iSat: Structure Visualization for SAT Problems

Ezequiel Orbe, Carlos Areces, Gabriel Infante-López*

Grupo de Procesamiento de Lenguaje Natural
FaMAF, Universidad Nacional de Córdoba, Argentina
{orbe|areces|gabriel}@famaf.unc.edu.ar

Abstract. We present **iSat**, a Python command line tool to analyze and find structure in propositional satisfiability problems. **iSat** offers an interactive shell to control propositional solvers and generate graph representations of the internal structure of the search space explored by them for visualization, with the final aim of providing a unified environment for propositional solving experimentation. **iSat** was designed to allow the simple integration of both new provers and new visualization graphs and statistics with a minimum of coding overhead.

1 Introduction

iSat[1] (interactive SAT) is a command line tool implemented in Python that helps a user to analyze and find structure in propositional satisfiability problems. It can be used, for example, to investigate the behavior of different provers over a given test set. The main service offered by **iSat** is a unified interface for experimentation with different theorem provers for propositional logics (SAT solvers) and visualization graphs. Moreover, it can be use to mechanize the repetitive tasks often performed during the development of SAT solvers (e.g., fine tuning heuristics) or the selection of the appropriate configuration options for a given solver working on a particular satisfiability problem. **iSat** computes different visualization graphs (e.g., Variable-Clause , Variable, Interaction, etc.) over the current clause set at different points during the exploration of the search space, and computes related statistics over these graphs (degree mean/max/min/standard deviation, clique number, clustering, number of cliques, etc.). **iSat** was designed to facilitate the integration of new provers, visualization graphs and statistics with a minimum of coding overhead. **iSat**is distributed under a GPL license and currently supports two SAT solvers out of the box: Minisat [3] and CryptoMinisat [11].

1.1 A Brief Overview on SAT Solving

Propositional satisfiability is the problem of deciding if there exists a Boolean assignment to variables, such that all clauses in a given propositional formula

* Consejo Nacional de Investigaciones Científicas y Técnicas
[1] Available at `https://cs.famaf.unc.edu.ar/~ezequiel/software/isat/`.

evaluate to true. Despite its complexity, current SAT solvers (e.g., [3,8,5]) efficiently solve many instances of the SAT problem.

Current SAT solvers can be classified in two broad classes: *incomplete* and *complete* systems. Incomplete solvers performs different kinds of stochastic local search to find a satisfying valuation; if the search is successful, satisfiability is established, but search failure does not imply unsatisfiability. Complete SAT solvers, on the other hand, perform an exhaustive, systematic search, and hence can establish both satisfiability and unsatisfiability. Most of them implement variants of the Davis-Putnam-Logemann-Loveland algorithm (DPLL) [2,1]. Complete SAT solvers can be further classified into conflict-driven and look-ahead. Conflict-driven solvers augment DPLL with conflict analysis, clause learning, non-chronological backtracking and restarts as its principal components. Look-ahead solvers, are also based on DPLL but invest substantial efforts choosing first the branching variable to be used (the different choice options are called decision heuristics) and then the truth value this variable will be assigned (using so called direction heuristics) aiming to achieve the largest reduction of the remaining search space. [4] provides an excellent overview of the area.

The SAT solving community is large and very active, with strong industrial involvement on application areas like planning [6], test pattern generation [7], etc. As a result of this demand, new algorithms and heuristics are being constantly developed, and the available solvers tuned to obtain the best behavior on particular problem domains. But interacting with solvers to gather statistics and explore their behavior when solving particular problems in order to find the best configuration parameters for a given problem class is a burdensome task.

iSat is a command line tool developed in Python that provides an interactive shell for multiple solvers and is capable of producing a visualization graphs and statistics, with the final aim of providing a unified environment for SAT solving experimentation. Currently, **iSat** provides access to the Minisat [3] and the CryptoMinisat [11] solvers; produces Variable-Clause graphs, Variable graphs, Literal-Clause graphs, Literal graphs and Interaction graphs that can be exported in `gml` and `dot` format and can be visualized using, for example, Cytoscape[2]; and computes statistics over these graphs like degree mean/max/min/standard deviation, clique number, clustering and number of cliques. Moreover, the architecture of **iSat** has been designed to allow the simple integration of new solvers and new analysis tools (i.e., new visualizations and statistics).

