Advanced Topics in SAT-Solving
Part III: Implementation Techniques

Carsten Sinz

Wilhelm-Schickard-Institute for Computer Science
University of Tübingen

29.09.2004

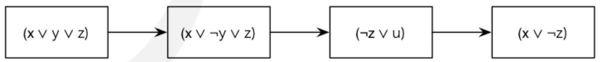
Outline

1. Basic Data Structures
2. Efficient Unit Propagation
3. Literal Selection Strategies
4. Clause Learning
5. Parallelization

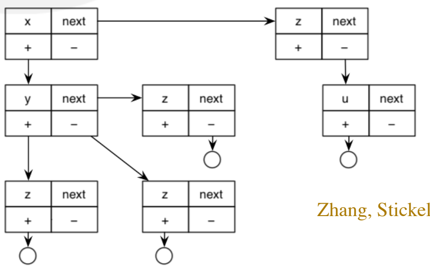
Data Structures for CNF Representation

How to represent a set of clauses?

A) As a clause list



B) As a trie data structure



Zhang, Stickel (1994)

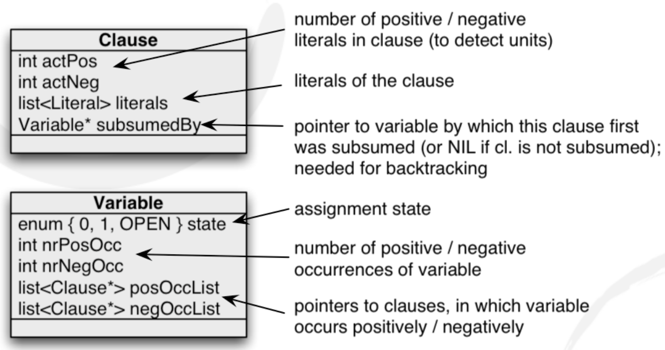
Repetition: DPLL Algorithm

```
boolean DPLL(ClauseSet S)
{
   while (S contains a unit clause {L}) {            // unit propagation
        delete from S all clauses containing L;      // u. subsumption
        delete ¬L from all clauses in S;             // u. resolution
   }
   if (∅ ∈ S) return false;
   if (S = ∅) return true;
   choose a literal L occurring in S;
   if (DP(S ∪ {{L}}) return true;
   else return DP(S ∪ {{¬L}});
}
```

Data Structures for DPLL: Requirements

- Allow for fast unit propagation
  - Detection of new units
  - Propagation of units
- Support back-tracking (restoration of clause data)
  Implementation alternatives:
  - Save copy of clause set data structure on each level
  - Remember changes (undo-stack)
  Goal: Minimize restore effort
- Compact representation of large clause sets

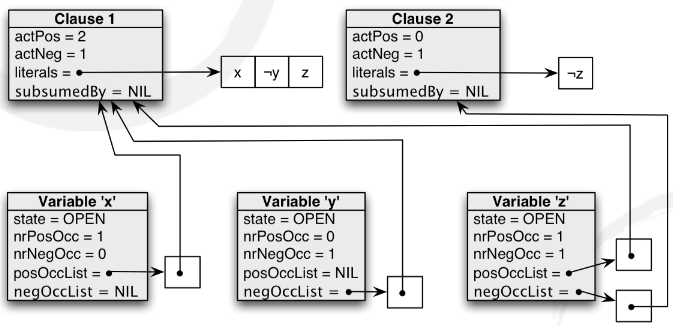
"Traditional" Approach



Crawford, Auton (1993)

Traditional Approach: Example

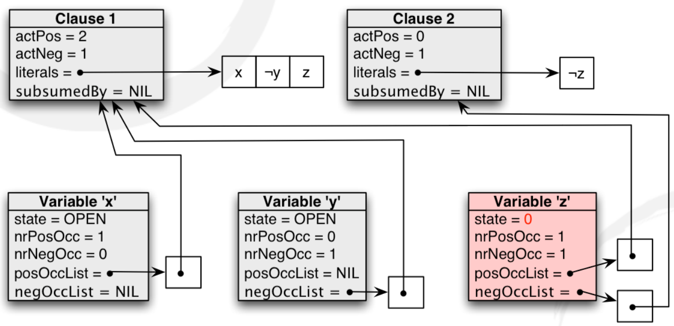
$$F = \{\{x, \neg y, z\}, \{\neg z\}\}$$



