UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT

Dave A.D. Tompkins and Holger H. Hoos

Department of Computer Science
University of British Columbia
Vancouver BC V6T 1Z4, Canada
{davet,hoos}@cs.ubc.ca

Abstract. In this paper we introduce UBCSAT, a new implementation and experimentation environment for Stochastic Local Search (SLS) algorithms for SAT and MAX-SAT. Based on a novel triggered procedure architecture, UBCSAT provides implementations of numerous well-known and widely used SLS algorithms for SAT and MAX-SAT, including GSAT, WalkSAT, and SAPS; these implementations generally match or exceed the efficiency of the respective original reference implementations. Through numerous reporting and statistical features, including the measurement of run-time distributions, UBCSAT facilitates the advanced empirical analysis of these algorithms. New algorithm variants, SLS algorithms, and reporting features can be added to UBCSAT in a straightforward and efficient way. UBCSAT is implemented in C and runs on numerous platforms and operating systems; it is publicly and freely available at http://www.satlib.org/ubcsat.

1 Introduction

The propositional satisfiability problem (SAT) is an important subject of study in many areas of computer science, and is the prototypical $\mathcal{NP}$-complete problem. MAX-SAT is the optimisation variant of SAT; while in unweighted MAX-SAT, the goal is to find a variable assignment that satisfies a maximal number of clauses of a given CNF formula, in weighted MAX-SAT, a weight is assigned to each clause, and the goal is to find an assignment that maximises the total weight of the satisfied clauses. MAX-SAT is a conceptually simple $\mathcal{NP}$-hard combinatorial optimisation problem of substantial theoretical and practical interest; many application-relevant problems, including scheduling problems or most probable explanation (MPE) finding in Bayes nets, can be encoded and solved as MAX-SAT.

Some of the best known methods for solving certain types of SAT instances are Stochastic Local Search (SLS) algorithms; these are typically incomplete, *i.e.*, they cannot determine with certainty that a formula is unsatisfiable, but they often find models of satisfiable formulae surprisingly effectively [8]. For MAX-SAT, SLS algorithms are by far the most effective methods for finding optimal or close-to-optimal solutions [5]. Although SLS algorithms for SAT and MAX-SAT differ in their details, the basic approach is mostly the same. In the following, we mainly focus on SLS algorithms for SAT, while MAX-SAT algorithms will be discussed in more detail in Section 6.

In Figure 1 we provide pseudo-code for a typical SLS algorithm for SAT. Each *run* of the algorithm starts by determining an initial, complete assignment of truth values to all variables in the given formula F (*search initialisation*). Then, in each *search step*, a set of variables is selected, whose truth values are then changed from true to false or vice versa. Each change of a single variable's truth value is called a *variable flip*; almost all SLS algorithms perform exactly one variable flip in each search step, but there are cases in which a given SLS algorithm may flip no variables in a given search step (a so-called *null-flip*), or several variables at once (which is known as a *multi-flip*). Variable flips are typically performed with the purpose of minimising an *evaluation function* that measures *solution quality* in terms of the number of unsatisfied clauses under a given variable assignment. The search process is terminated when a *termination condition* is satisfied; this is typically the case when either a solution, *i.e.*, a satisfying assignment, has been found or when a given bound on the run-time, which is also referred to as *cutoff time* and which may be measured in search steps or CPU time, has been reached or exceeded. To overcome or avoid search stagnation, many SLS algorithms for SAT make use of a *restart mechanism* that re-initialises the search process whenever a *restart condition* is satisfied. For example, all GSAT and WalkSAT algorithms restart the search periodically [14, 13]. While restart mechanisms are crucial for the performance of some SLS algorithms for SAT, such as basic GSAT [14], they have been found to be ineffective in other cases [8].

```
procedure SLS-for-SAT(F)
    input: propositional formula F
    output: satisfying assignment of F or 'no solution found'
    a := InitialiseSearch(F);
    while not Terminate(F, a) do
        if Restart(F, a) then
            a := ReInitialiseSearch(F);
        else
            X := SelectVarsToFlip(F, a);
            a := FlipVars(F, a, X);
        end
    end
    if Solved(F, a) then
        return a
    else
        return 'no solution found'
    end
end SLS-for-SAT
```

Fig. 1. Pseudo-code for a typical Stochastic Local Search algorithm for SAT; a is a variable assignment, X is a set of variables in the given formula F.

Even though SLS algorithms for SAT and MAX-SAT have achieved great levels of success, we believe that there is still significant potential for further improvements. To further explore this potential we developed UBCSAT: an implementation and experimentation framework for SLS algorithms for SAT and MAX-SAT. Our primary objective was to create a software environment that facilitates research on and development of SLS algorithms. Specifically, the development of UBCSAT was based on the following six design principles and goals:

1. include highly efficient, conceptually simple, and accurate implementations of a wide range of prominent SLS algorithms for SAT and MAX-SAT;
2. facilitate the development and integration of new algorithms (and algorithm variants);
3. provide support for advanced empirical analysis of the performance and behaviour of SLS algorithms without compromising implementation efficiency;
4. provide explicit support for algorithms designed to solve the weighted and unweighted MAX-SAT problems;
5. provide an open-source software package that is publicly available to the academic community;
6. implement the project in a platform-independent way, avoiding non-standard programming language extensions.

