BCAT: A framework for analyzing the complexity of algorithms

Zoltán Ádám Mann[*] and Tamás Szép[*]
[*] Budapest University of Technology and Economics,
Department of Computer Science and Information Theory,
Magyar tudósok körútja 2, H-1117 Budapest, Hungary
zoltan.mann@gmail.com, szep.tamas@mail.datanet.hu

Abstract—**This paper presents BCAT (Budapest Complexity Analysis Toolkit), a software package to facilitate research on algorithms and computational complexity. BCAT supports the implementation of computational problems, algorithms to solve the problems, and analyzers to analyze the problems. The paper contains details on the software's architecture and the related main design decisions, and reports on the first experiences with using the system.**

I. Introduction

This paper deals with combinatorial problems, i.e. computational problems on discrete structures such as graphs, matrices, Boolean formulas, and automata. Algorithms for combinatorial problems are widely used in informatics, for instance in electronic design automation, logistics planning, artificial intelligence, compilers, robotics etc [1][7][8].

Optimizing an algorithm for practical use always incurs a significant amount of experimentation, including the algorithm's implementation and testing on both synthetic and real-world problem instances. The first motivation of the presented work is to support this process by providing a library of frequently used building blocks for programming and experimenting with algorithms. This way, the empirical analysis of the algorithm and the evaluation of the findings become faster and more organized, thus reducing the time it takes to convert an algorithmic idea to a reliably applicable practical tool.

The other, more subtle but equally important objective of the presented work stems from the need to understand problem complexity. It is well known that some problems are much harder than some others. With the classic notions of complexity theory, most practically relevant problems are either solvable in polynomial time, or they are NP-hard, with no real hope for an efficient algorithm. On the other hand, classic complexity theory is concerned with worst-case complexity, and does not give much insight into the complexity of actual problem instances. From a practitioner's point of view, this is a major deficiency of the classic complexity theory. More recently, empirical research has uncovered very interesting patterns in the complexity of specific problem instances. In particular, even in the case of NP-hard problems, it is possible that many problem instances are efficiently solvable. A consequence of this phenomenon is a high variance in the runtime of algorithms for hard problems depending on the complexity of the specific problem instance [4][5][6].

For this reason, the empirical study of – especially NP-hard – problems has gained significant attention in recent years. By studying the runtime distribution of algorithms on different problem instances, one can obtain a better understanding of not only the algorithms' strengths and weaknesses, but also of the nature of problem complexity. The insights into the origins of complexity can be leveraged in many ways, e.g.:

- Design of enhanced algorithms that can deal with complex problem instances more effectively.
- The ability to predict the runtime of an algorithm on a specific problem instance provides a sound basis for deciding whether the problem instance can realistically be solved with the given algorithm, or a simpler heuristic should be used instead as fallback.
- Runtime prediction also enables the choice of the most appropriate algorithm for a given problem instance from an algorithm portfolio [3].
- Runtime prediction also enhances load balancing when several problem instances are solved in parallel on multiple machines.

The aim of BCAT is to enable detailed and sophisticated assessment of the complexity of different instances of a problem by running one or more algorithms on the given problem instances and by measuring their running time and other characteristics. The measured values can be compared with results of static analysis of the problem instances to analyze statistically the correlation between different aspects that may impact complexity. The aim of BCAT is to support such research in a comprehensive way.

BCAT is available as a research prototype from http://sourceforge.net/projects/bcat/.

II. System Overview

BCAT is centered around four main notions: problems, algorithms, analyzers, and converters.

A **problem** is an instance of any combinatorial problem. Both decision problems and optimization problems are supported. Problems can be loaded from a file, or generated dynamically. For problem generation, different *models* can be supported. For instance, there are multiple models to generate a graph:

- In the *Random(n,p)* model, the graph has n vertices and each pair of vertices is connected by an edge independently from each other with probability p.

- In the *SmallWorld(n,k,p)* model, one starts with a ring topology in which each of the *n* vertices has *k* neighbors. Then, each edge is moved with probability *p* to another pair of vertices.

