SBSAT User Manual and Quick Start Guide

Michal Kouril, Sean Weaver

January 7, 2007

SBSAT Version 2.5b-5, January 2007

Welcome to SBSAT:
a State Based Satisfiability Solver

SBSAT is a software package used primarily for solving instances of a generalization of the well-known Satisfiability problem. In particular, the problem solved by SBSAT is the following:

GIVEN: Input variable set $V = \{v_1, ..., v_n\}$ of Boolean variables, set of Boolean functions $B = \{f_1, ..., f_m\}$ where, for all i, f_i maps an assignment of values to variables of V to $\{T, F\}$.

RESULT: An assignment of values to variables of V such that, for all i, $f_i = T$, or *unsatisfiable* if no such assignment is possible.

If, for all i, f_i is a function corresponding to the conjunction of a subset of variables of V, then the problem is reduced to the well-studied Boolean Satisfiability Problem. If the variables of V are allowed to take arbitrarily many values, then the problem becomes the well-studied Constraint Satisfaction Problem.

The functions B may be specified in several different ways. But, there is one canonical input specification format, which we call the *canonical form*: a conjunction of a collection of BDDs[1]. Any recognized user input is translated to the canonical form, if it is not in that form already. Of course, the user is free to supply his/her own translation to BDDs which then may be input: in this way all possible input formats can be accommodated. Specific, supported input formats are:

- CNF (Conjunctive Normal Form - described in Sections 4.2 and 9.2)
- DNF (Disjunctive Normal Form - described in Section 9.3)
- BDD (SBSAT canonical form - described in Sections 4.3,9.1)
- SMURF (described in Section 9.7)
- Trace (From CMU benchmark examples - described in Section 9.5)
- Prove (Generated by CMU tool BMC - described in Section **??**)
- XOR (Each conjoint is an XOR of conjoints - see Sections 4.4,9.4)

Examples of how a user might develop a custom translation to the canonical form from other formats are found in Section 4.8.

[1] See Page 4 for the definition of BDDs and Section 10.1 for a description.

For maximum effectiveness, the user should be aware of and know how to control the three phases of SBSAT execution, shown schematically in Figure 1. In the first phase an input is read from an input file. The user must decide which input format to use and build the input file accordingly. There are three issues here: namely choosing the type of input format, writing the input in a way that can be exploited by elements of the remaining phases, and keeping the syntax correct. Format types and syntax are described in Sections 4 and 9. Comments on writing exploitable input may be found in Section 18. In the second phase various levels of preprocessing are applied to the input instance with the intention of producing an internal set of constraints (in canonical form) that are either logically equivalent [2] or equi-satisfiable [3] to the original and yields a smaller search space through advanced and intelligent search heuristics and learning. The user may control this phase using command line switches when launching the program. Details of the kinds of preprocessing available and their effects are found in Sections 5.1 and 10 along with examples of their use. In the third phase the internal form (that is, set of constraints in canonical form) is searched for a solution. The user must choose one of the ways to perform a search and the search heuristic which is used to select unassigned variables to be assigned values. Future versions will allow the user to define a search heuristic and coordinating preprocessing elements. Choices for searching are:

- SMURF (Default backtracking solver - Section 12.1)
- BDD WalkSAT (an incomplete solver - Section 12.2)
- WVF (Vanfleet's tinkering solver - Section 12.3)
- Simple (A stripped-down version of the SMURF solver - Section **??**)

Reasons for choosing one of the above are given in Sections 12.1-**??**. Search heuristics are used to help control the size of the search space. In the current version the user may choose one of the following to control the SMURF solver:

- VSIDS (Section 11.3)
- Locally Skewed, Globally Balanced (Johnson generalization - Section 11.2)
- Combination of the above two

The user may also choose one of the following to control the BDD WalkSAT solver:

[2] See Page 5 for the definition of logically equivalent.
[3] See Page 4 for the definition of equi-satisfiable.

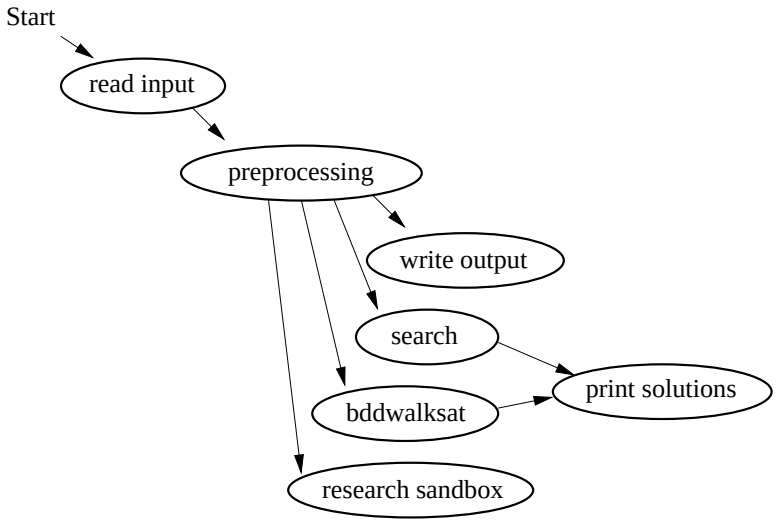


Figure 1: Schematic depiction of controllable execution paths of SBSAT.

- Adaptive Novelty+ (Section **??**)
- Novelty+ (Section **??**)
- Random (Section **??**)

The size of the search space can be further controlled through learning. As backtracks occur, new constraints, called Lemmas (described in Section 12.1), also referred to as conflict clauses, or learned clauses, are added to the internal constraint set. These can prevent some fruitless backtracking later in the search. However, there is some overhead incurred by Lemmas. Hence it is important to choose carefully which Lemmas are to be saved, how many Lemmas can be saved at a maximum, and which Lemmas to discard when the maximum is exceeded. These choices are controlled by switches on the command line and are described in Section 7. The results of operations initiated by those switches are explained in Section 12.1.

The solver was successfully tested and compiled on a number of Unix based platforms such as Linux, DEC, Solaris, Mac OS X, Windows/Cygwin with a number of different compilers such as gcc2.95, gcc3.x, solaris-cc, dec-cc, pgcc.

Contents

1 About the Manual **1**

2 Conventions and Definitions **2**

2.1 Conventions 2

2.2 Definitions 3

3 Quick Start - Getting `sbsat` ready to run **7**

3.1 Hardware Requirements 7

3.2 Getting SBSAT 7

3.3 Installing SBSAT 8

3.4 Compiling SBSAT 9

3.4.1 Configure options 10

3.5 Testing SBSAT 11

4 Quick Start - The basics of running SBSAT **12**

4.1 Command line 12

4.2 A CNF formula as input 12

4.3 An input in canonical (BDD) form 15

4.4 An input in XOR format 17

4.5 Reusing functions (Macros) 18

4.6 Printing functions 18

4.7 An example of a complex canonical form input 21

4.8 Translating an expression to canonical form 21

4.8.1 Interconnect synthesis in reconfigurable computing 21

4.8.2 Bounded Model Checking 24

4.9 Choosing a search procedure 29

4.10 Converting the input file 29

5 Quick Start - Some advanced features of `sbsat` **31**

5.1 Preprocessing 31

5.1.1 Preprocessor options 31

5.1.2 Preprocessor sequence 31

5.1.3 Preprocessor command 33

5.2 Heuristics 34

5.3 The lemma cache 34

5.4 Controlling preprocessing and search time 35

5.5 Creating and using an initialization file 36
5.6 Debugging 37

6 Quick Start - Getting more help quickly 38

7 Reference - Command line 39
7.1 General options 41
7.2 BDD table options 41
7.3 Input options 42
7.4 Output options 42
7.5 Preprocessing options 43
7.6 General solver options 44
7.7 SMURF Solver options 45
7.8 BDD WalkSAT solver options 46

8 Reference - Initialization file 47

9 Reference - Input formats 48
9.1 Canonical form 49
9.1.1 Comments 50
9.1.2 File Header 50
9.1.3 Boolean functions 50
9.1.4 Boolean function extensions 53
9.1.5 Manipulators 54
9.1.6 Directives 56
9.2 CNF format 61
9.2.1 Comments 61
9.2.2 File Header 61
9.2.3 Boolean functions 62
9.3 DNF format 62
9.3.1 Comments 62
9.3.2 File Header 62
9.3.3 Boolean functions 63
9.4 XOR format 63
9.4.1 Comments 63
9.4.2 File Header 63
9.4.3 Boolean functions 64
9.5 Trace format 65

9.5.1 Comments . . . 65
9.5.2 File Header . . . 65
9.5.3 Boolean functions . . . 66
9.6 Prove format . . . 66
9.7 SMURF format . . . 66
9.7.1 File Header . . . 67
9.7.2 Boolean functions . . . 67

10 Reference - Preprocessing **70**
10.1 Binary Decision Diagrams (BDDs) . . . 70
10.2 Pattern Matching: CNF . . . 72
10.3 Generalized Cofactor (GCF) . . . 73
10.4 Branch Pruning . . . 76
10.5 Strengthening . . . 79
10.6 Inferences . . . 80
10.7 Existential quantification . . . 81
10.8 Clustering + Existential Quantification . . . 83
10.9 Clustering + Existential Quantification + Safe . . . 85
10.10 Dependent variable clustering . . . 85
10.11 Rewind . . . 85
10.12 Splitter . . . 86
10.13 Universe . . . 87

11 Reference - Search heuristics **88**
11.1 State Machines Used to Represent Functions (SMURFs) . . . 88
11.2 Locally Skewed, Globally Balanced . . . 90
11.3 Chaff-like . . . 91
11.4 User defined search heuristic . . . 91

12 Reference - Search methods **92**
12.1 Backtracking and Lemmas . . . 92
12.1.1 Lemma cache . . . 92
12.1.2 Lemma effectiveness . . . 92
12.2 BDD WalkSAT . . . 92
12.3 WVF . . . 92

13 Reference - Output, results **93**
13.1 Raw 93
13.2 Fancy 93

14 Reference - Data structures **95**
14.1 BDD database 95
14.2 SMURF 95
14.3 Lemma database 95

15 Reference - Results: making BDDs from `bmc` **96**

16 Reference - Results: Experiments **99**

17 Reference - Debugging **106**
17.1 Converting to another format 106
17.2 Printing internal forms 106

18 Reference - Writing Exploitable Input **107**

1 About the Manual

The manual begins with sections describing conventions and definitions. The remainder of the manual has two parts: Sections 3 to 6 are written to get the novice acquainted with the use of `SBSAT` quickly; the following sections, beginning with Section 7, provide details needed for an accomplished user to fine-tune the use of `SBSAT`.

2 Conventions and Definitions

From now on we use SBSAT to refer to the package and `sbsat` to refer to the executable that is run to solve problems of the type stated at the beginning of the welcome section on Page i.

2.1 Conventions

When describing command line or file line syntax the following conventions apply. Items of important types are signified by enclosing the item in angle brackets. For example,

```
<var>
```

is an item of type `<var>`. Presumably the types used are defined in the text in close proximity to the first place they occur. The unterminated ellipsis (`...`) is used to indicate that arbitrarily many items of the type preceeding the ellipsis are possible after it. For example,

```
<var> <var> ...
```

means at least two items of type `<var>`, separated by blanks. and

```
<var><var>...
```

means at least two items of type `<var>`, not separated by blanks. A terminated ellipsis is used to indicate a list of finite size (one or more elements). For example,

```
var_1 ... var_n
```

means a list containing `n` items, $n \geq 1$ (the type of the items is described in the surrounding text). An optional flag or switch will be signified by enclosing it in square brackets. For example:

```
[-]<var> ...
```

means at least one `<var>` item may or may not be preceeded by the character '`-`'. The vertical bar ('`|`') separating items between square brackets ('`[`', '`]`') indicates a choice. For example:

```
[a|b|c]
```

means either `a` or `b` or `c`.

Various segments of an `sbsat` session will be highlighted using font changes to assist the reader in understanding the nature of command segments and results. Input and output will be specified using the typewriter font. For example, these segments appear like this

```
Reading file ...
```

The $ character at the beginning of a line is the *command line prompt* and indicates that what follows is a command to be executed. The prompt is usually followed by an `sbsat` command. For example, the following is a simple `sbsat` command:

```
$ sbsat file.cnf
```

Programming options appear in italics to contrast with option parameters which appear in plain text. For example, to get command line help use this command:

`$ sbsat` --*help*

An input file has keywords in boldface such as in the following:

and ($1, 2)

The `$` of the previous line is `not` the command line prompt: its use in that context will be explained in Section 9.1.

Boolean Quantifiers and operators shall be written in the usual manner. Thus,

$\forall x$	means	For all values of x
$\exists x$	means	There exists a value for x such that
$\neg$	means	negation or complementation
$\vee$	means	logical "or"
$\wedge$	means	logical "and"
$\Rightarrow$	means	logical "implies"
$\oplus$	means	logical "exclusive-or"
$\equiv$	means	equivalent
$\Leftrightarrow$	means	"if and only if"

2.2 Definitions

- **Backjumping** - Advancement of the search by skipping over some choice points that cannot possibly lead to a solution.

- **BDD** - A Binary Decision Diagram is a DAG-representation of a Boolean function expressed using only the operator `if-then-else`, plus constants T and F, Boolean variables, and parentheses. BDD representations are usually far more compact than truth table representations. The form of BDDs we used are reduced and ordered as these are canonical representations of functions.

- **Boolean Function** - A Boolean function has one or more variable or Boolean function arguments and may or may not return a Boolean value depending on values assigned to or returned from its arguments. Any Boolean function can be expressed in terms of a nesting of Boolean functions as BDDs. This fact is used to express arbitrary Boolean functions in our canonical form (see Section 9.1).

- **Boolean Variable** - A variable may or may not be assigned a value: if it is assigned a value that value is one of the atoms in the set $\{T, F\}$, where T and F may be thought of as corresponding to *true* and *false*, respectively. In this document we alternatively and interchangeably use the set $\{1, 0\}$ for $\{T, F\}$ since so much of the literature uses that notation. Hereafter, when we say variable we mean Boolean variable.

- **Choice point** - The point in a search where an uninferred variable is given a value decided upon by some heuristic.

- **Clause** - A disjunction ($\vee$) of literals. For example, $(x_{35} \vee \neg x_{42} \vee x_{12})$.

- **CNF** - Conjunctive Normal Form. A conjunction ($\wedge$) of clauses. This is an important form for Boolean expressions since there exists an efficient translation to a logically equivalent CNF expression from any Boolean expression.

- **DIMACS CNF** - Standard format accepted by all CNF SAT solvers. For a complete specification of this format see

  `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi`

  Skeletal descriptions are found in Sections 4.2 and 9.2.

- **Equi-satisfiable** - A scheme for translating one Boolean function to another such that the target function is satisfiable if and only if the source function is satisfiable is said to produce an equi-satisfiable target function.

- **Inference** - An assignment of a value to a variable that is forced due to the constraints of the given expression.

- **Lemma** - A Boolean expression inferred (*i.e.*, learned) during the search. SBSAT learns lemmas by analyzing why some branch of the search tree failed to find a solution. SBSAT's lemmas are *clauses*. A solver, such as SBSAT, that learns lemmas can often use previously learned lemmas to avoid researching the same failed variable assignments.

- **Literal** - A variable or its negation. For example, x_{35} or $\neg x_{35}$. If the variable x_{35} is assigned the value T then the value of literal x_{35} is T and the value of $\neg x_{35}$ is F.

- **Logically equivalent** - A scheme for translating one Boolean function to another such that the target function evaluates to the same truth value as the source function in every model is said to produce a logically equivalent target function.

- **Preprocessing** - Operations applied to an `sbsat` input expression *before* search commences. Many such operations are possible and running one operation may affect the result of others. A list of all preprocessing options and descriptions of their operation is given in Section 10.

- **Satisfiable** - A Boolean function is *satisfiable* if and only if there exists an assignment of values to its variables which causes it to evaluate to *1*. A section of the output generated by `sbsat` says whether the input expression is satisfiable. For example, see the next to last line of Figure 4 below.

- **Solution** - An assignment of values to variables of a Boolean function which causes it to evaluate to *1*. A section of the output generated by `sbsat` provides a solution, if one exists and if the proper command line switches are set. For example, see the last lines of Figure 5. A solution, as presented by `sbsat`, is a list of variable names and each that is preceeded by a '`-`' is assigned value F and all others are assigned value T.

- **Standard input** - An input stream from the console to a running executable, for example `sbsat`. Input may be redirected in Unix or Windows using the `<` character before the entity containing desired input, usually a file.

- **Standard output** - An output stream to the console from a running executable, for example `sbsat`. Output may be redirected in Unix or Windows using the `>` character before the entity which is to receive the stream, usually a file.

- **Switch** - an `sbsat` option given by the user on the command line. Switches are always preceeded either by a dash (`-`) or a double dash (`--`). All switches understood by `sbsat` are listed and described briefly in Section 7.

- **Truth Table** - The truth table for a particular Boolean function is a listing of all possible assignments of values to the variables of the function; and next to each assignment is the value the function takes under that assignment.

- **Unary, Binary, and Ternary Boolean Functions** (`not`, `and`, `nand`, `or`, `nor`, `equ`, `xor`, `imp`, `nimp`, `ite`) - A Boolean function of two variables. There are $2^{2^2} = 16$ different binary Boolean functions and 2 unary functions. Names associated with a subset of these that include only non-trivial functions are given in the following table where, for binary functions, the bits of the 1-0 strings correspond to function values given input values of `00`, `01`, `10`, and `11`, respectively, from left to right, and for the unary function the two bit strings correspond to input values of `0` and `1`, respectively, from left to right. An important ternary function is `if-then-else` which we call `ite`. Its functionality is also expressed in the table with the obvious correspondence between input values and function values.

Binary				**Unary**	
`and`	`0001`	`nand`	`1110`	`not`	`10`
`or`	`0111`	`nor`	`1000`		
`equ`	`1001`	`xor`	`0110`	**Ternary**	
`imp`	`1101`	`nimp`	`0010`	`ite`	`01010011`

- **Unsatisfiable** - A Boolean function is *unsatisfiable* if and only if it is not satisfiable. A section of the output generated by `sbsat` will say whether the input expression is unsatisfiable.

3 Quick Start - Getting sbsat ready to run

This and the following three sections are intended to provide enough information to begin using `sbsat` successfully, if not optimally.

3.1 Hardware Requirements

Currently, `sbsat` requires a Unix style operating system with a c++ compiler, preferably, but not necessarily, the GNU g++ compiler. All examples require at least 32MB of RAM beyond the requirements of the operating system. Disk requirements depend on the operating system but at least 50MB of free space is required.

By default, during execution, `sbsat` is allocated as much RAM as it needs, if available. The amount of memory requested by `sbsat` can be limited only indirectly by changing, for example, the number of lemmas it maintains in the cache or the size of the pools for different stacks[4]. There is no other option to limit the amount of memory it is allocated. Experiments confirm that the amount of memory requested linearly follows the size of the problem being solved. `sbsat` is not multi-threaded and does not take advantage of multiple processors.

3.2 Getting SBSAT

SBSAT is available for download from the following website:

```
http://www.cs.uc.edu/~weaversa/SBSAT.html
```

SBSAT may also be obtained by email request to `weaversa@gmail.com`, `mkouril@ececs.uc.edu`, or `franco@gauss.ececs.uc.edu`. The distribution comes in two forms: a single CDROM and a tarball named `sbsat-latest.tar.gz`. Those authorized to login to `boole.ececs.uc.edu` may use scp to download the tarball from directory /home/mkouril. The command to do this in unix (from your local host) is:

```
$ scp boole.ececs.uc.edu:/home/mkouril/sbsat-latest.tar.gz .
```

From a PC running Windows login to `boole` using TeraTerm Pro and transfer

[4] `sbsat` is allocated a new pool of the same size if and when it exhausts the current one.

the file using zmodem. On boole, There is also a CVS repository containing SBSAT. To check out the latest CVS sources, execute the following command in unix (from your local host):

```
$ cvs -d boole.ececs.uc.edu:/home/mkouril/CVS/ co sbsat
```

3.3 Installing SBSAT

These instructions are only for installing SBSAT on computers running unix. Instructions for Windows machines will be supplied in a future release.

Become `root` (This step may not be necessary). This entails knowing the superuser password. At the command line prompt, issue the command `su` and enter the superuser password when requested to do so.

If you have the CDROM, insert it into the CDROM drive and mount that drive, usually on `/mnt/cdrom`, using the following:

```
$ mount /dev/cdrom /mnt/cdrom
```

If this command fails, find a suitable mount point in place of `/mnt/cdrom` or find the correct `/dev` for the CDROM (for example, `/dev/scd0`) or both. If this still fails, consult a system administrator. The following assumes the above command succeeded. Change directory to the place where SBSAT is to be installed (for example `/usr/local`), make a directory called `sbsat`, change to that directory, and copy the contents of the CDROM to the current directory using the following commands:

```
$ cd /usr/local
$ mkdir sbsat
$ cd sbsat
$ cp -r /mnt/cdrom/* .
```

where the '`.`' is part of the command and means current directory. Use the `umount` command to unmount the CDROM as follows:

```
$ umount /mnt/cdrom
```

If you are installing the tarball, move it to the directory in which SBSAT is to reside. For example, if the target directory is `/usr/local` and `sbsat-latest.tar.gz` exists in the home directory of a user named `franco` then issue

the command

```
$ mv ~franco/sbsat-latest.tar.gz /usr/local
```

Unzip and unarchive the tarball using the following commands

```
$ cd /usr/local
$ tar -xvzf sbsat-latest.tar.gz
```

You may remove the tarball, if you wish, with

```
$ rm sbsat-latest.tar
```

The result of the above commands is that all files of the SBSAT package are in a directory such as

```
/usr/local/sbsat-<version>-<revision>
```

where `<version>` and `<revision>` are the version and revision you have installed: for example, on January 5, 2007 the version is `2.5b` and the revision is `5` so in this case the directory is

```
/usr/local/sbsat-2.5b-5
```

Set a link to this directory from `/usr/local` using a command like the following except with the correct version and revision numbers:

```
$ ln -s sbsat-2.5b-5 sbsat
```

3.4 Compiling SBSAT

Become `root` as in Section 3.3 (This step may not be necessary). Change to the directory containing the SBSAT files, called the *root directory of the SBSAT tree*. If you followed the instructions in Section 3.3 this is accomplished with the following command:

```
$ cd /usr/local/sbsat
```

Issue the commands

```
$ ./configure
$ make
```

From now on files and directories contained in or below the root directory of the SBSAT tree will be referred to with `.../` prepended to their paths originating

from that directory. If no errors are reported the SBSAT executable, named `sbsat`, exists in directory `.../src`. To use the executable conveniently from any directory it is advisable to set a link to it from some directory that is in your `PATH`. This can be done automatically by executing the command (as `root`)

```
$ make install
```

This command places `sbsat` in `/usr/local/bin`. To do this by hand, if `/usr/local/bin` is in your `PATH` (it normally is), as `root` change directory to `/usr/local/bin` then set a link as in the following

```
$ cd /usr/local/bin
$ ln -s .../src/sbsat .
```

where `.../` should be replaced by the path of the root directory of the SBSAT tree. Now issuing the command `sbsat` from any directory will start the solver. But, don't do this yet as there are some fine points to using SBSAT which must be discussed.