2 A Sample Session

We will describe a tipical session with **iSat** to illustrate its capabilities. The screen capture of the interaction can be seen in Figure 1.

Consider the case of a researcher who wants to visualize how the structure of a pigeon hole problem instance evolves during search. She suspects that if she can identify some structural properties of the problem, she could develop new

[2] `http://www.cytoscape.org/`

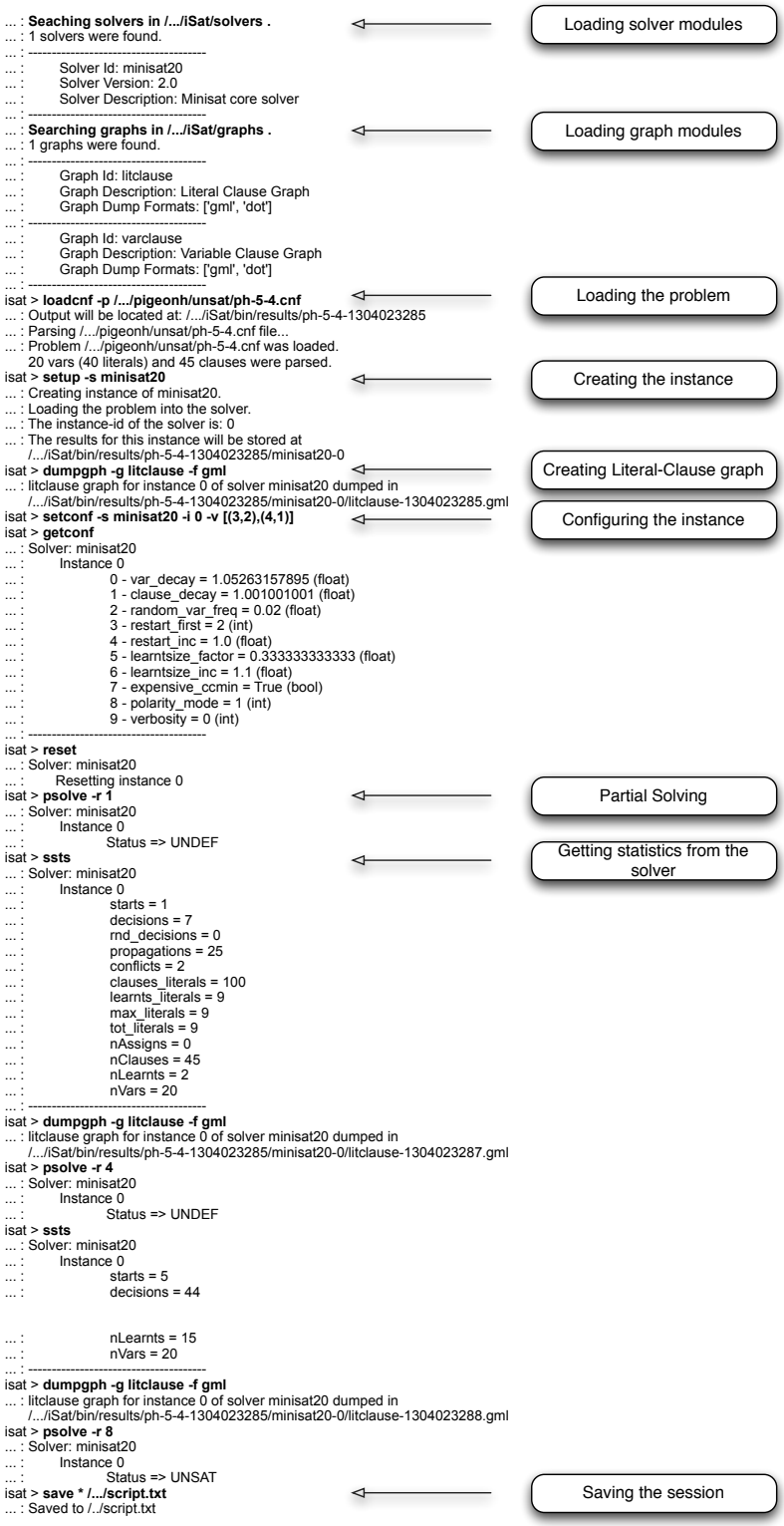


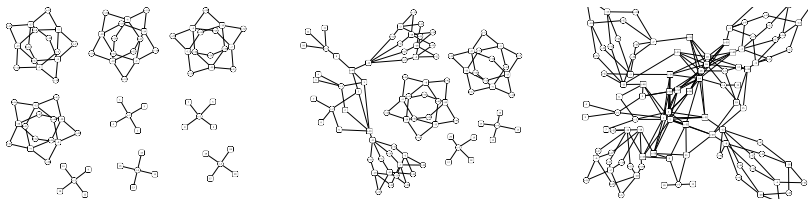
Fig. 1. A typical session.

heuristics to improve the search. She uses **iSat** with Minisat as SAT solver and the Literal-Clause graph to visualize the problem structure.

She loads (command **loadcnf**) the problem instance into the tool. **iSat** first parses and loads the problem, and then, creates an output folder where all result files of the session will be stored. She then creates an instance of Minisat (command **setup**) and **iSat** loads the problem into the solver and creates an output folder where all files related to this instance will be stored. Then she builds the Literal-Clause graph (command **dumpgph**) for the original problem. She plans to use it as a baseline against which to compare the different graphs she is planning to generate during the rest of her session. Before solving the problem, she configures the solver instance (command **setconf**) to make it restart its search after having found only 2 conflicts and to keep this upper bound constant during the next restarts. Next, she checks that the other configuration options of the instance are correctly set (command **getconf**).

Now, she can start the process of exploring how Minisat solve the problem. Given that she wants to see the structure of the problem at different points, she runs a partial solving of the problem (command **psolve**)[3]. Once the partial solving ends, she retrieves some statistics from the solver (command **ssts**) to check how the search is performing, and then builds another graph representation from the current state of the problem. She repeats this process until the problem is proved to be UNSAT. After that, she saves its activity as a script (command **save**) so she can easily re-run her experiments later.

After quitting **iSat** she will be able to visualize and analyze the generated graphs (see Figure 2) using a suitable graph analysis tool like Cytoscape, looking for structural properties that can be used in the heuristic she is developing.



(a) (b) (c)

Fig. 2. Evolution of the problem structure: a) Original problem. b) After one restart. c) After five restarts.

3 Services Provided by iSat

