Project report

When intuition misfires: Hyper Sudokus are harder than standard Sudokus

Knowledge Representation 2017

Mario Giulianelli[*], **Jack Harding**[†]
11567252, 11623608

Abstract: In this paper, we test the hypothesis that Hyper Sudokus are *easier* to solve than standard Sudokus in terms of added conflict literals, added conflict clauses, restarts, and variable flips. We encode Hyper Sudokus as SAT problems using three existing encodings and extend them to Hyper Sudokus. We then test each of the six encodings on three different SAT solvers. The results of our experiment falsify our hypothesis and we propose a plausible explanation why the performance of SAT solvers decreases for Hyper Sudokus.

Keywords: Sudokus • Hyper Sudokus • SAT solvers

1. Hyper Sudokus

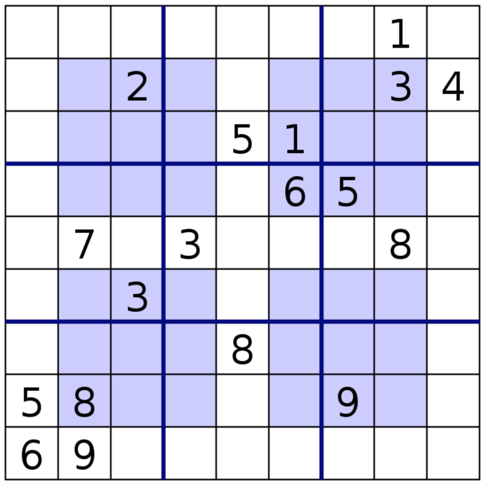
Hyper Sudokus are a variant of Sudokus characterised by the presence of additional constraints. Hyper Sudokus have four additional blocks and it is stipulated that these *Hyperblocks* contain each number from 1 to 9 exactly once. Figure 1 illustrates the Hyper Sudoku with its *Hyperblocks*. In this paper, we focus on 9×9 Hyper Sudokus.

For human intuition, Hyper Sudokus are easier than Sudokus. One means of arriving to this conclusion is to consider an individual cell of a Sudoku. Filling this cell with a number implies that 8 other numbers cannot appear in the same cell, and that the value in the cell will not appear in 8 other spaces respectively in the row, in the column and in the block. This allows us to eliminate 28 possibilities in the best case. On the other hand, if this cell is in a *Hyperblock*, as many as 4 additional possibilities can be eliminated.

We translate this intuition into our hypothesis: **Hyper Sudokus are easier than standard Sudokus**. The *hardness* of a standard or Hyper Sudoku problem is defined differently for each solver: in zChaff, *hardness* is expressed as the number of **added conflict literals** and **conflict clauses**; in WalkSAT, we monitor the number of **restarts** and **variable flips**.

[*] *E-mail: m.giulianelli.m@gmail.com*
[†] *E-mail: jackhardingwork@gmail.com*

Figure 1: A Hyper Sudoku puzzle. The blue regions are the *Hyperblocks*, for which it holds that every number from 1 to 9 should only appear exactly once.



Although we focus on Hyper Sudokus, it should be clear that our results will have ramifications for judgements about the difficulty of any Sudoku with additional constraints. Furthermore, it is interesting to note that comparing the performance of SAT solvers on Hyper and standard Sudokus actually constitutes a test of the solvers themselves. As we will see below, the only difference between encodings of Sudokus and encodings of Hyper Sudokus is that the latter contain additional clauses (corresponding to the additional constraints of the Hyper Sudoku). Hence, for each solver—and each encoding—we test whether the presence of additional clauses aids the satisfaction process. The experimental methodology we employed in the test of our hypothesis is described in the next section.

2. Experimental Setup

The hypothesis that a Hyper Sudoku is easier than a Sudoku is tested monitoring the number of added conflict literals, conflict clauses, restarts and variable flips. We chose these statistics because they best represent the *hardness* of a Sudoku problem with respect to a SAT solver. We additionally examined CPU elapsed time. Although it is not a good measure of *hardness*, elsasped time can still provide useful empirical insights.

The next subsections introduce the data sets, SAT solvers, and encodings that were employed in our experiment. The implementation of this test can be found at `github.com/Procope/hyperactive`.

