Ben-Jose SAT Solving Software Library
Tool Paper

Jose Luis Quiroga Beltran

Independent Researcher*
joseluisquiroga@yahoo.com
March 2016

Abstract

The software library Ben-Jose (https://github.com/joseluisquiroga/ben-jose) for solving instances (formulas) of the satisfiability problem (SAT) in CNF form (DIMACS format) is presented. Ben-Jose implements a trainable strategy that extends the traditional DPLL+BCP+CDCL resolution based approach, first introduced by Joao Marquez da Silva [9] and latter refined by others [16] [11]. Ben-Jose has an original technique (BDUST) to check during search if, for a given partial assignation of the solving formula variables, the resulting current sub-formula has been previously found unsatisfiable. It does that by finding the current sub-formula permutation to its "BDUST canonical form formula" (BCFF) and checking the BCFF existence in a database of unsatisfiable BCFFs. If the BCFF of the current sub-formula exists in the database, ben-jose entirely skips the search on the current sub-formula. The calculation of a BCFF introduces an original stabilization procedure (as in [2]) for the structure of CNF formulas. The calculation has linear complexity and is tightly coupled with the work done by BCP.

Contents

---

1   Introduction

The satisfiability problem (SAT) is the canonical decision problem by excellence [3] [14] [15]. It lays at the heart of the P vs NP question and its importance cannot be overstated [5]. People have proved both that "NP != P" and that "NP==P" [20] [17], and filed patent applications for optimal SAT solvers based on resolution [18].

The practical limitations of human verification of long and complex proofs, even under the peer-review system, highlights the importance of automated proof checking. This and the practical applications of automated theorem proving [8] highlights the importance of SAT.

Exponential lower bounds of resolution (RES) proof size for the pigeon hole principle (PHP) have been proven [12] [4], and polynomial size proofs for extended resolution (ER) have also been proven for the PHP [6] [7] [13] instances of the SAT problem. That work suggests that solvers based on RES [9] [16] [11] need "something else" in order to be "faster" [10] [1].

Based on the notion that theorems are proved with lemmas and the observation that the structure of PHP(n+1, n) can be matched with several substructures of PHP(n+2, n+1), the software library presented in this work (Ben-Jose) extends RES by learning the structure of unsatisfiable sub-formulas (proved unsatisfiable during its execution) and matching them against future structures of sub-formulas found during its execution, in order to skip the search, and directly backtrack on them, when ever a match is found.

This technique is here after called Backtrack Driven by Unsatisfiable Sub-formula Training (BDUST). BDUST introduces an original stabilization procedure (as in [2]) for the unsatisfiable sub-formulas found, uses the work done by BCP (each BCP step groups some variables). The stabilization procedure finds the current sub-formula permutation to its "BDUST canonical form formula" (BCFF). BDUST checks for the BCFF existence in a persistent database of unsatisfiable BCFFs. The stabilization procedure has linear complexity with respect to the size of the sub-formula. The BCFFs have the advantage that can be used also with different instances than the one they were found on. That is why BDUST is "training" and not "learning". It also means that it is possible that a run on a permutation of an already ran (trained) unsatisfiable instance may be done in a single backtrack (see Section 6).

This form of extending RES is not ER, as the system presented by Tseitin [19], because the subsumed isomorphism detection technique is not RES based, and the power and complexity of the resulting proof system has not been formally studied. The empirical results on the classical theoretical problem of PHP are presented here.

2   Objectives

Ben-jose is designed to be used as a support library for applications or libraries that need to solve both theoretical and practical instances of SAT. The expected user is a C/C++ programmer. In order to achieve that it has the following sub-objectives:

User objectives:

1. To be easy to use for the C/C++ programmer/developer.

2. To isolate the user from all difficulties of SAT solving.

3. To be the world most used trainable SAT solving programming library. ;)

Formal objectives:

1. To present an original, general, and practical alternative strategy to SAT solving.

2. To have a solid empirically checked algorithmic soundness and correctness.

Technical objectives:

1. To develop a software library that serves as solid "proof of concept" for the presented strategy.

2. To use other known SAT solving techniques as much as possible.

3. To do the less possible number of steps assuming unsatisfiability.

4. To do about the same number of steps for any permutation of the given instance.

5. To reuse as much information as possible during solving.

6. To use visualization techniques to debug target theoretical cases.

7. To optionally write proofs for unsatisfiable cases.

3   Status

ben-jose is in pre-alpha stage. It is research work in progress. Having said that, the author thinks it can serve as a proof of concept tool for the proposed strategy to solve SAT problems.

