Effective Optimizer Development for Solving Combinatorial Optimization Problems*

GÜNTHER BLASCHEK, THOMAS SCHEIDL
Institute of Pervasive Computing
Johannes Kepler University
Altenberger Straße 69, 4040 Linz
Austria
http://www.pervasive.jku.at

CHRISTOPH BREITSCHOPF
SCR SE
Siemens Corporate Research, Inc.
755 College Rd East, Princeton, NJ 08540
USA
http://www.scr.siemens.com

Abstract: Combinatorial optimization problems (COPs) are well-known in the optimization community and can be solved by many different techniques. Beside traditional ones, optimization frameworks offer an effective way to solve different types of optimization problems by providing a generic infrastructure that is used to develop problem-specific optimizers. In this paper, we present the development of a problem-specific optimizer for solving the symmetric Traveling Salesman Problem (TSP) using the OptLets Optimization framework** [1]. We also discuss upcoming ideas and reasons in the context of this development process.

Key-Words: Meta-heuristics, Heuristics, Framework, Combinatorial optimization, Incremental optimization, Traveling Salesman Problem.

1 Introduction

Combinatorial optimization problems influence many areas in science and industry. For example, Genetic Algorithms [2], Simulated Annealing (SA) [3] and Ant Systems [4] represent some meta-heuristics that have been developed in the past decades in order to solve such problems. These techniques contain some generic aspects, but for every problem class or variation of a problem new algorithms must be developed. This development process is often very time-consuming so that the advantages of the supporting techniques are mostly lost.

Beside these traditional techniques, several optimization frameworks have been developed that try to reduce the effort required for implementing problem-specific solvers. OpenTS [5] and the Tabu Search Framework (TSF) [6] use Tabu Search (TS) [7]. EasyLocal++ [8] additionally supports SA. The Meta-heuristics Development Framework (MDF) [9] is an enhancement of TSF and supports different meta-heuristics such as Evolutionary Algorithms (EAs) [2], Ant Colony Optimization (ACO) [4] etc.

The HotFrame framework [10] allows the user to develop algorithms based on TS, SA and EAs. HeuristicLab [11] represents an optimization environment where different optimization techniques (e.g. TS, SA, GA) can be applied to different problem classes (e.g. TSP).

In this paper, we demonstrate how COPs can be solved using the OptLets framework [1], using the TSP as an example. The framework offers a novel way to develop algorithms that are not based on existing optimization techniques. We describe the activities for developing an OptLets-based TSP solver and demonstrate ideas for effectively develop an OptLets-based optimizer. Even if other problems might be different, the basic procedure for implementing an OptLets-based optimizer will always be the same.

* This work was funded by Siemens AG, Corporate Technology, Munich.
** Patent pending.

2 OptLets Optimization Framework

The OptLets optimization framework is a software framework written in C++ that is able to solve different types of optimization problems.

The basic idea behind the framework is that many optimization problems share common properties that can be encapsulated in an immutable part. The framework administrates all problem-independent tasks such as monitoring the optimization process, invoking appropriate optimization operators, managing solutions etc. so that the user can concentrate on the problem-solving itself without considering any administrative issues.

The problem-specific part consists of the problem description, the representation of solutions and the operators called OptLets. The problem description contains information about the problem to be solved and the solution representation defines a data structure storing the solutions produced by the OptLets. OptLets must always be implemented for a certain problem. As the framework is independent of any existing optimization paradigms, the user can implement any algorithms that are suitable for solving the problem. Nevertheless, traditional techniques can also be used and combined with each other.

An optimization process typically starts with at least a single initial solution. The OptLets framework controls the optimization process by repeatedly invoking OptLets in order to select the best suitable algorithm. OptLets always create new solutions based on existing ones so that an already existing solution can never be overwritten by an OptLet.

The framework also monitors the work of the OptLets from time to time during an optimization run. Each OptLet has a score representing its success regarding the improvements of the solution quality in the past. The higher the score of an OptLet, the higher is its probability to be selected by the framework to work on a given solution. So, the combination of OptLet invocations influences the quality of the final solution.