Before discussing the design and features of UBCSAT in more detail, we briefly discuss two related software projects: OpenSAT and COMET.

The OpenSAT project [1] (`www.opensat.org`) was developed as a Java-based open source project for complete SAT solvers. A primary goal of OpenSAT was to make the advanced techniques and data structures used by state-of-the-art *complete* SAT solvers openly available in order to accelerate the development of new SAT solvers. Generally, the architecture and implementation of complete SAT solvers, which are based on the David-Putnam-Loveland procedure, differs considerably from that of SLS-based SAT algorithms, and traditionally there has been very little overlap between the algorithmic and implementation details used in these two types of SAT solvers. Therefore, using OpenSAT as the basis for achieving the previously stated goals, while probably not completely infeasible, appears to be problematic. In addition to the difficulty of supporting the development and implementation of SLS algorithms in a straight-forward way, the current lack of support for MAX-SAT solvers, and the fact that OpenSAT currently does not provide dedicated support for the advanced empirical analysis of SAT algorithms, it is somewhat questionable whether its Java-based implementation makes it possible to achieve performance that is competitive with the native reference implementations of high-performance SLS algorithms such as WalkSAT [13] or SAPS [9].

COMET [17] is an object-oriented language that supports a constraint-based architecture for local search. The COMET language is very sophisticated and can model SLS algorithms for solving advanced constraint satisfaction problems, but it neither offers explicit support for SAT/MAX-SAT nor does it provide tools for advanced empirical evaluation. While in principle, both of these issues could be addressed by realising the respective functionality within COMET, implementing UBCSAT in COMET seemed to pose the risk that in order to take full advantage of UBCSAT, users would have to understand both the idiosyncrasies

of COMET in as well as the architecture and interface of UBCSAT; we believe that as a consequence, UBCSAT would have been less accessible to its main target group, namely researchers interested in SAT and MAX-SAT. While there is evidence that COMET algorithm implementations are quite efficient, we do not have any perspective as to how these would compare with the native reference implementations of the state-of-the-art SLS algorithms covered by UBCSAT.

To achieve our goals of a platform-independent and highly efficient implementation, UBCSAT has been developed in strict ANSI C and tested on the most popular operating systems (Linux, WindowsXP, SunOS). In order to provide a state-of-the-art and platform-independent source of pseudo-random numbers, we have incorporated the "Mersenne Twister" pseudo-random number generator [10]. UBCSAT is publicly available for academic (non-commercial) use without restriction to encourage free and open use throughout the SAT research community[1].

For the remainder of this paper, we will describe the UBCSAT project in greater depth. In Section 2 we give an overview of the UBCSAT architecture and illustrate the fundamental concept of *triggered procedures*, which lies at the core of UBCSAT's efficient yet highly flexible design and implementation. In Section 3, we outline the current collection of SLS algorithms for SAT that are currently implemented within UBCSAT and compare their performance against that of the respective native reference implementations. In Section 4 we demonstrate how new algorithms are implemented within UBCSAT. In Section 5 we discuss the importance of empirical analysis in SLS research, and how UBCSAT can help facilitate empirical analysis. In Section 6, we describe how UBCSAT supports SLS algorithms for weighted and unweighted MAX-SAT. Finally, in Section 7 we summarise the key features and contributions of the UBCSAT project and outline some directions for future work.

2 The UBCSAT Architecture

One of the challenges of developing the UBCSAT project was to build a flexible, feature-rich environment without compromising algorithmic efficiency. To achieve our goals, UBCSAT has been designed according to what we have named a *triggered procedure architecture*. The main ideas underlying this architecture are closely related to certain concepts from object- and event-oriented programming. The UBCSAT software is structured around a set of *event points* that occur throughout the execution of a SLS algorithm for SAT, and a set of *triggers*, each of which associates a software procedure with an event point. For each event point p, a list of procedures is maintained that are executed whenever event point p is reached; this list is called the *trigger list* of p, and its elements are called the *triggered procedures* of p. Initially, all trigger lists are empty; when a trigger is *activated*, its associated procedure is added to the trigger list of the corresponding event point.

In addition to its associated procedure, event point, and activation status, a trigger t can have *dependency list* of other triggers that are automatically activated when t is activated. In UBCSAT, this is used, for example, to ensure that when using certain variable selection heuristics that rely on proper updating of special data structures after each flip, the triggers for the procedures that perform these updates are activated. Similarly, a trigger t can have a list of other triggers that are deactivated whenever t is activated, which allows overriding default routines and generally helps to avoid conflicts between incompatible routines. Finally, triggers can include precedence information that can be used to help properly sequence the execution of procedures with active triggers at the same event point in cases where the correct order of execution matters.

