General Object-oriented Framework for Iterative Optimization Algorithms

Zvonimir Vanjak and Vedran Mornar

Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia

It is usually impossible to exactly solve the hard optimization problems. One is thus directed to iterative algorithms. In implementation of these iterative algorithms, some common characteristics can be observed, which can be generalized in an object-oriented framework. This can significantly reduce the time needed for implementation of an iterative algorithm.

Keywords: object-oriented framework, iterative algorithms, optimization

1. Introduction

When developing any kind of system with the purpose to solve some optimization problems, one is confronted with the following common issues:

— representation of results

— tracking of the algorithm progress

— implementation of stop criteria to limit the algorithm duration

— preservation of intermediate results

The first issue can be handled easily. With the use of modern object-oriented programming languages, it is simple to define data structures that preserve the results of the calculation in appropriate form. But, this straightforward concept can be further improved. It is easy to define structures that represent not only the final solution, but also hold information on intermediate solutions which algorithm produced. Accordingly, the algorithm progress towards the final solution can be analyzed as well.

The second issue appears when calculations that need to be performed to reach solution are time consuming. It is not user-friendly to leave a blank screen in front of the user until the very moment when the calculation is done. So, it is necessary to find a way to periodically refresh screen with current solution to the problem. The third issue is related to the fact that sometimes it is extremely convenient to limit the duration of the algorithm. The number of iterations could be the stop criterion, but the total execution time is more natural.

Preservation of intermediate results is related to the representation of results and the tracking of the algorithm's progress. Its implementation depends on solutions to those two problems. When defining the representation of results, we implement the data structures that keep the intermediate results. But, besides the definition of the structure composition, it is necessary to define when the intermediate results will be saved.

The formal definition of an iterative optimization algorithm is now presented. In this work the iterative optimization algorithm is defined as any algorithm that can be stated by the following pseudo-code:

```
generate initial solution
do
      improve solution using iterative step
while condition is satisfied
end
```

One iteration of the algorithm can be a step of arbitrary complexity, as long as calculation in the i^{th} iteration depends only on the results obtained in the $(i-1)^{\text{th}}$ iteration. Condition for the termination of the calculation can be either

reaching some predefined number of iterations, reaching solution of a given quality, or elapsed time.

2. The Classes for Representation of the Results

Whenever we are confronted with a problem of optimization, whether it is a traveling salesman problem, quadratic assignment problem, or simply the problem of finding the maximum of some function, there is always (1) a goal function to optimize, (2) the value of the goal function and (3) the values of problem variables that present the solution.

Mathematically, we could define the goal function as $OptFunc : X \rightarrow Y$, where X is a set of variables of type _TypeValue and Y is a set of variables of type _TypeSol. This can be performed using templates, which is the way of doing generic programming in C++ [1], [2].

In correspondence with declaration of the optimization function, it is immediately obvious that the classes that will be used to hold the representation of our problems will have to be parameterized. Two parameters will be involved: first one that will define the type of the value of the function (which will usually be a floating point number, *float* or *double* in C++ and C), and the second one to define the type of the solution.

We can define a general class OptimizationResult, which can be used to represent the result of any kind of optimization problems. Except for the defined member variables, it is necessary to add a few more members to satisfy the *canonical form* [3] that enables the usage of declared class *al pari* with any predefined type in C++ (required members include constructor, destructor, copy constructor). Since this class represents the result of an iterative algorithm, two more members are added: one to represent the total number of iterations, and one to represent the total calculation time in milliseconds.

As mentioned in the introduction, sometimes it is convenient to have the possibility of recording some (or all) solutions that the iterative algorithm has achieved on its way to the final solution. This can be useful to analyze the progress of the algorithm, or to observe how the modification of algorithm parameters affects the algorithm performance. A class with capability of saving such results would have to enclose some kind of container where those results could be stored. For this purpose, a group of classes defined in *Standard Template library* (STL), which is the part of standard C++, can be very useful. Since we are interested in retrieving the data by some key (in our case it is the ordinal number of iteration for which we have saved the solution), the parameterized class map<class Key, class T> is perfectly suited. Here Key represents the type of the key used for retrieving the data, and T is the type of data to be saved in the container, So, we can define the class OptContResult (also parameterized in concordance with type of optimization problem), which inherits the class OptimizationResult. Additionally, we can define another class, named OptContResultWithSolutions. This class, besides the values of goal function during the iterations, saves the associated solution. Distinction between these two classes is necessary because sometimes we are interested only in goal function values, as preservation of the solutions often comes at extra cost in memory space. The hierarchy of these classes is shown in *Fig. 1*.