Solutions are stored in a solution pool with a limited capacity. Whenever the pool becomes full, the framework evaluates all solutions in the pool by determining the quality of each solution and performs a clean-up where only suitable solutions are kept for the future. Suitable solutions are good valid solutions as well as invalid solutions (if they are defined for the underlying problem) that violate the existing constraints only to a small degree.

As the framework always knows the quality of each solution in the pool, the user is able to ask for the current best solution at any time during the optimization. He does not need to wait until the optimization is finished.

3 Solving the Traveling Salesman Problem

Implementing an OptLets-based optimizer consists of the following steps:

- Define the appropriate data structures.
- Implement the OptLets.
- Implement the user interface and the interface to external systems.

In this paper, we describe the first two aspects, i.e. the data structures and OptLets used for the Traveling Salesman Problem.

3.1 Defining Problem-specific Data Structures

Whenever a specific problem shall be solved, some data structures must be provided to store information about the problem and the representation of the solutions. The internals of these data structures always depend on the underlying problem and can not be defined in general. But, it is recommended to have an eye on using efficient data structures as their access methods will be called very frequently by OptLets during the optimization process.

For solving the TSP, the problem description just contains a distance matrix of all locations to be visited. Solutions are represented by an array containing a permutation of all locations. As we define that no invalid solutions are allowed (i.e. it is not allowed to skip a location or to visit the same location twice), a solution always represents a complete tour.

3.2 OptLets

Implementing the OptLets is typically an incremental process: One starts with very simple OptLets that modify a solution in some way (e.g. OptLets that just swap two locations using a simple strategy). If the results are not yet good enough, additional, possibly more sophisticated OptLets, can be added. This is done until the results are good enough for the given task. There are no restrictions about the modifications OptLets can apply to a solution – one may experiment with any operations that come to mind.

In our example, the TSP optimizer contains 26 OptLets that can be divided into the following categories:

- Swapping OptLets (6)
- Shifting OptLets (12)
- Intersection OptLets (5)
- Starter OptLets (1)
- Other OptLets (2)

3.2.1 Swapping OptLets

Swapping OptLets swap two locations within a tour. The different swapping OptLets use the following different strategies:

- Select two locations randomly and swap them.
- Randomly select a location and swap it with its successor.
- Find the two adjacent locations A and B with the greatest distance and swap them.
- Find the two adjacent locations A and B with the greatest distance and swap B with its successor.
- Find the two adjacent locations A and B with the greatest distance and swap B with the location nearest to A.
- Try to find two adjacent locations A and B so that swapping these two locations decreases the total tour length.

3.2.2 Shifting OptLets

There are actually two types of shifting OptLets, those that shift a single location and those that shift an entire section. Some of the OptLets shifting an entire section additionally reverse the order of the locations in the section.

The following strategies are used by OptLets that shift a single location:

- Randomly select a location and shift it to a random position.
- Randomly select a location A, find the location B nearest to A and shift B right after A.
- Randomly select a location A, find the location B with the n-th lowest distance to A and shift B right after A. There are four OptLet instances for n=2 to n=5.

OptLets that shift an entire section always select the section randomly, i.e. beginning at a random location and containing a random number of locations. They differ in how they determine the position where the section is shifted to and whether they reverse the order of the locations within the section. The following different strategies are used:

- Shift the section to a random position.
- Shift the section to a random position and reverse the locations within the section.
- Reverse the locations within the section and try to find a position such that shifting the section to that position decreases the total tour length.
- Reverse the locations within the section and try to find a position such that shifting the section to that position does not increase the total tour length by more than 5%.
- Reverse the locations within the section and shift the section to the position (different from the original position) where the resulting tour length is as small as possible (albeit possibly greater than the original tour length).