There is also a special type of trigger called a *container trigger*, which has no associated procedure, but instead a list of secondary triggers that are activated whenever the container trigger is activated. Container triggers are used as convenient shortcuts for activating groups of triggers that are used simultaneously.

UBCSAT has over one hundred triggers, most of which have associated procedures that fall into one of the following four categories: heuristic selection (*e.g.*, of variables), data structure maintenance, report and statistic data collection, and file I/O. Triggers are activated based on the SLS algorithm to be run, the reports/statistics requested, and other system command line parameters. In the UBCSAT implementation, the trigger lists are simply arrays of function pointers, so when each event point is reached it is very efficient to call its triggered procedures.

[1] The UBCSAT source code and x86 executables for Windows and Linux are available for download at `http://www.satlib.org/ubcsat`. Throughout this paper we have endeavoured to keep our descriptions and examples consistent with the UBCSAT software package version 1.0, but as development on UBCSAT continues, it may deviate from these descriptions in some respects.

```
procedure UBCSAT
    SetupUBCSAT();
    ParseParameters();
    ActivateAlgorithmTriggers();
    ActivateReportTriggers();
*   RunProcedures(PostParameters);
*   RunProcedures(ReadInInstance);
*   RunProcedures(CreateData);
*   RunProcedures(CreateStateInfo);
*   RunProcedures(PreStart);
    StartClock();
    while iRun < iNumRuns do
*       RunProcedures(PreRun);
        while ((iStep < iCutoff) and (not bSolutionFound)) and (not bTerminateRun)) do
*           RunProcedures(PreStep);
*           RunProcedures(CheckRestart);
            if bRestart or (iStep = 1) then
*               RunProcedures(InitData);
*               RunProcedures(InitStateInfo);
            else
*               RunProcedures(ChooseCandidate);
*               RunProcedures(PreFlip);
*               RunProcedures(FlipCandidate);
*               RunProcedures(PostFlip);
            end
*           RunProcedures(PostStep);
*           RunProcedures(CheckTerminate);
        end
*       RunProcedures(PostRun);
    end
    EndClock();
*   RunProcedures(Final);
end UBCSAT
```

Fig. 2. High-level pseudo-code of UBCSAT; event points are indicated by asterisks.

Figure 2 shows a high-level pseudo-code representation of UBCSAT and indicates many of its most important event points. The following example further illustrates the use of event points and the concept of triggered procedures.

Let us consider WalkSAT/TABU, a well-known high-performance SLS algorithm for SAT that is based on the WalkSAT architecture [12]. As in most WalkSAT-based algorithms, WalkSAT/TABU starts each search step by uniformly selecting a clause from the set of currently unsatisfied clauses. Each variable in the clause is assigned a score, corresponding to the change in the number of unsatisfied clauses that would occur if that variable were flipped. The variable with the best score that is not *tabu* is selected as the flip variable (breaking ties randomly). A variable is tabu if it has been flipped within the last *TabuTenure* search steps, where *TabuTenure* is a parameter of the WalkSAT/TABU algorithm. If all of the variables in the selected clause are tabu, then no flip occurs at that step.

In UBCSAT, the main heuristic procedure for WalkSAT/TABU is *PickWalksatTabu()*, and a trigger of the same name exists which maps the procedure to the *ChooseCandidate* event point. Most algorithms in UBCSAT also activate the *DefaultProcedures* trigger, a container trigger that includes triggers for handling common tasks, such as keeping track of the current truth assignment, and reading the formula into memory. Efficient implementations of WalkSAT-based algorithms require a list of the currently unsatisfied clauses, which is maintained by a set of procedures whose triggers are all included in the *FalseClauseList* container trigger.

Different from, say, WalkSAT/SKC, WalkSAT/TABU needs to know when each variable has been flipped last, in order to determine its tabu status. This requires a simple data structure (an array of values) that is maintained using three triggered procedures: *CreateVarLastChange()* allocates the memory required for the data structure, *InitVarLastChange()* initialises it at the beginning of each run and after restarts, and *UpdateVarLastChange()* updates it after each flip. Each of these procedures has a trigger that associates it with the event points *CreateStateInfo*, *InitStateInfo*, and *PostFlip*, respectively. For convenience, these three triggers are grouped into a container trigger named *VarLastChange*. When the *PickWalksatTabu* trigger is registered in UBCSAT, it lists *VarLastChange* in its dependency list, so when the Walksat/TABU algorithm is selected, the *PickWalksatTabu* trigger is activated, which will activate the *VarLastChange* trigger, and hence the three previously described triggers. (See also Figure 3.)

