Theorem Proving in Propositional Logic

Introduction

I explain modern techniques for testing satisfiability of propositional formulas.

Satisfiability testing for propositional formulas is NP-complete. It therefore unlikely that there exists a polynomial algorithm.

Nevertheless, much progress has been made in recent ($\geq$ 1995) years, and modern SAT-solvers are able to solve impressively large formulas. Big enough to be useful in industrial applications.

Basic Definitions

I give the basic definitions. Since we already covered predicate logic, there will be no surprises:

Definition: We assume a set of propositional variables V. We call the elements of V atoms.

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

Definition: A clause is a finite set of literals

$$\{A_1, \dots, A_n\}.$$

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

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

An interpretation I a partial function from V to the set $\{\mathbf{f}, \mathbf{t}\}$.

We define an operator $\overline{\cdot}$ on $\{\mathbf{f}, \mathbf{t}\}$, with $\overline{\mathbf{t}} = \mathbf{f}$, and $\overline{\mathbf{f}} = \mathbf{t}$.

The interpretation I is extended to literals by defining $I(\neg A) = \overline{I(A)}$.

The interpretation I is extended to clauses as follows:

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

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

Resolution

Definition: Let c_1 and c_2 be two clauses. Let A be an atom. Then the clause $(c_1 \backslash \{A\}) \cup (c_2 \backslash \{\neg A\})$ is a resolvent of c_1 and c_2.

We write $\mathrm{RES}(A, c_1, c_2)$ for the resolvent.

Soundness of Resolution

Theorem: Resolution is a sound reasoning rule:

For every interpretation I, if $I \models c_1$ and $I \models c_2$, and A is a literal, then $I \models \text{RES}(A, c_1, c_2)$.

proof. Case analysis. First deal with cases where $A \not\in c_1$ or $\neg A \not\in c_2$.

After that, treat the case where $A \in c_1$ and $\neg A \in c_2$. Distinguish the cases where $I(A) = \mathbf{t}$, $I(A) = \mathbf{f}$, and A is uninterpreted.

Completeness of Resolution

Theorem: Resolution is a complete reasoning rule. This means the following:

Let C be a set of clauses. If C is unsatisfiable, then it is possible, by using repeated resolution, to derive the empty clause from C.

Proof: It will follow from the completeness proof of the DPLL + CDCL algorithm.

It also follows from the model construction that we have seen already.

Although ordered resolution + selection is complete, it turns out not succesful in applications.

Moral: CS is an empirical science! CS is closer to physics than to mathematics!

Translation to CNF

Propositional formulas can be translated into sets of clauses, by using the same techniques as for predicate logic.

In particular, exponentional blow up can be prevented by introducing new symbols.

Backtracking: A First Algorithm

We want to find an interpretation for a set of clauses C.

start: Start with $I = \{\}$.

fail: If there is a false clause $c \in C$, and there are decisions, then backtrack to the last decision made at **decide**, reverse it, and goto **fail**. If there is no decision left, report that C is unsatisfiable.

solution: If all variables are assigned, then report the solution I.

decide: Find an unassigned variable v in C, and pick a $b \in \{\mathbf{f}, \mathbf{t}\}$. Add $v := b$ to I, remember the decision and goto **fail**.

This is called backtracking, because you walk back following the track that you made when you came.

Optimizations

If one explores I_1, I_2 after each other, knowledge obtained during checking of I_1 can be used when checking I_2.

In particular, one can check if $v := \mathbf{t}$ played a role in the failure. If not, there is no need to consider $v := \mathbf{f}$.

The choice of variable v matters a lot. Choose a v that has many occurrences in clauses with few unassigned variables.

DPLL (Davis Putnam Loveland Logeman)

Definition: Let I be an interpretation. Let c be a clause. Clause c is productive if it can be written in form $c' \cup \{\pm A\}$, s.t. A is unassigned, and c' is false in I.

If $\pm A$ is positive, we say that c produces the assignment $A := \mathbf{t}$. If $\pm A$ is negative, we say that c produces the assignment $A := \mathbf{f}$.

We write $v(I, c)$ for the variable in the produced assignment, and $t(I, c)$ for the truth value in the produced assignent, if they exist.