SAT solvers are complex procedures that iteratively modify the internal state of the set of clauses still to be solved and a partial assignment. The computation starts with the initial state given by the set of clauses in the original formula

[3] Currently, **psolve** in Minisat stops search after a specific number of restarts. Other options are possible (e.g., returning the control to **iSat** after a fix number of steps).

and an empty assignment. The solver modifies this state step by step adding and removing clauses, and assigning variables. **iSat** is a tool that instruments this computation. It allows a user to retrieve the solver state, explore it, represent it as a graph and manually modify it.

iSat can run many instances of the same or different solvers in the same session. It allows a user to compare different intermediate states that might come from different solvers or from the same solver at different stages. Since **iSat** groups instances according to the underlying SAT solver, it is possible to interact with one specific instance, with all instances of one specific solver or with all instances of all solvers. The current version of **iSat** can interact with Minisat and CryptoMinisat, but other SAT solvers can be easily integrated.

The tool also allows the user to save sessions and to reproduce them as scripts. The shell interface (implemented with `cmd2`[4]) allows the user to retrieve command history, search the history, and execute Python code and shell commands. It provides means to inspect the computation state through visualization graphs. Currently, **iSat** can compute Interaction graphs [10], Variable-Clause graphs [9], Variable graphs [9], Literal-Clause graphs and Literal graphs (the last two are similar to the previous two but using literales as nodes instead of variables). The user can select at any time during the exploration the type of graph and the instance whose state she wants to export. The graph is then generated as a file for further analysis.

Commands available in the interactive shell can be grouped in four categories: C1) those that handle and modify the state of the SAT solver, C2) those used to inspect the current state (by means of relevant statistics or visualization graphs), C3) commands to create instances of a problem; and C4) commands to save sessions and to execute saved sessions. Most relevant commands are described in Figure 3.

iSat is a powerful tool that offers users relevant information. It allows the user to interact with different solver instances easily and intuitively. Moreover, its architecture allows easy integration of new solvers and visualization graphs.