As can be seen from these examples, the parameters required for problem generation are model-dependent.

An **algorithm** is capable of solving some problems. Several kinds of algorithms are supported:

- Algorithms for both decision problems and optimization problems
- Both deterministic and randomized algorithms
- Exact, approximation, as well as heuristic algorithms

BCAT also supports the inclusion of external solvers and optimization packages (e.g. SAT solver, ILP solver) as algorithms.

The implemented algorithms can build upon each other, e.g. use each other as subroutines.

BCAT records the run algorithms' performance in terms of runtime and quality of found solution.

An **analyzer** is similar to an algorithm in that it is also run on a problem. However, in contrast to an algorithm, an analyzer does not attempt to solve the problem. Rather, it gathers some useful information about it. This information can be used for research purposes to gain insight as to what aspects of a problem instance account for its complexity.

As an example, an analyzer can compute the average degree in a graph. BCAT can be used to run both a graph coloring algorithm and this analyzer on a set of graphs, and then use the measured algorithm runtimes and average degrees to assess the correlation between the graphs's average degree and the complexity of its coloring.

Although not enforced explicitly by the system, analyzers are usually significantly faster than algorithms.

Problems can be converted to other problems by means of a **converter**. For example, many combinatorial problems can be converted in a natural way to a Boolean satisfiability (SAT) problem. In a second step, the obtained SAT problem can be solved with an appropriate solver. This way, a SAT solver can be used in combination with the converter to solve the original problem. BCAT also supports more complex scenarios: e.g., a problem can be converted to multiple other formats, or a chain of converters can be used to convert through multiple intermediate problems to the desired problem that is then solved by an appropriate algorithm.

BCAT provides a simple text-based interface (see Section IV for details) for defining what converters, algorithms, and analyzers to run on what problems. This information is called **configuration**. In order to facilitate simple usage in various scenarios, the user provides the configuration in a descriptive manner in a separate file. This way, the user can easily change the configuration for a new run of BCAT without having to touch the code.

In order to facilitate reuse, BCAT provides a library of popular **data structures**. These data structures are implemented in a very general manner, so that they can be used in a wide array of problems, algorithms, and analyzers. For example, the implementation of the graph data structure makes it possible to access the graph's connectivity information either as an adjacency matrix or as a list of incident edges per vertex, so that an algorithm implementation can pick the most appropriate representation.

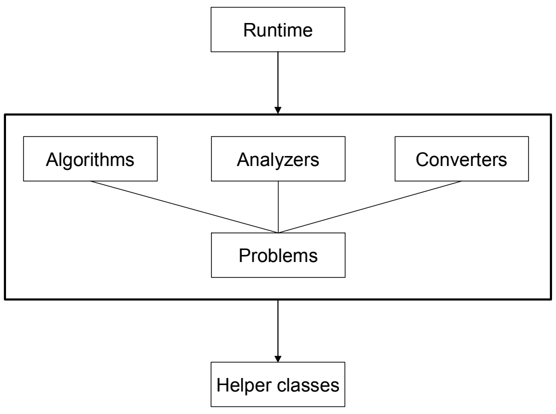

Figure 1. High-level overview of the BCAT system

BCAT's **output** are statistics files containing data about the run algorithm's performance per problem instance as well as the data emitted by the analyzers. The statistics files are in the CSV (comma-separated values) format for easy post-processing in a spreadsheet application. The post-processing may include generation of charts or statistical calculations, such as hypothesis testing or computation of correlation.

From a **technical** point of view, BCAT is intended to be a sound, well-structured, platform-independent and highly efficient framework for algorithm implementation. These requirements led to the choice of C++ as programming language.

BCAT supports the implementation and debugging of algorithms by means of unified logging capabilities, with conveniently settable level of logging detail.

