Proceedings of the 6th WSEAS Int. Conf. on EVOLUTIONARY COMPUTING, Lisbon, Portugal, June 16-18, 2005 (pp49-54)

OptLets: A Generic Framework for Solving Arbitrary Optimization Problems[*]

CHRISTOPH BREITSCHOPF
Department of Business Informatics – Software Engineering
Johannes Kepler University Linz
Altenberger Straße 69, 4040 Linz
AUSTRIA
http://www.se.jku.at

GÜNTHER BLASCHEK, THOMAS SCHEIDL
Institute of Pervasive Computing
Johannes Kepler University Linz
Altenberger Straße 69, 4040 Linz
AUSTRIA
http://www.soft.uni-linz.at

Abstract: Meta-heuristics are an effective paradigm for solving large-scale combinatorial optimization problems. However, the development of such algorithms is often very time-consuming as they have to be designed for a concrete problem class with little or no opportunity for reuse. In this paper, we present a generic software framework that is able to handle different types of combinatorial optimization problems by coordinating so-called OptLets that work on a set of solutions to a problem. The framework provides a high degree of self-organization and offers a generic and concise interface to reduce the adaptation effort for new problems as well as to integrate with external systems. The performance of the OptLets framework is demonstrated by solving the well-known Traveling Salesman Problem.

Key-Words: Meta-heuristics, Heuristics, Framework, Combinatorial optimization, Incremental optimization, Knapsack Problem, Traveling Salesman Problem, Real-world problems

1 Introduction

Routing and scheduling problems are often used in the optimization community to demonstrate the applicability of newly developed optimization algorithms. Typical examples are the Traveling Salesman Problem (TSP), the Capacitated Vehicle Routing Problem (CVRP), and the Job Shop Scheduling Problem (JSSP). All these problems are combinatorial optimization problems that are classified as NP-hard and therefore hard to solve.

Many different meta-heuristics such as Tabu Search (TS), Simulated Annealing (SA), Genetic Algorithms (GA) and techniques inspired by nature such as Ant Colony Optimization (ACO) [1] and Particle Swarm [2] have been developed in order to overcome the complexity of this problem class and provide feasible solutions in reasonable time. These techniques exhibit some generic aspects, but specific problems must nevertheless be tackled with newly developed adaptations of the basic techniques. These adaptations are often non-trivial and require in-depth knowledge of the problem domain.

Hybrid algorithms try to combine the strengths of several techniques by eliminating their weaknesses. Such hybrid algorithms often outperform their predecessors with respect to performance and the quality of the final solution. Unfortunately, these approaches are often based on traditional algorithms and therefore also require cumbersome and time-consuming adaptation for each concrete problem to be solved.

For practical use, we need a general optimization technique that lets us develop solvers for real-world problems with as little effort as possible. There, the framework concept comes into play. A framework for optimization tasks should be independent from the actual problem at hand, yet flexible and extensible enough to support rapid development of custom solvers for arbitrary problems. It should encapsulate the invariant parts from the problem-specific ones. It should especially take care about

[*] This work was funded by Siemens AG, Corporate Technology, Munich.

administration and monitoring of the optimization process so that the user can focus on the problem-solving tasks without bothering with “administrative issues”. The framework should not be restricted to traditional techniques or certain predefined hybrid approaches; it should rather allow the user to implement any kind of algorithm that is suitable for tackling the concrete problem. The framework should be able to deliver good solutions as fast as possible by incrementally improving the quality of delivered solutions over time, so that the search process can be interrupted at any time.

There exist many different approaches for software frameworks that have been designed to cope with the challenges discussed above. For instance, OpenTS (OTS) [3] and the Tabu Search Framework (TSF) [4] encapsulate the functionality that is common to all TS variants. However, both approaches support only TS. EasyLocal++ [5] extends the ideas of OTS and TSF by supporting the TS as well as the SA meta-heuristic. But it supports only a predefined set of hybridization models so that new schemes cannot easily be integrated into the framework. HotFrame [6] is a more sophisticated framework supporting various meta-heuristics such as TS, SA and Evolutionary Algorithms (EAs). Hybridization is supported by using inheritance and genericity to separate the invariant from the problem-specific parts. HeuristicLab [7] is an optimization environment enabling the user to apply different optimization techniques (e.g. TS, SA, GA) to different problem classes (e.g. JSSP, TSP). However, the user always has to choose an appropriate technique in advance and cannot mix several existing techniques. Compared to the frameworks discussed so far, the A-Team framework [8] offers the highest degree of flexibility in problem-solving. The architecture is based on a network of software agents that work together in order to solve a concrete problem. The framework uses different types of agents that must be implemented for the specific problem.