4   Functionality

4.1   Hello World example

The basic functionality of the library is best shown by the bj-hello-world.c program:

Listing 1: bj-hello-world.c program

```c
#include <stdio.h>
#include "ben_jose.h"

int main(int argc, char** argv)
{
  if(argc < 2){
     printf("args: <cnf_file_path>\n");
     return 1;
  }
  char* ff = argv[1];

  bj_solver_t ss = bj_solver_create("");

  bj_satisf_val_t vv = bj_solve_file(ss, ff);
  switch(vv){
```

```
16        case  bjr_yes_satisf :
17          printf ("%s_is_SAT_instance\n" ,  ff ) ;
18          break ;
19        case  bjr_no_satisf :
20          printf ("%s_is_UNS_instance\n" ,  ff ) ;
21          break ;
22        case  bjr_error :
23          printf ("ERROR_!_in_%s\n" ,  ff ) ;
24          break ;
25        default :
26          printf ("FATAL_ERROR_!_in_%s\n" ,  ff ) ;
27          break ;
28      }
29
30      // more  info  with  this  functions
31      //bj_output_t  oo = bj_get_output(ss);
32      //const  long* aa = bj_get_assig(ss);
33
34      bj_solver_release ( ss ) ;
35      return  0 ;
36   }
```

This program finds the solution to the SAT instance defined by the file "ff". The file must be in the simplified DIMACS format as described in http://www.satcompetition.org/2009/format-benchmarks2009.html.

Three functions are minimally needed to use the library bj_solver_create, bj_solve_file, and bj_solver_release.

4.2   Compiling

Once the library has been installed (see Section 7), a user program can be compiled using standard c++ compilation.

Assuming that the file libben-jose.a is in the directory BJ_LIB_PTH, the program in Listing 1 is build and compiled as in Listing 2:

Listing 2: compiling the bj-hello-world.c program

```
1  BJ_LIB_PTH = ../../../ build /gnu_make/bin
2
3  cc −o  bj−hello−world .o  −c  −MD  −Wall  −std=c99  −I ../../ library /api  bj−
        hello−world . c
4  cc −o  bj−hello−world  −rdynamic  −L${BJ_LIB_PTH}  bj−hello−world . o  −lben−
        jose  −lstdc++  −lgmpxx  −lgmp
```

4.3   Input format

The input format of a file called with bj_solve_filemust be the simplified DIMACS format.

The simplified DIMACS format is just that:

Listing 3: Simplified DIMACS CNF format example

```
1  c
```

```
2  c start with comments
3  c
4  c
5  p cnf 5 3
6  1 −5 4 0
7  −1 5 3 4 0
8  −3 −4 0
```

1. The file can start with comments, that is lines begining with the character c.

2. Right after the comments, there is the line p cnf nbvar nbclauses indicating that the instance is in CNF format; nbvar is the exact number of variables appearing in the file; nbclauses is the exact number of clauses contained in the file.

3. Then the clauses follow. Each clause is a sequence of distinct non-null numbers between -nbvar and nbvar ending with 0 on the same line; it cannot contain the opposite literals i and -i simultaneously. Positive numbers denote the corresponding variables. Negative numbers denote the negations of the corresponding variables.

5 Architecture

The architecture of the ben-jose library is strongly monolitic in the sense that the basic functionality of the whole library is basically one function with three presentations: bj_solve_file, bj_solve_data and bj_solve_literals.

That means that every piece of code within the "library" directory of the source tree is tightly coupled.

Having said that the following description of the library will give a good idea of it's architecture.

5.1 The Library

The following classes and names for attributes are the most important to explain how the solver works. They are explained in terms used in [9], [16] and [2].

