zChaff SAT solver overview

Lilia Yerosheva
CSE, University of Notre Dame, IN 46556
email: lyeroshe@cse.nd.edu

Outline

Overall purpose of this study

1. Learn about newly available SAT-based solver for applications with massive data sets - this presentation!
2. Study the complexity and the consumption of software and hardware resources for zChaff
3. Define an application specific time and space consumption for the given model
4. Find places for multithreading in branching, decision making, learning and backtracking processes
5. Develop a sequential reasoning engine that implements all the features of modern SAT solvers for model verification

SAT problem, CNF form

- Boolean Satisfiability Problem (SAT) is one of the central NP-complete problems that become the basis for many practical applications such as logic synthesis, automatic test generation, modeling and others.
- SAT solvers use a Conjunctive Normal Form (CNF) representation of the Boolean formula.
- CNF formula is presented as a conjunction of clauses, each clause is a disjunction of literals, a literal is a variable or its negation.
- CNF formula is satisfiable if each clause is satisfiable (or evaluate to true), otherwise, it is unsatisfiable.
- For a formula with *n* variables, there are *2^n* possible truth assignments.
- Maximum SAT problem. The maximum satisfiability problem is to find an assignment of values to the variables so as to have the maximum number of clauses evaluate to true.

SAT solver

- *Chaff SAT solver* is the most recent, complete and promising SAT solver.
- It is able to obtain up to *two orders of magnitude performance improvement* on difficult SAT benchmarks in comparison to other solvers.
- Competition links:

SAT03 competition:	http://www.Iri.fr/~simon/contest03/results/mainlive.php
SAT04 competition:	http://www.lri.fr/~simon/contest04/results/

SAT solver performance

Top 10 Sat-provers

This rank list is established over all benchmarks of SatEx where results are available. It is only indicative. <http://www.lri.fr/%7Esimon/satex/exp top.php3>.

Program	Time Used	Slow Ratio	#Solved	#Tested
zchaff	2 days, 22 h. 52 m. 29 s.	1.00	1280	1303
relsat-200	5 days, 19 h. 20 m. 50 s.	1.97	1260	1303
relsat	7 days, 9 h. 33 m. 20 s.	2.51	1243	1303
sato	8 days, 9 h. 35 m. 12 s.	2.84	1241	1303
satz-215	8 days, 22 h. 48 m. 56 s.	3.03	1237	1303
eqsatz	9 days, 0 h. 6 m. 8 s.	3.05	1237	1303
satz-213	9 days, 15 h. 18 m. 16 s.	3.26	1232	1303
sato-3.2.1	10 days, 9 h. 52 m. 52 s.	3.53	1221	1303
satz	10 days, 12 h. 59 m. 23 s.	3.57	1211	1303
modoc	11 days, 2 h. 27 m. 40 s.	3.76	1217	1303

SAT database can be summarized with the following numbers:

- Total CPU time of the SAT executions databases: 529 days, 17 hours, 53 minutes and 20 seconds (of a PII-400 Under Linux when counting Resigned executions as 10000s).
- Number of Benchmarks: 1303, grouped in different families.

Input for zChaff

- **zChaff input files use the DIMACS graph format to handle all types of SAT problems.**
- **File contains two major sections: the preamble and the clauses.**
- **Information is contained in lines, each line begins with a single character that defines the type of line.**

The preamble part

- comments:

c This is an example of a comment line

- problem line:

p FROMAT VARIABLES CLAUSES

where FORMAT - a field to determine the expected format ("CNF"
    VARIABLES - contains an integer n, the number of variable in the instance
    CLAUSES - contains an integer m, the number of clauses in the instance

The clauses part

- Appear immediately after the problem line.
- Variables are numbered from 1 up to ***n***, it is not necessary that every variable appear in an instance.
- A variable ***i*** is represented by ***i***, the negated version - by ***-i***.
- Each clause is terminated by the value 0.

```
c Example CNF format file
c
p cnf 4 3
1 3 -4 0
4 0 2
-3 0
```

```
c Sample SAT format
c
p sat 4
(*(+(1 3 -4)
+(4)
+(2 3)))
```

DIMACS implementation

- CNF format files have a .cnf extension.
- SAT format files have a .sat extension.

zChaff output

- Comment lines begin with character c and those can be anywhere in the file.
- Solution lines start with s and can be of two types:

s SOLUTION VARIABLES CLAUSES

s SOLUTION VARIABLES

where SOLUTION contains an integer corresponding to the solution value: for max problem - # of clauses satisfied, for SAT problem - 1, 0, or -1 if no decision was reached.

- Timing line reports the timing information for statistical analysis:

t TYPE SOLUTION VARIABLES CLAUSES CPUSECS MEASURE1...

where CPUSECS - flp number of CPU seconds used during the solution, MEASURE1,...- flp numbers to measure performance. It is what application thinks is the most significant measure of performance: # of nodes in search space, # of var-assignment changes, memory requirements (might vary by arch.), etc.

- Variable line - provides variable values (i - variable should be set true or -i - false)

v V

- Clause satisfaction line denotes whether a particular clause is satisfied or not (i or -i).

s C

Core code for zChaff

DPLL - Davis-Putnam-Logemann-Loveland algorithm (1960, 1962) - performs a methodical enumeration of assignments, looking for one that satisfies the formula.

