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 [8] and latter refined by others [15] [10]. 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. That in order to entirely skip 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

[*]Especial thanks to our heavenly Father, his anointed King, Magda Beltran de Quiroga and Federman Quiroga.

1 Introduction

The satisfiability problem (SAT) is the canonical decision problem by excellence [3] [13] [14]. 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” [19] [16], and filed patent applications for optimal SAT solvers based on resolution [17].

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 highlights the importance of SAT.

Exponential lower bounds of resolution (RES) proof size for the pigeon hole principle (PHP) have been proven [11] [4], and polynomial size proofs for extended resolution (ER) have also been proven for the PHP [6] [7] [12] instances of the SAT problem. That work suggests that solvers based on RES [8] [15] [10] need ”something else” in order to be ”faster” [9] [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 (subsumed isomorphism) 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), and has linear complexity with respect to the size of the sub-formulas. The structures learned with BDUST 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”.

This form of extending RES is not ER, as the system presented by Tseitin [18], 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.
2. To isolate the user from all difficulties of SAT solving.
3. To do 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 use other known SAT solving techniques as much as possible.
3. To have a solid empirically checked algorithmic soundness and correctness.

Technical objectives:

1. To do the less possible number of steps assuming unsatisfiability.
2. To do about the same number of steps for any permutation of the given instance.
3. To reuse as much information as possible during solving.
4. To use visualization techniques to debug target theoretical cases.
5. To optionally write proofs for unsatisfiable cases.

3 Functionality

3.1 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

```
#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){
    case bjr_yes_satisf:
      printf("%s is SAT instance\n", ff);
      break;
    case bjr_no_satisf:
      printf("%s is UNS instance\n", ff);
      break;
    case bjr_error:
      printf("ERROR ! in %s\n", ff);
      break;
    default:
      printf("FATAL ERROR ! in %s\n", ff);
      break;
  }

  // more info with this functions
  //bj_output_t oo = bj_get_output(ss);
  //const long* aa = bj_get_assig(ss);

  bj_solver_release(ss);
  return 0;
}
```

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.

3.2 Compiling

Once the library has been installed (see Section 6), 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

```
BJ_LIB_PTH=../../../build/gnu_make/bin

cc -o bj-hello-world.o -c -MD -Wall -std=c99 -I../../library/api bj-
    hello-world.c
cc -o bj-hello-world -rdynamic -L${BJ_LIB_PTH} bj-hello-world.o -lben-
    jose -lstdc++ -lgmpxx -lgmp
```

3.3 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

```
c
c start with comments
c
c
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-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.

4 Architecture

(Describe here the architecture)

4.1 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 [8], [15] and [2].

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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 Use case

(Describe here the use case)

6 Installation

Since there is no current release version of the library, it has to be installend from source. To do so:

1. Download the library from https://github.com/joseluisquiroga/ben-jose.
2. Build and install the library following the instructions in the README.txt file.

The README.txt file refers you to the installation alternatives:

1. To build and install with autotools follow:
   ./build/gnu_autotools/README.to_install_with_autotools
2. To build with gnu_make follow:
   ./build/gnu_make/README.to_install_with_gnu_make

It is very probable that when installing with autotools you will need to first follow the file:
./build/gnu_autotools/README.when_clean_all_non_human_made
because the source tree is usually cleaned from all non human made files.

6.1 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.

7 Comparison with other tools

7.1 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 achive this behavior some important modifications had to be done to the standard way of doing DPLL+BCP+CDCL. Learned clauses (neurons) are keept 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.

7.2 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 varible 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 varible 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.

7.3 Longest BCP

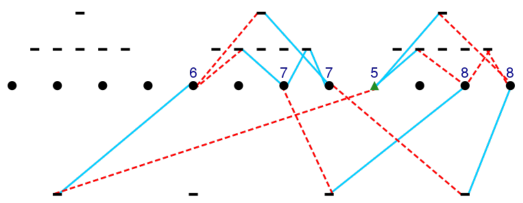
The solver uses a kind of "look up technique" that always chooses the "longest" path for BCP.

When ever a choice is been made it tries both values in orther to see wich 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".

8 Future Work



(Describe here future work)

9 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] 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.

[9] 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.

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

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

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

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