and population based metaheuristics, in order to solve combinatorial optimization problems. Since many methods are very common in literature, we provide efficient implementations for simple versions of these methods but the user can develop "smarter" versions of the methods considering problem-specific characteristics. Moreover, parallel support for both shared-memory and distributed-memory computers is provided. OptFrame has been successfully applied to model and solve some combinatorial problems, showing a good balance between flexibility and efficiency.*
KEYWORDS*: Optimization. Framework. Metaheuristics. Main area: MH – Metaheuristics.*

Resumo. *Este trabalho apresenta o OptFrame, um arcabouço computacional para o desenvolvimento de algoritmos heurísticos eficientes. O objetivo é prover uma interface simples em C++ de componentes comuns para resolver problemas de otimização combinatória por meio de metaheurísticas populacionais e de busca local. Como alguns métodos são amplamente explorados na literatura, são providas implementações eficientes para versões simples de tais métodos. Porém, ao considerar características específicas do problema, o usuário pode desenvolver versões mais "inteligentes" dos métodos. Além disso, o arcabouço tem suporte a paralelismo em arquiteturas de memória compartilhada e distribuída. O OptFrame tem sido aplicado com sucesso na modelagem e resolução de problemas combinatórios, demonstrando um bom equilíbrio entre flexibilidade e eficiência.*
PALAVRAS-CHAVE*: Otimização. Arcabouço. Metaheurísticas. Área principal: MH – Metaheurísticas.*

1 Introduction

In the development of optimization systems it is common to face up with combinatorial NP-Hard problems. To produce algorithms that solve such problems is often a hard and long task, since the algorithm must solve the problem with low gaps in short computational time. That is, the heuristic algorithm must find good solutions at each execution. The solutions should be good enough for the application that uses the method and the elapsed time to generate them must be acceptable in terms of the application. One way of speeding up the development of such algorithms is by using tools that provide famous algorithms for the resolution of combinatorial problems, both in practical and theoretical cases.

Moreover, several successful applications in the resolution of combinatorial problems combine different strategies in one algorithm. Algorithms with this feature are classified as hybrid algorithms. It is easy to observe that the development and maintenance of such systems is an even more complex task. This fact often motivates the use of a tool that provides adaptable software components that encapsulate a domain-specific abstraction, that is, a framework.

The architecture of a framework, that typically follows the object-oriented paradigm, defines a model for code reuse (Fink and Voß, 2002). This fact justifies the development of frameworks that seek to solve optimization problems by means of heuristics and metaheuristics. Mainly because metaheuristics are essentially independent of the addressed problem structure. In the context of metaheuristics development, the developers that do not use any framework or library in general expend much effort by writing and rewriting code. Thus, the focus that should be at the problem and its efficient resolution is often directed to many programming aspects.

This work presents OptFrame[1], a white-box object oriented framework in C++ for the development of efficient heuristic based algorithms. Our objective is to provide a simple interface for common components of trajectory and population based metaheuristics. Since many methods are very used in literature we provide efficient implementations for simple versions of these methods but the user can develop *smarter* versions of the methods considering problem-specific characteristics.

The present work is organized as follows. Section 2 describes some optimization frameworks in literature. Section 3 defines important optimization concepts about metaheuristics that are behind OptFrame architecture. In Section 4 we present OptFrame architecture in details. Section 5 concludes the work with some applications and proposes some future development on the framework.

2 Frameworks in Optimization

Many authors have already proposed frameworks for optimization problems, among which we cite: *Neighbor Searcher* (Andreatta et al., 1998), *TabOO Builder* (Gracho and Porto, 1999), *NP-Opt* (Mendes et al., 2001), *HotFrame* (Fink and Voß, 2002), *EasyLocal++* (Gaspero and Schaerf, 2003), ParadisEO (Cahon et al., 2004), *iOpt* (Dorne et al., 2005), *JFFO* (Neves et al., 2005), jMetal (Durillo et al., 2006) and *TOGAI* (Souza, 2008). Now, we present some of them in details.

[1] OptFrame website: http://sourceforge.net/projects/optframe/

Andreatta et al. (1998) point that comparison between algorithms, strategies and parameters of heuristics is crucial when applied to combinatorial problems. But, many times comparisons are biased by some reasons. Sometimes methods are compared in different programming languages so as different architectures. The authors present a C++ computational framework, *Neighbor Searcher*, in order to make fairer comparisons between heuristic algorithms. Many implementation aspects are showed, but no real comparison between heuristics and problems is presented.