- Most solvers are based on DPLL algorithm which performs a branching search with back-tracking
- It finds a solution iff the formula is satisfiable.

```
sat_solve()
  if preprocess() = CONFLICT then return UNSAT;
  while TRUE do
     if not decide-next-branch() then return SAT;
     while deduce() = CONFLICT do
        blevel <= analyze-conflict();
        if blevel=0 then return UNSAT;
        backtrack(blevel);
     done;
  done;
```

The basic skeleton of DLPP-based SAT solvers, adapted from the GRASP [1] work.

Core code functions

decide-next-branch() - chooses an unassigned branch variable and a value to it.

deduce() - unit propagation: if an unassigned variable exists, we make a decision about the variable assignments deductible from this decision using *BCP.*

BCP consists of iterative application of the unit clause rule (which is invoked when a clause becomes a unit clause).

BCP rule - the last unassigned literal is implied to be true. It avoids the search path where the last literal is also false, since such a path can't lead to a solution. If no conflict is discovered - choose the next var for making a decision, otherwise - resolve conflict by backtracking.

analyze-conflict() - finds out the reason for conflict (occurs when a var is implied to be true as well as false)

backtrack() - is performed in order to undo some decisions and their implications.

Basic definitions

- Unit clause - a clause where all but one of its literals are false, and the remaining literal is unassigned.
- Unit literal - the unassigned literal in a unit clause.
- Unit clause rule - if all but one of its literals has been assigned the value 0, then remaining (unassigned) literal must be assigned to 1 for this clause to be satisfied.
- Unit propagation, or *Boolean Constraint Propagation (BCP)* - a process of assigning 1 to all unit literals.
- Implication - a process of assigning the variable associated with the long remaining literal a value that makes the literal in the given clause true.
- Decision - a process of assigning a value to a variable (any value).
- Backtracking - a process of reassigning a decision which caused the conflict.

Components of zChaff

1. Branching heuristics
   - occurs in ***decide_next_branch()***, when no more deduction is possible, the function will choose one var from all the free vars and assign it to a value.
   - effects the efficiency of the solver, well choosing var can decrease search trees sizes drastically.

2. Deduction algorithm
   - occurs in ***deduce()***; when a branch var is assigned a value the entire clause database is simplified.
   - determines consequences of the node decision.
   - can use different BCP mechanisms. BCP takes the most significant part of the run time.

3. Conflict analysis and learning
   - when a conflicting clause is encountered, the solver performs *backtrack* and undo the decisions.
   - *Conflict analysis* finds the reason for a conflict and tries to resolve it.
   - *Learning* is a way to generate learned clauses with some direction in resolution process.

4. Data structure for storing clause database
   - to store vars, clauses, learned and new generated clauses for resolving instances.

5. Preprocess, restart and other techniques
   - *Preprocessor* is an extra deduction mechanism applied at the beginning of the search. Applied once => usually incorporates the most expensive deduction rules to every node of a tree.
   - *Restart* - randomly removes already used search space and starts from scratch (saves learned clauses).
   - *Portfolio design* uses different solving strategies in one solving process to make solver robust.
   - *Randomized backtracking* - performs unrestricted backtracking.

Branching heuristics

1. Bohm’ Heuristic, Maximum Occurrences on Minimum sized clauses (MOM), Jeroslow-Wang
   - early branching heuristics.
   - are greedy algorithms: trying to make the next branch generate the largest number of implications or satisfy most clauses.
   - use some f’ns to estimate the effect of branching on each free var, and choose the var that has the maximum function value.
   - work for some instances (random), but not for structured problems: all f’ns are based on statistics of the clause database such as clause length, etc.

2. Dynamic Largest Individual Sum (DLIS)
   - counts the number of unsatisfied clauses in which a given var appears in either phase, or at each decision choose the assignment that satisfies the most unsatisfied clauses.
   - the counts are state-dependent: different var assignments will give different counts.
   - considerable work is required to maintain the statistics necessary for this heuristic - for one implementation:

• must touch “every” clause that contains a literal that has been set to true (often restricted to initial, not learner, clauses.
• must maintain “sat” counters for each clause
• when counters transition 0->1, update ranking
• the counts for all free vars need to be recalculated.

Branching heuristics

3. Variable State Independent Decaying Sum (VSIDS) - in zChaff

- *Rank variables* by the number of occurrences of a literal in the initial clause database for a problem.
- Only increment counts as new clauses are added.
- Periodically, divide all counts by a constant.

- VSIDS score - is a literal occurrence count with higher weight on the most recently added clauses.
- Choose the free var with the *highest combined score to branch*.
- Decision strategy: when conflict occurs, all literals in the clauses that are responsible (involved in resolution that adds learned clauses) for the conflict will have their score increased.
- Score is *decaying* toward "recent" conflicts.
- Use *heap* to find unassigned *vars with the highest ranking*.
- Decision heuristic influences which vars appear in learnt clauses:
  decision => implications => conflicts => learnt clauses
- Important for decisions to keep *search strongly localized*.
- Quasi-static: it doesn't depend on var state but changes as new clauses are added.
- Cheap to maintain, takes small % of the total run-time for problems with ~million vars.

4. Look-ahead algorithms (satz) and Backbone-directed heuristics (cnfs)

- effective on difficult random problems (current solvers: only random 3-SAT problems with 100s vars most)
- expensive compared with VSIDS, unacceptable for the larger well-structured problems).

