How to design a Sudoku solver: A Case Study

Fall 2010
Recall Sudoku

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

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>5</th>
<th></th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td>8</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>1</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>6</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>2</td>
<td>4</td>
<td></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 1, 2, 3, 8
- Cannot be 9, 4, 6, 1
- Cannot be 1, 9, 7, 8
## Inference

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>5</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>5</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
<td></td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>5</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>1</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>6</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>7</td>
<td>4</td>
<td>2</td>
<td>1</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
- Can only be 5
Inference (contd.)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<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>9</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<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>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>---</td>
<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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></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 Conjointive 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.
Solving Sudoku Using SAT

- 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
  - 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 \(81 \times 4 = 324\) variables; lots of variables are set to particular values in given puzzle

- Code for this is 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
Conversion to SAT

• Every cell has at most one value
• Means at most one of \(v_{ij1}, v_{ij2}, \ldots v_{ij9}\) is a 1
• Suppose I have variables \(v_{ij1}, v_{ij2}, v_{ij3}\) and \(v_{ij4}\) and at most one of them should be a 1
  \[
  (v_{ij1} + v_{ij2}) (v_{ij1} + v_{ij3}) (v_{ij1} + v_{ij4}) (v_{ij2} + v_{ij3}) \\
  (v_{ij2} + v_{ij4}) (v_{ij3} + v_{ij4})
  \]
• 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.
• Try to figure out these n-clauses!
Converting n-SAT into 3-SAT

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)

a
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

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

(A)\(\equiv\) Decision

Decision diagram:

- Node a
- Node b
- Decision 0
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

Implication Graph
(makes checks more efficient)

(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
0
0
0

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

Conflict!
c=0
(a + c + d')
d=0
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)

Implication Graph
(makes checks more efficient)

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

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

\[\text{Conflict!}\]

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

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

d = 1

Conflict!

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

\[ \rightarrow \text{Decision} \]
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)

Conflict!

Forced Decision

Conflict!
Basic DPLL Procedure

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

\[\Rightarrow \text{Backtrack}\]
Basic DPLL Procedure

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

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

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

Diagram:
- Node a with children b and c
- Node b with child c
- Node c with children 0 and 1
- Decision arrow pointing to 0
Basic DPLL Procedure

(a' + b + c)
(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')

(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)

(a' + b' + c) \implies \text{Forced Decision}
Basic DPLL Procedure

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]
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=1
b=1
c=1
d=1

b
0 1
 0 1
 0 1
 0

a
0 1

1

A tree diagram is shown with nodes labeled a, b, c, and d, and branches labeled with clauses such as (a' + b' + c) and (b' + c' + d), illustrating the DPLL algorithm.
Basic DPLL Procedure

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

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

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

0 1
0 1
0 1
0 1
0 1
0 1

SAT

Implication

SAT
Code Base Using Immutable types
Code Base for SAT

- Immutables
  - Immutable list
  - Immutable Map
- Clausal
- Bool
- Sudoku
  - Recursive
  - DPLL solver
Data Structures as Productions

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

  – another example
    Tree\(<E>\) = Empty + Node(val: E, left: Tree\(<E>\),
        right: 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

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

```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
    what goes here?  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;}
}
```

```java
public class Empty<E> extends List<E> {
    public EmptyList () {size = 0;}
}
```

```java
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 }
}
```

Should we mark this as final?
public List<E> add(E e) {
    return new Cons<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 Cons<E> (element, l);
    }
}
Code Base for Sudoku

- **Immutables**
  - Immutable list
  - Immutable Map

- **Clausal**
  - SAT problem
  - Clause

- **Bool**

- **Sudoku**
  - Recursive
  - DPLL solver

**Enum Boolean type**

**Immutable maps from Variable to Bool**
(called Environment)
CNF Grammar

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

```java
private static Environment solve(List<Clause> clauses, Environment env) {
    if (clauses is empty) return env;  // we found a satisfying assignment
    if (clauses == null) return null;  // we could not find a satisfying assignment

    Clause c = pick a clause with fewest literals
    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);
}
```

Implication

No easy deduction, so we guess

Backtrack
Questions?