EasyAnalyzer: An Object-Oriented Framework for the Experimental Analysis of Stochastic Local Search Algorithms

Luca Di Gaspero[1], Andrea Roli[2], and Andrea Schaerf[1]

[1] DIEGM, University of Udine, Udine, Italy
[2] DEIS, University of Bologna, Cesena, Italy
{l.digaspero,schaerf}@uniud.it, andrea.roli@unibo.it

Abstract. One of the aspects of applying software engineering to Stochastic Local Search (SLS) is the principled analysis of the features of the problem instances and the behavior of SLS algorithms, which —because of their stochastic nature— might need sophisticated statistical tools.

In this paper we describe EasyAnalyzer, an object-oriented framework for the experimental analysis of SLS algorithms, developed in the C++ language. EasyAnalyzer integrates with EasyLocal++, a framework for the development of SLS algorithms, in order to provide a unified development and analysis environment. Moreover, the tool has been designed so that it can be easily interfaced also with SLS solvers developed using other languages/tools and/or with command-line executables.

We show an example of the use of EasyAnalyzer applied to the analysis of SLS algorithms for the k-GraphColoring problem.

1 Introduction

In recent years, much research effort has focused on the proposals of environments specifically designed to help the formulation and implementation of Stochastic Local Search (SLS) algorithms by means of specification languages and/or software tools, such as Localizer and its evolutions [3,1,2], HotFrame [4], ParadisEO [5], iOpt [6], EasyLocal++ [7,8], and others.

Unfortunately, as pointed out by Hoos and Stützle [9] in [9, Epilogue, pp. 533–534], the same amount of effort has not been oriented in the development of software tools for the experimental analyses of the algorithms.

To this regard, [10] proposes a suite of tools for visualizing the behavior of SLS algorithms, which is particularly tailored for MDF (Metaheuristics Development Framework) [11]. However, to the best of our knowledge, we can claim that at present there is no widely-accepted comprehensive environment.

In this paper we try to overcome this lack by proposing an object-oriented framework, called EasyAnalyzer, for the analysis of SLS algorithms. EasyAnalyzer is a software tool that belongs to the family of Object-Oriented (O-O) frameworks. A framework is a special kind of software library, which consists of a hierarchy of abstract classes and is characterized by the *inverse control*

T. Stützle, M. Birattari, and H.H. Hoos (Eds.): SLS 2007, LNCS 4638, pp. 76–90, 2007.

mechanism for the communication with the user code (also known as the *Hollywood Principle*: "Don't call us, we'll call you"). That is, the functions of the framework call the user-defined ones and not the other way round as it usually happens with software libraries. The framework thus provides the full control logic and, in order to use it, the user is required to supply the problem specific details by means of some standardized interfaces.

Our work is founded on *Design Patterns* [12], which are abstract structures of classes, commonly present in O-O applications and frameworks, that have been precisely identified and classified. The use of patterns allows us to address many design and implementation issues in a more principled way.

EasyAnalyzer provides a family of off-the-shelf analysis methods to be coupled to local search solvers developed using one of the tools mentioned above or written from scratch. For example, it performs various kinds of search space analysis in order to understand, study, and tune the behavior of SLS algorithms. The properties of the search space are a crucial factor of SLS algorithm performance [13,9]. Such characteristics are usually studied by implementing *ad hoc* programs, tailored both to the specific algorithm and to the problem at hand. EasyAnalyzer makes it possible to abstract from algorithm implementation and problem details and to design general search space analyzers.

EasyAnalyzer is specifically designed to blend in a natural way with EasyLocal++, the local search framework developed by two of these authors [8,7], which has recently been entirely redesigned to allow for more complex search strategies. Nevertheless, it is capable of interacting with other software environments and with stand-alone applications.

This is an ongoing work, and some modules still have to be implemented. However, the general architecture, the core modules, and the interface with EasyLocal++ and with command-line executables are completed and stable.

The paper is organized as follows. In Section 2 we show the architecture of EasyAnalyzer and its main modules. In Section 3 we go in details in the implementation of the core modules. In Section 4 we show some examples of use based on the classic k-GraphColoring problem. In Section 5 we draw some conclusions and discuss future work.

2 The Architecture of EasyAnalyzer

The conceptual architecture of EasyAnalyzer is presented in Figure 1 and it is split in three main abstraction layers. Each layer of the hierarchy relies on the services supplied by lower levels and provides a set of more abstract operations.

Analysis system: it comprises the *core classes* of EasyAnalyzer. It is the most abstract level and contains the control logic of the different types of analysis provided in the system. The code for the analyses is completely abstract from the problem at hand and also from the actual implementation of the solver. The classes of this layer delegate implementation- and/or problem-related tasks to the set of lower level classes, which comply with a predefined service interface (described in the following).