Deduction algorithm: BCP mechanisms

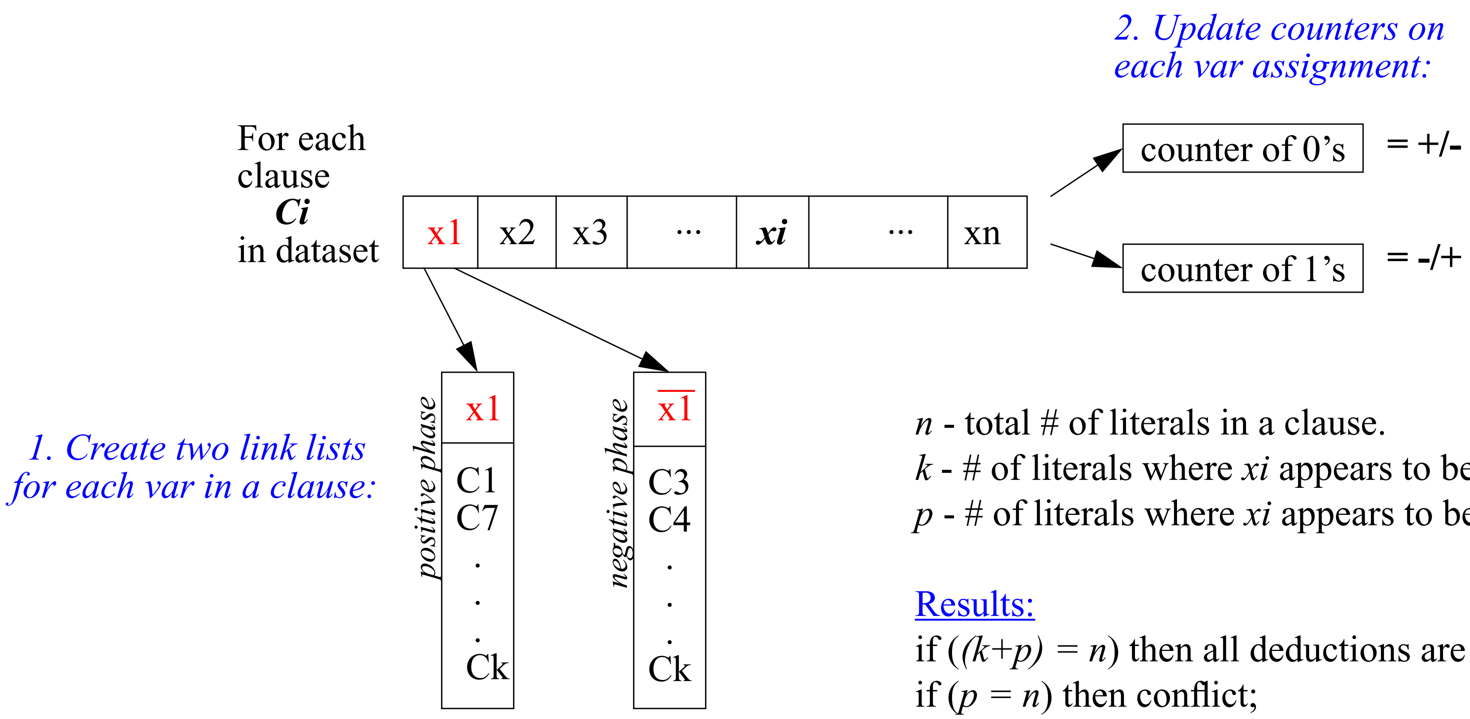
GRASP SAT solver

- *Each clause* keeps two counters: one for the number of value 1 literals in the clause and the other for the number of value 0 literals in the clause.
- Each var has two lists that contain all the clauses where that *var* appears as a positive and the negative literal, respectively.
- When a var is assigned a value, all the clauses that contain this literal must update their counters.
- If counter of 0's is equal to the total number of literals in the clause => *conflicting clause*.
- If counter of 0's is one less that the total number of literals in the clause and counter of 1's is 0 => *unit clause*.

Conclusions

- Not efficient: if the instance has m clauses and n vars, and the average each clause has l literals, then when a var gets assigned, on the average lm/n counters must be updated.
- On backtracking - lm/n undo on counters on average.
- Learning mechanism - increases the number of clauses!

GRASP SAT solver


2. Update counters on each var assignment:
For each clause Ci in dataset
x1
x2
x3
...
xi
...
xn
counter of 0's
= +/-
counter of 1's
= -/+
1. Create two link lists for each var in a clause:
positive phase
x1
C1
C7
.
.
.
Ck
negative phase
$\overline{x1}$
C3
C4
.
.
.
Ck