4 Extending iSat

iSat has been designed to be easily extended to include new sat solvers and different visualization graphs, with their respective statistics.

iSat uses a Client/Services architecture. The Client layer implements the user interface as an interactive shell. The Services layer provides the interface to different solvers and visualization graphs. Services can be either Solvers or Graphs. A graphical description of the architecture is shown in Figure 4.

Two components are needed to integrate a new SAT solver into **iSat**. The first wraps the API of the SAT solver into Python and provides Python bindings. Since this wrapper only traduces the SAT solver's API into Python, the resulting bindings might not be the ones required by the interactive shell. The second

[4] `http://packages.python.org/cmd2/`

C1	`addc`: Adds a list of clauses to a given instance. If no instance is given, they are added to all instances. `simplify`: Simplifies the set of clauses in a given instance. If no instance is given, it simplifies all instances. `solve`: Attempts to solve the problem in a given instance; it lets the solver compute a final state. If no instance is given, it solves all instances. `psolve`: Performs a partial solve in a given instance. If no instance is given, it partially solves all instances using the same number of restarts. `reset`: Resets the internal state. If no instance is given, it resets all instances.
C2	`ssts`: Gather statistics from the given instance. If no instance is specified, it gathers statistics from all instances. `gsts`: Gathers statistics from a specific visualization graph. If no instance is given, it gathers statistics from all instances. `dumpgph`: Generates a file with the visualization graph of the current state of a given instance. If no instance is specified, it generates visualization graphs for every instance. `getconf`: Gets the current configuration options from a given instance. If no instance is specified, it gets the current configuration options from all instances.
C3	`loadcnf`: Loads a problem into memory and creates a folder where the results of the session are stored. `setup`: Creates an instance of a solver and feeds it with the last problem that was loaded using `loadcnf`. For each instance, it creates a subfolder where it outputs information particular to the instance. `setconf`: Configures a SAT solver instance with the given configuration options.
C4	`save`: Saves the current session as a script. `load`: Loads and execute a script file.

Fig. 3. iSat commands

component addresses this issue adapting the wrapper to the specific needs of the interactive shell. Both components are SAT solver dependent and both have to be implemented when a new solver is added to **iSat**.

For solvers developed in C/C++ (this is the case for most current SAT solvers), the first component can be defined with the help of tools like the Simplified Wrapper and Interface Generator (Swig)[5] or the Boost libraries[6] which assist in the definition of bindings for a number of target programming languages, including Python. But even with the help of these tools, defining this component requires careful work and knowledge of the particular solver involved. In particular it is in this component where we should ensure that the resulting Python bindings provide all the necessary basic functionality required by **iSat** (like the ability to stop the run of the solver at a certain point, and retrieve the current state). For example, to build the Python bindings for Minisat, its original C++ API was first extended in the native language to provide the missing functionality required by **iSat**. Besides providing a simplified interface to some of the methods already present in the solver, this extension includes a partial solving method that continues the search till the next restart, and methods that returns the current set of clauses and learnt clauses. The Python bindings for this extension were then obtained using Swig.

The second component, on the other hand, is mostly bookkeeping, and adapts the previous functionality to the concrete function interfaces and datatypes used by **iSat**. In particular, implementing this component boils down to the definition of a subclass of the Python class `Solver` and uses the functionality provided

[5] `http://www.swig.org/`
[6] `http://www.boost.org/`

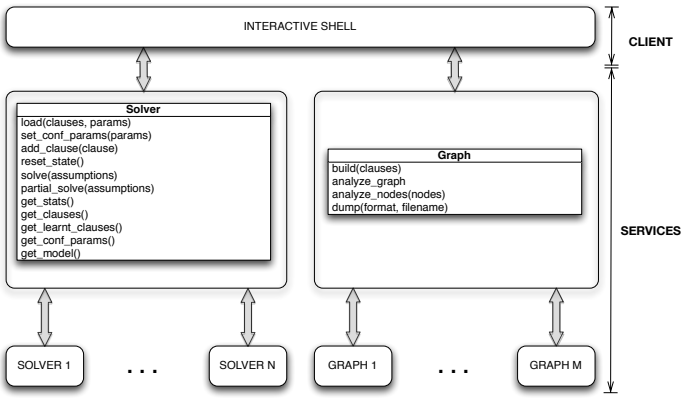


Fig. 4. The architecture of **iSat**.

