Theorem Proving in Propositional Logic
Summary

I explain some of the modern techniques for theorem proving for 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.
Theorem Proving

All theorem proving strategies work by satisfiability testing. They work on a subset of possible formulas (usually clauses). Before proof search takes place, the formula (or set of formulas) has to be transformed into a set of clauses.
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 
\[ A_1 \lor \cdots \lor A_n. \]

The meaning of \( \{ \} \) is \( \bot \).

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

The interpretation \( I \) is extended to literals as follows:

1. If \( I(A) = \text{true} \), then \( I(\neg A) = \text{false} \).
2. If \( I(A) = \text{false} \), then \( I(\neg A) = \text{true} \).
The interpretation $I$ is extended to clauses as follows:

1. If $C$ contains a literal $A$, for which $I(A) = \text{true}$, then $I(C) = \text{true}$. 

2. If for all literals $A$ in $C$, $I(A) = \text{false}$, then $I(C) = \text{false}$. 

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 clauses of form $\{A\} \cup R_1$ and $\{\neg A\} \cup R_2$. The clause $C_1 \cup C_2$ is called **resolvent** of $C_1$ and $C_2$.

Alternatively, one could give the following definition:

Let $C_1$ and $C_2$ be two clauses. Let $A$ be an atom. Then $(C_1 \setminus \{A\}) \cup (C_1 \setminus \{\neg A\})$ is a resolvent of $C_1$ and $C_2$.

The definitions are not exactly the same, but they are the same in normal cases. These are cases where $A$ and $\neg A$ occur in the clauses, and not in $R_1, R_2$. 
Soundness of Resolution

**Theorem:** Resolution is a sound reasoning rule:

For every interpretation $I$, if $I \models C_1$ and $I \models C_2$, then $I \models (C_1 \setminus \{A\}) \cup (C_2 \setminus \{\neg A\})$.

**proof.** Case analysis. First deal with cases where $A \notin C_1$ or $\neg A \notin C_2$.

After that, treat the case where $A \in C_1$ and $\neg A \in C_2$. Distinguish the cases where $I(A) = t$, $I(A) = f$, and $A$ is uninterpreted.
Completeness of Resolution

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

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

**Proof:** It follows from the completeness proof of the DPLL algorithm, so I will not prove it now.
Remarks

Resolution is very important, because of its simplicity. By using unification and adding an equality rewriting rule, it can be adopted to predicate logic with equality.

Superposition, which is an adaptation of resolution to predicate logic, is the most successful technique that is currently known, for theorem proving in predicate logic.

I will explain later, how general formulas can be translated into sets of clauses. (For propositional logic, it is actually quite easy.)

There exist many restrictions (refinements) of resolution, that preserve completeness. I will show a few in class.
Backtracking: A Simple Algorithm

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

```cpp
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 occurring in S for which I(A) 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 $\text{SAT}(S, I)$ returns $\text{true}$ 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 $\text{SAT}(S, I)$ returns $\text{false}$. 
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 \( I(A) = \text{true} \) has failed, try \( I(A) = \text{false} \) only in case the assignment \( I(A) = \text{true} \) 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 $I(R) = \text{false}$, then assign $I(A) = \text{true}$.

If $S$ contains a clause of form $R \cup \{\neg A\}$, with $I(R) = \text{false}$, then assign $I(A) = \text{false}$.
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 \( I(P_1) = \text{false} \), one can deduce \( I(Q_1) = \text{false} \). From this follows \( I(A) = \text{true} \).
Backtracking with Conflict Analysis and Forward Reasoning

We define a datastructure $\text{used } U$, that assigns to each atom $A$ a value from $\text{false, true}$. The value $U(A) = \text{true}$ means that $A$ contributed to a conflict. Initially $U(A) = \text{false}$ for all $A$.

```c
bool SAT( const clauseset& S, interpretation& I, 
          used& U )
{
    if for all clauses $C$ in $S$, $I(C) = \text{true}$, 
    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\} \mid 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, for which
    // I(A) 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 $I(A_i) = \text{true}$, then put $L_i = \neg A_i$,

if $I(A_i) = \text{false}$, 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)

Modify the DPLL-algorithm, 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’)

Definition Let $A$ be an atom, let $c_1$ and $c_2$ be clauses. The resolvent of $c_1$ and $c_2$ on $A$ is defined by

$$\text{RESOLVENT}(A, c_1, c_2) = (c_1 \setminus \{A\}) \cup (c_2 \setminus \{\neg A\}).$$

Theorem:

$\text{RESOLVENT}(A, c_1, c_2)$ is a logical consequence of $c_1$ and $c_2$. 
DPLL with Learning

The modified algorithm uses the following datastructures:

- An assignment $I$, which is a partial function from propositional variables to \{false, true\}.
- The clause set $S$ that we are trying to refute.
- A stack $P$ of propositional variables, which keeps track of the order in which assignments were made.
- A dependency graph $f$, which is a partial function from propositional variables to clauses. If $A$ received an assignment by forward reasoning with clause $C$, then $f(A) = C$. If $A$ has no assignment, or received its assignment by guessing, then $f(A)$ is undefined.
bool SAT( clauseset& S, interpretation& I )
{
    start:
        if( there is a clause C in S, s.t. I(C) = false
        {
            C = UNWIND( C, I, S, P, f )
            Add C to the clause set S
            // Returns either empty clause, or clause
            // with which one can do forward reasoning.
            if( C == { } )
                return false; // Clause set is unsatisfiable.
            else
            {
                Let L be the unassigned literal in C.
                FORWARD( C, L ); goto start;
            }
        } }
Try Forward Reasoning:

if( there a clause C in S, that can be
    written in form R \cup \{ L \},
    s.t. I(R) = false, I(L) = undefined )

{
    FORWARD( C, L );
    goto start;
}

Guess a Truth Value for Some Atom

if( for all atoms A, I(A) is defined )
{
    print I;
    return true; // Found model.
}

Let A be an atom for which I(A) = undefined.
Select t from { false, true }
    // One could try to use a heuristic.

P. push_back(A);
I[P] = t;
goto start;
The Administration of Forward Reasoning

FORWARD( C, L )
{
    // It must be the case that I[L] is unknown,
    // that L in C, and I( C \ { L } ) = false.

    Let A = |L|;       // The atom of L.
    P. push_back(A);  // Push A on the stack:

    // Assign proper truth value to A:

    I[A] = ( L is positive ) ? true : false;
    f[A] = C;       // Justify A by C.
}

Unwind Returns to the last Choice Point

clause UNWIND( clause C, interpretation& I,
        clauseset& S, stack& P, const graph& f )
{
    // Invariant: I(C) = false:

    while( P.size() )
    {
        A = P.back(); P.pop_back();
        // A is last assigned propositional symbol.
if( f[A] is defined )
{
    // A was assigned by forward reasoning.

    if( A occurs in C )
    {
        C = RESOLVENT( A, C, f[A] )
    }
    else
    {
        if( -A occurs in C )
            C = RESOLVENT( A, f[A], C )
    }
    f. erase(A)
    I. erase(A)
}
else {
    {
        // A was a guess:
        I. erase(A);
        return C;
        // C will be forward reasoning clause.
    }
}

// There was no choice point. Return empty clause.
return { };
}
Some final remarks about the algorithm:

- 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.

- 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.
### Solving Sudokus

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
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 $3 \times 3$-blocks.

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

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

And not more than once:
For $1 \leq i_1 < i_2 \leq 9$, $1 \leq j \leq 9$, $1 \leq k \leq 9$,
\[
\{ \neg V[i_1, j, k], \neg V[i_2, j, k] \}.
\]
(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.