SAT Solver Trials

Infinity Project – CRM Barcelona
December 9, 2009

Sam Buss
sbuss@math.ucsd.edu
Dept of Mathematics, UCSD

Some literature:

Clause learning algorithms for SAT:

- J. Marques Silva, K. Sakallah, “GRASP - A New Search Algorithm for Satisfiability”, Intl. Conf. on Computer Aided Design, ICCAD'96.
- M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, “Chaff: Engineering an Efficient SAT Solver”, Proc. Design Automation Conference (DAC) 2001.

Accessible introduction for logicians:

- P. Beame, H. Kautz, A. Sabharwal, “Towards Understanding and Harnessing the Power of Clause Learning, J. Artificial Intelligence Research, 2004, 22, 319-351. See also IJCAI’2003.

More papers at: http://www.math.ucsd.edu/~sbuss/CourseWeb/Math268_2007WS/.

S. Buss, “SatDiego”, C++ software package in development. Available by request.

S. Buss, J. Hoffmann, J. Johannsen, “Weak resolution trees with lemmas – Resolution refinements that characterize DLL-algorithms with clause learning.” Logical Methods in Computer Science, 2009.

SAT (Satisfiability Problems) –

- A clause is a finite set of boolean literals (variables or negated variables).
- A clause represents the disjunction (OR) of its member literals.
- A set of clauses represents a conjunctive normal form formula.
- A satisfying assignment must make at least one literal true in each clause.

SAT Problem:

- Given a set of clauses, does it have a satisfying assignment?
- If not, give refutation.
- If so, exhibit a satisfying assigment.

Determining Satisfiability is NP-complete.

Hence it is conjectured to have no feasible algorithm.

Nonetheless, many real world problems can be coded as instances of SAT (or as related Constraint Satisfaction Problems). Fortunately, these problems often have a lot of special structure, and can be feasible to solve in some cases.

There is a long history of SAT Solvers / SAT Competitions

- SAT competitions held annually since 2002.
- Best methods for “structured” or industrial problems are based on the clause learning methods of GRASP/Chaff.
- Best methods for *random* instances use belief propagation or survey propagation, and are incomplete.
- However, SAT is NP-complete. Many “concocted” problems are completely impossible for SAT solvers, e.g., pigeonhole principle tautologies, tautologies based on one-way functions such as integer factorization, etc.

Brute-force search DLL algorithm:

Input: Set of clauses.

- Set partial truth assignment initially equal to the empty assignment.
- Algorithm (recursive), maintaining a partial truth assignment.
  - If the assignment satisfies all clauses, halt. Have a satisfying assignment.
  - If the assignment falsifies a clause, return.
  - Choose a variable *x* and a truth value, set the variable to that value.
  - Call the algorithm recursively.
  - Undo the change to value of *x,* then set *x* to the opposite value.
  - Call algorithm recursively.
- When return at top level – unsatisfiable.

DLL search algorithm with unit-propagation:

Input: Set of clauses.

- Set a partial assignment initially equal to the empty assignment.
- Iterate, maintaining a partial truth assignment.
  - **Use unit-propagation, UP, (Boolean Constraint Propagation, BCP) to extend the partial assignment.**
  - If the assignment satisfies all clauses, halt. Have a satisfying assignment.
  - If the assignment falsifies a clause, return.
  - Choose a variable *x* and a value, set the variable to that value.
  - Call the algorithm recursively.
  - Undo the change to value of *x,* then set *x* to the opposite value.
  - Call algorithm recursively.
- When return at top level – unsatisfiable.

Unit propagation: Whenever a singleton clause occurs (i.e., all but one literal is set false), set the remaining literal to value *True.*

DLL search algorithm with unit-propagation and <u>clause learning</u>:

Input: Set of clauses.