2 The OptLet Approach

The OptLets framework is implemented in C++ and enables the use of different optimization paradigms as well as the combination of existing techniques. The framework takes care about the whole optimization process by selecting the currently best “technique” and manages the solutions produced over time. The user can concentrate on the problem without caring about any administrative issues. Users can easily create hybrid solvers and include arbitrary heuristic techniques. The effort for adapting the system to a new problem class is rather small, as the user has to implement only those features he really needs. The framework also enables the rapid and stepwise implementation of new problem-solving components by supporting experimental tests during the whole development cycle.

The development of new algorithms can be done in parallel and distributed to several developers. Each team member can contribute his or her ideas independently from others.

3 The Framework Concept

The basic assumption is that many optimization problems share common properties for which general algorithms can be implemented once, thus representing the invariant part. So, the user is able to concentrate on the optimization problem itself and can leave the administrative work to the framework.

The OptLets framework does not know anything about the problem to be solved. The user has to provide a problem description, the representation of solutions and the optimization entities called OptLets for a complete optimization system.

3.1 Solutions

The idea behind the framework is that an arbitrary optimization problem can be solved starting with one initial solution and then creating new solutions based on existing ones.

In the context of the OptLets framework, we use a relaxed definition of the term “solution”, not necessarily meaning a final solution to the problem. It would be actually more precise to speak about “candidate solutions” because a “solution” might be far from optimal or even trivial (e.g. an empty knapsack), incomplete or invalid.

The framework provides an abstract class for solutions. Value and validity are the key properties and must be specified by a problem-specific concrete solution class. The value specifies the solution quality (e.g. TSP tour length, profit of a knapsack). The invalidity describes how much a solution violates the given constraints. This allows the framework to compare invalid solutions by their “violation degree”. Depending on the problem, such invalid solutions are allowed as they might be good starting points for further improvement.

During the optimization process, many solutions are generated. Similar to the population-based approach of the Evolutionary Computation paradigm

[9], the OptLets framework keeps the solutions in a “solution pool”. Solutions in the pool are considered as read-only in the sense that an existing solution cannot be modified. Whenever a solution is used as a starting point, the framework creates a copy of the original solution and assigns it to a so-called OptLet that modifies this copy according to its optimization strategy. The new solution is then put back into the pool as another starting point for other OptLets.

The solution pool has a limited capacity so that it must be cleaned up occasionally. Whenever the pool becomes full, the framework evaluates all solutions and keeps only those that might be useful in the next iteration (where “iteration” is defined as the time between two clean-ups). The framework also keeps some invalid solutions as they can be modified to become valid. After the clean-up, the optimization process resumes by selecting existing solutions and assigning them to OptLets.

3.2 OptLet

An OptLet can be defined as problem-solving or optimization entity that produces a new solution based on an existing one.

The framework provides an abstract OptLet class containing those features that are common to all types of optimization problems. OptLets are always implemented for a concrete problem and contain optimization strategies how to modify and/or create a solution. In the sense of object-oriented programming, OptLets are strategy objects exhibiting a certain behavior defined by the user.

As the design and functionality of an OptLet is up to the user, it may represent different roles for solving an optimization problem. It could represent an external entity such as a customer who wants to have his order processed as early as possible. Or, it could concentrate on a particular aspect of a solution in order to improve a certain objective. An OptLet may also represent a constraint watcher, trying to fix invalid solutions. In general, an OptLet can be seen as an algorithm that “does something” to a solution. This operation can be anything from a small change to a complete heuristic algorithm for finding a good solution in a single step. Independently from the actual operation, the framework repeatedly selects an existing solution and lets an OptLet modify it.

The success of an optimization process depends on the combination of OptLets being used. The goal of the framework is to compute the best possible solution as fast as possible. So, selecting an OptLet is an important task of the framework as it has to decide which OptLet fits best for working on a given solution. The framework monitors the work of the OptLets by evaluating their success regarding the improvement of the solution quality. It selects more successful OptLets with a higher probability than less successful ones. In other words, the framework learns from past achievements and attempts to extrapolate them into the future.