Example: Unit Propagation

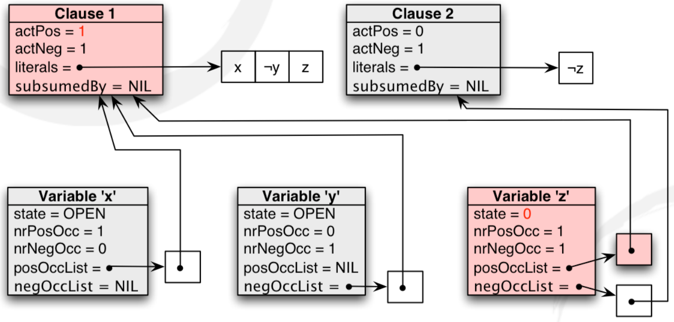
$$F = \{\{x, \neg y, z\}, \{\neg z\}\}$$

Unit propagation: set z=0



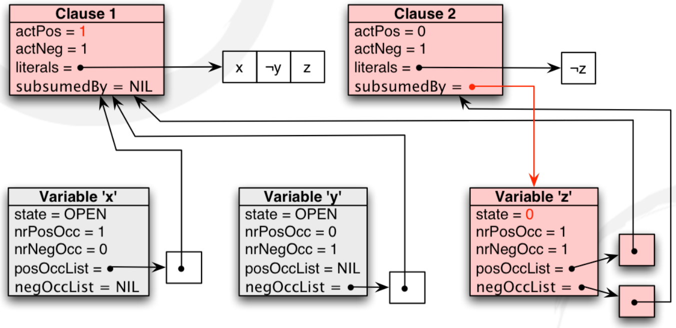
Example: Unit Propagation

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$ Unit propagation: set z=0



Example: Unit Propagation

$F = \{\{x, \neg y, z\}, \{\neg z\}\}$ Unit propagation: set z=0



Algorithm Unit-Propagation

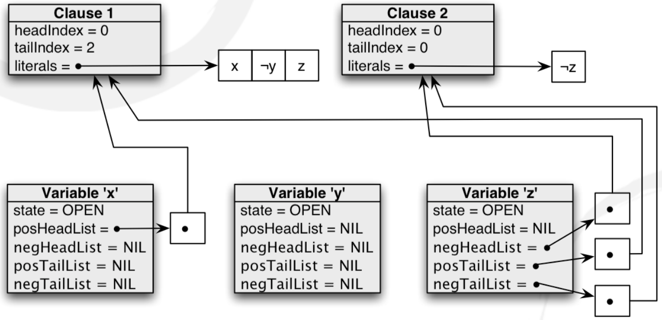
```
boolean UnitProp(Literal L)            // L: open literal; 'UnitProp' returns
{   if(L.isPositive()) {               // false on contradiction
        v = L.var(); v.state = 1;
        for(it = v.posOccList.begin(); it != v.posOccList.end(); it++) {
            clause = *it;
            if(clause.subsumedBy == NIL) clause.subsumedBy = v;
        }
        for(it = v.negOccList.begin(); it != v.negOccList.end(); it++) {
            clause = *it;  if(clause.subsumedBy == NIL) {
             clause.actPos--;          // shorten clause
             if(v.actPos + v.actNeg == 1) {   // new unit clause detected
                ok = HandleNewUnit(clause);
                if(!ok) return false;         // conflicting units?
    }   }   } }
    else {...}
    return true;
}
```

