Report of CourseWork2: Normal Forms, Validity, and Satisfiability

20163660 Junmin Choe

April 7, 2016

1 CONJUNCTIVE NORMAL FORM

1.1 DEVELOPMENT ENVIRONMENT

- OS - *OS X 10.11(El Capitan)*
- Bash - *GNU bash, 3.2.57 (x86_64-apple-darwin15)*
- Language - *C++11*
- Compiler - *g++ 4.9(installed using 'Homebrew')*

You can also run this program in ubuntu or other linux environment.

1.2 HOW TO APPROACH THE PROBLEM

According to specification of coursework, we can summarize what we have to do. The lists of them are as follows:

1. Take a single propositional formula in *Polish Notation* as input.
2. Convert entered formula to *CNF(Conjunctive Normal Form)*.
3. Print converted formula in *Polish Notation(Prefix) and infix notation.*
4. Check *Validity* of formula

Let me explain details of each work.

TAKE A SINGLE PROPOSITIONAL FORMULA In this problem, a single propositional formula is entered as an input in polish notation(prefix). First, we can think problem that how to store this input formula. I used **binary tree data structure** to solve it, because of property of operator in formula. **The operators always have two parameters**. Using this property, we can make binary tree structure representing formula(call it **FormulaTree**). In FormulaTree, each node appears as literal or operator. And operator node has two childs for each parameter(except not(-) operator, this one has only one child). For example, if formula is "> & - p q & p > r q", there are 11 nodes in FormulaTree and '>', '&', '&', '>' have two children and '-' has only one children. I will show the implementation details in **Implementation** section. For now, we just think about how to store formula. And FormulaTree can be solution for it.

CONVERT ENTERED FORMULA TO CNF After making FormulaTree, we should transform it to have CNF. At initial state, FormulaTree have several operator such as equivalence(=), implication(>) and reverse implication(<). Because CNF does not have these operators, we should remove these operators. In class, we learned that **every formula can be transformed to CNF** and how to transform. So, implementing that method, we can easily convert FormulaTree to new FormulaTree having CNF.

PRINT CONVERTED FORMULA Now the FormulaTree has CNF, so we can print it in prefix notation and infix notation using **traversal method of binary tree**. Implementation will be showed in next section.

CHECK VALIDITY OF FORMULA We can also check validity of formula using what we learned in class. In CNF, each AND term must be true for validity. So, we will confirm whether each term is always true. For this, there must be a literal whose negation exists in OR term. We can also check this with traversing the FormulaTree.

1.3 IMPLEMENTATION

Any operation about the FormulaTree is implemented in header file, *"FormulaTree.h"* and main sequence of program is implemented in c++ file, *"cnf.cc"*. You can see how this implemented and in this section, i will show pseudocode of each operation.

TAKE A SINGLE PROPOSITIONAL FORMULA As discussed in above section, we have to make the FormulaTree to store input. I defined FormulaTree as struct(c++ syntax) that only have one member variable 'root'. The type of root is pointer of 'Node' which is also struct. The Node struct has many variables such as name of literal, child, negativity and operation. And we store input information to the FormulaTree using function named 'MakeTree'. This function is **recursive function**. In tree data structure, recursive function is widely used to generate tree or print or traverse. In rest operation(function), you can detect use of recursive method. Let me show pseudocode of 'MakeTree'.

Algorithm 1: MakeTree

Data: *FormulaTree T*, input size *n*, input strings *input*, current input info
Result: *FormulaTree T* that store input information

```
begin
    initialization of tree;
    if No inputs are remained then
        return;
    end
    Make new node to store current input token;
    if token is literal then
        save name of literal to node;
    else
        if token is operator NOT then
            save operator to node;
            LeftChild = MakeTree(T, next input);
        else
            save operator to node;
            LeftChild = MakeTree(T, next input);
            RightChild = MakeTree(T, next input);
        end
    end
end
```

CONVERT ENTERED FORMULA TO CNF As i mentioned above, converting step is absolutely same as learned in class. These steps are *implication free, elimination non-literal negation, distribution*. In addtion, we have two other operators not learned in class, equivalence(=) and reverse implication(<). So we should handle this operators. We can think that formula "p = q" is same as "(- p | q) & (p | - q)" and formula "p < q" as "q > p". As a result, we can transform each operator to original operator. So, i added one step *"equivalence free"* and in *implication free*, i added process about handling reverse implication. Let me show pseudocode for main converting sequence, equivalence free, implication free, convert to NNF(eliminate non-literal negation), distribution. All this functions are **recursive function**. (You can find more detail implementation in the code and feel that tricky, but overall flow is almost same as below pseudocode)

Algorithm 2: ConvertToCNF(Main Sequence)

Data: *FormulaTree T*
Result: *a CNF FormulaTree T*
begin
    EquivTree(T);
    ImpliTree(T);
    ConvertToNNF(T);
    ConverToCNFSub(T);
    return T;
end

Algorithm 3: EquivFree(Eliminate all equivalence operators)

Data: *FormulaTree T*
Result: *An equivalence-free FormulaTree T*
begin
    switch *Root token of T* **do**
        case *token is literal* **do**
            return T;
        end
        case *token is equivalence* **do**
            Copy LeftChild to LeftChild2;
            Copy RightChild to RightChild2;
            return (-EquivFree(LeftChild) | EquivFree(RightChild)) & (EquivFree(LeftChild2)
             | -EquivFree(RightChild2));
        end
        otherwise do
            return EquivFree(LeftChild) *oper* EquivFree(RightChild);
        end
    end
end

Algorithm 4: ImpliFree(Eliminate all implication(also reverse) operators)

Data: *An equivalence-free FormulaTree T*
Result: *An equivalence-free, implication-free FormulaTree T*
begin
    switch *Root token of T* **do**
        case *token is literal* **do**
            return T;
        end
        case *token is reverse-implication* **do**
            Change token to Implication;
            Swap LeftChild and RightChild;
            return ImpliFree(T);
        end
        case *token is implication* **do**
            return -ImpliFree(LeftChild) | ImpliFree(RightChild);
        end
        case *token is NOT* **do**
            return -ImpliFree(LeftChild);
        end
        otherwise do
            return ImpliFree(LeftChild) *oper* ImpliFree(RightChild);
        end
    end
end

Algorithm 5: ConvertToNNF(Eliminate all non-literal negations)

Data: *An equivalence-free, implication-free FormulaTree T*
Result: *An equivalence-free, implication-free, NNF FormulaTree T*
begin
    switch *Root token of T* **do**
        case *token is literal* **do**
            return T;
        end
        case *token is NOT-NOT(next child is also not)* **do**
            return ConvertToNNF(T);
        end
        case *token is NOT-AND* **do**
            return ConvertToNNF(-LeftChildOfAND | -RightChildOfAND);
        end
        case *token is NOT-OR* **do**
            return ConvertToNNF(-LeftChildOfOR & -RightChildOfOR);
        end
        otherwise do
            return ConvertToNNF(LeftChild) *oper* ConvertToNNF(RightChild);
        end
    end
end

Algorithm 6: ConvertToCNFSub(Convert to CNF using distribution)

Data: *An equivalence-free, implication-free, NNF FormulaTree T*
Result: *A CNF FormulaTree T*
begin
    switch *Root token of T* **do**
        case *token is literal* **do**
            return T;
        end
        case *token is AND* **do**
            return ConvertToCNFSub(LeftChild) & ConvertToCNF(RightChild);
        end
        case *token is OR* **do**
            return DISTR(ConvertToCNFSub(LeftChild), ConvertToCNFSub(RightChild));
        end
        otherwise do
            return error;
        end
    end
end

Algorithm 7: DISTR(Used in ConvertToCNFSub for distribution)

```
Data: CNF FormulaTrees T1, T2
Result: A CNF FormulaTress for T1 | T2
begin
    switch Root tokens of T1, T2 do
        case T1 token is AND do
            return T;
        end
        case token of T1 is AND do
            return DISTR(LeftChildOfT1, T2) | DISTR(RightChildOfT1, T2);
        end
        case token of T2 is AND do
            return DISTR(T1, LeftChildOfT2) | DISTR(T1, RightChildOfT2);
        end
        otherwise do
            return T1 | T2;
        end
    end
end
```

PRINT CONVERTED FORMULA We have two types to print, prefix and infix. Such jobs are easily implemented by traversal method of binary tree. I wrote pseudocode each traversal below.

Algorithm 8: PrintPrefix

```
Data: A CNF FormulaTree T
Result: Print FormulaTree in Polish notation
begin
    if Root token of T is literal then
        if Is negative then
            print '- ';
        end
        print name of literal;
    else
        print token(operator);
        PrintPrefix(LeftChild);
        PrintPrefix(RightChild);
    end
end
```

Algorithm 9: PrintInfix

```
Data: A CNF FormulaTree T
Result: Print FormulaTree in infix notation
begin
    if Root token of T is literal then
        if Is negative then
            print '- ';
        end
        print name of literal;
    else
        PrintInfix(LeftChild);
        print token(operator);
        PrintInfix(RightChild);
    end
end
```

CHECK VALIDITY OF FORMULA As explained in "How to approach problem" section, i traversed FormulaTree and confirmed validity of each AND term recursively. Below pseudocode describe real implementation of this. Note that each AND term has operators with **only OR**.

Algorithm 10: IsValid(Check validity of FormulaTree)

```
Data: A CNF FormulaTree T
Result: Validity of FormulaTree T
begin
    res = true;
    if Root token of T is literal then
        res = false;
    else
        if token is AND then
            res &= IsValid(LeftChild) & IsValid(RightChild);
        else
            Clear literal state storage;
            res = CheckTerm(T);
        end
    end
    return res;
end
```

Algorithm 11: CheckTerm(Check validity of each AND term)

Data: *AND term of a CNF FormulaTree T*
Result: *Validty of AND term*
begin
  if *Root token of T is literal* **then**
    if *negation token is saved on literal state storage* **then**
      return true;
    else
      Save token on *literal state storage*;
    end
  else
    return CheckTerm(LeftChild) | CheckTerm(RightChild);
  end
end

1.4 TEST AND RESULTS

Compiling with g++, the program existed in 'cnf' folder with its code. You can execute this program anytime for testing and i tested with 5 testcases as follows:

1. ./cnf "> & - p q & p > r q"
2. ./cnf "- & - | a b - & a - b"
3. ./cnf "- - - - - - - p"
4. ./cnf "| | a b | c d"
5. ./cnf "= = a b = - a - b"

The results of each case are as shown:

1. & | | p - q p | | p - q | - r q
   (p | - q | p) & (p | - q | - r | q)
   Not Valid
2. & | | a b a | | a b - b
   (a | b | a) & (a | b | - b)
   Not Valid
3. - p
   - p
   Not Valid
4. | | a b | c d
   a | b | c | d
   Not Valid

5. & & & & | | a - a | a - b | | a - a | - a b & | | a b | a - b | | a b | - a b & & | | - b - a | a - b | | - b - a | - a b & | | - b b | a - b | | - b b | - a b & & & | | - a b | - a a | | - a b | - a - b & | | - a b | b a | | - a b | b - b & & | | a - b | - a a | | a - b | - a - b & | | a - b | b a | | a - b | b - b
(a | - a | a | - b) & (a | - a | - a | b) & (a | b | a | - b) & (a | b | - a | b) & (- b | - a | a | - b) & (- b | - a | - a | b) & (- b | b | a | - b) & (- b | b | - a | b) & (- a | b | - a | a) & (- a | b | - a | - b) & (- a | b | b | a) & (- a | b | b | - b) & (a | - b | - a | a) & (a | - b | - a | - b) & (a | - b | b | a) & (a | - b | b | - b)
Valid

I confirmed this results are all correct. You can test more complicated cases.

2 NONOGRAM AS SAT

2.1 DEVELOPMENT ENVIRONMENT

- OS - *OS X 10.11(El Capitan)*
- Bash - *GNU bash, 3.2.57 (x86_64-apple-darwin15)*
- Language - *C++11*
- Compiler - *g++ 4.9(installed using 'Homebrew')*
- Tools - *minisat*

You can also run this program in ubuntu or other linux environment.

2.2 HOW TO APPROACH THE PROBLEM

We can also summarize steps to solve nonogram problem. The lists of them are as follows:

1. Take inputs and store all information
2. Using stored information, encode to string as SAT
3. Make FormulaTree that has encoded string and Convert it to CNF
4. Print CNF FormulaTree with minisat input format
5. Execute minisat and print results

Let me explain details of each work.

TAKE INPUT Our task is that take a single puzzle as input, formatted in the CWD format. Entering each line of input, we can store information such as number of row, column and conditions of puzzle. We can be sure that this task is easily solved.

ENCODE INPUT TO STRING AS SAT In this step, i will encode stored input to string as SAT. This is because we need to perform 'ConvertToCNF' after encoding(and this is because *minisat* inputs are always be CNF). As i implemented CNF converter in advance, i will use it again to make jobs more smaller. But for the use of CNF converter, we must enter the formula string to the CNF converter in Polish Notation. This **formula string in Polish Notation** is what said 'input'. So in summary, our job is to make the formula string that can be input of CNF converter.
So, how can we encode? First of all, i generate (row number * column number) literals to represent if current box might be filled or not. In other words, **literal** $x_{i,y}$ defined as boolean value, whether box on ith row, jth column is filled. And we can make propositional formula using nonogram conditions in addition to this literals. For example, if ith row condition is '2 1' and the column size is 5, then we know $(x_{i,1}\&x_{i.2}\&-x_{i,3}\&x_{i,4}\&-x_{i,5})$ can be true and $(x_{i,1}\&x_{i,2}\&-x_{i,3}\&-x_{i,4}\&x_{i,5})$ also be true. We lists all this formula that meet the condition and connect each by OR operator.(Because only one of these is true) then we are done for one row. For each row and column, connecting it by AND operator(because all conditions must be true), do it again. Using this method, we can make input string! The detail implementation will be showed in later.

TRANSFORM ENCODED STRING TO FORMULATREE AND CONVERT IT TO CNF After making formula string, we can using FormulaTree structure which is implemented for previous problem. We just use it and can easily convert to CNF. Note that the encoded string has operators with OR and AND. So, in the function *'ConvertToCNF'* we don't need to have steps, *'EquivFree', 'ImpliFree', 'ConvertToNNF'*.

PRINT CNF FORMULATREE AS MINISAT INPUT FORMAT Now, we should make minisat input to solve SAT. Suppose our CNF is (x1 | -x5 | x4) & (-x1 | x5 | x3 | x4) & (-x3 | x4). Then, the minisat input for this CNF will be:
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
A 3rd number of first line is same as the number of literals and 4th is the number of AND terms in CNF. And each line after first is composed of OR terms. If current term is "x1 | -x5 | x4", then corresponding input is "1 -5 4 0". Zero(0) signifies end of term. We can make this minisat input using recursive function on CNF FormulaTree. You can see details in implementation section and confirm it might generate correct minisat input.

EXECUTE MINISAT AND PRINT RESULTS Just use *'system'* function to execute minisat. Next, we can see results of that and with searching them, print the solution of puzzle easily.

2.3 IMPLEMENTATION

I used FormulaTree structure used in previous problem(CNF converter), as named *"FormulaTree.h"* and main sequence of program is implemented in c++ file, *"nonogram.cc"*. In this

section, i will show only two pseudocodes about SAT encoding and printing CNF as minisat input.(Because other implementations are almost mentioned in previous problem and appropriately discussed on approach section, we can implement them so easy)

ENCODE INPUT TO STRING AS SAT See below pseudocode.

Algorithm 12: Encoding

```
Data: current row(or column) number, current nonogram condtions, literal states
Result: Encoded string
begin
    if no more literals to indicate state then
        return;
    end
    if current condition cannot be satisfied then
        return;
    end
    update literal states to satisfy current condition;
    if all conditions are met then
        add this literal states to output string;
    end
    Encoding(current row(or col) number, next nonogram conditions, literal states);
    turn literal state back not to have updated states;
end
```

This Encoding function(recursive) will called for all rows and columns.

PRINT CNF FORMULATREE AS MINISAT INPUT FORMAT See below pseudocode.

Algorithm 13: PrintMinisatInput

```
Data: A CNF FormulaTree T
Result: Print minisat input
begin
    if Root token of T is AND then
        PrintMinisatInput(LeftChild);
        PrintMinisatInput(RightChild);
    else
        if Root token of T is OR then
            Print this term(T) recursively in prefix notation.
        else
            Print literal;
        end
    end
end
```

2.4 TEST AND RESULTS

Compiling with g++, the program existed in 'nonogram' folder with its code. Using command "./nonogram [data-file-name]", you can execute this program anytime for test and i tested with 2 testcases as follows:

1. **data1.in**
   6
   2 1
   1 3
   1 2
   3
   4
   1
   1
   5
   2
   5
   2 1
   2

2. **data2.in**
   2
   6
   3
   2 1
   1
   2
   1
   2
   0
   0

The results of each case are as shown:

1. ##...#
   .#.###
   .#.##.
   .###..
   .###
   #.
   ...#..

2. .###..
   ##.#..

I confirmed this results are all correct. You can test more complicated cases. But it can take more bigger time to generate results and be killed by OS. This is Because of trying to search all cases(usually called 'backtracking' method). So, i am willing to try to make time-efficient solver in future.