For efficient usage of BCAT, it has a simple but powerful text-based user interface program, with which all files used or generated by BCAT can be easily viewed and edited. Other programs, that perform pre- and post-processing operations, can be attached to the system at this point. For example, in the case of graph colouring, the GraphViz package (http://www.graphviz.org/) has been attached for visualization of the coloured graphs.

III. System Architecture

A high-level overview of the system is shown in Fig. 1. As can be seen, BCAT exhibits a layered architecture. The components in the middle layer correspond to BCAT's basic notions of problems, algorithms, analyzers, and converters. The bottom layer consists of a set of loosely coupled helper classes that serve general purposes and can be used by all middle-layer components. Such helper classes provide the following features:

- Data structures
- Collection of runtime information
- Logging support

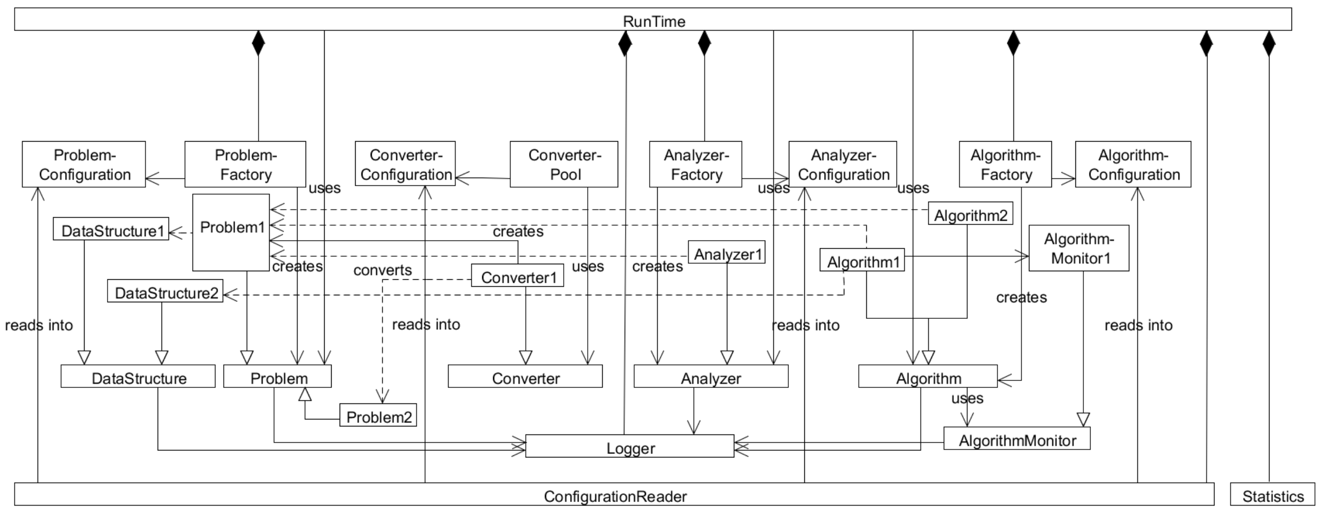

Figure 2. An example for the connection between classes in BCAT

The `Runtime` component in the top layer is responsible for integrating the components of the system and for organizing the main flow of the program by appropriate configuration handling.

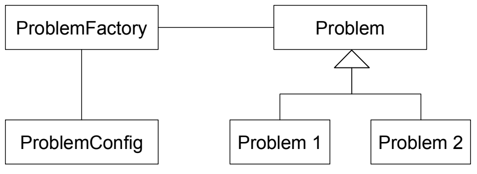

Figure 3. Classes inside the `Problems` component

When zooming into one of the middle-layer components, e.g. the `Problems` component, one can see that the component is not merely a loose collection of problem classes (see Fig. 3. for details). The `Problems` component is also responsible for iterating through all problem instances specified in the problem section of the configuration, involving loading of problem instances from files and generating problem instances on the fly. The appropriate logic is implemented in the `ProblemFactory` class, while the relevant configuration information is contained in the `ProblemConfiguration` class. Moreover, the component also contains a `Problem` class, the superclass of all specific problem classes.

