SAT Solver

Lorenzo Ciampiconi
School of Computing
National University of Singapore

1 Introduction

This document aim to present the SAT solver developed by Lorenzo Ciampiconi during his semester in National University of Singapore.

2 Design of the Sat-Solver

2.1 Enconding of variables

I've begun the design of the SAT-solver by choosing how to represent a formula with n variables and m clause. Because i decided to use python language, that it has a lot of good library for handling matrices, i tried to encode the formula to be solved in a matrix $n \times m$ A, $a_{ij} \in \{0, 1\}$. I've done the implementation of DPLL algorithm with this representation but i find out that this was really demanding representation as the row of the column were really sparse and lot of memory was burnt and also a lot of time iterating on it. So i decided to encode a clause as a list of integer, so that if $c = x_1 \vee \neg x_4 \vee x_6$ this will be represented as a python list [1, -4, 6]. This has reduced a lot the memory usage and time to run the basic function of the sat.

2.2 Unit-Propagation

The phase of unit propagation was probably the one that i struggle the most with. I was trying to optimize the passage of propagate a literal in the way to don't have to go through every clause every time to see if a new assignments has been inducted or the clause cause contradiction.

2.2.1 First Solution: Induction Graph

The first thing that I tried was to build an *Induction Graph* where clauses were organized and there was a relation between them represented as a *directed edge*. So the solver was considering in the graph only the clause that were not satisfied at the moment. If a clause c_1 was constraining a set of *unassigned* variable X_1 and clause c_2 was constraining a set X_2 such that $X_1 \subseteq X_2$, then c_2 was son of

c_1 and a directed edge was considered between them. Every time a literal was supposed to be propagated the solver was going through the higher level of the graph, so through the *nodes* (clauses) that have no parents (that are not son of any other clause), and if a new assignment was inducted by one of these nodes p *then* was a *necessary* condition to induct other assignments from his sons, otherwise the sons of p were not *strong* enough to propagate new literal, so they were not considered. This operation was repeated until no analyzed node was propagating a new literal. This data structure were optimizing the propagation of the literals, but it was really demanding to maintain the correctness of it so that the trade-off was a little better than going though every clause every time, but non satisfactory.

2.2.2 Second Solution: Watcher

After implementing the *Induction Graph* the performance were still not good enough so I decide to read about lazy data-structure in the way to improve this aspect of my SAT. I've discovered the watch-list in a paper [2] that are a really low-cost way to propagate literals and find contradiction. Basically while encoding a clause (considering $|c| > 1$) randomly two literals of the clause are chosen and this clause is put on the watch list of this two literals. For example $c_1 = x_1 \vee x_2 \vee \neg x_3$, assume that x_1 and $\neg x_3$ were chosen, then c_1 will be added to the *watcher* of $\neg x_1$ and x_3. Then when an assignment $x_1 = 0$ or $x_3 = 1$, let's consider for example $x_1 = 0$, will be done for the first time we will analyze this clause and:

- if x_2 was not assigned then we just move c_1 from the watch list of the now watched literals to $\neg x_2$ watcher.
- if x_2 was assigned to 1, then c_1 is satisfied, nothing is done.
- if x_2 was assigned to 0, then there's just one more literals $\neg x_3$ and a new assignment $x_3 = 1$ will be generate.

In this way the propagation overhead is dramatically reduced and the performance improved. The watch-list has been implemented and used in the SAT.

2.3 CDCL Implementation

I will report here only the current CDCL implementation and not everyone that I tried. CDCL has been implemented exploiting the unit propagation model and following insight found in mini-sat implementation [3]. Every time that a variable has been propagated and a clause watched () then the `list` used to contain the literal of the clause were re-organized, moving the clause watched, in case there were no contradiction, to the rear of the list. In this way, after inducting an assignment or finding contradiction, the first element of `list` will be the last observed literal so, in case considering a clause c' that caused a propagation, that is the literal inducted.

