how to design a SAT solver, part 2

Spring 2009
### Recall Sudoku

- **No repeats on rows**
- **No repeats on columns**
- **No repeats within each 3x3 block**

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>5</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>1</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td></td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>2</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```
## Inference

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>4</td>
<td></td>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td></td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Cannot be 1, 2, 3, 8
- Cannot be 3, 4, 6, 1
- Cannot be 1, 3, 7, 8
Inference

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>2</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Cannot be 1, 2, 3, 8

Cannot be 3, 4, 6, 1

Cannot be 1, 3, 7, 8

Can only be 5
### Inference (contd.)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>5</th>
<th></th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>5</td>
<td>4</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>8</td>
<td>7</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td></td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cannot be 3, 7, 8, 9

Might be 1, 2, 4, 5, 6

Two strategies:
(a) Guess for this square
(b) Try other squares
Boolean Satisfiability
(SAT)
SAT in a Nutshell

OR, SUM

AND, PRODUCT

NOT, same as $\overline{a}$

- Given a Boolean formula in **Conjunctive Normal Form** (also called product of sums) find a variable assignment such that the formula evaluates to 1, or prove that no such assignment exists.

$$F = (a + b)(a' + b' + c)$$

- For $n$ variables, there are $2^n$ possible truth assignments to be checked.

- First established NP-Complete problem in 1973 by Cook.
Exploration 2

• Convert given Sudoku puzzle into a SAT problem
  \[(a + b + c') (d + e' + f) (a' + d) (f + g + h')\]

• To use popular SAT solvers the problem should be 3-SAT, i.e., at most 3 literals in each clause like in the above example
  • We do not require this in Exploration 2 since you are writing your own solver

How to do the conversion?
One Way

- We have 9 different values for each slot \((i, j)\)
- Encode using 4 Boolean variables \(a_{ij}, b_{ij}, c_{ij}, d_{ij}\)
- Disallow 0000, 1010, 1011, 1100, 1101, 1110, 1111 for each \((i, j)\) variable set
- Write row, column and block constraints as clauses
- A maximum of \(81 \times 4 = 362\) variables; lots of variables are set to particular values in given puzzle

- Code for this was a little messy!
A Different Way

• Suppose we instead had $v_{ijk}$ where $v_{ijk} = 1$ if and only if the value in cell $(i, j)$ is $k$

• 729 variables

• But generation of clauses is simpler
  – No need to disallow values as in previous strategy
  – Constraints are a little easier to write
Some Hints for Exploration 2 Conversion

- Every cell has at most one value
- Means at most one of $v_{ij1}$, $v_{ij2}$, … $v_{ij9}$ is a 1
- Suppose I have variables $a$, $b$, $c$ and $d$ and at most one of them should be a 1
  \[(a' + b') (a' + c') (a' + d') (b' + c') (b' + d') (c' + d')\]
- For every region (row, column, 3x3 block) there is at least one of the numbers in that region, meaning one of the $v_{ij1}$’s is a 1, one of the $v_{ij2}$’s is a 1, etc.
- You get to figure out these $n$-clauses
Converting n-SAT into 3-SAT (optional in Exploration 2)

Given \((a + b + c + d) (e + f + g)\)

Convert \((a + b + c + d)\) into 3-clauses

\((a + b + n_1') (a' + n_1) (b' + n_1)\)  // \(a + b == n_1\)
\((n_1 + c + d)\)

\(n_1\) is a new variable
Solving SAT using the DPLL Algorithm

- Davis, Logemann and Loveland


- Basic framework for many modern SAT solvers
Basic DPLL Procedure

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)
Basic DPLL Procedure

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

⇐ Decision
Basic DPLL Procedure

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]

\[\text{Decision}\]
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

(a' + b + c) Implication Graph
(makes checks more efficient)

(a + c + d)
(a + c + d')

(a + c + d)
(a + c + d')

Conflict!
Basic DPLL Procedure

Implication Graph
(makes checks more efficient)

Conflicts!
Basic DPLL Procedure

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]
Basic DPLL Procedure

\[(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)\]

\[c=1\]
\[d=1\]
\[d=0\]

Conflict!
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

⇐ Backtrack
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Forced Decision
Basic DPLL Procedure

(a’ + b + c)
(a + c + d)
(a + c + d’)
(a + c’ + d)
(a + c’ + d’)
(b’ + c’ + d)
(a’ + b + c’)
(a’ + b’ + c)

Conflict!
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

← Backtrack
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

d=1
c=1
a=0
d=1
c=1
(a' + b + c')
(a + c' + d')

Conflict!
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

⇐ Backtrack
Basic DPLL Procedure

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]

Forced Decision
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Decision
Basic DPLL Procedure

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

(\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\])

Backtrack
Basic DPLL Procedure

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]

Forced Decision

\[\Rightarrow\]

\[\Rightarrow\text{Forced Decision}\]
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Forced Decision

Implication
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

\[ \begin{align*}
    a &= 1 \\
    b &= 1 \\
    c &= 1 \\
    d &= 1 
\end{align*} \]
Basic DPLL Procedure