Following an analogous logic, the `Algorithms` component consists of the `Algorithm` class, the specific algorithm classes (which are subclasses of `Algorithm`), and the `AlgorithmFactory` and `AlgorithmConfiguration` classes. Also the `Analyzers` component has analogous contents.

Between the generation of the main problem and the solution and/or analysis of the problem, conversion is possible. The `ConverterPool` encapsulates the whole conversion process, including the iteration through the conversion tree that describes how problems should be converted to each other. The `ConverterPool` delivers the converted problems to the `Runtime`, thus there is no need for the `Runtime` to deal with converters directly. Implementing conversion logic in a different component serves reusability of our previously written algorithms and analyzers. The `ConverterPool` uses the `ConverterConfiguration` in order to work as described in the configuration file.

Fig. 2. shows the connection between the classes in a case when there are two problem classes (`Problem1` and `Problem2`), one converter class (`Converter1`), two algorithm classes (`Algorithm1` and `Algorithm2`), and one analyzer class (`Analyzer1`). As can be seen in the Figure, `Algorithm1`, `Algorithm2`, and `Analyzer1` all work on `Problem1`, which in turn makes use of `DataStructure1`. `Converter1` converts `Problem1` from `Problem2`. `Algorithm1` uses another data structure `(DataStructure2)`.

BCAT's output is generated by the `Statistics` class, which gathers the necessary information from the factories and the `Algorithm` and `Analyzer` classes, which – as described above – are exported to the public interface of the appropriate component.

IV. CONFIGURATION DESCRIPTION

As mentioned earlier, BCAT receives the configuration information – that is: what algorithms, analyzers, and converters to run on what problems – in a separate file. For this purpose, a configuration description language (CDL) was designed.

CDL's syntax is based on the C/C++ syntax. Basically, a CDL file contains the reference to zero or more algorithm(s), zero or more analyzer(s), zero or more converter(s), and one or more problem instances. Each problem instance is either defined by making reference to a model to generate it or by the name of a file from which the problem instance should be loaded. Algorithms, analyzers, converters, and problem models have zero or more parameter(s). A parameter has a name and either one value or a set of values; a set of values can be defined either one by one or, in case of equidistant values, with a construct similar to a `for`-loop.

```
#define LogLevel SHOW_ALL

Problem GraphColoring
{
   Model RandomNP X 100
   {
      Parameter number_of_nodes = 70
      Parameter probability = for( 0; 1; 0.01 )
   }
   Files:
      le450\le450_15a.col
      le450\le450_5b.col
}

Converter ColoringToSat
{
   InputProblem = GraphColoring
   OutputProblem = Satisfiability
}

Analyzer GraphAnalyzer
{
   InputProblem = GraphColoring
   Parameter GreekLetters = enable
}

Algorithm GraphColouringBB
{
   InputProblem = GraphColoring
   Parameter SaveIfSuccess = disable
}

Algorithm GraphColouringGA
{
   InputProblem = GraphColoring
   Parameter MaxPopulation_size = { 100 200 300 }
   Parameter Mutation_probability = 0
   Parameter Structured_init_population_probability = 0.5
}

Algorithm SatSolver
{
   InputProblem = Satisfiability
   Parameter ClauseLearning = true
}
```

Figure 4. Example configuration file

CDL ::= Block*

Block ::= ProblemBlock | AlgorithmBlock | AnalyzerBlock | ConverterBlock

ProblemBlock ::= **Problem** *ProblemName* { ModelBlock* FilesBlock? }

AlgorithmBlock ::= **Algorithm** *AlgorithmName* ParameterBlock

AnalyzerBlock ::= **Analyzer** *AnalyzerName* ParameterBlock

ConverterBlock ::= **Converter** *ConverterName* ParameterBlock

ParameterBlock ::= **{** ParamStatement* **}**

ParamStatement ::= **Parameter** *ParamName* **=** *Value* | **Parameter** *ParamName* **{** *Value+* **}** | **Parameter** *ParamName* **=** **for** **(** *StartValue***;** *EndValue***;** *Step* **)**