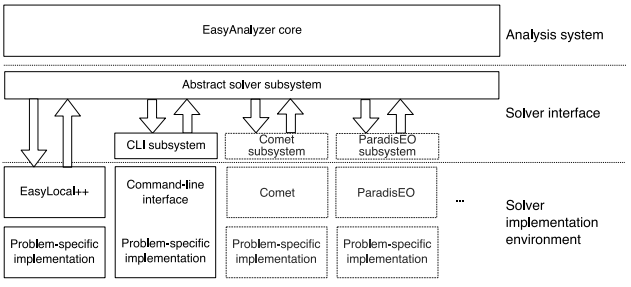

Fig. 1. EASYANALYZER layered architecture

Solver interfaces: this layer can be split into two components: the top one is the interface that represents an abstract solver subsystem, which simply prescribes the set of services that should be provided by a concrete solver in order to be used in the analyses. The coupling of the analysis system with the implementation is dealt with by this component.

The lower component is the concrete implementation of the interface for a set of SLS software development environments. Notice that in the case of EASYLOCAL++, this component is not present since EASYANALYZER directly integrates within the development framework classes. The reason is that in the design of the solver interface we reuse many choices already made for EASYLOCAL++ thus allowing immediate integration.

For other software environments, instead, the solver subsystem component must be explicitly provided. Depending on the capabilities of the software environment, these interfaces can be implemented in a problem-independent manner (so that they can be directly reused across all applications) or it might require to be customized for the specific problem. Although in the second case the user could be required to write some additional code, our design limits this effort since our interfaces requires just a minimal set of functionalities.

Solver environment: it consists of the (possibly generic) SLS software development environment plus the problem-specific implementation. In some cases these two components coincide, as for solvers that do not make use of any software environment. In this case the interaction with the solver can make use of a simple command-line interface.

At present, we have implemented the direct integration with EASYLOCAL++ and to the command-line interface[1] by means of a set of generic classes (i.e., C++ classes that make use of templates that should be instantiated with the concrete command-line options). We plan to implement also the interfaces to other freely available software environments like, e.g., ParadisEO [5] and Comet [2].

[1] In Figure 1 the implemented components are denoted by solid lines while dotted lines denote components only designed.

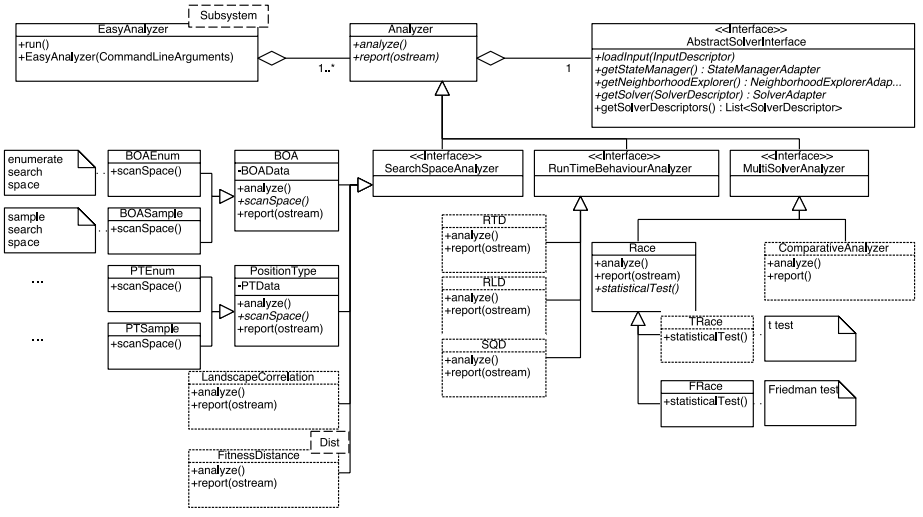

Fig. 2. UML class diagram of the analysis system

In the following subsections we present more in detail the problem-independent layers of the EasyAnalyzer architecture and we give some example of code.

2.1 The Analysis System

The main classes of the analysis system are shown in Figure 2 using the UML 2.0 notation [14]. As in Figure 1 we report in solid lines the fully implemented components (dotted lines for the forthcoming ones).

Let us start our presentation with the `EasyAnalyzer` class. This class relies on the *Factory method* pattern to set up the analysis system on the basis of a given solver interface. Notice that the interface is specified as a template parameter, so that we are able to write the generic code for instantiating the analysis system regardless which of the concrete implementations is provided. Furthermore, the `EasyAnalyzer` class provides a standardized command-line interface for the interaction with the analysis system. This task is accomplished by managing a command-line interpreter object that is directly configured by the analysis techniques. That is, each analysis technique "posts" the syntax of the command-line arguments needed by the interpreter object that is in charge of parsing the command line and dispatching the actual parameters to the right component.

