A Pearl on SAT Solving in Prolog

Jacob Howe and Andy King

Funded by EPSRC grants EP/E033106 and EP/E034519
and a Royal Society Industrial Fellowship

FLOPS'10, Sendai, 21st April

CITY UNIVERSITY LONDON

Introduction

- SAT solving: DPLL with watched literals
- Stability tests in fixpoint calculations
- A solver exploiting delay in Prolog
- Some quick experiments
- Discussion

CITY UNIVERSITY LONDON

The DPLL algorithm

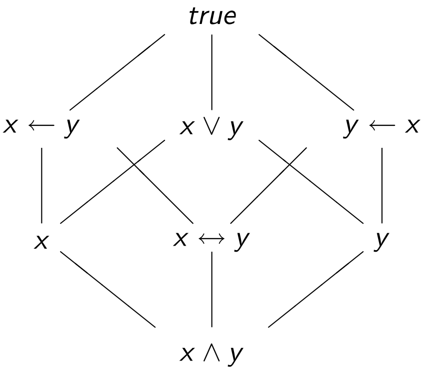
```
function DPLL(f: CNF formula, θ : truth assignment)
begin
    θ1 := θ ∪ unit-propagation(f, θ);
    if (is-satisfied(f, θ1)) then
        return θ1;
    else if (is-conflicting(f, θ1)) then
        return ⊥;
    endif
    x := choose-free-variable(f, θ1);
    θ2 := DPLL(f, θ1 ∪ {x ↦ true});
    if (θ2 ≠ ⊥) then
        return θ2;
    else
        return DPLL(f, θ1 ∪ {x ↦ false});
    endif
end
```

Unit propagation with Watched Literals

- Where the variables are $\{u, v, w, x, y, z\}$, consider:
  $\neg x \vee z, u \vee \neg v \vee w, \neg w \vee y \vee \neg z$
- With the partial assignment $\theta = \{x \mapsto \mathit{true}\}$ this becomes:
  $\mathit{false} \vee z, u \vee \neg v \vee w, \neg w \vee y \vee \neg z$
- For the first clause to be satisfied, the only unassigned variable z must be assigned to *true*, and θ is extended with this, becoming $\theta' = \{x \mapsto \mathit{true}, z \mapsto \mathit{true}\}$
  $\mathit{false} \vee \mathit{true}, u \vee \neg v \vee w, \neg w \vee y \vee \mathit{false}$
- It is only necessary to monitor two unassigned variables in a clause.
- With θ' extended to $\theta'' = \{x \mapsto \mathit{true}, z \mapsto \mathit{true}, y \mapsto \mathit{false}\}$, unit propagation leads to w being assigned to *false*, but the second clause does not react to this as w is not monitored
  $\mathit{false} \vee \mathit{true}, u \vee \neg v \vee \mathit{false}, \mathit{true} \vee \mathit{false} \vee \mathit{false}$

CITY UNIVERSITY LONDON

Background: a Pos-based groundness analyser



$Pos_{\{x,y\}}$

- Stability in a fixpoint calculation might be checked by testing whether $f_i \models f_{i+1}$ and $f_{i+1} \models f_i$.
- These entailments become SAT problems, for example $(x \rightarrow y) \models x \vee y$ becomes cnf problem $\neg x \vee y, \neg x, \neg y$.
- This has satisfying assignment $\{x \mapsto 0, y \mapsto 0\}$, indicating that the entailment does not hold.
- Whereas, $x \models x \leftarrow y$ becomes $x, y, \neg x$. This does not have a satisfying assignment, hence the entailment holds.

CITY UNIVERSITY LONDON

Delay in Prolog

- Logic Programming = Logic+ Control
- Delay is a fundamental aspect of Control
- It is used to suspend execution until arguments are appropraitely instantiated:

```
:- block merge(-,?,-), merge(?,-,-).
merge([], Y, Y).
merge(X, [], X).
merge([H|X], [E|Y], [H|Z]) :- H @< E,  merge(X, [E|Y], Z).
merge([H|X], [E|Y], [E|Z]) :- H @>= E, merge([H|X], Y, Z).
```

- Delays solve the control generation problem: it is always possible to introduce delays into clauses so as to induce a terminating control strategy.
- That is, by adding control (delays) to clauses, the logical specification of an algorithm can be implemented.

Code (SICStus)

```
sat(Clauses, Vars) :-
      problem_setup(Clauses), elim_var(Vars).

elim_var([]).
elim_var([Var | Vars]) :-
      elim_var(Vars), (Var = true; Var = false).

problem_setup([]).
problem_setup([Clause | Clauses]) :-
      clause_setup(Clause),
      problem_setup(Clauses).

clause_setup([Pol-Var | Pairs]) :-
      set_watch(Pairs, Var, Pol).
```

Code (SICStus)

```
set_watch([], Var, Pol) :- Var = Pol.
set_watch([Pol2-Var2 | Pairs], Var1, Pol1):-
        watch(Var1, Pol1, Var2, Pol2, Pairs).

:- block watch(-, ?, -, ?, ?).
watch(Var1, Pol1, Var2, Pol2, Pairs) :-
      nonvar(Var1) ->
            update_watch(Var1, Pol1, Var2, Pol2, Pairs);
            update_watch(Var2, Pol2, Var1, Pol1, Pairs).

update_watch(Var1, Pol1, Var2, Pol2, Pairs) :-
      Var1 == Pol1 -> true; set_watch(Pairs, Var2, Pol2).
```

CITY UNIVERSITY LONDON

Example

$z \quad x \quad y \quad u \quad v$

$$\neg x \vee \neg y \vee z$$
$$\neg x \vee \neg z \vee \neg u$$
$$\neg y \vee \neg z \vee u$$
$$\neg x \vee \neg v \vee z$$
$$\neg x \vee y \vee u$$
$$y \vee v \vee z$$

```
Block: sat_engine:watch(_X,false,_Y,false,[true-_Z])
Block: sat_engine:watch(_X,false,_Z,false,[false-_U])
```

Example

$Z \mapsto true$

```
Unblock: sat_engine:watch(_X,false,true,false,[false-_U])
Block: sat_engine:watch(_X,false,_U,false,[])
```

$X \mapsto true$

```
Unblock: sat_engine:watch(true,false,_Y,false,[true-true])

Unblock: sat_engine:watch(true,false,_U,false,[])
```

$U \mapsto false$, results in failure
$X \mapsto false$

```
Unblock: sat_engine:watch(false,false,_Y,false,[true-true]
Unblock: sat_engine:watch(false,false,_U,false,[])
```

etc...

CITY UNIVERSITY LONDON

Delay in other Prolog systems

SWI: `when`

```
set_watch([], Var, Pol) :- Var = Pol.
set_watch([Pol2-Var2 | Pairs], Var1, Pol1):-
        when(;(nonvar(Var1),nonvar(Var2)),
                  watch(Var1, Pol1, Var2, Pol2, Pairs)).

watch(Var1, Pol1, Var2, Pol2, Pairs) :- ...
```

SWI (plus...): `freeze`

```
set_watch([], Var, Pol) :- Var = Pol.
set_watch([Pol2-Var2 | Pairs], Var1, Pol1):-
        freeze(Var1,V=u), freeze(Var2,V=u),
        freeze(V, watch(Var1,Pol1,Var2,Pol2,Pairs)).

watch(Var1, Pol1, Var2, Pol2, Pairs):- ...
```

CITY UNIVERSITY LONDON

Extensions

- Static variable ordering: order variables by frequency of occurrence in the problem. This wins in two ways: the problem size is quickly reduced by satisfying clauses and the amount of propagation achieved is greater.
- Preprocessing with resolution: a popular tactic is to change the problem by restructuring it using limited applications of resolution steps.
- Backjumping: allows the solver to avoid exploring fruitless branches of the search tree.
- Dynamic variables ordering: reorder variables during search. Reordering can be implemented using similar tactics to backjumping, but a good implementation also needs learning...

Experiments

benchmark	*vars*	*clauses*	*sat*	*sics*	*mini*	*assigns*
chat_80_1.cnf	13	31	true	0	1	9
chat_80_2.cnf	12	30	true	0	1	5
uf20-0903.cnf	20	91	true	0	1	8
uf50-0429.cnf	50	218	true	10	1	89
uf100-0658.cnf	100	430	true	20	1	176
uf150-046.cnf	150	645	true	290	15	3002
uf250-091.cnf	250	1065	true	2850	171	13920
uuf50-0168.cnf	50	218	false	0	1	79
uuf100-0592.cnf	100	430	false	50	6	535
uuf150-089.cnf	150	645	false	770	18	8394
uuf250-016.cnf	250	1065	false	t/o	1970	
2bitcomp_5.cnf	125	310	true	130	1	7617
flat200-90.cnf	600	2237	true	380	12	1811

CITY UNIVERSITY LONDON

Limitations

- Large problems: the programmer does not have the fine-grained memory control required to store and access hundreds of thousands of clauses.
- Learning: clauses are added to the problem that express regions of the search space that do not contain a solution. Unfortunately, it is not clear how to achieve this cleanly in this Prolog solver, as calls to the learnt clauses would be lost on backtracking.

Conclusions

- A SAT solver can be cleanly and simply implemented in Prolog using logic: variables and assignment; and control: unit propagation with watched literal.
- However, the solver will struggle with large problems, owing to the lack of fine grained memory control required.
- The solver presented provides an easy entry to SAT solving, and is useful for small to medium sized SAT instances.