# A Problem Meta-Data Library for Research in SAT

Markus Iser, Carsten Sinz

Institute for Theoretical Informatics (ITI)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

**Abstract**

Experimental data and benchmarks play a crucial role in developing new algorithms and implementations of SAT solvers. Besides comparing and evaluating solvers, they provide the basis for all kinds of experiments, for setting up hypothesis and for testing them. Currently – even though some initiatives for setting up benchmark databases have been undertaken, and the SAT Competitions provide a "standardized" collection of instances – it is hard to assemble benchmark sets with prescribed properties. Moreover, the origin of SAT instances is often not clear, and benchmark collections might contain duplicates. In this paper we suggest an approach to store meta-data information about SAT instances, and present an implementation that is capable of collecting, assessing and distributing benchmark meta-data.

## 1 Introduction

The experiment is the core sources of knowledge in science. It is the vital source of data in a feedback loop between hypothesis formation and theory falsification. Algorithm engineers use experiments to evaluate their methods. Runtime experiments are often based on publicly available sets of benchmark problems. A set of characteristics or meta-data can be associated with each benchmark problem.

Benchmark meta-data can be used for benchmark classification and for differentiated analysis of algorithmic methods. It is very common that a certain algorithmic method or heuristic configuration works well on a specific *type* of problems but not so on others. Such correlations between the feature space of our benchmarks and the configuration space (or even fundamental algorithms) are of great interest. Disclosure and analysis of such correlations can lead to better hypotheses and thus more solid theories.

Recent approaches in SAT solver development show that automatic feature extraction and algorithm selection or heuristic configuration through machine learning are crucial to state-of-the-art solver performance [4, 3]. Increasing the number of automatically extracted problem features can improve predictions made by machine learning. But problem meta-data is not only useful for the training of machine models, it can also help improve the interpretation of experimental results. And if problem meta-data is used like this, there is no restriction to data that can be automatically and efficiently extracted from the problem.

Attempts to provide organized sets of benchmarks have been realized e.g. by Hoos et al. with SATLib [6, 5]. The organizers of the bi-anual SAT Competition [2] assemble sets of benchmarks that are sufficiently hard but not too hard for state-of-the-art solvers. Unfortunately, problem meta-data is very diverse and is, like the problems themselves, distributed over multiple sources, and usually it has to be manually gathered by the experimenter. Given a set of anonymous SAT problems (e.g. from SAT competition), some information is hard to get (e.g. problem

| Meta Data | Solutions | Feature Extraction |
|---|---|---|
| Author | Solution to SAT | Problem Class (Horn, 2-SAT, etc.) |
| Generator | Number of Solutions | Max Clause Length |
| Application Domain | Isomorphic Problems | Tree-Width |
| Local Path | Size of shortest recorded proof | Number of connected components |
| Online Source | Best recorded runtime | ... |
| Competition Usage | ... | |
| ... | | |

Table 1: Meta-data of various types that can be made available in our system

origin), and some types of data are computationally hard to calculate (e.g. solution, number of solutions).

The main goal of the presented tool is to easily gather and exchange meta-information about benchmark problems. Databases of benchmark meta-data should be easily maintainable and be distributed to researchers around the world. Meta information shall be collected from different sources and users should be able to easily add their knowledge to the pool.

The paper is structured as follows. In section 2 we will present some use-cases for the presented tool. In section 3 we will explain the structure of the database and the reasons for the choice or benefits of the chosen structure. Section 4 will provide insights into the implementation and usage of our tool. In section 5 we present some early experience with our tool and possibilities for future extensions.

# 2 Use Cases

In order to create exchangeable collections of meta-data in a decentralized manner, the benchmark to data association has to be established via benchmark fingerprinting, i.e. well defined hash values of benchmark problems. As meta-data is associated with hash values of benchmark problems, it is *easy to exchange* meta-data with colleagues (see section 2.1).

Our system can be used to *find duplicate problems* (2.2) in competition sets or own collections of problems. It can be used to *find correlations* (2.3) that are particularly suitable for algorithm selection and interpretation of results.

The database can contain *collections of metrics* (e.g. "tree-width" or "number of connected components"). Also information about the source of the problem might be helpful, e.g. *urls of a benchmark repository*, *problem author* and *application name*.

Even *algorithmic results* that are hard to calculate, such as the solution to the SAT problem, the solution to the #SAT problem (i.e. number of solutions) or the solution to an isomorphism check can be exposed. It might also contain the shortest known proof for an unsatisfiable problem or just its size. *Reporting and comparison* of runtime results can easily be accomplished with our tool. Table 1 summarizes several types of data that can be made available in our system.

