Zchaff: A fast SAT solver

* We'd like to build a complete decision procedure for SAT which is efficient.

Generalized *D-P-L* algorithm:

```
while (true)  {
        if (! decide( ))  /* no unassigned variables */
                return (sat)
        while (! bcp ( )) {
                if (! resolved_conflict ( ))
                        return (not sat);
        }
}

bool resolve_conflict ( )  {
        d= most recent assignment which is not exhausted;
        if (d==Null)
                return (false);
        else  {
                flip value of d;
                mark d as exhausted;
                undo invalidated assumptions; (i.e., u_bcp(d))
                return (true)
        }
}
```


Zchaff: A fast SAT solver

* Decide ( ) chooses an unassigned variable and sets it to a value. Keep a list of current assignments of length (the decision depth). (Returns false only if all variables are exhausted.)

* bcp ( ) Boolean Constraint Propagation

  bcp ( ) identifies unit clauses and trosetively sets forced literals until:
  a) conflict is found (i.e., current assignments are not consistent)
  b) all unit clauses are assigned.

  $\Rightarrow$ a unit clause is one which every literal but one is false $\Rightarrow$ to satisfy clause that literal must be true.

  Eg: $(a' + b + c)(a + b + c')(a + d)$

  if $a \rightarrow$ false; last clause forces $d \rightarrow$ true.

  or if $a \rightarrow$ true, $b \rightarrow$ false; first clause must set $c \rightarrow$ true.

  bcp is a powerful look-ahead mechanism since the forced literals can create additional unit clauses $\Rightarrow$ run to transitive closure.

* On the other hand: such implications are based only on the set of current variable decisions. $\Rightarrow$ if a decision is backtracked, we must also remove all the implied assignments made by bcp ( ). It is typical to record the decision values by the level ℓ.

$\Rightarrow$ Thus if we associate each implied assignment by it level, we need only remove those implications higher than ℓ' for a new assignment at level ℓ'. This is what u_bcp ( ) does.

Zchaff: A fast SAT solver

Observation (Zhang): Majority of run time of solver is in bcp.

To make an efficient solver, we must have an efficient bcp. It seems that we need to check each clause to see if it is a unit clause, but this is not so:

(we only care if the clause has all but one literal set.)

⇒ choose any two variables (not zero) in each clause. Then need only check the clauses that have one of these values assigned to zero. ⇒ in any other case, the clause has two non-zero or unassigned values and is not unit.

So to make bcp, we visit only clauses with changed to zero watched variables.

a) Clause may still not be unit ⇒ there is another literal in clause which is not zero and is not one of the watched variables. ⇒ choose it to replace the watched variable just assigned.

b) Clause is unit ⇒ other watched variable is implied to be true.

c) Note: if some clause variable is true, remove it from bcp table since it cannot be made false by implications.

Zchaff: A fast SAT solver

Note that when we backtrack, we need not change the watched literals
⇒ undoing an assignment is very fast.

better: if a variable is assigned and unassigned, re-assignment is likely to be faster since number of clauses in which the variable is watched is likely to decrease.

Eg: $\left(v_1' + v_4 + v_7' + v_{11} + v_{12} + v_{15}\right)$:

value → $\frac{v_1 v_4 v_7 v_{11} v_{12} v_{15}}{------}$

watched ptn. → $\frac{w \quad\quad w}{0 - - - - -}$ $v_1 \leftarrow 1$

$\frac{w \quad\quad w}{0 \quad 0\,0 \quad 0}$ $v_{15} \leftarrow 0, v_7 \leftarrow 1, v_{11} \leftarrow 0$

$\frac{w \quad\quad w}{0\,0 \;\; 0\,0 \;\; 1 \;\; 0}$ $v_4 \leftarrow 0 \Rightarrow v_{12} \leftarrow 1$

$\frac{w \quad\quad w}{0 \;\; \underset{w}{-} - - \underset{w}{-} -}$ conflict backtrack $v_{15} = v_{11} = v_7 = v_4 = x$

Zchaff: A fast SAT solver

decide ( ) heuristic

Tradeoff between time and accuracy is very strong here…

Chaff/Zchaff uses this one:

1) Each literal has a counter assigned
2) For each added clause, the counter for each contained literal is incremental.
3) Choose largest unassigned literal at each decision point.
4) Ties broken randomly
5) Rescale all counts periodically.