3.4.1 Configure options

There are quite a few options one can use when running `./configure`. For the complete set of options run `./configure --help`. The following few can be very useful.

- Use a different compiler
  `$ ./configure CXX=g++`
- Link the libraries statically
  `$ ./configure --static`
- Enable some compiler optimization flags
  `$ ./configure -enable-optimization`
- Enable the compiler debugging flags, allowing debuggers like `gdb` to hook into `sbsat`
  `$ ./configure -enable-debug`

```
==========================================
Checking longer benchmarks
you may interrupt it any time
All SAT
==========================================
5-wid_300-var_r3.cnf ... Satisfiable
5-wid_400-var_Slide_r3.cnf ... Satisfiable
c5-wid.cnf ... Satisfiable
==========================================
Done - Sucess
==========================================
```

Figure 2: Result of running the cnf tests in `.../tests`.

3.5 Testing SBSAT

A series of regression tests may be run by issuing the following command while in the root directory of the SBSAT tree:

```
$ make check
```

To run any of these tests individually, change to the `tests` directory in the SBSAT directory using the following

```
$ cd .../tests
```

where `...` is replaced by the path of the root directory of the SBSAT tree. In this directory check that the following files are there: `cnf_tests.sh`, `longer_tests.sh`, `trace_tests.sh`, `xor_tests.sh`. Run any of these to test some aspect of the solver. For example, using the command

```
$ /bin/sh cnf_tests.sh
```

results in the output of Figure 2.

4 Quick Start - The basics of running SBSAT

This section illustrates some of the ways a user can fine-tune a run of `sbsat` on a given input. It is assumed that a link to the executable has been set as per Section 3.4. Doing so makes the command `sbsat` accessible to everyone from every directory. The complexity of options which are available necessitates two preliminary sections describing conventions and defining terms that will be used later. All examples in this manual are part of the SBSAT distribution and may be found in the `.../examples` directory.

4.1 Command line

The usage of SBSAT is:

```
$ sbsat [options] [inputfile [outputfile]]
```

There are two basic options required by GNU standard. One is

```
--version
```

This displays the current version. The other is

```
--help
```

This shows all the command line options. More information on these is given later.

If `sbsat` is launched without parameters it expects the input data on standard input.

The first parameter without a dash is the input data file, the second parameter without a dash is the output file. If no output file is specified `sbsat` it will print all output to the terminal.

4.2 A CNF formula as input

SBSAT recognizes a CNF formula as input if it is expressed in an ascii file that is formatted according to the DIMACS standard[5]. Such an input file begins with a line such as the following

[5] See ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi for a complete specification.

```
p cnf 6 8
c This is a demonstration of the CNF format for the SBSAT solver
1 2 3 0
2 3 4 0
3 4 5 0
4 5 6 0
-1 -2 -3 0
-2 -3 -4 0
-3 -4 -5 0
-4 -5 -6 0
```

Figure 3: Contents of the ascii file `small.cnf`

```
Reading File small.cnf  ....
Reading CNF ...  Done
Preprocessing .... Done
Satisfiable
Total Time: 0.008
```

Figure 4: Output generated by the command `$ sbsat small.cnf`.

```
p cnf <number_of_variables> <number_of_clauses>
```

where `<number_of_variables>` is the number of distinct variables present in the file and `<number_of_clauses>` is the number of clauses present in the input file. Lines starting with the character `c` indicate a comment and are ignored. Variables are represented as positive numbers, beginning with 1. A positive literal is a positive number and a negative literal is a negative number. Each clause is expressed on one line as a 0 terminated list of numbers, separated by blanks, and representing the literals of the clause. The contents of a file named `small.cnf` is shown in Figure 3.

Use the following commands to run `sbsat` on file `small.cnf` (assume the starting point is the root directory of the SBSAT tree):

```
$ cd examples
$ sbsat small.cnf
```

The output is shown in Figure 4.

In order to get the actual satisfiable assignment from the solver we need to add the input parameter to the command line which instructs the solver to

```
Reading File small.cnf  ....
Reading CNF ...  Done
Preprocessing .... Done
Preprocessing .... Done
1 2 -3 4 5 -6
Satisfiable
Total Time: 0.009
```

Figure 5: Output generated by command `$ sbsat` -*R* r `small.cnf`.

output the solution. The following command is this:

```
$ sbsat -R r small.cnf
```

Remark: The format of `small.cnf` is automatically determined by `sbsat` so a command line switch to set a format is not necessary.

Remark: The order of the parameters on the command line usually does not matter[6] provided the values remain immediately to the right of the switches they associate with. So, in this case the following command line would do exactly the same as the one above.

```
$ sbsat small.cnf -R r
```

Output from the above command is shown in Figure 5.

Remark: In case the same switch is used more than once on the command line, the rightmost switch applies. For example,

```
$ sbsat small.cnf -R r -R f
```

prints fancy instead of raw output format (see Pages 93 and 93 for the meaning of these formats).

The default output mixes solution information with execution information. Solution information may be separated from execution information as follows.

```
$ sbsat small.cnf -R r --output-file output.txt
```

Output from this run is sent to the file `output.txt` as follows:

```
1 2 -3 4 5 -6
```

[6]With the exception of *-All* preprocessing switch and preprocessing enable/disable switches. All switches are listed and described in Section 7.

```
SBSAT is a SAT solver.
Usage: sbsat [OPTIONS]...  [inputfile [outputfile]]

Options:
  --help, -h              Show all program options
  --version               Show program version
  --create-ini            Create ini file
  --ini <string>          Set the ini file [default=``~/sbsat.ini'']
  --debug <number>        debugging level (0-none, 9-max) [default=2]
  --debug-dev <string>    debugging device [default=``stderr'']
  --params-dump, -D       dump all internal parameters before processing
  --input-file <string>   input filename [default=``-'']
  --output-file <string>  output filename [default=``-'']
  --temp-dir <string>     directory for temporary files [default=``$TEMP'']
  --show-result <string>, -R <string>
                          Show result (n=no result, r=raw, f=fancy) [default=``n'']
  ...
```

Figure 6: Beginning of output generated by the command `sbsat --help`.

Remark: Some of the command line options have both a short and a long flag which can be used interchangably. For example the '*--show-result*' switch is interchangeable with '*-R*'.

All available switches can be printed using the following command:

```
$ sbsat --help
```

which gives the output shown, in part, in Figure 6.

4.3 An input in canonical (BDD) form

The preferred input type is the canonical form referred to on Page i. A detailed explanation is given in Section 9.1. The canonical form depends on the notion of BDDs which is explained in Section 10.1.

An ascii file containing input in canonical form begins with a line such as the following:

```
p bdd <number_of_variables> <number_of_functions>
```

where `<number_of_variables>` is the number of distinct variables present in the file and `<number_of_functions>` is the number of Boolean functions present in the file. Variables are given names which are strings of alphabetic and numeric characters and the underscore character, in any order. A comment begins with

';' and may start anywhere on a line and applies to the end of the line. Each line starting with a Boolean function identifier listed in the **Boolean Function** item of Section 2.2, or the identifier of an extension of one of those functions, or a manipulator (see Section 9.1 for extensions and manipulators) represents a Boolean function. For example, the following lines can be in a file containing a canonical form expression:

```
imp(-x3, -x4)
xor(x1, -x5)
xor6(x8, x3, -x2, x7, -x4, -x1)
```

Remark: Since no extension of a binary function can take 1 argument, `xor1(-x1)` is not admitted.

A function argument may be a variable, a function, or a reference to a function defined elsewhere in the file. To support the latter, every function is assigned a unique index integer corresponding to the order the function appears in the file. The first function has index 1, the next has index 2 and so on. There may be several commented lines between two functions but those functions still have consecutive index numbers. A function may be referenced by appending its index number to the '$' character. One or more arguments of a function may contain function references but the references may not point forward: that is, the index in a function reference cannot be greater than or equal to the index of the function in which the reference is made. Here is an example:

```
p bdd 4 5
or(x2, x3)
and(x3, x4)
imp(x3, $2)
xor4($3, $1, x4, x1)
and(x2, x3)
```

The fourth line of this group is equivalent to

```
xor4(imp(x3, and(x3, x4)), or(x2, x3), x4, x1)
```

which is *also* recognized by `sbsat`.

Because it is possible to reference functions, it is possible that some functions which are *not* at the *top-level* (that is, not among those to be satisfied) exist

as functions specified in an input file. Such functions are distinguished from top-level functions by prepending '*' to top-level functions only. For example:

```
p bdd 4 5
or(x2, x3)
and(x3, x4)
imp(x3, $2)
; The next line is the first *ed line of the file
*xor4($3, $1, x4, x1)
*or(x2, x3)
```

represents the problem

$$((x_3 \Rightarrow (x_3 \wedge x_4)) \oplus (x_2 \vee x_3) \oplus x_4 \oplus x_1) \wedge (x_2 \vee x_3)$$

If no functions have '*' prepended, then all functions are treated as top-level functions.

4.4 An input in XOR format

The XOR format is a specialization of the canonical form allowing up to two levels of function nesting. However, the grammar of this format is very different. The following is the example `.../examples/xortest.xor`:

```
p xor 12 3
x123 x125 x156 = 0
x134 x155x127x167 = 1
x1x2x3 x45x145x167 = 0
```

which is equivalent to the following canonical form

```
p bdd 12 3
*equ(xor3(123, 125, 156), F)
*equ(xor(134, and3(155, 127, 167)), T)
*equ(xor(and3(1, 2, 3), and3(45, 145, 167)), F)
```

This may be solved using the following command:

```
$ sbsat xortest.xor
```

The result is shown in Figure 7.

One peculiarity of this format is that all variables must have names that begin with `x` and end with a number. No other variable names are allowed. See Section 9.4 for important details concerning this format.

4.5 Reusing functions (Macros)

The canonical form only (that is, not `cnf` or `xor` formats, among others) supports a rudimentary macro facility. A macro is defined using the directive `#define` with the following syntax:

```
#define <pattern> # <function-specifier>
```

where `<pattern>` consists of an identifier and a parenthesized argument list. Wherever the `<pattern>` is matched in the file, `<function-specifier>` is substituted. Then, `<function-specifier>` takes as parameters those arguments specified in `<pattern>`.

Many inputs, particularly those representing an "unfolding" of some form of temporal logic, consist of a high percentage of functions which are identical except that all variable input numbers are displaced by some amount. In such a case there is a shortened way to express those functions using `#define`. For example, the file of Figure 8 may be written equivalently as the file of Figure 9. More information about `#define` is found in Section 9.1.6, Page 58.

4.6 Printing functions

In canonical form files only, one may use `print_tree` or `pprint_tree` to print the truth table of a function as a BDD. For example,

```
print_tree(or3 (x4, x5, -x6))
```

prints the following to standard output

```
-------------------------------------------------------
                        x6
            x5                          T
      T           x4
                T       F
-------------------------------------------------------
```

```
Reading File xortest.xor  ....
Reading XOR ...  Done
Preprocessing .... Done
Satisfiable
Total Time: 0.008
```

Figure 7: Result of executing the command `sbsat xortest.xor`.

```
p bdd 44 5
equ(xor(1, and(-17, 33)), ite(15, or(33, -40), -33)))
equ(xor(2, and(-18, 34)), ite(16, or(34, -41), -34)))
equ(xor(3, and(-19, 35)), ite(17, or(35, -42), -35)))
equ(xor(4, and(-20, 36)), ite(18, or(36, -43), -36)))
equ(xor(5, and(-21, 37)), ite(19, or(37, -44), -37)))
```

Figure 8: A canonical form input of "sliding" functions.

```
p bdd 44 5
#define slide(1, 17, 15, 33, 40)
#       equ(xor(1, and(-17, 33), ite(15, or(33, -40), -33)))
slide(1, 17, 15, 33, 40)
slide(2, 18, 16, 34, 41)
slide(3, 19, 17, 35, 42)
slide(4, 20, 18, 36, 43)
slide(5, 21, 19, 37, 44)
```

Figure 9: An equivalent input using macros.

```
p bdd 30 11 ; 30 vars, 11 functions
Initial_Branch(#1, var*%25.1111, a%10, b, t*e)
  ; These variables will be branched on first.
  ;‘*’ is a wildcard. a % influences the heuristic value.
Initial_Branch(#2, x, var*, b%10.3948, 5, v2)
  ; These variables will be branched on second.
  ; b is ignored here because it appeares in an Initial_Branch statement above.
ite(var1, T, F) ; Function $1, no BDD created.
var1            ; Function $2, no BDD created.
                ; The two preceeding lines created identical functions.
                ; T is built in for True, F is built in for False.
and(var1, var2)          ; Function $3, no BDD created.
and4(a, b, 1, 2)         ; Function $4, no BDD created.
*or3($3, not($4), -var1) ; Function $5, BDD 1.
*ite 4 5 6               ; Function $6, BDD 2.
                         ; Notice parathensis and commas are not required.
ite 2            ; Function $7, no BDD created.
    ite 3 4 5    ; Comments are ignored, even in the middle of a function.
    and or 6 7 8 ; Lines can be hard to read without parathensis and commas.
  ; Defining the ternary majority-vote operator.
#define majv(x, y, z) # ite(x, or(y, z), and(y, z))
  ; Defining a quintal  operator.
#define AndOr4(a, b, c, d) # and3(OR(a, b), OR(b, c), OR(c, d))
  ; Previously defined functions can be used to define more complex functions.
#define AndOr4_MAJV(v1, v2, v3, v4)
# AndOr4(majv(v1, v2, v3), majv(v1, v3, v4), majv(v1, v2, v4), majv(v2, v3, v4))
*AndOr4_majv(tem1e, tem2e, tem3e, tem4e) ; Function $8, BDD 3.
                                         ; There is no case sensitivity.
  ; Overloading the AND operator to be the OR of 3 variables.
#define and(x, y, z) # or3(x, y, z)
*and(var1, var2, var3) ; Function $9, BDD 4.
  ; Returning the AND operator to normal.
#define and(x, y) # ite(x, y, F)
print_tree  $7 ; print - No function created, no BDD created.
pprint_tree $7 ; pretty print - No function created, no BDD created.
*minmax(4, 0, 1, x10, x9, x8, x7) ; eqn $10, BDD 5
print_tree minmax(4, 0, 1, x10, x9, x8, x7) ; No function created, no BDD created.
*or(pprint_tree(print_tree(and(2, 3))), 4) ; Function $11, BDD 6.
  ; This function is identical to the function created by ‘*or(and(2, 3), 4)’
  ; The difference is that function $11 also prints ‘and(2, 3)’ in two
  ; different printing styles.
*print_tree($11) ; No function created, no BDD created.
                 ; A ‘*’ in front of directives is ignored.
```

Figure 10: A complex canonical form input.

See Page 59 for more information on the use of print directives.

4.7 An example of a complex canonical form input

Figure 10 illustrates the use of all the points discussed in previous sections related to the canonical form of input. Some annotations in the example illustrate additional file format points not covered in the text. See Section 9.1 for details.

Remark: Although both parentheses and commas are optional, their use is recommended to improve the reader's understanding of the input.

4.8 Translating an expression to canonical form

Two examples of translating an expression, arising from real problems, to canonical form are presented. The steps involved are: 1) construct that expression in domain-specific terms; 2) translate to a conjunction of functions; 3) translate to canonical form. Step 3) is usually straightforward. The first example is related to reconfigurable computing and is interesting because it is naturally expressed as a *Quantified Boolean Formula* (QBF) with one alternation, which is often a difficult problem to solve. The second example relates to formal verification.

4.8.1 Interconnect synthesis in reconfigurable computing

Many reconfigurable computers consist of multiple field-programmable processors (FPGAs) connected through a Programmable Interconnect Network (PIN) as shown in Figure 11. Interconnect synthesis is the process of configuring the PIN to match the communication requirements of the designs implemented on the processors. The general architecture of a PIN is depicted in Figure 12. A PIN routes signals between various input and output pins of the FPGAs: the specific routing is determined by the control signals on each of the routing blocks. One of many available routing blocks is shown in Figure 13, this one on the well-known Wildforce board.

Typically, but not necessarily, the control signals define a permutation of the inputs of the block and the permuted signals are routed to the corresponding output pins of the block. Each control signal can take a value from $\{T, F\}$

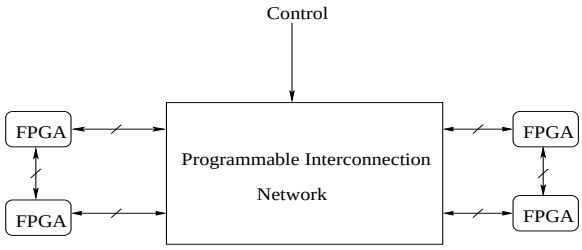


Figure 11: *Example of a Reconfigurable Computer*

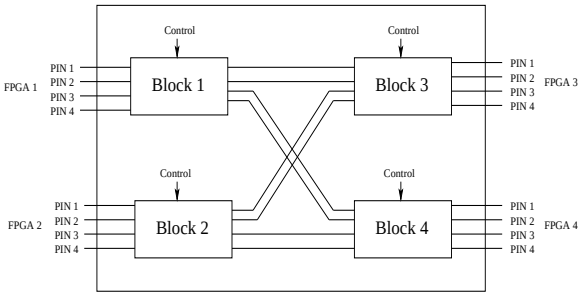
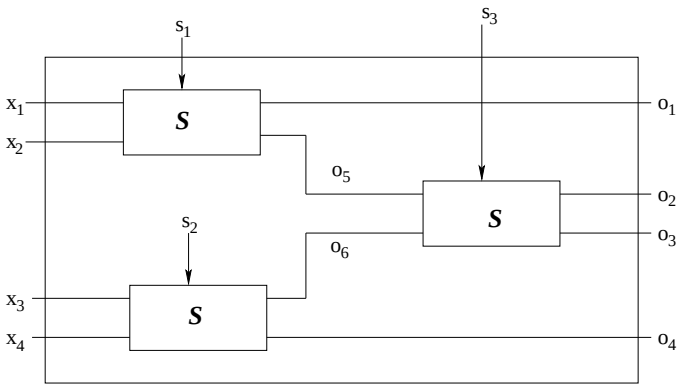


Figure 12: *A Programmable Interconnection Network*



$$o_1 = x_1\bar{s}_1 + x_2 s_1 \qquad o_2 = (x_3\bar{s}_2 + x_4 s_2)\bar{s}_3 + (x_2\bar{s}_1 + x_1 s_1)s_3$$

$$o_3 = (x_2\bar{s}_1 + x_1 s_1)\bar{s}_3 + (x_3\bar{s}_2 + x_4 s_2)s_3 \qquad o_4 = x_4\bar{s}_2 + x_3 s_2$$

Figure 13: *A typical routing block*

or be unassigned. An assignment of values to control signals is said to be a *program* of the interconnection network. Thus, a program defines a routing of the signals through the interconnection network. A required routing may be realizable through one or more programs or not realizable at all depending upon the routing capabilities of the interconnection blocks and how they are connected. A *configuration* of an interconnection network refers to a set of routes realized by a program. Whereas a program defines a configuration, it is not necessary that each configuration is realizable by a unique program.

The problem of interconnect synthesis can be formulated as a problem of determining the satisfiability of a class of QBFs. For a PIN, let PI be the set of *primary inputs* (those connecting to FPGA outputs), PO be the set of *primary outputs* (those connecting to FPGA inputs), and IO be the set of *intermediate outputs* (those not directly accessible through pins). Let M be a desired routing from PI to PO and $Constraints_{M,IO}(PI, PO)$ be a set of contraints which evaluates to T if and only if values on PO match values on a given PI according to M without any inconsistencies among IO. The QBFs have the following form:

$$\forall\ PI\ \exists\ PO\ \&\ IO\ s.t.\ Constraints_{M,IO}(PI, PO)$$

For this class, there is an efficient method for eliminating the Quantifiers resulting in a system of quantifier-free formulas that can be determined using ordinary satisfiability solvers. The key idea, called *impulse response*, is to establish constraints that force exactly one route from a single input to its destination at a time, and to repeat this process for all inputs.

Given an n dimensional Boolean vector $V = \{x_1, x_2, \cdots x_n\}$, define *impulse*$(i)$ to be an assignment of F to variable x_i and T to all the other variables in V. Clearly, there are n impulses for an n dimensional vector. For each impulse, it is straightforward to build constraints that force the target primary output to take value F and all other primary outputs to take value T while enforcing consistency among intermediate values (an example follows). Call such a constraint, for *impulse*(i), $ImpConstraint_{M,IO}(i)$. Then the QBF above can be replaced with the following Boolean expression:

$$\wedge_{i=1}^{n} ImpConstraint_{M,IO}(i) \wedge x_i \equiv F \wedge_{j \neq i} x_j \equiv T$$

which, it can be shown, evaluates to T if and only if the QBF above does.

Consider, for example, just the routing block of Figure 13. The primary inputs are $\{x_1, x_2, x_3, x_4, s_1, s_2, s_3\}$, the primary outputs are $\{o_1, o_2, o_3, o_4\}$, and the two intermediate outputs are $\{o_5, o_6\}$. Suppose each subblock S (there are three of them) either routes its two inputs directly to its two outputs (for example, x_1 is routed to o_1 and x_2 is routed to o_5 through the upper left subblock if $s_1 = F$) or crosses its routes (for example, x_1 is routed to o_5 and x_2 is routed to o_1 if $s_1 = T$). Then one can write the four equations shown in the Figure that relate primary outputs to primary inputs. Those equations are the basis for the consistency constraints needed.