ModelBlock ::= **Model** *ModelName* **X** *Number* ParameterBlock

FilesBlock ::= **Files:** *FileName**

Figure 5. CDL syntax

Algorithms, analyzers, and converters have a special parameter called "`InputProblem`" that defines on which input problem they should be run. In addition, a converter also has a parameter "`OutputProblem`" that defines what problem the converter should emit. For these special parameters, the keyword "`Parameter`" can be omitted. An example configuration file is given in Fig. 4.

CDL can be defined using EBNF notation as in Fig. 5. (**bold underline**: terminal symbols, *italics*: tokens).

The CDL file defines iteration through all possible combinations of several dimensions:

- All problems that are specified to be loaded from files
- All problems that are specified to be generated according to a model
- For each model, the specified number of instances are generated
- All specified (chains of) converters are run
- All specified analyzers are run
- All specified algorithms are run
- For each parameter whose value is specified as a set (i.e., a list of values within curly brackets or in a `for`-loop-like manner), all given values are considered

A CDL file may also contain C/C++ style comments (both line comments and block comments) as well as preprocessor directives to adjust general settings, e.g. logging detail level.

Extensibility was an important goal in the design of CDL, so there are also possibilities that are not yet exploited, but preserved for future use – such as hierarchic variable definition by directives.

There are also several semantical constraints, not enforced by the EBNF syntax, that a CDL file must conform to in order to be valid:

- All the problems, algorithms, analyzers, and converters mentioned in the CDL file must refer to actual problem, algorithm, analyzer, and converter classes implemented in the system.
- The parameters in the CDL file must refer to the actual parameters required by the problem, algorithm, analyzer, and converter implementations. Moreover, the parameter values must be valid.
- The specified algorithms, analyzers, and converters must be compatible with the specified input problem.
- The input problem of an algorithm, analyzer, or converter must be specified either as a base problem or as the output problem of a converter in the CDL file.
- There must be no cycle of conversions.
- The given files must exist and must contain problem instance data in the appropriate format, as required by the problem class.

V. Operation of the System

After starting the system, the following steps are carried out in BCAT:

1. A `Runtime` object is created in the `main` function.
2. Since all of the classes contained within `Runtime` are implemented by the Singleton pattern [2], objects of those types do not have to be explicitly created, just initialized.
3. This is followed by the reading and interpretation of the configuration file. This is performed by the

`ConfigurationReader` class. It parses the CDL file, checks its syntax and semantics, and creates configuration objects for all factories and the `ConverterPool`, which describe for the specific factory or pool the relevant part of the read configuration.

4. All factories and the `ConverterPool` get the needed configuration object and – based on these configuration objects – they create in turn the required specific problem, analyzer, algorithm, and converter instances.
5. If needed, external monitors can be started to monitor the state of algorithm execution. The external monitors communicate with BCAT via files or sockets.
6. Now the whole machinery is in place for starting the execution phase, in which the `Runtime` iterates through all problems via the `ProblemFactory`'s iterator interface.
7. Before solving and analyzing problems for each problem instance, `ConverterPool` lets the specified converters perform the conversion to the new problem instances.
8. For each (converted) problem instance, the `Runtime` runs all analyzers and algorithms as described in the conversion and solution tree, and sends the results to `Statistics` for processing. Analyzer- and algorithm factories first parameterize the analyzers and algorithms based on the configuration objects, run them, and then give back their results to the `Runtime`.

Every class, in which a failure may occur (problem, analyzer, algorithm, and converter classes) during the execution, uses the `Logger` to record its events.

Built-in data structures can be used by problems (for example, the graph data structure can be used by the graph colouring problem implementation), analyzers, converters, and algorithms (for example trees or hash tables for storing any dynamic information during the run of the algorithm/analyzer/converter).