2.1. Data Sets

In this experiment, we used two data sets: a freely available data set[1] consisting of 1 million standard Sudoku puzzles with solutions and a data set of 100 Hyper Sudokus, which was manually scraped from a web page[2] offering multiple variants of Sudoku games.

2.2. SAT Solvers

We use three freely available SAT solvers to find solutions to Sudokus and Hyper Sudokus: PicoSAT, zChaff, and WalkSAT.

PicoSAT is based on conflict-driven backtracking.. We use *pycosat*[3], the Python bindings for PicoSAT, only to test that the generated encodings are correct as *pycosat* does not allow to straightforwardly obtain useful statistics. zChaff is widely used for industrial purposes. It features the Two Literal Watching scheme, conflict-driven clause learning and non-chronological back-tracking. WalkSAT is a family of non-deterministic algorithms that start from a randomly chosen variable assignment and then select the clauses which are violated by the current assignment. Variable occurring in these falsified clauses are flipped to increase the total number of satisfied clauses. We deploy three variants of WalkSAT denoted as *Best*, *Novelty*, and *R-Nnovelty*.

These SAT solvers and their variants are presented more thoroughly in Appendix A.

2.3. Encodings

To prove our hypothesis that Hyper Sudokus are easier than Sudokus we use three different Sudoku encodings: the *minimal encoding*, *efficient encoding*, and *extended encoding*. It might still be unclear why different sets of constraints, i.e. sets of clauses, should influence a SAT solver, since the SAT problem they encode is ultimately the same. However, different sets of constraints impact the way SAT solvers empirically decide on the satisfiability of the problem.

SAT problems are represented using propositional variables $x_1, \dots x_n \in \{0, 1\} = \{false, true\}$. Each variable x_i and its complement $\neg x_i$ are called literals and a disjunction of literals forms a clause. A CNF (Conjunctive Normal Form) formula is a conjunction of clauses. The SAT problem consists of deciding whether there exists a truth assignment to the variables such that the formula becomes satisfied [1]. In this project, we use 9×9 Sudokus and Hyper Sudokus. A puzzle has 81 cells and each cell can take the values 1-9. Hence, the required number of propositional variables for a CNF encoding is $9^3 = 729$. We use the notation s_{xyz} to refer to variables. This indicates that if the value in row x and column y is z, then $s_{xyz} = 1$. Otherwise, $s_{xyz} = 0$. In matrix notation, $s_{xyz} = 1$ translates into $S_{xy} = z$ or $S[x][y] = z$.

The following sets of clauses were used in our Sudoku encoding:

(1a) At least one number appears in each cell.
$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz}$

[1] *kaggle.com/bryanpark/sudoku*
[2] *sudokucentral.co.uk/hypersudoku.php*
[3] *github.com/ContinuumIO/pycosat*

(1b) At most one number appears in each cell.
$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{z=1}^{8} \bigwedge_{i=z+1}^{9} (\neg s_{xyz} \vee \neg s_{xyi})$

(2a) In each row, each number appears at least once.
$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{x=1}^{9} s_{xyz}$

(2b) In each row, each number appears at most once.
$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{i=x+1}^{9} (\neg s_{xyz} \vee \neg s_{iyz})$

(3a) In each column, each number appears at least once.
$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{y=1}^{9} s_{xyz}$

(3b) In each column, each number appears at most once.
$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{y=1}^{8} \bigwedge_{i=y+1}^{9} (\neg s_{xyz} \vee \neg s_{xiz})$

(4a) In each 3×3 block, each number appears at least once.
$\bigvee_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} s_{(3i+x)(3j+y)z}$

(4b) In each 3×3 block, each number appears at most once.
$\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} \bigwedge_{k=y+1}^{3} (\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+x)(3j+k)z})$
$\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} \bigwedge_{k=x+1}^{3} \bigwedge_{l=1}^{3} (\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+L)(3j+l)z})$

Additionally, the Hyper Sudoku encoding requires:

(5a) In each 3×3 *Hyperblock*, each number appears at least once.
$\bigwedge_{i=0}^{1} \bigwedge_{j=0}^{1} \bigwedge_{x=0}^{2} \bigwedge_{y=0}^{2} \bigvee_{z=1}^{9} s_{(2+4i+x)(2+4j+y)z}$