The precise constraints depend on the routing desired. Suppose we wish to determine whether there is a program (assignment to $\{s_1, s_2, s_3\}$) that realizes the configuration x_1 to o_1, x_2 to o_3, x_3 to o_2, and x_4 to o_4. For *impulse*(1) the consistency constraints are

$$o_{11} \equiv ite(s_1, x_{12}, x_{11}) \wedge o_{12} \equiv ite(s_3, o_{16}, o_{15}) \wedge o_{13} \equiv ite(s_3, o_{15}, o_{16}) \wedge o_{14} \equiv ite(s_2, x_{13}, x_{14})$$
$$\wedge o_{15} \equiv ite(s_1, x_{11}, x_{12}) \wedge o_{16} \equiv ite(s_2, x_{14}, x_{13})$$
$$\wedge x_{11} \equiv o_{11}$$

These are conjoined with the constraints forcing *impulse*(1) which are

$$x_{11} \equiv F \wedge x_{12} \equiv T \wedge x_{13} \equiv T \wedge x_{14} \equiv T$$

Similar constraints may be constructed for *impulse*(2) through *impulse*(4). The conjunction of all four sets of constraints is the Boolean expression of interest: if some assignment to $\{s_1, s_2, s_3\}$ satisfies that expression, that assignment routes primary inputs to primary outputs as desired. The next step is to write the constraints in canonical form. This is straightforward and the result is shown in Figure 14.

4.8.2 Bounded Model Checking

A sequential circuit differs from a combinational circuit in that output values depend not only on current input values but also on the history of change of those values. This may be modeled by a digraph such as the one in Figure 15 where

```
p bdd 43 32
; Consistency constraints for impulse(1)
equ(o11, ite(s1, x12, x11))
equ(o12, ite(s3, o16, o15))
equ(o13, ite(s3, o15, o16))
equ(o14, ite(s2, x13, x14))
equ(o15, ite(s1, x11, x12))
equ(o16, ite(s2, x14, x13))
equ(x11, o11)
; Constraint forcing impulse(1)
and4(-x11, x12, x13, x14)

; Consistency constraints for impulse(3)
equ(o21, ite(s1, x22, x21))
equ(o22, ite(s3, o26, o25))
equ(o23, ite(s3, o25, o26))
equ(o24, ite(s2, x23, x24))
equ(o25, ite(s1, x21, x22))
equ(o26, ite(s2, x24, x23))
equ(x21, o21)
; Constraint forcing impulse(2)
and4(x21, -x22, x23, x24)

; Consistency constraints for impulse(3)
equ(o31, ite(s1, x32, x31))
equ(o32, ite(s3, o36, o35))
equ(o33, ite(s3, o35, o36))
equ(o34, ite(s2, x33, x34))
equ(o35, ite(s1, x31, x32))
equ(o36, ite(s2, x34, x33))
equ(x31, o31)
; Constraint forcing impulse(3)
and4(x31, x32, -x33, x34)

; Consistency constraints for impulse(4)
equ(o41, ite(s1, x42, x41))
equ(o42, ite(s3, o46, o45))
equ(o43, ite(s3, o45, o46))
equ(o44, ite(s2, x43, x44))
equ(o45, ite(s1, x41, x42))
equ(o46, ite(s2, x44, x43))
equ(x41, o41)
; Constraint forcing impulse(4)
and4(x41, x42, x43, -x44)
```

Figure 14: Interconnect synthesis example in canonical form.

each node represents a state of all output and intermediate values based on some input change history and each arc is labeled by input(s) whose changing value(s) cause(s) a transition from one state to another. Each transition is referred to below as a *time step*.

Let a circuit *property* be expressed as a propositional Boolean expression. An example of a property for a potential JK flip flop might be `J` $\wedge$ `K` $\wedge$ `Q` meaning that it is possible to have an output `Q` value of T if both inputs `J` and `K` have value T.

The following time-dependent expressions are among those that typically need to be proved for a sequential circuit.

1. For every path (in a corresponding digraph) property P is *true* at the next time step.
2. For every path property P is *true* at some future time step.
3. For every path property P is *true* at every future time step.
4. For every path property P is *true* until property Q is *true*.
5. There exists a path such that property P is *true* at the next time step.
6. There exists a path such that property P is *true* at some future time step.
7. There exists a path such that property P is *true* at every future time step.
8. There exists a path such that property P is *true* until property Q is *true*.

To construct a Boolean expression which must have value T if and only if the desired time-dependent expression holds, the Boolean expression must have components which:

1. force the property or properties of the time dependent expression to hold,
2. establish the starting state,
3. force legal transitions to occur.

In order for the Boolean expression to remain of reasonable size it is generally necessary to bound the number of time steps in which the time-dependent expression is to be verified; hence, bounded model checking.

For example, consider a simple 2-bit counter whose outputs are represented by variables v_1 (LSB) and v_2 (MSB). Introduce variables v_1^i and v_2^i whose values are intended to be the same as those of variables v_1 and v_2, respectively, on the

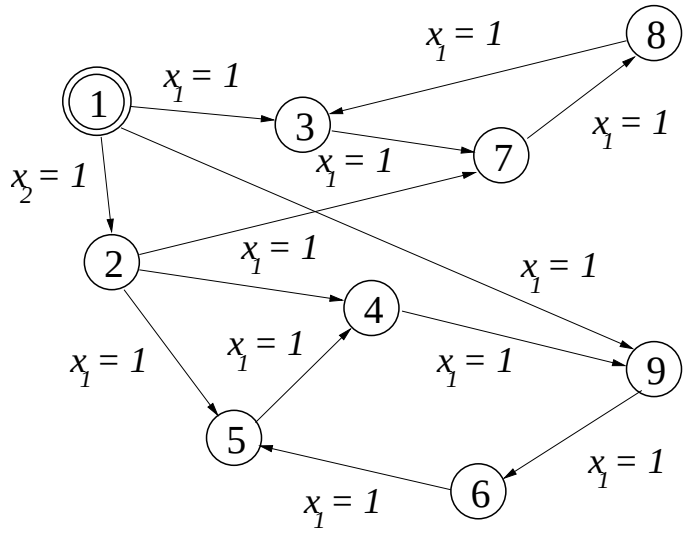


Figure 15: A state machine representing a sequential circuit

ith time step. Suppose the starting state is the case where both v_1^0 and v_2^0 have value 0. The transition relation is

Current Output		**Next Output**
00	:	01
01	:	10
10	:	11
11	:	00

which can be expressed as the following Boolean function:

$$(v_1^{i+1} \equiv \neg v_1^i) \wedge (v_2^{i+1} \equiv v_1^i \oplus v_2^i).$$

Suppose the time-dependent expression to be proved is:

Can the two-bit counter reach a count of 11 in exactly three time steps?

Assemble the propositional formula having value T if and only if the above query holds as the conjunction of the following three parts:

1. **Force the property to hold:**

$$(\neg(v_1^0 \wedge v_2^0) \wedge \neg(v_1^1 \wedge v_2^1) \wedge \neg(v_1^2 \wedge v_2^2) \wedge (v_1^3 \wedge v_2^3))$$

```
p bdd 8 11
not(and(v10, v20)) ; Force a property to hold
not(and(v11, v21))
not(and(v12, v22))
and(v13, v23)

and(not(v10), not(v20)) ; Express a starting state

equ(v11, not(v10)) ; Force legal transitions
equ(v21, xor(v10, v20))
equ(v12, not(v11))
equ(v22, xor(v11, v21))
equ(v13, not(v12))
equ(v23, xor(v12, v22))
```

Figure 16: Bounded model checking example in canonical form.

2. **Express the starting state:**

$$(\neg v_1^0 \wedge \neg v_2^0)$$

3. **Force legal transitions (repetitions of the transition relation):**

$$\begin{aligned}(v_1^1 \equiv \neg v_1^0) \wedge (v_2^1 \equiv v_1^0 \oplus v_2^0) \wedge \\ (v_1^2 \equiv \neg v_1^1) \wedge (v_2^2 \equiv v_1^1 \oplus v_2^1) \wedge \\ (v_1^3 \equiv \neg v_1^2) \wedge (v_2^3 \equiv v_1^2 \oplus v_2^2)\end{aligned}$$

The reader may check that the following satisfy the above expressions:

$$v_1^0 = 0, v_2^0 = 0, v_1^1 = 1, v_2^1 = 0, v_1^2 = 0, v_2^2 = 1, v_1^3 = 1, v_2^3 = 1.$$

This assignment can be found by running `sbsat` on the example file `bmc_example.ite` with the flag `-R r`. It may also be verified that no other assignment of values to v_1^i and v_2^i, $0 \leq i \leq 3$, satisfies the above expressions by running the previous command with two extra flags, namely `--max-solutions 0` and `-All 0` (details on these flags can be found in Section 7). The constraints are shown in canonical form in Figure 16.

4.9 Choosing a search procedure

By default, `sbsat` searches using the backtracking SMURF solver. But this can be changed using command line switches. The table below summarizes the switches and results.

Search	Default	Switch	Description
SMURF	yes	*-b*	Backtrack w/ learning
BDD WalkSAT	no	*-w*	Local search
WVF (experts)	no	*-m*	For debugging and research
Simple (experts)	no	*-t*	For debugging and research
No solver	no	*-n*	Exit `sbsat` after preprocessing

4.10 Converting the input file

SBSAT supports conversion between some of its input formats. For example, an input format such as `xor` may be converted to `cnf`. In order to get as direct a translation as possible, the preprocessing should be disabled when performing conversions. This can be achieved by using *-In* 0 *-All* 0 switches on the command line.

However, in some cases conversion is done so as to take advantage of preprocessing. Thus, given a file in `smurf` format, one could preprocess with a result in the same format or a different format like `cnf`.

One might also want to convert between formats so that a problem might be attempted on a variety of solvers. For example, many problems come in `trace` or `BDD` formats but traditional SAT solvers do not recognize those formats so they must be converted to `cnf`.

Format conversions supported by SBSAT are listed in the following table (more functionality is currently being developed). To determine whether format `A` can be converted to format `B`, locate `A`'s row and the answer appears in `B`'s column. For example, format `xor` converts to `cnf` but not from `cnf` to `xor`.

	cnf	**dnf**	**bdd**	**smurf**	**trace**	**prove**	**xor**
cnf	yes	no	no	yes	no	no	no
dnf	yes	no	no	yes	no	no	no
bdd	yes	no	no	yes	no	no	no
smurf	yes	no	no	yes	no	no	no
trace	yes	no	no	yes	no	no	no
prove	yes	no	no	yes	no	no	no
xor	yes	no	no	yes	no	no	no

A sample command for translating a file of one format to another is:

```
$ sbsat -c flowers.ite > flowers.cnf
```

which translates a file in trace format to one in cnf format. To get a *direct* translation, preprocessing must be turned off. Thus:

```
$ sbsat -c -All 0 -In 0 flowers.ite > flowers.cnf
```

or

```
$ sbsat -c -All 0 -In 0 flowers.ite flowers.cnf
```

or

```
$ sbsat -c -All 0 -In 0 flowers.ite --output-file flowers.cnf
```

would be acceptable.

5 Quick Start - Some advanced features of `sbsat`

5.1 Preprocessing

The preprocessor attempts to manipulate a given expression into an internal form that should lead to a smarter search. This section highlights the main points regarding the use of preprocessing.

5.1.1 Preprocessor options

Most of the many possible preprocessing options available to `sbsat` users are shown in the table below (options not listed are considered unstable). See Section 10 for an explanation of these options and when an option might pay off and when it might be a liability.

Name	Default	Option	Formats	Description
Pattern Matching	yes	Cl	CNF	see Section 10.2
Generalized Cofactoring	yes	Co	All	see Section 10.3
Branch Pruning	yes	Pr	All	see Section 10.4
Strengthening	yes	St	All	see Section 10.5
Inferences	yes	In	All	see Section 10.6
Existentially Quantify	yes	Ex	All	see Section 10.7
Cluster + ExQuant	yes	Ea	All	see Section 10.8
Cluster + ExQuant + Safe	yes	Es	All	see Section 10.9
Dependent Var. Clustering	yes	Dc	All	see Section 10.10
Rewind	yes	Rw	All	see Section 10.11
Splitter	yes	Sp	All	see Section 10.12

5.1.2 Preprocessor sequence

Preprocessor options can be applied in any order, as desired, and repeated by specifying the order and repetitions on the command line in the *preprocessor sequence*. A preprocessor sequence consists of a list of *preprocessor run*s, or just *run*s. Each run may be followed by a positive integer or another run. A run is a concatenation of preprocessor options from the above table, wrapped inside

matching curly braces. The preprocessing operations specified by the options of a run are applied in the order in which they are specified. The run may be repeated if it is immediately followed by a positive integer R. In that case the run is repeated R times or until the internal form is the same before and after the run (that is, until reaching a fixed point). If the run is not followed by a number then it will repeat until reaching a fixed point. For example, the following is a run which applies existential quantification, followed by dependent variable clustering:

```
{ExDc}
```

The following is a preprocessor sequence with three runs, the first repeated at most 3 times, the second at most 2 times and the third at most 10 times:

```
{ExDc}3{ExSt}2{ExPr}10
```

Nesting of runs is allowed as in

```
{{StDc{Pr{ExDc}3}{Ex}10}Ex}
```

There may be times when a preprocessor run should be run the maximum number of times specified, instead of stopping early once reaching a fixed point. There may also be times when only certain functions of a preprocessing run should be considered when determining a stopping condition. All of this can be controlled by bounding preprocessor runs, or options, inside of square brackets. A set of square brackets should be followed by either 0 or 1 where a 0 forces the internal form to be recognized as not altered, and a 1 forces the internal form to be recognized as altered. For example:

```
{[ExDc]1}10
```

This sequence will causes the preprocessing run to loop 10 times even if the internal form reachs a fixed point prior to 10 iterations. This happens because the square brackets force `sbsat` to consider the internal form as having been modified, even though it may not have been.

```
{Ex[St]0Dc}
```

This sequence causes the preprocessing run to loop only when either ‘Ex’ or ‘Dc’ modify the internal form. ‘St’ may modify the internal form, but the looping process ignores this information.

5.1.3 Preprocessor command

Proprocessing is specified on the command line using the *--preprocess-sequence* or *-P* switch followed by a preprocessor sequence. For example,

```
$ sbsat --preprocess-sequence {ExDc}3{ExSt}2{ExPr}10 small.cnf
```

The following does the same thing:

```
$ sbsat -P {ExDc}3{ExSt}2{ExPr}10 small.cnf
```

For some problems, preprocessing might take too long or may not produce a desired result. In this case the user may enable or disable preprocessing options or change their sequence. For example,

```
$ sbsat -St 0 small.cnf
```

skips the strengthening operation in the current sequence. For example,

```
$ sbsat -P {Dc{ExSt}{ExPr}St}10 -St 0 small.cnf
```

is the same as

```
$ sbsat -P {Dc{Ex}{ExPr}}10 small.cnf
```

Also, the user may want to turn off every preprocessing option except one or two. This can be achieved by using the *-All* 0 command, which turns off all preprocessing options, followed by (for example) *-St* 1, which turns strengthening back on. For example,

```
$ sbsat -P {Dc{ExSt}{ExPr}St}10 -All 0 -St 1 small.cnf
```

is the same as

```
$ sbsat -P {{St}St}10 small.cnf
```

Remark: One may avoid long preprocessing by saving the problem after preprocessing in SMURF file format[7] using, for example,

```
$ sbsat --output-file newfile.smurf -s myoldfile
```

and disabling preprocessing the next time `sbsat` is applied to the same input using

```
$ sbsat -All 0 newfile.smurf.
```

Remark: Input files in canonical form (see Sections 4.3 and 9.1) allow pre-

[7] See Section 9.7 for a description of the SMURF file format.

processing operations (and some other operations) to be performed *while the input file is parsed*. The canonical form admits directives which specify such operations in the order and precisely at the point they appear in the input file. See Section 9.1.6 for a list of such operations.

5.2 Heuristics

Two search heuristics are included with SBSAT. One of these we refer to as the LSGB heuristic for Locally Skewed, Globally Balanced. It is designed to work well on inputs with little (known) structure and therefore favors assignments that reveal inferences which can be derived from information about individual functions (locally skewed) and simultaneously favors assignments that tend to balance the entire search space (globally balanced). For more information on this heuristic and insights for its control see Section 11.2.

The other search heuristic is a variant of the VSIDS heuristic Chaff uses. See Section 11.3 for information about the use and control of this heuristic.

There is also an option, which we call *combined*, that allows the user to mix the two heuristics. What this mixing accomplishes is given in Section 11.

The following table shows command line switches for selecting these heuristics and associated parameters.

Heuristic	Default	Switch	Description
LSGB	yes	-*H* `j`	Locally Skewed, Globally Balanced
VSIDS	no	-*H* `l`	Number of occurrences of literals
Combined	no	-*H* `jl`	Combination of the two above

5.3 The lemma cache

The size of the cache in which the lemmas are stored is fixed throughout the branching process. The memory needed to maintain the cache is automatically allocated and accomodates all lemmas in the cache. Usually, the bigger the memory cache, the slower the search process. Therefore one is confronted with the following optimization problem: choose the lemma cache size small enough to avoid burdensome overhead yet large enough that lemmas in the cache will

significantly reduce search. The parameter to use for controlling the lemma cache is *--max-cached-lemmas* or *-L* and an example of its use is this:

```
$ sbsat -L 1000 problem.cnf
```

It is possible to set the lemma cache to 0. This will prevent any lemma from being created.[8] For example, use

```
$ sbsat -L 0 slider_80_unsat.ite
```

For some problems this will yield significantly better results than when the lemmas are used. See Section 12.1.2 for a discussion of when to use lemmas and when not to use lemmas.

Remark: The heuristic used to discard lemmas from the cache when newly created lemmas are added to a full cache is not compatible with the option that disables lemma creation. Also the effectivness of the lemma heuristic decreases with descreasing lemma cache size. See Section 12.1.2 for a discussion.

Remark: The Chaff-like (VSIDS) heuristic requires lemma creation and therefore is not compatible with the option that causes the lemmas not to be created (*-L* 0).

Backjumping is inherently tied to lemmas and therefore, the backjumping feature is active when *-L* has an argument greater than 0, and the backjumping feature is inactive when *-L* 0 is used. To turn the backjumping feature on, but store 'almost' no lemmas, use the flag *-L* 1. Please consider the *--backjumping* flag deprecated.

5.4 Controlling preprocessing and search time

In some situations preprocessing time exceeds the savings in time realized during search. In this case `sbsat` offers some ways to change the amount of preprocessing time performed. These include:

1. Change the preprocessing sequence to perform less iterations (see Section 5.1.2).
2. Specify a time limit, in seconds, for how long preprocessing can take. After the time limit has been reached the preprocessor will quit and `sbsat` will

[8] See Section 12.1.1 for details.

enter the search phase.

3. The user can terminate preprocessing interactively with `^C` provided the switch *--ctrl-c 1* is used on the command line.

4. The user can fast forward through preprocessing with the arrow key (a feature which is soon to be added).

An example of item 2 is the following:

```
$ sbsat small.cnf --max-preproc-time 180
```

which allows 3 minutes for preprocessing and continues to the search phase after that. This time constraint is checked between preprocessing options, so preprocessing could potentially terminate much later than desired.

Search time can also be controlled on the command line using a similar switch. For example,

```
$ sbsat small.cnf --max-branching-time 180
```

limits search time to 3 minutes.

5.5 Creating and using an initialization file

When working on a problem that requires using a long command line over and over, it is convenient to create an initialization file to prevent having to reenter the switches on every run. The initialization file contains a list of settings that are translated to switches by `sbsat` when it is invoked. SBSAT automatically looks for `sbsat.ini` in the user's home directory (that is, it looks for a file `~/sbsat.ini`).

To create an ini file with the default values for all available options use the following command:

```
$ sbsat --create-ini > ~/sbsat.ini
```

The initialization file may be created and/or edited by the user.

Remark: Command line options take precedence over `ini` file settings. This allows short command lines with many custom settings and is useful for experimentation.

Remark: It is possible to maintain several initialization files and load a desired one from the command line. Do this, for example, as follows:

```
$ sbsat --ini myini.ini small.cnf
```

which loads the options of initialization file `myini.ini`.

5.6 Debugging

It is possible that the particular command line settings will cause an inefficient search and/or preprocessing on a given input. The following is a list of suggestions for helping `sbsat` to yield a result.

- Try converting to another format. See Section 17.1.
- Debug prints (in ITE format). See Section 17.2.
- Print internal data from the solver. See Section 17.2.
- Be familiar with BDDs and operations applied to them.
- Output the BDDs before preprocessing by using commands of Section 7.
- Match the BDDs to your original problem.
- If you think you discovered a bug in SBSAT email us!

6 Quick Start - Getting more help quickly

- Check out the SBSAT Web Page [9]
- Email us:
  John Franco franco@gauss.ececs.uc.edu
  Michal Kouril mkouril@ececs.uc.edu
  Sean Weaver weaversa@gmail.com

[9] http://www.cs.uc.edu/~weaversa/SBSAT.html

7 Reference - Command line

The executable file that does the work of SBSAT is called `sbsat` and is run from the command prompt of a Unix shell. The command line has the following structure:

```
sbsat [options] [inputfile [outputfile]]
```

where `inputfile` is the name of a file containing a problem to be solved, and `outputfile` names a file to which output from a run of `sbsat` can be directed. The `inputfile` can take several formats, all somewhat different from each other, which are described in Section 9. Options customize the execution of `sbsat`: they control preprocessing, search, input, and output specifics and more. Observe that `options`, `inputfile`, and `outputfile` are all optional, but that if `outputfile` is used, it is expected to be placed after `inputfile`.

Options are invoked using switches. A switch is preceeded by one or two dashes ('`-`' or '`--`) and should be immediately followed by a parameter which is either an integer or a string, depending on the switch (at the moment there are no integer switches associated with `sbsat`. If a switch requires a parameter, one or more blank characters separates it from the parameter. Switch/parameter pairs are separated from each other and the file names by blanks. There are many switches and they are organized below by type.

Some example runs are as follows:

`$ sbsat` *--help*
: Lists all command line options

`$ sbsat` *-R* `r small.cnf`
: Solve the problem in `small.cnf`, show the result raw

`$ sbsat` *-P {ExDc}3{ExSt}2{ExPr}10* `small.cnf`
: Preprocess the problem in `small.cnf` a certain way, then solve.

Switch-parameter pairs and file names may be placed anywhere on a command line after `sbsat`. Thus, the following two runs are identical:

```
$ sbsat -R r small.cnf
$ sbsat small.cnf -R r
```

A switch may appear more than once on a command line. In that case the

rightmost switch applies. In case contradictory switches are given, the rightmost applies. For example:

```
$ sbsat -b -w
```

will invoke the BDD WalkSAT search (see Page 42 for option `-w`). Some switch combinations cooperate with each other. For example:

```
$ sbsat -All 0 -St 1
```

turns all preprocessing off then turns strengthening on (see Page 43 for preprocessing switches).