The main component of the analysis system is the `Analyzer` class, which relies on the *Strategy* pattern. This component represents the interface of an analysis technique, whose actual "strategy" will be implemented in the `analyze()` method defined in the concrete subclasses. The `report(ostream)` method is used to provide on an output stream a human- and/or machine-readable report of the analysis, depending on the parameters issued on the command line.

The `Analyzer` class is then specialized on the basis of the SLS features that are subject of the analysis into the following three families:

`SearchSpaceAnalyzer:` these analyzers deal with features that are related to the search space. Several crucial properties of the search space can be analyzed with these modules, such as landscape characteristics and states reachability.

`RunTimeBehaviorAnalyzer:` their aim is to analyze the run-time behavior of the solvers. Analyses belonging to this family are, e.g., *run-time distribution* (`RTD`), *run-length distribution* (`RLD`) and *solution quality distribution* (`SQD`).

`MultiSolverAnalyzer:` they handle and evaluate groups of solvers. For example the `Race` analyzer tries to find-out the statistically best configuration of a solver among a set of candidate configurations by applying a racing procedure [15].

The interface with the services provided by the analysis system is established with the `AbstractSolverInterface` abstract class, which relies on the *Façade* pattern whose aim is to provides a simple interface to a complex subsystem. This class and the underlying classes and objects responsibilities are going to be detailed in the following subsection.

2.2 The Solver Interface

The architecture of the solver interface is shown in the top part of Figure 3. The derived classes on the bottom are the implementation of this interface in the EasyLocal++ framework.

The `SolverInterface` class acts as a unified entry point (the *Façade*) and as the coordinator of a set of underlying classes (*Abstract Factory* and *Factory method* patterns). Indeed, according to the EasyLocal++ design, we identify a set of software components that take care of different responsibilities in a SLS algorithm and we define a set of *adapter* classes for them. These adapters have a straight implementation in EasyLocal++ (Figure 3, bottom part), and are those components that instead must be implemented for interfacing with different software environments. The components we consider are the following:

`StateManagerAdapter:` it is responsible for all operations on the states of the search space that are independent of the definition of the neighborhood. In particular, it provides methods to enumerate and to sample the search space, and it allows us to evaluate the cost function value on a given state. The component relies on `StateDescriptor`s for the exchange of information with the analysis system (in order to avoid the overhead of sending a complex space representation).

`NeighborhoodExplorerAdapter:` it handles all the features concerning the exploration of the neighborhood. It allows to enumerate and to sample the neighbors of a given state, and to evaluate the cost function.

`SolverAdapter:` it encapsulates a single SLS algorithm or a complex solution strategy that involves more than one single SLS technique. Its methods allow us to perform a full solution run (either starting from a random initial state

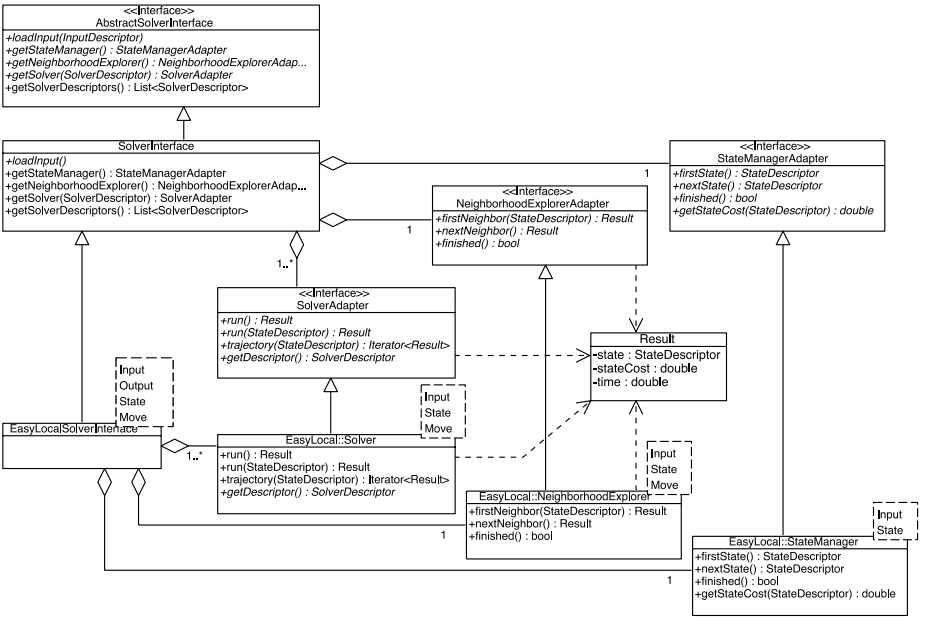

Fig. 3. UML class diagram of the solver interface

or from a state given as input), possibly storing all the trajectory from the initial state to the final one. This component returns also information on the running time and on the state costs.

