4th WSEAS Int. Conf. on COMPUTATIONAL INTELLIGENCE, MAN-MACHINE SYSTEMS and CYBERNETICS Miami, Florida, USA, November 17-19, 2005 (pp12-17)

# Adaptive Guidance of the Search Process in Evolutionary Optimization*

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:* Evolutionary optimization is a well-known paradigm for solving large-scale combinatorial optimization problems. Evolutionary algorithms typically consider the fitness of solutions to decide which solution should be processed by an operator. In the presence of multiple operators to choose from, similar strategies are needed to choose an appropriate operator. In this paper, we present an adaptive target-oriented approach for evaluating and selecting operators on the fly. This technique has been integrated into the OptLets framework** [1], which monitors the success of operators and uses the results of this evaluation for operator selection in the future. Although this paper describes the technique and illustrates the results in the context of the OptLets framework, the evaluation strategy is applicable for other population-based optimization systems as well.

*Key-Words:* Framework, Combinatorial optimization, Incremental optimization, Evaluation, Selection, Population, Knapsack Problem, Operator selection, Target-oriented optimization

## 1 Introduction

Many optimization problems can be classified as NP-hard. For solving such problems, several optimization techniques such as Tabu Search (TS) [2], Simulated Annealing (SA) [3], Genetic Algorithms (GA) [4] as well as nature-inspired approaches like Ant Colony Optimization (ACO) [5] and Particle Swarm Optimization (PSO) [6] exist that try to find a feasible solution in a reasonable time.

All these techniques have in common that they explore the search space in order to improve existing solutions. Based on a given solution, each technique has to decide which move is best for achieving a good solution at the end of the optimization. How the neighborhood of a solution is explored, depends on the used optimization technique.

GA, ACO and PSO are promising techniques that work with a population in order to achieve a feasible solution. Here, finding an appropriate "operator" (which performs a transformation or move) is a non-trivial task that seriously influences the quality of the final solution.

For practical use, we need an approach that is able to guide the search process independently from the underlying problem and the used optimization technique in order to find good solutions.

Several software frameworks have been developed that try to make smart guesses in order to choose an appropriate starting solution and to perform a move that improves that solution.

The *Meta-heuristics Development Framework* (MDF) [7] is an enhancement of the Tabu Search Framework (TSF) [8] and enables the use of different meta-heuristics such as EA, ACO etc. MDF provides interfaces for moves and constructing the neighborhood. These interfaces do not deal with the actual heuristic, but they provide a common infrastructure in which different techniques can share information and collaborate. A centralized control mechanism enables the user to guide the

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

search process based on problem-specific decision handlers. *HotFrame* [9] is an optimization framework that supports TS, SA and Evolutionary Algorithms (EAs). It also provides a problem-independent implementation of neighborhood operations (e.g. shifting or swapping moves), but it does not provide any control mechanism, i.e. the user is not able to adaptively guide the search process. *HeuristicLab* (HL) [10] is an optimization environment in which various optimization techniques (e.g. TS, SA, GA) can be applied to different optimization problems (e.g. JSSP, TSP). HL provides a basic implementation for evaluating the fitness of a solution, but it still requires additional problem-specific components. The *DREAM* (Distributed Resource Evolutionary Algorithm Machine) framework [11] is a peer-to-peer software infrastructure that allows the development of EAs. It provides a so-called evolution engine to handle any population of objects that have a fitness attribute. The framework handles the selection and replacement of individuals within the population. *ParadisEO* (Parallel and Distributed Evolving Objects) [12] is an extension of the EO framework [13] and supports the design of Local Search based algorithms and EAs. It provides basic functionality for the selection and replacement of solutions, but requires additional components to define how the neighborhood should be explored. The *A-Team* framework ([14], [15]) uses a network of software agents that work together for solving a concrete problem. A so-called evaluator agent is used for evaluating the solutions. This agent type provides an arbitrary evaluation function for solutions.

The aforementioned approaches encapsulate some basic aspects of the evaluation and selection of solutions in the search space. They partly provide some generic implementation for the operator selection. Nevertheless, problem-specific components are needed in order to make a decision for a concrete problem and optimization technique.