7.1 General options

--help, -h	-	Show all program options
--version	-	Show program version
--create-ini	-	Create ini file
--ini <string>	-	Set the ini file [default="~/sbsat.ini"]
--debug <number>	-	debugging level (0-none, 9-max) [default=2]
--debug-dev <string>	-	debugging device [default="stderr"]
--params-dump, -D	-	dump all internal parameters before processing
--input-file <string>	-	input filename [default="-"]
--output-file <string>	-	output filename [default="-"]
--temp-dir <string>	-	directory for temporary files [default="$TEMP"]
--show-result <string>, -R <string>	-	Show result ((n)one, (r)aw, (f)ancy) [default="n"]
--verify-solution <number>	-	Verify solution [default=1]
--expected-result <string>	-	Report error if the result is not as specified Options are SAT, UNSAT, TRIV_SAT, TRIV_UNSAT, SOLV_S, SOLV_UNSAT [default=""]
--comment <string>	-	Comment to appear next to the filename [default=""]
--ctrl-c <number>	-	Enable/Disable Ctrl-c handler to end preproc/search [default=0]
--reports <number>	-	Reporting style during branching (0 - standard, 1 - crtwin) [default=0]
--competition <number>	-	Competition reporting style [default=0]
--sattimeout <number>	-	For SAT Competition SATTIMEOUT [default=0]
--satram <number>	-	For SAT Competition SATRAM [default=0]
--parse-filename	-	For testing purposes

7.2 BDD table options

--num-buckets <number>	-	Set the number of buckets in power of 2 [default=16]
--size-buckets <number>	-	Set the size of a bucket in power of 2 [default=5]
--bdd-pool-size <number>	-	The size of the bdd pool increment [default=1000000]
--gc <number>	-	Use garbage collection [default=1]

7.3 Input options

--limit-and-equ <number>	-	Min # of literals to flag sp. function and_equ [default=2]
--limit-or-equ <number>	-	Min # of literals to flag sp. function or_equ [default=2]
--limit-or <number>	-	Min # of literals to flag sp. function plainor [default=8]
--limit-xor <number>	-	Min # of literals to flag sp. function plainxor [default=5]
--break-xors <number>	-	Break XORS into linear and non-linear functions during search [default=1]

7.4 Output options

-b	-	Start SMURF solver [default]
-w	-	Start BDD WalkSAT solver
-m	-	Start WVF solver
-t	-	Start a stripped down version of the SMURF solver
-n	-	Don't start any brancher or conversion
-s	-	Output in SMURF format
-c	-	Output in CNF format
-v	-	Output in VHDL/FPGA format
-p	-	Output in tree like format
--formatout <char>	-	Output format [default='b']
--cnf <string>	-	Format of CNF output (3sat, qm, noqm) [default="noqm"]
--tree	-	Output BDDs in tree representation (used in conjunction with -p)
--tree-width <number>	-	Set BDD tree printing width [default=64]
--prover3-max-vars <number>	-	Max variables per BDD when reading 3 address code (input format 3) [default=10]

7.5 Preprocessing options

--preset-variables <string>	-	Variables forced during preprocessing [default=""]
--preprocess-sequence <string>, *-P <string>*	-	The preprocessing sequence [default="{ExDc}{ExSt}{ExPr}{ExSp}{Ff}"]
--All, -All	-	Enable/Disable All Preprocessing Options (1/0)
--Cl <number>, -Cl <number>	-	Enable/Disable Clustering (1/0) [default=1]
--Co <number>, -Co <number>	-	Enable/Disable Cofactoring (1/0) [default=1]
--Pr <number>, -Pr <number>	-	Enable/Disable Pruning (1/0) [default=1]
--St <number>, -St <number>	-	Enable/Disable Strengthening (1/0) [default=1]
--In <number>, -In <number>	-	Enable/Disable Inferences (1/0) [default=1]
--Ex <number>, -Ex <number>	-	Enable/Disable Existential Quantification (1/0) [default=1]
--Ea <number>, -Ea <number>	-	Enable/Disable AND-Existential Quantification (1/0) [default=1]
--Es <number>, -Es <number>	-	Enable/Disable AND-Safe Assign + Existential Quantification [default=1]
--Sa <number>, -Sa <number>	-	Enable/Disable Searching for Safe Assignments (1/0) [default=1]
--Ss <number>, -Ss <number>	-	Enable/Disable SafeSearch (1/0) [default=1]
--Pa <number>, -Pa <number>	-	Enable/Disable clustering to find possible values to variables (1/0) [default=1]
--Dc <number>, -Dc <number>	-	Enable/Disable Dependent Variable Clustering (1/0) [default=1]
--Sp <number>, -Sp <number>	-	Enable/Disable Large Function Splitting (1/0) [default=0]
--Rw <number>, -Rw <number>	-	Enable/Disable Rewinding of BDDs back to their initial state (1/0) [default=1]
--Cf <number>, -Cf <number>	-	Enable/Disable Clearing the Function Type of BDDs (1/0) [default=1]
--Ff <number>, -Ff <number>	-	Enable/Disable Searching for the Function Type of BDDs (1/0) [default=1]
--P3 <number>, -P3 <number>	-	Enable/Disable Recreating a new set of prover3 BDDs (1/0) [default=1]
--max-preproc-time <number>	-	Set the time limit in seconds (0=no limit) [default=0]
--do-split-max-vars <number>	-	Threshold above which the Sp splits BDDs [default=10]
--ex-infer <number>	-	Enable/Disable Ex Quantification trying to safely infer variables before they are quantified away (1/0) [default=1]
--gaussian-elimination <char>, *-gauss <char>*	-	Enable/Disable Gaussian Elimination in the preprocessor (1/0) [default='0']

7.6 General solver options

--brancher-presets <string>	-	Variables that are preset before brancher is called. String tokens are [=|!|#|+var|-var]* [default=""]
--dependence <char>	-	Modify Independent/Dependent Variables (n=no change, r=reverse, c=clear) [default='c']
--max-solutions <number>	-	Set the maximum number of solutions to search for. 0 will cause the solver to search for as many solutions as it can find. The algorithm does not guarantee that it reports all possible solutions. [default=1]
--max-brancher-time <number>	-	Set the time limit in seconds (0=no limit) [default=0]

7.7 SMURF Solver options

--lemma-out-file <string>	-	File to dump lemmas to [default=""]
--lemma-in-file <string>	-	File to read lemmas from [default=""]
--csv-trace-file <string>	-	File to save execution trace in CSV format [default=""]
--var-stat-file <string>	-	File to save var stats [default=""]
--cvs-depth-breadth-file <string>	-	Save depth/breadth statistic [default=""]
--backjumping <number>	-	Enable/Disable backjumping (1/0) [default=1]
--max-cached-lemmas <number>, *-L <number>*	-	Set the maximum # of lemmas [default=5000]
--autarky-smurfs <number>	-	Use Autarky Smurfs in the solver (1/0) [default=0]
--autarky-lemmas <number>	-	Use Autarky Lemmas in the solver (Currently Unavailiable) (1/0) [default=0]
--sbj <number>	-	Super backjumping [default=0]
--max-vbles-per-smurf <number>, *-S <number>*	-	set the maximum number variables per smurf [default=8]
--backtracks-per-report <number>	-	set the number of backtracks per report [default=10000]
--max-brancher-cp <number>	-	set the choice point limit (0=no limit) [default=0]
--brancher-trace-start <number>	-	number of backtracks to start the trace (when debug=9) [default=0]
--heuristic <string>, *-H <string>*	-	Choose heuristic j=LSGB, l=Chaff-like lemma based, i=Interactive [default="j"]
--jheuristic-k <number>, *-K <number>*	-	set the value of K [default=3.000000]
--jheuristic-k-true <number>	-	set the value of True state [default=0.000000]
--jheuristic-k-inf <number>	-	set the value of the inference multiplier [default=1.000000]

7.8 BDD WalkSAT solver options

--cutoff <number>	-	BDD WalkSAT number of flips per random restart [default=100000]
--random-option <number>	-	BDD WalkSAT option for random walk (1=Pick a random path to true in current BDD, 2=Randomly flip every variable in current BDD, 3=Randomly flip one variable, 4=Randomly flip one variable in current BDD) [default=1]
--bddwalk-heur <char>	-	BDD WalkSAT Heuristic (a=adaptive novelty+, n=novelty+, r=random) [default='a']
--taboo-max <number>	-	BDD WalkSAT length of taboo list (used in conjunction with novelty+ heuristic) [default=6]
--taboo-multi <number>	-	BDD WalkSAT multiplier for the probablity of picking variables with taboo (used in conjunction with novelty+ heuristic) [default=1.500000]
--bddwalk-wp-prob <number>	-	BDD WalkSAT probablity of making a random walk (used in conjunction with novelty+ heuristic) [default=0.100000]
--bddwalk-prob <number>	-	BDD WalkSAT probablity of picking second best path (used in conjunction with novelty+ heuristic) [default=0.100000]

8 Reference - Initialization file

An initialization file allows the user to launch `sbsat` with a much abbreviated command line. That is, the file may contain all switches and parameters that would normally be part of the `sbsat` command line when it is launched. More than one initialization file can exist to allow the user to define different settings for different problem types by creating initialization files for each type. An initialization file may be loaded during launch using the following:

```
$ sbsat --ini <path-to-initialization-file>
```

If only a filename is specified instead of a path, SBSAT looks in the current directory for the file. If the *--ini* switch is not used, then SBSAT looks in the user's home directory for `sbsat.ini` and loads that file if it is found. If the file is not found, the default internal settings are used. If the file exists and additional command line options are given, those options specified on the command line override any that appear in the initialization file.

An initialization file is a simple text file which may be created and edited using a standard text editor or created by using the following command which dumps the default settings, in the initialization file format, to a file whose name the user specifies:

```
$ sbsat --create-ini > <user-specified-filename>
```

Each line of the initialization file is either whitespace or a command which sets an internal parameter to a value. A command consists of a parameter name on the left, followed by an '`=`' followed by a value on the right. A comment is any text on any line following and including the first occurrence of the '`#`' character on that line. The sample initialization file shown in Figure 17 presets variables 1-15 at the start of preprocessing, sets the debug output level to 3, and changes the default preprocessing sequence to `{ExDc}{ExSt}{ExPr}{{ExEs}1}`. An initialization file only needs to contain the settings the user wants modified from the default settings.

```
# Preprocessing options:
#
# Variables forced during preprocessing.
preset-variables="-1 -2 +3 -4 +5 -6 -7 -8 +9 -10 -11 +12 -13 +14 -15"

debug=3

# The preprocessing sequence
preprocess-sequence="{ExDc}{ExSt}{ExPr}{{ExEs}1}"
```

Figure 17: Example initialization file.

9 Reference - Input formats

Problems to be solved are typically presented to `sbsat` as a file formatted in a particular way (input is also allowed from standard input). This section describes the various input formats accepted by `sbsat`. The canonical format is the most general and is described first. An important format is DIMACS CNF which is specified in

`ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi`

and a description is given here. Other useful formats include XOR, for handling a rather specific type of problem where each conjunct is an `xor` of conjuncts, DNF, Trace (a language used in CMU benchmarks), Prove (automatically generated by the CMU tool named BMC), and SMURF (a special form directly tied to SBSAT data structures).

The inputs formats support up to five different types of statements and each applicable type is described separately in a subsection of the format's description section. The types are as follows:

1. **Comment**: A comment is text beginning with a special character (depending on the input format) and ending with the first newline following that character. All comments are ignored by `sbsat` hence comments can be placed anywhere as separate lines or on the same line after other line types.

2. **File Header**: There is one file header and it occupies one line of the file. It specifies the type of input format and must be placed before all other lines except fully commented lines. The header is immediately checked by

`sbsat` which then applies the appropriate parser to the remainder of the file.

3. **Boolean Function**: Certain syntax, given below, is used to express a Boolean function in a file. Arguments may be variables or Boolean functions or, in some formats, references to Boolean functions. If an assignment to the variables of the file results in values of T for all of a particular subset of (possibly all) Boolean function statements then that assignment is a solution to the problem represented in the input file.

4. **Manipulator**: A manipulator is a Boolean function composed from one or more Boolean functions it takes as arguments. Its purpose is to provide alternative, simpler, or modified forms of its input functions which will lead to faster search.

5. **Directive**: A directive is a statement of control, for example, of search or output, or of substitution. A directive is applied *at the point it occurs in the input file while the file is being parsed*. A directive does not apply to any of the input file that has not been parsed by the time the directive is executed.

Directives, Boolean functions, and manipulators can appear in any order after the file header.

9.1 Canonical form

This is the most general input format. Variables are represented as strings containing letters, numbers, or underscore characters, mixed in any way. Boolean functions (including manipulators), and directives can be split over many lines of the input file (no continuation character is used). Boolean functions are assigned reference numbers in the order they appear in the file, beginning with the number 1. If, say, the 10th Boolean function is to be an argument to the 24th Boolean function, then the reference '`$10`' may be used as the argument instead of writing the entire 10th function again. Doing so wherever possible can considerably shorten input file length, but requires writing nested functions on separate lines. It follows that some way is needed to distinguish those functions from the top-level functions (those to be conjoined as specified in the given problem instance): the character '`*`' is used in the input file to mark a function

as a top-level function and appears as the first character of the first line specifying such a function. A solution to the problem defined in a file is an assignment satisfying those functions (including manipulators) which are preceeded by '`*`' (values of all non-starred functions and non-starred manipulators do not matter). If no function is preceeded by a '*', all functions are considered top level functions.

Lines that get a `$` number start with `var`, `not`, `and`, `nand`, `or`, `nor`, `equ`, `xor`, `imp`, `nimp`, `ite`, `gcf`, `strengthen`, `prune`, `exist`, `universe`, `truth_table`, `minmax`, `safe`, a single positive or negative variable, a function previously defined in a `#define` statement, or a single number with a `$` in front of it. Lines that do not get a `$` number start with `initialbranch`, `print_tree`, or `pprint_tree`, `#define`, `print_xdd`, or `print_flat_xdd`.

9.1.1 Comments

A comment begins with the special character '`;`'.

9.1.2 File Header

The header conforms to the DIMACS standard which is as follows:

```
p bdd <#vars> <#functions>
```

The letter '`p`' must appear at the start of the header. The second header token, which in this case is '`bdd`', identifies the input format. The header contains two integer fields: `<#vars>` is the number of distinct variables contained in the file; `<#functions>` is the number of Boolean functions (including manipulators) contained in the file. The following is an example of a valid header for a file containing 97 functions composed from 56 distinct variables.

```
p bdd 56 97
```

9.1.3 Boolean functions

A Boolean function may be expressed two ways:

```
<variable>
<function-identifier> arg1 arg2 ...
```

where `<function-identifier>` is one of the following predefined function identifiers, and `arg#` is either a variable, a Boolean function or a reference to a Boolean function. The token `<variable>` is a character string and its use in an input file creates a simple Boolean function which is identified by that name, the value of which is the value of the variable named `<variable>`. Every Boolean function is implemented as a BDD (see Section 10.1) and may have value T, or F, or have no value, depending on its arguments: values of Boolean functions are given as follows for each `<function-identifier>`.

Remark: Each function takes a fixed number of arguments, hence, commas and parenthesis are considered whitespace and are ignored, however, their use is recommended to enhance the human readability of input files.

VAR

Variables may be defined as Boolean functions using `VAR` followed by a single integer argument, which identifies the created variable. The use of `VAR` is now deprecated and its use is discouraged but it remains supported (for now) for the benefit of early users. It is now assumed that all unknown valid character strings, not previously defined, and unless preceded by a '$', be defined as single variable Boolean functions.. All integers preceded by a '$' refer to previously defined functions or manipulators instead of variables.

NOT

The unary Boolean function `NOT` has a value equal to that of the complement of its single argument. A variable may also be complemented by using '`-`' in place of `NOT` (functions cannot be complemented in this way). Examples are:

```
not(x32)
-x32
```

AND

The binary Boolean function `AND` has value T if both its arguments have value T and has value F if at least one of its arguments has value F. Examples are the following:

```
and(x3, x4)
and x3 x4
```

NAND

The binary Boolean function `NAND` has value F if both its arguments have value T and has value T if at least one of its arguments has value F. An example is:

```
nand(x3, x4)
```

OR

This binary Boolean function has value T if one of its arguments has value T and has value F if both its arguments have value F. An example is:

```
or(and(x3, x4), x5)
```

NOR

This binary Boolean function has value F if at least one of its arguments has value T and has value T if both arguments have value F. An example is:

```
nor(x3, ite(a, b, -c))
```

EQU

This binary Boolean function has value T if both its arguments have been assigned the same value and has value F if both its arguments have been assigned different values. In the following example `equ` acts as a way to assign the value T to variable `x4`.

```
equ(T, x4)
```

XOR

This binary Boolean function has value T if its two arguments have been assigned opposite values and has value F if its two arguments have been assigned the same value. An example is:

```
xor(x3, x4)
```

This is functionally equivalent to:

```
equ(x3, not(x4))
```

IMP

This binary Boolean function has value T if either both its arguments have value T or if its second argument has value F and has value F if its first argument has value F and second value T. An example is:

```
imp(x3, x4)
```

NIMP

This binary Boolean function has value F if either both its arguments have value T or if its second argument has value F and has value T if its first argument has value F and second value T. An example is:

```
nimp(x3, x4)
```

ITE

The ternary Boolean function `ITE` has the value of its second argument if its first argument has value T, and has the value of its third argument if its first argument has value F. Examples of this function are the following:

```
ite(ite( x3, $5, -x5), x4, x7)
ite ite x3 $5 -x5 x4 x7
```

9.1.4 Boolean function extensions

The functions of this section are extensions to the binary functions of the previous section in that they admit more than two arguments. The syntax is the same except that the `<function-identifier>` is immediately followed by a number (≥ 2) indicating the number of arguments the function has. Below, the character `#` is used to indicate the position of this number when defining function names.

AND#

This Boolean function has value T if all its arguments have value T and has value F if at least one argument has value F. Examples of its use are as follows:

```
and4(x1, x2, x3, x4)
and5(-x1,
     and3(x2, x3, $4),
     x5, $6, x7)
```

NAND#

This Boolean function has value F if all its arguments have value T and has value T if at least one argument has value F.

OR#

This is an extension to `OR` analogous to the extension `AND#` of `AND`.

NOR#

This is an extension to `NOR` analogous to the extension `NAND#` of `NAND`.

EQU#

This Boolean function has value T if an even number of its arguments have value F and has the value F otherwise.

XOR#

This Boolean function has value F if an even number of its arguments have value T and has the value T otherwise.

9.1.5 Manipulators

GCF

The syntax for this manipulator is

```
gcf <function> <cofactor>
```

where `<cofactor>` and `<function>` are both Boolean functions or variables. The Boolean function returned by `gcf` is the Generalized Cofactor of `<function>` with respect to `<cofactor>`. See Section 10.3 for a description of generalized cofactor and when it might be useful.

STRENGTHEN

The syntax for this manipulator is

```
strengthen <function>_1 <function>_2
```

where `<function>_1` and `<function>_2` are both Boolean functions or variables. The Boolean function returned by this manipulator is `<function>_1` strengthened by `<function>_2`. To find out more about strengthening see Section 10.5.

PRUNE

The syntax for this manipulator is

```
prune <function>_1 <function>_2
```

where <function>_1 and <function>_2 are both Boolean functions or variables. The Boolean function returned by this manipulator is <function>_1 with branches overlapping <function>_2 pruned. That is, if t is an assignment satisfying <function>_2 but not <function>_1, then t also satisfies the function returned by prune. To find out more about branch pruning see Section 10.4.

EXIST

The syntax for this manipulator is

```
exist <function> <variable>
```

where <function> is a Boolean function and <variable> is a Boolean variable that <function> depends on. This manipulator existentially quantifies <variable> out of <function>. That is, let $f|_T$ be <function> with <variable> set to T and let $f|_F$ be <function> with <variable> set to F; then the Boolean function returned is $f|_T \vee f|_F$. For more information about existentially quantifying away variables see Section 10.7.

UNIVERSE

The syntax for this manipulator is

```
universe <function> <variable>
```

where <function> is a Boolean function and <variable> is a Boolean variable that <function> depends on. This manipulator universally quantifies <variable> out of <function>. That is, let $f|_T$ be <function> with <variable> set to T and let $f|_F$ be <function> with <variable> set to F; then the Boolean function returned is $f|_T \wedge f|_F$. For more information about universally quantifying away variables see Section 10.13.

TRUTH_TABLE

Functions can be defined by their truth table using TRUTH_TABLE. Syntax for this manipulator is

```
truth_table <#vars> <var> ...  <var> <TF-string>
```

where <var> are variables, <#vars> is the number of variables in the variable argument list, and <TF-string> is a string of symbols 'T' and 'F' which defines the truth table. There must be exactly $2^{<\#\texttt{vars}>}$ symbols in the <TF-string>. An example is this:

```
truth_table 3 x1 x2 x3 TTFFTFFT
```

MINMAX

The syntax of this function is as follows:

```
minmax <#args> <min_#> <max_#> <blank_separated_var_list>
```

where the first three arguments are positive integers. This function returns T if at least <min_#> and at most <max_#> of the variables in the <..._var_list> have value T and returns F if more than <max_#> or less than <min_#> of the variables in the <..._var_list> have value T. The number <#args> is the number of variables in the <..._var_list>. The following is an example,

```
minmax(4, 1, 3, x13, x12, x11, x10)
```

which has value T if at least 1 and no more than 3 of the variables x10, x11, x12, x13 have value T. See the PRINT_TREE directive below to view the truth table for this function.

SAFE

The syntax of this function is as follows:

```
safe <function> <variable>
```

This function returns either a safe assigment for <variable> in <function> if one exists. The Boolean function returned is either equ(<variable>, T), equ(<variable>, F), T (if no safe assignment exists), or F (if <variable> is safe for both truth values T and F). For more information about safe assignments see Section **??**.

9.1.6 Directives

The syntax of a directive is

```
<command> <arg>, ...
```

where `<arg>` are Boolean functions or manipulators or variables. The following are commands and a description of how they apply their arguments.