## 2.1 Global repository for SAT meta-data

Collections of SAT benchmarks can take up a large amount of storage. Also filenames can be ambiguous and the same problem might occur with different names in several collections of benchmark problems. Thus, a meta-data database should only reference benchmark problems via their fingerprints in the form of hash values and not the benchmark problem itself.

This is beneficial as researchers around the world can easily exchange benchmark attributes without the need to care about benchmark identification. The system solves the identification problems automatically via hash values.

### 2.2 Finding Duplicates

Benchmark collections tend to contain duplicate benchmarks. For example the benchmark collections that are regularly compiled for the annual SAT Competition often contain benchmarks that are used in previous collections. We even found that some benchmarks are present with different filenames in the same benchmark collection multiple times.

Our system detects simple duplicates (those with an empty `diff`) directly, as they carry equal fingerprints or hash-values. However there are also duplicates, like isomorphic problems through variable renaming and clause reordering, which are not that easy to detect. Checks for isomorphism in benchmark problems impose hard algorithmic problems. It is beneficial to store and exchange this information. Our system assigns ids of equivalence classes to each benchmark. Once an isomorphism has been detected by one researcher, this information can be exposed via our systems equivalence-class table.

### 2.3 Finding Correlations

Most new algorithms or heuristic configurations under test are unsuccessful in improving general solver performance. They might be beneficial for a specific subset of the benchmarks but catastrophic on another. Many such outcomes are considered being *inconclusive*.

Given a huge collection of meta-data, correlations of solver behavior and attributes that the experimenter might not even have thought of might be revealed. Experimental results that would otherwise be classified as being inconclusive might expose new coherences and lead to conclusions and even new theoretic models.

## 3 Database Layout

The database should be small enough to be distributed quickly. Furthermore the data model should be extensible, without changes to the meta model, i.e. it should be possible to add new types of meta data without changing the software. The tool should be able to access different sources of data; for example a local database (e.g., for own experiments) and a global database (for the SAT community). Researchers can share their individual database with a collection of certain metrics in the web.

### 3.1 Hashing Benchmarks

In order to keep the database small, the benchmarks themselves should not be part of the database. The same benchmark problem can come with different filenames. Anyway the identification of a specific problem must be possible. Therefore we use hash values to identify a benchmark.

Our system creates an `md5`-hash for the unpacked and normalized benchmark. Unpacking ensures that the hash value is invariant to benchmark compression. Normalization includes removal of comment-lines and leading or duplicate whitespace characters. Also the line-separator characters are normalized and replaced by unix-style line-separators. The hash value is created

```
usage: gbd.py [-h] [-d [DB]] {init,reflect,group,tag,query,resolve} ...

Access and maintain benchmark databases.

positional arguments:
  {init,reflect,group,tag,query,resolve}
                        Available Commands:
    init                Initialize Database
    reflect             Reflection, Display Groups
    group               Create or modify an attribute group
    tag                 Associate attribues with benchmarks
    query               Query the benchmark database
    resolve             Resolve Hashes

optional arguments:
  -h, --help            show this help message and exit
  -d [DB], --db [DB]    Specify database to work with
```

Figure 1: The basic commands of the GBD Command-line Interface

such, that a call to md5sum on an already unpacked and normalized DIMACS-file would create the same hash-value.

During an initial local bootstrapping process, a local benchmark table is created that maps the hash values to local paths where the actual problems reside. The system treats the such recorded local path like a specific type of meta information that is attached to each hash-value. However, the benchmark table has a special meaning, as it is used to resolve problem hashes against actual paths in the local filesystem.

### 3.2 Attribute Groups

Problem attributes are organized in groups. For each attribute group, our system uses a two column table to store hash and value pairs. By default, attribute groups store strings such that queries run a substring search on the attribute group. Numeric types are supported as well, in which case queries can also use “less than” or “greater than” constraints.

Furthermore, attributes of a certain group can be constrained to be unique (i.e. attach at most one value to each problem) or have a default value (e.g. SAT, UNSAT and the default value UNKNOWN). If an attribute group is configured to have a default value and to be unique, our system ensures that there is always exactly one value stored for each locally available benchmark problem.