UP Alg.: Complexity and Improvements

- Setting variable x (to true) ...
  - subsumes |posOccList(x)| clauses
  - shortens |negOccList(x)| clauses
  - thus: requires a total of #occ(x) clause modifications
- Can we improve on this?
  - We only have to detect unit clauses
  - Idea (Zhang, Stickel (1996)):
    1. **Delay testing for subsumption**
       'subsumedBy' is not used any more; instead, the *test for a new unit* has to check whether clause is subsumed
    2. **Restrict unit resolution to first and last open literal in clause**
       maintain pos/negHeadList and pos/negTailList instead of pos/negOccList

Head and Tail Lists: Example

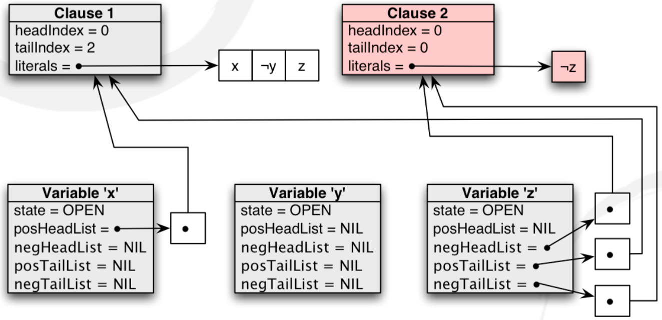
$$F = \{\{x, \neg y, z\}, \{\neg z\}\}$$



Head and Tail Lists: Unit Propagation

$$F = \{\{x, \neg y, z\}, \{\neg z\}\}$$

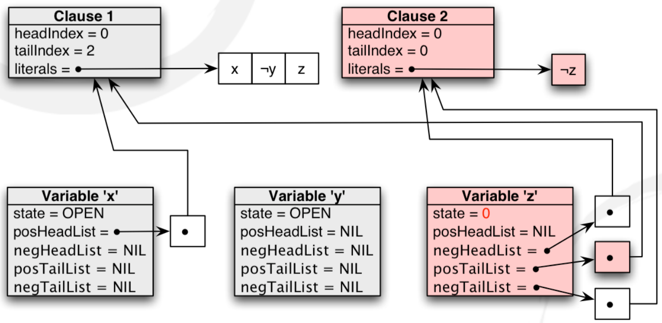
Unit propagation: set z=0



Head and Tail Lists: Unit Propagation

$$F = \{\{x, \neg y, z\}, \{\neg z\}\}$$

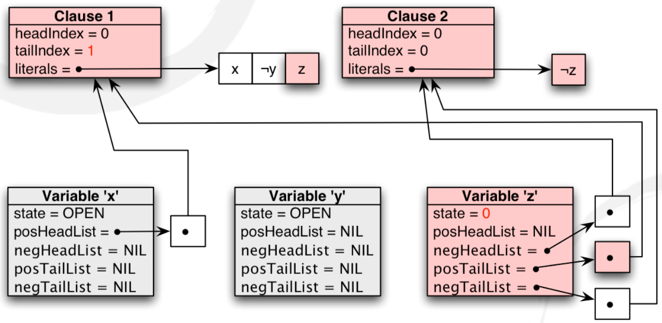
Unit propagation: set z=0



Head and Tail Lists: Unit Propagation

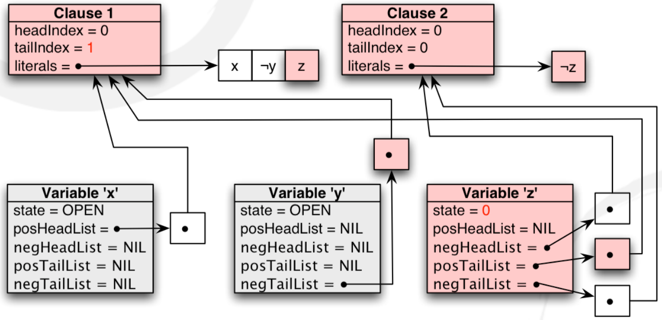
$$F = \{\{x, \neg y, z\}, \{\neg z\}\}$$

