More SAT and Functions over Recursive Data Structures

Spring 2012
Recall Sudoku

No repeats on rows

No repeats on columns

No repeats within each 3x3 block
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>
<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></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></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
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)
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.

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

First established **NP-Complete** problem in 1973 by Cook.
get citation count for cook
Sharad Malik, 4/20/2004
Convert given Sudoku puzzle into a SAT problem

\[(a + b + c') (d + e' + f) (a' + d) (f + g + h')\]

How to do the conversion?
One Way

**Binary Encoding**

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
- A maximum of \(81 \times 4 = 362\) variables;
- Lots of variables are set to fixed values in given puzzle

Write row, column and block constraints as clauses

This strategy is a bit messy!
A Different Way

Unary Encoding

Suppose we instead had $v_{ijk}$ where

$$v_{ijk} = 1 \text{ if and only if the value in cell } (i, j) \text{ is } k$$

729 variables, but generation of clauses is simpler

- No need to disallow values as in previous strategy
- Constraints are a little easier to write
Sample constraints

**Every cell has at most one value**
- Means at most one of $v_{i1j}$, $v_{i2j}$, ... $v_{i9j}$ is a 1

**For every region (row, column, 3x3 block) there is at least one of the numbers in that region**
- i.e. one of the $v_{i1j}$’s is a 1, one of the $v_{i2j}$’s is a 1, etc.

Suppose at most one of $a$ $b$ $c$ should be 1
- You can encode that as:
  $$(a' + b') (a' + c') (a' + d') (b' + c') (b' + d') (c' + d')$$
Solving SAT with the DPLL Algorithm

Martin Davis, George Logemann and Donald Loveland

Based on previous algorithm by Davis and Hilary Putnam

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)

⇐ 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)

⇐ Decision
Basic DPLL Procedure

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

c = 0

d = 1

Conflict!

Implication Graph
(one way to describe 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)

c=0
b=0
a=0

Implication Graph
(one way to describe conflicts)

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

(c = 1 ⇐ Forced Decision

Conflict!

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

(a' + b + c')

(d = 1)

(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)
Basic DPLL Procedure

(a' + b + c)  
(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)

Conflict!

⇐ Decision

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)

Conflicts!

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)

⇐ Backtrack
Basic DPLL Procedure

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)

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

\[
\begin{align*}
(a' + b + c) \\
(a + c + d) \\
(a + c + d') \\
(a + c' + d) \\
(a + c' + d') \\
(b' + c' + d) \\
(a' + b + c') \\
(a' + b' + c)
\end{align*}
\]

\[
\begin{align*}
\text{a=1} \\
\text{b=1} \\
\text{c=1}
\end{align*}
\]

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

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

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

(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)

b=1

c=1

d=1

a=1

b

0 1

1

0

1

0

1

1

1

1

Implication

SAT
Recursive DPLL Solver

Pseudo code

private static Environment solve(ImList<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);
}
IMPLEMENTING FUNCTIONS OVER IMMUTABLES
Data Structures as Productions

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:  \( \text{Tree} = \text{Empty} + \text{Node}(\text{val}: \text{Object}, \text{left}: \text{Tree}, \text{right}: \text{Tree}) \)
Polymorphic datatypes

suppose we want lists over any type

- that is, allow list of naturals, list of clauses
- called “polymorphic” or “generic” lists

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

\[
\text{Tree}<E> = \text{Empty} + \\
\quad \text{Node}(\text{val}: E, \text{left}: \text{Tree}<E>, \text{right}: \text{Tree}<E>)
\]
Variant as Class pattern

create an abstract class for the datatype, one subclass for each variant

Ex:

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

```java
public abstract class List<E> {
    public abstract int size();
}

public class Empty<E> extends List<E> {
    public int size() {return 0;}
}

public class Cons<E> extends List<E> {
    private final E first;
    private final List<E> rest;
    public int size() {return 1 + rest.size();}
}
```
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(first:e, rest: l)) = 1 + size(rest)