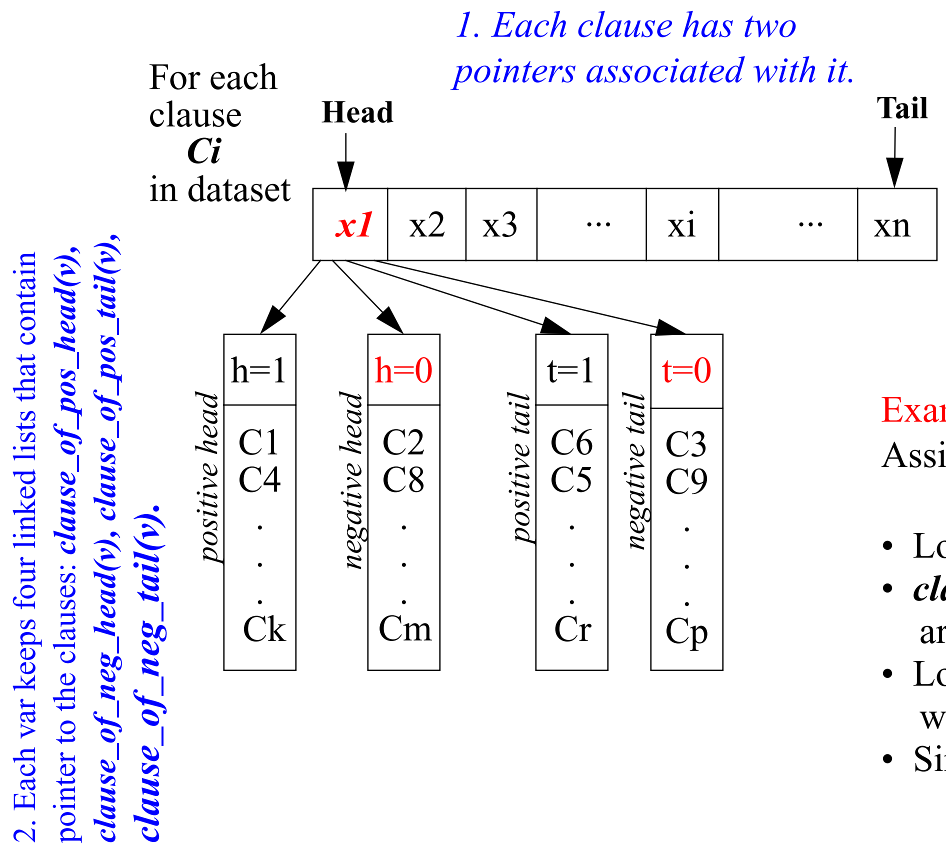
n - total # of literals in a clause.
k - # of literals where *xi* appears to be 1.
p - # of literals where *xi* appears to be 0.

Results:
if ((*k+p*) = *n*) then all deductions are done;
if (*p* = *n*) then conflict;
if (*p* = *(n-1)*) then unit clause;

Backtracking:
- undo var assignments for current decision;
- update counters for each clause.

BCP mechanisms (cont.)

SATO SAT solver (head/tail)


1. Each clause has two pointers associated with it.
For each clause ***Ci*** in dataset
Head
Tail
x1
x2
x3
...
xi
...
xn
h=1
C1
C4
Ck
h=0
C2
C8
Cm
t=1
C6
C5
Cr
t=0
C3
C9
Cp
positive head
negative head
positive tail
negative tail
2. Each var keeps four linked lists that contain pointer to the clauses: ***clause_of_pos_head(v)***, ***clause_of_neg_head(v)***, ***clause_of_pos_tail(v)***, ***clause_of_neg_tail(v)***.

A clause is stored in an array of literals: the head points to the beginning of the array, the tail - to the last literal of the clause.

Example.
Assigning a variable v = ***x1=1***.

- Look at lists with *head/tail = 0*;
- ***clause_of_pos_head(x1)*** and ***clause_of_pos_tail(x1)*** are ignored.
- Look for each ***Ci*** in ***head=0***, and check if value ***x1=1*** won't produce a *conflict*.
- Similar for ***tail=0***, only search is in reverse direction.

SATO SAT solver (head/tail)

- For each clause C in the ***clause_of_neg_head(v)***, the solver searches for a literal that does not evaluate to 1 *from* the position of the head literal of C *to* the position of the tail literal of C.

Four cases may occur:

- If first we found a literal that evaluates to 1 - clause is ***SAT***, do nothing.
- If first we found a literal ***l*** that is *free* and ***l*** is not the tail literal, then we *remove C from* ***clause_of_neg_head(v)*** and *add C to* the head list of the var corresponding to l (moving a head pointer).
- If all literals between pointers are 0's, and the tail is unassigned - *unit clause*.
- If all literals in between these pointers and the tail literal are 0's - *a conflicting clause*.