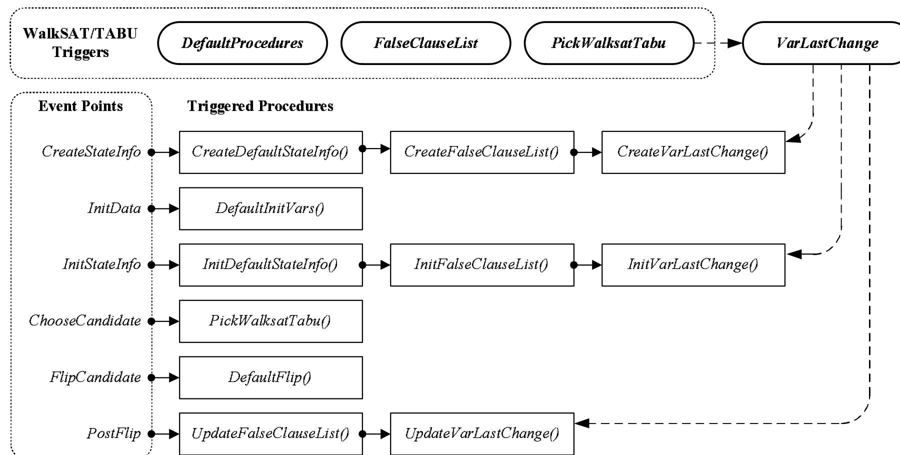

Fig. 3. The WalkSAT/TABU algorithm triggers, and the triggered procedures that appear in the event point trigger lists. The dashed arrows illustrate how the *VarLastChange* procedures were added to the trigger lists by the activation of the *PickWalksatTabu* trigger. Note that some procedures and event points are not listed, including a few additional procedures triggered by *DefaultProcedures*.

The primary advantage of the triggered procedure architecture is that of the many procedures that are needed to realise the many SLS algorithms and report formats supported by UBCSAT, only those required in any given run are activated and used, while the remaining, inactive or non-triggered procedures do not affect UBCSAT's performance. A secondary advantage is that different algorithms and reports can share the same data structures and procedures, saving much programming effort. Potential drawbacks stem from the implementation overhead of having to register all triggers, and from the fact that in this framework, algorithms are typically split into many rather small procedures, which can lead to decreased performance compared to more monolithic implementations. However, we have found that these disadvantages are far outweighed by the advantages of UBCSAT's triggered procedure architecture. In particular, as we will demonstrate in the following section, the performance of UBCSAT is very competitive with native reference implementations of the respective SAT algorithms.

3 A Collection of Efficient Algorithm Implementations

UBCSAT is not an SLS algorithm, but rather a collection of many different SLS algorithms. Compared to the respective reference native implementations of these algorithms, by integrating them into the UBCSAT framework several advantages can be realised: Generally, by using a single executable with a uniform interface, working with different algorithms becomes easier and more convenient. From an implementation point of view, different algorithms share common data structures and procedures, which reduces implementation effort and the likelihood of programming errors. And from an empirical algorithmics point of view, comparing two algorithms is facilitated by the fact that UBCSAT allows fairer comparisons between algorithms that share components, such as common operations, and use the same statistical calculations, input and output formats.

The UBCSAT software package currently implements the following SLS algorithms for SAT:

- GSAT [14]
- GWSAT [13]
- GSAT/TABU [11]
- HSAT [3]
- HWSAT [4]
- WalkSAT/SKC [13]
- WalkSAT/TABU [12]
- Novelty and R-Novelty [12]
- Novelty$^+$ and R-Novelty$^+$ [6]
- Adaptive Novelty$^+$ [7]
- SAPS and RSAPS [9]
- SAPS/NR [16]

UBCSAT is designed to support weighted MAX-SAT versions (see also Section 6) as well as variants, which may differ in their behaviour or implementation from the basic version of a given algorithm. Con-

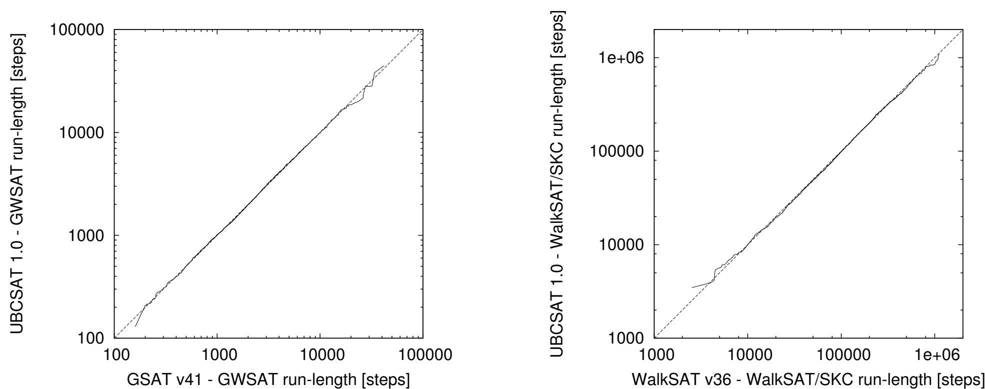

Fig. 4. Quantile-quantile plot of the run-time distributions for UBCSAT *vs.* GSAT v41 (left) and WalkSAT v43 (right) based on 5 000 runs per algorithm (run-time is measured in search steps). For GWSAT (left) the instance is `uf200-easy` and for WalkSAT/SKC (right) the instance is `bw_large.a`.

sequently, each algorithm within UBCSAT is identified as a triple (*"algorithm"*, $bWeighted$, *"variant"*), selectable on the command line as `ubcsat -alg` $algorithm$ `[-w] [-v` $variant$`]`.