INITIAL_BRANCH

The directive `INITIAL_BRANCH` may be invoked to request that certain variables be assigned values before any others during search. This directive is also capable of influencing which branch (the *True* branch or the *False* branch) will be tested first, for specified variables, during search. This directive takes as argument a list of variables, wrapped between parentheses. The list may be of any length and may be continued over many lines. Variables in this list may optionally be postpended with a percent sign followed immediately by any real number between 0 and 100. The value following a percent sign will be factored into all the search heursitics embedded into `sbsat` such that a value close to 0 influences the heursitic to more often assign *False* to the prepended variable, likewise for values close to 100; a value of exactly 50 has no effect on the heuristic. This directive can also take an optional first argument, a pound sign (`#` followed by a positive number. This argument determines the tier used to determine the order in which whole set of variables will be assigned values during search; smaller numbers are given precedence. This directive is intended to be used when some key domain-specific information suggesting a particular search order has been revealed. An example of this directive is the following:

```
initial_branch (x1, x39, x5, x4, x24, x3)
```

This marks variables x_1, x_{39}, x_5, x_4, x_{24}, x_3 to be assigned values first. The order in which these variables are selected for search completely depends on the search heuristic employed: all that `initial_branch` does here is prevent variables which are not listed from being selected for assignment during search until all the listed variables have values. It is possible for a non-listed variable to be inferred, however. This directive is the only one which requires parenthesis around its argument list. Wildcards (standard regular expressions) are allowed: '`*`' matches any combination of alphabetic, numeric, and underscore characters. Thus, `initial_branch(x3*3)` means branch on all variables beginning with `x3` and ending with `3`.

The order in which wild card variables are searched is determined by the search heuristic. There can be multiple `initial_branch` directives in one. An example of this is the following:

```
initial_branch (#1, x1, x39%20.5, x5)
initial_branch (#2, x4, x24, x3%80.25)
```

This marks variables x_1, x_{39}, and x_5, to be assigned values first and variables x_4, x_{24}, x_3 to be assigned values second; all variables not listed will be considered by the heuristic after those appearing in `initial_branch` have been assigned values. Also, the heuristic value of variable x_{39} will be influenced such that it is more likely be assigned the value *False*, likewise x_3 is more likely be assigned the value *True*.

#DEFINE

A rudimentary macro facility. The syntax is the following:

```
#define <pattern> # <Boolean-function>
```

where `<pattern>` is a `<function-identifier>` and comma separated argument list enclosed in parentheses. Wherever the `<pattern>` is matched in the input file, the `<function-specifier>` is substituted with arguments corresponding to those used in the `<pattern>`. All the `<pattern>` arguments must be used in the `<function-specifier>` and all the parameters used in `<function-specifier>` must be arguments in `<pattern>`. For example,

```
#define slide(x1, x17, x15, x33, x40)
#       equ(xor(x1, and(-x17, x33),ite(x15, or(x33, -x40), -x33)))
```

substitutes the `equ...` function for `slide...`. See Figure 9, Page 19 for the completion of this example. All `#define` macros are effective *after* they are defined in the file. All functions, including built-in functions, may be redefined using `#define`. Thus,

```
#define and3(x, y, z) # or3(x, y, z)
```

causes three input `and3` functions occurring *after* the above `#define` to behave as three input `or3` functions. A line such as

```
#define and3(x, y, z) # and(and(x, y), z)
```

redefines `and3` to what it had been initially (one cannot use `and3` in the right argument because it has already been redefined) and all `and3` functions *after* that `#define` behave as `and3` functions. At no point in the file can there exist two macros with the same `<function-identifier>` but different arguments: a `#define` always replaces previous `#define`s with an identical `<function-identifier>`. For example, after the following:

```
#define and(x, y, z) # or3(x, y, z)
#define and(x, y, z, w) # or4(x, y, z, w)
```

The function `and(x, y, z, w)` is defined but `and(x, y, z)` is not.

PRINT_TREE

The PRINT_TREE function takes one Boolean function, extension or manipulator as argument and prints an unfolded BDD representation (tree form) of that function, top down, to standard output. All function prints are displayed before the solution. In the printed tree, the left branch is the T branch and the right branch is the F branch. For example,

```
print_tree(or3 (4, 5, -6))
```

prints the following to standard output:

```
------------------------------------------------------------
                          6
              5                           T
      T               4
                T       F
------------------------------------------------------------
```

It is possible to control the variable ordering of the printed trees. Variables are ordered based on their where they first occur in the input file. The ordering can be controlled by creating a dummy define statement, for example:

```
#define ordering(a, b, c, d) # or4(a, b, c, d)
```

This forces variable a to occur as or near the leaves of printed trees, and d to occur as or near the root. For example, the lines

```
#define ordering(b, a, d, c) # or4(b, a, d, c)
print_tree(minmax(4, 1, 3, a, b, c, d))
```

prints the following to standard output (assuming a, b, c, and d have not been previously defined):

```
------------------------------------------------------------
                        c
              d                   d
        a           T       T           a
     b      T                        T      b
  F    T                                 T    F
------------------------------------------------------------
```

PPRINT_TREE

Does the same thing as `PRINT_TREE` except the output is in text form instead of graphical form with `ite` used to indicate T and F branches. For example,

```
pprint_tree(or3 (4, 5, -6))
```

prints to standard output:

```
------------------------------------------------------
ite 6
   ite 5
      T
      ite 4 T F
   T
------------------------------------------------------
```

PPRINT_XDD

Does a similar thing to `PRINT_TREE` except the output is shown in terms of nested ANDs and XORs. For example,

```
print_xdd(or3 (4, 5, -6))
```

prints to standard output:

```
------------------------------------------------------------------
x[6]*(x[5]*(x[4] + 1) + x[4] + 1) + 1 = 1
------------------------------------------------------------------
```

PPRINT_FLAT_XDD

Does the same thing as `PRINT_XDD` except the function is displayed in its unnested form. For example,

```
print_flat_xdd(or3 (4, 5, -6))
```

prints to standard output:

```
------------------------------------------------------------------------
x[4]*x[5]*x[6] + x[5]*x[6] + x[4]*x[6] + x[6] = 0
------------------------------------------------------------------------
```

9.2 CNF format

This input format is only for expressing Conjunctive Normal Form or "product of sums" formulas. Variables are represented as positive integers. All top-level Boolean functions are disjunctions or clauses. There is one clause per line of file. A solution to the problem defined in a file is an assignment satisfying all clauses. There are no manipulators or directives in this format. This format is well-known as the DIMACS format[10].

9.2.1 Comments

A comment begins with the special character '`c`'.

9.2.2 File Header

The header conforms to the DIMACS standard which is as follows:

```
p cnf <#vars> <#clauses>
```

The letter '`p`' must appear at the start of the header. The second header token, which in this case is '`cnf`', identifies the input format. The header contains two integer fields: `<#vars>` is the number of distinct variables contained in the file; `<#clauses>` is the number of clauses contained in the file. The following is an example of a valid header for a file containing 97 clauses composed from 56 distinct variables.

```
p cnf 56 97
```

[10] See `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi` for a complete description.

9.2.3 Boolean functions

The only Boolean functions accepted in this format are clauses. A clause has the following syntax:

```
[-]<var> ... [-]<var> 0
```

where `<var>` is a variable and `'-'` in front of a `<var>` makes it a negative literal and no `'-'` in front of `<var>` makes it a positive literal. There are no commas in the variable list and the list is terminated with a 0. Variables may appear in a variable list in any order and are separated by blank characters. For an example see Figure 3, Page 13.

9.3 DNF format

This input format is only for expressing Disjunctive Normal Form or "sum of products" formulas. Variables are represented as positive integers. All top-level Boolean functions are conjunctions, called terms. There is one term per line of file. A solution to the problem defined in a file is an assignment satisfying at least one term. There are no manipulators or directives in this format. This format is similar to the well-known DIMACS format[11]. Internally, DNF problems are transformed to CNF then solved as CNF problems.

9.3.1 Comments

A comment begins with the special character '`c`'.

9.3.2 File Header

The header conforms to the DIMACS standard which is as follows:

```
p dnf <#vars> <#terms>
```

The letter '`p`' must appear at the start of the header. The second header token, which in this case is '`dnf`', identifies the input format. The header contains two integer fields: `<#vars>` is the number of distinct variables contained in the file; `<#terms>` is the number of terms contained in the file. The following is an example of a valid header for a file containing 97 terms composed from 56

distinct variables.

```
p dnf 56 97
```

9.3.3 Boolean functions

The only Boolean functions accepted in this format are conjunctions of literals, called terms. A term has the following syntax:

```
[-]<var> ... [-]<var> 0
```

where `<var>` is a variable and `'-'` in front of a `<var>` makes it a negative literal and no `'-'` in front of `<var>` makes it a positive literal. There are no commas in the variable list and the list is terminated with a 0. Variables may appear in a variable list in any order and are separated by blank characters.

9.4 XOR format

This input format is intended for specialized applications involving constraints that are exclusive-ors of conjunctions. Variables are represented as positive integers preceeded by the character `x` as in `x45`. All top-level Boolean functions are xors of conjunctions. A solution to the problem defined in a file is an assignment satisfying all functions. There are no manipulators or directives in this format.

9.4.1 Comments

A comment begins with the special character '`;`'.

9.4.2 File Header

The header conforms to the DIMACS standard which is as follows:

```
p xor <max-var-number> <#functions>
```

The letter '`p`' must appear at the start of the header. The second header token, which in this case is '`xor`', identifies the input format. The header contains two integer fields: `<max-var-number>` is the maximum variable number contained in the file; `<#functions>` is the number of Boolean functions contained in the file.

The following is an example of a valid header for a file containing 97 functions composed from variables `x1` to `x56`.

```
p xor 56 97
```

9.4.3 Boolean functions

Each line of the file is a function. Each line has the following form:

```
<grouping> ...  <grouping> = [0|1]
```

where `<grouping>` has the form

```
<variable>...<variable>
```

and `<variable>` has the form

```
x<number>
```

where `<number>` is a positive integer no greater than `<max-var-number>`. The following are examples:

```
x1 x2 x3 = 1
x1x2x3 = 1
x1x2x3 x2x3 x4x5x6 = 0
```

`sbsat` interprets a line as an `xor` of conjunctions, each consisting of variables identified in a `<grouping>`. The following table shows the above as expressions in canonical form:

This ...	is equivalent to this
`x1 x2 x3 = 1`	`equ( xor3(1, 2, 3), T)`
`x1x2x3 = 1`	`equ( and3(1, 2, 3), T)`
`x1x2x3 x2x3 x4x5x6 = 0`	`equ(xor3(and3(1, 2, 3), and(2, 3), and3(4, 5, 6)), F)`

```
MODULE <name>
INPUT <var>, ..., <var>;
OUTPUT <var>, ..., <var>;
STRUCTURE
<statement>;
...
<statemnt>;
ENDMODULE
```

Figure 18: Trace format specification

9.5 Trace format

Trace files follow a format inspired by the `dlx` processor verification examples made available by Carnegie Mellon University. A trace file has three sections: input, output, and structure. Format for the entire file is shown in Figure 18. The field `<name>` is any string of contiguous characters naming the module. We regard the input and output sections as the file header and the structure section as the place where functions are specified. The three sections are described below. There are no directives or manipulators in this format.

9.5.1 Comments

A comment begins with the special character ';'. This is a departure from the use of ';' in the actual CMU trace format.

9.5.2 File Header

The file header consists of the input and output sections. Both sections list variables which appear in functions specified in the structure section. Specification of these sections is shown in Figure 18. The field `<var>` is the name of a variable and uses any combination of alphabetic and numeric characters and underscore. A variable list may be continued over several lines. Its terminating character is ';'.

9.5.3 Boolean functions

Boolean functions are specified in the structure section which consists of any number of statements (<statement> in Figure 18), each terminated by ';'. The keywords STRUCTURE and ENDMODULE must appear by themselves on separate lines. A statement is either

```
are_equal(<var>, <var>)
```

which forces both <var>s to have the same value, or

```
<var> = new_int_leaf([0|1])
```

which forces <var> to have value *F* or *T* if the argument of new_int_leaf is *0* or *1*, respectively, or

```
<var> = <function-identifier>(<var>, ..., <var>)
```

where <function-identifier> is one of not, and, nand, or, nor, equ, xor, imp, limp, lnimp, rimp, rnimp, ite. The variable lists of all but ite and not can be arbitrarily long. The argument lists for ite and not must have exactly three and exactly one argument(s), respectively. Observe there is no nesting of functions as in the case of the canonical form. Instead, an equality is defined and the leftside (temporary) variable, not appearing in either input or output section, is used as argument in other functions. It is permissable to reference a variable that appears for the first time further ahead in the file. A solution is an assignment to input variables which causes all functions (statements) to have value *T*.

9.6 Prove format

See Section 15 for details.

9.7 SMURF format

This input format is intended for low-level truth table input. Variables are represented as positive integers. Any set of Boolean functions can be accomodated. Each function is specified in a section of three lines: the first line numbers the function, the second line specifies the variables that the function depends on, the third line specifies the truth table of the function or uses function symbols

to denote commonly used functions. All function sections are separated by a hash ('#') character, which is itself on a separate line. A list of functions is terminated with the character '@' on a line by itself at the end of the file. The header contains a line indicating a goal value for each function. A solution to the problem defined in a file is an assignment causing all Boolean functions to attain their goal value. There are no comments, manipulators, or directives in this format.

9.7.1 File Header

The file header consists of three lines. The first line begins with a number equal to the number of input variables. On the second line is the number of functions (that is, function sections), on the fourth line is a *0-1* vector, represented as a string of *0*'s and *1*'s, which specifies goal values for each of the functions. An example is the following:

```
5 # Number of Input Variables
6 # Number of Functions
110101 # Output values
```

for a file with variables numbered 0-4 and 6 output functions.

9.7.2 Boolean functions

Each Boolean function is represented by a three line section of the input file. All function sections are separated by the character '#'. A '#' also separates the header from the first function section. The format of a function section is the following:

```
<number>
<var> ...  <var> -1
[ <truth_table> | <function_identifier> <polarity_list> ]
```

The first line of each function section is a number which is the identity of the function, or function number. Typically, function numbers are assigned in the order the functions appear in the file, beginning with 1. The second line of a function section is a blank separated list of variables, terminated with -1. The third line may be either a truth table or a function identifier and polarity list.

One of two possible formats for the third line is a truth table. A truth table is a string of *0*'s and *1*'s, the number of which must be $2^{\#inp}$ where $\#inp$ is the number of variables on the second line of the function section. Each *0-1* character represents the function's value given a particular assignment of inputs values: the ith *0-1* character in the truth table string (counting from the left starting with 0) is the function value given an input assignment matching the bits of the binary representation for i where the *least* significant bit corresponds to the value assigned to the *leftmost* variable of the second line and a bit value of *0* (*1*) represents an assignment of F (T). An example of a function section with a truth table is the following:

```
2
8 9 2 4 -1
1001011011110000
```

This function has a value F if, for example, variable 8 has value T and variables 2,9, and 4 have value F. This function has value T if, for example, variables 2 and 8 have value T and variables 4 and 9 have value F.

Another possible format for the third line of a function section is the specification of a function identifier and polarity list. The intention of this format is to allow compact specification of commonly used functions (especially in circuit problems) with quite a few arguments that would otherwise require extremely large truth tables. Function identifiers accepted are: `and=`, `or=`, and `plainor`. The identifier `plainor` correspond to the "or" function described in Section 2.2. Identifiers with `=` equate a single variable on the left of the `=` with a simple expression on the right of the type indicated by the identifier. For example, `and=` corresponds to a function of the form:

```
<var> = and(<var>, ...  <var>)
```

This function has value T if and only if the value of the variable on the left of `=` is the same as the logical "and" of the variables on the right of `=`. For identifiers with `=`, the polarity list is a string of characters from the set $\{0, 1, 3\}$ with exactly one *3*. The number of characters is the number of variables identified on the second line of the function section and each character corresponds to an input variable: the leftmost character corresponding to the leftmost input variable. The '*3*' identifies the input variable that is on the left side of `=`. The remaining *0*'s and *1*'s determine the polarity of the variables on the right side

```
4 # number of variables
4 # number of functions
1101 # output vector
#
0
1 2 3 -1
00110110
#
1
1 2 4 -1
and= 130
#
2
2 3 4 -1
or= 311
#
3
1 2 3 -1
11100010
@
```

Figure 19: A SMURF formatted file

of =. An example of a function section with function identifier containing = is the following:

```
41
4 11 12 186 187 188 189 193 382 -1
and= 011000031
```

This would be identical to the following:

```
193 = and(-4, 11, 12, -186, -187, -188, -189, 382)
```

If this function were to have a truth table instead of `and=`, the truth table would have 512 characters. A similar description applies to the polarity list of function identifiers with = except that no '*3*' exists in such lists. Figure 19 shows an example of a SMURF formatted file: for the problem depicted the assignment of T to variable 2 and F to all other variables is a solution.

10 Reference - Preprocessing

Top-level Boolean functions are expressed internally as Binary Decision Diagrams or BDDs. This allows the use of a number of new and old BDD operations to be used to, in some sense, reduce the complexity of an input problem before applying search. A simple description of BDDs is given and that is followed by a description of the preprocessing operations made available for input problem reduction.

10.1 Binary Decision Diagrams (BDDs)

A Binary Decision Diagram (BDD) is a rooted, directed acyclic graph. A BDD is used to compactly represent the truth table, and therefore complete functional description, of a Boolean function. Vertices of a BDD are called *terminal* if they have no outgoing edges and are called *non-terminal* otherwise. There is one non-terminal vertex, called the *root*, which has no incoming edge. There is at least one and there are at most two terminal vertices, one labeled *0* and one labeled *1*. Non-terminal vertices are labeled to represent the variables of the corresponding Boolean function. A non-terminal has exactly two outgoing edges, labeled T and/or F, and the vertices incident to edges outgoing from vertex v are called $true(v)$ and $false(v)$, respectively. Associated with any non-terminal vertex v is an attribute called $index(v)$ which satisfies the properties $index(v) > \max\{index(true(v)), index(false(v))\}$ and $index(v) = index(w)$ if and only if vertices v and w have the same labeling (that is, correspond to the same variable). Thus, the *index* attribute imposes a linear ordering on the variables of the BDD.

A Reduced Ordered Binary Decision Diagram (ROBDD) is a BDD such that: 1) There is no vertex v such that $true(v) = false(v)$; 2) The subgraphs of two distinct vertices v and w are not isomorphic. A ROBDD represents a Boolean function uniquely in the following way. Define $f(v)$, v a vertex of the ROBDD, recursively as follows:

1. If v is the terminal vertex labeled F, then $f(v) = F$;
2. If v is the terminal vertex labeled T, then $f(v) = T$;
3. Otherwise, if v is labeled x, then $f(v) = (x \wedge f(true(v))) \vee (\neg x \wedge f(false(v)))$.

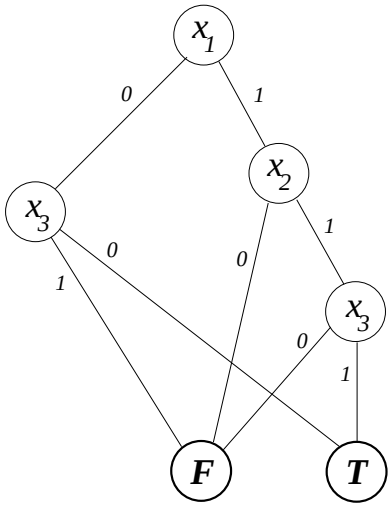


Figure 20: A BDD representing $(x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$. The topmost vertex is the root. The two bottom vertices are terminal vertices. Edges are directed from upper vertices to lower vertices. Vertex labels (variable names) are shown inside the vertices. The *true* branch out of a vertex is identified with *0*. The *false* branch is identified with *1*. The *index* of a vertex is, in this case, the subscript of the variable labeling that vertex.

Then $f(root(v))$ is the function represented by the ROBDD. Observe that a Boolean function has different ROBDD representations, depending on the variable order imposed by *index*, but there is only one ROBDD for each ordering. Thus, ROBDDs are known as a canonical representation of Boolean functions. Observe also that a path from root to terminal in a ROBDD corresponds to one or more rows of a truth table associated with the function represented by the ROBDD: the labels of the vertices encountered on the path are the variables and their assigned values are determined from the outgoing edges traversed, the assignment being T (F) if the *true* (*false*) edge is taken, respectively. The collection of all paths specifies the truth table completely. Although ROBDDs are actually used internally in SBSAT, they are referred to as BDDs in this manual.

Figure 20 shows an example of a BDD and the function it represents. See the data structures section for details on how the BDDs are implemented in SBSAT. The following are some simple BDD operations that are used by preprocessing operations which are described in subsequent sections.

A BDD is constructed by attaching BDDs h_T and h_F, representing a *true*

and a *false* branch, respectively, to a vertex v with some labeling x representing the root. We may think of the operation to do this as being the following, in pseudo C++ style:

```
BDD ite(variable x, BDD h_T, BDD h_F);
```

That is, `ite` returns a BDD with root v labeled x and such that $h_T = true(v)$ and $h_F = false(v)$. But the actual construction is such as to avoid building BDDs which are isomorphic to existing ones, so the following is used to implement the construction instead (it is too complicated to state here):

```
BDD find_or_add_node (variable x, BDD h_T, BDD h_F);
```

This operation returns an existing BDD if there is one that matches `ite(`x`,` h_T`,` h_F`)` already, and otherwise builds a new BDD with root v labeled x, *true* branch h_T and *false* branch h_F (that is, $false(v) = h_F$ and $true(v) = h_T$). The BDDs h_T and/or h_F may have to be constructed as well.

The following two simple operations are used several times in describing important BDD operations in subsequent sections. They are given in pseudo C++ style:

```
BDD Reduce_T (variable x, BDD f) {
    if (root(f) == x) return true(root(f));
    return f;
}

BDD Reduce_F (variable x, BDD f) {
    if (root(f) == x) return false(root(f));
    return f;
}
```

`Reduce`$_T$`(`x`,`f`)` returns f constrained by the assignment of T to variable x and `Reduce`$_F$`(`x`,`f`)` returns f constrained by the assignment of F to the variable x.