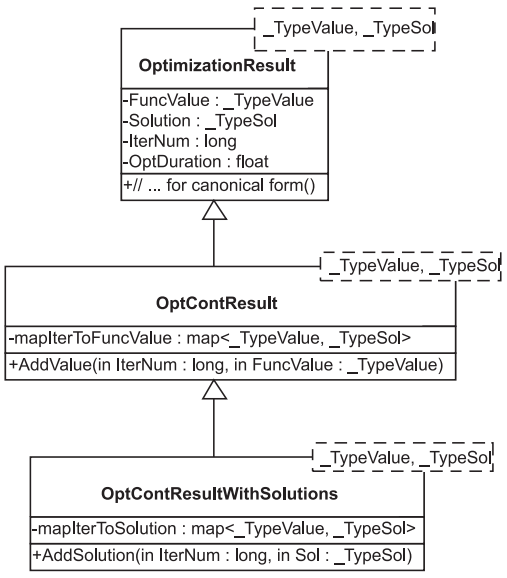

Fig. 1. Class hierarchy for classes that represent solutions.

3. The Classes for Encapsulation of the Iterative Algorithm

Since we have given the definition of an iterative algorithm, we can immediately write an interface that such an algorithm must satisfy:

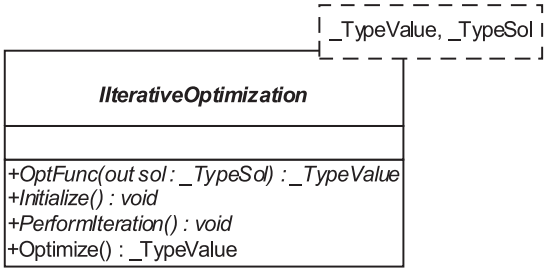

Fig. 2. Interface for iterative algorithms.

But, encapsulation of this kind is meaningless unless it facilitates the implementation of issues mentioned in the introduction: the termination of optimization, display of the current solution and preservation of intermediate results.

3.1. The Termination of Optimization

The length of iterative calculation is commonly expressed with the number of iterations to perform. However, sometimes it is more convenient to have the possibility of defining the maximum duration of the calculation. Also, sometimes it is necessary that the calculation is performed until some desired solution is found. This can be commonly solved by writing the separate control routines for different versions of duration control. The object-oriented paradigm offers another solution. We can define different objects that will handle different ways of control of the algorithm. What is even more important, all of these objects will have the same interface, which means that with the use of polymorphism we can handle all of them in the same way.

Such an interface is shown in *Fig. 3*. All objects that define their own criteria for termination will have to implement this interface. At the beginning of the optimization, the object must receive the message `Start()` in order to perform the necessary initializations. The function NextStep() is called after each iteration so that the object can adjust its state to the performed calculation, e.g. increase iteration number, refresh current time, or remember current solution (the argument void *pData is necessary because some objects that will implement this interface will have to pass some data to this function in order to adjust its state). The function IsOptimizationFinished() implements the criterion for decision whether optimization has come to an end or not. The objects that implement this interface could be IterativeOptTerminator, that terminates optimization when some predefined number of iterations is reached, or TimeOptTerminator that does the same when a

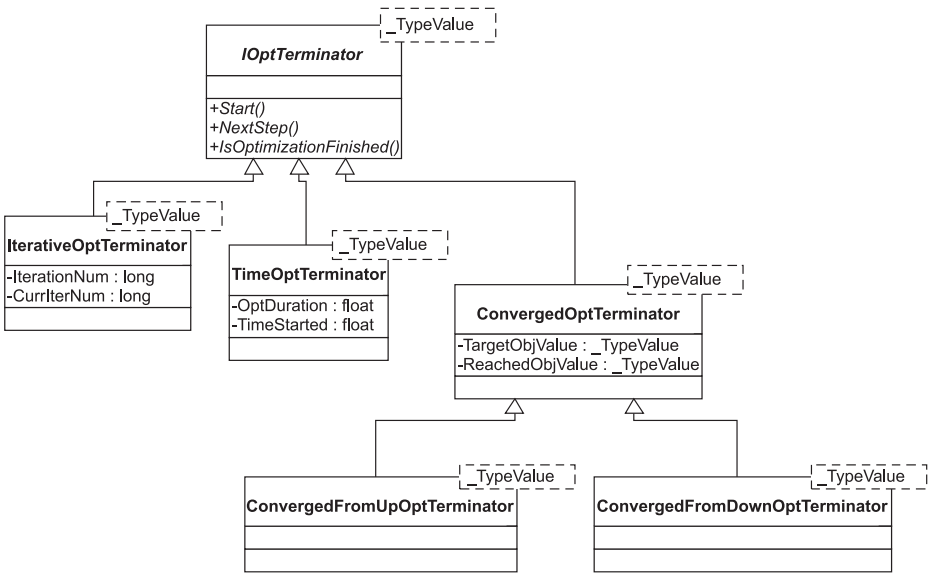

Fig. 3. Class hierarchy for classes implementing IOptTerminator interface.

predefined interval of time has passed. Also, as in class hierarchy diagram shown on the *Fig. 3*, we can define ConvergedOptTerminator object that terminates optimization when some defined objective function value has been reached (two distinct classes are defined since it provides a more elegant solution for handling minimization and maximization problems).

