Homework 2: Component wiring

Bart Massey

2013/10/27

It is good to be able to encode a constraint satisfaction problem as a CNF propositional formula. You can then use a fast SAT solver to find out whether a satisfying assignment exists, and translate such an assignment back to a solution to the original problem.

Imagine that you are designing a printed-circuit board for a piece of electronics. At some point, you come to a situation where you have two rows of components facing each other. You can rearrange components within either row as much as you like, but you must place every component somewhere within its home row. Further, you are told what connections have to be made between components on the left and components on the right in order to get a functioning circuit. Finally, comes the kicker: you want to put all the wires on the same layer of the printed circuit board, so you cannot allow any two wires to cross (or to “go around the outside”).

Instances of this problem are thus naturally described by a height $n$ of the component rows, and a Boolean matrix describing connectivity between left and right components. For example, the simple instance of height 3 described by the matrix

<table>
<thead>
<tr>
<th></th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1$</td>
<td>$f$</td>
<td>$t$</td>
<td>$f$</td>
</tr>
<tr>
<td>$l_2$</td>
<td>$f$</td>
<td>$t$</td>
<td>$f$</td>
</tr>
<tr>
<td>$l_3$</td>
<td>$t$</td>
<td>$t$</td>
<td>$t$</td>
</tr>
</tbody>
</table>

has as a solution

```
  l_3 ------- r_1
     |         |
    l_2 ------- r_3
        |         |
       l_1 ------- r_2
```

Let us solve instances of this wiring problem by transforming them to SAT instances, solving those with a SAT solver, and transforming any satisfying assignment back to a solution.

To start with, let us define propositional atoms for the solution as follows. $L_{ij}$ is true iff left component $i$ is at position $j$. Similarly, $R_{ij}$ is true iff left component $i$ is at position $j$. Finally, $W_{ij}$ is true iff there is a wire from left position $i$ to right position $j$.

We then need to define propositional atoms describing the instance. This is as simple as defining $C_{ij}$ which is true iff left component $i$ is must be connected to right component $j$.

Now we just need to define the axioms that govern the placement of the components. First, we posit the usual existence axioms (every component is somewhere)

$$\forall i . \exists j . L_{ij}$$
$$\forall i . \exists j . R_{ij}$$
and uniqueness axioms (only one component is at each location)
\[ \forall ijk . \ (i \neq j) \land L_{ik} \rightarrow \neg L_{jk} \]
\[ \forall ijk . \ (i \neq j) \land R_{ik} \rightarrow \neg R_{jk} \]
for the layout itself. (Thanks to Daniel Cristofani for correcting the uniqueness axioms.)

We then require that positions are connected by a wire iff their components are connected by a wire.
\[ \forall hijk . \ L_{hj} \land R_{ik} \rightarrow (C_{hi} \leftrightarrow W_{jk}) \]

Finally, we require that wires not cross. A moment’s thought gives us the axiom that wires may be crossed neither from above nor from below.
\[ \forall ijkm . \ W_{ij} \land (k < i) \land (m > j) \rightarrow \neg W_{km} \]
\[ \forall ijkm . \ W_{ij} \land (k > i) \land (m < j) \rightarrow \neg W_{km} \]

The tools we might use as Prop CNF SAT solvers include MiniSat2 [http://minisat.se/MiniSat.html] and PicoSat [http://fmv.jku.at/picosat/]. Like most modern SAT solvers, these tools want DIMACS input format [http://www.satlib.org/Benchmarks/SAT/satformat.ps or http://svcs.cs.pdx.edu/circuit-wiring/satformat.pdf]. This format is an ASCII text representation of a Prop CNF formula, designed to be easy to parse. It begins with a line like:

```
p cnf nvars nclauses
```

that gives the number of variables and clauses in the formula. It then continues with 0-terminated lists of integers representing disjunctive clauses: all the lists are conjoined to make the final formula. Each integer is the index of an atom of the formula; negated atoms are represented by negative integers. Thus, the formula
\[(A \lor B) \land (C \lor \neg B)\]
might be represented as:
```
p cnf 3 2
 1 2 0
 3 -2 0
```

So our job is to number the atoms in the formulæ somehow, and transform the formulæ into CNF so that we can feed them to the SAT solver.

To number the atoms, just use array indexing, essentially. Hopefully it’s clear. Remember that 0 is not a valid atom number! It is probably best if the atom numbers used are sequential; the SAT solver may not be efficient if there are wild gaps corresponding to unmentioned atoms.

The easiest way to transform the formulæ is probably to first take each of the axioms above and transform them into prenex CNF; this will give a bunch of clauses. Hand tweaking is also acceptable. The result looks like:

\[ \forall i . \ \exists j . \ L_{ij} \quad \text{(1)} \]
\[ \forall i . \ \exists j . \ R_{ij} \quad \text{(2)} \]
\[ \forall ijk . \ (i \neq j) \rightarrow \neg L_{ik} \lor \neg L_{jk} \quad \text{(3)} \]
\[ \forall ijk . \ (i \neq j) \rightarrow \neg R_{ik} \lor \neg R_{jk} \quad \text{(4)} \]
\[ \forall hijk . \ \neg L_{hj} \lor \neg R_{ik} \lor \neg C_{hi} \lor W_{jk} \quad \text{(5)} \]
\[ \forall hijk . \ \neg L_{hj} \lor \neg R_{ik} \lor \neg W_{jk} \lor C_{hi} \quad \text{(6)} \]
\[ \forall ijkm . \ (k < i) \land (m > j) \rightarrow \neg W_{ij} \lor \neg W_{km} \quad \text{(7)} \]
\[ \forall ijkm . \ (k > i) \land (m < j) \rightarrow \neg W_{ij} \lor \neg W_{km} \quad \text{(8)} \]
Hopefully it should be easy to see how to encode these. Note that for the forms that do index arithmetic \([3, 4, 7, 8]\) one simply only generates some of the constraints; the SAT solver doesn’t need to do any arithmetic here. Daniel Cristofani points out that \([8]\) is the same as \([7]\) up to renaming of indices, so it can be dropped.

You will also need to encode the constraints \(C_{ij}\) that specify the connections to be made. These are just singleton clauses. Encode the negative cases \(\neg C_{ij}\) as well as the positive ones.

Now you feed the resulting DIMACS into your SAT solver. You then read the solution, if any, off the signs of the \(L_{ij}\) and \(R_{ij}\) atoms.

I would recommend obtaining or implementing a solution checker that checks that the columns of your proposed solution are a permutation of \(1..n\) and that no wires cross in the proposed solution. I have made a Haskell checker available at [http://svcs.cs.pdx.edu/circuit-wiring/wiring-check.hs](http://svcs.cs.pdx.edu/circuit-wiring/wiring-check.hs) that you can run as

```
runghc wiring-check.hs inst-file soln-file
```

if you have GHC installed on your system.

Good luck and have fun!

---

Here are the instances you are being asked to solve. Note that the solutions may not be unique, and that in fact no solution may exist. Output your answer in the form of two columns of integers separated by whitespace, where the integers indicate which component \((1..n)\) is at that position.


```
<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>f</td>
<td>t</td>
<td>f</td>
</tr>
</tbody>
</table>
```


```
<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
<th>r3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>f</td>
</tr>
</tbody>
</table>
```


```
<table>
<thead>
<tr>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
<th>r8</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>t</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
</tbody>
</table>
```

Please submit the usual material—code and brief writeup—in the usual fashion.

Have fun!