VI. EXTENSIBILITY

One of the most important purposes of BCAT is to help in the implementation and testing of new algorithms on built-in or new problems. Accordingly, one can implement new problems and/or algorithms easily so that the showed functionality of BCAT will be applicable in the context of the newly created problem/algorithm as well.

For example, in order to add a new algorithm, a user just has to derive their class from the existing `Algorithm` class and implement the abstract functions, with which instances of the class can communicate with the `Runtime`. Moreover, the new class has to be made known to the system by inserting the necessary line of code into the `AlgorithmFactory`. After integrating an algorithm this way into the system, BCAT ensures that it will be used appropriately, parameterized as specified in the configuration file.

For problems, analyzers, and converters the method is analogous.

Using BCAT's UI one can easily generate new problem, converter, analyzer and algorithm classes from templates, which show how one should implement the abstract functions in order to work properly in the system.

VII. PRACTICAL EXPERIENCE

BCAT has been used successfully in the authors' research on the complexity of graph coloring. It helped in designing, implementing and testing various graph colouring algorithms such as greedy algorithms, branch-and-bound, genetic algorithm and hybrids of them. In the course of this research, also a number of graph analyzers have been implemented to compute various statistical properties of the input graph, such as number of components, average degree of the vertices, and average distance of pairs of vertices. The analyzers also include heuristics to estimate some more complex graph properties, such as clique number and maximum stable set size.

The most complex algorithm that has been implemented is the coloring algorithm based on branch-and-bound. It clearly demonstrates the power and convenience of using BCAT. The algorithm is implemented in a layered fashion:

- The lower layer is responsible for providing the abstract branch-and-bound logic that can also be used with other problems. It includes the ways of moving in the search tree, administrating various variables, branching, backtracking, and learning. This layer defines many abstract functions, which should be implemented in higher layer(s).
- The graph colouring layer is responsible for the implementation of these functions. This layer does not care about the concrete branch and bound implementation, just uses the functions of the lower layer. It implements how branch-and-bound can be used in the context of the graph colouring problem.
- For convenience, a third layer has also been created which functions as a "graph colouring toolbox" based on the graph colouring layer. This layer is mainly responsible for pre- and post-processing steps, running modes (see below) and combination with other algorithms.

The separation of the first and second layer fits perfectly into the object-oriented design of BCAT. `BranchAndBound` is an abstract C++ class, and it is also logically abstract. It realizes an algorithm, which can be used in other algorithms, but cannot be run itself. Therefore, the `RunTime` communicates only with the graph colouring branch-and-bound object. In order to use branch-and-bound for other problems, only the second layer needs to be implemented, and the generic part of the implementation can be reused.

BCAT's configuration handling mechanism and the CDL language proved very useful in the experimentation phase. This was further improved with the "graph coloring toolbox". For example, it supports regression testing: to test a new version of an algorithm, BCAT is provided with the problem and the expected results, and switched to `TestAlgorithm` mode with the appropriate directive. BCAT then runs the algorithm to check whether it behaves as expected.

Currently, the implementation of further problems and algorithms is in progress. Along the way, improvements to the framework itself are constantly identified and implemented. The authors also invite other researchers to use BCAT by adding their own problems and algorithms.

An external monitor application was developed in Java for visualization of the current state and the results of the algorithms. This monitor communicates with BCAT via socket and receives XML messages from the framework. These can be algorithm messages, describing the connection with a running algorithm, system messages like logs, events of the algorithms (e.g., the algorithm solved the problem), or object representations of the running algorithms. This last one can be especially useful in the debugging of a complicated algorithm.

VIII. Related work

There is a huge number of algorithm implementations available both as open-source and as commercial packages. Prominent examples include solvers for linear programming, integer programming, convex optimization, constraint satisfaction and Boolean satisfaction, as well as meta-heuristics such as genetic algorithms and tabu search. On the other hand, there are few generic packages that allow uniform manipulation of a variety of problems and algorithms for research purposes, with adequate support for complexity research.