For each of the previously listed algorithms, we ensured that the UBCSAT implementation behaves identically to the respective original reference implementation, taking into consideration the stochastic nature of the algorithms. This is illustrated in Figure 4, in which run-time distributions for the UBCSAT implementations of GWSAT and WalkSAT/SKC are compared with those for the original GSAT (version 41) and WalkSAT (version 43) implementations.

At the same time, the UBCSAT versions of all algorithms were optimised for efficiency, with the goal of matching or exceeding the time performance of the respective reference implementations. For many SLS algorithms the key to an efficient implementation lies in the way crucial data structures are organised and incrementally maintained. For example, many algorithms (*i.e.* GSAT-based algorithms) assign a score to each variable that is defined as the number of clauses made satisfiable minus the number of clauses made unsatisfiable if that variable is flipped. Rather than recomputing all variable scores in each step, they can be stored and incrementally updated, such that after each flip only the scores affected by that flip are recalculated [8]. However, we have found that in some situations too much time can be spent by using this scheme; in particular, using it in the implementation of WalkSAT algorithms actually decreases their performance. To further complicate matters, the optimal incremental update strategy often depends on the characteristics of the given problem instance.

In our UBCSAT implementation, we strove to use data structures and incremental updating schemes that are efficient, yet reasonably straightforward to understand and implement. The UBCSAT architecture supports functionally identical algorithm variants that are implemented using different data structures and/or incremental updating schemes in a straight-forward way, which makes it easy to implement new developments in this area (such as Fukunaga's recent scheme [2]).

The performance of the UBCSAT implementations of all supported algorithms has been tested against that of the respective reference implementations in order to ensure that the former are at least as efficient (in terms of run-time) as the latter. More importantly, for GSAT- and WalkSAT-algorithms, the UBCSAT implementations have been shown to be significantly faster (see Table 1 for representative results).

4 A Framework for Developing New Algorithms

As discussed in the previous section, the UBCSAT environment includes a wide variety of algorithms and data structures. To facilitate the development and integration of new SLS algorithms, UBCSAT has been designed in such a way that new algorithms can easily re-use the existing procedures and data structures from other algorithms; the basis for this is provided by the triggered procedure architecture discussed in Section 2.

To illustrate how new algorithms are added to UBCSAT, in Figure 5 we present the pseudo-code required to add a new WalkSAT/TABU algorithm variant to UBCSAT. We have named the new variant

Algorithm	uuf100-01			uuf400-01			jnh202			rg-200-2000-4-11		
	UBCSAT	Original	*s.f.*	UBCSAT	Original	*s.f.*	UBCSAT	Original	*s.f.*	UBCSAT	Original	*s.f.*
WalkSAT/SKC	97.7	144.7	**1.48**	98.5	150.3	**1.53**	134.0	217.2	**1.62**	142.1	310.7	**2.19**
Novelty	117.1	151.6	**1.29**	114.5	153.4	**1.34**	168.4	230.8	**1.37**	159.5	323.0	**2.02**
GSAT	106.7	305.0	**2.86**	114.1	316.5	**2.77**	202.3	1541.6	**7.62**	233.0	397.8	**1.71**
GWSAT	172.1	590.1	**3.43**	266.8	768.2	**2.88**	254.3	1894.7	**7.45**	541.5	1354.5	**2.50**

Table 1. Total run times (in seconds) on 1 GHz Linux machine for 100 000 000 search steps. The speedup factor *(s.f.)* shows the software speedups of the UBCSAT implementation over the original implementations (GSAT v41 and WalkSAT v43). Note by choosing *unsatisfiable* instances for this speed comparison, we ensured that in all cases exactly the same number of search steps have been performed. The `uuf-*` instances are uniform random 3-SAT, the `jnh` instance is random P-SAT, and the `rg` instance is a structured encoding of a graph colouring instance.

```
procedure AddWalksatTabuNoNull()
    CreateAlgorithm("walksat-tabu", false, "nonull",        % algorithm, bWeighted, variant
        "WalkSAT/TABU without null flips",                  % description
        "McAllester, Selman, Kautz [AAAI 97] (modified)",   % authors
        "PickWalksatTabuNoNull",                            % heuristic trigger(s)
        ...);                                               % details omitted
    InheritDataTriggers("walksat-tabu", false, "");
    InheritParameters("walksat-tabu", false, "");
    CreateTrigger("PickWalksatTabuNoNull",                  % trigger name
        ChooseCandidate,                                    % event point
        PickWalksatTabuNoNull,                              % pointer to procedure
        ...);
end AddWalksatTabuNoNull

procedure PickWalksatTabuNoNull()
    PickWalksatTabu();
    if iFlipCandidate = NULL then
        iFlipCandidate := PickRandomVarFromClause(iWalkSATClause);
    end
end PickWalksatTabuNoNull
```

Fig. 5. Pseudo-code for adding a new WalkSAT/TABU algorithm variant to UBCSAT.