(4b) In each 3×3 *Hyperblock*, each number appears at most once.
To increase readability, we use a slightly declarative notation[4]:
$\bigwedge_{z=1}^{9} \bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{k=1}^{9} \bigwedge_{i=1}^{9} \bigwedge_{j=1}^{9} (\neg s_{xyz} \vee \neg s_{yxz})$ `if hyper_incompatible(`s_{xyz}, s_{xyz}`) = true`

The *minimal encoding* consists of (1a), (2b), (3b), and (4b). The set (1a) of at-least-one clauses produces 81 nine-ary clauses, 8,748 binary clauses result from the three sets (2b), (3b), and (4b) of at-most-one clauses, for a total of 8,829 clauses in the resulting CNF formula.

The *extended encoding* includes all the sets of clauses introduced for the Sudoku encoding, hence it explicitly asserts that each cell has exactly one number, and likewise for each row, each column, and each 3x3 block [1]. The resulting CNF formula will have 11,988 clauses,: 324 nine-ary clauses ((1a), (2a), (3a), (4a)) and 11,664 binary clauses ((1b), (2b), (3b), (4b)).

The *efficient encoding* is a trade-off between the previous two and consists of (1a), (1b), (2b), (3b), and (4b). It essentially expands the *minimal encoding* asserting that at most one number appears in each cell.

[4] *see Appendix B for a definition of* `hyper_incompatible`

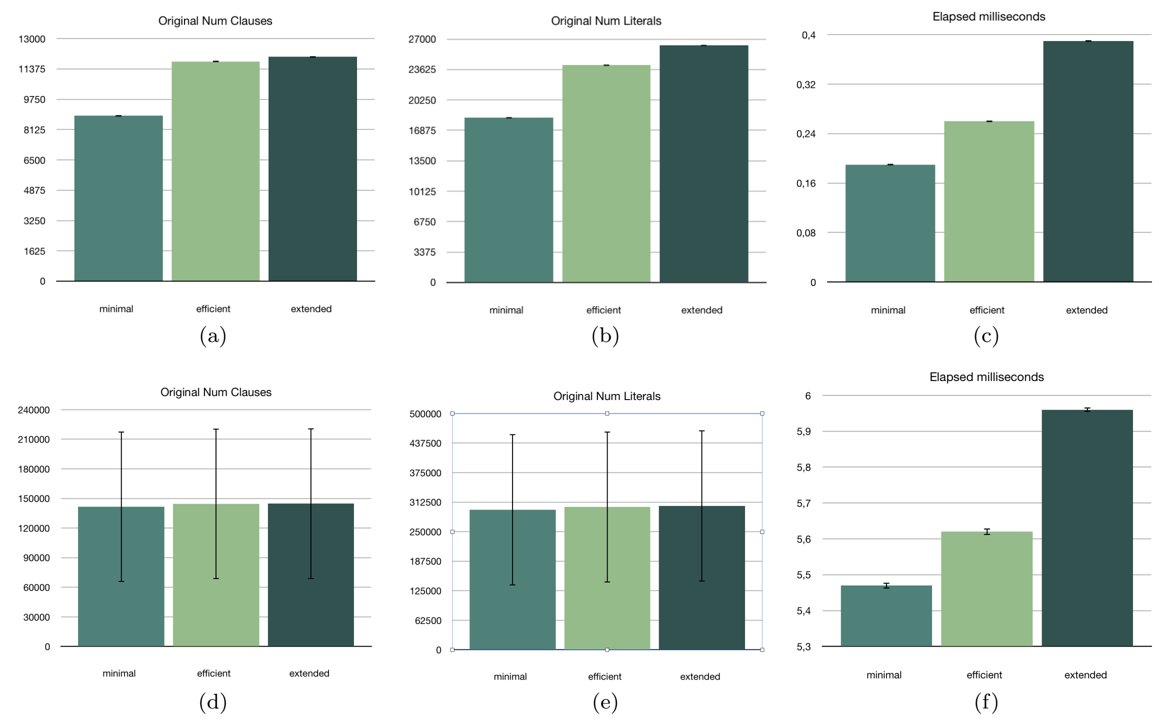


Figure 2: The number of clauses, literals, and elapsed time that zChaff requires to solve a Sudoku (first row) and a Hyper Sudoku (second row). Encoded clauses and literals include the unit clauses immediately deduced from an incomplete puzzle.