2.3 How to Use EasyAnalyzer

In order to use EasyAnalyzer it is only needed to instantiate the `Solver` template of the `EasyAnalyzer` class with the proper implementation of the `AbstractSolverInterface`. As for the EasyLocal++ solver, this interface is already provided with the framework, whilst for the command-line interaction the functionalities must be implemented by the user in the stand-alone executable.

The various analyses can be executed by issuing command line options to the EasyAnalyzer executable. For example, `-ptenum` requires a position type analysis to be performed (with a complete enumeration of the search space). Additional parameters, depending on the analysis at hand, can be required and they have to be specified on the command line as well. The different types of analysis and an explanation of the options available can be obtained by issuing a `-help` command.

3 Implementation of EasyAnalyzer

In this section we describe two representative examples of the analyzers currently implemented, with emphasis on the design process that relies on the abstractions provided by the Solver interfaces.

3.1 SearchSpaceAnalyzer

In this section we illustrate the design and implementation of an analyzer for *Basins of attraction* (BOA), useful for studying the reachability of solutions. Given a deterministic algorithm, the basin of attraction $\mathcal{B}(\overline{s})$ of a search space state $\overline{s}$ (usually a minimum), is defined as the set of states that, taken as initial states, give origin to trajectories that end at point $\overline{s}$. The quantity $rBOA(\overline{s})$, defined as the ratio between the cardinality of $\mathcal{B}(\overline{s})$ and the search space size (assumed finite), is an estimation of the reachability of state $\overline{s}$. If the initial solution is chosen at random, the probability of finding a global optimum s^* is exactly equal to $rBOA(s^*)$. Therefore, the higher is this ratio, the higher is the probability of success of the algorithm. The estimation of basins of attraction characteristics can help in the *a posteriori* analysis of local search performance, to provide explanations for the observed behavior. Moreover, it can also be useful for the *a priori* study of the most suitable models of a problem, for instance for comparing advantages and disadvantages of models that incorporate symmetry-breaking or implied constraints [16]. In section 4.2, we will discuss an example of a typical application of this kind of *a posteriori* analysis.

The development of a specific analyzer starts from the implementation of the interface `SearchSpaceAnalyzer` that declares the basic methods `analyze()`, for the actual analysis to be performed, and `report()`, defining the output of the analysis. The main goal of a BOA analyzer is to find the size of all, or a sample of, the local and global minima basins of attraction, corresponding to the execution of a given (deterministic) algorithm $\mathcal{A}$. Therefore, a BOA analyzer must be fed with problem instance and search algorithm and its task is to scan the search space for finding attractors and their basins. The procedure of search space scanning can be implemented in several ways, and it could primarily be either an exhaustive enumeration or a sampling. Attractors and their basins can be then computed by running algorithm $\mathcal{A}$ from every possible initial state s, returned by the scan method, till the corresponding attractor.[2] The main parts of the `analyze()` method for the `BOA` class are as detailed in Listing 1.1.

Listing 1.1. The `analyze()` method for the `BOA` class

```
void BOA::analyze()
{ BOAData data;
  initializeAnalysis(); // loads instance and solver
  StateDescriptor state = scanSpace();
  while (state.isValid()) // while there are feasible states
  { const Result& result = solver.run(state);
    updateBOAInfo(result.getStateDescriptor());
    state = scanSpace();
  }
}
```

[2] There are also other ways for performing this task; for instance, the *Reverse hill-climbing* technique [17]. Moreover, in this discussion, we only consider the case of deterministic algorithms.

The BOA analyzer is designed through the *Template Method* pattern, that enables the designer to define a class that delegates the implementation of some methods to the subclasses. In this case, the implementation of the method `scanSpace()` is left to the subclasses, so as to make it possible to implement a variety of different search space scanning procedures, such as enumeration (`BOAEnum`, Listing 1.2) and uniform sampling (`BOASample`, Listing 1.3). These methods rely on `StateManagerAdapter` for enumeration and random sampling of the search space, respectively.