10.2 Pattern Matching: CNF

The current version of `sbsat` supports clustering only when CNF input is given. Our clustering algorithm is influenced solely by observing patterns in CNF formulas due to the `dlx` benchmarks from CMU. Those benchmarks, before translation to CNF, consist of numerous lines almost all of which have a form similar

to one of the following:

$$x = and(v_1, v_2, ..., v_k)$$
$$x = or(v_1, v_2, ..., v_k)$$
$$x = ite(v_1, v_2, v_3)$$

A pass is made through all clauses of a given CNF formula looking for patterns similar to the following:

$$(v_1 \vee \bar{v}_2 \vee \bar{v}_3 \vee ... \vee \bar{v}_k)$$
$$(\bar{v}_1 \vee v_2) \quad (\bar{v}_1 \vee v_3) \quad ... \quad (\bar{v}_1 \vee v_k)$$

which in this case represents the first of the three expressions above. Clauses equivalent to the second expression are similar (one large clause and several binary clauses) differing only in which literals are negated. If a set of clauses matching the form above is found, then those clauses are replaced by a single BDD representing the corresponding $x = and(...)$ or $x = or(...)$ expression. In the case of the $ite(...)$ expression a pattern of six clauses of the following form:

$$(v_1 \vee v_2 \vee \bar{v}_4) \quad (v_1 \vee \bar{v}_2 \vee \bar{v}_3) \quad (\bar{v}_1 \vee \bar{v}_2 \vee v_3) \quad (\bar{v}_1 \vee v_2 \vee v_4)$$

is replaced by a BDD representing the third expression above. In addition, if such a pattern is found the following two clauses may also be removed from the original CNF formula during the clustering operation without consequence:

$$(v_1 \vee \bar{v}_3 \vee \bar{v}_4) \quad (\bar{v}_1 \vee v_3 \vee v_4)$$

Any BDD constructed in this way is marked with a special function identifier so that the corresponding SMURF will be smaller than otherwise.

10.3 Generalized Cofactor (GCF)

The `generalized cofactor` operation, denoted by `gcf` here and also known as `constrain` in the literature, uses *sibling substitution* to reduce BDD size. However, unlike `Prune`, it can produce BDDs far larger than the original. Given two functions, f and c, the function $g = \texttt{gcf}(f, c)$ is such that $f \wedge c$ is the same as $g \wedge c$. In the process, g may be somehow reduced compared to f as is the case

for `Prune`. Unlike `Prune`, the following is true as well:

> Given Boolean functions $f_1, ..., f_k$, for any $1 \leq i \leq k$, $f_1 \wedge f_2 \wedge ... \wedge f_k$ is satisfiable if and only if $\texttt{gcf}(f_1, f_i) \wedge ... \wedge \texttt{gcf}(f_{i-1}, f_i) \wedge \texttt{gcf}(f_{i+1}, f_i) \wedge ... \wedge \texttt{gcf}(f_k, f_i)$ is satisfiable. Moreover, any assignment satisfying the latter can be extended to satisfy $f_1 \wedge ... \wedge f_k$.

This means that, for the purposes of a solver, `generalized cofactoring` can be used to eliminate one of the BDDs among a given conjoined set of BDDs: the solver finds an assignment satisfying $\texttt{gcf}(f_1, f_i) \wedge ... \wedge \texttt{gcf}(f_k, f_i)$ and then extends the assignment to satisfy f_i, otherwise the solver reports that the instance has no solution. However, unlike `Prune`, generalized cofactoring cannot by itself reduce the number of variables in a given collection of BDDs. Other properties of the `gcf` operation are:

1. $f = c \wedge \texttt{gcf}(f, c) \vee \neg c \wedge \texttt{gcf}(f, \neg c)$.
2. $\texttt{gcf}(\texttt{gcf}(f, g), c) = \texttt{gcf}(f, g \wedge c)$.
3. $\texttt{gcf}(f \wedge g, c) = \texttt{gcf}(f, c) \wedge \texttt{gcf}(g, c)$.
4. $\texttt{gcf}(f \wedge c, c) = \texttt{gcf}(f, c)$.
5. $\texttt{gcf}(f \wedge g, c) = \texttt{gcf}(f, c) \wedge \texttt{gcf}(g, c)$.
6. $\texttt{gcf}(f \vee g, c) = \texttt{gcf}(f, c) \vee \texttt{gcf}(g, c)$.
7. $\texttt{gcf}(f \vee \neg c, c) = \texttt{gcf}(f, c)$.
8. $\texttt{gcf}(\neg f, c) = \neg\, \texttt{gcf}(f, c)$.
9. If c and f have no variables in common and c is satisfiable then $\texttt{gcf}(f, c) = f$.

Care must be taken when cofactoring in "both" directions (exchanging f for c). For example, $f \wedge g \wedge h$ cannot be replaced by $\texttt{gcf}(g, f) \wedge \texttt{gcf}(f, g) \wedge h$ since the former may be unsatisfiable when the latter is satisfiable.

The pseudo C++ description of `gcf` is as follows:

```
BDD gcf (BDD f, BDD c) {
    if (f == F || c == F) return F;
```

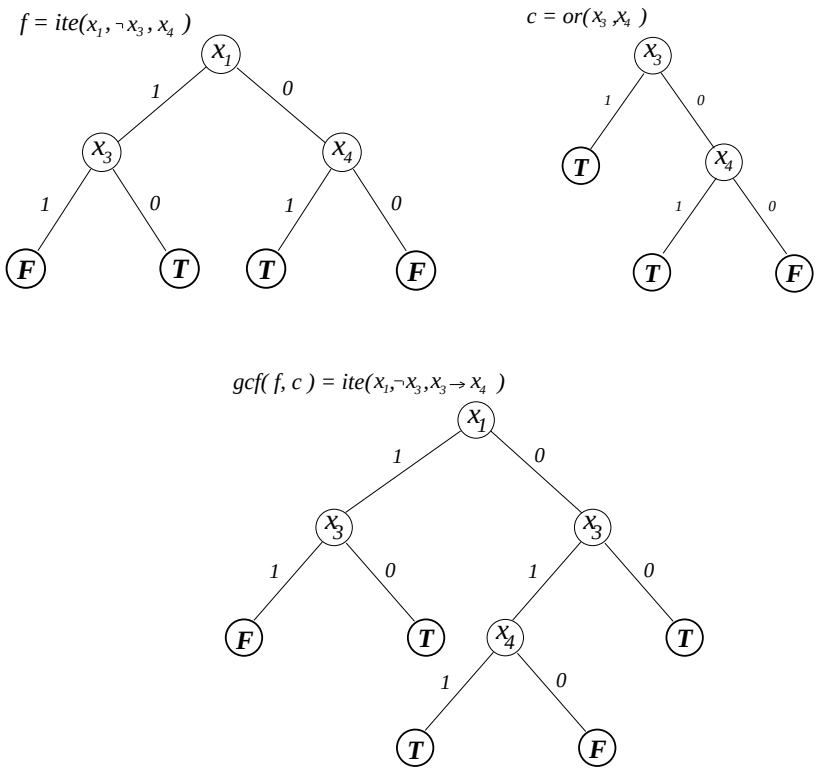


Figure 21: Generalized cofactor operation on f and c as shown. In this case the result is more complicated than f. The variable ordering is x_1, x_2, x_3, x_4.

```
    if (c == T || f == T) return f;
    let x_m = index^{-1}(min{index(root(c)), index(root(f))});
    // x_m is the top variable of f and c
    if (Reduce_F(x_m, c) == F) return gcf(Reduce_T(x_m, f), Reduce_T(x_m, c));
    if (Reduce_T(x_m, c) == F) return gcf(Reduce_F(x_c, f), Reduce_F(x_c, c));
    let h_T = gcf(Reduce_T(x_m, f), Reduce_T(x_m, c));
    let h_F = gcf(Reduce_F(x_m, f), Reduce_F(x_m, c));
    if (h_T == h_F) return h_T;
    return find_or_add_node(x_m, h_T, h_F);
}
```

Figure 21 presents an example of its use illustrating the possibility of increasing BDD size. Figure 22 presents the same example after swapping x_1 and x_4 under

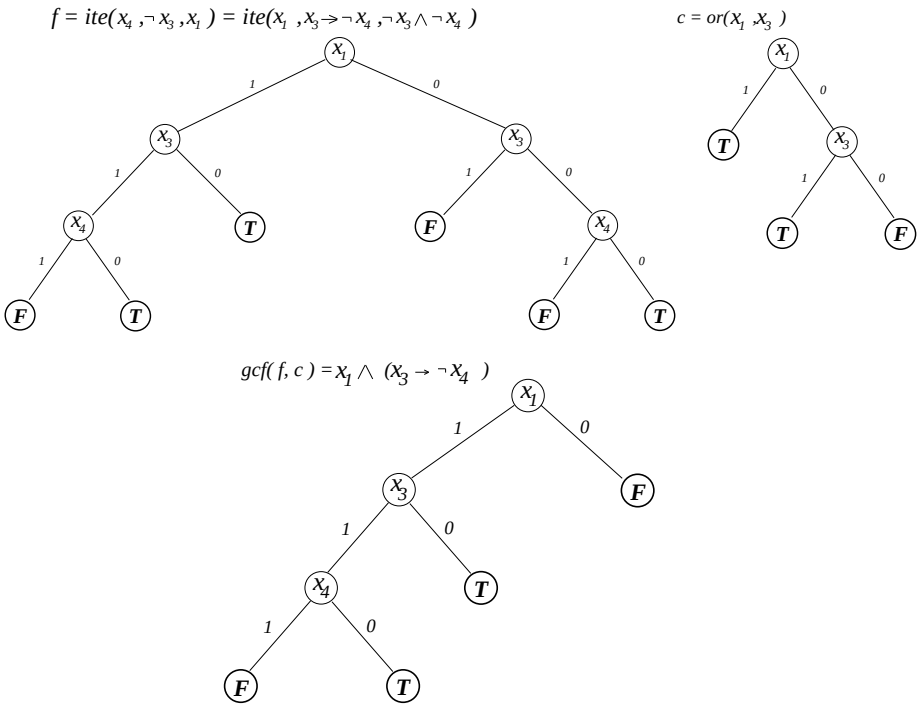


Figure 22: Generalized cofactor operation on the same f and c of Figure 21 and with the same variable ordering but with x_1 and x_4 swapped. In this case the result is less complicated than f.

the same variable ordering and shows that result produced by `gcf` is sensitive to variable ordering. Observe that the functions produced by `gcf` in both Figures have different values under the assignment $x_1 = T$, $x_3 = T$, and $x_4 = F$. Thus, the function returned by `gcf` depends on the variable ordering as well.

10.4 Branch Pruning

Branch pruning is an operation on two BDDs. The intention is to remove paths from one BDD which are made irrelevant by the other BDD. The following specifies how this is done in pseudo-C++ style:

```
BDD Prune (BDD f, BDD c) {
    if (c == T || f == F || f == T) return f;
    if (c == ¬f) return F;
    if (c == f) return T;
```

```
    // f and c have a non-trivial relationship
    let x_f = root(f); // x_f is a variable
    let x_c = root(c); // x_c is a variable
    if (index(x_f) > index(x_c)) return Prune(f, ExQuant(c, x_c));
    if (Reduce_F(x_f, c) == F) return Prune(Reduce_T(x_f, f), Reduce_T(x_f, c));
    if (Reduce_T(x_f, c) == F) return Prune(Reduce_F(x_f, f), Reduce_F(x_f, c));
    let h_fT = Prune(Reduce_T(x_f, f), Reduce_T(x_f, c)); // h_fT is a BDD
    let h_fF = Prune(Reduce_F(x_f, f), Reduce_F(x_f, c)); // h_fF is a BDD
    if (h_fT == h_fF) return h_fT;
    return find_or_add_node(x_f, h_fT, h_fF);
}
```

The procedure *Prune* takes two BDDs which are top-level functions as input and returns a BDD which can replace the BDD of argument f at the top-level. Figure 23 shows an example.

Branch pruning can reveal inferences but this depends on the variable ordering. Figure 24 shows `Prune` applied to two BDDs with no result. BDDs representing the same two functions but under a different variable ordering are pruned in Figure 25 revealing the inference $x_3 = F$.

Branch pruning is similar to a procedure called generalized cofactor or constrain (see Section 10.3 for a description). Both $\texttt{Prune}(f, c)$ and $\texttt{gcf}(f, c)$ agree with f *on interpretations where* c *is satisfied,* but are generally somehow simpler than f. Both are highly dependent upon variable ordering, so both might be considered "non-logical." Branch pruning is implemented in SBSAT because the BDDs produced from it tend to be smaller. In any case, unlike for gcf, BDDs can never gain in size using branch pruning.

There appear to be two gains to using branch pruning. First, it can make SMURFs smaller (see Section 11.1 for information about SMURFs). Second, it often *appears*, by avoiding duplicated information, to make the LSGB search heuristic's evidence combination rule work better.

On the negative side, it can, in odd cases, lose local information. Although it may reveal some of the inferences that strengthening would (see below), branch pruning can still cause the number of choicepoints to increase. Both these issues are related: branch pruning can spread an inference that is evident in one BDD over multiple BDDs (see Figure 26 for an example).

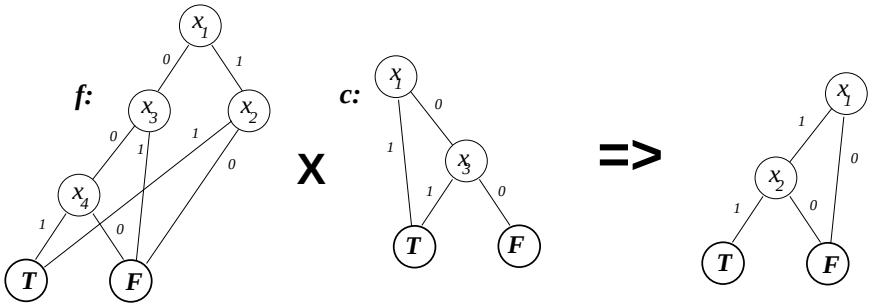


Figure 23: Example of **prune**. Procedure `Prune` is applied to the left two BDDs and returns the right BDD.

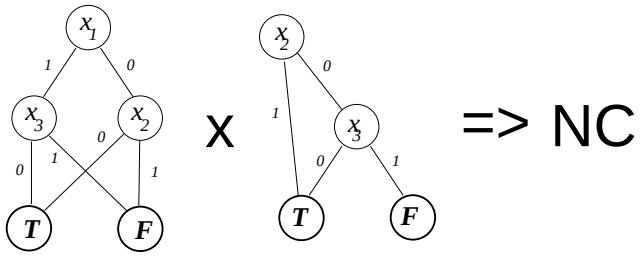


Figure 24: Prune on one variable ordering produces no result.

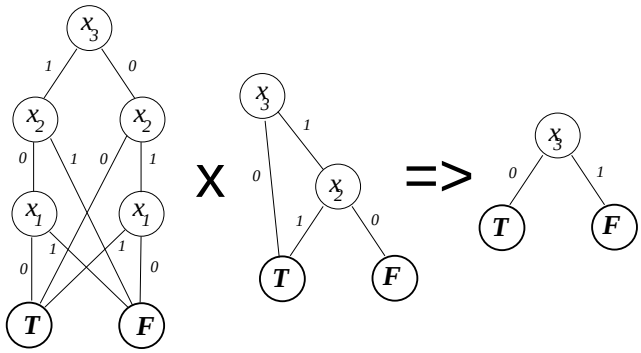


Figure 25: But changing the variable order results in an inference from pruning the two functions in Figure 24.

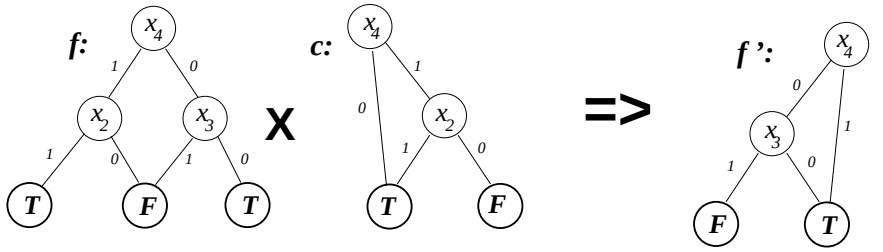


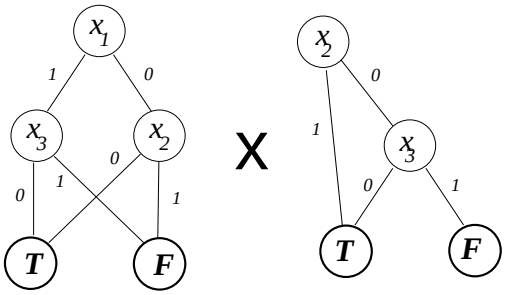
Figure 26: Example of **branch pruning** spreading an inference from one BDD to another. If x_2 is assigned 0 in f then $x_4 = 0$ and $x_3 = 0$ are inferred. After applying `Prune` to f and c and replacing f with f', to get the inference $x_3 = 0$ from the choice $x_2 = 0$ visit c to get $x_4 = 0$ and then f' to get $x_3 = 0$. Thus, branch pruning can increase work if not used properly. In this case, pruning in the reverse direction leads to a better result.

10.5 Strengthening

This binary operation on BDDs helps reveal inferences that are missed by branch pruning due to its sensitivity to variable ordering. Given two BDDs, b_1 and b_2, strengthening conjoins b_1 with the *projection* of b_2 onto the variables of b_1: that is, $b_1 \wedge \exists \vec{x} b_2$, where $\vec{x}$ is the set of variables appearing in b_2 but not in b_1. Strengthening each b_i against all other b_js sometimes reveals additional inferences or equivalences as above. Figure 27 shows an example. The following is pseudo C++ code implementing strengthening:

```
BDD Strengthen (BDD b1, BDD b2) {
    let x⃗ = {x : x ∈ b2, x ∉ b1};
    for all (x ∈ x⃗) b2 = ExQuant(b2, x);
    return b1 ∧ b2;
}
```

Strengthening is a way to pass important information from one BDD to another without causing a size explosion. No explosion can occur because before b_1 is conjoined with b_2, all variables in b_2 that don't occur in b_1 are existentially quantified away. If an inference (of the form $x = T$, $x = F$, $x = +y$, or $x = -y$) exists due to just two BDDs, then strengthening those BDDs against each other (pairwise) can "move" those inferences, even if originally spread across both BDDs, to one of the BDDs. Because strengthening shares information between BDDs it can be thought of as sharing intelligence and "strengthening"



Strengthening example: Existentially quantify away x_1...

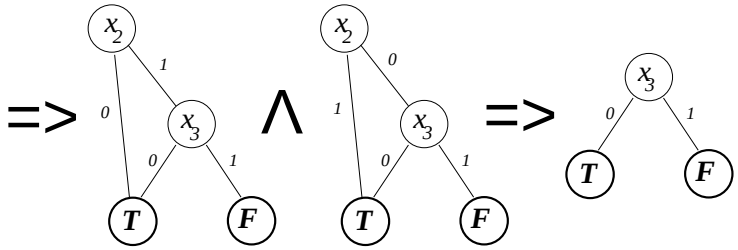


Figure 27: ...then conjoin the two BDDs. Inference revealed is $x_3 = F$.

the relationships between functions; the added intelligence in these strengthened functions can be exploited by a smart search heuristic. We have found that, in general, strengthening decreases the number of choicepoints when used in conjunction with the LSGB heuristic, though in strange cases it can also increase the number of choicepoints. We believe this is due to the delicate nature of some problems where duplicating information in the BDDs leads the heuristic astray. Strengthening may be applied to CNF formulas and in this case it is the same as applying Davis-Putnam resolution selectively on some of the clauses. When used on more complex functions it is clearer how to use it effectively as the clauses being resolved are grouped with some meaning. Evidence for this comes from from examples from Bounded Model Checking (see Section 16).

10.6 Inferences

At BDD build time, meaning every time a BDD is modified by any preprocessing function, inferences are collected and attached to every pertinent node of the BDDs. This makes it a very simple process to check a BDD for inferences:

just look at the list attached to the top node of any BDD and you'll see it's inferences. There are 4 types of inferences in the form of $x = T$, $x = F$, $x = y$, $x = -y$. As soon as any BDD has inferences to give, it's inferences are applied to every applicable BDD.

10.7 Existential quantification

A Boolean function which can be written

$$f(x, \vec{g}) = (x \wedge h_1(\vec{g})) \vee (\neg x \wedge h_2(\vec{g}))$$

can be replaced by

$$f(\vec{g}) = h_1(\vec{g}) \vee h_2(\vec{g})$$

where $\vec{g}$ is a list of one or more variables. There is a solution to $f(\vec{g})$ if and only if there is a solution to $f(x, \vec{g})$ so it is sufficient to solve $f(\vec{g})$ to get a solution to $f(x, \vec{g})$. Obtaining $f(\vec{g})$ from $f(x, \vec{g})$ is known as *existentially quantifying* x *away from* $f(x, \vec{g})$. This operation is efficiently handled if $f(x, \vec{g})$ is represented by a BDD. However, since problems handled by SBSAT include conjunctions of functions and therefore "conjunctions" of BDDs, existentially quantifying away a variable x succeeds easily only when only one of the "conjoined" BDDs contains x. Thus, this operation is typically used in conjunction with other preprocessing options for maximum effectiveness.

The following is a pseudo C++ implementation ($\vee$ denotes the "or" of two BDDs):

```
BDD ExQuant (BDD f, variable x) {
    if (root(f) == x) return true(root(f))∨false(root(f));
    if (index(x) > index(root(f))) return f; // If x is not in f do nothing
    let h_fT = ExQuant(true(root(f)), x);
    let h_fF = ExQuant(false(root(f)), x);
    if (h_fF == h_fT) return h_fT;
    return find_or_add_node (root(f), h_fT, h_fF);
}
```

Although this operation itself can uncover inferences (see, for example, Figure 28), those same inferences are revealed during BDD construction due to the particular way we build BDDs which includes developing inference lists for each

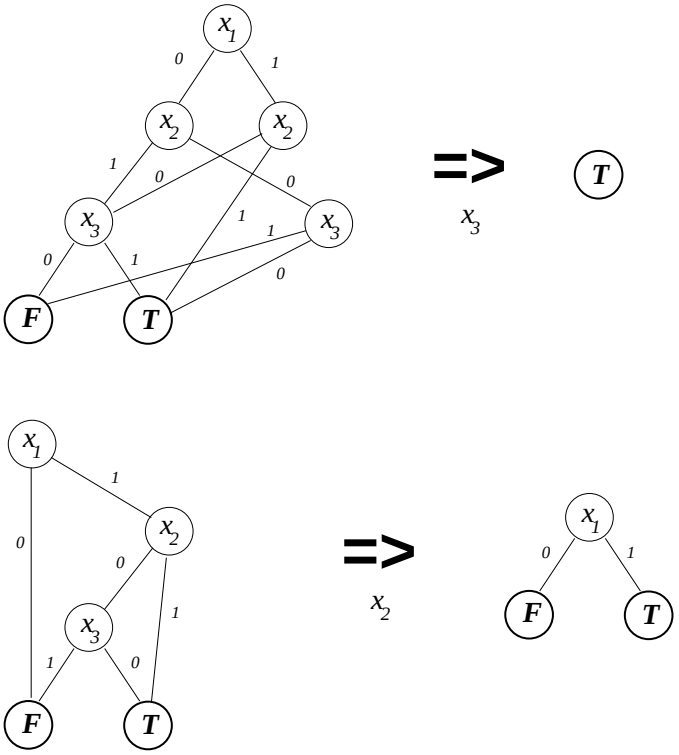


Figure 28: Two examples of **existentially quantifying away a variable from a function**. Functions are represented as BDDs on the left. Variable x_3 is existentially quantified away from the top BDD leaving T, meaning that regardless of assignments given to variables x_1 and x_2 there is always an assignment to x_3 which satisfies the function. Variable x_2 is existentially quantified away from the bottom BDD leaving the inference $x_1 = T$.

node (see Section 14.1). Inferences that would be caught later by existential quantification exist in the BDD root node inference lists and may be applied early. Therefore, existential quantification is used by `sbsat` primarily to assist other operations, such as strengthening (see Section 10.5), to uncover those inferences that cannot be found during BDD construction. Examples of the use of this operation are shown in Figure 28.