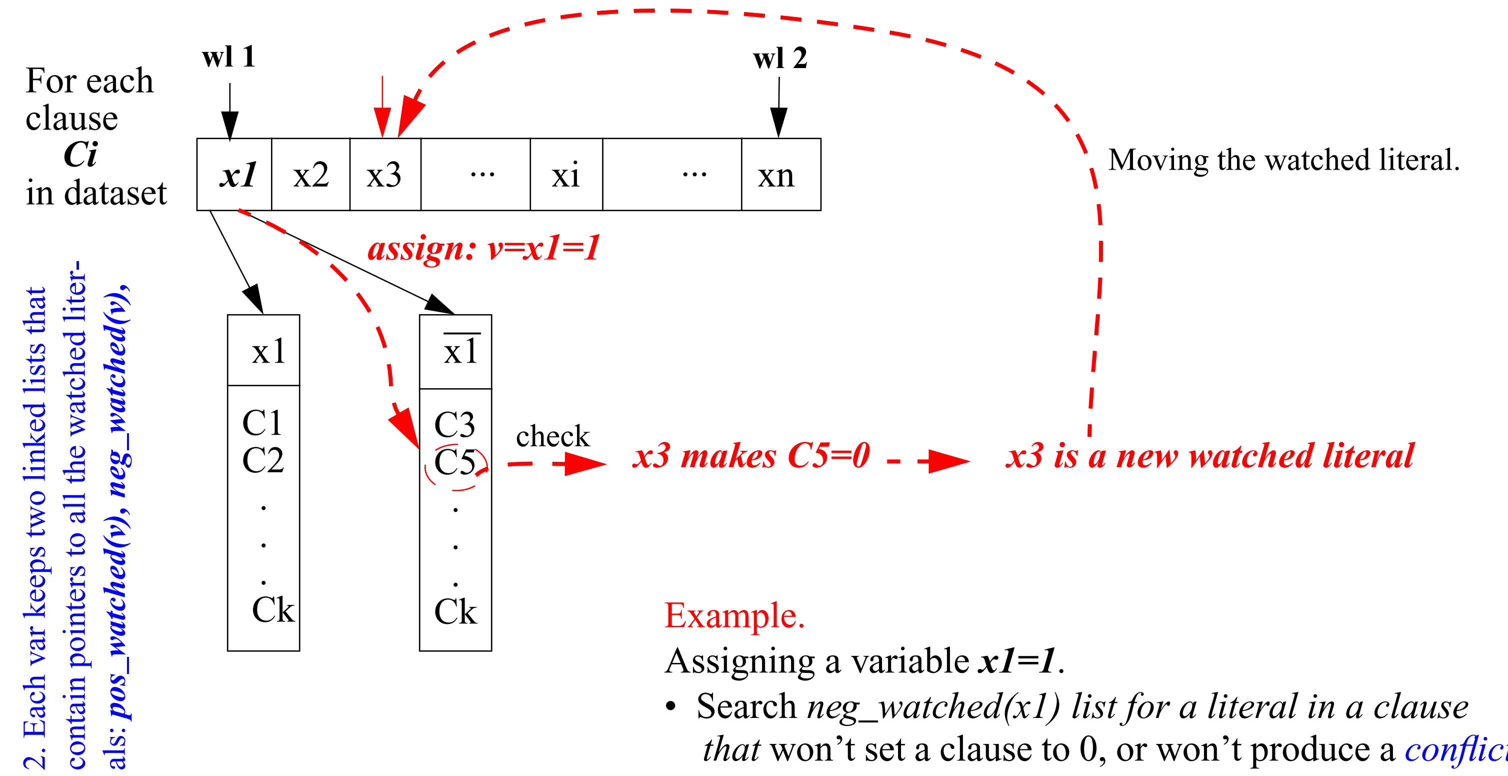
Conclusions

- Method is faster than GRASP: each clause has only 2 pointers => ***m/n*** clauses must be updated on average.
- Undoing a var's assignment during backtracking has the same computational complexity as assigning the variable.

BCP mechanisms (cont.)

zChaff SAT solver (2-literal watching)

1. Each clause has two watched literals that initially free, can move in either direction.


For each clause *Ci* in dataset
wl 1
wl 2
x1
x2
x3
...
xi
...
xn
assign: v=x1=1
Moving the watched literal.
x1
C1
C2
Ck
$\overline{x1}$
C3
C5
Ck
check
x3 makes C5=0
x3 is a new watched literal
2. Each var keeps two linked lists that contain pointers to all the watched literals: *pos_watched(v), neg_watched(v),*

Example.
Assigning a variable ***x1=1***.

- Search *neg_watched(x1) list for a literal in a clause that* won't set a clause to 0, or won't produce a *conflict*.

zChaff SAT solver (2-literal watching)

- When var ***v*** is assigned 1, for each literal ***p*** pointed to by a pointer in the list of ***neg_watched(v)***, the solver searches for a literal ***l*** in the clause (containing ***p***) that ***is not set to 0.***

Four cases may occur:

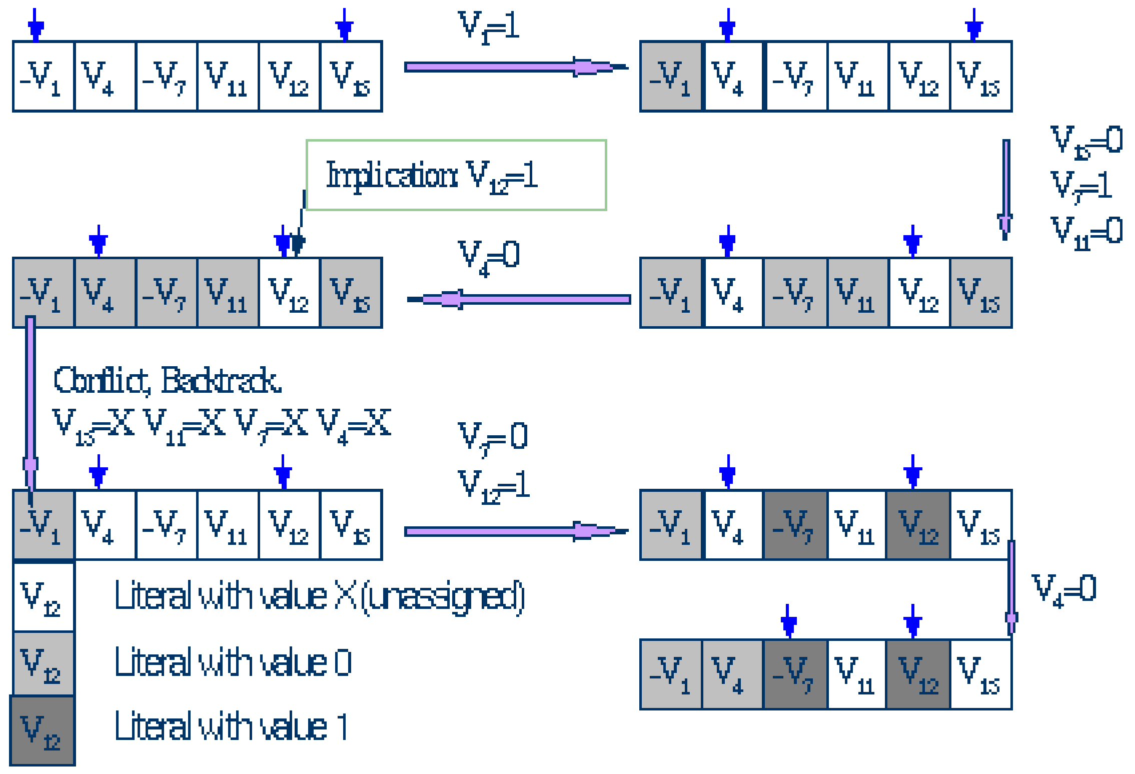
- if the only such ***l*** is the other watched literal and it *evaluates to 1* - clause is *SAT*, do nothing.
- if exists such literal ***l*** and it is not the other watched literal, then e remove to p from ***neg_watched(v)***, and add pointer to ***l*** to the watched list of the variable corresponding to ***l*** - moving watched literal.
- if the only such ***l*** is the other watched literal and it is *free*, then - unit clause, and the other watched literal is the unit literal.
- if all literals in the clause are 0's, and no such ***l*** exists - a conflicting clause.

Conclusion

- Undoing a var assignment during backtrack takes constant time.
- The two watched literals are the last to be assigned to 0 => any backtracking will make sure that the literals being watched are either unassigned or assigned to 1.
- No updates on pointers for the literals being watched.
- The fastest algorithm today.

zChaff SAT solver (2-literal watching)

- Example of BCP for a clause (v1’+v4+v7’v11+v12+v15).



Other deduction algorithms

a) Unit clause rule (considered previously)
- guarantees that all unit clauses are consistent with each other.

b) Pure literal rule:
- if a var occurs only in a single phase in all unresolved clauses assign its value 1.
- Simple, but generally slow down the solving process for most cases.

c) Equivalence reasoning (eqsatz)
- add a pattern-matching scheme for equivalence clauses
- proposes to include more patterns in the matching process for simplification purpose in deduction.

d) Recursive learning
- reasoning technique for SAT problems with a logic circuit representation of a formula.

e) Subsequent research
- reasoning technique for SAT benchmarks generated from combinational circuit equivalence checking problems.

Conflict analysis and learning

DPLL algorithm

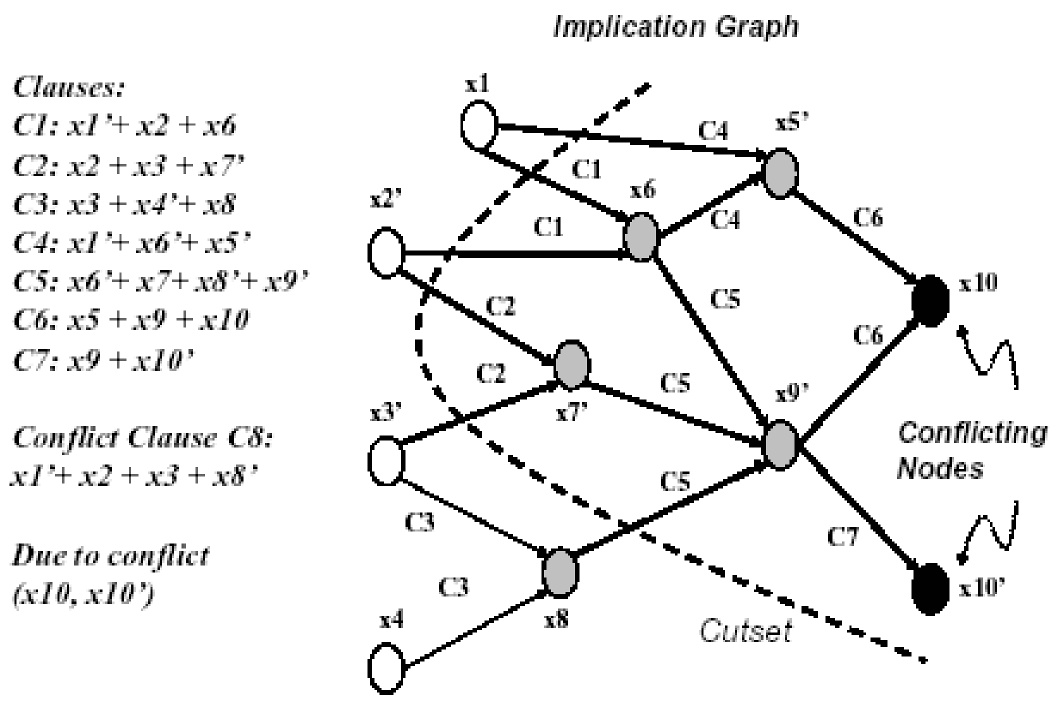
- solver keeps a flag indicating whether it has been tried in both phases (flipped) or not;
- when a conflict occurs, the decision var with the highest decision level that has not been flipped is marked as flipped; all assignments b/w that decision level and current decision level undone, and other phase for decision var is performed (chronological backtracking) - well for random generate SAT instances.
- usually backtracking to earlier decision level that the last unflipped decision (non-chronological backtracking or conflict-directed backjumping) - good for structured problems.
- adding learned “conflict” clauses to clause database - conflict-directed learning