3.3 Architecture

The OptLets framework uses abstraction and inheritance as primary mechanisms to build problem-specific components. The architecture encapsulates all framework internals against access from outside and provides narrow interfaces for connecting the problem-specific components as well as for communication with external systems.

Fig. 1 shows a simplified diagram of the major components of the OptLets framework.

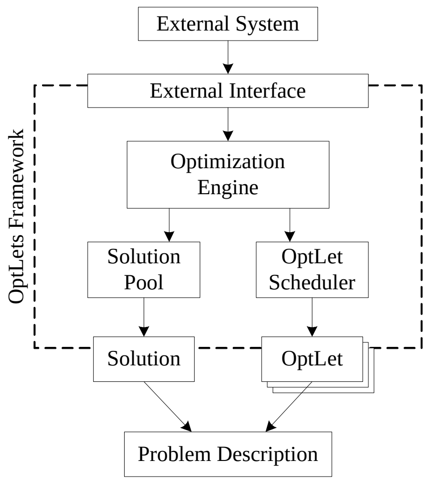

Fig.1: Architecture of the OptLets framework

The *Optimization Engine* is the central component of the OptLets framework. It controls the optimization process and represents the key connection point for external systems. As an OptLet works autonomously and has no knowledge about other OptLets, the framework administrates them via the *OptLet Scheduler*. This component coordinates the OptLets by distributing the work among them and deciding which OptLet should work on a given solution. The *Solution Pool* contains all solutions generated during an iteration and takes care of the clean-up process when the pool becomes full.

The OptLets framework provides several interfaces for connecting the problem-specific components. The abstract class *Solution* is used for specifying the representation of the solutions (data structures, operations). The *OptLet* component represents a generic interface for developing the individual OptLets. Typically, several OptLets are implemented for solving a concrete problem. The

Problem Description contains information about the problem to be solved (e.g. data structure, constraints) and is used by the *Solution* as well as by the *OptLet* components for gathering information in order to modify a solution.

For starting and controlling the optimization process, the OptLets framework provides an *External Interface*.

4 Using the Framework

Developing a problem-specific OptLets optimizer requires the following parts to be implemented:

- Embedding into an external system
- Representation of the problem description
- Representation of solutions and operations to manipulate solutions
- OptLets

After the foundation (i.e., at least a simple user interface and representations of the problem description and the solution) has been implemented, the OptLets can be developed using evolutionary prototyping. One will typically start with a few simple OptLets, experiment with them and evaluate the results. Based on this experience, more OptLets can be added until the results are satisfying.

4.1 Foundation

Some kind of *user interface* is needed to run the optimizer. This can be anything from a simple command-line interface to a sophisticated GUI. The optimizer can also be embedded in larger systems.

Problem instances are represented by a *problem description* object. The framework does not know anything about the internal structure of this object; it just passes the object along to the OptLets. Defining an appropriate data structure and filling it with values (e.g. from a file) is up to the implementer of a problem-specific optimizer.

For the *representation of solutions*, it is necessary to define an appropriate data structure together with specific operations to manipulate a solution. In addition to that, a few operations required by the framework must be implemented. This includes methods for obtaining the solution value and degree of invalidity, comparing two solutions for equality and creating a copy of a solution object.

4.2 OptLets

An OptLet is implemented by deriving a concrete class from an abstract class in the framework and overriding the *work* method. This method receives a solution object from the framework and is expected to modify that solution in some way.

Depending on the problem, there are many different ways how an OptLet can modify a solution. OptLets can make very simple modifications (e.g. pack in an item into a knapsack or swap two locations in a tour) or encapsulate more complex algorithms (e.g. eliminate intersections in a tour). Usually, an OptLet will modify a solution in a way that is expected to improve some aspect of the solution, but it may also be reasonable to let OptLets apply random modifications (in order to escape from local optima) or even add OptLets that deliberately deteriorate solutions, hoping to create useful starting points for other Optlets.

4.3 Strategies for developing OptLets

It is recommended to start with simple OptLets. In many cases, already a few simple OptLets lead to acceptable results. Analyzing the results might give ideas for further, maybe more sophisticated OptLets. Generally, OptLets should not contain too complex algorithms. Optimization is done by letting all these simple OptLets work on solutions produced by other OptLets, controlled by the framework. It is desirable to obtain a high rate of OptLet invocations, which can only be achieved if individual OptLets do not consume too much time.