3.2.3 Intersection OptLets

As it turned out during our first experiments, the swapping and shifting OptLets often created tours with many intersections. Therefore, we implemented specific intersection OptLets searching for intersections within the tour and trying to remove them. Each of these OptLets first searches for an intersection between two lines A-B and C-D. If the tour does not contain any intersection, the OptLet rejects the solution. Otherwise, it tries to remove the intersection. How this is done, depends on the particular OptLet. The following strategies are used:

- Swap the beginning of the first with the end of the second line, i.e. A and D.
- Swap the end of the first with the beginning of the second line, i.e. B and C.
- Swap A with its predecessor and C with its predecessor.
- Swap B with its successor and D with its successor.
- Reverse the order of all locations between B and C.

3.2.4 Starter OptLets

As the initial tour used in the TSP optimizer contains just the locations in the same order as they appear in the input data, the quality of the initial solution (its tour length) is usually very bad. It then takes quite a long time until the OptLets find a reasonable solution. This leads to the idea of so-called “Starter OptLets” that use a greedy heuristic in order to create better initial solutions.

For the TSP, there is one such OptLet in this category that constructs tours using a nearest neighbor heuristics. This OptLet constructs as many tours as there are locations, so that each location is used once as starting point for the tour, as different starting points typically lead to different tours using this heuristic. After this OptLet has constructed all possible tours, it deactivates itself and is no longer called by the framework.

3.2.5 Other OptLets

The following two OptLets do not fit in any of the categories described above. Both OptLets are pure random OptLets:

- Randomly reverse the order of the locations within a section of the tour.
- Randomly permute all locations in the tour.

These OptLets will typically not create very good solutions, but they might help to escape from local optima.

4 Analyzing the Results

The results achieved by the OptLets TSP optimizer have already been published in [1]. This paper focuses on analyzing the results, i.e. what impact particular OptLets have on the result and how the results differ when disabling certain OptLets.

All comparisons use the results obtained by a one minute run with all OptLets enabled as a reference. Table 1 shows the results for five problem instances taken from the TSPLIB [12]. For each problem, we ran the optimizer 5 times and computed how much the average value lies above the known optimum. The results were obtained on a Pentium D 3GHz computer with 1024 MB RAM on Windows XP x64 edition.

Problem	Result
eil101	0%
lin105	0.12%
ch150	0.32%
ts225	1.44%
lin318	2.98%

Table 1: Reference results

Note that the results for most problems are better than those published in [1], due to the faster machine and some code optimizations in the framework.

4.1 Impact of the Starter OptLet

The Starter OptLet using a nearest neighbor heuristics has the purpose to create better initial solutions so that good solutions can be found considerably faster. Test runs shall show the actual benefit of this OptLet. Table 2 shows a comparison of the results with and without the Starter OptLet:

Problem	with Starter OptLet	without Starter OptLet
eil101	0%	1.81%
lin105	0.12%	2.10%
ch150	0.32%	1.48%
ts225	1.44%	3.51%
lin318	2.98%	6.41%

Table 2: Impact of the Starter OptLet

This shows that the Starter OptLet has a significant impact on the quality of the solution. Without the Starter OptLet, it takes much longer to find decent solutions. Figure 1 shows the evolution of the solution value for the lin318 problem with and without the starter OptLet:

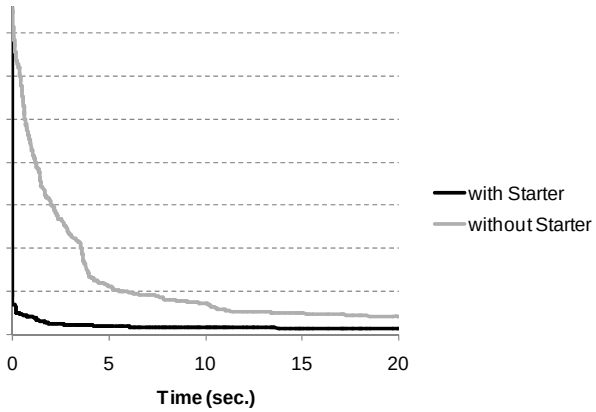

Fig.1: Evolution with and without the Starter OptLet

With the Starter OptLet, there is a rapid improvement in the beginning. The other OptLets then continue to improve the initial solutions. Otherwise, the OptLets have to start working from a far inferior initial solution.