5.1.1 DPLL+BCP+CDCL classes

To explain the most important parts of DPLL+BCP+CDCL, the following classes will be used.

neuron. class for CNF clause behavior. So there is one neuron per clause.

quanton: class for CNF variables (each variable has a positon and a negaton). There are two quantons per variable. neurons hold references to quantons called fibres. They are used for BCP.

prop_signal: class for representing BCP propagation data: which quanton fired by which neuron (which clause forced a given variable). BCP is done with the two watched literals technique (two watched fibres in the library's terminology).

deduction: class for holds the result of analyzing (doing resolution) of a conflict. It has the data for learning new neurons (clauses).

brain: class that holds all data used to solve a particular CNF instance. So there is one brain per CNF instance. It is created to solve an instance, and destroyed after solving that particular instance.

deducer: class that holds the data used to analyze a conflict.

leveldat: A level is all that happens between choices during BCP. So there is one level per choice. This class holds level relevant data.

5.1.2 Stabilization classes

The process of calculating a BDUST canonical form formula (BCFF) is called stabilization. The following classes will be used to explain the most important aspects of CNF stabilization:

sort_glb: It holds all global data used to stabilize a group of items (neurons and quantons representing a sub-formula of a CNF). It does not handle neurons and quantons, instead it handles sortees.

sortee: It is an item to be stabilized. Each neurons contains one sortee and each quantons contains one sortee. Each sortee "knows" (void pointer) which neuron or quanton contains it. It is a one-to-one relation that is used to stabilize CNF sub-formulas. During stabilization, the sort_glb handles the sortees not the neurons and quantons containing them.

sorset: It is a group of sortees. In order to stabilize a group of sortees the sort_glb class (or sortor) groups sortees (representing neurons and quantons in our case) into sorsets. A sub-formula is represented within stabilization by a group of sorsets. Each step of stabilization refines the group of sorsets that represent the stabilizing sub-formula, so that every step there are more sorsets, each one having less sortees, until the process cannot refine each sorset anymore. The ideal stabilization ends with each sorset containing only one sortee. Since stabilization handles only sortees. This class is used for such iterated sub-grouping.

sortrel: It represents a relation between two sortees. In our case every sortee representing a neuron holds one sortrel per fiber (literal), and each sortee representing a quanton holds one sortrel per neuron in wick the quanton is found. They must be properly initiated before each stabilization. They define the stabilizing sub-formula's relations between it's neurons and quantons by relating their respective sortees. They represent relations between a particular sub group (sub-formula) of neuron's sortees and quanton's sortees.

5.1.3 Matching classes

Matching consists basically of two steps. Stabilization and finding the resulting BCFF in the database of BCFFs. The following classes will be used to explain the most important aspects of CNF matching:

coloring: The initial and final state for an stabilization is a coloring. A color is just an integer. A coloring of a sub-formula is an assignation of an integer (neuron-color) to each neuron and an integer (quanton-color) to each quanton of the sub-formula. An stabilization may start with all neurons having the same neuron-color and all quantons having the same quanton-color and finalize with each neuron having a unique neuron-color and each quanton having a unique quanton-color, called a complete coloring.

colorings are "loaded" into the sort_glb class in order to start stabilization. After stabilization the final coloring may be "saved". Each color in a coloring will correspond to one sorset during stabilization.

This class is used to specify only the input to the stabilization process (the initial state). The class canon_cnf is used for the output (it is the result of applying the output coloring (stabilized coloring) to the sub-formula it defines: neurons and quantons in the coloring. To initialize the sortor, it "loads" the initial coloring into the sort_glb instance that will stabilize the CNF sub-formula.

canon_cnf: It is a BCFF. It represents the output of an stabilization process: the stabilized CNF sub-formula. It is the interface class to the database class that handles all disk operations

(the skeleton class). This class contains some disk handling related information (paths and sha info). A canon_cnf basically is a set of canon_clauses (which are basically arrays of numbers).

neuromap: This class represents a CNF sub-formula. It is the pivot class to do all stabilization. It is maintained during BCP and used during backtracking in order to know what CNF sub-formulas are to be stabilized and searched for in the database (skeleton class). There is one neuromap per leveldat and they are either active or inactive. Active when they are candidates for stabilization, matching and search in database (or saving), at backtrack time. When a CNF sub-formula, during search, is found to be unsatisfiable, is not trivial (BCP could not figure it out), and both search branches had the same variables (so that it can latter be searched only with one of them), it is saved, stored in the database (skeleton class). Every time an still active neuromap has done its first branch of BCP, it is stabilized and searched for in the database (skeleton class). Trivial sub-formulas are called anchors in the code because they serve as a start point for stabilizing not trivial ones.