WalkSAT/TABU-NoNull, and it differs from the regular WalkSAT/TABU algorithm in only one detail: if all of the variables in the selected clause are tabu, then a variable will be selected from the clause at random and flipped. (This variant is interesting from a practical point of view, since WalkSAT/TABU is one of the best-performing WalkSAT algorithms, but often suffers from search stagnation as a consequence of null-flips.)

Within UBCSAT, the new algorithm will be identified as (*"walksat-tabu"*, $false$, *"nonull"*); it differs from the already supported WalkSAT/TABU only in its variable selection procedure, whose trigger we name *PickWalksatTabuNoNull*. An algorithm can explicitly specify the data structure procedures required, or it can *inherit* them from another algorithm. In this case, we will simply inherit everything from regular WalkSAT/TABU (*"walksat-tabu"*, $false$, *""*). When an algorithm requires algorithm-specific command-line parameters (such as the *tabuTenure* parameter in WalkSAT/TABU) they must be defined or optionally inherited from an existing algorithm. In addition to creating and registering the new trigger in the system, its associated procedure, here also called *PickWalksatTabuNoNull*, has to be implemented, which in this example simply calls the regular WalkSAT/TABU variable selection procedure and then handles the special case when a null-flip occurs. While this example illustrates a particularly straight-forward variant of an existing algorithm, the process of adding implementations of new SLS algorithms to UBCSAT is typically similarly straightforward.

5 An Empirical Analysis Tool

Empirical analysis plays an important role in the development and successful application of SAT algorithms. To characterise or measure the behaviour of an SLS algorithm, typically data needs to be collected from multiple independent *runs* of the algorithm. Each run corresponds to a complete execution of the algorithm outline in Figure 1; note that the pseudo-code of Figure 2 contains code to perform multiple runs. (Note that when restart mechanisms are used, a single run can be punctuated by one or more restarts, but this does

```
integer iCurVarAge;                                          % global variable for statistic

procedure AddAgeStat()
   AddColumn("agemean",                                      % column name
      "Mean Age of Variables when flipped",
      &iCurVarAge,                                           % pointer to data variable
      "UpdateCurVarAge",                                     % trigger to activate
      TypeMean,                                              % type of statistic to collect on data
      ...);
   CreateTrigger("UpdateCurVarAge", PreFlip, UpdateCurVarAge,
      "VarLastChange",                                       % trigger dependency
      ...);
end AddAgeStat

procedure UpdateCurVarAge()
   iCurVarAge := iStep - aVarLastFlip[iFlipCandidate];
end UpdateCurVarAge
```

Fig. 6. Pseudo-code for adding a new statistic that measures the mean age of variables when flipped.

not partition it into multiple runs.) As an example, consider the run-time data shown in Figure 4, which is based on $5\,000$ independent runs of each algorithm involved in the respective experiment. To facilitate the advanced empirical analysis of the SLS algorithms it implements, UBCSAT not only provides support for measuring and reporting basic descriptive statistics over multiple runs, but also strongly supports the analysis of run-time distributions (RTDs) [8]. In particular, UBCSAT can measure and report RTDs in a format that can be easily plotted (see left side of Figure 7) or further analysed with specialised statistical software.

Additional reports and statistics can be added to UBCSAT in a manner that is as straightforward as, and conceptually closely related to the way in which new algorithms are added. Reports can be in any format, and are implemented based on a list of triggered procedures that collect and print the required information. Reports currently implemented in UBCSAT include the satisfying assignments found in each run, detailed information about the search state at each search step, flip statistics for individual variables, and many others. In UBCSAT, statistics are special objects that are used to collect and summarise data for the default reports. Statistics can be shown for each individual run (column objects), or be summarised over all runs (stat objects).

In Figure 6, we show the creation of a column object that will calculate the average *age* of variables flipped during a run. The age of a flipped variable is calculated as the number of steps that have occurred since the last time the variable was flipped (the calculation is shown in *UpdateCurVarAge()*. For this statistic, the trigger *UpdateCurVarAge* is required to ensure that the correct age value is calculated at the event point *PreFlip*. The trigger *UpdateCurVarAge* depends on the trigger *VarLastChange* (see Figure 3), so if the algorithm already collects this data (*e.g.*, WalkSAT/TABU) then the statistic will simply share the same data structure, but if the algorithm does not normally require this data, then the trigger will ensure that it is collected. Because this column statistic has been identified as a *TypeMean* (average over all search steps of a run), an additional trigger will be automatically activated to collect the data at the end of each search step. Like many statistics added to UBCSAT, this age statistic is now available to *all* algorithms (that use a single-flip strategy); comparisons between algorithms on statistics such as these help further our understanding of how SLS algorithms behave.