➢ implement with one subclass method per case

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

look at this implementation

➤ representation is mutable, but abstractly object is still immutable!

```java
public abstract class List<E> {
    int size;
    boolean sizeSet;
    public abstract int size();
}
public class Empty<E> extends List<E> {
    public int size () {return 0;}
}
public class Cons<E> extends List<E> {
    private final E first;
    private final List<E> rest;
    public int size () {
        if (sizeSet) return size;
        int s = 1 + rest.size();
        size = s; sizeSet = true;
        return size;
    }
}
```
Representation Invariants

public abstract class List<E> {
    int size;
    boolean sizeSet;
    public abstract int size();
}

size and sizeSet are part of the representation

Many representations for the same list

Concrete Lists

Abstract Lists
Representation Invariants

**size and sizeSet are part of the representation**

- Many representations for the same list
- Some representations do not correspond to a valid list

Concrete Lists

<table>
<thead>
<tr>
<th>Cons size=2</th>
<th>Cons size=1</th>
<th>Empty size=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeSet=T</td>
<td>sizeSet=T</td>
<td>sizeSet=F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cons size=0</th>
<th>Cons size=1</th>
<th>Empty size=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeSet=F</td>
<td>sizeSet=T</td>
<td>sizeSet=F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cons size=0</th>
<th>Cons size=1</th>
<th>Empty size=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeSet=T</td>
<td>sizeSet=T</td>
<td>sizeSet=F</td>
</tr>
</tbody>
</table>

Abstract Lists

- Cons → Cons → Empty
- Cons → Empty
- Cons → Empty
Representation Invariants

**size and sizeSet are part of the representation**

- Many representations for the same list
- Some representations do not correspond to a valid list

A representation invariant is a predicate that describes the valid representations!

\[ \neg \text{sizeSet} \lor \left( \text{size} == \text{rest.size} + 1 \right) \]
size, finally

in this case, best just to set upon creation

- can determine size on creation -- and never changes because immutable

```java
public abstract class List<E> {
    final int size;
    public int size () {return size;}
}

public class Empty<E> extends List<E> {
    public Empty () {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 }
}
```

Well, almost; there is a small bug here; can you see it?
size, finally

**in this case, best just to set in constructor**

- can determine size on creation -- and never changes because immutable

```java
public abstract class List<E> {
    final int size;
    List(int size){ this.size = size; }
    public int size () {return size;}
}

public class Empty<E> extends List<E> {
    public Empty () {super(0)}
}

public class Cons<E> extends List<E> {
    private final E first;
    private final List<E> rest;
    private Cons (E e, List<E> r) { super(r.size()+1); first = e; rest = r;}
}
```
public ImList<E> add(E e, ImList<E> l) {
}

public ImList<E> remove(E e) {
  ??
  Empty case:
  Cons case:
    if (t.equals(e))
      return l.remove(e)
    else
      return Cons(t, l.remove(e))
}
Back to our equation solver

Expression :=

    NumExpr( val:double ) +
    VarExpr( var:String ) +
    PlusExpr( left:Expression, right:Expression ) +
    MinusExpr( left:Expression, right:Expression ) +
    TimesExpr( left:Expression, right:Expression ) +
    DivExpr( left:Expression, right:Expression ) +
solve(Expression e, Expression res)

march e:
    VarExpr(var): return res
    PlusExpr(l, r): if(l.hasVar())
        return solve(l, MinusExpr(res, r))
    else
        return solve(r, MinusExpr(res, l))
    MinusExpr(l, r): if(l.hasVar())
        return solve(l, PlusExpr(res, r))
    else
        return solve(r, MinusExpr(res, l))
    TimesExpr(l, r): if(l.hasVar())
        return solve(l, DivExpr(res, r))
    else
        return solve(r, DivExpr(res, l))

You can now implement this with the interpreter pattern!

Solves an equation of the form
\[ e = res \]
Assumes e contains the variable \( x \)
Issues with the interpreter pattern

**Pros**
- It’s easy to implement
- Code is simple and pretty

**Cons**
- Every time you want to add a new function you have to modify all your variant classes!
- Imagine if every time you wanted to implement a function over a list you had to go into the source code of the List library and modify it.

This is a common problem, so yes, there is a design pattern to address it! (more on this on the next lecture).