5.1.4   Database classes

The skeleton class handles all disk related functions and management. The database is basically a directory and all its sub-directories in disk. The directory (skeleton) is seen as a group of ("key","value") pairs. Just like a common database "index", a "dictionary" class, or a "map" class. A path within the skeleton is a "key" and the files in the path are the "value". To see if a "key" exists is to see if a path exists within the skeleton. Unsatisfiable canon_cnfs are saved and searched by the SHA function of their content. They are saved in a path ("key") that is constructed with the SHA and other relevant search info.

Since an unsatisfiable sub-formula might not be minimal (have some unnecessary clauses for unsatisfiability), each unsatisfiable CNF sub-formula has three relevant canon_cnf:

1. The guide. It is the canon_cnf resulting of stabilizing the CNF sub-formula covered by first search branch variables. So it is a satisfiable part of the unsatisfiable CNF sub-formula that is a "guide" for the search.

2. The tauto. It is the full unsatisfiable CNF sub-formula. It is the canon_cnf resulting of stabilizing the CNF sub-formula covered by both search branches charged quantons (used variables).

3. The diff. This canon_cnf contains all canon_clauses in tauto but not in guide. Each diff is saved in a path called 'variant' in the skeleton. So one guide can have several variants.

A search of a target CNF sub-formula is conducted in two phases: the search for the guide of the target and the search for the variant that is a sub-formula of the target diff. Once the guide is stabilized the search for it is a simple: "see if its path exists" (remember that its path contains the SHA of its content). If the target canon_cnf is not equal to a variant (the path does not exist), the second phase is more time consuming because it involves reading each variant and comparing it to the target diff to see if the the variant is a sub-formula of the target diff (which would mean that the target is unsatisfiable and therefore can be backtracked).

5.1.5   The solve_instance function

To help in the understanding of the functioning of the ben-jose library and to get familiarized with the backbone functions of the library look at the file:

```
ben-jose/src/library/api/macro_ben_jose.cpp
```

5.2   Goodies

The library comes with some tools.

5.2.1   Proofs

ben-jose can optionally generate proofs for unsatisfiable formulas in JSON format. They can serve as input to an automated proof checker.

Assuming the program "`bj-batch-solver`" is in the path, and from the source tree directory:

```
cd ben-jose/src/programs/tests/solver
sh clean_sh
bj-batch-solver ./h4.cnf -proof -rr
firefox ./SKELETON/CNF/Show_Proof.htm &
```

the `bj-batch-solver` program will use the `bjp_write_proofs` library option to generate the proof of the unsatisfiable instance "`h4.cnf`" in JSON format.

The proof can be read using the HTML page:

```
./SKELETON/CNF/Show_Proof.htm
```

and copying the (very long) path printed in the standard output under "`LAST_PROOF_PATH=`" to the text area field in the `Show_Proof.htm` HTML page in your favorite `firefox` navigator, and then clicking "`LOAD_FILE`". Paths must begin with "`/SKELETON`", so after copying, get rid of the prefix before it, before clicking "`LOAD_FILE`".

Remember to run

```
sh clean_sh
```

before **each** time you run the `bj-batch-solver` program in order to see the **whole** proof.

5.2.2   PHP visualizing

A `bj-phi` program that generates HTML pages with `cytoscape.js` (http://js.cytoscape.org/) graphs to visualize the solving of PHP instances (see Section 6).

5.2.3   Debugging

ben-jose has a configurable built-in debugging system (debug directory) to help in the debugging of the library. The `bj-solver-debug` program is compiled with the `libdbg-ben-jose.a` library for that purpose.

6   Use case

The target case when development of ben-jose started was PHP instances. It is a well studied theoretical case that shows exponential behavior for RES.

ben-jose comes with a program that helped in the development of ben-jose called `bj-phi`.

