

EasySyn++: A Tool for Automatic Synthesis of Stochastic Local Search Algorithms

Luca Di Gaspero and Andrea Schaerf

DIEGM, University of Udine, Udine, Italy
{l.digaspero,schaerf}@uniud.it

Abstract. We present a software tool, called EASYSYN++, for the automatic synthesis of the source code for a set of stochastic local search (SLS) algorithms. EASYSYN++ uses C++ as object language and relies on EASYLOCAL++, a C++ framework for the development of SLS algorithms. EASYSYN++ is particularly suitable for the frequent case of having many neighborhood relations that are potentially useful.

1 Introduction

In this work we present a software tool, called EASYSYN++, for the automatic synthesis of the source code for a set of local search algorithms. The synthesized algorithms range from basic local search methods, like hill climbing, simulated annealing, and tabu search, to complex strategies, such as variable neighborhood search and iterated local search.

The tool is implemented in PHP and it uses C++ as object language. EASYSYN++ relies on the C++ local search framework, EASYLOCAL++ [1,2], which has recently been entirely redesigned, and on EASYANALYZER [3] for the experimental analysis of SLS algorithms.

EASYSYN++ input is an XML description of the basic building blocks of local search, namely the search space, the neighborhood relation(s), and the cost function components (hard and soft) for the specific problem. Its output is a set of C++ classes that have to be completed with some problem-specific code developed by the user, and have to be compiled against the problem-independent abstract classes of EASYLOCAL++.

EASYSYN++ is particularly useful for the frequent case of many neighborhood relations (see, e.g., [4]). For these cases, EASYSYN++ manages automatically the composition of basic neighborhoods defining more complex ones. As a consequence, the set of possible combinations of algorithms is relatively large, and the human coding of the whole set of potential algorithms would be time consuming and prone to errors and inconsistencies.

Needless to say, EASYSYN++ cannot replace the human experience in designing the full-fledged algorithm with all its peculiarities. Nevertheless, we believe that this tool can help in a preliminary exploratory phase in which many alternatives are evaluated before focusing on the most promising ones. Thus, EASYLOCAL++, EASYANALYZER, and EASYSYN++ allow the user to obtain with

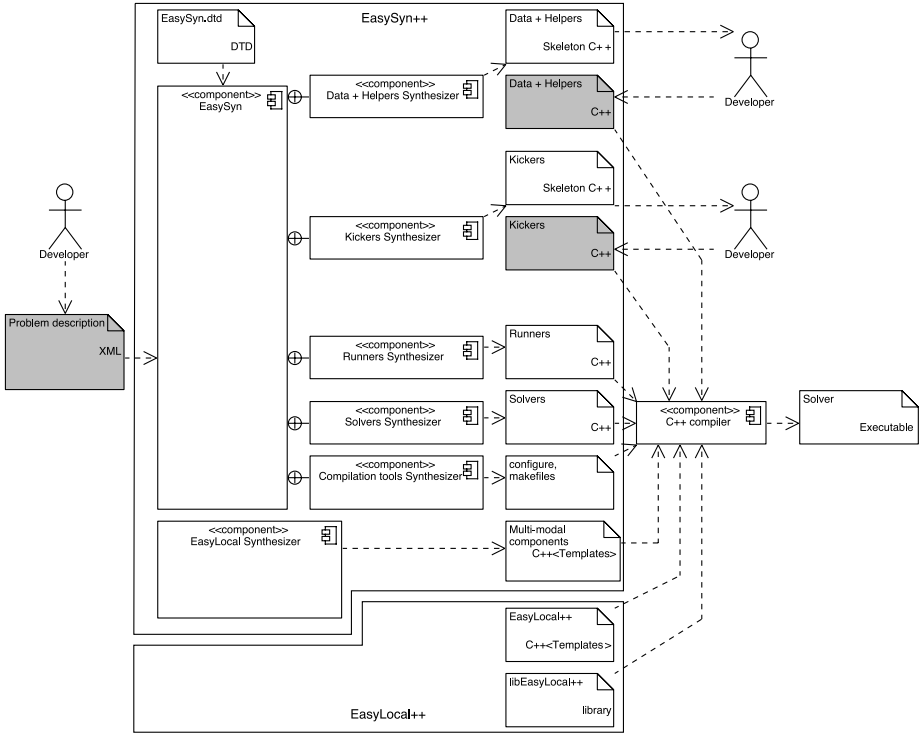


Fig. 1. The architecture of EASYSYN++

limited manual programming a first picture of the behavior of local search for his/her problem, although at the cost of a lot of computational power. However, being CPU time much cheaper than programmer’s time, we believe that the use of such a tool could help in engineering local search algorithms.