DPLL

start: Start with $I = \{\}$.

fail/produce: Do the following as long as possible:

- If there is a false clause $c \in C$, then invert the last choice made at **decide** if there is a choice left. Otherwise, report failure.
- If there is a productive clause $c \in C$, extend I by adding $v(I, c) := t(I, c)$.

solution: If all variables are assigned, then report the satisfying interpretation I.

decide: Pick an unassigned variable v, pick a truth value t, and assign $v := t$. Continue at **fail/produce**, remembering the decision.

Possible Improvements

1. As with simple backtracking, one should try to analyse which assignments contributed to the failure, and backtrack only those.

   This can be done by marking assignments in I, but very soon we will learn about CDLL, which is much better.

2. Select an (v, t) that is likely to cause a lot of productive clauses, or a conflict.

On the next slide, I give DPLL again, using a stack S.

fail/produce: As long as possible do: If there exists a false clause $c \in C$, then goto **backtrack**. If there exists a productive clause $c \in C$, extend I with $v(I,c) := t(I,c)$. Push **det**($v(I,c)$) to S.

solution/decide: If all atoms are assigned, then report the satisfying interpretation I.

If there are unassigned variables, then pick an unassigned variable v. Pick a $b \in \{\mathbf{f}, \mathbf{t}\}$. Add $v := b$ to I. Push **dec**(v) to S. Goto **fail/produce**.

backtrack: As long as the last element of S has form **det**(v), remove the assignment to v from I and remove **det**(v) from S.

If S is empty, then report that no satisfying interpretation exists. If the last element of S has form **dec**(v), then replace it by **det**(v). Let $b = I(v)$. Replace $I(v) := b$ by $I(v) := \overline{b}$. Goto **fail/produce**.

Example

Consider:

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

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

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

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

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

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

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

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

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

Non-Chronological Backtracking

In the interpretation I, assignments are marked when they are used. Assignments are initially without mark.

- If I makes a clause c false, then mark all variables in c as used.
- If clause c produces assignment $I(v) = b$, and this assignment is backtracked, then mark all remaining variables in c as used, when v was used.

When a decide is backtracked, the other assignment $\overline{b}$ is considered only when the assignment was used.

This is called non-chronological backtracking (also intelligent backtracking).

Non-chronological backtracking improves very much over simple (chronological) backtracking, but the introduction of CDLL (Conflict Driven Clause Learning) makes it totally obsolete.

Backtracking with Learning

Modify the algorithm, so that it also remembers the clause:

fail/produce: As long as possible do:

> If there exists a false clause $c \in C$, then goto **backtrack**.
>
> If there exists a productive clause $c \in C$, extend I by adding $v(I, c) := t(I, c)$. Push $\mathbf{det}(v(I, c), c)$ to S.

backtrack: Let z be a false clause, which is non-empty.

- If the last element of S has form $\mathbf{det}(v, c)$, then let $b = I(v)$, remove the assignment to v from I, and remove $\mathbf{det}(v, c)$ from S.

  If z is not false anymore in I, then replace z by

  $$\begin{array}{lll} \mathrm{RES}(A, c, z) & \text{if} & b = \mathbf{t}, \\ \mathrm{RES}(\neg A, c, z) & \text{if} & b = \mathbf{f}. \end{array}$$

  If z is empty, then report that no satisfying interpretation exists.

- If the last element of S has form $\mathbf{dec}(v)$, then remove $\mathbf{dec}(v)$ from S, and remove the assignment to v from I.

  If z is not false anymore, it has become productive. Goto **fail/produce**.

Learning

The derived clauses can be permanently added to C.

Usually, only the final, productive clause is learnt.

Learning improves the search algorithm very much, because it avoids that similar work is repeated on different branches (of the search tree).

Learnt clauses can be remembered forever, or forgotten when they are not used often enough, using a time out mechanism.

The learnt clause is a subset of the marked assignments.

Improvements

The deterministic part of the stack should be viewed as a graph, rather than a stack.

For example, if $I(A) = I(B) = \mathbf{t}$, then clauses $\neg A, \neg B, C$ and $\neg A, \neg B, D$ are productive. The order does not matter.

The resulting data structure is called dependency graph.

Unique Implication Points (UIP)

The current algorithm backtracks until it reaches a **dec**(v) for which v occurs in the conflict clause.

At this point, the conflict clause becomes productive.

Instead of backtracking down to the last relevant decision, one can stop at the first productive clause. (It can be found by looking at the dependency tree.)