- Conflict-driven learning - consists of adding conflict clauses to the formula in order to avoid the same conflict in the future
- Conflict driven backtracking - allows nonchronological backtracking up to the closest decision which caused the conflict.

Implication graph is an analyzing tool for learning and non-chronological backtracking.

Implication graph

- is a component of conflict analysis which captures the current state of SAT solver, where:
nodes represent assignments to variables;
edges - represent clauses, which cause implications due to source nodes on sink nodes;
decision assignments - nodes with no incoming edges;
conflict - when there are two nodes with opposite values assigned to the same variable.



Clauses:
C1: x1'+ x2 + x6
C2: x2 + x3 + x7'
C3: x3 + x4'+ x8
C4: x1'+ x6'+ x5'
C5: x6'+ x7+ x8'+ x9'
C6: x5 + x9 + x10
C7: x9 + x10'

Conflict Clause C8:
x1'+ x2 + x3 + x8'

Due to conflict
(x10, x10')

Conflict analysis - is performed by following back the edges from the conflicting nodes, up to any edge cutset which separates the conflicting nodes from the decision nodes.

Here: x1=1, x2=0, x3=0, x8=1 always leads to conflict. For *conflic-driven learning*: C8 clause is added to the clause database in order to avoid the same conflict in the future).

Implication graph resolution

Conflict with node x10

c4: x1’+x6’+x5’-> x1=1(was), x5=0(impl) -> 0+x6’+1 (sat)

c1: x1’+x2+x6 -> x1=1, x2=0(was), x6=1(impl - unit clause var) -> 0+1+1 (sat)

c6: x5+x9+x10 -> x5=0(was), x10=1(impl) -> 0+x9+1

c5: x6’+x7+x8+x9’ -> x6=1(was), x9=0(impl) -> 0+x7+x8+1 => x7=0 (from c2) -> 0+0+x8+1 (sat)

c2: x2+x3+x7’ -> x2=0, x3=0(was), x7=0(impl. - unit clause var) -> 0+0+1 (sat)

c7: x9+x10’ -> x9=0(assign.), x10=0(unit clause var) -> 0+1 (sat) **=> conflict on x10 with c6 assignment!**

- We did not use var x8 in the resolution process => did not include in the cutset for backtracking.

Conflict on node x8

c3: x3+x4’+x8 -> x3=0, x4=1(was), x8=1(unit clause assign) -> 0+0+1 (sat)

c8: x1’+x2+x3+x8’ -> x1=1, x2=0, x3=0(was), x8=0(unit clause assign) -> 0+0+0+1 (sat) =>
conflict on x8!

Conclusion:

When referring to an implication graph, we only consider the connected component that has the conflicting var in it. The rest of the implicit graph is not relevant for the conflict analysis.

Data structures overview

- allow to optimize clauses for memory utilization, access locality, prevent duplication, etc.

1. Sparse matrix representation is a linear way to store clauses.

- Each clause occupies its own space.
- No overlap exists between clauses.
- *The dataset size* is linear in the number of literals in the clause database.

2. Linked lists and array of pointers pointing to structures to store the clause database.

- Convenient for manipulating the clause database (add/delete clauses).
- They are not memory efficient.
- Cause a lot of cache misses during solving process because of lack of access locality.

Data structures overview

3. SATO: data structure “trie” - a ternary tree.

- Each *internal node* in the trie structure is *a var index*, and its ***3 children*** are ***Pos, Neg, DC*** *(don't care)*.
- A leaf node is either *True* or *False*.
- Each path from root of the trie to a True leaf *represents a clause*.
- A trie is said to be *ordered* of for every internal node V, Parent(V) has a smaller var index than the index of var V.
- Ordered trie str able to detect duplicates and tail subsumed clauses of a database quickly.
- A clause is *tail subsumed* by another clause if its first portion of the literals is also a clause in the clause database.

4. Chaff: stores clauses in large arrays

- Arrays are not so flexible => garbage collection is needed on deletion.
- It's very efficient in memory utilization.
- Occupies a contiguous memory space => access locality is increased.
- It has advantage on cache misses => substantial speed-up in solving process.

zChaff verification

- Used to generate counterexamples when solving SAT problems: opinion on satisfactory solution of a hard-to find problems.
- Two model checking techniques:

  - bounded - to search for counter-examples to a given problem property in the space of all executions that is bounded by some integer *k*.

  - unbounded - include methods that can prove the correctness of a property on a problem as well as find counter-examples for failing properties in unbounded space of all executions.