3.2. The Progress Tracking

As mentioned earlier, when lengthy calculations are performed, it is necessary to periodically refresh user screen with the current solution, so that the user has some sense of progress of the calculation. Also, the opportunity to observe the advance of the algorithm in search for solution sometimes comes handy. It becomes natural that there should be several ways to determine the refreshing period:

— refreshing after determined number of iterations

— refreshing after time interval

— refreshing when objective function has improved by some amount.

A simple solution, such as direct coding of all those possibilities in code and choosing one of the options depending on the function argument, would result in repetition of the same program code. Therefore, another approach has been implemented. As in the previous paragraph, where we have defined an object that takes care of optimization termination, here we define an object that handles the refreshing. The interface, together with two simple objects that implement it, is shown in *Fig. 4*.

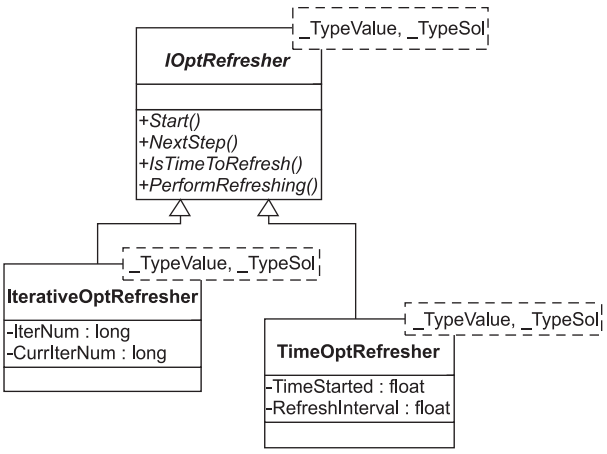

Fig. 4. Class hierarchy for IOptRefresher.

Like in the case of IOptTerminator interface, we define objects that implement the IOptRefresher interface and provide different ways to determine the refreshing period.

IterativeOptRefresher would implement the function IsTimeToRefresh in a way that returns true each time when algorithm has performed some number of iterations. The class TimeOptRefresher implements IsTimeToRefresh in such a way that value true is returned after predetermined period of time has passed.

3.3. The Preservation of Intermediate Results

The approach to this problem is similar to those mentioned above. We define an interface that objects must implement, and then define those objects with appropriate criteria implemented.

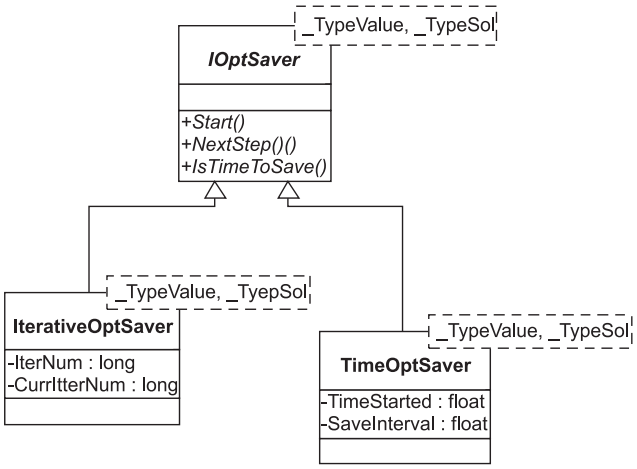

Fig. 5. Class hierarchy for IOptSaver.

Defined object handles only the timing of preservation, i.e. when intermediate results will be saved. Of course, the space for these results must be allocated, which can be done in different ways: we can simply declare one member variable in our class of type OptContResultWithSolutions and save the results by calling AddSolution() member function. Or, we could define a special object to handle intermediate results.