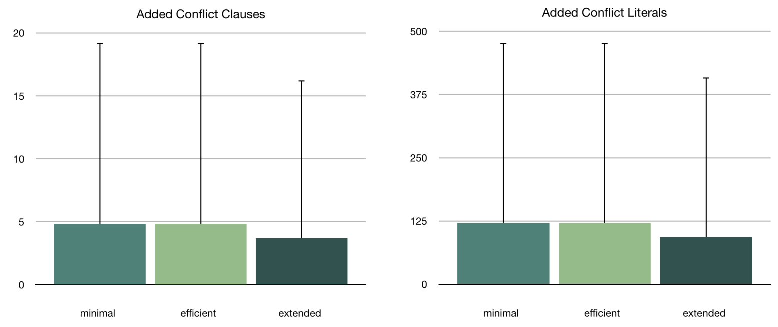


Figure 3: The number of conflict clauses and conflict literals that zChaff adds while it searches for a solution.

3. Results and Interpretation

For the standard Sudoku, the choice of encodings causes a significant difference in the number of clauses and of literals that are required to describe the SAT problem. A direct consequence of the increase in clauses and literals is a growth of the running time. This is evident for both solvers, as in Figure 2(c) and Figure 4(c) show.

In the Hyper Sudoku encoding, the number of clauses and literals fluctuates much less as a function of the encoding strategy because of the influence of the additional *hyperconstraints*. As a consequence, the running time is also more balanced. Although Figure 2(c) and Figure 2(f) appear to show the same trend, the absolute values of elapsed seconds show that the difference between encodings is much stronger for the standard Sudoku case.

Furthermore, using the extended encoding for Hyper Sudokus yields less conflict clauses and conflict literals (Figure 3) This result appears surprising at first, as one would expect additional constraints to increase the probability of conflicts. It seems, however, that these constraints limit the search space in such a way that conflicts are partially avoided; in other words, the supplementary constraints exclude erroneous tentative assignments that would lead to conflicts. If this outcome partially supports our hypothesis that harder constraints should improve performance, the fact that the tested SAT solvers were able to find solutions to standard puzzles without adding new conflict clauses shows that Hyper Sudokus are harder in terms of the number of conflicts that arise when solving them.

The number of variable flips needed to find a valid assignment for a Hyper Sudoku (Figure 4(b), 4(e)) is significantly higher than the number of flips a standard Sudoku requires. This is the second clear indication against our original hypothesis: Hyper Sudokus seem to also be *harder* in terms of the amount of variable flips they require. A possible explanation for this outcome is that the probability that a variable assignment will be incorrect increases with the number of constraints. In other words, the larger the set of clauses, the harder it is for a variable to satisfy all of them.

The last monitored value is the number of restarts. Figure 4(a) and 4(d) show that there is no significant difference between standard and Hyper Sudokus in terms of restarts. The only distinction is in the variance of our measurements. Indeed, all statistics regarding Hyper Sudokus show high variance. After some data exploration, we found that a small subset of Hyper Sudoku puzzles are remarkably *harder* for solvers, as all the monitored values grow very rapidly when such puzzles are solved. It was nevertheless unclear why SAT solvers found these outliers to be more challenging.

4. Conclusion and Future Work

With this project, we wanted to test the intuition that Hyper Sudokus are easier than standard Sudokus. We defined the *hardness* of a Hyper Sudoku problem in terms of the number of added conflict literals and conflict clauses, for zChaff; and in terms of the number of restarts and variable flips, for WalkSAT. The results of our experiment falsified our hypothesis: for a SAT solver, it appears to be easier—in terms of the metrics described above—to solve a standard Sudoku. Although, intuitively, Hyper Sudokus allow for a more constrained logical reasoning, the very form these constraints take, i.e. the form of additional clauses and literals, negatively affects the performance of the tested SAT solvers. This outcoume seems to imply that, at least for an accessible SAT problem like a Sudoku puzzle, the performance of solvers is inversely proportional to the number of clauses and literals used to encode that problem. It is important to note that the Sudoku dataset we employed contains mostly so-called *easy* to *medium* puzzles. We deduce this from the fact that all solvers could find a satisfying assignment regardless of the type of encoding. Indeed, it was shown [2] that *hard* Sudoku puzzles require at least the efficient encoding, and some are only solvable using the extended encoding.

