Introduction to Artificial Intelligence (ARIN)
SAT and the DPLL algorithm

James Cussens
University of York, UK

Date : 2012/02/0511 : 01 : 25

Why SAT matters for AI

- SAT is NP-complete; it's a hard problem.
- Many other hard problems can be converted to SAT.
- AI is all about solving hard problems.
- Otherwise there'd be no 'intelligence', right?

Satisfying a clause

- A *literal* is a propositional symbol or the negation of one.
- For a clause to be true in a model it's enough for one of the literals to be true in that model.

This clause:

$$A \vee \neg B \vee C$$

is satisfied if at least one of these is the case:

1. $A = \text{true}$
2. $B = \text{false}$
3. $C = \text{true}$

- Think of long clauses as 'loose' constraints

‘Breaking’ clauses

- For a clause to be false in a model all literals must be false in that model.
- We say the model *breaks* the clause.

This clause:

$$A \vee \neg B \vee C$$

is broken if (and only if) all of these are the case:

1. $A = \text{false}$
2. $B = \text{true}$
3. $C = \text{false}$

When a model does not satisfy a CNF formula

- Recall that CNF = Conjunctive Normal Form
- A CNF formula is a conjunction of clauses
- Write 'a CNF' instead of 'a formula in CNF' in the interests of brevity.
- Since it's a conjunction, if a model breaks even one clause it fails to satisfy the CNF.
- If all clauses are satisfied then so is the CNF.

Determining satisfiability

- With n propositional symbols there are 2^n models.
- We can determine whether a CNF is satisfiable by enumerating all models.
- If we come across a satisfying model then answer YES.
- Otherwise (after checking all 2^n models) answer NO.

A very naive SAT solver - part 1

```
def dpll_satisfiable(sentence):
    clauses = convert_to_cnf(sentence)
    symbols = symbols_in(sentence)
    return dpll_really_naive(clauses,symbols,{})
```

A very naive SAT solver - part 2

```
def dpll_really_naive(clauses,symbols,model):
    if is_empty(symbols):
        for clause in clauses:
            # truth_value must be either True or False
            # since all symbols assigned a truth value
            truth_value = get_truth_value(clause,model)
            if truth_value == False:
                return False
        return True

    p = first(symbols)
    rest = rest_of(symbols)
    return
        dpll_really_naive(clauses,rest,model+'p=True')
        or
        dpll_really_naive(clauses,rest,model+'p=False')
```

Early termination

- Surely we can do better than this!
- Insight We typically don't need a fully defined model to decide whether a clause is satisfied.
- Ditto whether a clause is broken.
- If $A = \text{true}$ then $(A \vee \neg B \vee C)$ is satisfied regardless of the truth-values of B and C.
- If $A = \text{false}, B = \text{true}, C = \text{false}$ then $(A \vee \neg B \vee C)$ is broken, regardless of the truth value of, say, D.

A naive, but not very naive, SAT solver

```
def dpll_naive(clauses,symbols,model):

    # can we determine sat/unsat already?
    all_true = True
    for clause in clauses:
        # truth_value = None if the (partially-built)
        # model does not determine truth value of clause
        truth_value = get_truth_value(clause,model)
        if truth_value == False:
            return False
        if truth_value == None:
            all_true = False
    if all_true:
        return True

    # recursive call as before ...
```

SAT solving as search

- Can view the SAT problem as a search for a satisfying model.
- The states are partially-defined models, i.e. truth assignments for some of the propositional symbols.
- We can move to a new state by assigning true/false to a variable.
- And can also backtrack to an earlier state.
- DPPL is depth-first search—with simple, but effective, heuristics.

Pure symbols

Consider:

1. $A \vee \neg B$
2. $\neg B \vee \neg C$
3. $A \vee C$

- A and B are *pure*, since they have the same 'sign' in all clauses. C is *impure*.
- Insight If a CNF has a model, then it has one with all pure symbols set to make their literals true.
- All clauses containing a given pure symbol will be satisfied . . .
- . . . and other clauses don't depend on it.
- So **fix** the truth-values of pure symbols.

Pure symbols: it gets better

1. $A \vee \neg B$
2. $\neg B \vee \neg C$
3. $A \vee C$

- If we have $B = \text{false}$ then $(\neg B \vee \neg C)$ is already true and C 'becomes' pure.
- In general, when looking for pure symbols we can ignore clauses already known to be true.

Looking first for pure symbols

```
# ... if we are not already done
p, value = find_pure_symbol(symbols,clauses,model)
if p is not None:
    return dpll_naive(clauses,symbols-p,model+'p=value')

# ... <sigh> no pure symbols so have to branch
#            on some variable
```

Unit clause heuristic

- A *unit clause* contains a single literal.
- If $B = \mathrm{true}$ then $(\neg B \vee \neg C)$ simplifies to $\neg C$ and so C must be set to false.
- In general, if all literals bar one are false in a (partially-built) model, then **fix** the last one to satisfy the clause.

Adding the unit clause heuristic

```
# ... if we are not already done
p, value = find_pure_symbol(symbols,clauses,model)
if p is not None:
    return dpll_naive(clauses,symbols-p,model+'p=value')
p, value = find_unit_clause(clauses,model)
if p is not None:
    return dpll_naive(clauses,symbols-p,model+'p=value')

# ... <sigh> have to resort to branching
#            on some variable
```

Unit propagation

- Forcing a variable to take a particular value may generate a 'cascade' of forced assignments.
- For example, suppose $C \vee A$ is one of our clauses. If C is set to false, then A must be set to true.
- This is called *unit propagation*.

Component analysis

- As we assign variables, satisfied clauses can be removed and literals can be removed from yet-to-be-satisfied clauses. Which literals?
- The resulting CNF may end up being representable as $X \wedge Y$ where X and Y are both CNFs with no overlapping variables.
- X and Y are then *components* which can be worked on separately.

Variable and value ordering

- Which variable to try next? Which value to try first?
- *Degree heuristic* chooses the variable which appears most frequently over the remaining clauses.

Other tricks

- Intelligent backtracking (as opposed to chronological backtracking).
- Clause learning.
- Random restarts.
- Good programming!