- Set a partial assignment initially equal to the empty assignment.
- Loop, maintaining a partial truth assignment.
  - Use unit-propagation, UP, (Boolean Constraint Propogation, BCP) to extend the partial assignment**.**
  - If the assignment satisfies all clauses, halt. Have a satisfying assignment.
  - If the assignment falsifies a clause, **then learn one (or fewer or more) clauses that are added to the database (the input),** and return.
  - Choose a variable *x* and a value, set the variable to that value.
  - Call the algorithm recursively.
  - Undo the change to value of *x,* then set *x* to the opposite value.
  - Call algorithm recursively.
- When return at top level – unsatisfiable.

Learning: By examining the unit-propagation structure that leads to a falsified clause, learn a new clause (or multiple clauses). The hope is that the new clauses will “accelerate” future branches of the search tree.

DLL algorithm with unit-propogation, clause learning and <u>fast-backtracking</u>:

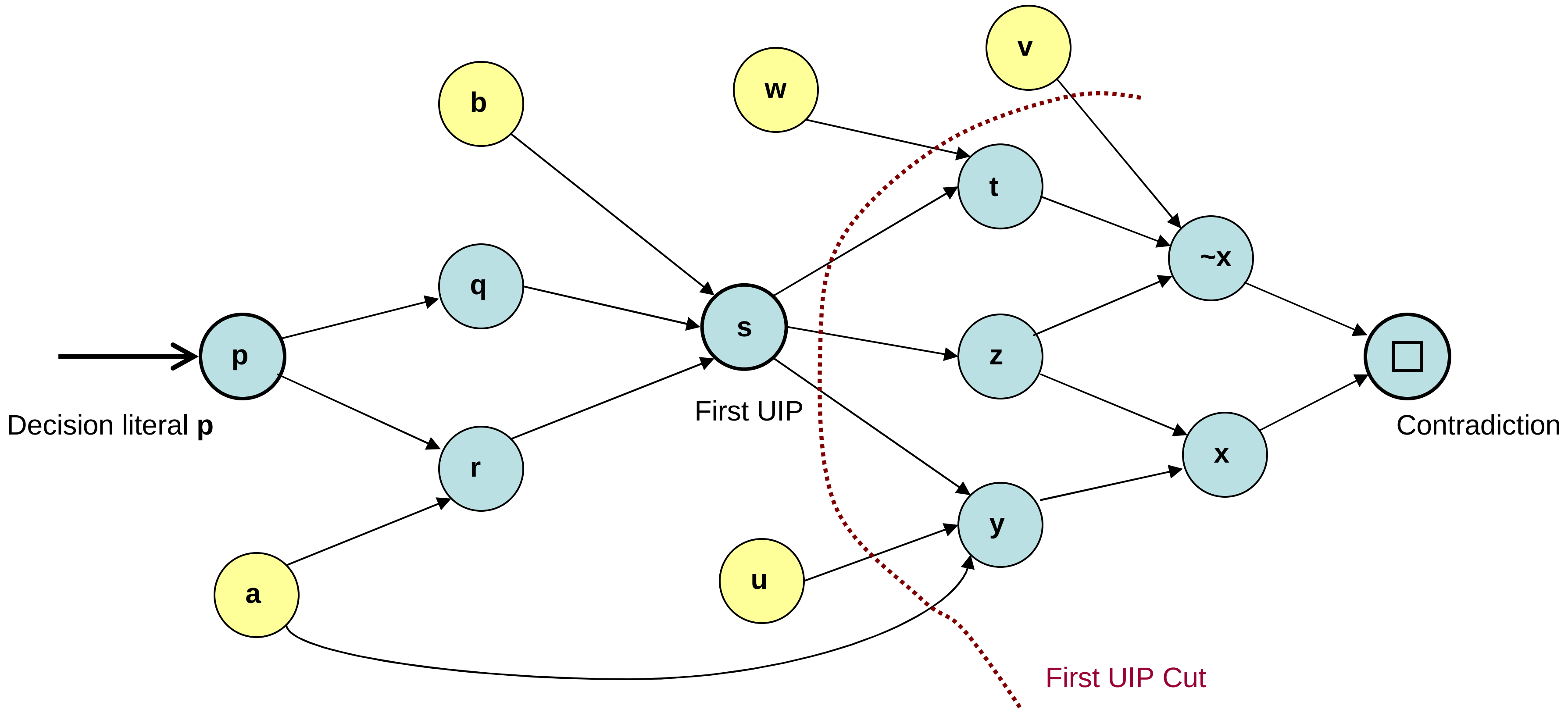
Input: Set of clauses.