## 2 The OptLets Framework

The *OptLets* framework [1] is a generic software framework that is able to handle different types of combinatorial optimization problems. It is implemented in C++ and enables the use of different optimization paradigms.

The basic assumption is that many optimization problems share common properties for which general algorithms can be encapsulated in an invariant part. The framework takes care about all administrative issues such as monitoring the optimization process, invoking appropriate optimization "techniques" (operators) and keeping track of the population, so that the user can concentrate on the actual problem-solving task.

For solving a concrete problem, the user has to provide a problem description that contains information about the problem to be solved, the representation of the solutions as well as the optimization entities called OptLets. The rest is entirely done by the OptLets framework.

Solving a concrete problem, the framework starts with a small set of initial solutions. It then invokes OptLets to produce new solutions based on existing ones. Solutions can be valid or invalid. Invalid solutions violate at least one constraint. The quality of invalid solutions is determined by a constraint violation degree. The solution with the lowest violation degree is the minimum invalid solution. During an optimization, many (valid and invalid) solutions are produced.

According to the principle of Evolutionary Computation [3], the framework administrates the solutions in a solution pool. Solutions in the pool cannot be modified. If a new solution should be generated based on an existing one, the framework makes a copy and assigns this copy to an OptLet.

The solution pool has a limited capacity so that the framework must clean it up occasionally. Whenever the pool becomes full, the framework evaluates all solutions and keeps only those that might be useful during the next iteration (where the term iteration is defined as the time between two clean-ups). The framework also keeps invalid solutions as well as solutions not yet considered as they might become useful in the next iteration. All other solutions are discarded and the optimization process continues by selecting solutions from the pool and assigning them to OptLets.

An OptLet is defined as a problem-solving or optimization entity that produces new solutions based on existing ones. An OptLet can represent any kind of algorithm that modifies a solution in some way. OptLets must always be implemented specifically for a concrete problem.

The OptLets framework has successfully been used for solving academic problems (e.g. KP, TSP) and real-world problems (e.g. steel mill production process optimization, object placement by a robot).

## 3 OptLet Selection

The success of the optimization depends on the combination of OptLets used for the underlying problem. The key question is which OptLet should

be assigned to a given solution. In general, the framework selects the OptLets randomly. The basic assumption is that OptLets that have been successful in the past will also be successful in the future. This leads to the idea of a probabilistic approach where each OptLet has a different selection probability, based on its success in the past.

The framework evaluates the success of the OptLets after each iteration. This is done by assigning each OptLet a "score" that reflects its success during the optimization – the higher the success, the higher the score. The selection probability $P_i$ of an OptLet *i* depends on its score $S_i$ and is calculated by dividing its score by the sum of all *n* OptLet scores $S_1$ to $S_n$:

$$P_i = \frac{S_i}{\sum_{j=1}^{n} S_j} \quad (1)$$

At the beginning of the optimization, the framework assigns each OptLet the same score. As the success of an OptLet can vary over time, its score and therefore its selection probability might also change. Some OptLets might perform well in the beginning but could fail to produce good solutions in a later optimization phase, which results in different scores and different probability distributions over time, as shown in Fig.1:

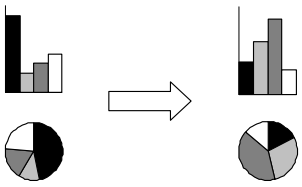

Fig.1: OptLet scores and selection probabilities

# 4 OptLet Evaluation

During the clean-up of the solution pool, the framework re-evaluates the success of all OptLets. It first calculates a "bonus" according to each OptLet's success during the last iteration, and then updates the score by combining it with the bonus.

## 4.1 Calculating the Bonus

It is desirable to assign a bonus to an OptLet for improving solutions. For each improvement, the bonus is incremented by a certain value.

As new solutions are always generated based on existing ones, several chains of solutions result during an iteration. These chains also contain information about the involved OptLets, as shown in Fig.2:

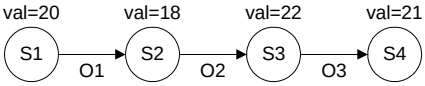


Fig.2: Solution chain

Starting from solution *S1* with a value of 20, the OptLet *O1* produced a solution *S2* with a value of 18. From this solution *S2*, the OptLet *O2* managed to produce a solution *S3* with a value of 22. Finally, OptLet *O3* produced a solution *S4* with a value of 21 from *S3*. Evaluating each OptLet based on its immediate improvements would mean that only *O2* would receive a bonus. However, *O2* might not have been able to produce the solution *S3* without the previous deterioration made by *O1*. This is a typical scenario in which a combination of OptLets can help to escape from a local optimum.