Listing 1.2. `BOAEnum::scanSpace()`

```
StateDescriptor
  BOAEnum::scanSpace()
{
  if(!stateManager.finished())
    return
       stateManager.nextState();
 else return NON_VALID_STATE;
}
```

Listing 1.3. `BOASample::scanSpace()`

```
StateDescriptor
  BOASample::scanSpace()
{ if (numberOfSamples <
     maxNumberOfSamples)
   return
     stateManager.randomState();
  else return NON_VALID_STATE;
}
```

We remark that the implementation of these BOA analyzers is very simple and compact, as it is totally independent from the problem specific part of the software, thanks to the intermediate software level of Solver interfaces. With few lines of code it is possible to implement other BOA analyzers, for instance by using samplings based on non-uniform distributions, in order to bias the sampling in areas containing global minima.

In an analogous way, it is possible to implement analyzers which can scan the search space and classify each state as (strict) local minimum/maximum, plateau, slope or ledge as a function of the cost of its neighbors. According to [9], we call this kind of classification *position type* analysis. The class `PositionType` delegates the subclasses `PTEnum` and `PTSample` for the implementation of the method `scanSpace()`, that relies on the class `NeighborhoodExploreAdapter` for enumerating the neighborhoods. The method `scanSpace()` can enumerate or sample the search space. The current implementation includes enumeration and uniform sampling, while the sampling through different distributions or along trajectories is part of ongoing work.

3.2 `MultiSolverAnalyzer`

In many cases, the people working on SLS algorithms face the problem of evaluating the behavior of a family of solvers (usually on a set of a benchmark instances) rather than analyzing a single SLS algorithm. For example one could be interested in comparing a set of SLS solvers to determine whether one or more of them perform better than the others. Another common case is to consider different settings for the same solver as a mean for tuning the parameters

of the solver. In both cases statistical procedures are needed to assess the choice of the "winning" solver in a sound way.

To deal with this situation, we decided to design also a set of analyzers that manage a set of SLS solver and whose aim is to perform comparative analysis. As in the previous example, we rely on the abstraction levels of EasyAnalyzer to design general multi-solver analyzers.

We have developed the set of classes that implement the Race approach by Birattari et al. [15]. This procedure aims at selecting the parameters of a SLS algorithm by testing each candidate configuration on a set of trials. The configurations that perform poorly are discarded and not tested anymore as soon as sufficient statistical evidence against them is collected.

This way, only the statistically proven good configurations continue the race, and the overall number of tests needed to find the best configuration (or the equally good configurations) is limited. Each trial is performed on the same randomly chosen problem instance for all the remaining configurations and a statistical test is used to assess which of them are discarded.

In order to perform the analysis, the user must specify a set of solvers that are going to be compared in the Race and a set of instances on which the solvers will be run.

We present here the method `analyze()` of the class `Race` (Listing 1.4). The method works in a loop that evaluates the behavior of the configurations on an instance and collects statistical evidence about them. We would like to remark that our implementation follows the lines of the R package [18].

Listing 1.4. The `analyze()` method for the class `Race`

```
void Race::analyze()
{ initializeAnalysis(); // loads instances, solvers and sets up the set
    of aliveSolvers
  replicate = 0;
  do
  { performReplicate(instances[replicate % instances.size()], replicate);
    if (replicate >= min_replicates) // the test is performed only after
        a minimum number of replicates
    { TestResult res = statisticalTest(seq(0, replicate), aliveSolvers,
        conf_level);
      updateAliveSolvers(res.survived);
      statistics[replicate] = res.statistic;
      p_values[replicate] = res.p_value;
    }
    replicate++;
  }
  while (aliveSolvers.size() > 1 && replicate < max_replicates);
}
```

The evaluation of the candidate configurations is performed by calling the method `performReplicate` (whose code is reported in Listing 1.5). This method

relies on the solver interfaces to load the current input instance and to invoke the different solvers configurations (only for the solvers that still survive the race).

Listing 1.5. The `performReplicate()` method of the class `Race`

```
void Race::performReplicate(const std::string& instance, unsigned int i)
{ sub.loadInput(instance);
  for (unsigned int j = 0; j < solvers.size(); j++)
    if (aliveSolvers.find(j) != aliveSolvers.end())
    { Result r = solvers[j]->run();
      outcomes[i][j] = r.getCostValue();
    }
}
```

The class `Race` makes use of the *Template Method* pattern: the selection algorithm relies on the implementation of the abstract `statisticalTest()` method, which is implemented in two different sub-classes for the Student's t-test (`TRace`) and the Friedman's test (`FRace`).

Notice that the presented method makes use of the solvers just as black-boxes that from an initial state lead to a final solution. Indeed, the only information exploited in the analysis is the final solution cost and the running time. More sophisticated analyses can also exploit the trajectory from the initial state to the solution. For example this information can be used to compare the quality of SLS solvers throughout the evolution of the search (as suggested by Taillard [19]). This will be subject of future work.