To be precise: As soon as one has obtained a conflict clause that is productive in the interpretation I' that one obtains when the last choice is removed from I, one quits from **backtrack**, and goes to **fail/produce**.

The **(1)** does not necessarily flip the last choice, and **(2)** one can backtrack much deeper.

backtrack: Let z be a false clause. We assume that z is not empty.

- If S contains a decision, then let S' be the part of S up to (not including) the last decision **dec**(v).

  Let I' be obtained from I by restricting I to the variables in S'. (i.e. everything before the last decision.)

  Let z' be the subset of z that is still false in I'. We are sure that $\|z\backslash z'\| \geq 1$. If $\|z\backslash z'\| = 1$, we have reached a unique implication point, and we backtrack and learn:

  – As long as the variable v of the last element of S does not occur in z', remove it from S, and remove the assignment to v from I.

  Now z has become productive. Goto **fail/produce**, using z.

- The last element of S must have form **det**(v, c). Let $b = I(v)$, remove the assignment, and proceed as before $\cdots$

DPLL + CDCL

Some final remarks about the algorithm:

- In real provers, the stack is implemented as a graph, so that UIPs can be found more easily.
- In the dependency graph, one can try to attach a weight to clauses. In case more than one forward derivation of one literal is possible, one can keep the lightest. One can also try to keep all derivations.

The Two-Literal Watching Scheme

fail/produce requires look up of clauses that are false or productive. This can be costly, because there may be many clauses, and clauses can be big.

When variable v is assigned **t**, only clauses containing $\neg v$ need to be checked. When v is assigned **f**, only clauses containing v need to be checked. $\Rightarrow$ use an index with keys (variable, truthval).

One can do even better, because only the last and second last variable matter: Of each non-productive clause, store only two unassigned variables in the index.

If one of the variables receives a conflicting assignment, one can either find another unassigned variable, or the clause has become productive/false.

Solving Sudokus

		4		8				2
	2					9		
		6	1		2			3
	3					6		
	9		3	7	4	2		
	4		6	5	9	1		
	6					7		
		3		2				6
		7	9		1			5

Sudokus (2)

- Empty places have to be filled with digits $1 \leq i \leq 9$.
- No multiple occurrences of same digit in same row.
- No multiple occurrences of same digit in same column.
- No multiple occurrences of same digit in little 3×3-blocks.

Sudokus are designed in such a way, that they can be mostly solved by forward reasoning.

Let $V[i,j,k]$ denote: Position $[i,j]$ contains k.

The set of conditions can be formulated as SAT-problem:

Each field contains at least one digit: For $1 \leq i,j \leq 9$,

$$\{\ V[i,j,1],\ V[i,j,2],\ V[i,j,3],\ V[i,j,4],\ V[i,j,5],$$
$$V[i,j,6],\ V[i,j,7],\ V[i,j,8],\ V[i,j,9]\ \}.$$

No field contains two digits: For $1 \leq i,j \leq 9,\ \ 1 \leq k_1 < k_2 \leq 9$,

$$\{\neg V[i,j,k_1],\ \neg V[i,j,k_2]\ \}.$$

Each digit must occur at least once in each row:

For $1 \leq j \leq 9, \quad 1 \leq k \leq 9,$

$$\{\ V[1,j,k],\ V[2,j,k],\ V[3,j,k],\ V[4,j,k],\ V[5,j,k],$$
$$V[6,j,k],\ V[7,j,k],\ V[8,j,k],\ V[9,j,k]\ \}.$$

And not more than once:

For $1 \leq i_1 < i_2 \leq 9, \quad 1 \leq j \leq 9, \quad 1 \leq k \leq 9,$

$$\{\ \neg V[i_1,j,k],\ \neg V[i_2,j,k]\ \}.$$

(same for columns and the little blocks)

The given entries:

$$\{ \; V[2,2,2] \; \},$$

$$\{ \; V[2,4,3] \; \},$$

$$\{ \; V[2,5,9] \; \},$$

$$\{ \; V[2,6,4] \; \},$$

$$\{ \; V[2,7,6] \; \},$$

$$\ldots$$

$$\{ \; V[9,1,2] \; \},$$

$$\{ \; V[9,3,3] \; \},$$

$$\{ \; V[9,8,6] \; \},$$

$$\{ \; V[9,9,5] \; \}.$$

Summary

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

We have seen an exciting application.