The main difference with respect to other tools, such as for example Comet [5] and its predecessors, is that in those software environments the program corresponding to the specified algorithm is assembled internally by the system and therefore its execution is performed “behind the scenes”. Conversely, EASYSYN++ is completely *glass-box* (or *white-box*), and the exact C++ statements are “before user’s eyes”, and can be modified and adapted at user’s preferences.

2 EasySyn++ Architecture

The architecture of EASYSYN++ is shown in Figure 1. The input is a XML description file (which refers to a DTD file for its validation) that is parsed by the core module, which calls the modules responsible for generating the code.

The description file specifies the local search structure and the set of runners, kickers and solvers that we want to synthesize. An example description file for the N-Queens problem is given in Listing 1.1. The <data> element specify the names

of the classes and their role. In this case, for example, there is just one state representation (<state> element) and two neighborhoods (<move> elements).

The <helpers> element specifies the helpers, giving the data types they manage and the optional class names. Class names not supplied are generated by appending the standard name to the problem prefix (for example, NQ.StateManager).

The <kickers> and <runners> elements specify to which neighborhoods these components have to be applied. If more than one neighborhood is specified, EASY-SYN++ generates all possible ones that can be obtained using the union operator. In this context, the tag <use-all-neighborhoods/> declares that EASY-SYN++ has to synthesize one runner (or kicker) for each possible union neighborhood. For example, EASY-SYN++ generates three different HillClimbing runners: one for each move type and one for their union.

The number of generated runners is exponential in the number of neighborhoods: the size of the powerset minus one (the empty set). Therefore, when the number of neighborhoods is large, the use of <use-all-neighborhoods/> must be limited, and instead only subsets of the neighborhoods should be used.

Listing 1.1. The XML description file

```

<?xml version="1.0" standalone="no"?><!DOCTYPE problem SYSTEM "easysyn.dtd">
<problem name="nQueens" prefix="NQ">
<data dir="data/">
  <input classname="BoardSize" />
  <output classname="ChessBoard" />
  <state classname="QueensArray" />
  <move classname="Exchange" />
  <move classname="Swap" />
</data>
<helpers dir="helpers/">
  <state-manager state="QueensArray"/>
  <cost-component classname="Diagonal"/>
  <cost-component classname="Row"/>
  <neighborhood-explorer move="Exchange"/>
  <neighborhood-explorer move="Swap" classname="SwapExplorer"/>
  <output-manager output="ChessBoard"/>
  <prohibition-manager move="Exchange"/>
  <prohibition-manager move="Swap"/>
</helpers>
<kickers>
  <kicker id="sk"> <use-neighborhood ref="Swap"/> </kicker>
  <kicker id="ak"> <use-all-neighborhoods/> </kicker>
</kickers>
<runners>
  <runner type="HillClimbing" id="hc"> <use-all-neighborhoods/> </runner>
  <runner type="TabuSearch" id="ts"> <use-neighborhood ref="Exchange"/> </runner>
</runners>
<solvers>
  <solver type="IteratedLocalSearch"> <use-runner ref="ts"/> <use-kicker ref="ak" type="
    random"/> </solver>
  <solver type="TokenRing" max_length="3"> <use-all-runners/> </solver>
  <solver type="VariableNeighborhoodDescent"> <use-kicker ref="sk" type="best"/> </solver>
</solvers>
</problem>

```

Each module of EASY-SYN++ is responsible for a type of class to generate. The modules are briefly described in the following.

Data and helper synthesizer. This module creates the skeleton for Data and Helper classes. The actual code describing the attributes of states and moves of the application, together with the cost functions and the neighborhood exploration strategy, must be provided by the user.

Kickers Synthesizer. Kickers are synthesized almost completely. The human intervention regards only the notion of synergy. For each ordered pair of neighborhoods, EASYSYN++ defines a method of the kicker that takes a move of each one, and returns a boolean value. The body of the method must be supplied by the user by specifying the conditions on the values of the attributes of the moves to be synergic.

Runners Synthesizer. Runners are synthesized completely. This module has only the task to create the runner objects with the correct template instantiations and the links to the type-compatible helpers.