The idea is to calculate the bonus not only based on the improvements of a single OptLet, but to look at solution chains and award the bonus to a sequence of OptLets that managed to find a solution better than the first solution in the chain.

At the beginning of an iteration, the pool contains several solutions which survived the last clean-up. Each of these solutions is taken as a starting point for a new solution chain. At the end of the iteration, the framework looks at all solution chains and identifies the best solution in each chain. The bonus is then divided among all OptLets that were involved in finding the best solution in the chain. In the example shown in Fig.2, these are the OptLets *O1* and *O2*. Both would get half of the bonus awarded for the chain. Note that when the first solution in the chain is already the best one, no OptLet gets a bonus, which makes sense as there was no improvement to the original solution.

Depending on the best solution in the chain, different bonus values are assigned to the chain:

| | |
|---|---|
| $B_{val}$ | for finding an improved valid solution |
| $B_{best}$ | for finding a new best valid solution |
| $B_{inv}$ | for finding an improved invalid solution |
| $B_{minv}$ | for finding a new minimum invalid solution |

$B_{val}$ is not only awarded for improving a valid solution, but also for turning an invalid solution into a valid one. $B_{best}$ and $B_{minv}$ are awarded when the solution was the current best or minimum invalid at the time it was found. All these values are parameters that can be configured by the user. Typically, $B_{best}$ and $B_{minv}$ will be higher than $B_{val}$ and $B_{inv}$ in order to honor OptLets that manage to find a new current best or minimum invalid solution.

The algorithm for calculating the OptLets' bonuses can be described informally as follows:

```
bonus[1:n] = 0   // set all bonuses to 0
for (all solution chains) {
  s = best solution in the chain
  if (s is valid) {
    if (s was  best) { b = B_best }
    else { b = B_val }
  }
  else {   // s is invalid
    if (s was  min. invalid) { b = B_minv }
    else { b = B_inv }
  }
  nr = number of OptLets involved for finding s
  for (all involved OptLets o) {
    bonus[o] = bonus[o] + b/nr
  }
}
```

### 4.2 Updating the Score

After the bonuses for all OptLets have been calculated, their scores need to be updated. In order to honor the success during recent iterations more than that of older ones, the old score is multiplied by a degression factor *d* before adding the bonus of the last iteration. This ensures that the score of an OptLet that was good at the beginning of the optimization decreases continuously if it is no longer successful later. The degression factor is a configurable parameter.

In order to prevent the score from becoming 0, there is a guaranteed minimum score $S_{min}$ for each OptLet. This ensures that every OptLet gets a chance to be called by the framework even if it has not been successful recently.

Furthermore, the framework considers the runtime of OptLets. If an OptLet manages to reach the same improvements as another OptLet in only half of the time, this should be honored appropriately. Therefore, the bonus $B_i$ of each OptLet *i* is divided by its average runtime $R_i$ before it is added to the overall score $S_i$. This leads to the following formula for calculating an OptLet's score, based on the old score at time *t*:

$$S_{i,t+1} = S_{\min} + (S_{i,t} - S_{\min}) \cdot d + \frac{B_i}{R_i} \qquad (2)$$

The average runtime $R_i$ is a relative value, depending on the OptLet's actual runtime $T_i$ and the average runtime $R_{avg}$ of all OptLets:

$$R_i = \max(0.01, \frac{T_i}{R_{avg}}) \qquad (3)$$

For an OptLet that has an average runtime near the total average runtime of all OptLets, $R_i$ is about 1, for faster OptLets less (bonus becomes greater) and for slower OptLets greater (bonus becomes less). In order to prevent the score from exploding for very fast OptLets, $R_i$ is restricted to a minimum value of 0.01.