It is possible to implement special starting OptLets that are called only once or a few times at the beginning of the optimization and provide better initial solutions. These OptLets will typically use some sort of greedy strategy.

OptLets are well suited for being developed in a team, as they are completely independent from each other. As the success of an OptLet can be estimated by looking at statistics generated by the framework, developers can organize an “OptLet contest”, striving to create the most successful OptLets.

5 Case studies and results

We tested the framework with the well-known Knapsack Problem (KP) and Traveling Salesman Problem (TSP). Our primary goal was to show that good results can be achieved for these problems with simple OptLets, not to reach the global optimum or to outperform existing specialized solvers. The main focus of the framework are real-world problems that are much more complex and often difficult to handle with traditional techniques.

5.1 Traveling Salesman Problem

The TSP is one of the best investigated combinatorial optimization problems. Many sample problems with proven optimal solutions can be found in TSPLIB ([10]). So, we chose the TSP to evaluate the results of our framework.

5.1.1 Implementation aspects

The solution representation is a permutation of the locations, numbered from 1 to *n*. This representation does not allow invalid solutions. The initial solution just contains all locations in ascending order.

We implemented 26 OptLets which can be divided in three categories:

- *Swapping OptLets* swap two locations. The OptLets differ in how they choose the locations to be swapped (e.g. by random, adjacent locations with the greatest distance, etc.)
- *Shifting OptLets* shift a location or a sequence of locations within a tour, using different strategies.
- *Intersection removing OptLets* try to detect and remove intersections within a tour, using different strategies.

Furthermore, a starting OptLet produces solutions using a “nearest neighbor” heuristic, starting with a different location on each call, resulting in *n* different solutions for a problem with *n* locations.

A class hierarchy is used to group similar OptLets and concentrate common operations (e.g. finding an intersection) in abstract superclasses. No OptLet class (including the abstract superclasses) contains more than 80 lines of code; most contain only about 30 lines (not counting the header files).

5.1.2 Results

Generally, when plotting the evolution of the best solution value over time, we can observe a steep ascent in the beginning and slight improvements later on. Figure 2 shows a typical evolution of the solution value (the first 20 seconds for the *ch150* problem instance).

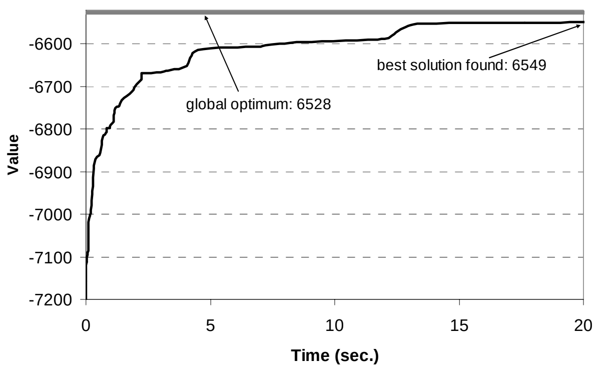

Fig.2: Evolution of the solution value

Note that the tour length is represented as a negative value. This is because the OptLet framework always tries to maximize the solution value. The horizontal line at the top represents the global optimum.

For the results presented in this paper (see Table 1), we ran the optimizer 10 times per problem instance and for 3 minutes per run. We then looked at the intermediate results after 10 seconds, after 1 minute and the final result after 3 minutes. The table shows how much the average values reached lie above the known optimum. The column “Opt.” shows in how many of the 10 runs the optimizer found the global optimum. For problems where the global optimum was reached on some runs, the average time needed to reach the optimum is given. The results were obtained on a Pentium 4 2.4GHz computer with 1024 MB RAM on Windows XP.

Problem	10s	60s	180s	Opt.	Time
berlin52	0%	0%	0%	10	0.39
eil101	0.70%	0.48%	0.41%	5	28.00
lin105	0.79%	0.28%	0.07%	8	48.11
ch150	0.62%	0.32%	0.32%	0	
ts225	2.03%	1.29%	1.26%	0	
lin318	6.41%	3.73%	2.74%	0	
pcb442	11.25%	6.21%	4.25%	0	
p654	15.86%	8.99%	5.43%	0	

Table 1: TSP results

For the smaller problems (up to 150 locations), we come very close to the optimum (<1%) within the first 10 seconds. The bigger the problem, the more time the OptLets need to work on a solution. For the largest problem listed in the table (654 locations), the average value after 3 minutes is about 5% above the optimum.