- Set a partial assignment initially equal to the empty assignment.
- Loop, maintaining a partial assignment **and a " decision level" d (number of arbitrary decisions).**
  - Use unit-propogation, UP, (Boolean Constraint Propogation, BCP) to extend the partial assignment**. Label newly assigned variables as set at current decision level d.**
  - If the assignment satisfies all clauses, halt. Have a satisfying assignment.
  - If the assignment falsifies a clause, then learn one (or fewer or more) clauses that are added to the database (the input). **Analyze the decision level of the variables involved in the falsification, and return with the level information.**.
  - Choose a variable *x* and a value, set the variable to that value. **Label the variable's decision level as d+1.**
  - Call the algorithm recursively **with level equal to d+1.**
  - Undo the change to value of *x,* then set *x* to the opposite value. **Label the variable's level based on the information passed back from the recursive call (will be at most d).**
  - Call algorithm recursively **with level equal to d.**
  - **Based on level information returned, backtrack to appropriate level unsettting variables at higher levels.**
- When exit at top level – unsatisfiable.

Keeping track of variable levels allows the algorithm to "fast backtrack" up multiple levels of the tree. This makes the algorithm is potentially less severely impacted by bad choices of variables.

The use of unit-propagation makes tracking levels easy and natural.

Example of clause learning



Clauses: {~y,~z, x}, {~p,~a,r}, etc. (One per unit propagation.)
First UIP Learned Clause: {~a,~u,~s,~w,~v}.
Whole top level learned clause: {~p,~a,~u,~b,~w,~v}.
With First-UIP: Both **p** and **s** can be set false when backtracking.
New level of **~s** is set to max. level of **u,v,w.**

🔵 Blue for top level

🟡 Yellow for lower level literal

Additional refinements:

- **Clause unlearning:** Clauses that are old, not recently used, can be discarded. Likewise, clauses that are too large should not be learned, or should be quickly discarded. (SatDiego software does not do this [yet], but instead remembers all clauses forever.)

- **Restarts.** After a short period of not succeeding, restart the process from scratch, maintaining the list of learned clauses. (The SatDiego software does not do this.)

- **Pure literals:** These are literals that appear only positively or only negatively. These may be set without loss of generality. Doing this will not affect learned clauses since the pure literals will not appear in learned clauses (since only falsified clauses contribute to learned clauses), nor does this affect the falsification or satisfaction of the other clauses.

  Can affect the variable selection process.

- **Variable watching:** Many implementations use "2-literal watching" to speed up the update process for setting (and un-setting) values for variables. The SatDiego software does not do this, partly since it is not compatible with all variable selection methods. Instead we update all clauses containing a given variable, and update all variables occuring in these clauses to maintain counts of the occurences.

  As an alternate optimization, SatDiego uses a **literal resorting** method that dynamically sorts clause numbers in variables' arrays to put the active (unsatisfied) clauses at the beginning of the variables' lists, and the unactive clauses at the end of the lists. This approximately doubles the speed of updating the status of clauses and literals in the clauses (the improvement is especially good in the unsetting part).

Can we improve by using 2-CNF reasoning instead of just Unit Propagation?

Motivation: 2-CNF reasoning is complete, and easy. Could potentially improve the search process by drawing more conclusions earlier.

Observation: All 2-CNF conclusions can follow from Unit Propagation, provided that the correct variable is selected.