Existential quantification tends to speed up searching (that is, it results in more choicepoints per second) but tends to increase the number of choicepoints. The reason for the latter is that the elimination of a variable may cause subfunctions that had been linked only by that variable to become merged with the result that the distinction between the subfunctions becomes blurred. This is illustrated in Figure 29.

On the other hand there are fewer inferences to be made during search ("or"ing two functions removes all the F terminals that can be) so the time per choicepoint decreases. The speedup can overcome the lost intelligence but it is sometimes better to turn it off. The major benefit of existential quantification is the smaller search space.

If existential quantification is selected to occur during preprocessing (named in the command-line preprocessing sequence), when invoked, for every variable, the number of BDDs in which that variable is included is determined. If the number is one, existential quantification is applied to that variable in that BDD and the process continues until no variables are included in a single BDD. Thus, after existential quantification, all variables are in at least two BDDs.

10.8 Clustering + Existential Quantification

This preprocessing function conjoins BDDs so that at least one variable can be existentially quantified away from the entire collection. This operation repeatedly finds the variable that occurs in the least number of BDDs, conjoins those BDDs, and, if the number of variables in the result is less than a hard-wired constant (called MAX_EXQUANTIFY_VARLENGTH in the code), existentially quantifies that variable away from the resulting BDD. The operation ends when the lowest number of BDDs a variable occurs in is greater than some hard-wired constant (right now called MAX_EXQUANTIFY_CLAUSES in the source code).

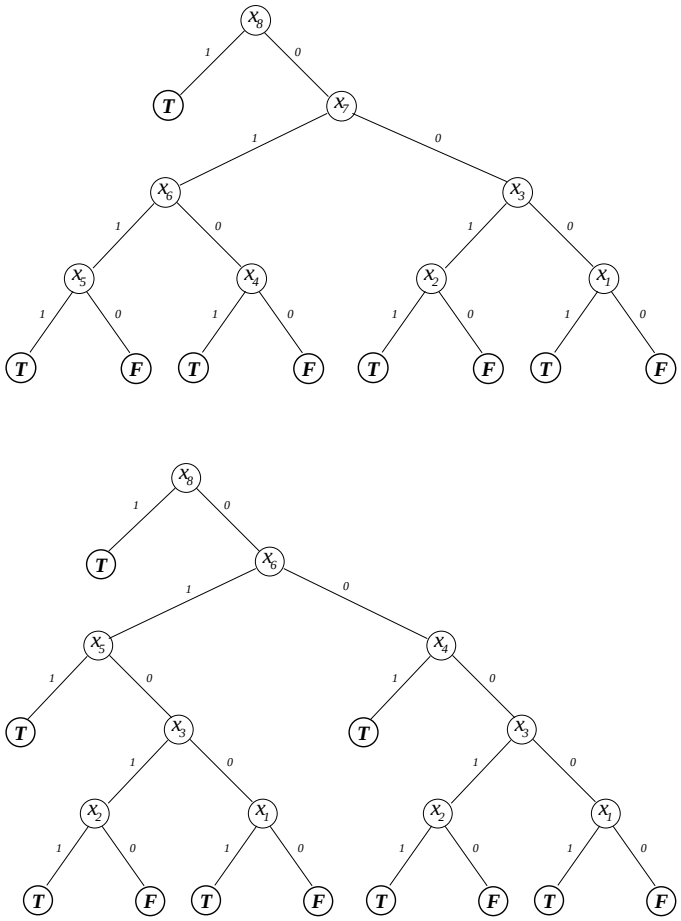


Figure 29: Existential quantification can cause blurring of functional relationships. The top function is seen to separate variables x_1, x_2, and x_3 from x_4, x_5, and x_6 if x_7 is chosen during search first. Existentially quantifying x_7 away from the top function before search results in the bottom function in which no such separation is immediately evident. Without existential quantification the assignment $x_8 = F$, $x_7 = T$, $x_6 = T$ reveals the inference $x_5 = T$. With existential quantification the assignment must be augmented with $x_2 = F$ and $x_1 = F$ (but x_7 is no longer necessary) to get the same inference.

10.9 Clustering + Existential Quantification + Safe

This process functions like 'Clustering + Existential Quantification' but before a variable is quantified it is checked for a safe assignment. If a safe assignment exists, it is recorded and applied.

10.10 Dependent variable clustering

Equations such as the following:

```
x = and(a, b, c, d, ...)
x = or(a, b, c, d, ...)
```

appear often in many applications, particularly those related to circuits. In such a case, the BDD corresponding to the equation is conjoined with all other BDDs containing variable x, the variable appearing on the left side of equals, and then x is existentially quantified out of the expression entirely. For example, the equations

```
x = and(a, b, c, d)
y = and(x, e, f, g)
```

are replaced by

```
y = and(a, b, c, d, e, f, g)
```

and x is eliminated.

As it currently exists in the tool, this operation is limited to cases where the BDDs involved all depend on 8 or fewer variables unless the BDDs are special functions and can be conjoined to a SMURF which is reasonably small in size.

10.11 Rewind

This function causes the current set of BDDs to be replaced with the initial set of BDDs. SBSAT currently saves all of the inferences detected over the course of preprocessing. After a rewind, those saved inferences are immediately applied to the rewound set of BDDs, reducing them. This technique has been found to be useful when working with preprocessing techniques which cause BDDs to be built that are too big to be built into SMURFs. We first apply

those techniques, then rewind, and then solve. This allows inferences found by the preprocessing techniques (such as clustering) to be saved and applied to the original set of BDDs, thus allowing the solver to operate on the original (manageable) functions with some reductions applied, instead of on a set of unmanageable clustered BDDs with those same reductions.

10.12 Splitter

The splitter is intended to replace large BDDs with sets of small BDDs. This is required in two circumstances. First, one of the objectives of preprocessing is to reveal inferences that may be used to reduce the size of the input before search is applied. This is done by applying various BDD operations which may result in some BDDs being fairly large. But SMURFs must be created from small BDDs. So the large BDDs must be split into small ones which are turned into SMURFs. This is accomplished by the splitter. Second, when using the three-address code available from the output of the `bmc` tool (see Section 15), large BDDs result and the splitter is used to create smaller ones from the larger ones so reasonable sized SMURFs can be created from them.

The splitter can be turned on by the user with the `-Sp 1` command line option. The maximum number of variables to split on is controlled from the command line using the `--do-split-max-vars <number>` switch (Page 43). The number of variables to split on is 10 by default.

The splitter will first try to break up all big BDDs by selecting a big BDD f and projecting f onto all 10-variable subsets of it's variable set. We could think of each projection f_i as a weak approximation to f. We collect these projections together and use branch pruning to simplify the collection. To "project" an f onto a set of variables means to quantify out all variables *not* in that set (see Section 10.5 for details).

Finally, see how close we've come to f: conjoin all these approximations f_i together, yielding f', and replace f with $and(f, not(f'))$. If some BDDs still exist with more than 10 variables then the splitter will break all remaining big BDDs into clauses.

10.13 Universe

Universally quantify a variable away from a BDD. This operation exists only in canonical form. The pseudo C++ description of this operation is as follows:

```
BDD Universe (BDD f, variable x) {
    if (root(f) == x) return true(root(f))∧false(root(f));
    if (index(x) > index(root(f))) return f; // If x is not in f do nothing
    h_{f_T} = Universe (true(root(f)), x);
    h_{f_F} = Universe (false(root(f)), x);
    if (h_{f_T} == h_{f_F}) return h_{f_T};
    return find_or_add_node (root(f), h_{f_T}, h_{f_F});
}
```

11 Reference - Search heuristics

11.1 State Machines Used to Represent Functions (SMURFs)

Prior to searching, Boolean functions become implemented as acyclic Mealy machines, called SMURFs (for "State Machine Used to Represent Functions"). SMURFs help lower the overhead of searching and make complex search heuristics feasible: all important data needed for the search process and able to be computed prior to search is memoized in SMURF states and transitions for immediate access during search. The inputs to a SMURF are literals that are assumed or inferred, during search, to be true; the outputs are sets of literals that are forced to be true (analogous to unit resolution in CNF) by the newly assigned inputs; and the states correspond to what "portion", or *residual*, of the constraint remains to be satisfied[11]. SMURFs are described in Figure 30. For a set of constraint BDDs, we compute the SMURFs for each of the separate BDDs and merge states with equal residual functions, maintaining one pointer into the resultant automaton for the current state of each constraint.

The SMURF structure described in the figure, for a Boolean function with n variables, can have, in the worst case, close to 3^n states. Thus, an Achilles' heel of SBSAT can be handling long input functions. In most benchmarks, that has not been a serious practical problem because all individual constraint are reasonably short *except*[12] for a small special group of functions: long clauses, long exclusive disjunctions, and "assignments" $\lambda_0 = \lambda_1 \wedge \cdots \wedge \lambda_k$ and $\lambda_0 = \lambda_1 \vee \cdots \vee \lambda_k$ (where the λ_i's are literals). To solve the space problem for these special functions, we create special data structures; these take little space and can simulate the SMURFs for the functions exactly with little time loss. For a long clause we store only (i) whether the clause is already satisfied, and (ii) how many literals are currently not assigned truth values. Storing exclusive disjuncts is similar. For the assignments, we store both the value (*0*,*1*, or unassigned) of the left-hand-side literal and the number of right-hand-side literals with undefined truth values.

[11] In SMURFs, each constraint implies no literals, since those would have been trapped during preprocessing.

[12] as In, for example, dlx benchmark suite made available by Miroslav Velev.

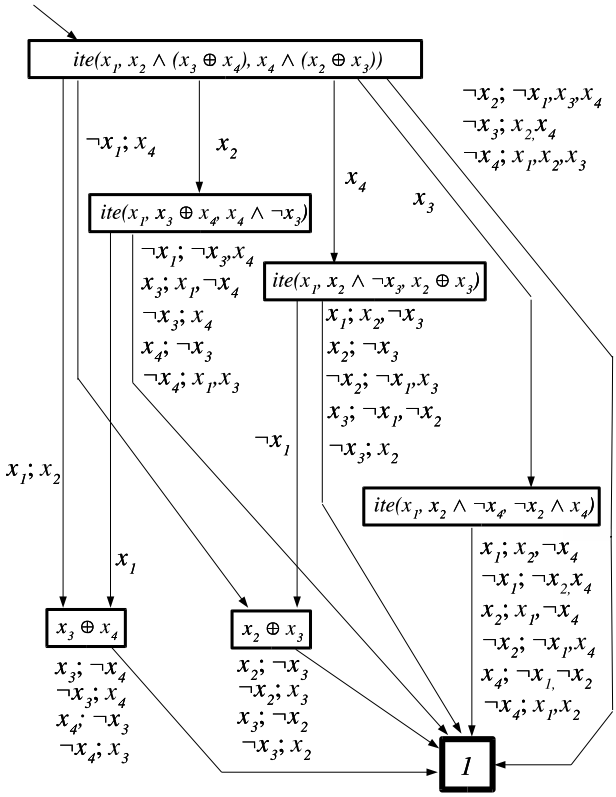


Figure 30: BDDs are preprocessed into deterministic Mealy machines called "SMURFs." This example explains construction. *ite* denotes if-then-else and $\oplus$ denotes exclusive or.

The SMURF above represents $ite(x_1, x_2 \wedge (x_3 \oplus x_4), x_4 \wedge (x_2 \oplus x_3))$. It represents, in part, BDDs for the function under all possible variable orderings — since we cannot know in what order the brancher considers the variables.
The start state (upper left) represents the original function. On the left is a transition from the start state labeled "$x_1; x_2$"; this means that, from that state, on input x_1, the automaton makes a transition and outputs $\{x_2\}$. If the brancher guesses, or infers, that x_1 is true, it will "tell" the automaton to branch on x_1. The output of $\{x_2\}$ tells the brancher that x_2 must also be true — the analogue of unit inference in CNF. This transition goes to a state labeled $x_3 \oplus x_4$, meaning that, after x_1, x_2 are set to 1, what remains to be satisfied — the *residual function* — is $x_3 \oplus x_4$. On the upper right are three transitions shown with one arrow. The first is from the start state on input $\neg x_2$; it outputs $\{\neg x_1, x_3, x_4\}$ and goes to state 1 — meaning the original BDD is now satisfied, *i.e.*, that there is no residual constraint to satisfy.

11.2 Locally Skewed, Globally Balanced

Memoized information is currently tailored for the primary search heuristic called *Locally Skewed, Globally Balanced* or LSGB. The *weight* of a SMURF transition counts the number of literals forced on the transition, plus the expected number of literals forced below that state, where a forced literal after m additional choices is weighted $1/K^m$. (K, set experimentally, is currently 3 by default.) In Figure 30, the transition out of the start state on $\neg x_1$ has weight

$$1 + (\frac{1}{K} + \frac{1}{K} + \frac{1}{K} + \frac{1}{K})/4;$$

the transition out on x_4,

$$0 + (\frac{1}{K^2} + \frac{2}{K} + \frac{1}{K} + \frac{2}{K} + \frac{2}{K} + \frac{1}{K})/6.$$

Computing these weights is expensive but they are memoized in SMURFs during preprocessing and, during search, they are looked up in a table instead of being recomputed each time they are needed.

For the special data structures defined above, the calculation above is simulated. If a disjunction $\lambda_1 \vee \cdots \vee \lambda_m$ with k still unassigned variables were represented as a SMURF, the weight of λ_i is 0 (since the clause immediately becomes satisfied, nothing more can be forced), and the weight of $\neg\lambda_i$ is $1/(2K)^{k-1}$. This is directly coded in the simulated SMURF. Exclusive disjunctions are similar. Assignments are similar but break into cases; one recurrence relation is hard to solve, so weights are precomputed as a function of the number of unassigned λ_i's and are looked up during search.

The LSGB search heuristic is similar to the "Johnson heuristic" on CNF formulas where $K = 2$. The intuition is to branch toward forced inferences as quickly as possible to narrow the search space (or get lemmas fast). To pick the next variable to branch on: For each variable x_i, compute (i) the sum S_i^+ of the *weight*s of transitions on x_i out of all current SMURF states and (ii) the sum S_i^- of the *weight*s of transitions on $\neg x_i$. A high sum represents a high "payoff." For an ideal branching variable x_i, both x_i and $\neg x_i$ force many literals, so we branch on the variable x_i where $S_i^+ \cdot S_i^-$ is maximum. For that variable, branch first toward the larger of S_i^+, S_i^-.[13]

[13]The idea of taking the product is due to Freeman.

There are circumstances where other search heuristics are known to work well. LSGB was intended for applications where not much is known about, or easily determined about, the given problem. If the problem is known to have a lot of exploitable structure, it may be better to specify a different heuristic. We allow the experienced user some choice (see Sections 11.3 and 11.4 below for more information). The SMURF structure admits such heuristics as well; on a simple heuristic, it may not be needed, but (except for preprocessing time) it does not hinder either.

In Section 16, we present benchmark problems comparing SBSAT with LSGB to other solvers such as zChaff.

11.3 Chaff-like

11.4 User defined search heuristic

12 Reference - Search methods

12.1 Backtracking and Lemmas

12.1.1 Lemma cache

12.1.2 Lemma effectiveness

12.2 BDD WalkSAT

12.3 WVF

13 Reference - Output, results

13.1 Raw

If you use raw format, `-R r` the output looks as follows:

```
// Solution #1
-arg1 -arg2 -arg3 -arg4 x3 x2 x1 1 2 3 -5 -bob 4 -1000 22 300 -40 -400 -50
var1 var2 var3 var4 -var5 -var6
```

13.2 Fancy

If you use fancy format, `-R f` the output looks as follows:

```
// Solution #1
arg1     (1)    val:F
arg2     (2)    val:F
arg3     (3)    val:F
arg4     (4)    val:F
x3       (5)    val:T
x2       (6)    val:T
x1       (7)    val:T
1        (8)    val:T
2        (9)    val:T
3        (10)   val:T
5        (11)   val:F
bob      (12)   val:F
4        (13)   val:T
1000     (14)   val:F
22       (15)   val:T
300      (16)   val:T
40       (17)   val:F
400      (18)   val:F
50       (19)   val:F
var1     (20)   val:T
var2     (21)   val:T
var3     (22)   val:T
```

```
var4     (23)   val:T
var5     (24)   val:F
var6     (25)   val:F
```

14 Reference - Data structures

14.1 BDD database

14.2 Smurf

14.3 Lemma database

15 Reference - Results: making BDDs from `bmc`

Among the experiments we have run, those inputs relating specifically to bounded model checking benchmarks have been obtained from the output of the `bmc` program obtainable from Carnegie Mellon University. That program inputs a model checking problem and a number of time steps and outputs a propositional logic formula representing the BMC problem in three formats: a large propositional logic formula, three-address code representing the parse tree for that formula, and a CNF translation of the formula. Program `bmc` internally represents all formulas recursively as

$$\texttt{<function>} = \texttt{<variable>};$$
$$\texttt{<function>} = \neg\texttt{<variable>};$$
$$\texttt{<function>} = \texttt{<function>}\ \mathbf{op}\ \texttt{<function>};$$

where **op** is one of $\vee$, $\wedge$, $\rightarrow$, $\equiv$. The binary tree associated with such a recursion is stored as a tree of pointers. Each node of the tree is represented as a triple of pointers: to the left descendent, the right descendent, and the parent. A pointer to the root of such a tree represents the output formula in three-address code. Further processing inside `bmc` converts this to a CNF expression which is also available as output. As an example, we use `bmc` to generate the three-address code problems for queue benchmarks (see next section) as follows:

```
genqueue # > queue#
bmc -k # queue# -prove
```

where `genqueue` is part of the `bmc` suite and `#` is replaced by a number representing problem complexity. The CNF versions are created by replacing the last line above with this:

```
bmc -k # queue# -dimacs
```

We use `bmc` to generate three-address and CNF inputs directly, instead of taking already generated CNF formulas "off the shelf" so we have equivalent three-address and CNF data. Thus, times we report for zChaff, Berkmin, and Siege may differ from published times.

The largest propositional logic formula output by `bmc` is a conjunction of smaller formulas, so the obvious course for SBSAT is to read in each of those smaller formulas as a BDD. Nevertheless, for some of the `bmc` outputs, those propositional logic formulas were much too large even to store as BDDs. Of

course, we also did not want to use the three-address code or the CNF representation directly, since that would negate the benefits of SMURFs which are to retain potentially exploitable domain-specific relationships. Our current approach is successful in spite of being amazingly simplistic.

1. We read in the three-address code and recreate the large propositional formula so as not to lose domain-specific information. Starting at the bottom of this formula we start building a BDD. We use a greedy algorithm: when the BDD gets too large (10-18 variables) we insert a new variable to represent the BDD so far, include a BDD asserting that is what the new variable represents, replace the part we have translated with the new variable, and continue the process. This particular translation goes against our intention of staying in the original domain, however, this simple process still proves useful. In future research we hope to find a better algorithm.

2. To break each resultant BDD f down to a 10-variable maximum (so that the SMURFs remain suitably small), we do the following (see also Section 10.12):

   (a) Compute all projections f_i of the BDD onto 10-variable subsets of its variable set (see Section 10.5 for the meaning of projection).

   (b) Simplify the f_i's against each other and delete resultant f_i's which become `True`. Below we call the final simplified f_i's $f_1, \ldots, f_k$.

   Note that f logically implies each f_i; we can think of them as "approximations" to f, in the sense that each is false on some, but probably not all, of the truth assignments on which f is false.

   (c) Recall that the goal is to replace f with a set of smaller BDD's. Now f is logically equivalent to the conjunction of the set $\{f_1, f_2, \ldots, f_k, f^\star\}$ where

   $$f^\star = (f_1 \wedge f_2 \wedge \cdots \wedge f_k) \rightarrow f$$

   ($f^\star$ just excludes the truth assignments where all the f_i's are true but f is is false).

   If $f^\star$ has ≤ 10 variables, we replace f with $\{f_1, f_2, \ldots, f_k, f^\star\}$. If $f^\star$ has > 10 variables, we replace f with $\{f_1, f_2, \ldots, f_k\}$ plus the translation of $f^\star$ to CNF. (Typically, $f^\star$ is satisfied in most truth assignments, so its CNF translation should be fairly short.)

Again, this procedure is simplistic. We hope in the future to find a better algorithm.

16 Reference - Results: Experiments

SBSAT was tested on several popular benchmark suites. We also ran current versions of Berkmin (v. 561), zChaff (v. 2003.10.9), and Siege (v. 4) on these benchmarks for comparison. In addition, we concocted a class of random problems, called *sliders*, which resemble BMC problems in that copies of the same function, each differing only in the input variables it depends on, are conjoined. Making those functions random, in some sense, makes sliders hard. Specifically, sliders are defined as follows:

> Choose m, even, the number of constraints and the number of variables. Choose k, and l, the number of variables input to constraint functions. Choose constraint functions $f(x_1, x_{i_1}, ..., x_{i_{k-2}}, x_{m/2})$ and $g(x_1, x_{j_1}, ..., x_{j_{l-2}}, x_{m/2})$, with variables explicitly listed, in increasing order of subscript, and k and l are small compared to m. Form the constraint set
>
> $$\{f(x_{1+h}, x_{i_1+h}, ..., x_{i_{k-2}+h}, x_{(m/2)+h}) : 0 \leq h \leq m/2\} \cup$$
> $$\{g(x_{1+h}, x_{j_1+h}, ..., x_{j_{l-2}+h}, x_{(m/2)+h}) = o_h : 0 \leq h \leq m/2\}$$
>
> where each o_h is independently and uniformly chosen from $\{0, 1\}$.

