CS441 WRITEUP

Yiming Lin

yl6@pdx.edu

Atoms

L_{ij} is true iff left component i is at left position j

R_{ij} is true iff right component i is at right position j

W_{ij} is true iff there is a wire from left position i to right position j

C_{ij} is true iff left component i must be connected to right component j.

Predicates

Consider there are n position on the left side and n position on the right side, also there are n left components and n right components. Every component on each side can be installed on any position, thus we have L_{11}, L_{12}, …, L_{1n}, L_{21}, L_{22}, …, L_{2n}, ..., L_{n1}, L_{n2}, …, L_{nn}, which are n^2 literals. Same, we have R_{11}, R_{12}, …, R_{1n}, …, R_{n1}, R_{n2}, …, R_{nn}, which are n^2 literals. Each position on the left can have n wires that connect to each position on the right, which has already covered the situation where each position on the right can have n wires that connect to each position on the left, thus we have W_{11}, W_{12}, W_{1n}, W_{21}, W_{22}, … W_{2n}, …, W_{n1}, W_{n2}, …, W_{nn}, which are n^2 literals. Finally, C_{ij} is same as W_{ij}, thus we have C_{11}, …, C_{nn}, which are n2 literals.

From above, four predicates, L, R, W, and C, produces the following literals that has been encoded:

Predicate	**Range**	**Returned Value**
L(i, j)	1 to n^2	$1 + i \times n + j$
R(i, j)	$n^2 + 1$ to $2 \times n^2$	$1 + n^2 + i \times n + j$
W(i, j)	$2 \times n^2 + 1$ to $3 \times n^2$	$1 + 2 \times n^2 + i \times n + j$
C(i, j)	$3 \times n^2 + 1$ to $4 \times n^2$	$1 + 3 \times n^2 + i \times n + j$

table 1 *predicate table, the arguments i and j are 0-based*

Modeling -- quantified propositional logic

a. Existence

$$\forall\, i\, \exists\, j\, L_{ij}$$

$$\forall\, i\, \exists\, j\; R_{ij}$$

```
For i in 1..n:
    Clause-left ← [L(i, j) for j in 1..n]
    Clause-right ← [R(i, j) for j in 1..n]
    Cnf-clauses.add-clause(Clause-left)
    Cnf-clauses.add-clause(Clause-right)
```

b. Uniqueness

$$\forall\, i, j, k\, .\, (i \neq j) \rightarrow \neg L_{ik} \cup \neg L_{jk}$$

$$\forall\, i, j, k\, .\, (i \neq j) \rightarrow \neg R_{ik} \cup \neg R_{jk}$$

```
For k in 1..n:
    For i in 1..n:
        For j in i+1..n:
            Clause-left ← [-L(i, k), -L(j, k)]
            Clause-right ← [-R(i, k), -R(j, k)]]
            Cnf-clauses.add-clause(Clause-left)
            Cnf-clauses.add-clause(Clause-right)
```

c. Connection

$$\forall\, h, i, j, k\; \neg L_{hi} \cup \neg R_{ik} \cup \neg C_{hi} \cup W_{jk}$$

$$\forall\, h, i, j, k\; \neg L_{hi} \cup \neg R_{ik} \cup \neg W_{jk} \cup C_{hi}$$

```
For h in 1..n:
    For i in 1..n:
        For j in 1..n:
            For k in 1..n:
                Clause-1← [-L(h, i), -R(i, k), -C(h, i), W(j, k)]
                Clause-2 ← [-L(h, i), -R(i, k), -W(j, k), C(h, i)]
                Cnf-clauses.add-clause(Clause-1)
                Cnf-clauses.add-clause(Clause-2)
```

d. Non-crossing

$$\forall\, i,j,k,m\,.\,(k < i) \cap (m > j) \rightarrow \neg W_{ij} \cup \neg W_{km}$$

```
For k in 1..n:
    For j in 1..n:
        For i in k+1..n:  // i > k
            For m in j+1..n:  // m > j
                Clause ← [-W(i, j), -W(k, m)]
                Cnf-clauses.add-clause(Clause)
```

e. Singleton

$$\forall\, i,j\,.\,ConnectionMatrix(i,j)\ is\ True\ \rightarrow\ C_{ij}$$

$$\forall\, i,j\,.\,ConnectionMatrix(i,j)\ is\ False\ \rightarrow\ \neg C_{ij}$$

```
For k in 1..n:
    For j in 1..n:
        If Connection-matrix(i, j) == 0:
            Clause ← [-C(i, j)]
        Else:
            Clause ← [C(i, j)]
        Cnf-clauses.add-clause(Clause)
```

Decoding

After generating CNF clauses and output it to the .dimacs file, it is fed into a SAT solver, which will produce the results. Then the result is read into the program to execute decoding by reading the signs of L_{ij} and R_{ij}.

```
// Read-file() returns a list that contains the result from SAT solver.
Result-list ← Read-file()
For i in 1..n:  // i is the position on the left side
    For j in 1..n:  // j is the left component number
        If L(i, j) in Result-list:
            Left-components.append(j)
For k in 1..n:  // k is the position on the right side
    For m in 1..n: // m is the right component number
```

```
If R(i, j) in Result-list:
    Right-components.append(j)
```