4.2 Other important OptLets

It is not obvious which OptLets have a big impact on the quality of the solution and which do not. However, hints about the success of an OptLet might be gathered by analyzing how often it is called by the framework. If an OptLet is called frequently, this is because it has a high score, i.e. it contributed in finding good solutions. The most frequently called OptLets should therefore be those OptLets that are most important for the success of the optimizer.

When we analyzed the OptLet statistics for the 1 minute TSP optimizer runs, we found out that the most frequently called OptLet for all tested problem instances is the one that tries to find two adjacent locations so that swapping these two locations decreases the total tour length.

To see how this OptLet actually influences the results, we ran the optimizer without this OptLet. Table 3 compares the results:

Problem	with	without
	"best" OptLet	
eil101	0%	0%
lin105	0.12%	0.29%
ch150	0.32%	0.34%
ts225	1.44%	1.52%
lin318	2.98%	5.06%

Table 3: Impact of the most frequently called OptLet

The results become worse without the OptLet in all cases except the first where the optimum could still be found without the OptLet. The biggest impact can be observed for the lin318 problem.

Other OptLets that seem to be important according to the calling frequency are the intersection OptLets. The number of calls for this OptLet category is above average in most test runs. Table 4 shows how the results change when all intersection OptLets are disabled:

Problem	with	without
	intersection OptLets	
eil101	0%	0.25%
lin105	0.12%	0.42%
ch150	0.32%	0.34%
ts225	1.44%	2.29%
lin318	2.98%	5.97%

Table 4: Impact of intersection OptLets

Disabling the intersection OptLets leads to a deterioration in all cases. The bigger the problem, the bigger the difference becomes. This shows that these OptLets are important for the quality of the result.

5 Conclusion

In this paper, we described how to use the OptLets optimization framework for solving a concrete optimization problem, in our case the TSP. Solving an optimization problem using the OptLets framework includes the definition and implementation of appropriate data structures for the problem description and the solutions as well as the implementation of OptLets. The TSP optimizer contains 26 OptLets and produces satisfying results for small and medium size problems. The OptLets consist of a total of about 1000 lines of code, the input and solution data representations including the operations add another 400 lines.

The analysis of the results showed that "Starter OptLets" are important for greatly reducing the time needed to find reasonable solutions. This is an experience we also made during the implementation of optimizers for other problems.

Furthermore, analyzing the calling frequency of the OptLets showed that disabling the most frequently called OptLets leads to a deterioration of the results. So, these OptLets are actually important and it is therefore comprehensible that they are preferred by the framework's evaluation mechanism.

References:

[1] Breitschopf, C.; Blaschek, G.; Scheidl, T.: OptLets: A Generic Framework for Solving Arbitrary Optimization Problems, *WSEAS Transactions on Information Science and Applications* (Special Issue: Selected papers from the 6th WSEAS Int. Conference on Evolutionary Computing, Lisbon, Portugal, June 16-18, 2005), 2005, pp. 501-506.

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

[3] Laarhoven, P. J. M. v.; Aarts, E. H. L.: *Simulated Annealing: Theory and Applications*, Kluwer Academic Press, 1988.

[4] Dorigo, M.; Stützle, T.: *Ant Colony Optimization*, MIT Press, 2004.

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

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

[7] Glover, F.: Tabu Search - Part I, *ORSA Journal of Computing*, 1/1989, pp. 190-206.

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

[9] Lau, H. C.; Wan, W. C.; Halim, S.; Toh, K.: A software framework for fast prototyping of meta-heuristics hybridization, *International Transactions in Operational Research*, Vol. 14, Issue 2, 2007, pp. 123–141.

[10] 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.

[11] Wagner S., Affenzeller M., HeuristicLab: A Generic and Extensible Optimization

Environment, *Proceedings of the International Conference on Adaptive and Natural Computing Algorithms (ICANNGA)*, 2005.

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