2.3.1 Clause Learning

For clause learning I've implemented First UIP in the following way. When a contradiction has been found let's consider the clause c that caused the contradiction. Let's consider one by one the literals contained in c. For each literal (that was not already seen) $l \subset c$ let's retrieve the node n in the graph that assigned l. If the node assign in a level lower (levels starts from TOP = 0 and increasing) then then l is add to the learnt clause. If the node assign in the same level of the contradiction then resolve on $v(l)$ with clause c' that caused assignment in node n. So consider now c', consider every literal l' excluding the first in the `list` (by the observation that we've done before, we're resolving on this exact literal, so we can exclude it!), then for each literal (that was not already seen) do recursively what we've just explained. Here's python pseudo-code: Note

Algorithm 1 Clause Learning 1

```
function AnalyzeConflict(Clause, CurrentLevel)
    for all Literal $\in$ Clause do
        if Literal $\notin$ SeenLiterals then
            SeenLiterals $\leftarrow$ SeenLiterals $\cup$ {Literal}
            Node $\leftarrow$ GetNodeOfAssignment($\neg Literal$)
            if NodeLevel $\geq$ CurrentLevel $\wedge$ Node $\neq$ EmptyNode  then
                ResolutionOnAnalysis(...)
            else
                if Node.Level $>$ 0 then
                    LearntClause $\leftarrow$ LearntClause $\cup$ {Literal}
                end if
            end if
            return LearntClause

```

that literal assigned in TOP-LEVEL can be excluded as it's possible to resolve with a unit clause.

Algorithm 2 Clause Learning 2

```
function ResolutionOnAnalysis(Node, Seen, CurrentLevel, Levels, LearntClause)
    Clause = Node.Clause
    for all Literal $\in$ Clause do
        if Literal $\notin$ SeenLiterals then
            SeenLiterals $\leftarrow$ SeenLiterals $\cup$ {Literal}
            NextNode $\leftarrow$ NodeCausingAssignment($\neg Literal$)
            if NodeLevel $\geq$ CurrentLevel $\wedge$ Node $\neq$ EmptyNode then
                ResolutionOnAnalysis(...)
            else
                if Node.Level $> 0$ then
                    LearntClause $\leftarrow$ LearntClause $\cup$ {Literal}
                end if
            end if
        else
            if $\neg$ NextNode.Literal $\notin$ LearntClause $\wedge$ Node.Level $> 0$ then
                SeenLiterals $\leftarrow$ SeenLiterals $\cup$ {$\neg$NextNode.Literal}
                LearntClause $\leftarrow$ LearntClause $\cup$ {$\neg$NextNode.Literal}
            end if
        end if
    end for
return LearntClause
end function=0
```

2.3.2 Backtracking

The level of non-chronological backtracking is chosen analyzing the learnt clause c_l in the way that in the level in which we will backtrack, retracting all the subsequent assignment, a new assignment will be inducted by watching the literal of c_l. If the learnt clause is the empty clause, then the Formula is UNSAT.

2.4 Pick-Branching Heuristic

During the development of the SAT, while the structure was not well defined, i was doing Pick-Branching of variables randomly. When the was defined i tried to connect in the best way the Heuristic and the method that i was using to propagate variables in the way that the computing for the pick-branch less demanding as possible and using the information about clauses and variables already used for the basic functioning of the sat. That's approach was followed because i think that a good heuristic should not add too much overhead of calculation and use in the most smart way the data and information that we still have, without adding complexity unless the trade of is really good. So for the Heuristic I've decided to use the information given by the unit-propagation core. I got the general idea referring to [1] and [2], in which a dynamic evaluation of each variable during the running of the solving is proposed.

2.4.1 Heuristic with Induction Graph

The first Heuristic i developed was based on the core described in 2.2.1. The idea was to assign weight to each unassigned variable and its two possible values (x_i and$\neg x_i$) by the appearance in the root of the Induction Graph (so the clauses that has no parents). The weight assigned was depending on the number of sons that the node has, more node the graph has more weight was assigned to the formula. So the weight for each literal $w(l)$ was computed as:

$$\begin{aligned} &w(l) = \Sigma \widehat{w}(c_i) \forall c_i \supset l \\ &\widehat{w}(c_i) = m \end{aligned} \tag{1}$$

Where m is the number of sons that node of c_i has.

2.4.2 Heuristic with watcher

The second Heuristic I developed was based on the unit propagation core described in 2.2.2. The insight is really similar: using the information that the unit propagation collect during the solving. Basically every time a literal is propagated for every clause watched a value a_i is updated, this value represent how many variables constraint the clause c_i has not been assigned a value $\in \{0, 1\}$. The clause has an higher weight the lower is a_i. Then for every literal l, that can be assigned at some point, the weight is computed based on the appearance in a clause, and the weight of the clause:

$$w(l) = \Sigma \widehat{w}(c_i) \forall c_i \supset l$$
$$\widehat{w}(c_i) = \frac{1}{1 + a_i} \tag{2}$$

The literal with the higher weight will be the next assignment. A variant has been developed in which also the causing of conflict by a literal was considered so that: every time a conflict was caused, then if a literal was involved in a conflict a fixed weight was added in function of the level of the assignment. There was a little improvement with SAT formula in some cases, but the behavior with UNSAT was really bad so that I've abandoned the idea to use this and try to improve it.