For example: If the variable **x** is forced true by 2-CNF reasoning, then one can reach the same conclusion by setting **x** to the value false and using Unit Propogation to reach a contradiction.

Thus, 2-CNF reasoning can be implemented via variable selection and unit propagation.

Experiments on Pigeonhole Tautologies.

PHP_{n}^{n+1} - Tautology expressing exists no injection from [n+1] to [n].

	Clause Learning		No Clause Learning	
Formula	Steps	Time (s)	Steps	Time
PHP_{3}^{4}	5	0.0	5	0.0
PHP_{6}^{7}	129	0.0	719	0.0
PHP_{8}^{9}	769	0.0	40319	0.3
PHP_{9}^{10}	1793	0.5	362879	2.5
PHP_{10}^{11}	4097	2.7	3628799	32.6
PHP_{11}^{12}	9217	14.9	-	-

Problems from SAT Race 2006 Test Suite #1

Tests were run on the 18 easiest qualifying problems chosen by the SAT-Race 2006 organizers.

The problems are "real-world" industrial problems.

Problem	# variables	# clauses
een-tipb-sr06-par1	163647	484831
een-tipb-sr06-tc6b	40196	115791
grieu-vmpc-s05-27r	729	96849
hoons-vbmc-s04-07	25900	77643
manol-pipe-c10ni_s	204664	609478
manol-pipe-c6nid_s	148051	438562
manol-pipe-f6b	37002	109570
manol-pipe-f6n	37452	110920
manol-pipe-g6bid	40371	118192
manol-pipe-g7n	23936	70492
schup-l2s-s04-abp4	14809	48483
stric-bmc-ibm-10	59056	323700
stric-bmc-ibm-12	39598	194660
vange-color-inc-54	117426	469472
velev-eng-uns-1.0-04	6944	66654
velev-eng-uns-1.0-04a	7000	67586
velev-live-sat-1.0-03	224920	3596474
velev-sss-1.0-cl	1453	12531

Variable selection methods tested

1. **Greedy variable:** Choose variables in the order presented in the problem. E.g., find first unsatisfied clause, select first unset literal. Set it True.
2. **DLCS:** Find the variable with the most number of total occurrences. Set it to either True or False (based on whether more positive occurrences or more negative occurrences).
3. **Conflict – Fading.** Each variable used in a conflict clause has is priority increased by a constant value. Priorities fade multiplicatively. Select highest priority unset variable.
4. **Anti-greedy variable.** Find the last unsatisfiable clause, usually this means the last clause to have been learned that is not satisfied. Choose first unset literal from that clause. Set it True.
5. **Anti-greedy – clause.** If no current clause, find last unsatisfiable clause and make it current. Find first unset literal in current clause, and set it False.

Variable selection methods – continued

2-CNF methods:

Process all 2-clauses, to find all implications. If any variable has a value forced, set that variable. Otherwise, do one of:

Method (a): Find the variable ***x*** for with the set $S_{\boldsymbol{x}}$ is as "large" as possible. Here $S_{\boldsymbol{x}}$ is the set of variables that will be forced to have a value by ***x***'s being set to a particular value.

Method (b): Find the longest chain of variables $\mathbf{x_0, x_1, \ldots, x_k}$ each of which implies the next by 2-CNF reasoning. Set true in **reverse order**, one at a time.

6. **2-CNF – Conflict fading.** Measure the priority of a chain $\mathbf{x_0, x_1, \ldots, x_k}$ by using the sum of their conflict fading priorities.

7. **2-CNF – DLCS.** Measure the priority of a chain by using the sum of the DLCS scores (literal occurrence counts).

Only methods of type (b) have been implemented.