## 5 Case Studies and Results

We successfully tested this evaluation approach with the Knapsack Problem (KP) and Traveling Salesman Problem (TSP) as well as with two real-world problems. Our primary goal was to show that an effective OptLet evaluation can help finding good solutions faster.

We will demonstrate our results with the well-known Knapsack Problem. The problem instances used for our experiments have been randomly generated using the generator described in [16]. For all instances, the optimum is known and has been computed with NEOS [17].

We performed our experiments on a Pentium 4 2.4 GHz computer with 1 GB RAM running Windows XP.

### 5.1 Parameter Settings

The OptLets framework contains several parameters for controlling the optimization process. During the development of the OptLets framework, we could identify a manually tuned configuration that enables the framework to deliver good results for most problem classes.

For our experiments, we used the following default parameter settings:

| Parameter | Value |
|---|---|
| $B_{val}$ | 1 |
| $B_{best}$ | 20 |
| $B_{inv}$ | 0.5 |
| $B_{minv}$ | 10 |
| $d$ | 0.6 |

Table 1: Parameter settings

To show how the optimizer performs without evaluation, the parameters above have been set to 0.

### 5.2 Evolution of the Solution Value

The following diagram compares the evolution of the best solution value over time with and without evaluation for a problem with 50000 items.

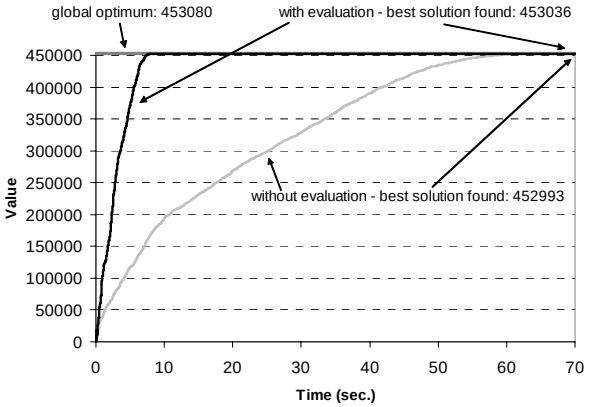


Fig.3: Impact of the OptLet evaluation

The curve without OptLet evaluation has a steep ascent in the first second only. From then on, the solution quality slowly improves until the curve stagnates after about 60 seconds.

With OptLet evaluation, the ascent is much steeper and the solution quality gets close to the known optimum within 8 seconds. The optimizer improves the solution further on until it reaches a value of 453036 after 70 seconds.

### 5.3 Evolution of the OptLet Scores

Fig.4 shows the score ratio of the five best OptLets and the cumulated average score ratio of all other OptLets (as percentage) over runtime in relation to the quality of the solution.

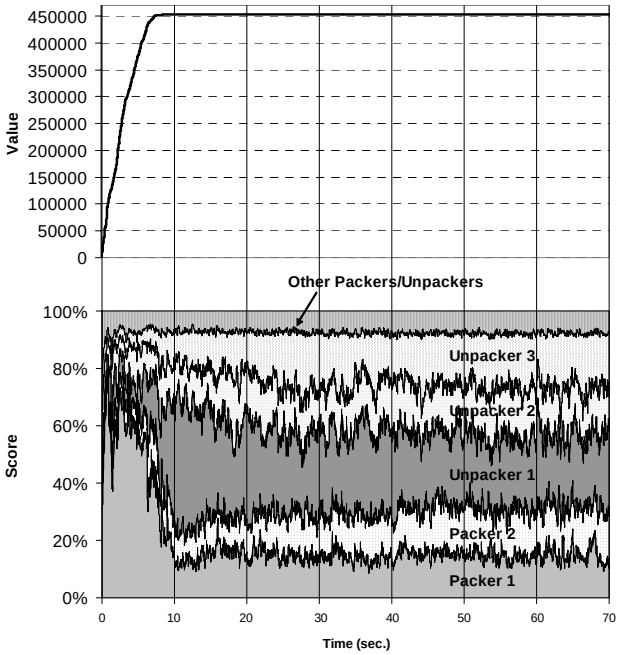


Fig.4: Distribution of OptLet scores

Two packing OptLets (Packer 1 and 2 – with different packing strategies) and three unpacking OptLets (Unpacker 1, 2 and 3) have been most successful within the selected time span. As the knapsack is empty at the beginning of the optimization, the packing OptLets can improve the quality quickly. After about 8 seconds, most knapsacks (solutions) in the pool are almost filled and the packing OptLets often generate invalid solutions. These OptLets now become less successful and the framework gradually reduces their score.