3 Testing:

Test has been conducted on the behavior considering randomly generated 3-CNF Formula fixing n variables, k clauses and dividing in case the Clause was SAT or UNSAT. All the results were verified with Mini-Sat [3]. Two indicators are considered in those tests, the number of calls of subroutine `in_depth_assignments()` n_c and the number of assignments a (both picked up and inducted by propagation)

- $n = 100$, $k = 200$, 200 trials, SAT Formulas
  Random pick-branching: $n_c \approx 53.74$, $a \approx 117.18$
  Heuristic pick-branching: $n_c \approx 75.18$, $a \approx 100.18$
  Note that random do less calls of the subroutine, but still the Heuristic do less assignments, this mean that the tree depth before finding contradiction and backtracking is dramatically lower with the heuristic

- $n = 80$, $k = 640$, 200 trials, UNSAT Formulas
  Random pick-branching: $n_c \approx 270.14$, $a \approx 4674$
  Heuristic pick-branching: $n_c \approx 50.36$, $a \approx 1095.3$

- $n = 150$, $k = 525$, 200 trials, SAT Formulas
  Random pick-branching: no results - the experiment was unfeasible with this heuristic as it was going out of time ($t > 30s$) for trial
  Heuristic pick-branching: $n_c \approx 66.2$, $a \approx 346.9$ ($t_{cpu} \approx 1.5s$)

- $n = 200$, $k = 400$, 200 trials, SAT Formulas
  Random pick-branching: no results - the experiment was unfeasible with this heuristic as it was going out of time ($t > 30s$) for trial
  Heuristic pick-branching: $n_c \approx 153.86$, $a \approx 204.12$ ($t_{cpu} \approx 1.87s$)

- $n = 500$, $k = 1000$, 200 trials, SAT Formulas
  Random pick-branching: no results - the experiment was unfeasible with this heuristic as it was going out of time ($t > 30s$) for trial
  Heuristic pick-branching: $n_c \approx 377.1$, $a \approx 551.7$ ($t_{cpu} \approx 19.87s$)

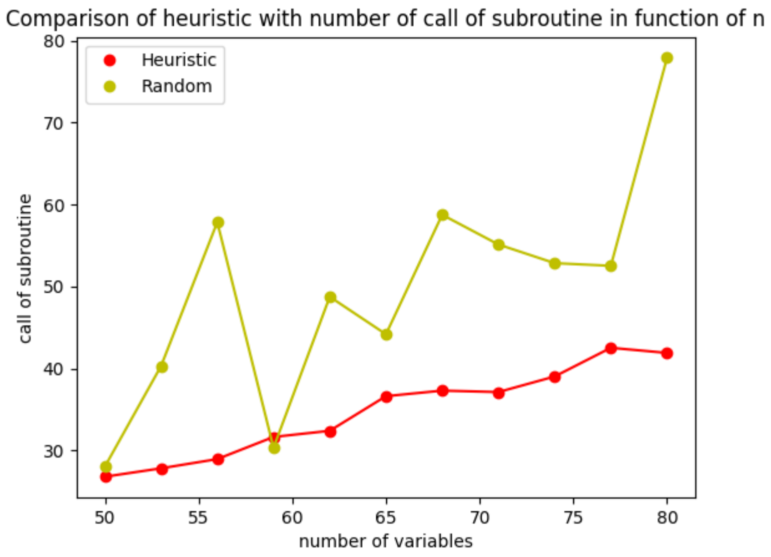

Figure 1: $n \in [50, 80]$, $k = r \cdot n$, $r = 3$, Number of calls in function of n with randomly generated SAT formulas

I've also done a plot for SAT and UNSAT formulas considering $k = r \cdot n$, with $r = 3$ and $n \in [50, 80]$. I've used *small* n as comparing the random branching is not feasible with $n > 100$, especially with UNSAT formulas as we can see in Figure 3.