by the wrapper in its implementation. The `Solver` class interface is shown in Figure 4. Methods in this class can be grouped into three categories: configuration methods, information methods, and solving methods. In the configuration category we have methods to load a problem in the solver (`load` and `add_clause`), methods to configure the solver parameters (`set_conf_params`), and methods to reset the internal state of the solver (`reset_state`). In the information category, we have methods that grant access to the internal state of the solver (`get_clauses`, `get_learnt_clauses`), a method to obtain statistics (`get_stats`), a method that returns a model of the current problem, if available (`get_model`), and a method that returns the current configuration parameters (`get_conf_params`). Finally, the solving category includes methods to run the solver, or to run it till a certain predefined condition is met, over the current problem (`solve` and `partial_solve`).

Integrating new visualization graphs into **iSat** follows a similar strategy but is usually simpler as we don't have to deal with the internal complexity of a SAT solver. The common interface is defined by the class `Graph` also shown in Figure 4. This interface includes methods to build the graph (`build`), dump the graph to a file (`dump`), and gather statistics both at graph and at node level (`analyze_graph` and `analyze_node`). There is no restriction on how the graph is implemented internally, or on how the statistics are gathered. Graphs modules already implemented in **iSat** has been developed using the Python package NetworkX[7] to build graphs and to gather associated statistics.

5 Conclusions

iSat is an interactive command line tool that can be used to investigate the internal structure of the search space explored by propositional solvers. It can be used to assist developing new heuristics and to compare different stages of the same or

[7] `http://networkx.lanl.gov/`

different solvers. **iSat** also provides different graphical visualizations of the current problem state, and related statistics. The current version of **iSat** provides the general architecture, integration with the Minisat and CryptoMinisat solvers, and implementations for computing Variable-Clause, Variable, Literal-Clause, Literal and Interaction graphs; it provides access to the statistics obtained from Minisat and CryptoMinisat (number of decisions made, propagations, conflicts detected, current number of variables, etc.) together with statistics over the graphs computed (degree mean/max/min/standard deviation, clique number, clustering, number of cliques, etc.). **iSat** was developed with two concrete design goals in mind: to simplify extensibility and to provide an agile interaction with different provers. The outcome is a unified interface for experimentation where new SAT solvers and visualization graphs can be easily integrated.

In contrast to tools like DPViz [10], which mainly have a pedagogical purpose, the main target of **iSat** are researchers investigating the behavior of SAT solvers over particular problem instances (e.g., SAT solver developers) or skilled users fine tuning a SAT solver for a particular application (e.g., SAT solvers used in particular industrial applications).

This is the first release of **iSat**. We are currently working on the integration of new SAT solvers, and different graph visualizations. The flexibility offered by the current implementation opens the way to many customizations possibilities. It would be interesting to see in which ways the SAT solving community will make use of **iSat** and contribute to its development.

References

1. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. of the ACM*, 5(7):394–397, 1962.
2. M. Davis and H. Putnam. A computing procedure for quantification theory. *J. of the ACM*, 7(3):201–215, 1960.
3. N. Een and N. Sörensson. An extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, 2004.
4. M. Heule. *SmArT solving: Tools and techniques for satisfiability solvers*. PhD thesis, TU Delft, 2008.
5. M. Heule and H. van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *J. on Sat., Boolean Modeling and Comp.*, 2:47–59, 2006.
6. H. Kautz and B. Selman. Planning as satisfiability. In *Proc. of ECAI-92*. John Wiley and Sons, Inc, 1992.
7. J. Marques-Silva and K. Sakallah. Robust search algorithms for test pattern generation. In *Proc. of the Fault-Tolerant Computing Symp.* IEEE, 1997.
8. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the 38th Design Automation Conf.*, 2001.
9. E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, and H. Hoos. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Int. Conf. on Principles and Practice of Constraint Programming (CP)*, pages 438–452, 2004.
10. C. Sinz. DPvis - a tool to visualize the structure of SAT instances. *Theory and Applications of Satisfiability Testing*, 2005.
11. M. Soos. CryptoMiniSat — a SAT solver for cryptographic problems, 2009. `http://www.msoos.org/cryptominisat2`.