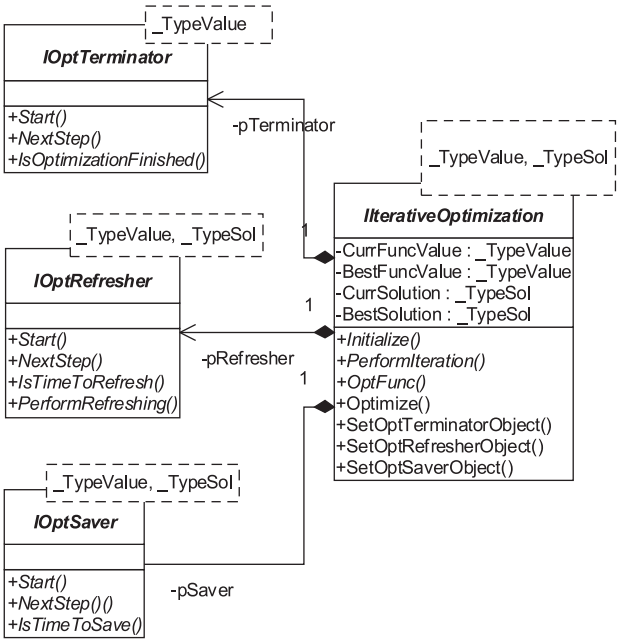

Fig. 6. Class hierarchy for IIterativeOptimization class.

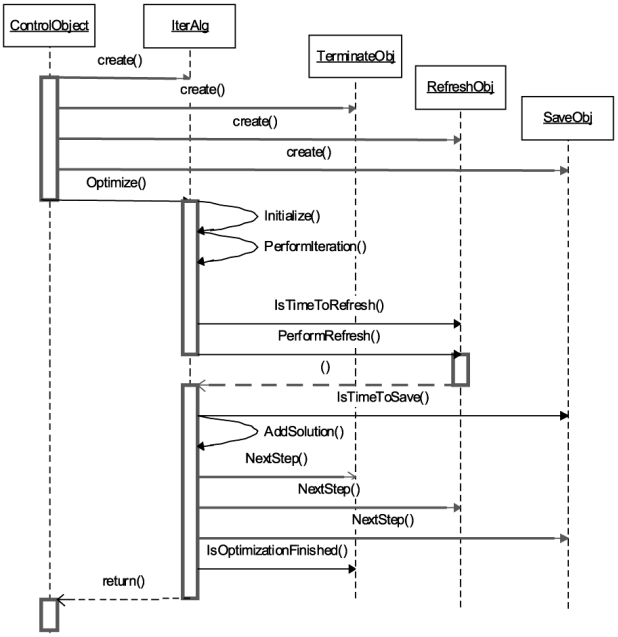

Fig. 7. Interaction between objects in function Optimize.

After defining all these interfaces, we can complete the definition of class IIterativeOptimization. We'll add three member variables, pointers to objects that implement interfaces IOptTerminator, IOptRefresher and IOptSaver. Now the complete class hierarchy is shown on *Fig. 6*. In *Fig. 7*. interaction diagram for function Optimize() is shown.

4. The Application

As an example of usage of the described class hierarchy, a method for solving the quadratic assignment problem can be given. Quadratic assignment problem is defined as

$$\min_{\pi \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} a_{\pi(i)\pi(j)} b_{ij}$$

where $A = (a_{ij})$ and $B = (b_{ij})$ are matrices of dimensions $n \times n$, and S_n is set of all permutations of $\{1, 2, \ldots, n\}$. Quadratic assignment problem first occurred in [4], and it consisted of assigning n facilities to n locations, where matrix a_{ij} represented flow between facilities i and j, and b_{kl} represented distance between locations k and l. Exact methods of solving fail for problem instances with $n > 20$, because of NP-complexity of the problem, so one is directed to the use of iterative algorithms. Most often, QAP is solved with tabu search methods [5], simulated annealing [6], genetic algorithms [7], or some hybrid techniques that combine different approaches.

The tabu search algorithm works on the following principle: the best solution in the neighborhood of current solution is selected, and is chosen as the next solution. If algorithm reaches a local optimum, all solutions in the neighborhood are worse that the current one. In this case the least degrading solution is chosen. To prevent from returning to the local optimum, most recently visited solutions are placed in the tabu-list of forbidden moves, which is implemented as a circular list of predefined length. When the list is full, the oldest move is taken out from the list, which makes the associated solution a candidate for selection again.

In implementation of tabu search optimization routine, solutions were first randomly generated in function Initialize(), which class TabuSearchQAPSolver inherited from interface IIterativeOptimization (see *Fig. 8*). The function PerformIteration implements the code for one step of tabu-search algorithm.