## 4 Technology and Implementation

We developed the system GBD (Global Benchmark Database) which is maintained in a publicly available GIT repository [1]. The database is stored in a `SQLite` file and is created and maintained by a set of `Python` scripts. The GBD Command-line Interface is organized into six basic commands: init, reflect, group, tag, query and resolve. Figure 1 shows the help page of

GBD. Each individual command has its own help page that can be studied for further reference. In the following we give an overview on the usage and parameters of the most important commands.

The *init* command (section 4.1) provides access to bootstrapping and other initialization functionality, whereas the *reflect* command (section 4.4) provides reflection related functionality, such as gathering information about existing groups and their properties.

The *group* command (section 4.3) provides access to group creation and modification functionality. The *tag* command (section 4.5) can be used to associate attributes with benchmarks. With the *query* command (section 4.6) attributes and groups can be used to search for sets of hashes and to combine them using set operations. Using the *resolve* command (section 4.7) a query resultset of benchmark hashes can be translated to benchmark paths in the filesystem.

Note that, in order to simplify usability, most commands use the parameters `-n` to specify a group name and `-v` to specify an attribute value. In general hash values of benchmarks are printed line-wise to stdout and read line-wise from stdin, i.e. usage of pipe functionality is encouraged.

## 4.1 Initialization

The *init* command creates a database and some basic tables. Via parameter `-p` the path of a directory is given. This directory is recursively searched for benchmarks and their hashes and paths are integrated into the benchmark database in the local benchmark table. With this bootstrapping command the benchmark table is filled with local benchmark data (i.e. the local physical path to the benchmark). The benchmark is hashed as described in 3.1 and that hash is stored alongside its path in the local benchmark table. The local benchmark table is treated not different than a special attribute group (see section 4.3).

## 4.2 Database Selection

GBD supports database selection via the `--db` paramater. The argument must be a local path to a SQLLite database that was created by GBD. This command simplifies working with multiple datasources, e.g. including meta-data collections of colleagues in ones own research. If not specified otherwise the default database "local.db" in the GBD directory is used.

## 4.3 Creation of Groups

**Usage: gbd group [-h] [-v VALUE] [-u] [-t text,integer,real] [-r] [-c] name**

| | |
|---|---|
| name | Name of group to create (or modify) |
| -h, –help | show this help message and exit |
| -v VALUE, –value VALUE | Specify a default value for the group |
| -u, –unique | Specify if the group stores unique or several attributes per benchmark |
| -t TYPE, –type TYPE | Specify the value type of the group. TYPE ∈ {"text", "integer", "real"} |
| -r, –remove | If group exists: remove the group with the specified name |
| -c, –clear | If group exists: remove all values in the group with the specified name |

Table 2: The parameters of the group command of the GBD Command-line Interface

Groups are distinct categories for attribute values. The group name is basically the name of a certain attribute that can be specified for a benchmark, such as “author” or “solution”. Table 2 summarizes the paramaters of the group commandy. Groups can have a default value (e.g. “UNKNOWN”), in which case every benchmark gets at least the default value assigned. Groups can be *unique*, i.e. every benchmark gets at most one value in that group. E.g. making the “solution” group *unique* prevents benchmarks from having solution “UNKNOWN” and “UNSAT” at the same time. Groups have a value type, which can be “text”, “integer” or “real” (the default value type is “text”). The type determines the possibilities of how to add attribute values and query for values in the group (see Section 4.6).

#### 4.3.1 Example:

Creation of group *smallest_tracked_proofsize*:

```
gbd group -n smallest_tracked_proofsize -u -t integer
```

### 4.4 Reflection on Groups

The *reflect* command prints information about existing groups. If a specific group is given with `-g` or `--group` then the properties of that specific group are shown. If the `-v` flag is used, then the distinct values in that group are shown.

### 4.5 Association of Attributes with Benchmarks

| **Usage: gbd tag [-h] -v VALUE [-r] name** | |
|---|---|
| name | Name of attribute group |
| -h, –help | show this help message and exit |
| -v VALUE, –value VALUE | Attribute value |
| -r, –remove | Remove attribute from hashes if present, instead of adding it |

Table 3: The parameters of the tag command of the GBD Command-line Interface

Attribute values can be associated with a set of benchmark problems in a specific attribute group. An attribute value and a group name must be specified. Table 3 summarizes the parameters of the *tag* command. The attribute value must fit the type of the group. If the group is *unique* (see section 4.3), each hash can only have one associated attribute value in that group such that existing values are overwritten automatically.

The hashes are read from `stdin`. Common usage is to first query for specific benchmarks hashes and then to pipe them through the command. Several queries (see section 4.6) can be combined and the hashes can all be tagged or attributed at once.