Furthermore, our experiment complements the research done in encoding optimisation. We conclude that, unless a SAT problem *requires* a more complex or explicit encoding, it is best to opt for the encoding that minimises the computational burden caused by large numbers of clauses and literals. This interpretation is limited insofar as the types of solvers that were chosen.

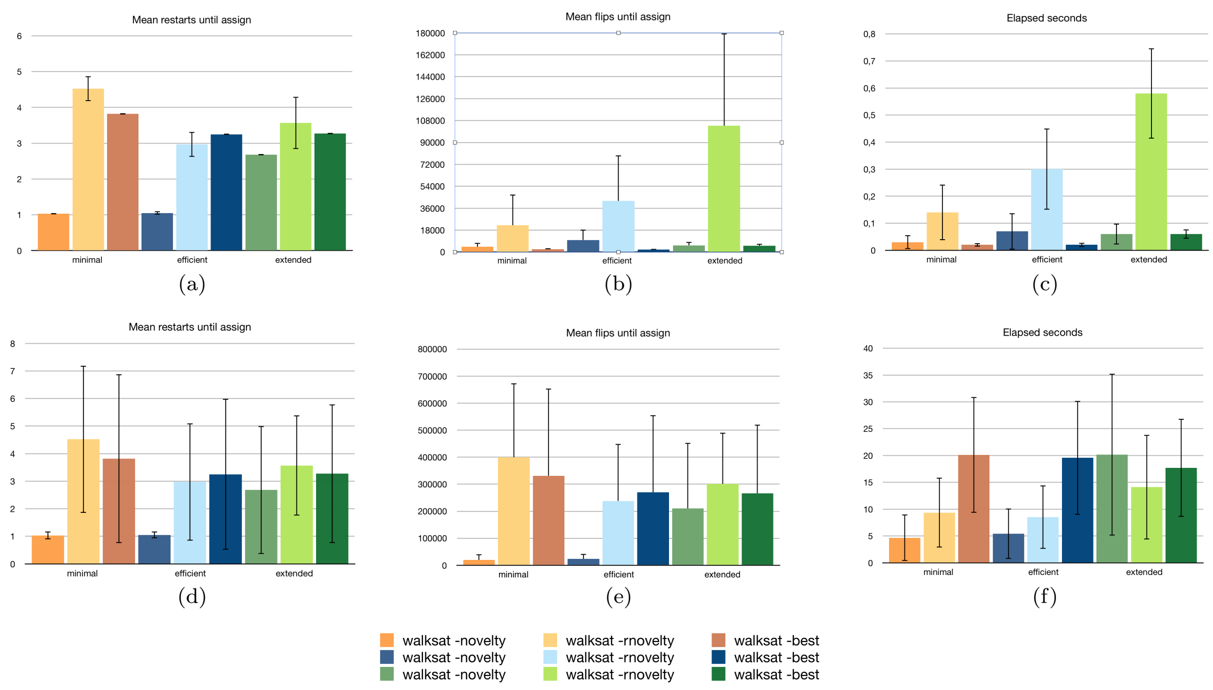


Figure 4: The average number of restarts and variable flips that WalkSAT goes through to find a successful assignment, as well as the elapsed seconds required to solve a Sudoku (first row) and a Hyper Sudoku (second row).

In future, we intend to further test our hypothesis using non-conflict-driven solvers. We also plan to develop new encodings for the Hyper Sudoku that limit the number of required clauses, as the *minimal encoding* does for standard Sudokus. Finally, we intend to repeat the tests presented in this report using the optimised encoding [2], which introduces rules to prune the set of clauses needed to encode a Sudoku before feeding the puzzle to a SAT solver.

References

[1] Lynce, Ins, and Jol Ouaknine. ”Sudoku as a SAT Problem.” ISAIM. 2006.