6.1   `bj-phi`

Assuming the program "`bj-phi`" is in the path, and from the source tree directory:

```
cd ben-jose/src/programs/tests/phi
sh clean_sh
bj-phi 4
firefox ./phi_hole_4/phi_hole_4_all_steps.htm &
```

will execute the `bj-phi` program for PHP-4 and it will use the file

```
./dbg_ben_jose.conf
```

to configure the debugging system to generate an HTML log file of the the solving process for PHP-4. It will be written in:

```
./phi_hole_4/phi_hole_4_all_steps.htm
```

It is very important to observe that the contents of **the file is different for the second time you run the `bj-phi` program** because the second time the ben-jose finds BCFFs that were written in the first run. The second time ben-jose will find the sub-formulas PHP-2, PHP-3, and itself PHP-4 to determine unsatisfiability in the first backtrack. This behavior can be observed in this HTML log file.

In order to restart the original state before the first run: that is to delete the skeleton and the generated files by `bj-phi` simply run:

```
sh ./clean_sh
```

The HTML log file, when opened with your favorite firefox navigator, should show you a web page with graphs of the following style:
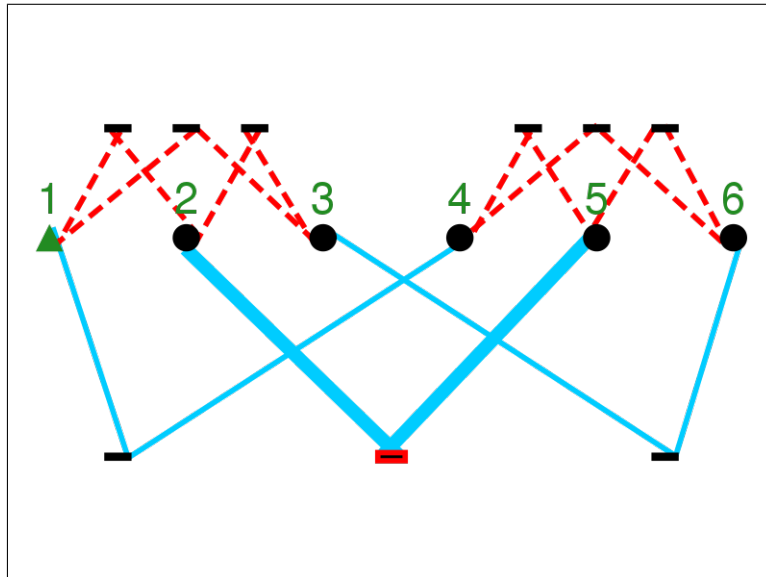

Figure 1: bj-phi representation of a PHP instance of size 2

The image represents the DIMACS-CNF file:

Listing 4: PHP of size 2 DIMACS file

```
1   p cnf 6 9
2   −1 −2 0
3   −1 −3 0
4   −2 −3 0
5
6   −4 −5 0
7   −4 −6 0
8   −5 −6 0
9
10   1  4 0
11   2  5 0
12   3  6 0
```

Where dashed lines represent negative literals and solid lines represent positive literals. Circles represent variables and little rectangles represent clauses.

In Figure 1 the thick little rectangle joining two thick solid lines represent the clause [2 5 0] of line 11 in Listing 4.

