Theorem Proving in Propositional Logic

Hans de Nivelle

Summary

I explain some of the modern techniques for theorem proving in propositional logic. (SAT-solving)

Satisfiability testing for propositional formulas is NP-complete. Therefore, it is unlikely that polynomial algorithms exist.

Nevertheless, much progress has been made in recent years, and modern SAT-solvers are able to solve problems that are large enough to be useful in industrial applications.

Definition: We assume a set of propositional symbols $\mathcal{P}$. We call the elements of $\mathcal{P}$ atoms.

A literal is an atom A or a negated atom $\neg A$. We will assume that $\neg\neg A = A$.

Definition: A clause is a finite set of literals

$$\{A_1, \ldots, A_p\}$$

.

The meaning of a clause $\{A_1, \ldots, A_n\}$ is the disjunction $A_1 \vee \cdots \vee A_n$.

The meaning of $\{\ \}$ is $\perp$.

An interpretation I a partial function from $\mathcal{P}$ to the set $\{\mathbf{false}, \mathbf{true}\}$.

The interpretation I is extended to literals as follows:

1. If $I(A) = \mathbf{true}$, then $I(\neg A) = \mathbf{false}$.
2. If $I(A) = \mathbf{false}$, then $I(\neg A) = \mathbf{true}$.

The interpretation I is extended to clauses as follows:

1. If C contains a literal A, for which $I(A) = \mathbf{true}$, then $I(C) = \mathbf{true}$.
2. If for all literals A in C, $\quad I(A) = \mathbf{false}$, then $I(C) = \mathbf{false}$.

An interpretation I is a model of a set of clauses S if it is a model of every $C \in S$.

Backtracking: A Simple Algorithm

bool SAT(S, I) either returns **true** and extends I to a complete model, or it returns **false**.

```
bool SAT( const clauseset& S, interpretation& I )
{
   if there is a clause C in S, s.t.  I(S) = false,
      then return false;
   if for all clauses C in S, I(C) = true,
      then return true;
         // I is now a model for C.
```

```
    A = an atom that occurs in S,  in a clause C for
                   which I(C) is undefined;

    I(A) = true;
    bool b = SAT( S, I ); if(b) return true;

    I(A) = false;
    b = SAT( S, I );      if(b) return true;

    I. erase(A);    // Remove assignment for A.
    return false;
}
```

Theorem If there is an $I' \supseteq I$, s.t. I' makes S true, then $\mathbf{SAT}(S, I)$ returns **true** and extends I into a model $I'' \supseteq I$ of S.

If there is no $I' \supseteq I$, s.t I' makes S true, then $\mathbf{SAT}(S, I)$ returns **false**.

Possible Improvements of Backtracking

1. Deducing is better than guessing. Deduce as many consequences as possible, and guess only when nothing more can be deduced.
2. When assigning $I(A) = \mathbf{true}$ has failed, try $I(A) = \mathbf{false}$ only in case the assignment $I(A) = \mathbf{true}$ was part of the reason of the failure. (This is called relevant backtracking, or conflict analysis)
3. Select an A that is likely to cause a lot of forward reasoning, or a conflict.

Deducing Consequences

In case S contains a clause of form $R \cup \{A\}$, for which $I(R) =$ **false**, then assign $I(A) =$ **true**.

If S contains a clause of form $R \cup \{\neg A\}$, with $I(R) =$ **false**, then assign $I(A) =$ **false**.

Example

Consider:

$\{P_1, Q_1, A\}$,

$\{P_1, \neg Q_1\}$,

$\{\neg P_1, Q_1\}$,

$\{\neg P_1, \neg Q_1\}$,

$\{P_2, Q_2, \neg A, \}$,

$\{P_2, \neg Q_2\}$,

$\{\neg P_2, Q_2\}$,

$\{\neg P_2, \neg Q_2\}$.

In the interpretation defined by $I(P_1) = \textbf{false}$, one can deduce $I(Q_1) = \textbf{false}$. From this follows $I(A) = \textbf{true}$.

Backtracking with Conflict Analysis and Forward Reasoning

We use a datastructure `used& U`, that assigns to each atom A a value from **false**, **true**. The value $U(A) = $ **true** means that A contributed to a conflict. Initially $U(A) = $ **false** for all A.

```
bool SAT( const clauseset& S, interpretation& I,
          used& U )
{
   if for all clauses C in S,  I(C) = true,
      then return true;
```

```
if there is a clause C in S, s.t. I(S) = false,
then
{
   for each literal L in C, do
      U( |L| ) = true;
         // |L| denotes the atom part of L.
         // Each atom |L| of C contributed to the
         // conflict.

   return false;
}
```

```
if there is a clause of form {A} | R in S,
   s.t. I(R) = false and I(A) is undefined
then
{
   I(A) = true;
   bool b = SAT( S, I, U );  if(b) then return true;
   I. erase(A);
   if( U(A) )
   {
      U(A) = false;
      for each literal L in R do
         U( |L| ) = true;
   }
   return false;
}
```

```
if there is a clause of form {-A} | R in S,
   s.t. I(R) = false and I(A) is undefined
then
{
   I(A) = false;
   bool b = SAT(S,I,U);  if(b) then return true;
   I. erase(A);
   if( U( -A ))
   {
      U( -A ) = false;
      for each literal L in R do
         U( |L| ) = true;
   }
   return false;
}
```

```
    A = select( I, S );
            // Select atom in S, occurring in a clause
            // C for which I(C) is not defined.

    I(A) = true;
    bool b = SAT(S,I,U);  if(b) return b;

    if( U(A) )
    {
       I(A) = false;
       b = SAT(S,I,U);   if(b) return b;
       U(A) = false;
    }
    I. erase(A);
    return false;
}
```

Backtracking with Conflict Analysis and Forward Reasoning

- The previous algorithm is called Davis-Putnam-Loveland-Logemann algorithm. (DPLL-algorithm).
- Realistic implementations do not use recursion, but an explicit stack.
- Marking avoids lots of unnecessary backtracking. But it is still easy to fool the algorithm into marking too much.

Semantics of Marking

What is the semantics of the structure U? What does it mean when an atom A is marked?

Learning of Conflict Clauses (1)

The meaning is that I, restricted to the marked atoms, cannot be extended to an interpretation for S.

This also can be encoded in a clause as follows:

At some state of the algorithm, assume that $A_1, \ldots, A_n$ are the atoms that are marked. Define a clause $\{L_1, \ldots, L_n\}$ as follows:

If $I(A_i) = \mathbf{true}$, then put $L_i = \neg A_i$,

if $I(A_i) = \mathbf{false}$, then put $L_i = A_i$.

The clause $\{L_1, \ldots, L_n\}$ is called a conflict clause.

Theorem: At each state of the search algorithm, the conflict clause is a logical consequence of S.

Learning of Conflict Clauses (2)

The DPLL-algorithm can be modified, so that it generates the conflict clauses directly. The advantages are:

1. Explicit conflict clauses provide a more accurate conflict analysis than marking.
2. Contrary to markings, conflict clauses can be kept and reused.
3. Using conflict clauses, the DPLL-algorithm is able to output proofs, (instead of only saying 'unsatisfiable')

Backtracking with Conflict Clauses

Function **bool SAT**(S, I) either

- returns **true**, and extends I into a model for S.
- returns **false**, and extends S with a clause C, s.t. $I(C) = \textbf{false}$. In that case I is not changed. The new clause C is a logical consequence of S.

DPLL with Conflict Analysis

```
bool SAT( clauseset& S, interpretation& I )
{

   if for all clauses C in S,  I(C) = true,
      then return true;

   if there is a clause C in S, s.t. I(S) = false,
   then
      return false;
```

```
if there is a clause of form {A} | R in S,
   s.t. I(R) = false and I(A) is undefined,
then
{
   I(A) = true;
   bool b = SAT(S,I);    if(b) return true;

   clause C = a clause of S for which I(C) = false.
   if( -A  in C ) then
      S. insert( resolvent( A, {A} | R, C );
   I. erase(A);
   return false;
}
```

```
if there is a clause of form {-A} | R in S,
   s.t. I(R) = true and I(A) is undefined
then
{
   I(A) = false;
   bool b = SAT(S,I);  if(b) return true;

   clause C = a clause of S for which I(C) = false.
   if( A in C ) then
      S. insert( resolvent( A, C, {-A} | R );
   I. erase(A);
   return false;
}
```

```
A = select( I, S );
I(A) = true;
bool b = SAT(S,I);   if(b) return b;

clause C1 = a clause of S for which I(C1) = false.
if( -A in C1 ) then
{
   I(A) = false;
   bool b = SAT(S,I);   if(b) return b;

   clause C2 = a clause of S for which I(C2) = false.
   if( A in C2 ) then
      S. insert( resolvent( A, C2, C1 ));
}
I. erase(A);
return false;
```

```
}
```

And some remarks:

- In real provers, the algorithm is implemented, not by recursion, but with a real stack.
- Often, there exists more than one conflict at the same time, and they result in different conflict clauses. There exist different heuristics for choosing a conflict clause. One normally selects the one with the lowest backtracking level.

Two Applications

I discuss two types of applications of propositional theorem proving:

1. Application in theorem proving.
2. Applications in finite model finding.

There exist much more applications, for example reasoning about boolean circuits, reasoning about programs with a finite number of variables, model checking.

Theorem Proving through Propositional Logic

A set of predicate clauses S is unsatisfiable iff there exists a set of propositional instances $S_1, \ldots, S_n$ of S, s.t. $S_1, \ldots, S_n$ is propositionally unsatisfiable.

1. Generate some set S_{prop} of propositional instances.
2. Check satisfiability of S_{prop}. If no, then C is unsatisfiable.
3. If yes, add some more instances to S_{prop} and goto 2.

Theorem Proving through Propositional Logic (2)

Let S be a set of clauses. $T_1, \ldots, T_n$ be a set of background theories (for example about equality, transitivity, integers, fact that all instances are needed)

1. Let $S_{prop} = \emptyset$.
2. Check satisfiability of S_{prop}. If no, then report S unsatisfiable.
3. If yes, let M_{prop} be a model of S_{prop}. For each T_i in the theories $T_1, \ldots, T_n$ do
   - (a) If M_{prop} is not a T_i-model of S, then add a propositional reason to S_{prop}.
   - (b) If nothing was added to M_{prop}, then report model M_{prop}. Otherwise goto 2.

Finding Finite Models through Propositional Logic

Definition A literal is flat if it has one of the following forms:

1. $P(x_1, \ldots, x_n)$ or $\neg P(x_1, \ldots, x_n)$ with $x_1, \ldots, x_n$ variables.
2. $f(x_1, \ldots, x_n) \approx y$ or $f(x_1, \ldots, x_n) \not\approx y$, with $x_1, \ldots, x_n, y$ variables.
3. $x \approx y$ with x, y variables.

A clause is flat when all its literals are flat.

The following procedure transforms C into a flat clause. Write $C = [A_1, \ldots, A_n]$.

1. If one of the A_i contains a functional term $f(t_1, \ldots, t_a)$ on a place where it should have contained a variable, then replace $[A_1, \ldots, A_n]$ by
$$[Y \not\approx f(t_1, \ldots, t_a),$$
$$A_1[f(t_1, \ldots, t_a) \Rightarrow Y], \ldots, A_n[f(t_1, \ldots, t_a) \Rightarrow Y]\ \ ].$$
2. If one of the A_i has form $X \not\approx Y$, then replace C by
$$[\ A_1[X := Y], \ldots, A_{i-1}[X := Y], A_{i+1}[X := Y], \ldots, A_n[X := Y]\ ].$$

Grounding a set of Flattened Clauses (1)

Let S be a set of flattened clauses.

Let $E = \{e_1, \ldots, e_n\}$ be a finite set of contants. For each function symbol f with arity a in S, add the following axiom to S.

- $[\ f(X_1, \ldots, X_a) \approx e_1, \ldots, f(X_1, \ldots, X_a) \approx e_n\ ].$

Next replace S by the set S_E of its instances within E.

- For each $C \in S$, with variables $X_1, \ldots, X_m$, for each sequence $d_1, \ldots, d_m$ of constants in D, add to S_E the instance $C[X_1 := e_1, \ldots, X_m := d_m]$.

Grounding a Set of Flattened Clauses (2)

- If $C \in S_E$ contains a negative equality of form $e \approx e$, then remove C from S_E.
- If $C \in S_E$ contains a negative equality of form $e \approx e'$ with $e \neq e'$, then remove this equality from C.

Believe it or not, this method is able to find some models. MACE is implemented in this way.

Summary

We have seen the DPLL algorithm with learning which uses backtracking, forward reasoning and learning of conflict clauses.

We have seen two applications.