At this time, the optimizer has reached a solution quality already very close to the optimum and the curve is stagnating. The framework still improves the solution quality in “micro steps”. As the OptLets cannot achieve big improvements now, their scores no longer changes as drastically as in the beginning.

As already discussed, the score ratio of each OptLet also represents its selection probability, i.e. the higher the score, the higher the probability for being selected. For example, Packer 1 has a selection probability of about 60% after the first 3 seconds.

For the other problems (TSP, steel mill, robot), the results show a similar picture. Similar experiments with other types of problems showed that OptLet evaluation significantly speeds up the finding of good solutions in all cases.

## 6 Conclusion

In this paper, we presented a new approach for evaluating operators working in population-based optimization environments. This evaluation strategy is part of the OptLets framework that is able to solve different types of optimization problems.

Our approach assumes that the success of operators (OptLets) is an essential aspect of evolutionary optimization. The work of the OptLets is evaluated and they receive a bonus depending on their previous success. This bonus influences the score of an OptLet which reflects its selection probability during the next iteration.

We could show that using an efficient evaluation strategy, good results can be achieved much faster than using a uniformly distributed OptLet selection. The framework is able to take advantage of successful OptLets and lets them work more often than not successful ones.

Another important aspect is that some OptLets are good in the beginning whereas others are able to improve the quality of near-optimal solutions at the end of the optimization process. Continuous re-

evaluation of OptLets automatically adapts the scores to such special abilities over time.

OptLets are typically developed independently by members of a team, who have different ideas which operators could help to produce better solutions. Without further intervention of the developers, the framework automatically prefers OptLets that are doing well in a particular optimization phase. If an OptLets turns out to be ineffective, it receives the minimum score and therefore does not consume much optimization time. As an additional benefit, analysis of the scores can give the developers valuable hints: OptLets which never receive high scores might be candidates for removal, whereas successful OptLets can lead to ideas for similar – probably even better – OptLets.

Currently, the evaluation strategy is implemented as part of the OptLets framework. However, it can be used for any evolutionary optimization technique that uses multiple operators working on a population of solutions.

Hence, our approach seems to be a promising strategy to adaptively guide a search process in evolutionary optimization.

*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.

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

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

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

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

[6] Kennedy, J.; Eberhart, R. C.; Yuhui, S.: *Swarm Intelligence*, Morgan Kaufmann Publishers, 2001.

[7] Lau, H. C.; Wan, W. C.; Lim, M. K.; Halim, S.: A Development Framework for Rapid Metaheuristics Development, *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, September 28 - 30, 2004, Hong Kong, 2004.

[8] 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, pp. 362-367, 2003.

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

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

[11] Arenas, M. G.; Collet, P.; Eiben, A. E.; Jelasity, M.; Merelo, J. J.; Paechter, B.; Preuß, M.; Schoener, M.: A Framework for Distributed Evolutionary Algorithms, *Proceedings of PPSN VII*, Granada, 2002.

[12] Cahon, S.; Melab, N.; Talbi, E.: PardisEO: A Framework for the Resusable Design of Parallel and Distributed Metaheuristics, *Journal of Heuristics*, 10/2004, pp. 357-380.

[13] Keijzer, M.; Merelo, J. J.; Romero, G.; Schoenauer, M.: Evolving Objects: A General Purpose Evolutionary Computation Library, *Proceedings of th 5th Intl. conference on Artificial Evolution (EA'01)*, Le Creusot, France, 2001.

[14] Chang, P.; Dolan, J.; Terk, M.: Asynchronous Team Toolkit User's Guide, Carnegie Mellon University, 1996.

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

[16] Pisinger, D.: Advanced Generator for 0-1 Knapsack Problems, 2005, Online: http://www.diku.dk/~pisinger/codes.html, [last visited: 23.2.2005].

[17] NEOS: NEOS, 2005, Online: http://www-neos.mcs.anl.gov/, [last visited: 29.7.2005].