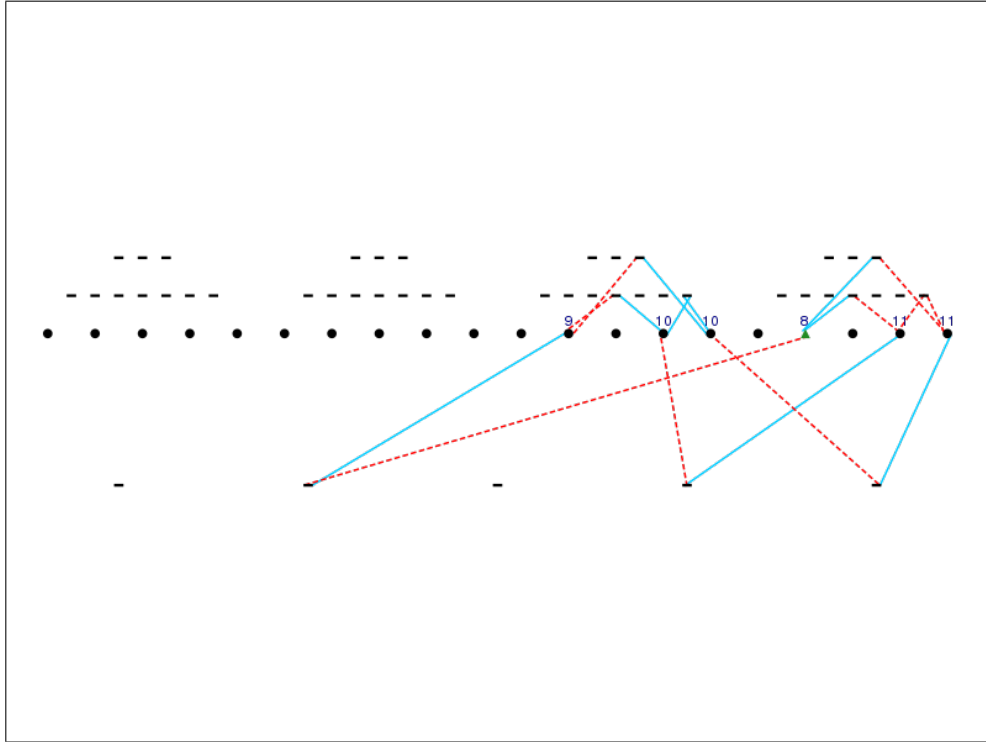The graphs shown in the HTML page have more implicit information:

Figure 2: bj-phi graph for the first time PHP-2 is matched within PHP-4 solving

The equivalent consecutive numbering of variables shown in Figure 1 is assumed in Figure 2 according to their position from left to right, therefore the numbering shown in Figure 2 is not that of the literal but the "tier" of the "charged" quanton. In Figure 2 variable 17 was charged

in tier 8, variable 12 has tier 9, variables 14 and 15 have tier 10, and variables 19 and 20 have tier 11.

The dashed and solid lines in Figure 2 do not represent the sign of literals as in Figure 1. In Figure 2 the signs of literals are also assumed according to their position in the graph: all above the circles are negative and all below are positive.

The dashed lines and solid lines in Figure 2 represent the current assignation of a variable:

1. If the lines match the original state (were dashed and still dashed) the variable is positively charged.

2. If they do not match the original, the variable is negatively charged.

In Figure 2 variables 12, 19 and 20 are positive, and variables 14, 15 and 17 are negative. The little triangle indicates a choice therefore in Figure 2 variable 17 is a choice.

7   Installation

To install ben-jose:

1. Download and unzip a ZIP file from the library from https://github.com/joseluisquiroga/ben-jose.
2. `cd ben-jose-master/build/gnu_make`
3. `make -s`
4. `cd lib/ben-jose`
5. `ls`

That will built and list all the files of the library. Put the directory in the path.

7.1   Required Packages

There are minimal requirements for building and compiling ben-jose:

- A Linux system.
- GNU c++ (g++) installed.
- GMP Library (gmpxx) installed.

Almost every Linux system comes with the second two in the basic installation, so there is basically one requirement: a Linux system.

8   Comparison with other tools

8.1   Trainable

The most important difference of ben-jose with any other solver available today is the fact that it matches sub-formulas to already known to be unsatisfiable formulas (BCFFs) in order to skip the search on sub-formulas that match.

That technique allows this solver to be trainable. It holds a database of BCFFs that can be reused in later instances of SAT.

In order to archive this behavior some important modifications had to be done to the standard way of doing DPLL+BCP+CDCL. Learned clauses (neurons) are kept only until they are backtracked. That imposes a performance penalty when that clause can actually prune the search after it has been backtracked since it has to be deduced again by RES.

8.2   Monos

They are variables (quantons) that occur either only in positive form, or only in negative form after a BCP step, that is a level has been processed (leveldat hold the data for levels). In the code they are called "monos". So:

- When it is detected that there are only positons of a variable in the current sub-formula, the variable can safely be assumed to be set "true".