We find sliders appealing because they resemble some real-world problem domains and because f and g can be designed to force inferences to occur only when nearly all inputs of f and g are assigned values. This fact makes conflict analysis useless, and is challenging to a search heuristic which is looking for information contained in groups of variables.

At this stage of our SBSAT implementation, lemmas are handled in a rather primitive manner so we observe an unusually low number of backtracks per second. All experiments were run on a single processor Pentium 4, 2 GHz, with 2 GB RAM.

Our first set of results, shown in Table 1, is for the problem of verifying a long multiplier. The circuit definition is available from Carnegie Mellon University. All inputs are unsatisfiable. The left column of the table shows the number of time steps involved in the verification of each benchmark (see Section 15). Experiments were run from 4 time steps to 70 time steps. The next three columns present the observed performance of SBSAT on three-address inputs

	SBSAT on Three-Address			SBSAT on CNF			zChaff on CNF		Siege	Berkmin
#time steps	number choices	total (sec)	branch (sec)	number choices	total (sec)	branch (sec)	number choices	total (sec)	total (sec)	total (sec)
4	720	2.3	0.16	687	1.47	0.86	1041	0.45	0.2	0.27
8	10000	14.78	7.12	13110	41.02	39.28	33272	50.37	12.73	18.9
12	19398	42.14	27.31	31963	167.8	163.8	122522	357.1	71.61	96.9
16	17508	61.05	38.89	32969	247.3	240.4	125026	366.7	177.4	200.6
20	14077	72.63	41.65	34426	347.0	335.1	164373	585.9	165.2	178.8
24	17775	118.5	77.03	23854	270.0	252.0	214263	790.3	542.8	312.2
28	18872	134.1	81.71	23847	319.0	293.8	220045	888.2	805.4	255.0
32	18538	155.6	90.5	16718	262.9	228.3	216916	882.8	1035	334.6
36	20356	186.8	109.9	14750	278.0	233.5	269856	1055	576.8	420.4
40	19141	203.3	113.5	11703	281.1	225.0	289687	1103	845.3	442.6
50	21867	263.0	134.4	11306	378.6	286.9	472053	2032	1552	466.9
60	24985	434.1	239.4	10844	450.2	313.6	461867	2183	3340	709.2
70	26907	618.4	335.9	11270	632.8	164.3	850942	5875	2860	844.7

Table 1: SBSAT, zChaff, Siege, Berkmin times on the Long Multiplier benchmarks

in total number of choice points, total time, and search time. The next three columns present the same information except when translated CNF formulas are input (see Section 15). The next two columns present the performance of zChaff in choice points and total time and the last two columns present the results of Siege and Berkmin on the CNF versions we generated.

Observe that SBSAT working in the user domain on three-address code shows a slight advantage to working with the CNF translation. It is interesting that in the case of CNF inputs, more preprocessing seems to result in less searching. The fact that preprocessing varies so much from benchmark to benchmark on CNF inputs may reflect the imprecision of guesses made when trying to recreate domain-specific information from given CNF formulas. Such preprocessing fluctuations are not as pronounced when three-address codes are input to SBSAT.

Observe that zChaff and Siege cannot compete with SBSAT on long multiplier benchmarks. The problem seems to be due to encountering many more choicepoints during search. Berkmin visits only about an order of magnitude more choicepoints than SBSAT on CNF inputs but the slower implementation of lemmas in SBSAT enables Berkmin to be only a fraction slower than SBSAT, in general. The difference in choicepoints suggests the success in this case is due to the complex search heuristic used natively in SBSAT.

Table 2 shows timings for the set of barrel benchmarks. The three-address code equivalents were generated by applying the `bmc` tool to the output of the `genbarrel` utility in the `bmc` suite. All inputs are unsatisfiable. Runs were cut off prematurely if not completed before 3600 seconds. This is reflected as a line (—) through a table entry.

	SBSAT on Three-Address			SBSAT on CNF			zChaff on CNF		Siege	Berkmin
Name	number choices	total (sec)	branch (sec)	number choices	total (sec)	branch (sec)	number choices	total (sec)	total (sec)	total (sec)
barrel2	0	0.00	0.00	3	0.05	0.00	3	0.00	0.01	0.0
barrel3	0	0.11	0.00	13	0.08	0.00	48	0.00	0.01	0.0
barrel4	0	0.12	0.00	33	0.15	0.01	201	0.02	0.01	0.01
barrel5	0	0.72	0.00	354	0.66	0.21	8856	0.58	0.67	0.65
barrel6	0	1.48	0.00	1205	2.89	1.96	28110	2.81	5.97	5.56
barrel7	0	2.84	0.00	2848	11.10	8.51	66959	11.37	21.19	29.96
barrel8	0	5.05	0.00	4304	25.15	18.71	116858	31.98	136.7	298.3
barrel9	0	67.87	0.00	—	—	—	649532	254.6	41.24	89.27
barrel10	0	108.9	0.00	—	—	—	1801476	1191	86.34	184.0
barrel11	0	166.2	0.00	—	—	—	—	—	134.7	238.3
barrel12	0	243.8	0.00	—	—	—	—	—	927.1	999.3
barrel13	0	348.4	0.00	—	—	—	—	—	629.9	1049
barrel14	0	481.9	0.00	—	—	—	—	—	2122	3389
barrel15	0	655.9	0.00	—	—	—	—	—	—	—
barrel16	0	859.7	0.00	—	—	—	—	—	—	—

Table 2: SBSAT, zChaff, Siege, Berkmin times on the Barrel benchmarks

Observe that in all cases, SBSAT solved the problems constructed from the three-address code without any search. This raises the question of whether a BDD tool might also do as well. This appears not to be the case, since we build a collection of BDDs of about 10 variables each and then strengthen them against each other. The inferences resulting from this process are enough to generate a contradiction before search is applied. We suppose a BDD tool would either have attempted to build a single BDD from the three-address code, in which case it would have been forced to give up due to unmanageable sizes, or it would have used the conjoin operation instead of the strengthening operation to combine the BDDs, probably again taking too much space. Although the time taken by SBSAT in preprocessing is considerable, it is shown to be well-spent as SBSAT, zChaff, Siege, and Berkmin all have difficulty with the larger CNF versions of the barrel benchmarks. Thus, it appears staying closer to the user-domain and preprocessing to reveal inferences early has paid off on these benchmarks.

Tables 3 and 4 show timings for a set of queue benchmarks and permute benchmarks generated by `genqueue` and `genpermute`, respectively, from the `bmc` suite. Cutoff of runs was set at 3600 seconds for the queue benchmarks and 60000 seconds for the permute benchmarks. All inputs are unsatisfiable. The pattern observed is similar to the previous sets of runs. When SBSAT works with three-address code timings are much better than when equivalent CNF inputs are used. Working in three-address code gets results faster than other solvers on equivalent CNF inputs.

The story changes on the queue invariant benchmarks of Table 5. In this case, SBSAT experienced memory problems. In order to fit the resulting SMURFs into memory, the BDDs upon which they were based were required to be so small we had to change their maximum size manually, that is, after preprocessing.

	SBSAT on Three-Address			SBSAT on CNF			zChaff on CNF		Siege	Berkmin
Name	number choices	total (sec)	branch (sec)	number choices	total (sec)	branch (sec)	number choices	total (sec)	total (sec)	total (sec)
queue4	41	0.1	0.0	19	0.11	0.00	32	0.00	0.01	0.0
queue8	651	3.04	0.07	291	0.49	0.10	561	0.05	0.04	0.05
queue12	4351	5.53	1.02	3875	5.52	4.38	11752	3.09	1.04	0.96
queue16	30835	22.3	14.7	41029	107	104	73407	62.22	30.27	32.38
queue20	311127	265	227	565559	2420	2412	698914	1874	400.4	401.0
queue22	1052750	843	798	2016859	9367	9356	—	—	1886	1050
queue24	3262464	2666	2613	—	—	—	—	—	—	2724

Table 3: SBSAT, zChaff, Siege, Berkmin times on the Queue benchmarks

	SBSAT on Three-Address			SBSAT on CNF			zChaff on CNF		Siege	Berkmin
Name	number choices	total (sec)	branch (sec)	number choices	total (sec)	branch (sec)	number choices	total (sec)	total (sec)	total (sec)
permute2	0	0.01	0.00	1	0.05	0.00	1	0.00	0.01	0.00
permute3	5	0.04	0.00	14	0.07	0.00	11	0.00	0.01	0.00
permute4	68	0.65	0.00	47	0.11	0.00	52	0.00	0.01	0.01
permute5	174	10.1	0.01	304	0.27	0.10	199	0.02	0.02	0.03
permute6	893	11.46	0.09	1655	1.44	1.15	2021	0.28	0.17	0.16
permute7	5537	23.24	0.81	8551	9.21	8.77	16485	9.51	2.88	1.12
permute8	64607	71.16	70.21	58051	244.1	243.2	110492	172.93	21.6	15.7
permute9	454726	686.6	685.0	471422	2575	2573	361422	1018	315	228
permute10	1311291	2064	2062	—	—	—	2118409	12101	3003	3891
permute11	20462503	39260	39257	—	—	—	—	—	—	—

Table 4: SBSAT, zChaff, Siege, Berkmin times on the Permute benchmarks

The result was an unexpectedly large amount of garbling of domain-specific information and dismal results. We did not feel it was worthwhile reporting them. Although SBSAT did solve the CNF versions of these problems, the other solvers performed better as in previous benchmark sets.

For completeness, we include results on the dlx suite available from Carnegie Mellon University in Table 6. Some inputs are satisfiable and some are unsatisfiable. We applied SBSAT to two variations: namely Trace and CNF formats (both available). All problems in this suite are easy for all the solvers and that is about all that can be said about them. We did not include results of dlx9 benchmarks because SBSAT had some memory problems.

Finally, Table 7 shows the result of applying all the solvers to a family of

	SBSAT on CNF			zChaff on CNF		Siege	Berkmin
Name	number choices	total (sec)	branch (sec)	number choices	total (sec)	total (sec)	total (sec)
queueinv4	83	0.08	0.01	136	0.00	0.01	0.01
queueinv8	438	0.17	0.07	1122	0.04	0.06	0.06
queueinv12	1429	0.98	0.42	4368	0.22	0.31	0.12
queueinv16	2411	1.04	0.75	7721	0.27	0.53	0.24
queueinv20	4787	6.81	2.91	16258	1.63	0.73	0.81
queueinv24	7379	13.62	6.00	26995	2.96	1.89	1.90
queueinv28	10914	25.61	11.23	38145	5.69	3.88	3.40
queueinv32	15403	16.73	14.56	68641	3.20	3.74	4.20
queueinv36	21324	116.6	35.01	103281	23.58	9.59	10.33
queueinv40	27404	189.2	52.54	145691	38.08	17.62	16.46
queueinv44	35820	309.2	88.65	166634	46.42	57.38	25.16
queueinv48	44719	476.2	135.8	217615	79.95	62.0	43.61
queueinv52	52320	683.8	189.9	297830	179.2	155.5	55.93
queueinv56	51768	928.0	238.9	397142	239.1	514.9	82.13

Table 5: SBSAT, zChaff, Siege, Berkmin times on the Queue Invariant benchmarks

	SBSAT on Trace			SBSAT on CNF			zChaff on CNF		Siege	Berkmin
Name	number choices	total (sec)	branch (sec)	number choices	total (sec)	branch (sec)	number choices	total (sec)	total (sec)	total (sec)
dlx1_c	525	0.12	0.02	592	0.12	0.03	1082	0.02	0.01	0.01
dlx2_aa	1755	0.22	0.06	2062	0.26	0.08	5224	0.10	0.06	0.02
dlx2_ca	7247	1.49	1.00	6861	1.60	0.91	9800	0.30	0.17	0.12
dlx2_cc	9655	2.60	2.03	9631	2.83	1.97	17825	0.95	0.36	0.26
dlx2_cl	9375	2.14	1.56	8872	2.33	0.57	25390	1.50	0.71	0.29
dlx2_cs	8489	1.84	1.31	7916	2.15	1.37	16310	0.77	0.20	0.23
dlx2_la	6233	1.06	0.64	6814	1.41	0.84	9246	0.26	0.11	0.10
dlx2_sa	2938	0.35	0.16	2168	0.38	0.15	5563	0.14	0.08	0.03
dlx2_cc_bug01	6603	1.77	1.20	6448	2.11	1.25	14471	0.84	0.18	0.28
dlx2_cc_bug02	6584	1.80	1.22	6432	2.09	1.25	13717	0.79	0.48	0.26
dlx2_cc_bug03	6861	1.81	1.23	6628	2.09	1.23	22776	1.05	0.01	0.10
dlx2_cc_bug04	6932	1.92	1.33	6699	1.12	1.28	12860	0.52	0.08	0.08
dlx2_cc_bug05	3743	1.24	0.65	3413	1.47	0.62	376	0.01	0.22	0.13
dlx2_cc_bug06	3630	1.19	0.60	3581	1.52	0.67	374	0.01	0.01	0.10
dlx2_cc_bug07	4601	1.36	0.77	3567	1.50	0.65	316	0.01	0.03	0.05
dlx2_cc_bug08	5964	1.65	1.06	5353	1.75	0.92	747	0.02	0.01	0.04
dlx2_cc_bug09	2549	0.92	0.42	2693	1.18	0.33	321	0.01	0.01	0.02
dlx2_cc_bug10	3423	1.15	0.55	3564	1.41	0.56	259	0.00	0.02	0.02
dlx2_cc_bug11	6037	1.60	1.03	6886	2.20	1.35	10528	0.43	0.02	0.06
dlx2_cc_bug12	7099	2.00	1.43	5702	1.91	1.05	11099	0.44	0.07	0.10
dlx2_cc_bug13	5998	1.69	1.12	6133	1.91	1.08	12049	0.50	0.03	0.02
dlx2_cc_bug14	253	0.59	0.01	298	0.87	0.01	234	0.01	0.12	0.02
dlx2_cc_bug15	4405	1.93	1.27	3756	1.99	0.99	296	0.01	0.01	0.06
dlx2_cc_bug16	252	0.58	0.01	297	0.86	0.01	233	0.01	0.13	0.01
dlx2_cc_bug17	504	1.16	0.06	4453	2.97	1.01	5806	0.40	0.01	0.01
dlx2_cc_bug18	1066	1.06	0.10	3236	2.51	0.78	337	0.01	0.01	0.02
dlx2_cc_bug19	269	0.63	0.02	302	0.89	0.02	4452	0.15	0.01	0.00
dlx2_cc_bug20	703	0.60	0.03	777	0.89	0.50	521	0.01	0.01	0.02
dlx2_cc_bug21	331	0.59	0.02	360	0.85	0.02	458	0.01	0.01	0.01
dlx2_cc_bug22	744	0.62	0.40	865	0.91	0.05	4456	0.19	0.01	0.04
dlx2_cc_bug23	620	0.60	0.03	323	0.86	0.02	4726	0.14	0.01	0.10
dlx2_cc_bug24	270	0.59	0.02	313	0.86	0.02	4034	0.14	0.01	0.04
dlx2_cc_bug25	3931	1.32	0.75	3233	1.44	0.59	4406	0.14	0.01	0.02
dlx2_cc_bug26	4200	1.42	0.83	3687	1.58	0.48	543	0.02	0.02	0.02
dlx2_cc_bug27	591	0.52	0.02	2979	1.16	0.46	293	0.01	0.03	0.00
dlx2_cc_bug28	2205	0.88	0.22	5275	2.05	1.09	339	0.01	0.08	0.01
dlx2_cc_bug29	324	0.58	0.01	334	0.87	0.02	243	0.01	0.19	0.03
dlx2_cc_bug30	311	0.60	0.02	267	0.89	0.02	323	0.01	0.30	0.02
dlx2_cc_bug31	294	0.58	0.02	325	0.88	0.02	247	0.00	0.24	0.02
dlx2_cc_bug32	278	0.59	0.02	317	0.86	0.02	242	0.00	0.02	0.01
dlx2_cc_bug33	299	0.58	0.02	305	0.88	0.02	272	0.01	0.19	0.06
dlx2_cc_bug34	329	0.60	0.02	506	0.86	0.03	298	0.01	0.30	0.02
dlx2_cc_bug35	282	0.59	0.02	328	0.89	0.02	318	0.01	0.32	0.03
dlx2_cc_bug36	279	0.61	0.02	325	0.86	0.02	316	0.01	0.08	0.07
dlx2_cc_bug37	3643	1.28	0.71	3214	1.45	0.60	329	0.01	0.05	0.01
dlx2_cc_bug38	6249	1.70	0.43	5854	1.93	1.09	9500	0.36	0.44	0.07
dlx2_cc_bug39	3307	1.07	0.54	6058	1.88	1.04	12314	0.50	0.04	0.40
dlx2_cc_bug40	8046	2.21	1.64	6748	2.26	1.40	9972	0.41	0.12	0.02

Table 6: SBSAT, zChaff, Siege, Berkmin times on the DLX benchmarks. Benchmarks with bug in the name are satisfiable (verified) and the rest are unsatisfiable.

slider problems, some satisfiable and some unsatisiable, based on the following:

sliderxx_sat:

$$f = (x_1 \oplus (\neg x_{i_3} \wedge x_{i_1}) \oplus \neg(x_{m/2} \wedge x_{i_4})) \equiv ite(x_{i_2}, x_{i_1} \vee \neg x_{m/2}, \neg x_{i_1})$$
$$g = \neg x_1 \oplus (x_{j_2} \oplus (\neg x_{j_3} \wedge x_{j_4}) \oplus x_{j_3}) \oplus (x_{m/2} \equiv x_{j_1})$$

f:

m	i_1	i_2	i_3	i_4
60	13	15	17	24
70	12	15	17	24
80	15	17	33	24
90	15	17	24	33
100	15	17	24	43
110	15	17	24	43
120	15	24	43	57

g:

m	j_1	j_2	j_3	j_4
60	12	16	18	27
70	12	15	19	27
80	12	16	18	27
90	12	16	18	27
100	18	26	27	42
110	20	26	27	42
120	6	18	27	42

sliderxx_unsat:

$$f = (x_1 \oplus (\neg x_{i_3} \wedge x_{i_1}) \oplus \neg(x_{m/2} \wedge x_{i_4})) \equiv ite(x_{i_2}, x_{i_1} \vee \neg x_{m/2}, \neg x_{i_1})$$
$$g = (x_{m/2} \equiv (\neg x_1 \oplus (x_{j_2} \oplus (\neg x_{j_3} \wedge x_{j_4}) \oplus x_{j_3}) \oplus (x_{j_5} \equiv x_{j_1})))$$

f:

m	i_1	i_2	i_3	i_4
60	13	15	17	24
70	12	15	17	24
80	15	17	33	24
90	15	17	24	33
100	15	17	24	43
110	15	17	24	43
120	15	24	43	57

g:

m	j_1	j_2	j_3	j_4	j_5
60	12	16	18	19	27
70	12	16	18	19	27
80	12	16	18	27	29
90	12	16	18	27	29
100	18	19	26	27	42
110	18	29	26	27	42
120	6	18	27	29	42

If "unsat" is in the name of the benchmark, then it is unsatisfiable, otherwise it is satisfiable. The number in the name of each benchmark refers to the value of m. The value of k for all benchmarks is fixed at 6 and the value of l is 6 or 7 (see the beginning of this section for an explanation of this family of benchmarks and the meaning of m, k and l). The two functions were chosen to yield somewhat balanced BDDs, requiring nearly all inputs to have a value before an inference could be established. These are hard problems and only zChaff was able to approach the runtimes of SBSAT. Table 8 shows why these problems are hard. We turned off lemmas in SBSAT and reran all the benchmarks. The number

	SBSAT			zChaff		Siege		Berkmin	
Name	number choices	total (sec)	branch (sec)	number choices	total (sec)	number choices	total (sec)	number choices	total (sec)
slider60_sat	1051	0.25	0.10	534	0.02	2900	0.16	2114	0.09
slider70_sat	622	0.27	0.06	1511	0.07	329	0.01	425	0.01
slider80_sat	79884	39.4	39.2	149153	52.8	38044	6.20	209805	73.5
slider90_sat	2765	0.64	0.44	66152	14.3	47180	9.56	41372	8.63
slider100_sat	36761	15.4	15.9	104054	85.5	70693	35.8	120468	48.2
slider110_sat	171163	113.4	113.2	280126	173.3	576670	437.4	1909731	801.4
slider60_unsat	9227	1.49	1.27	27414	3.4	19505	2.63	25251	4.37
slider70_unsat	7957	1.46	1.29	18157	1.93	17735	2.21	17543	2.17
slider80_unsat	148242	78.8	78.6	245112	116.6	215436	104.4	—	—
slider90_unsat	429468	263.4	263.0	685026	513.4	501539	302.5	—	—
slider100_unsat	1600514	1066	1065	1495633	3094	2482913	6540	—	—

Table 7: SBSAT, zChaff, Siege, Berkmin times on the Slider benchmarks

	SBSAT with Lemmas			SBSAT without Lemmas		
Name	number choices	total (sec)	branch (sec)	number choices	total (sec)	branch (sec)
slider60_sat	1051	0.25	0.10	1152	0.14	0.04
slider70_sat	622	0.27	0.06	1265	0.22	0.05
slider80_sat	79884	39.4	39.2	111575	5.22	5.12
slider90_sat	2765	0.64	0.44	3576	0.30	0.18
slider100_sat	36761	15.4	15.9	51994	2.83	2.69
slider110_sat	171163	113.4	113.2	282213	16.0	15.8
slider120_sat	—	—	—	1539977	86.3	86.1
slider60_unsat	9227	1.49	1.27	10004	0.46	0.37
slider70_unsat	7957	1.46	1.29	9373	0.50	0.39
slider80_unsat	148242	78.8	78.6	190177	8.67	8.57
slider90_unsat	429468	263.4	263.0	626812	29.8	29.7
slider100_unsat	1600514	1066	1065	2403878	124.2	124.1
slider110_unsat	—	—	—	10256075	564.7	564.5

Table 8: SBSAT times and choice points on the Slider benchmarks, with and without Lemmas.

of choicepoints generated did not change very much. Thus, for these problems, learning from conflict analysis during search seems to help little. Notice also that SBSAT running time changes by an order of magnitude. This clearly points to adjustments that must be made to lemma handling.

17 Reference - Debugging

17.1 Converting to another format

17.2 Printing internal forms

18 Reference - Writing Exploitable Input