⇒ Idea is to concentrate effort on set of recent conflict clauses. They claim 10x (1000%) improvement over random in hard problems.

Zchaff: A fast SAT solver

Problem Learning and Conflict Clauses (Chaff, Grasp, Sato, Rel-SAT)

Idea: Use the set of trials as a means to derive constraints on the search by adding new clauses which prevent re-searching some parts of the space.

Idea 2: When resolving the conflicts, back track to best level, regardless of chronology of search. i.e., non-chronological backtrack.

DPL - with learning:

```
while (1)  {
        if (choose_next_branch ( ))  {
                while (deduce ( ) ==conflicts)  {
                        bk_level = analyze_conflicts ( );
                        if (bk_level = 0)
                                return (unsat);
                        else backtrack (bk_level)
                }
        }  else
                return (sat);
}
```

choose_next_batch ( ) = heuristic to select next literal

deduce = bcp ( ), but now if a conflict is created, we analyze it to see the reason.

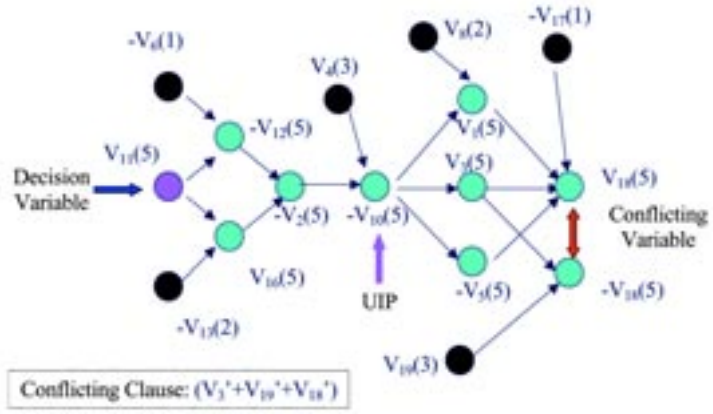
Zchaff: A fast SAT solver

analyze_conflicts ( ) adds some new clauses to the database based on state of current search and determines the level to be backtracked to to avoid redoing the problems.

* How to determine what caused the conflict?

Consider the unit implications which lead to a conflicting assignment: if we add a vertex for each literal with a value; i.e., assigned then implications from unit conflicts provide directed edges to new literal valuations.

Since a literal can have only one value, each vertex corresponds to a single literal, except for the conflicting assignment literal.



Zchaff: A fast SAT solver

def: a dominator of a vertex is a node ∋ ∀ path from the decision node of the dominator to the vertex must traverse the dominator.

def: a unique implication pt. (UIP) is a vertex at the current level which dominates both conflicting variable nodes.

Eg: In the graph, we are at level 5, so decision node is v_{11}. $-v_{10}$ dominates both v_{18} and $-v_{18}$.
$\Rightarrow v_{10}$ is UIP

(the decision variable is always a UIP)
note: v_{11},$-v_{10}$ and $-v_2$ all UIP's.

Intuitively, a UIP is a the reason for the conflict.

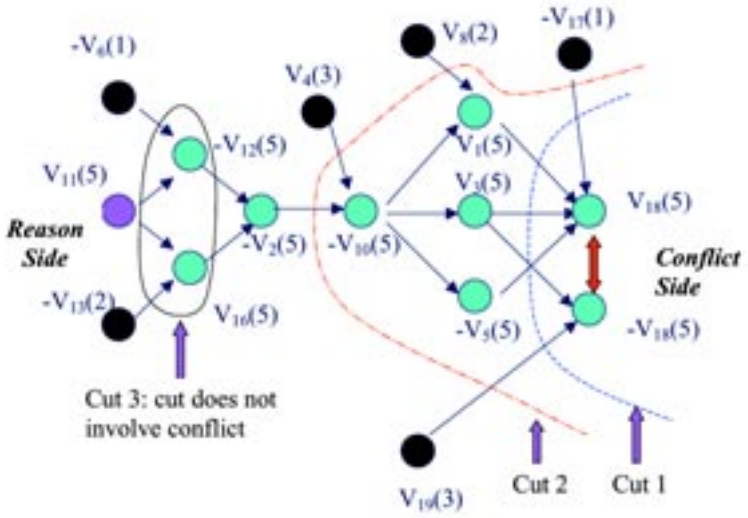
→ (to implement, we point each implied variable to the unit clause which implied it.)