Examples of zChaff benchmarks (SATLIB)

- B28: VLIW-SAT-4.0: 10 sat CNF formulas from formal verification of VLIW processors with instruction queues and 9-stage pipelines. These formulas are from buggy variants of the most complex model in VLIW-UNSAT-4.0. Formats: [CNF <http://www.cs.ubc.ca/%7Ehoos/SATLIB/I-Velev03/vliw_sat_4.0.tar.gz>] Note: The biggest file is over 300 MB.
- B27: VLIW-UNSAT-4.0: 4 unsat. CNF formulas from formal verification of VLIW processors with instruction queues and 9-stage pipelines. Formats:[CNF <http://www.cs.ubc.ca/%7Ehoos/SATLIB/I-Velev03vliw_unsat_4.0.tar.gz>. Note: The biggest file is over 180 MB.
- B26: VLIW-SAT-2.1: 10 sat. CNF formulas from formal verification of VLIW processors with instruction queues. These formulas are from buggy variants of a more complex VLIW processor than the one used for VLIW-SAT-2.0. Formats: [CNF <http://www.cs.ubc.ca/%7Ehoos/SATLIB/I-Velev03/vliw_sat_2.1.tar.gz>]. Note: The biggest file is over 390 MB.
- B25: VLIW-SAT-2.0: 10 sat. CNF formulas from formal verification of VLIW processors with instruction queues. Formats:[CNF <http://www.cs.ubc.ca/%7Ehoos/SATLIB/I-Velev03/vliw_sat_2.0.tar.gz>]. Note: The biggest file is over 190 MB.
- B24: VLIW-UNSAT-2.0: 9 unsat. CNF formulas from formal verification of VLIW processors with instruction queues. Formats:[CNF <http://www.cs.ubc.ca/%7Ehoos/SATLIB/I-Velev03/vliw_unsat_2.0.tar.gz>]. Note: The biggest file is over 210 MB.
- B23: PIPE-SAT-1.1: 10 sat. CNF formulas from buggy variants of the pipe benchmarks, and generated as presented in paper [C20<http://www.ece.cmu.edu/%7Emvelev/sat_benchmarks.html#C20>]. Formats: [CNF <http://www.cs.ubc.ca/%7Ehoos/SATLIB/I-Velev03/pipe_sat_1.1.tar.gz>]. Note: The biggest file is over 100 MB.
- B22: LIVENESS-UNSAT-2.0: 9 unsat. CNF formulas from formal verification of liveness of single-issue pipelined and dual-issue superscalar DLX processors. The formulas are generated after applying abstraction techniques. Formats: [CNF <http://www.cs.ubc.ca/%7Ehoos/SATLIB/I-Velev03/liveness_unsat_2.0.tar.gz>]. Note: The biggest file is over 130 MB.

SATLIB

Web site link:

http://www.lri.fr/~simon/satex/exp/menu_choix.php3

, where can choose exactly what kind of results you want to view, select a number of programs and a number of families of benchmarks, then the type of cpu-time summary requested.

Families of benchmarks:

- DLL-Backtrack
- Original DP
- Non Classical
- Randomized DLL
- Uniform Random-3-SAT
- Random-3-SAT Instances with Controlled Backbone Size

Sources

1. Boolean Satisfiability Research Group at Princeton. http://www.princeton.edu/~chaff/
2. Zchaff: A fast SAT solver. 2002. http://bears.ece.ucsb.edu/class/256bd/RCFB256B3450-Zchoff.pdf
3. SAT04 Contest: Live progress of zchaff http://www.lri.fr/~simon/contest04/results/livesolver.php?idsolver=18
4. Chaff: Engineering an efficient SAT solver, Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik, DAC 2001, June 18-22, 2001, Nevada, USA.
5. Satisfiability suggested format, DIMACS official format, ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/
6. A survey of recent advances in SAT-based formal verification, Mukul Prasad, Armin Biere, Aarti Gupta, in Software Tools for Technology.
7. The quest for efficient boolean satisfiability solvers, Lintao Zhang, Sharad Malik, 39th DAC, 2001, http://staff.science.uva.nl/~bcate/AR2005/zhang02quest.pdf
8. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver, http://www.sigda.org/Archives/ProceedingArchives/Iccad/Iccad2001/papers/2001/iccad01/pdffiles/06a_1.pdf
9. 2000 IEEE International Conference on Computer Design: VLSI in Computers; Processors(ICCD'00), Sept. 17 - 20, 2000, Austin, Texas.
10. 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02) September 16 - 18, 2002, Freiburg, Germany.
11. Efficient data structures for backtrack search SAT solvers. http://www.ingentaconnect.com/content/klu/amai/2005/00000043/F0040001/05379425
12. Boolean Satisfiability Solver Performance Comparison. Conclusion. Chaff: VSIDS(Variable State Independent Decaying Sum). http://www.cis.upenn.edu/~lee/02cis640/slides/SAT.ppt
13. Parallel propositional satisfiability checking with distributed dynamic learning. Source: Parallel Computing archiveVolume 29, Issue 7(July 2003) table of contents Pages: 969 - 994. 2003 ISSN:0167-8191. Wolfgang Blochinger: Symbolic Computation Group, Wilhelm-Schickard-Institute, Univ. of Tübingen, Sand 14, 72076 Germany.

Future work

- Study the complexity and the consumption of software and hardware resources for zChaff.
- Define an application specific time and space consumption for the given model.
- Find places for multithreading in branching, decision making, learning and backtracking processes.
- Develop a sequential reasoning engine that implements all the features of modern SAT solvers for model verification.