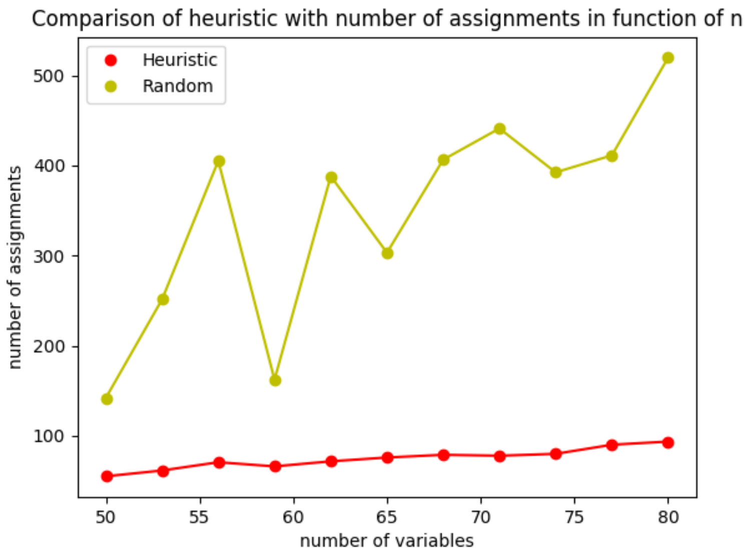

Figure 2: $n \in [50, 80]$, $k = r \cdot n$, $r = 3$, Number of assignments in function of n with randomly generated SAT formulas

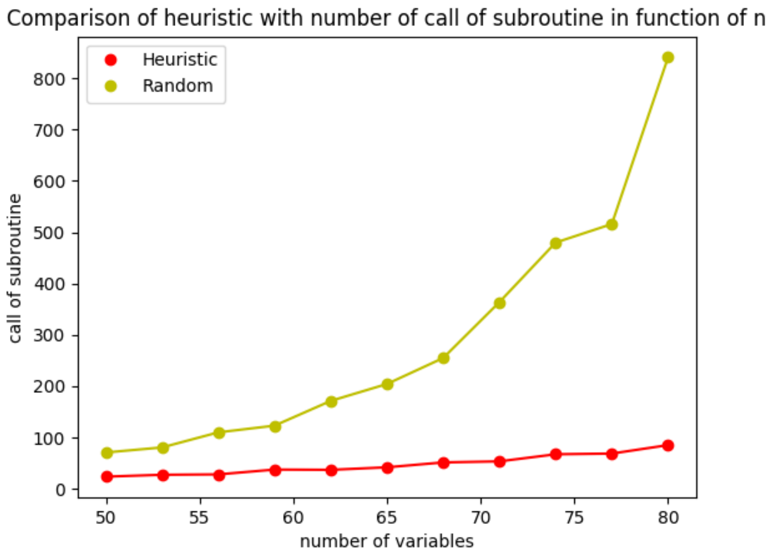

Figure 3: $n \in [50, 80]$, $k = r \cdot n$, $r = 3$, Number of calls in function of n with randomly generated UNSAT formulas

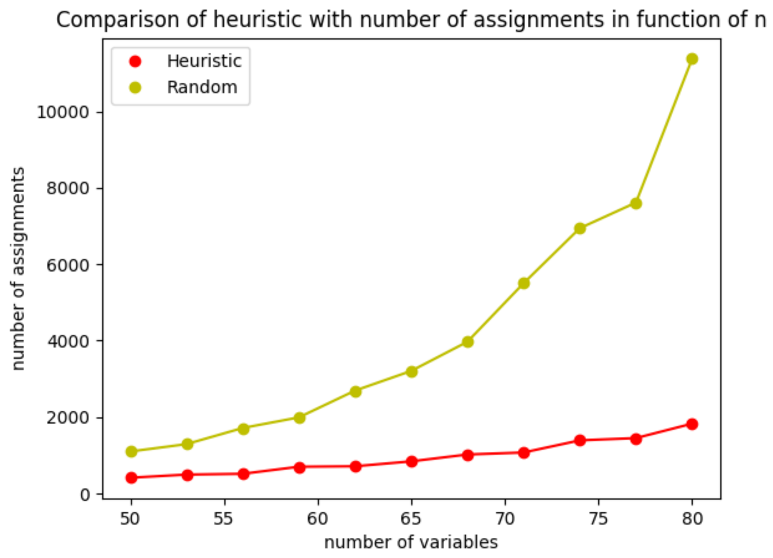

Figure 4: $n \in [50, 80]$, $k = r \cdot n$, $r = 3$, Number of assignments in function of n with randomly generated UNSAT formulas