In Mendes et al. (2001) NP-Opt is presented, a computational framework for NP class problems. The framework proposes to minimize code rewriting when the focused problem is changed. NP-Opt supports five distinct problems: Single Machine Scheduling, Parallel Machine Scheduling, Flowshop Scheduling with job families, Grid Matrix Layout (VLSI design) and non-linear continuous function optimization. It is provided an interface for the extension of the framework for other problems. The methods for the resolution of problems are Memetic and Genetic Algorithms, so as Multiple Start. The authors of NP-Opt points to a code reuse of 75% when dealing with a new problem. The framework is programmed in Java language.

Fink and Voß (2002) present the C++ computational framework HotFrame, that shares some similarities with OptFrame, proposed in this work. HotFrame, so as OptFrame, was firstly designed for Iterated Local Search, Simulated Annealing and Tabu Search metaheuristics. And also in this sense HotFrame is very complete, since the authors show many implementation details and many variations of these metaheuristics. According to the authors a framework provides adaptable software components, which encapsulate common domain abstractions. To develop a framework requires solid knowledge in the considered domain.

Gaspero and Schaerf (2003) point that local search is a common interest theme of scientific community, at the same time that there isn't a standard software in this sense. So, the authors propose EasyLocal++, a computational object-oriented framework for the design and analysis of local search algorithms. According to the authors the architecture of EasyLocal++ allows code modularization and the combination of basic techniques and neighborhood structures. Some successful applications of EasyLocal++ are showed and according to the authors EasyLocal++ provides flexibility enough for the implementation of many scheduling problems.

ParadisEO (Cahon et al., 2004) is a white-box object-oriented framework written in C++ and dedicated to the reusable design of parallel and distributed metaheuristics. This framework is based on a conceptual separation of the solution methods from the problems they are intended to solve. According authors, this separation confers to the user a maximum code and design reuse. ParadisEO provides some modules that deals with population based metaheuristics, multiobjective optimization, single-solution based metaheuristics, and it also provides tools for the design of parallel and distributed metaheuristics. ParadisEO, as the OptFrame, is one of the rare frameworks that provide parallel and distributed models. Their implementation is portable on distributed-memory machines as well as on shared-memory multiprocessors, as it uses standard libraries such as MPI, PVM and PThreads.

The Intelligent Optimization Toolkit (iOpt), proposed by Dorne et al. (2005) can be seen as an IDE for the rapid construction of combinatorial problems. The iOpt takes as

input problems modeled in "*one-way constraints*" and uses metaheuristics to solve them. The authors show how to model the Vehicle Routing Problem with iOpt and good results are reported. Finally, the authors conclude that a better understanding of the problem can be achieved by a fairer comparison between heuristic methods.

jMetal (Durillo et al., 2006) is an object-oriented Java-based framework aimed at facilitating the development of metaheuristics for solving multi-objective optimization problems (MOPs). According authors, this framework provides a rich set of classes which can be used as the building blocks of multi-objective metaheuristics; thus, taking advantage of code-reusing, the algorithms share the same base components, such as implementations of genetic operators and density estimators, so making the fair comparison of different metaheuristics for MOPs possible.

In general, frameworks are based on the authors experience with the implementation of many methods for different problems. In this work we also review some important concepts of combinatorial problems and metaheuristics, in order to propose an architecture that is both problem and heuristic independent. The following section shows the theoretical modeling of combinatorial problems behind OptFrame architecture.

3 Metaheuristics

We present now some important concepts of metaheuristics and combinatorial optimization problems.

Let S be a set of discrete variables s (called *solutions*) and $f : S \rightarrow \mathbb{R}$ an objective function that associates each solution $s \in S$ to a real value $f(s)$. We seek any $s^* \in S$ such that $f(s^*) \leqslant f(s), \forall s \in S$ for minimization problems, or $f(s^*) \geqslant f(s), \forall s \in S$ for maximization problems. The solution s^* is called a *global optimum*.

A function N associates a solution $s \in S$ to a set $N(s) \subseteq S$ (called *neighborhood* of s). This is also an important concept in the subject of heuristic based algorithms. This way, a *neighbor* s' of s is such that $s' = s \oplus m$, where m is called a *move* operation. The cost of a move m is defined as $\widehat{f} = f(s') - f(s)$, which means that $s' = s \oplus m \implies f(s') = f(s) + \widehat{f}$. So, a *local optimum* (in terms of a neighborhood N) is a solution s' such that $f(s') \leqslant f(s), \forall s \in N(s')$ for minimization problems, or $f(s') \geqslant f(s), \forall s \in N(s')$ for maximization problems.

Many combinatorial optimization problems are classified as NP-Hard and it is common to use *approximate* algorithms (or *heuristics*) to solve these problems. These methods have the capability of finding good local optimums in short computational time. Classical local search heuristics stop on the first local optimum found. However, metaheuristics can go beyond the local optimum and thus these methods are able to produce final solutions of better quality.

4 OptFrame

OptFrame is a white-box object oriented framework in C++. In the following sections its implementation and design aspects are presented and discussed.

4.1 Representation and Memory

The OptFrame framework is mainly based on two important structures: the solution representation and the memory.

The representation is the data structure you use to represent a valid solution for a specific problem. For example, for the Traveling Salesman Problem (Applegate et al., 2006) a user may wish to represent the solution as an array of integers, so the representation in this heuristic approach for TSP is $vector < int >$.

On the other hand, the memory is a set of auxiliary data structures needed for a *smarter* version of your method.

4.2 Solution and Evaluation

There are two important *container* classes[2] in OptFrame: Solution and Evaluation. Solution carries a reference to a Representation of your problem, while a Evaluation carries a reference to a Memory structure for your problem. When you implement a *smart* version, you can use the information of the Memory to reevaluate a Solution in a smarter way.

Both Solution and Evaluation classes implement the *Cloning Pattern* (Henney, 1999) in order to provide copies of their objects in real time. The carried objects (representation or memory) need to be in the Orthodox Canonical Form and also to provide an implementation of the *operator* $<<$.

Figure 1 depicts the Solution and Evaluation classes.

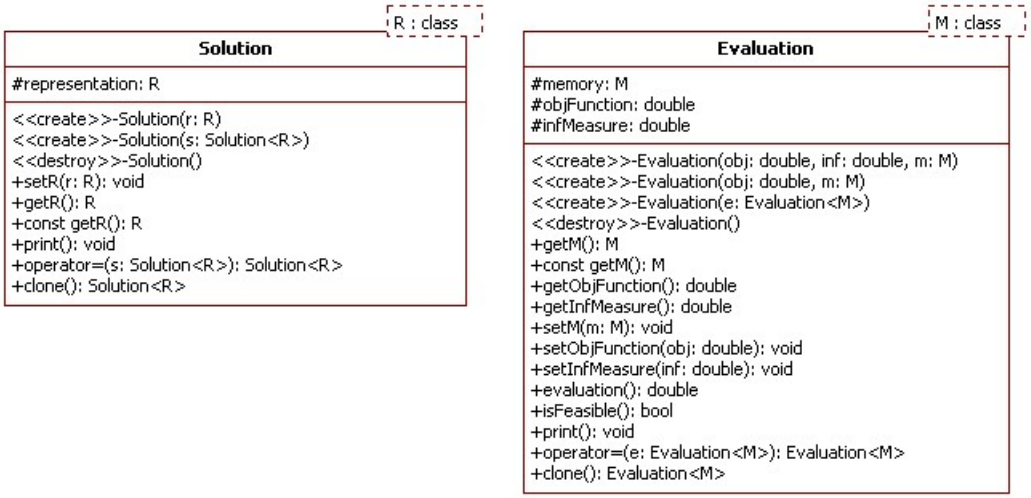

Figure 1. Solution<R> and Evaluation<M> template classes

4.3 Evaluators

The Evaluator concept is a very important in OptFrame. It encapsulates the function $f : S \rightarrow \mathbb{R}$ (defined in Section 3) as an specific case of its function $f : S \rightarrow E$, where $E = (\mathbb{R}, \mathbb{R}, M)$. The tuple E can be seen as the Evaluation class defined in Subsection 4.2.

The first value of the tuple E is the *objective function value* itself and the second one is an *infeasibility measure value*. By evaluating a solution this way you can implement

[2]What we name here as a *container* class is in some ways related to with Proxy Pattern (Gamma et al., 1995) since the idea is to carry a reference to an object (representation or memory) and to delete it when the container itself is destroyed. But in this case a *container* is also used to provide some extra operations over the carried object like *printing*, *reference counting* and *cloning*.

heuristic methods that *are able to see* unfeasible solutions, by giving a high penalty value to the *infeasibility measure value*. When the *infeasibility measure value* is zero the solution is considered *feasible*. So, the *evaluation function value* over a solution consists in the sum of $objective\_function\_value + infeasibility\_measure\_value$.

The third value M of the tuple E is called *memory* defined in Subsection 4.1. In this context the *memory* can record some steps of the evaluation algorithm, so they won't be repeated in future evaluations. This way, some future computational effort can be avoided.

There is also a more general definition for the $evaluation$ method where the function f is defined by $f : (S, E) \rightarrow E$. This way it is possible to develop *smarter* versions of a Evaluator by using informations of a previous evaluation E.

An approach for multiobjective optimization (Arroyo, 2009) is also provided, although many kinds of multiobjective evaluation functions can be written as a single Evaluator. The MultiObjectiveEvaluator class is built given an array of Evaluators of the same direction (minimization or maximization) and its evaluation consists on the sum of the partial evaluations over a single Solution. In future versions of the framework it will be possible to provide more abstractions for the final user in terms of multiobjective optimization, like on Durillo et al. (2006) and Cahon et al. (2004), but we don't plan to go to much deeper in order to keep simplicity.

Figure 2 depicts the Evaluator and MultiObjectiveEvaluator classes.

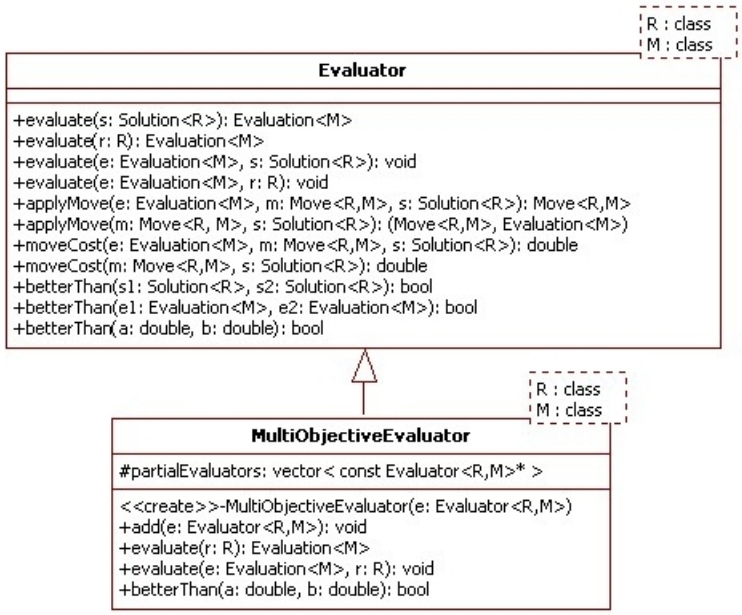

Figure 2. Evaluator<R,M> and MultiObjectiveEvaluator<R,M> template classes

4.4 Moves

A move operation defines a neighborhood structure. In OptFrame the Move class has two most important methods: $canBeApplied$ and $apply$.

The $canBeApplied$ method of a Move object m returns *true* if the application of m to a solution s will produce a valid solution. Otherwise it returns false. This is method is often used before the $apply$ method.

The $apply$ method of a Move m to a solution s transforms s into a neighbor s' and returns another Move $\overline{m}$ that can undo the changes made by m. Since complete copies of solutions are expensive operations it is possible to avoid them by developing efficient implementations of the reverse Move $\overline{m}$.

Figure 3 depicts the Move class.

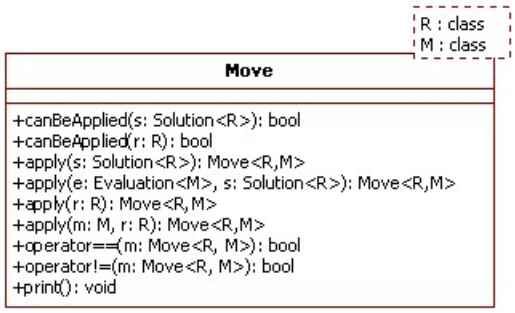

Figure 3. Move<R,M> template class

4.5 Neighborhood Structures

There are three types of neighborhood structure in OptFrame: NS, NSSeq and NSEnum.

NS is the simplest definition of a neighborhood structure. It only requires the user to define a $move(s)$ method, that returns a random move operation of the neighborhood type. Although it's not in focus in this paper, it is possible to define neighborhood structures for continuous problems optimization using this kind of structure.

NSSeq is a more elaborated version of NS. It also requires the user to define a $getIterator(s)$ method, that returns an object capable of generating moves of the neighborhood structure in a sequential way. The returned object must implement the NSIterator interface, that itself implements the Iterator Pattern (Gamma et al., 1995).

NSEnum is the most complete definition of a neighborhood structure in OptFrame. It provides an enumerable set of move operations for a given combinatorial problem. Although it only requires the user to define the $move(int)$ and $size()$ methods, with these methods it is possible to define default implementations for the $move(s)$ and $getIterator(s)$ methods of NS and NSSeq.

Figure 4 depicts the NS, NSSeq and NSEnum classes.

4.6 Heuristic based methods

Heuristic methods are mainly divided in two classes: *trajectory based* and *population based* methods (Ribeiro and Resende, 2010).

In order to maximize the code reuse and to favor testing of Hybrid Metaheuristics (Blum and Roli, 2008), all heuristic methods should be implemented using the Heuristic class abstraction. With this abstraction we have already been able to implement the following methods: First Improvement, Best Improvement, Hill Climbing and other classical heuristic strategies (Hansen and Mladenović, 2006); Iterated Local Search, Simulated Annealing, Tabu Search, Variable Neighborhood Search and other basic versions of many

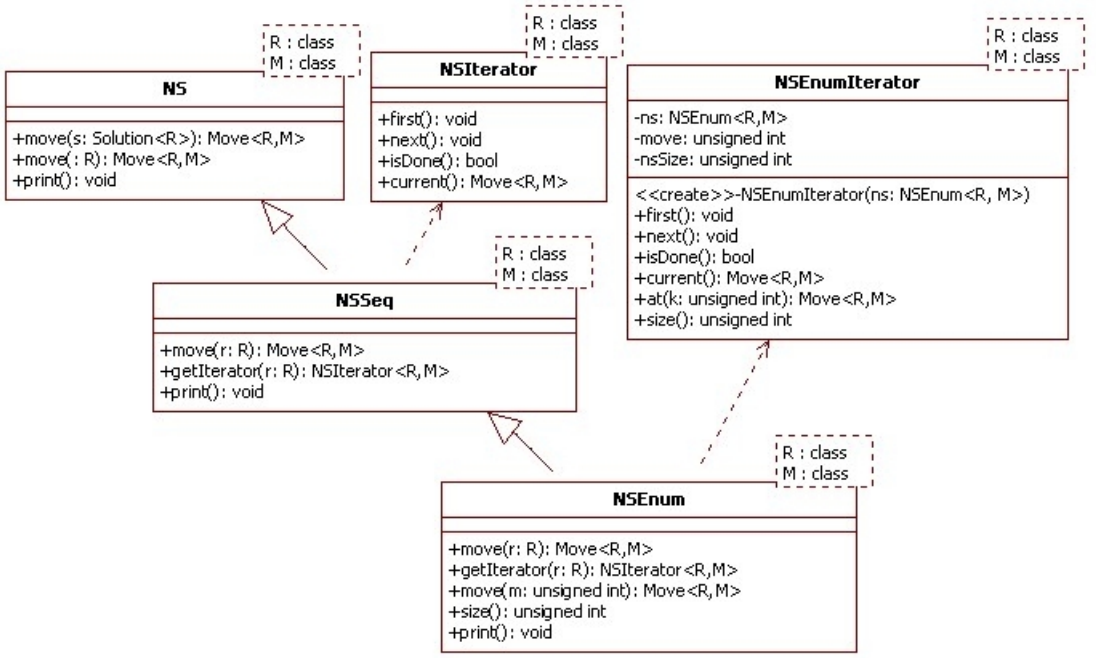

Figure 4. NS<R,M>, NSSeq<R,M> and NSEnum<R,M> template classes

famous trajectory based metaheuristics (Glover and Kochenberger, 2003); and, finally, the basic versions of population based metaheuristics Genetic Algorithm and Memetic Algorithm (Glover and Kochenberger, 2003).

So, there are four definitions of the method *exec* and the user must implement at least two of them. For trajectory based heuristics, the user must implement:

```
void exec(Solution){ ... }
void exec(Solution, Evaluation){ ... }
```

For population based heuristics:

```
void exec(Population){ ... }
void exec(Population, FitnessValues){ ... }
```

where: `Population` is a list of `Solutions` and `FitnessValues` is a list of `Evaluations`.

The first one is the simplest version of the method while the second is a more elaborated version. But if the user wish to implement only one of them there is a trivial way of doing this (since the heuristic class uses at least one Evaluator):

```
void exec(Solution s)
{
    Evaluation e = eval(s)
    call exec(s,e)
    delete e
}
```

```
void exec(Solution s, Evaluation e)
{
    call exec(s)
    e = eval(s)
}
```

For population based heuristics the same idea can be applied.

Figure 5 depicts the Heuristic class.

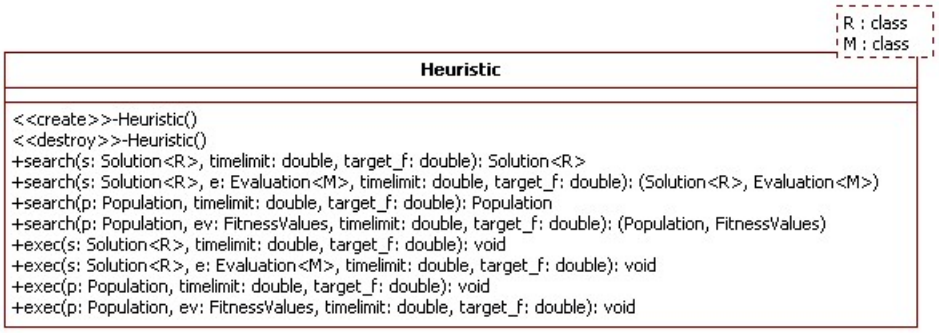

Figure 5. Heuristic<R,M> template class

4.7 Other structures

Some metaheuristics may require specific structures, but they can also be defined in specific files, e.g., Perturbation for Iterated Local Search; Mutation and Crossover operators for Genetic and Memetic Algorithms.

4.8 Parallel and Distributed Processing Support

One of the most important concepts in optimization local search heuristics is the notion of neighborhood. All classic local search heuristics, as well as local search metaheuristics, explore the search space by means of neighborhood structures. Thus, it is easy to observe that by parallelizing basic heuristics like Best/First Improvement many local search metaheuristics will be automatically parallelized.

As the search space of solutions is generally very wide, checking all possible neighbors of a solution is a costly task. To accelerate this process, OptFrame has a parallel generator of best neighbors. Thus, at each step, the solution space is divided among the processing nodes, reducing it into smaller sub-spaces.

The parallelization developed on OptFrame is similar to that proposed in Ribas et al. (2010). In that work, the authors present the framework MaPI and apply the parallelization to optimization algorithms. To test such an application the authors implemented an algorithm widely used in literature, the Hill Climbing, and applied it to a classic optimization problem, the Traveling Salesman Problem. When using MaPI, any user is able to implement a parallel application without worrying about the communication scheme between processes or how the system makes the parallelization.

OptFrame implements the best neighborhood generator in a Heuristic class called BestImprovement, which has two parallel versions, one focused on multi-core computers and other for MPI-based clusters. In order to develop an efficient parallelization but without worrying about problematic and advanced aspects of distributed programming, we use the library MapMP (Ribas et al., 2009) and MaPI framework (Ribas et al., 2010), both developed in the project MapReduce++ [3]. This project, under the GNU LGPLv3, provides different implementations of the MapReduce abstraction (Dean and Ghemawat, 2008) in C++ programming language.

[3]MapReduce++ project is available on http://sourceforge.net/projects/mapreducepp/ under GNU LGPLv3 license

Figure 6 presents the BestImprovement heuristic with MapReduce abstraction. In this case, each process is responsible for analyzing a sub-space neighborhood of a given solution. The input of MapReduce is a list of solutions. The *map* function receives a solution and a sub-space of a neighborhood and returns the movement that generated the best solution neighbor in this sub-space. The output of MapReduce is the best move generated among all sub-spaces.

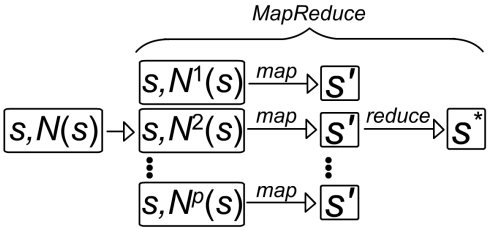

Figure 6. BestImprovement heuristic with MapReduce abstraction

The best neighbor of a solution s is generated by the move m that have the most favorable value of the evaluation function in the neighborhood of s. Since a neighbor can be generated by direct application of a movement and its manipulation is less costly than the manipulation of a neighbor solution, we chose to parallelize the method $BestMove(.)$ instead of $BestNeighbor(.)$. The idea is to generate the set M of moves that defines the neighborhood $N(s)$ of a solution s, and then divide this set M in $|P|$ disjoint subsets $\{(s, M^1), \cdots, (s, M^{|P|})\}$, where $|P|$ is the number of mapping processors. The next step is to apply the *map* function to each subset in order to generate their best moves. In possession of the best moves, the *reduce* function chooses the best among them.

The pseudo-code of the parallel generator of $BestMove(.)$ with the MapReduce abstraction is presented below.

Procedure `bestMove(`*s, N*`)`

$M \leftarrow$ Set of possible moves in $N(s)$;
Split M in $|P|$ parts and let M^i be the i-th part of M;
return $mapReduce(mapmm, reducemm, \{(s, M^1), ..., (s, M^{|P|})\})$;

Procedure `mapmm(`*s,* M^i`)`

foreach *each* $m_k^i \in M^i$ *with* $k = 1, ..., |M^i|$ **do**
  Evaluate the application of m_k^i to the solution s;
  $m^{*i} \leftarrow$ *acceptanceCriterion*(m^{*i}, m_k^i);
end
$c^{*i} \leftarrow$ cost of the application of m^{*i} to the solution s;
return $(m^{*i}, bestCost)$;

Procedure `reducemm(`B`)`

Let $B = \{(m^{*1}, c^{*1}), ..., (m^{*|P|}, c^{*|P|})\}$ be the set of mapped elements;
$m^* \leftarrow$ move m^{*i} such that c^{*i} be the minimum B, $\forall i = 1, ..., |P|$;
return m^*;

In the procedure $mapmm$, $acceptanceCriterion$ (m^{*i}, m_k^i) returns the better of two movements, with m^{*i} the best so far of i-th part the neighborhood of s and m_k^i the current movement in this part of the neighborhood.

5 Concluding remarks, Applications and Future Work

This work presents OptFrame, a white-box object oriented framework in C++ for the development of efficient heuristic based algorithms. Our objective is to provide a simple interface for common components of trajectory and population based metaheuristics.

OptFrame's architecture is intended to minimize the differences among code and theoretical concepts of combinatorial optimization. Thus, this paper describes a C++ modeling of the framework, but this model can also be applied to other programming languages, since generic programming features are available.

OptFrame is a free software licensed under LGPLv3. You can get the newer stable version of OptFrame on *http://sourceforge.net/projects/optframe/* or browse the project SVN files repository for development versions. It has been successfully applied to model many realistic optimization problems. See Souza et al. (2010), Coelho et al. (2009), Ribas et al. (2009) and Munhoz et al. (2009) for applications in open-pit-mining and single-machine scheduling problems, for example.

In future works we plan to compare pure *C* language implementation of some methods and combinatorial problems, in order to validate the implementation of OptFrame so as to benchmark its efficiency limitations. Currently, we are working on a language for automated testing of heuristic methods and an interactive mode for OptFrame.

You're invited to visit our homepage and collaborate with the project, as an user or as a developer. Code reuse must be maximized, with clear abstractions based on optimization concepts, but always keeping in mind that the target user should use only simple C++ on his/her code.

Acknowledgments

The authors are grateful to CNPq (CT-INFO and UNIVERSAL), CAPES (PRO-CAD and PRO-ENG), FAPERJ and FAPEMIG that partially funded this research.

References

Andreatta, A. A.; Carvalho, S. E. R. and Ribeiro, C. C. (1998). An object-oriented framework for local search heuristics. *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, p. 1–33, Washington, USA. IEEE Computer Society.

Applegate, D. L.; Bixby, R. E.; Chvatal, V. and Cook, W. J. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, United Kingdom.

Arroyo, J. E. C. *Heuristics and metaheuristics for multi-objective combinatorial optimization (in Portuguese)*. PhD thesis, Unicamp, Campinas, (2009).

Blum, C. and Roli, A. (2008). *Hybrid Metaheuristics*. Springer.

Cahon, S.; Melab, N. and Talbi, E.-G. (2004). Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, v. 10, n. 3, p. 357–380.

Coelho, I. M.; Ribas, S.; Souza, M. J. F.; Coelho, V. N. and Ochi, L. S. (2009). A hybrid heuristic algorithm based on grasp, vnd, ils and path relinking for the open-pit-mining operational planning problem. *Proceedings of the XXX Iberian-Latin-American Congress on Computational Methods in Engineering – CILAMCE*, Búzios.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, v. 51, n. 1, p. 107–113.

Dorne, R.; Mills, P. and Voudouris, C. (2005). Solving vehicle routing using iOpt. *Proceedings of MIC 2005 - The 6th Metaheuristics International Conference*, Viena, Áustria.

Durillo, Juan J.; Nebro, Antonio J.; Luna, Francisco; Dorronsoro, Bernabé and Alba, Enrique. (2006). jMetal: A java framework for developing multi-objective optimization metaheuristics. Technical Report ITI-2006-10, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, E.T.S.I. Informática, Campus de Teatinos.

Fink, A. and Voß, S. (2002). HotFrame: a heuristic optimization framework. Voß, S. and Woodruff, D. L., editors, *Optimization Software Class Libraries*, p. 81–154. Kluwer Academic Publishers, Boston.

Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.

Gaspero, L. Di and Schaerf, A. (2003). EasyLocal++: an object-oriented framework for the flexible design of local-search algorithms. *Softw. Pract. Exper.*, v. 8, n. 33, p. 733–765.

Glover, F. W. and Kochenberger, G. A. (2003). *Handbook of Metaheuristics*. Springer. ISBN 1402072635.

Graccho, M. and Porto, S. C. S. (1999). TabOOBuilder: An object-oriented framework for building tabu search applications. *Proceedings of the Third Metaheuristics International Conference*, p. 247–251, Angra dos Reis, Rio de Janeiro.

Hansen, P. and Mladenović, N. (2006). First vs. best improvement: an empirical study. *Discrete Appl. Math.*, v. 154, n. 5, p. 802–817. ISSN 0166-218X.

Henney, K. (1999). Coping with copying in c++. *Overload*, v. 7, n. 33, p. 1–16.

Mendes, A.; França, P. and Moscato, P. (2001). NP-Opt: an optimization framework for np problems. *Proceedings of the IV SIMPOI/POMS 2001*, p. 11–14, Guarujá, São Paulo.

Munhoz, P. L. A.; Perché, M. H. P. and Souza, M. J. F. (2009). A new algorithm based on Iterated Local Search for a class of scheduling problems on a single machine with earliness and tardiness penalties (in portuguese). *Proceedings of the XXIX ENEGEP*, Salvador.

Neves, T. A.; Souza, M. J. F. and Martins, A. X. (2005). Construction of a prototype of an optimization framework and its use to solve the heterogeneous fleet vehicle routing problem with time windows (in portuguese). *Proceedings of the XXVIII CNMAC*, Santo Amaro.

Ribas, S.; Coelho, I. M.; Souza, M. J. F. and Menotti, D. (2009). Parallel Iterated Local Search applied to the open-pit-mining operational planning problem (in portuguese). *Proceedings of the XLI SBPO*, p. 2037–2048, Porto Seguro.

Ribas, S.; Perché, M. H. P.; Coelho, I. M.; Munhoz, P. L. A.; Souza, M. J. F. and Aquino, A. L. L. (2010). MaPI: a framework for algorithms parallelization (in portuguese). *Learning and Nonlinear Models*, v. 8. Accepted for publication.

Ribeiro, C. and Resende, M. (2010). Path-relinking intensification methods for stochastic local search algorithms. Technical Report NJ 07932, AT&T Labs Research.

Souza, F. D. (2008). TOGAI: A tool for development of genetic algorithms (in portuguese). Master's thesis, Programa de Pós-Graduação em Pesquisa Operacional e Inteligência Computacional, Universidade Cândido Mendes, Campos dos Goytacazes.

Souza, M. J. F.; Coelho, I. M.; Ribas, S.; Santos, H. G. and Merschmann, L. H. C. (2010). A hybrid heuristic algorithm for the open-pit-mining operational planning problem. *European Journal of Operational Research*, v. . Accepted for publication.