- When it is detected that there are only negatons of a variable in the current sub-formula, the variable can safely be assumed to be set "false".

  This difference is crucial for to ease the matching of BCFFs.

8.3   Longest BCP

The solver uses a kind of "look up technique" that always chooses the "longest" path for BCP.

When ever a choice is been made it tries both values in order to see which value propagates more variables.

This difference is also crucial for to ease the matching of BCFFs.

In the code it can bee seen in the function "start_propagation" when it calls "select_propag_side".

9   Future Work

Things to do with ben-jose:

- Test, test and test. And when necessary debug, debug and debug.

- Reduce combinatoric redundancy for PHP.

- Make a first release.

- Follow in detail for is-prime instances. It implies writing code to generate cytoscape.js graphs that help visualize the solving process.

- Write a concurrent version of ben-jose. Since a search branch can use the work done by an other search branch (use BCFFs written by the other branch) it is conceptually efficient to do concurrency.

  The code has already considered this possibility by using the file system as a database and considering concurrency during file access.

10   Acknowledgments

1. Our heavenly Father YHWH (Yahweh).

2. Our Lord Yashua (Jesus Christ).

3. Magda Beltran de Quiroga (my mother).

4. Federman Quiroga Rios (my father).

5. Joao Marquez da Silva for his work on the SAT problem.

6. All the authors in the bibliography.

References

[1] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning sat solvers. http://www.aaai.org, 2010.

[2] Oliver Bastert. Stabilization procedures and applications. Zentrum Mathematik. Technische Universitat Muchen, 2002.

[3] Armin Biere. Handbook of satisfiability, 2009.

[4] Samuel R. Buss and Gyorgy Turan. Resolution proofs of generalized pigeonhole principles, 1988.

[5] Stephen Cook. The importance of the p versus np question. JACM 50, 1 (50th Anniversary Issue), pp 27-29, 2003.

[6] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. SIGACT News 8(4):28-32, 1976.

[7] Stephen A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. Journal of Symbolic logic. 44, pp. 25-38, 1979.

[8] Dawson Engler Cristian Cadar, Daniel Dunbar. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. http://klee.github.io, 2008.

[9] Joao Marques da Silva. Search algorithms for satisfiability problems in combinatorial switching circuits. a dissertation submitted in partial fulfillment of the requirements for the degree of doctor of philosophy (electrical engineering). The University of Michigan, 1995.

[10] Heidi E. Dixon, Matthew L. Ginsberg, David Hofer, Eugene M. Luks, and Andrew J. Parkes. Generalizing boolean satisfiability i, ii and iii: Background and survey of existing work, theory and implementation. Journal of Artificial Intelligence Research 21, 22 and 23, 2004.

[11] Niklas Een and Niklas Sorensson. An extensible sat-solver. Chalmers University of Technology, Sweden, 2004.

[12] A Haken. The intractability of resolution. Theoretical computer science, 39, pp. 297-308, 1985.

[13] Matti Jarvisalo. Impact of restricted branching on clause learning sat solving. Research Reports 107. pp.28. Helsinki University of Technology Laboratory for Theoretical Computer Science, 2007.

[14] Daniel Kroening and Ofer Strichman. Decision procedures: An algorithmic point of view, 2008.

[15] Victor W. Marek. Introduction to mathematics of satisfiability, 2009.

[16] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. DAC 01. Proceedings of the 38th annual Design Automation Conference, 2001.

[17] Matthias Muller. Polynomial sat-solver. http://vixra.org/pdf/1212.0109v2.pdf, 2013. Ben-Jose author thinks this informal proof is valid.

[18] J. L. Quiroga. Optimal circuit verification method. http://www.google.com/patents/US20040250223, 2001. Quiroga is ashamed of this title.

[19] G. Tseitin. On the complexity of proofs in propositional logics. Automation of Reasoning: Classical Papers in Computational Logic 1967-1970. volume 2. Springer- Verlag, 1983.

[20] GJ Woeginger. The p-versus-np page. http://www.win.tue.nl/~gwoegi/P-versus-NP.htm, 2016.