Unit propagation: set z=0



Head and Tail Lists: Unit Propagation

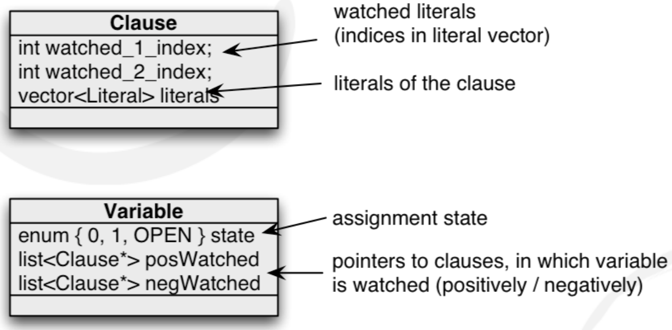
$F = \{\{x, \neg y, z\}, \{\neg z\}\}$ Unit propagation: set z=0



H/T Lists: Pros, Cons, Improvements

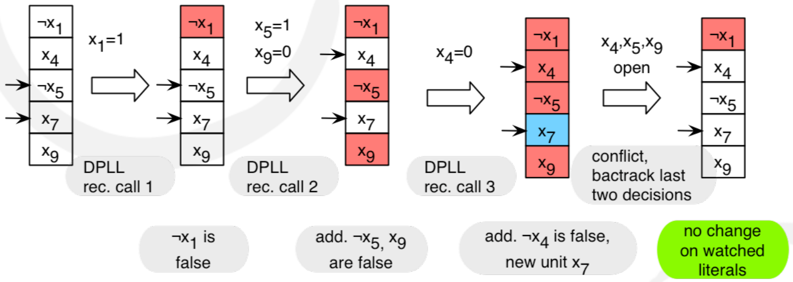
- Positive: Faster unit propagation
- Negative: Backtracking becomes more complicated (head and tail lists have to be restored)
- Further improvement: watched literals
  - Instead of head/tail literals: 2 watched literals per clause
  - Watched literals point to *arbitrary* open (different) literals
  - on backtracking: no update of data structure needed
  - First implemented in **chaff** (Moskewicz *et al.*, 2001)

Watched Literals: Data Structures



Moskewicz *et al.* (2001)

Watched Literals: Example



Literal Selection Strategies

- Which literal to select best in case-distinction step?
  - Size of search space (and thus run-time) can drastically depend on literal selection heuristics
  - Highly problem-dependent, no general "best" strategy
- Ideas for selection heuristics:
  1. Maximal simplification, e.g. maximize number of subsumed (deleted) clauses
  2. Try to reach tractable subclass of SAT, e.g. 2-SAT, Horn-SAT, only positive clauses
  3. Based on conflict analysis / clause learning (with preference for literals in recently learned clauses)

Literal Selection Strategies

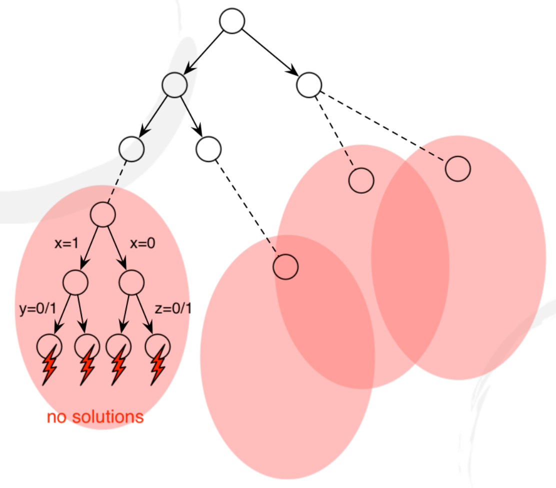
- **MOM** (maximum occurrences in minimal clauses): maximize $(occ_2(l) + occ_2(\neg l)) \cdot 2^{\alpha} + occ_2(l) \cdot occ_2(\neg l)$
  (where α is a 'large enough' number)
- **SATO**: build test set of k shortest positive clauses and choose literal that maximizes $(po(l)+1)(no(l)+1)$
  (where $po(l)$ ($no(l)$) denotes number of positive (negative) occurrences of literal l)
- **VSIDS** (variable state independent decaying sum): initial *score* is number of literal occurrences; for each learned clause, increase score by constant c for all literals in clause; periodically divide all scores by a factor f; choose literal with highest score