6 Support for MAX-SAT

One area where SLS algorithms have been very successful, and have defined the state-of-the-art for more than a decade, is in solving the MAX-SAT problem, and in particular, the *weighted* MAX-SAT problem, which is why supporting MAX-SAT was one of our primary goals. Although there are interesting differences between the state-of-the-art SLS algorithms for SAT and MAX-SAT, at the conceptual and implementation level, there are many similarities. Unweighted MAX-SAT can be seen as a special case of weighted MAX-SAT where all clauses have weight one; therefore, in the following, we will mostly focus on the weighted MAX-SAT problem. It should be noted, however, that in terms of implementation, SLS algorithms for unweighted MAX-SAT are much more closely related to SLS algorithms for SAT. In UBCSAT, unweighted MAX-SAT algorithms are therefore typically equivalent to the corresponding SAT algorithm,

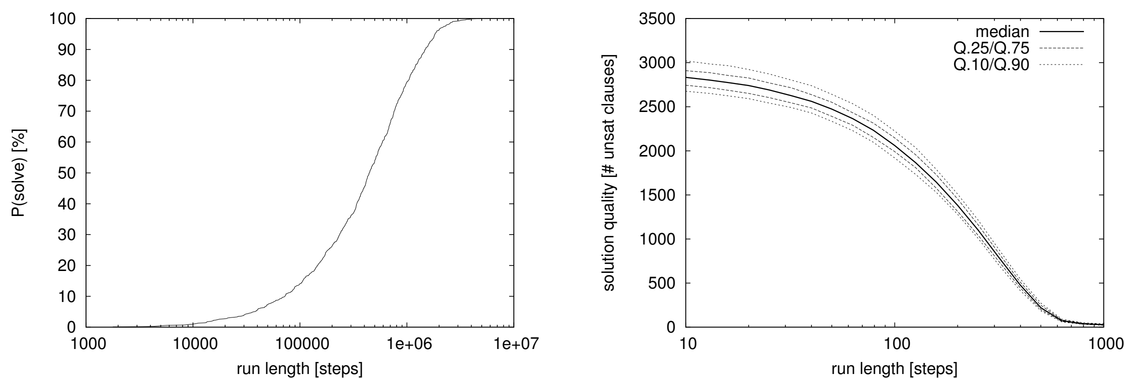

Fig. 7. An example of a run-length distribution (RLD) (left), and a time-dependent solution quality statistics (SQT) plot (right). The data underlying these curves can be easily generated by the UBCSAT software package and plotted using gnuplot scripts which are available on the UBCSAT website.

while weighted MAX-SAT algorithms are implemented separately, which facilitates conceptually simpler and highly efficient implementations for both cases.

The main differences between SAT and MAX-SAT is that the optimal solution quality (*i.e.*, maximal total weight) for a given problem instance is often unknown. Hence, the best assignment encountered during the search, the so-called *incumbent assignment*, is memorised and returned at the end of the search. This memorisation of the incumbent assignment is accomplished in UBCSAT via a report. Typically, SLS algorithms for MAX-SAT are not guaranteed to find *optimal solutions*, *i.e.*, maximal weight assignments, but many state-of-the-art SLS algorithms for MAX-SAT have the property that if they search long enough, the probability of finding an optimal solutions approaches one (the so-called *PAC property*, see also [6, 8]), and in many practical cases assignments that are provably optimal or believed to be optimal can be found within reasonable run-times. UBCSAT supports termination criteria that end a run whenever a user-specified solution quality (for example, the known optimal solution quality for the given problem instance) is reached or exceeded; alternatively, particularly when dealing with instances whose optimal solution quality is unknown, UBCSAT can be configured with advanced criteria to determine when to terminate a run.

Currently, UBCSAT includes implementations of two dedicated algorithms for MAX-SAT, SAMD [5] and IRoTS [15], as well as weighted MAX-SAT variants for many of the SLS algorithms listed in Section 3. The mechanism for implementing new MAX-SAT algorithms within UBCSAT is exactly the same as described for the case of SAT in Section 4. While for unweighted MAX-SAT instances, the same DIMACS CNF file format as for SAT is used, for weighted MAX-SAT instances, UBCSAT currently supports a straight-forward extension of the this format known as the *weighted CNF file format* `(.wcnf)`. To support the empirical analysis of the behaviour and performance of SLS algorithms for MAX-SAT, besides the previously mentioned statistics and reports (see Section 5), UBCSAT supports advanced analysis methods for stochastic optimisation algorithms. In particular, the following types of empirical performance characteristics can be easily measured (see also [8]):

– qualified run-time distributions (QRTDs), *i.e.*, empirical probability distributions of the run-time required for reaching or exceeding a specific target solution quality measured over multiple runs of the algorithm;
– solution quality distributions (SQDs), *i.e.*, empirical probability distributions of the best solution quality reached within a given amount of run-time, measured in terms of search steps or CPU time over multiple runs of the algorithm;
– time-dependent solution quality statistics (SQTs), *i.e.*, the development of descriptive statistics (such as quantiles) of the SQDs as run-time increases.