4 A Case Study: The *k*-GraphColoring Problem

We show an example of the use of EasyAnalyzer by providing some analyses on a family of solvers for the k-GraphColoring problem. Our aim is not to say the ultimate word on the problem, but rather to exemplify the use of the analyzers presented so far.

4.1 *k*-GraphColoring Problem Statement and Local Search Encoding

Here we briefly recall the statement of the k-GraphColoring problem, which is the decision variant of the well-known min-GraphColoring problem [20, Prob. GT4, page 191].

Given an undirected graph $G = (V, E)$ and a set of k integer colors, the problem is to assign to each node $v \in V$ a color value $c(v)$ such that adjacent nodes are assigned different colors.

As the search space of our SLS algorithms we consider the set of all possible colorings of the graph, including the infeasible ones; the number of conflicting nodes is the cost function value in a given state. The neighborhood relation

Table 1. Position types in 3-colorable graphs

Position type	Edge Density						
	0.010	0.016	0.020	0.026	0.030	0.036	0.040
Strict local min	$< 10^{-4}$	0%	0%	0%	0%	0%	0%
Local min	0%	0%	0%	0%	0%	0%	0.46%
Interior plateau	0%	0%	0%	0%	0%	0%	92.74%
Ledge	84.42%	99.42%	99.80%	100%	100%	100%	0.10%
Slope	0.77%	0.02%	$< 10^{-4}$	0%	0%	0%	0%
Local max	14.77%	0.56%	0.20%	$< 10^{-4}$	$< 10^{-4}$	0%	6.70%
Strict local max	0.04%	0%	0%	0%	0%	0%	0%

is defined by the color change of one conflicting node (as in [21]) and for the tabu search prohibition mechanism, we consider a move inverse of another one if both moves insist on the same node and the first move tries to restore the color changed by the second one.

4.2 Search Space Analysis

To illustrate the use of EasyAnalyzer for studying properties of the search space, we consider a simple analysis on 3-colorable graphs. Instances were generated with Culberson's graph generator [22] with equi-partition and independent random edge assignment options and with varying edge density, so as to span the spectrum from lowly to highly constrained instances. All instances are guaranteed to be 3-colorable and have 100 nodes.

One of the main search space features of interest is the number of local minima and, more generally, the type of search space positions. Table 1 reports a summary of the position type analysis out of 10^6 random samples.

As discussed in [9], for random landscapes we would expect a position type distribution characterized by a majority of least constrained positions, such as ledges, which are states with neighbors with higher, lower and equal cost. From the results in Table 1, we observe that ledge is the predominant type. The most constrained instance (density = 0.04) shows instead a very different landscape structure, as it is dominated by plateaus. This difference with respect to the other instances is particularly apparent also when local search is used to solve these instances. Figure 4 shows the box-plots corresponding to the execution of 100 independent short runs of a simple hill-climbing (draw a random move, accept it if improving or sideways). The algorithm stops after 10 iterations without improvements. The performance of local search on the most constrained instance is significantly worse than that on the other instances. This result can be explained by the presence of many plateaus that strongly impede local search.

The performance on instances with edge density equal to 0.036 is also statistically different than that on lower densities and this cannot be explained by the results of position types analysis. The analysis of basins of attraction of local and global minima can shed some light on this point, as it enables us to estimate the probability of reaching a solution to the problem. Basins have been estimated by uniformly sampling the search space with 10^6 samples and by applying a deterministic steepest descent local search. The first outcome of this analysis

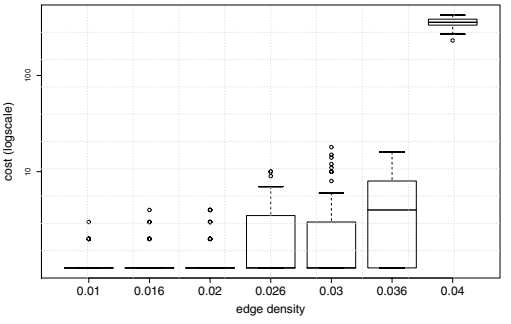

Fig. 4. Box-plots of the performance of randomized non-ascent local search

Table 2. Summary of statistics of relevant characteristics of attractors and their basins