The usage of the framework described in this paper significantly reduced the number of lines of code required to implement the iterative algorithm. The entire algorithm for solving the

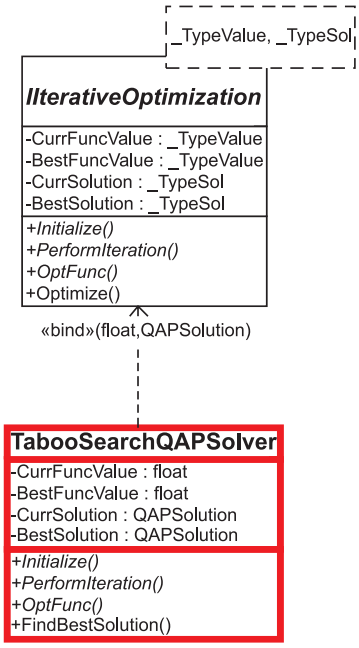

Fig. 8. Class TabuSearchQAPSolution.

quadratic assignment problem with basic tabu-search method, with all features described in the text, was written in 135 lines of code in functions Initialize(), PerforIteration() and FindBestMove().

The presented example is just a fraction of the class hierarchy that was developed by the authors, where various methods of solving the exam scheduling problem at the authors' institution were implemented. Several approaches to this problem, which is a special case of quadratic assignment problems, were tested. Besides the tabu-search, the problem was solved by simulated annealing and genetic algorithms, which were successfully incorporated in the proposed framework. All algorithms were implemented in the same way, as objects implementing the described IIterativeOptimization interface. In this way we achieved the goal of separating implementation of the algorithm from program code for refreshing, saving results and control of duration of algorithm execution. Another advantage of this approach is that new objects can be added to the class hierarchy, implementing some new criteria for refreshing or optimization termination, without changing the implementation of algorithm itself, i.e., algorithm can readily use all new objects as long as they comply with the described interfaces.

5. Conclusion

When implementing any kind of iterative algorithms, it is always necessary to devote a certain amount of time and effort to write the program code that is not directly associated to the actual problem solving, but is common to all iterative algorithms. This code should handle the screen refreshment with the current solution, the intermediate results storage and examination of the stop criteria.

This paper presents a class hierarchy, which can be used as an object-oriented framework for the solutions of the iterative problems. Utilizing the options available in object-oriented languages, it separates the class interface from the implementation. This framework defines the interfaces for the objects, which will implement their own criteria for the common issues described above.

This framework was utilized to facilitate the solution of the exam-scheduling problem at the authors' institution. This problem, which is a special case of the quadratic assignment problem, was solved by several iterative methods. Besides the tabu-search, which is described in this paper, the simulated annealing and various genetic algorithms where implemented. All methods were easily fitted into the described framework, resulting in the significant decrease of number of lines of code and simpler program maintenance.

Work is in progress to extend this framework to enable the multithreaded execution, which would greatly improve framework usability and make it useful for development of applications in domain of numerical optimization.

References

[1] STROUSTRUP, B. (1997), *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts.

[2] AUSTERN, M. H. (1998), *Generic Programming and the STL*, Addison Wesley, Reading, Massachusetts.

[3] COPLIEN, J. O. (1992), *Advanced C++, Programming Styles and Idioms*, Addison-Wesley, Reading, Massachusetts.

[4] KOOPMANS, T. C. AND BECKMANN, M. J. (1957), "Assignment problems and location of economic activities", *Econometrica* **25**, pp. 53–76.

[5] SKORIN-KAPOV J., (1990), "Tabu search applied to the Quadratic Assignment Problem", *ORSA Journal on Computing* **2**, pp. 33–45.

[6] WILHELM, M. R. AND WARD, T. L. (1987), "Solving quadratic assignment problems by simulated annealing", *IIEE Transactions*, pp. 107–119.

[7] AHUJA R. K., ORLIN J. B. AND TIWARI A. (2000), "A greedy genetic algorithm for the quadratic assignment problem", *Computers & Operations Research* **27**, pp. 917–934.

Received: June, 2001
Accepted: September, 2001

Contact address:
Zvonimir Vanjak
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3
10000 Zagreb
Croatia
e-mail: zvonimir.vanjak@fer.hr

Vedran Mornar
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3
10000 Zagreb
Croatia
e-mail: vedran.mornar@fer.hr

ZVONIMIR VANJAK received his B.Sc and M.Sc degrees at the Faculty of Electrical Engineering and Computing, University of Zagreb, in 1997 and 2001, respectively. He works at the same faculty as a researcher and was involved in the development of several projects. His main interests are object-oriented programming and numerical methods.

VEDRAN MORNAR is an Associate Professor of Computer Science at the Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia. He received the PhD degree in computer science from the same university, where he currently runs several graduate and undergraduate computing courses. His academic interest is in operational research and database design and implementation in real world information systems.