QRTDs, SQDs, and SQTs are determined from so-called *solution quality traces*, which contain information on every point in time the incumbent solution was updated during a given run of the algorithm. The solution quality traces are collected by UBSAT with minimal overhead during the run of any MAX-SAT algorithm. Figure 7 (right) shows a sample SQT measured by UBCSAT.

7 Conclusions and Future Work

In this paper we have introduced UBCSAT, a new software environment that we created with the specific goal of facilitating and supporting research on SLS algorithms for SAT and MAX-SAT. UBCSAT is built on the basis of a novel triggered procedures architecture and includes highly efficient, conceptually simple, and accurate implementations of a wide range of prominent SLS algorithms for SAT and MAX-SAT. UBCSAT facilitates the development and integration of new algorithms (and algorithm variants). It provides support for advanced empirical analysis of the performance and behaviour of SLS algorithms without compromising implementation efficiency. UBCSAT has been implemented in a platform-independent way and is publicly available to the academic community as an open-source software package.

While this paper has summarised the work on the UBCSAT project to date, UBCSAT is an ongoing effort, and we are very enthusiastic about expanding and building upon the project in the future. We plan to expand UBCSAT by including existing and new SLS algorithms for SAT and MAX-SAT. While we have so far focussed on an ANSI C compliant implementation, there is some interest in adding C++ interfaces, as well as extending our implementation beyond the 32-bit boundary for counters. We will continue to add more sophisticated reports and empirical analysis tools, and we are also interested in providing more external support features, such as gnuplot scripts and better integration with the R statistical software package. We are very interested in adding features that will make the software more accessible and useful to the research community, and welcome feedback and suggestions for further improvements.

But above all else, we hope that our UBCSAT framework will help advance state-of-the-art research in SLS algorithms, to help better understand how and why SLS algorithms behave the way they do, and to unlock some of the unexplored potential of SLS algorithms for SAT and MAX-SAT.

References

1. G. Audemard, D. Le Berre, O. Roussel, I. Lynce, and J. Marques-Silva. OpenSAT: an open source SAT software project. In *Sixth Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT2003)*, pages 502–509, 2003.
2. A. Fukunaga. Efficient implementations of SAT local search. In *Seventh Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT2004)*, 2004 (this volume).
3. I. P. Gent and T. Walsh. Towards an understanding of hillclimbing procedures for SAT. In *Proc. of the Eleventh Nat'l Conf. on Artificial Intelligence (AAAI-93)*, pages 28–33, 1993.
4. I. P. Gent and T. Walsh. Unsatisfied variables in local search. In *Hybrid Problems, Hybrid Solutions*, pages 73–85, 1995.
5. P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.
6. H. H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proc. of the Sixteenth Nat'l Conf. on Artificial Intelligence (AAAI-99)*, pages 661–666, Orlando, Florida, 1999.
7. H. H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proc. of the 18th Nat'l Conf. in Artificial Intelligence (AAAI-02)*, pages 655–660, 2002.
8. H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufman, 2004 (forthcoming).
9. F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *LNCS 2470: Proc. of the Eighth Int'l Conf. on Principles and Practice of Constraint Programming (CP-02)*, pages 233–248, 2002.
10. M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. on Modeling & Computer Simulation*, 8(1):3–30, 1998.
11. B. Mazure, L. Saïs, and É. Grégoire. Tabu search for SAT. In *Proc. of the Fourteenth Nat'l Conf. on Artificial Intelligence (AAAI-97)*, pages 281–285, 1997.
12. D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proc. of the Fourteenth Nat'l Conf. on Artificial Intelligence (AAAI-97)*, pages 321–326, 1997.
13. B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. of the 12th Nat'l Conf. on Artificial Intelligence (AAAI-94)*, pages 337–343, 1994.
14. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proc. of the Tenth Nat'l Conf. on Artificial Intelligence (AAAI-92)*, pages 459–465, 1992.
15. K. Smyth, H. H. Hoos, and T. Stützle. Iterated robust tabu search for MAX-SAT. In *Proc. of the 16th Conf. of the Canadian Society for Computational Studies of Intelligence*, pages 129–144, 2003.
16. D. A. D. Tompkins and H. H. Hoos. Warped landscapes and random acts of SAT solving. In *Proc. of the Eighth Int'l Symposium on Artificial Intelligence and Mathematics (ISAIM-04)*, 2004.
17. P. Van Hentenryck and L. Michel. Control abstractions for local search. In *LNCS 2833: Proc. of the Ninth Int'l Conf. on Principles and Practice of Constraint Programming (CP-03)*, pages 65–80, 2003.