	Edge Density						
	0.010	0.016	0.020	0.026	0.030	0.036	0.040
rGBOA	1.0	0.9561	0.9084	0.5782	0.4831	0.2529	$< 10^{-5}$
Number of cost levels	1	3	4	7	11	14	41
Cost with highest frequency	0	0	0	0	0	2	100
Max cost	0	4	6	12	12	14	100
Median cost	0	0	0	0	2	2	100

is that almost every initial state leads to a different attractor, i.e., almost all basins have size 1.[3] This result can be explained by observing that the problem model induces a search space with many symmetric states, as colors can be permuted [16]. The most relevant statistical characteristics of those attractors are summarized in Table 2. The table reports the fraction of states from which a solution can be reached (*rGBOA*), the number of different levels of cost of the attractors, the most frequent, max and median cost. We can observe that the fraction of attractors corresponding to a solution decreases while edge density increases. The *rGBOA* for instance at density = 0.030 is about 50% of the search space and it halves at density 0.036, till vanishing for the most constrained instance. This difference provides an explanation for the degrading performance of the local search used, that heavily relies on cost decreasing moves. Furthermore, this analysis brings also evidence for the positive correlation between edge density and search space ruggedness.

4.3 Multi-solver Analysis: Tabu Search Configuration Through *F*-Race

As an example of using the multi-solver analysis classes we provide a description of the tuning of tabu search parameters by means of the *F*-Race analysis class.

[3] In this analysis the states belonging to the trajectory from the initial state to the attractor are not counted.

Table 3. F-Race results for the configuration of the tabu-list length. Solver values are running-times to a feasible coloring.

Replicate	Solvers						Statistic value	p value
	TS_{5-10}	TS_{10-15}	TS_{15-20}	TS_{20-25}	TS_{25-30}	TS_{30-35}		
01	2.21	21.04	7.62	4.9	4.7	9.68	—	—
02	5.52	3.49	4.3	3.3	8.55	4.64	—	—
03	41.47	12.34	2.9	6.97	12.09	4.43	—	—
04	6.94	2.48	2.29	1.72	4.59	9.74	—	—
05	3.04	2.44	6.13	3.45	7.02	8.71	—	—
06	3.89	7.09	3.02	3.41	2.56	3.29	3.80952	0.577153
07	3.95	4.09	3.1	3.01	2.59	8.99	5.28571	0.382015
				...				
27	28.65	6.73	26.77	7.01	5.9	31.40	11.381	0.044328*
28	7.12	4.3	4.71	3.77	3.5	—	5	0.287297
				...				
37	15.32	25.88	2.99	6.62	8.48	—	9.70811	0.045642*
				...				
40	—	8.35	3.2	15.16	5.3	—	3.99	0.262546

Solvers survived after 40 replicates: TS_{10-15}, TS_{15-20}, TS_{20-25}, TS_{25-30}

Our tabu search implementation employs a dynamic short-term tabu list (called Robust Tabu Search in [9]), so that a move is kept in the tabu list for a random number of iterations in the range $[k_{min}..k_{max}]$. In this example we want to find out the best values of these two parameters among the following set of options: $(k_{min}, k_{max}) \in \{(5,10),(10,15),(15,20),(20,25),(25,30),(30,35)\}$.

As a set of benchmark instances we generate a set of 40 3-colorable equi-partitioned graphs with independent random edge assignment; the graphs have 200 nodes and edge density 0.04. The performance measure employed in this study is the running time needed to reach a feasible coloring.

The results of the F-Race are reported in Table 3 and are those obtained as the output of the `report()` method of the class `Race`. We limit the maximum number of replicates to 40 and the confidence level for the Friedman test is 0.95; the first test is performed after 5 replicates. The table summarizes the whole Race procedure, by providing the raw running time values, the value of the statistic employed in the test (the F statistic in the present case) and the p value of the hypothesis testing. For the replicates that lead to discarding one of the candidates, the p value is marked with an asterisk, indicating that the test was significant at the confidence level 0.95. The last line reports the final outcome of the Race and shows the number of replicates performed and the list of solvers that survived the Race.

The results confirm the robustness of employing a dynamic tabu-list. Indeed, only the two most extreme configurations were discarded by the analysis, namely TS_{5-10} and TS_{30-35}.

Of course, these results prompted for additional analysis (for example on different graph sizes), but as in the previous case, this is out of the scope of this presentation since our aim was just to exemplify how to perform an analysis and report its results with a limited effort (see, e.g., [23]).

5 Conclusions

We have presented EasyAnalyzer, a software tool for the principled experimental analysis of SLS algorithms. The tool is very general and can be used across a variety of problems with a very limited human effort. In its final version, it will be able to interface natively with a number of development environment, whereas in its current form it is interfaced with EasyLocal++, but also with any solver at the price of configuring a command-line interface.