#### 4.5.1 Example:

Adding an attribute value to a benchmark problem:

```
echo $hash | gbd tag -n smallest_tracked_proofsize -v 123
```

| Usage: gbd query [-h] [-v VALUE] [-o union,intersection,difference,symdiff] [name] | |
|---|---|
| name | Specify attribute group to query |
| -h, –help | show this help message and exit |
| -v VALUE, –value VALUE | Specify attribute value to query for. Searches for substring in groups of textual type. In groups of numeric types prepend "<", ">" or "=" to specify type of comparison. |
| -o TYPE, –operation TYPE | Optionally specify how to combine with hashes read from stdin. TYPE ∈ {"union", "intersection", "difference", "symdiff"} |

Table 4: The parameters of the query command of the GBD Command-line Interface

## 4.6 Query

The query command provides methods to find benchmarks that match the specified attributes. A set of hash values is printed line-wise. Table 4 summarizes the parameters of the query command. Several queries can be combined via pipe operations. The parameter `-o` is used to specify how the hashes of the current query shall be combined with previous queries.

### 4.6.1 Example:

The following query will return all hashes that contain the string 2017 in their path:

```
gdb query -n benchmarks -v 2017
```

### 4.6.2 Example:

The following sequence of commands will create the intersection of two queries. The first query returns all problems that carry the attribute "2017" in the group "competition". The second query returns all problems with the attribute "sat" in the group "solution":

```
gdb query -n competition -v 2017
 | gdb query -o intersection -n solution -v sat
```

## 4.7 Resolve

As the query command only returns hashes of problems, in order to obtain the paths to the problems, the resolve command is used to resolve these hashes against the paths in the local benchmark table. The local benchmark table was created by the bootstrapping procedure as described in section 4.1.

As many problems occur in multiple publicly available compilations, it is not uncommon that `resolve` finds multiple representatives of one hash-value on the disk. Therefore the resolve command can be used with additional parameters. The parameter `-c` (collapse) is used to print only one representative per hash, that is usually the first in the list. Similarly, the `-p [str]` command constrains the returned path to contain the given substring. Both parameters can be used in combination in order to prefer resolution against a specific problem compilation.

### 4.7.1 Example:

The query returns all problems that carry the attribute "2017" in the group "competition". Then it is resolved against the local benchmark table. So the command returns a set of paths to problems that are locally available and fit the given query.

```
gdb query -n competition -v 2017 | gbd resolve
```

#### 4.7.2 Example:

Like in the previous example, the query returns all problems that carry the attribute "2017" in the group "competition". Then it is resolved against the local benchmark table. The resolve command has additional constraints to return only one path per hash, and to return only paths that contain the substring "folder1". So the command returns a set of paths to unique problems that are locally available (e.g. in "folder1") and fit the given query.

```
gdb query -n competition -v 2017 | gbd resolve -c -p folder1
```

## 5 First Results and Future Work

Initializing the database with benchmarks from the agile set of SAT Competition 2017 showed that more than half of them are duplicates of one another. This could easily be detected, as only less than 50% unique hashes have been generated and the `resolve` command often returned multiple candidates for the same hash within the agile set of benchmark problems. There is no problem with the hash function, a quick check with the `diff` command showed that in fact the problems are identical.

In addition to present database selection, which requires physical exchange of files, new methods to setup a REST web-service for database-exposure are currently under development. Setup of a directory service where sources of benchmarks together with their meta-information can be publicly collected might be a future step of our work, or even a community effort. Functions to import and merge tables from another database are currently under development.

Future work includes functionality to automatically extract meta-information from benchmark files. This could include meta-data from specially formatted comments, but also the automatic execution of specialized algorithms.

Currently, only the most basic ideas of decentralized meta-data collection are implemented. The implementation of specific use-cases on top of basic meta-data collection and distribution is considered future work. Such use-cases include automatic detection of correlations between solver performance and other meta-data and automatic solver comparison.

## References

[1] GBD public repository. git@git.scc.kit.edu:fv2117/gbd.git. Accessed: 2017-04-17.

[2] SAT competitions. http://satcompetition.org/. Accessed: 2017-04-17.

[3] Tomás Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT Race 2015. *Artif. Intell.*, 241:45–65, 2016.

[4] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Thomas Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger H. Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. Aslib: A benchmark library for algorithm selection. *Artif. Intell.*, 237:41–58, 2016.

[5] Holger Hoos and Thomas Stützle. Satlib. http://www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html. Accessed: 2017-04-17.

[6] Holger Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. 2000.