Zchaff: A fast SAT solver

Consider cut 1 in the figure below: Every implication crossing the cut contributes to the conflict.

* ⇒ note we can write this condition as follows:

$$\left(v_{17} + v_1' + v_3' + v_5 + v_{19}'\right)$$



Zchaff: A fast SAT solver

Thus we need never search this subspace again. By adding the new clause, it will imply variable settings to avoid this subspace in the future.

Two problems:

1) Size of added clause may make the implication very weak; i.e., hard to make unit.
2) Number of such clauses grows rapidly ⇒ size of CNF can exceed memory limits.

note: we could choose cut 2 ⇒

$$\left(v_{17} + v_8' + v_4' + v_2 + v_{19}'\right)$$

which involves variables (except v_2) all from early decision levels. Since we must backtrack, the earlier we can backtrack and the larger clause (earlier variables), the larger subspace of the current search is removed.

note: we can generalize UIP for a given cut by making a vertex at level ℓ UIP iff any path from the vertex decision variable for level ℓ to the conflict must either go through *a* or a vertex at a higher level than *a*.

It turns out that this new def. is actually independent of the cut…

note: cut 3 picks out the antecedents for $-v_2$ ⇒

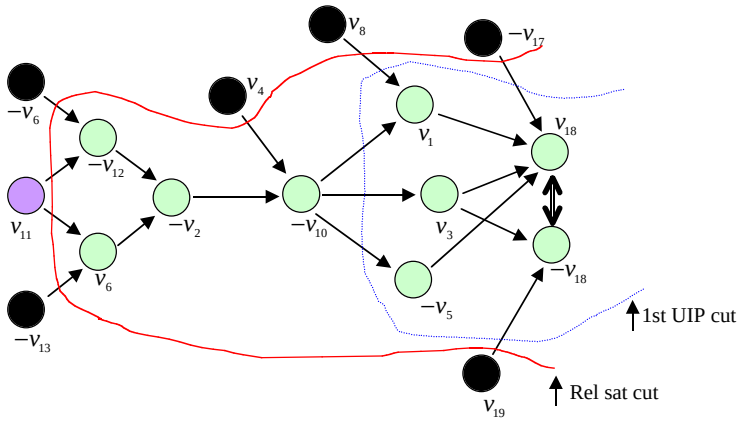
$$\left(v_2' + v_{16}' + v_{12}\right)$$ (this clause is consistent with the current set, possibly shorter…)

Such learning is very powerful since we can now backtrack to any level (even the start) and still not repeat this search.

Zchaff: A fast SAT solver

Rel-sat scheme:

rel-sat recurses until only the conflict variable and lower level variables remain. (last UIP cat below)



this adds: $\left(v_{11}' + v_6 + v_{13} + v_4' + v_8' + v_{17} + v_{19}'\right)$

note: when we backtrack v_{11} $\Rightarrow$ this clause is unit $\Rightarrow$
forces v_{11} into another subspace

Zchaff: A fast SAT solver

One could add all these decision nodes on any subspace of a cut, … (too small a subspace)

It seems clear that we should exploit the reconvergence at UIP's to cover larger subspace, however there are lots of UIP's.

zChaff: (1-UIP scheme) $\Rightarrow$ make the conflict as close to the conflict as possible (i.e., make it relevant) then remove it later to some space.

i.e., put all variables assigned after first UIP (that one closest of conflict) with paths to conflict on the conflict side of the cut, everything also on other side.

(i.e., first UIP cut above)

	9vliw_bp_mc (unsat, 19148 vars)			longmult10 (unsat, 4852 vars)			sat/bw_large.d.cnf (sat, 6325 vars)		
	1UIP	rel_sat	GRASP	1UIP	rel_sat	GRASP	1UIP	rel_sat	GRASP
Branches(x10e6)	1.91	4.71	1.07	0.13	0.19	0.12	0.019	0.046	0.027
Added Clauses(x10e6)	0.18	0.76	1.06	0.11	0.17	0.36	0.011	0.030	0.071
Added Literals (x10e6)	73.13	292.08	622.96	17.16	28.07	77.47	0.56	1.61	6.12
Added lits/cls	405.08	384.31	587.41	150.17	162.96	213.43	50.87	52.94	86.19
Num. Implications(x10e6)	69.85	266.21	78.49	71.72	108.00	74.83	7.23	17.84	16.81
Runtime	522.26	1996.73	2173.14	339.99	612.64	762.94	23.44	52.64	124.13