[2] Kwon, Gihwon, and Himanshu Jain. ”Optimized CNF encoding for sudoku puzzles.” Proc. 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR2006). 2006.

[3] Biere, Armin. ”PicoSAT essentials.” Journal on Satisfiability, Boolean Modeling and Computation 4 (2008)

[4] Fu, Zhaohui, Yogesh Marhajan, and Sharad Malik. ”Zchaff sat solver.” 2012-01-31]. `http://www.princeton.edu/chaff` (2004).

[5] McAllester, David, Bart Selman, and Henry Kautz. ”Evidence for invariants in local search.” AAAI/IAAI. 1997.

[6] Moskewicz, Matthew W., et al. ”Chaff: Engineering an efficient SAT solver.” Proceedings of the 38th annual Design Automation Conference. ACM, 2001.

Appendix A

In this appendix, we present the three solvers as well as their variants.

PicoSAT [3] is based on conflict-driven backtracking [] and it was created to improve low-level performance of existing SAT solvers such as Chaff [] and MiniSat []. We use *pycosat* [], the Python bindings for PicoSAT, to test that the generated encodings are correct, i.e. that the encodings lead to a satisfiable formula and that the resulting instantiations indicated are indeed a solution to the input Sudoku or Hyper Sudoku puzzle. Indeed, *pycosat* does not allow to straightforwardly obtain useful statistics.

zChaff is widely used for industrial purposes. It implements the Chaff algorithm [6] and, performance-wise, versions of zChaff have emerged as the Best Complete Solver in the industrial and handmade instances categories in the SAT 2002 Competition and as the Best Complete Solver in the industrial category in the 2004 SAT Competition [4]. In short, the main features of zChaff are: (i) the Variable State Independent Decaying Sum (VSIDS) decision strategy, which keeps a score for each literal of a variable; (ii) the very efficient Boolean Constraint Propagation and (iii) the Two Literal Watching scheme, both introduced by Chaff [6]; (iv) Conflict Driven Clause Learning and Non-chronological Back-tracking, which become essential for solving structured problems; (v) aggressive clause deletion, which removes some learned clauses based on usage statistics and clause lengths; (vi) frequent restarts, which were shown to determine the solving time of a SAT solver.

WalkSAT is a family of non-deterministic algorithms that start from a randomly chosen variable assignment and then select the clauses which are violated by the current assignment. Variable occurring in these falsified clauses are flipped to increase the total number of satisfied clauses. We deploy three variants of WalkSAT: denoted as *Best*, *Novelty*, and *R-Nnovelty*. In *Best*, if multiple variables can be flipped without violating other clauses, one of these is randomly chosen. Otherwise, if flipping a variable violates other clauses, then a variable is picked which minimises the number of clauses that are currently satisfied but would become violated by the variable flip. More recently introduced WalkSAT algorithms [5] are defined by the following heuristics for selecting the variable to be flipped within the selected clause: *Novelty* sorts the variables in the selected clause based on the difference in the total number of satisfied clauses a flip would cause, and selects the best variable according to this ordering. However, if the selected variable happens to be the most recently flipped one, it is only flipped with a fixed probability. In the remaining cases, the second-best variable is flipped. *R-Novelty* is similar to *Novelty* in that it also assigns a score to each variable depending on the difference in the total number of satisfied clauses a flip of that vatiable would cause. In contrast with *Novelty*, if the best variable is the most recently flipped one, the decision between the best and second-best variable depends on their score difference. Furthermore, this heuristic is ignored every 100 steps: instead of using a score, the variable to be flipped is randomly picked from the selected clause.

Appendix B

```
def hyper_incompatible(x, y):
   return x != y
      and top_left_hypersquare(x) != None
      and top_left_hypersquare(y) != None
      and x[2] == y[2]
      and top_left_hypersquare(x) == top_left_hypersquare(y)
```

where `x, y` are triples $(r, c, v) \in [1, 9] \times [1, 9] \times [1, 9]$ representing propositional variables, and `top_left_hypersquare` simply returns, for a cell c, the top left cell of the hyperblock c lies in. For a more detailed account, see our repository: `github.com/Procope/hyperactive`.