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

(a' + b' + c)
(b' + c' + d)

a=1
b=1
c=1
d=1

 contraseña

SAT

Implication
Optional Optimizations in Exploration 2

• There are many optimizations that public-domain SAT solvers use
  – Conflict resolution, non-chronological backtracking, learning new clauses, etc
  – Contact course staff if you want to know more

• We’ll have a prize for the fastest SAT solver according to runtime on new hard Sudoku puzzles (participation is optional and does not affect your grade)
Code Base for Exploration 2
(Immutable types and “functional programming”)
Code Base for Exploration 2

- Immutables
  - Immutable list
  - Immutable Map
- Clausal
- Bool
- Sudoku

You write recursive DPLL solver
Data Structures as Productions

• Many data structures can be described as productions of a grammar
  – tuples:  $\text{Tuple} = \text{Tup}(\text{fst}: \text{Object}, \text{snd}: \text{Object})$
  – lists:  $\text{List} = \text{Empty} + \text{Cons}(\text{first}: \text{Object}, \text{rest}: \text{List})$
  – trees:  $\text{Tree} = \text{Empty} + \text{Node}(\text{val}: \text{Object}, \text{left}: \text{Tree}, \text{right}: \text{Tree})$

• Read this as: A List is the Empty list or the cons (concatenation) of an Object and a List
Polymorphic datatypes

• suppose we want lists over any type
  – that is, allow list of naturals, list of clauses
  – called “polymorphic” or “generic” lists
    \[\text{List}<E> = \text{Empty} + \text{Cons}(\text{first}: E, \text{rest}: \text{List}<E>)\]
  – another example
    \[\text{Tree}<E> = \text{Empty} + \text{Node}(\text{val}: E, \text{left}: \text{Tree}<E>, \text{right}: \text{Tree}<E>)\]
Variant as Class pattern

- create an abstract class for the datatype, and one subclass for each variant, with field and getter for each arg
- production

\[
\text{List}\langle\text{E}\rangle = \text{Empty} + \text{Cons}(\text{first: E, rest: List}\langle\text{E}\rangle)
\]

```java
public abstract class List<E> {}
public class Empty<E> extends List<E> {}
public class Cons<E> extends List<E> {
    private final E first;
    private final List<E> rest;
    public Cons (E e, List<E> r) {first = e; rest = r;}
    public E first () {return first;}
    public List<E> rest () {return rest;}
}
```
Interpreter pattern

• how to build a recursive traversal
  – write type declaration of function
    size: List<E> -> int

  – break function into cases, one per variant
    List<E> = Empty + Cons(first:E, rest: List<E>)
    size (Empty) = 0
    size (Cons(e, list)) = 1 + size(list)

  – implement with one subclass method per case
Implementation of size

• Set size in constructor
• Can determine size on creation -- it never changes* because immutable

```java
public abstract class List<E> {
    int size;
    public int size() { return size; }
}
public class Empty<E> extends List<E> {
    public EmptyList() { size = 0; }
}
public class Cons<E> extends List<E> {
    private final E first;
    private final List<E> rest;
    private Cons(E e, List<E> r)
    { first = e; rest = r; size = r.size()+1 }
}
```

Can’t mark it final!
public List<E> add(E e) {
    return new NonEmptyList<E>(e, this);
}

public List<E> remove(E e) {
    if (element.equals(e)) {
        return rest;
    } else {
        List<E> l = rest.remove(e);
        if (l == rest) return this;
        else return new NonEmptyList<E>(element, l);
    }
}
Code Base for Exploration 2

Immutables
- Immutable list
- Immutable Map

Clausal
- SATProblem
- Clause

Bool

Sudoku
- You write recursive DPLL solver

enum Boolean type
Immutable maps from Variable to Bool
(called Environment)
SATProblem = SATProblem(clauses: List<Clause>)
Clause = Clause(literals: List<Literal>)
Literal = PosLiteral(var: Variable) + NegLiteral(var: Variable)
Variable = Variable(name: String)
public abstract class Literal {
    private Variable var;
    protected Literal negation; // want to set this in subclasses Pos/NegLit
    ...
}

Public class clause {
    private final List<Literal> literals;
    ...
}

Public class SATproblem {
    private final List<clause> clauses;
    ...
}
Recursive DPLL Solver

Pseudo code

```
private static Environment solve(List<Clause> clauses, Environment env) {
    if (clauses is empty) return env; // we are done!
    Clause c = pick a clause with fewest literals
    if(c is empty) return null;
    if(c has a single literal) {
        Variable v = variable referred to in clause c
        Environment env2 = set v in env to satisfy clause c
        return solve(reduceClauses(clauses, v, env2), env2);
    }
    Variable v = pick some variable
    Environment env2 = set v in env to be true
    Environment answer = solve(reduceClauses(clauses, v, env2), env2);
    if (answer != null) return answer;
    Environment env3 = set v in env to be false
    return solve(reduceClauses(clauses, v, env3), env3);
}
```
Questions?