The design of EasyAnalyzer deliberately separates the problem-/implementation-specific aspects from the analysis procedures. This allows, for example, to (re)use directly new analyses classes —developed at the framework level— by applying them to all the solvers for which a Solver interface already exists.

We believe that our attempt to define such an environment can be regarded as an initial step toward engineering the experimental analysis of SLS algorithms.

For the future, we will implement the interface modules for the most common environment. We also plan to test EasyAnalyzer on more complex problems, with the aim of obtaining also significant results for the research on SLS-based solvers. Finally, we plan to implement other analyses, such as those proposed in [24,19].

References

1. Michel, L., Van Hentenryck, P.: Localizer. Constraints 5(1–2), 43–84 (2000)
2. Van Hentenryck, P., Michel, L. (eds.): Constraint-Based Local Search. MIT Press, Cambridge (MA), USA (2005)
3. Van Hentenryck, P., Michel, L.: Control abstractions for local search. Constraints 10(2), 137–157 (2005)
4. Fink, A., Voß, S.: HotFrame: A heuristic optimization framework. In [25] pp. 81–154
5. Cahon, S., Melab, N., Talbi, E.G.: ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. Journal of Heuristics 10(3), 357–380 (2004)
6. Voudouris, C., Dorne, R., Lesaint, D., Liret, A.: iOpt: A software toolkit for heuristic search methods. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 716–719. Springer, Heidelberg (2001)
7. Di Gaspero, L., Schaerf, A.: Writing local search algorithms using EasyLocal++. In[25]
8. 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)
9. Hoos, H., Stützle, T.: Stochastic Local Search Foundations and Applications. Morgan Kaufmann, San Francisco (CA), USA (2005)
10. Halim, S., Yap, R., Lau, H.: Viz: a visual analysis suite for explaining local search behavior. In: Proceedings of the 19th annual ACM symposium on User interface software and technology (UIST '06), pp. 57–66. ACM Press, New York (2006)
11. Lau, H., Wan, W., Lim, M., Halim, S.: A development framework for rapid meta-heuristics hybridization. In: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004), pp. 362–367 (2004)

12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: elements of reusable object-oriented software. Addison-Wesley Publishing, Reading (MA), USA (1995)
13. Fonlupt, C., Robilliard, D., Preux, P., Talbi, E.G.: Fitness landscapes and performance of metaheuristic. In: Voß, S., Martello, S., Osman, I., Roucairol, C. (eds.) Metaheuristics – Advances and Trends in Local Search Paradigms for Optimization, pp. 255–266. Kluwer Academic Publishers, Dordrecht (1999)
14. Pilone, D., Pitman, N.: UML 2.0 in a Nutshell. O'Reilly Media, Inc. Sebastopol (CA), USA (2005)
15. Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K.: A racing algorithm for configuring metaheuristics. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), New York, USA (9-13 July 2002), pp. 11–18. Morgan Kaufmann Publishers, San Francisco (2002)
16. Prestwich, S., Roli, A.: Symmetry breaking and local search spaces. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, Springer, Heidelberg (2005)
17. Jones, T., Rawlins, G.: Reverse hillclimbing, genetic algorithms and the busy beaver problem. In: Genetic Algorithms: Proceedings of the Fifth International Conference (ICGA 1993), San Mateo (CA), USA, pp. 70–75. Morgan Kaufmann Publishers, San Francisco (1993)
18. Birattari, M.: The race package for R. racing methods for the selection of the best. Technical Report TR/IRIDIA/2003-37, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium (2003)
19. Taillard, E.: Few guidelines for analyzing methods. In: Proceedings of the 6th Metaheuristics International Conference (MIC'05), Vienna, Austria (August 2005)
20. Garey, M.R., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York (1979)
21. Hertz, A., de Werra, D.: Using tabu search techniques for graph coloring. Computing 39(4), 345–351 (1987)
22. Culberson, J.: Graph coloring page. URL (2004) Viewed: March 2007, Updated: March 2004, `http://www.cs.ualberta.ca/~joe/Coloring/`
23. Di Gaspero, L., Chiarandini, M., Schaerf, A.: A study on the short-term prohibition mechanisms in tabu search. In: Proc. of the 17th European Conf. on Artificial Intelligence (ECAI-2006) Riva del Garda, Italy pp. 83–87 (2006)
24. Chiarandini, M., Basso, D., Stützle, T.: Statistical methods for the comparison of stochastic optimizers. In: Proceedings of the 6th Metaheuristics International Conference (MIC'05), Vienna, Austria, pp. 189–195 (2005)
25. Voß, S., Woodruff, D. (eds.): Optimization Software Class Libraries. OR/CS. Kluwer Academic Publishers, Dordrecht, the Netherlands (2002)