One such package is the Java Combinatorial Optimization Platform (jcop, see http://jcop.sourceforge.net/en/index.html). Similarly to BCAT, jcop also supports generic problem and algorithm classes that can be extended to add new problems and algorithms. It also supports runtime measurements, with a rich set of display possibilities. However, the problem and algorithm model of jcop is constrained. jcop models a problem by the associated search space, thus only allowing algorithms that work directly on the search space. For instance, a branch-and-bound algorithm does not fit into this model because it manipulates partial solutions instead of states in the search space. Moreover, jcop runs the algorithms step by step. This prohibits the usage of external solvers that do not provide such an interface.

A similar package is the Optimization Algorithm Toolkit (OAT, http://optalgtoolkit.sourceforge.net/index.php). OAT also provides a set of problems and algorithms that can be extended with user-defined ones. It also provides a graphical user interface for users to experiment with problems and algorithms, as well as statistical analysis tools. On the other hand, similarly to jcop, also OAT's problem and algorithm model is limited, focusing on search problems in artificial intelligence.

A comparable effort is the HotFrame project (Heuristic OpTimization FRAMEwork, http://www1.uni-hamburg.de/IWI/hotframe/hotframe.html). As the name suggests, HotFrame is targeted at heuristics such as simulated annealing, tabu search and evolutionary algorithms. It offers a rich set of features that form the foundation of such algorithms (e.g., related to search neighborhood and tabu criteria). HotFrame, too, can be extended with custom problem and algorithm types, and can be embedded into applications as a solver. It is clear though that HotFrame is limited to heuristics and not meant for other kinds of algorithms.

There are also some commercial software packages with several implemented optimization routines, e.g. the NAG Libraries (http://www.nag.co.uk/numeric/numerical_libraries.asp) or the IMSL Numerical Libraries (http://www.vni.com/products/imsl/). However, these packages do not support experimentation with new algorithms, but rather the usage of the available methods.

The COIN-OR project (COmputational INfrastructure for Operations Research, http://www.coin-or.org/index.html) is similar in its objectives to this work, as it also defines a generic software infrastructure for algorithmic research. However, in contrast to this work, COIN-OR is a collection of loosely coupled software projects, and not a single environment for analyzing the performance of different algorithms on different problems.

IX. Conclusions and future work

This paper described BCAT, a framework for the implementation of problems, algorithms, analyzers, and converters for the purposes of algorithmic complexity research. The system's main characteristics, its architecture, and way of operation were presented. Details of configuration handling in BCAT and the underlying CDL language were also shown.

The authors' experience with BCAT in the field of graph coloring has been very promising. Nevertheless, BCAT is work in progress, as it is constantly improving and extended with new features.

The mentioned graph colouring algorithms are being further refined and work continues on the implementation of new problems and appropriate algorithms.

External solvers will also be increasingly integrated into the system.

Acknowledgment

This work was partially supported by the Hungarian National Research Fund and the National Office for Research and Technology (Grant Nr. OTKA 67651).

References

[1] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 2nd ed., MIT Press, 2001.

[2] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

[3] C. P. Gomes and B. Selman, "Algorithm Portfolios," *Artificial Intelligence*, vol. 126, pp. 43-62, 1-2/2001.

[4] C. P. Gomes, B. Selman, N. Crato, and H. Kautz, "Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems," *Journal of Automated Reasoning*, vol. 24, pp. 67-100, 1-2/2000.

[5] I. P. Gent and T. Walsh, "Easy problems are sometimes hard," *Artificial Intelligence*, vol. 70, pp. 335-345, 1-2/1994.

[6] T. Hogg and C. P. Williams, "The hardest constraint problems: A double phase transition", *Artificial Intelligence*, vol. 69, pp. 359-377, 1-2/1994.

[7] Z. Á. Mann and A. Orbán, „Optimization problems in system-level synthesis," *Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, 2003.

[8] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*, 3rd ed., Prentice Hall, 2009.