Coursework 2 Report

Alisher Tortay

April 2017

1 Requirements

I used Python 3 for this project. To run nonogram.py *minisat* should be on path.

2 Conjunctive Normal Form

To run the program use following command:

$ python3 cnf.py ”> & - p q & p > r q”

It is important to have the formula in double quotes. cnf directory contains only one file: *cnf.py*. This file contains two classes: FNode and Formula. Tree data structure is used to represent formulae. Formula class uses FNode class and contains all the functions required for this assignment.Most of them are declared as static functions.
to-cnf is the main function of this class. This method performs following procedures:

1. Remove all equivalence expressions by changing $a = b$ with $(a > b)(b > a)$
2. Remove all implications by changing $a > b$ with $-a \vee b$
3. Eliminate all non-literal negations using De Morgan's Law
4. Finally convert to CNF by recursively distributing $(a \wedge b) \vee c$ to $(a \vee c) \wedge (b \vee c)$
5. Convert formula tree into list of lists where each inner list is disjunction
6. Print CNF in Polish Notation
7. Print CNF in infix notation
8. Decide whether CNF is valid

Before calling this method, we first build formula tree using build-tree method. It recursively builds tree from input polish notation. Each node contains either literal or operation. Each operation node has two children for two operands. For negation, only left child is used.

After we convert our formula (tree) into CNF, we convert it into list of lists where each inner list is disjunction. For example, $(a \vee b) \wedge (p \vee q)$ is converted to [[a, b], [p, q]]. We then use this form to check validity of the CNF. We do it by simply by searching for complimentary literals in every inner list (disjuntion list). We also use this list to print CNF in infix notation. We use tree structure to print CNF in polish notation.

3 Nonogram as SAT

nonogram.py : Main file.
nonogram.in : Input nonogram in CWD format. No empty line between vertical and horizontal rules.
minisat.in : Input file for *minisat*. Created by *nonogram.py*.
minisat.out : Output of *minisat*.
minisat.result : Statistics of *minisat*.

To run the program use following command:

$ python3 nonogram.py

In my implementation I have an atom (literal) for every square. An atom is true if and only if the corresponding square is filled. I also have auxiliary atoms for every number in clue. Suppose we have a following clue: $r_1, r_2, ..., r_t$ where r_i is a number. Suppose b_i is a block of coloured squares corresponding to r_i. That implies that b_i consists of r_i squares. Let us compute the number of possible positions of first (leftmost or topmost) square of b_i (according to the clue). The index of leftmost (topmost) possible position for the first square is $\sum_{k=1}^{i-1} r_k + i$ (1) and the index of the rightmost (bottommost) possible position for the first square is $N - \sum_{k=i}^{t} r_k - t + i + 1$ (2) where N is length of row (column). Therefore, we can find the number of possible positions of first square of b_i by subtraction (1) from (2) and adding 1:

$$c = N - \sum_{k=i}^{t} r_k - t + i + 1 - \sum_{k=1}^{i-1} r_k - i + 1 = N - \sum_{k=1}^{t} r_k - t + 2$$

Here we can see that for all blocks b_i, number of possible positions of first square is the same. For every b_i we set c auxiliary atoms $a_{i,1}, a_{i,2}, ..., a_{i,c}$ in following way: $a_{i,j}$ is true if and only if b_i starts on the jth possible position.

Now we are ready to define four correct solution conditions.

1. For every r_i at least one of its auxiliary atoms is true.
   $\bigwedge_{allclues} \bigwedge_{i=1}^{t} \bigvee_{j=1}^{a} a_{i,j}$

2. For every r_i at most one of its auxiliary atoms is true.
   $\bigwedge_{allclues} \bigwedge_{i=1}^{t} \bigwedge_{allpairs} (\neg a_{i,j} \vee \neg a_{l,k})$

3. If $a_{i,j}$ then $a_{i+1,j}$ or $a_{i+1,j+1}$ or ... or $a_{i+1,c}$.
   $\bigwedge(a_{i,j} \Rightarrow a_{i+1,j} \vee a_{i+1,j+1} \vee ...) = \bigwedge(\neg a_{i,j} \vee a_{i+1,j} \vee a_{i+1,j+1} \vee ...)$

4. A square atom is true if and only if disjunction of corresponding auxiliary atoms is true. $\bigwedge(s_{k,l} \iff a_{i,j} \vee a_{i,j+1} \vee a_{i+1,0} \vee ...) = \bigwedge(\neg s_{k,l} \vee a_{i,j} \vee a_{i,j+1} \vee a_{i+1,0} \vee ...) \wedge (s_{k,l} \vee \neg a_{i,j}) \wedge (s_{k,l} \vee \neg a_{i,j+1}) \wedge ...$

In condition 3 we do not need to add left implications ($a_{i,j} \Leftarrow a_{i+1,j}$) because combination of right implications and condition 2 make sure that the left implications hold. You can see that every condition is already in cnf form. Thus, we don't need to do any conversions.

In my implementation I did not declare any variables for atoms. Instead I used number (in the form of strings) to represent atoms. 1 to N*M are square atoms and from N*M + 1 I have auxiliary atoms.

nonogram.py prints its output in terminal. The output is completed puzzle ("#" for filled squares and "." for blank). Neighboring squares are separated by one whitespace. I did not include clues to the output.

I tested my code on puzzles from http://webpbn.com/export.cgi My code successfully solved the biggest puzzle I could find there - 99x99 (http://webpbn.com/export.cgi/webpbn017967.cwd). However, before inputting the puzzle, blank line that separates vertical and horizontal clues in the puzzle should be removed.