Comparison of Variable Selection Algorithms

Variable selection method	# times best	# times in top three	# times failed
2 CNF variable – conflict fading	2 / 2	5 / 6	8
2 CNF variable – DLCS	2 / 3	**8 / 8**	10
DLCS	**3 / 4**	6 / 5	11
Conflict fading	2 / 1	**8 / 9**	**4**
Variable – Greedy order	**4 / 3**	2 / 2	11
Variable – Anti-greedy order	**4 / 3**	2 / 2	7
Variable from clause – Anti-greedy order	2 / 3	4 / 5	10

Comparison of 19 easier tests from qualifying rounds of Sat Race 2006.
Counts "n / m" mean w.r.t. number of decision branches and number of variables set (respectively).

Algorithms are in "pure" form: No restarts, and no learned clauses are discarded.

Number of tests is small and results perhaps not statistically significant. (Behavior has a large variance.) Nonetheless, the difference between methods is surprisingly small.

Overall however, Conflict Fading is the best (as is usual in the literature.)

[New] Method of FirstDIP Clause Learning

FirstUIP - *Unique Implication Point* - clause learning (illustrated earlier) is usually considered the best clause learning method.

SatDiego implements a FirstDIP – Dual Implication Point – method too. In some cases, there is a natural smallest DIP clause to be learned (that is not a consequence of the current set of 2-SAT clauses).

Potential advantages:

(1) FirstDIP Clauses are smaller than FirstUIP clauses.

(2) FirstDIP Clauses are not immediately falsified, hence may be more useful.

Improvement in 51 tests (various variable selection methods) was statistically significant:,

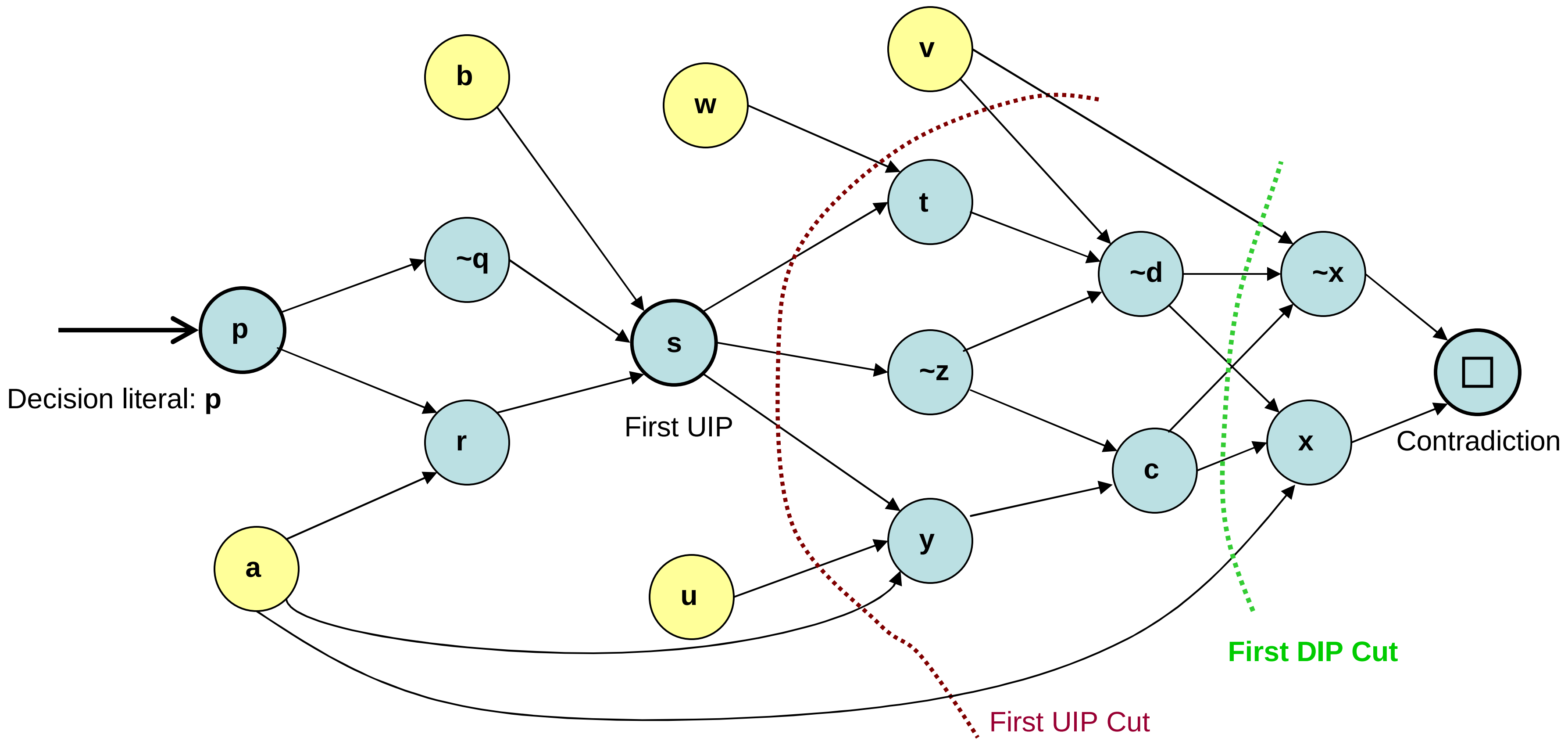
L = Log_{10}(UIP+DIP clause learning performance / UIP clause learning performance):