5.2 Real-world problems

So far, we tested the OptLets framework with two real-world problems and compared the results with existing solvers that are used in practice.

The first problem was optimizing the movements of a robot that places objects in a room. Here, the OptLets optimizer is able to produce solutions that are better than the ones produced by the practically used algorithm in most cases and comparable to the solutions produced by another specialized heuristic optimizer.

The second problem was the optimization of a production process in a steel mill. The optimizer is responsible for controlling the speeds of different aggregates and has to deal with several objectives (maximize the throughput, minimize speed changes) that are combined into a weighted sum to be maximized. Testing the optimizer with a simulator showed that our results are comparable to that of the currently used and well-proven (LP-based) solver.

These two problems were chosen to be able to compare the OptLets framework with existing, practically used solvers. As the results are quite promising, the OptLets framework is a serious option for future real-world applications.

5.3 Implementation effort

Table 2 gives an overview about the number of OptLets used for each problem and the proportion of the problem-specific part (measured in net LOC) opposed to the invariant framework part for each optimizer.

Problem	OptLets	specific	invariant
Knapsack Problem	14	9.5%	90.5%
Traveling Salesman	26	17.3%	82.7%
Robot	14	23.8%	76.2%
Steel Mill	27	35.8%	64.2%

Table 2: Implementation effort

The portion of the problem-specific parts becomes larger for more complex problems, but it still is much smaller than the problem-invariant part, represented by the framework.

6 Conclusion

In this paper, we presented OptLets, a generic framework for solving combinatorial optimization problems. The main benefit of the framework is that optimizers for arbitrary problems can be developed with little effort.

The central parts of an optimizer are OptLets that modify a solution in some way. OptLets can be very simple (and thus easy to implement). The optimization is done by letting many different OptLets work on a pool of solutions. Management of the solutions and selection of OptLets is done by the framework and does not bother the implementer of an OptLet.

First results can be achieved quickly, after developing a few simple OptLets. Then, developing an optimizer means adding more OptLets or tweaking existing ones until the results are satisfying.

OptLets are completely independent from each other and are therefore well suited for being developed in a team.

Experiments with both academic and real-world problems show that it is possible to achieve satisfying results with relatively simple OptLets. For two real-world applications, the OptLets-based optimizer was able to compete with existing solvers. This suggests that OptLets can be a promising alternative for dealing with hard combinatorial optimization problems, saving valuable development time.

References:

[1] Dorigo M., Stützle T., *Ant Colony Optimization*, The MIT Press, 2004.

[2] Parsopoulos K., Vrahatis M., Particle Swarm Optimization Method for Constrained Optimization Problems, In Sincak P., Vascak J., Kvasnicka V., Pospichal J. (Eds.), *Intelligent Technologies – Theory and Application: New Trends in Intelligent Technologies*, Vol. 76 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2002, pp. 214-220.

[3] Harder R., *OpenTS – Java Tabu Search Framework*, Online: http://opents.iharder.net, 2001.

[4] Lau H. C., Wan W. C., Jia X., A Generic Object-Oriented Tabu Search Framework, *Proceedings of the 5th Metaheuristics International Conference, (MIC'03)*, 2003, pp. 362-367.

[5] Gaspero L., Schärf A., EasyLocal++: An object-oriented framework for the flexible design of local-search algorithms, *Software: Practice and Experience*, Vol. 33, 2003, pp. 733-765.

[6] Fink A., Voß S., HotFrame: A Heuristic Optimization Framework, In Voß, S., Woodruff, D. (Eds.), *Optimization Software Class Libraries*, Kluwer Academic Publishers, 2002, pp. 81-154.

[7] Wagner S., Affenzeller M., HeuristicLab: A Generic and Extensible Optimization Environment, *Proceedings of the International Conference on Adaptive and Natural Computing Algorithms (ICANNGA)*, 2005.

[8] Rachlin J., Goodwin R., Murthy S., Akkijaru R.; Wu F., Kumaran S., Das R., A-Teams: An Agent Architecture for Optimization and Decision Support, *Lecture Notes In Computer Science. Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*, Springer, 1998, pp. 261-276.

[9] Goldberg D., *The Design of Innovation*, Kluwer Academic Publishers, 2002.

[10] TSPLIB, Online: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95