Solvers Synthesizer. Solvers also are synthesized completely. This module only creates the solver objects with the template instantiations and the links to runners and kickers. For each concrete solver there is a value of a command-line argument to invoke it. Therefore, all the necessary links with the runners are inserted one by one in a code branch guarded by the given value for such a command-line argument.

Compilation tools. This module does not generate C++ code but sets up a basic `configure` script and a set of `Makefiles` for building the system. The target compilation environment is the GNU *Autotools* suite, which comprises Autoconf, Automake and Libtool. This environment can be considered as the de-facto standard for portably building and installing applications across many systems (including various UNIX/Linux distributions, Mac OS X and Cygwin on Windows).

The result of the synthesis is a set of C++ source files and the portable compilation environment so that the generated code can be immediately compiled just out-of-the-box. Yet, the methods that must be supplied by the user are implemented as a single `throw` statement that raises a run-time exception when executed. This way the programmer is prompted to take care of the actual implementation of those methods, whereas if we had left the methods empty the programmer could forget to implement some method thus giving raise to errors whose effects could be unpredictable.

As shown in Figure 1, EASYSYN++ provides also the abstract EASYLOCAL++ classes for *multimodal* runners and kickers based on neighborhood compositions, such as unions and sequences, as defined in [4]. These components are automatically synthesized for an arbitrary number k of neighborhoods, without the need to define new helper classes and to write any code.

EASYSYN++ provides also kickers for an arbitrary composition of neighborhoods (multimodal kickers). They provide methods for performing both random and best sequences of moves from the composite neighborhood.

In order to generate the best moves (restricted or general) the kicker uses a backtracking algorithm that backtracks as soon as two consecutive moves are not feasible.

3 Discussion and Conclusions

First, although important parts of the code must be provided by the user, the advantages of using EASYSYN++ are significant. Indeed, instead of writing the code from scratch, the user has only to insert specific fragments in the right places (indicated by a `throw` statement), without having to worry about object communication and high level control structures.

As a matter of fact, the programming effort needed to come up with a family of 16 solvers for the N-Queens problem requires only 168 user-supplied lines of code. The overall application is composed also of 750 synthesized lines of code, and 6,898 coming from EASYLOCAL++ (including the multimodal runners and kickers). Even though the number of lines is clearly a very rough measure of the human effort, this is very limited and EASYSYN++ can be considered as a software environment for the fast prototyping of SLS algorithms.

Another advantage of EASYSYN++ is that it makes an intensive and principled reuse of the code. For example, the exploration and the evaluation of the union of two or more neighborhoods is performed relying completely on the helpers of the underlying basic neighborhoods, without the need of any additional code. This design allows the user to avoid duplications that would make the maintenance of the code very problematic.

Needless to say, there are also some limits in using EASYSYN++ and EASYLOCAL++. Their architecture prescribes a precise structure for the design of a local search algorithm: the entities of the problem are factorized in groups of related classes in the framework and the user is forced to use them in a controlled way. If, on the one hand, this helps in term of conceptual clarity and it is one of the main sources of software reuse, on the other hand, since the control logic is completely defined at the framework level, this feature poses some restrictions in the design of *ad hoc* SLS algorithms that need to be tightly tailored to some specific feature of the problem at hand.

References

1. Di Gaspero, L., Schaerf, A.: EasyLocal++: An object-oriented framework for flexible design of local search algorithms. *Software—Practice and Experience* 33(8), 733–765 (2003)
2. Di Gaspero, L., Schaerf, A.: Writing local search algorithms using EasyLocal++. In: Voß, S., Woodruff, D.L. (eds.) *Optimization Software Class Libraries. OR/CS series*, pp. 155–176. Kluwer Academic Publishers, Boston (2002)
3. Di Gaspero, L., Roli, A., Schaerf, A.: EasyAnalyzer: an object-oriented framework for the experimental analysis of stochastic local search algorithms. In: Stützle, T., Birattari, M., Hoos, H. (eds.) *Engineering Stochastic Local Search Algorithms (SLS-2007)*. LNCS, vol. 4638, pp. 76–90. Springer, Heidelberg (2007)
4. Di Gaspero, L., Schaerf, A.: Neighborhood portfolio approach for local search applied to timetabling problems. *Journal of Mathematical Modeling and Algorithms* 5(1), 65–89 (2006)
5. Van Hentenryck, P., Michel, L.: *Constraint-Based Local Search*. MIT Press, Cambridge (MA), USA (2005)