% = Corresponding percent improvement of UIP+DIP over DIP.

	Number Decisions	Number Unit Propagations	Run time
Approx. % improvement	49%	44%	34%
L (Log Ratio)	0.17	0.16	0.13
Std dev L	0.21	0.24	0.30

Similar work: Pipatsrsawat-Darwiche [t.a.,2009+] discuss DIP learning with UIP learning in cases where the DIP is not locally "absorbed", and report similar improvements. They achieve this without needing 2-CNF reasoning.

Example of DIP clause learning



First DIP Learned Clause: {~a,~c,d,~v}.
First UIP Learned Clause: {~a,~u,~s,~w,~v}.
With First-UIP/DIP: Both **p** and **s** can be set false when backtracking.

🟦 Blue for top level
🟨 Yellow for lower level literal

Fifo versus Lifo Variable order for Unit Propagation

Our original tests used a “Lifo” (Last-in-first-out) variable order during unit propagation.

That is, as new variables became unit variables, used the last one that became unit first.

Later, we tried a a “Fifo” (First-in-first-out) order.

Improvement in 26 tests (conflict fading and 2-SAT conflict fading only) was somewhat significant:

(Improvements look dramatic, but test size is smaller, and standard deviation is larger.)

L = Log_{10}(Fifo Unit Propagation performance / Lifo Unit Propagation performance):

% = Corresponding percent improvement of UIP+DIP over DIP.

	Number Decisions	Number Unit Propagations	Run time
Approx. % improvement	44%	88%	108%
L (Log Ratio)	0.15	0.28	0.32
Std dev L	0.45	0.71	0.83

I do not know what order is used in the extant SAT solver programs.

Discussion points – possible future research

- **How helpful are**
  - **(a) clause learning versus**
  - **(b) tracking decision levels carefully?**

  Experimental results?
- **Are there better clause learning methods?**
- **Are there better variable order selection methods?**
- **Can the FIFO/LIFO/2CNF variable order insights be extended to give substantially better methods?** (This is my next project for SAT solvers.)
- **Better statistical analysis.**
  The effectiveness of variable selection order seems to have a high variance. E.g., small changes in the conflict fade rate can have large differences in the algorithm's performance.
  Analyses of experimental comparisons different algorithms ought to come with a "level of confidence"/"statistical significance" rating.
- **SatDiego should discard old learned clauses (i.e., garbage collection).**
- **Restarts are reported to be very important.** SatDiego does not yet implement these.
- **Experiment with changing variable selection order dynamically,** for instance, combine restarts with changes in variable selection order.
- **Since conflict fading works best (??), SatDiego should implement 2-literal watching.** Or, should just be based on extant efficient solvers.