4 Einstein Puzzle

4.1 Class of Attributes and Positions

First of all I defined attributes and class of attributes.
There are 5 class of attributes (Color, Nationality, Pet, Drink, Cigarette) and for each class there are 5 attributes.
The first step for the encoding is to associate variable to attribute.
For each attribute there are 5 variables, each of one represent one of the position that the attribute can assume.
So for each attribute:

$$\begin{gathered} \forall \mathbf{a} = \{a_1, a_2, a_3, a_4, a_5\} \\ a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \end{gathered} \tag{3}$$

they are in XOR because the attribute can assume only one position. Furthermore for each class of attribute $C_i = \{a, b, c, d, e\}$ there's the constraint on which there must be one and only one attribute for each position so:

$$\begin{gathered} \forall C_i \forall j \in \{1, 2, 3, 4, 5\} \\ a_j \oplus b_j \oplus c_j \oplus d_j \oplus e_j \end{gathered} \tag{4}$$

So for each class of attributes I translated 3 and 4 to CNF and generate clauses.

4.2 Hints

- Type 1: *The German smokes Prince*
  This kind of hint associate two attribute a', a'' belonging to different classes to the same position. We can translate this saying that all the variables of attribute 1 are equal to the variables of attribute 2:
  $$\mathbf{a}' = \mathbf{a}'' \tag{5}$$

- Type 2: *The Green house is on the left of the White house*
  We can translate a' is on the left of a'':
  $$(a'_1 \wedge a''_2) \vee (a'_2 \wedge a''_3) \vee (a'_3 \wedge a''_4) \vee (a'_4 \wedge a''_5) \tag{6}$$

- Type 3: *The Norvegian lives next to the blue house*
  Similiar to Type 2, a' is near a'':
  $$\begin{aligned}&(a'_1 \wedge a''_2) \vee (a'_2 \wedge a''_1) \vee (a'_2 \wedge a''_3) \vee (a'_3 \wedge a''_2) \vee \\ &(a'_3 \wedge a''_4) \vee (a'_4 \wedge a''_3) \vee (a'_4 \wedge a''_5) \vee (a'_5 \wedge a''_4)\end{aligned} \tag{7}$$

- Type 4: *The Norvegian lives in the First House*
  This is a simple 1-literal clause, in this case assume attribute Norvegian as a, we simply add this clause:
  $$a_1 \tag{8}$$

4.3 Solution

I've attached a python script with the SAT that generates the CNF Formula following the above constraint, call the SAT and interpret the model.
That's the output:

```
BRIT in 3
SWEDE in 5
DANE in 2
NORVEGIAN in 1
GERMAN in 4
RED in 3
WHITE in 5
YELLOW in 1
GREEN in 4
BLUE in 2
PALLMALL in 3
DUNHILL in 1
BLENDS in 2
PRINCE in 4
BLUEMASTER in 5
TEA in 2
```

```
MILK in 3
COFFEE in 4
BEER in 5
WATER in 1
CATS in 1
DOGS in 5
BIRDS in 3
HORSE in 2
FISH in 4
```

So the answer to the question *"Who own the fish?"* is **the German guy**.

5 Conclusions

The here proposed SAT is complete and correct, it works with both SAT and UNSAT formulas and a good-working heuristic has been recognized. The performance are acceptable, considering the language used, expecially with UNSAT formulas, in which the heuristic behaves way better than the random pick-branching as we can see in Figure 3, 3, with this heuristic we are able to solve in reasonable time formula with $n > 150$ as verified by the tests. It's curious to notice that the random pick-up behaves in a really similar way to the Heuristic with $n < 50$, they both share the initial phase of exponential complexity, but random starts to diverge earlier. Another really important goal reached with the heuristic is to reduce the depth of the tree of assignment, as we can see by the data, even when the heuristic use more calls of the subroutine, there are less assignment done. By the way in SAT problem the basic function and implementation of lazy data-structure are important as the Heuristic (as in fact they influence also its behavior) and a lot of effort has been done in this direction, in fact the perfomance, before a good implementation of structure and functions, were really poor and the SAT was not really usable for large formulas.

References

[1] Ed Zulkoski Atulan Zaman Jia Hui Liang, Vijay Ganesh and Krzysztof Czarnecki. Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. *University of Waterloo, Waterloo, Canada*, 2015.

[2] Y. Zhao L. Zhang M. W. Moskewicz, C. F. Madigan and S. Malik. Engineering an efficient sat solver. *Design Automation Conference*, pages 530–535, 2001.

[3] Niklas Een Niklas Sorensson. An extensible sat-solver. *Chalmers University of Technology, Sweden*, 2003.