Conflict Analysis & Clause Learning

- Try to avoid repeated search of parts of the search tree with no solutions
- Compensate for badly selected case distinction literals
- Method: find weakest assumption under which a contradiction arises
  - Each selected branching literal counts as an 'atomic' reason
  - Find minimal necessary condition (i.e. minimal literal set) that produces the same conflict
- Also called "no-good learning" in the CSP community

Conflict Analysis: Example

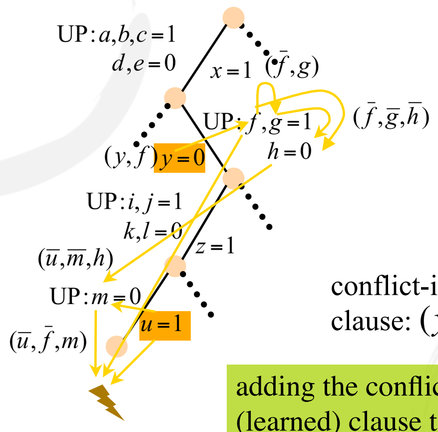
{¬x,y},
{¬x,¬y},
{x,¬z}.
{x,z}



Lemma Generation

(Marques-Silva, Sakallah, 1996)

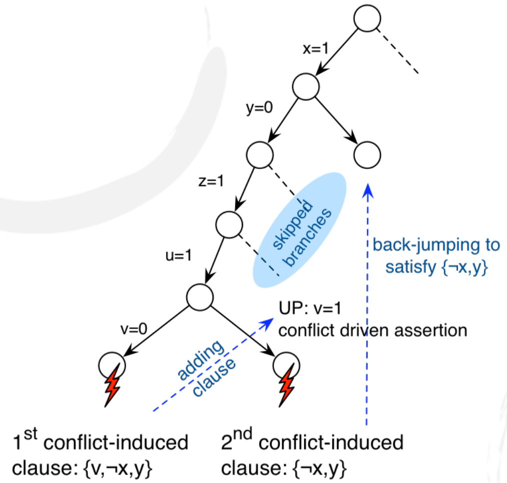
$(\bar{u},\bar{f},m)$
$(\bar{u},\bar{m},h)$
$(\bar{f},\bar{g},\bar{h})$
$(y,\bar{f})$
$(\bar{f},g)$
$\vdots$



conflict-induced clause: $(y,\bar{u})$

adding the conflict-induced (learned) clause to the instance avoids repeated search

Non-Chronological Back-Jumping



Parallelization

- Allow several processors work collaboratively on the same SAT instance
- Questions to answer:
  - How to partition search space between processors?
    - Once at the beginning or on demand during search?
    - How to deal with unreliable communication / network failure / shtudown of computers
  - Exchange learned clauses between processes?
  - Effects of combining clause learning and parallelization
- Experimental results:
  - Good speed-upds attainable on n processors (n≈8-32)
  - Parallel learning and clause exchange highly problem dependent

Dynamic Search Space Splitting

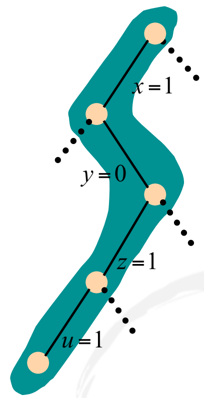
- Guiding path (H. Zhang *et al.* 1996) describes state of search, e.g.

  $((x,\mathrm{B}),(\bar{y},\mathrm{N}),(z,\mathrm{B}),(u,\mathrm{B}))$

- Partitioning of search-space at each (_,B) entry possible, e.g.

  $((x,\mathrm{N}),(\bar{y},\mathrm{N}),(z,\mathrm{B}),(u,\mathrm{B}))$

  $((\bar{x},\mathrm{N}))$



Combining Learning and Parallelization

- Acceleration by lemma generation may limit speed-ups attainable by parallelization: