CS402
Introduction to Logic in Computer Science
Coursework 2: Normal Forms, Validity, and Satisfiability

20120696 이성민

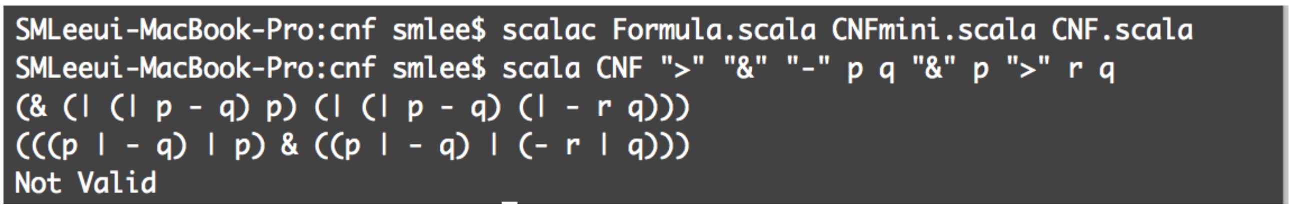
0. basic informations

Programming language: Scala
Executing environment: Mac OS X

1. Conjunctive Normal Form

1) Build and run

```
$ scalac Formula.scala CNFmini.scala CNF.scala
$ scala CNF ">" "&" "-" p q "&" p ">" r q
```



```
SMLeeui-MacBook-Pro:cnf smlee$ scalac Formula.scala CNFmini.scala CNF.scala
SMLeeui-MacBook-Pro:cnf smlee$ scala CNF ">" "&" "-" p q "&" p ">" r q
(& (| (| p - q) p) (| (| p - q) (| - r q)))
(((p | - q) | p) & ((p | - q) | (- r | q)))
Not Valid
```

2) Codes

cnf\Formula.scala
    declare classes representing formula, and it's utility objects
cnf\CNFmini.scala
    exist function that checks the validity of CNF
cnf\CNF.scala
    main converter of formula to CNF

cnf\Formula.scala

```
===== Operator case classes =====
trait Operator {
    val op: String
}
case object And extends Operator
...
case object Equiv extends Operator

===== Formula case classes =====
===== Terminal, Nonterminal, Nothing(use for unary operator “-“) =====
trait Formula
case class Terminal(name: String) extends Formula
case object Nothing extends Formula
case class NonTerminal(var left: Formula, op: Operator, var right: Formula)
extends Formula {
===== insert(f: Formula) insert formula to empty left or right child =====
    def insert(f: Formula) = {...}
}

===== FormulaUtil =====
===== Several utility method for Formula =====
object FormulaUtil {
    def insert(f: Formula, n: NonTerminal) = {...}

===== Print Formula in prefix order =====
    def prefix(f: Formula): String = {...}
===== Print Formula in infix order =====
    def infix(f: Formula): String = {...}
}

===== class NonTerminalList =====
===== use for stack to create Formula from input string list =====
class NonTerminalList {
    var nextList: List[NonTerminal] = List()
...
    def append(f: NonTerminal) = {...}
    def pop: NonTerminal = {...}
}
```

cnf\CNFmini.scala

```
===== object CNFmini =====
===== most of function used for miniSAT =====
===== exist CNF formula’s validity checking method =====
object CNFmini {
...
===== checkValid(cNF: Formula): Boolean =====
===== check CNF formula’s validity by checking whether each clause has at least
one complementary pair =====
    def checkValid(cNF: Formula): Boolean = {
===== change it to miniSAT input form(integer form) =====
      val miniSAT: List[List[Int]] = cNFToMiniSAT(cNF)
        val varNum: Int = idMap.size
===== array for check each clause =====
        var varTrace: Array[Array[Boolean]] = Array()
        var validity: Boolean = true

        for (lst <- miniSAT) {
            var tempValidity: Boolean = false
            varTrace = Array()
            for (i <- 0 to varNum - 1) varTrace = varTrace :+ Array(false, false)
            for (el <- lst) {
===== check “-a” and “a” exists =====
                if (el > 0) varTrace(el - 1)(0) = true
                else varTrace(-el - 1)(1) = true
            }
===== if both “-a” and “a” exists in clause, it is valid clause =====
            for (i <- 0 to varNum - 1) if(varTrace(i)(0) && varTrace(i)(1))
tempValidity = true
            validity = validity && tempValidity
        }
        validity
    }
}
```

cnf\CNF.scala

```
===== CNF object =====
===== main object to convert formula to CNF =====
object CNF {
    def main(args: Array[String]) {
===== convert input string Array to Formula =====
        val formula: Formula = toFormula(args.toList)
===== convert Formula -> CNF =====
        val cNF: Formula = formToCNF(formula)
===== print CNF as prefix and infix notation =====
        println(FormulaUtil.prefix(cNF))
        println(FormulaUtil.infix(cNF))
===== check CNF formula’s validity =====
        if(CNFmini.checkValid(cNF)) println("Valid") else println("Not Valid")
    }

===== Formula -> CNF = Formula -> implication elimination -> double negation
elemination -> CNF =====
    def formToCNF(formula: Formula): Formula =
nNFToCNF(imFeToNNF(formToImFe(formula)))

===== input string Array to Formula =====
    def toFormula(argval: List[String]): Formula = {
        var args: List[String] = argval
        if(args.isEmpty) System.err.println("no list argument")
===== root formula, initialize nextNode: NonTerminalList(place to insert next new
node)=====
        var root: Formula = null

        args.head match {
            case "&" => root = NonTerminal(null, And, null)
            case "|" => root = NonTerminal(null, Or, null)
            case ">" => root = NonTerminal(null, Implic, null)
            case "<" => root = NonTerminal(null, RevImp, null)
            case "=" => root = NonTerminal(null, Equiv, null)
            case "-" => root = NonTerminal(Nothing, Not, null)
            case s: String => {
===== if start with terminal, it’s end of formula, finish =====
                root = Terminal(s)
                CNFmini.addNewId(s)
                return root
            }
        }
        args = args.tail

        var nextNode: NonTerminalList = new NonTerminalList
        nextNode.append(root.asInstanceOf[NonTerminal])
```

===== start constructing formula =====

```
      while(!args.isEmpty){
           args.head match {
               case "&" => {
                   val newNode = NonTerminal(null, And, null)
                   nextNode.pop.insert(newNode)
                   nextNode.append(newNode)
               }
               ...
               case s: String => {
                   val newNode = Terminal(s)
                   CNFmini.addNewId(s)
                   nextNode.pop.insert(newNode)
               }
           }
           args = args.tail
        }
        root
    }
```

===== formToImFe(formula: Formula): Formula. convert Formula to implication, reverse-implication, equivalence elminated formula =====

```
    def formToImFe(formula: Formula): Formula = formula match {
        case Terminal(s) => formula
        case NonTerminal(_, Not, r) => NonTerminal(Nothing, Not, formToImFe(r))
        case NonTerminal(l, Implic, r) => NonTerminal(NonTerminal(Nothing, Not, 
formToImFe(l)), Or, formToImFe(r))
        case NonTerminal(l, RevImp, r) => formToImFe(NonTerminal(r, Implic, l))
        case NonTerminal(l, Equiv, r) => formToImFe(NonTerminal(NonTerminal(l, 
Implic, r), And, NonTerminal(r, Implic, l)))
        case NonTerminal(l, op, r) => NonTerminal(formToImFe(l), op, 
formToImFe(r))
    }
```

===== imFeToNNF(imFe: Formula): Formula. convert imFe to double negation elminated formula =====

```
    def imFeToNNF(imFe: Formula): Formula = imFe match {
        case Terminal(s) => imFe
        case NonTerminal(_, Not, r1) => r1 match {
            case NonTerminal(_, Not, r2) => imFeToNNF(r2)
            case NonTerminal(l2, And, r2) => 
imFeToNNF(NonTerminal(NonTerminal(Nothing, Not, l2), Or, NonTerminal(Nothing, Not, 
r2)))
            case NonTerminal(l2, Or, r2) => 
imFeToNNF(NonTerminal(NonTerminal(Nothing, Not, l2), And, NonTerminal(Nothing, 
Not, r2)))
            case _ => imFe // -p, -q etc.
        }
        case NonTerminal(l, And, r) => NonTerminal(imFeToNNF(l), And, 
imFeToNNF(r))
        case NonTerminal(l, Or, r) => NonTerminal(imFeToNNF(l), Or, imFeToNNF(r))
    }
```

```
===== distr(eta1: Formula, eta2: Formula): Formula, nNFToCNF(nNF: Formula):
Formula =====
===== convert double negation, implication, reverse-implication, equivalence
eliminated formula to CNF =====
   def distr(eta1: Formula, eta2: Formula): Formula = {
        (eta1, eta2) match {
            case (NonTerminal(eta11, And, eta12), e2) => NonTerminal(distr(eta11,
e2), And, distr(eta12, e2))
            case (e1, NonTerminal(eta21, And, eta22)) => NonTerminal(distr(e1,
eta21), And, distr(e1, eta22))
            case (e1, e2) => NonTerminal(e1, Or, e2)
        }
    }


    def nNFToCNF(nNF: Formula): Formula = nNF match {
        case Terminal(s) => nNF
        case NonTerminal(_, Not, r) => nNF // -p, -q etc.
        case NonTerminal(l, And, r) => NonTerminal(nNFToCNF(l), And, nNFToCNF(r))
        case NonTerminal(l, Or, r) => distr(nNFToCNF(l), nNFToCNF(r))
    }

}
```

1. Nonogram as SAT

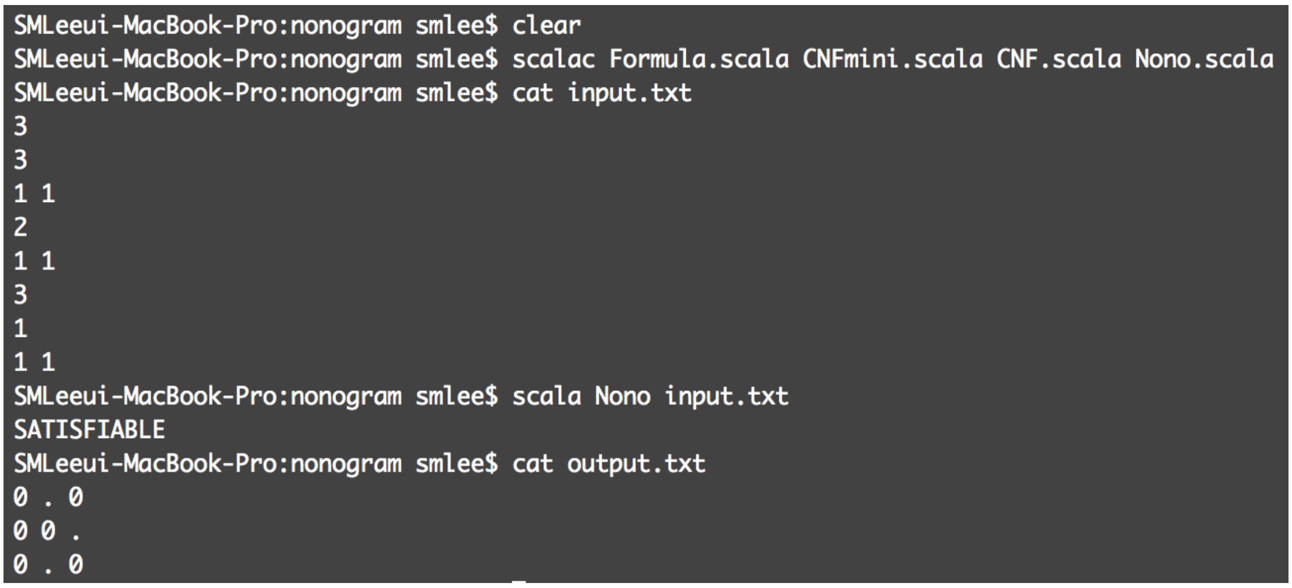
0)

In this "Nonogram as SAT" project, my solution has problem and return wrong solution for some Nonogram input.
In this report, I'll explain
- the algorithm that I used
- specific nonograms that can be solved by this algorithm
- problem in this algorithm
- way to solve the problem(but didn't materialized)

1) Build and run

```
input file name : "input.txt"
output file name : "output.txt"
```

```
$ scalac Formula.scala CNFmini.scala CNF.scala Nono.scala
$ scala Nono input.txt
```



```
SMLeeui-MacBook-Pro:nonogram smlee$ clear
SMLeeui-MacBook-Pro:nonogram smlee$ scalac Formula.scala CNFmini.scala CNF.scala Nono.scala
SMLeeui-MacBook-Pro:nonogram smlee$ cat input.txt
3
3
1 1
2
1 1
3
1
1 1
SMLeeui-MacBook-Pro:nonogram smlee$ scala Nono input.txt
SATISFIABLE
SMLeeui-MacBook-Pro:nonogram smlee$ cat output.txt
0 . 0
0 0 .
0 . 0
```

2) Codes

nonogram\Formula.scala
    declare classes representing formula, and it's utility objects
nonogram\CNFmini.scala
    utilities for constructing miniSAT input file
nonogram\CNF.scala
    converter of formula to CNF
nonogram\Nono.scala
    main constructer and solver converting Nonogram problem input to miniSAT input form

Since, nonogram\Formula.scala, nonogram\CNF.scala has been explained on 1., only explain about nonogram\Nono.scala and rest part of nonogram \CNFmini.scala

nonogram\CNFmini.scala

```
object CNFmini {
##### idMap: Map[String, Int]. map between terminal in Formula and identifier for
miniSAT #####
    var idMap: scala.collection.immutable.Map[String, Int] =
scala.collection.immutable.Map[String, Int]()
##### miniSAT: List[List[Int]]. column list of miniSAT input #####
    var miniSAT: List[List[Int]] = List()
    var idCounter = 1

##### adding mapping between new terminal and new id #####
    def addNewId(name: String) {
        if (!idMap.contains(name)) {
            idMap = idMap + (name -> idCounter)
            idCounter = idCounter + 1
        }
    }

##### cNFToMiniSAT(cNF: Formula): List[List[Int]]. #####
##### convert CNF formula to miniSAT input form #####
    def cNFToMiniSAT(cNF: Formula): List[List[Int]] = cNF match {
        case Terminal(s) => List(List(idMap(s)))
        case NonTerminal(_, Not, r) => r match {
            case Terminal(s) => List(List(-idMap(s)))
        }
        case NonTerminal(l, Or, r) => List(cNFToMiniSAT(l)(0) ::: cNFToMiniSAT(r)(0))
        case NonTerminal(l, And, r) => cNFToMiniSAT(l) ::: cNFToMiniSAT(r)
    }

##### miniSATGen(filename: String, cNF: Formula). #####
##### print out miniSAT input with comment for information about mapping terminal and
id #####
    def miniSATGen(filename: String, cNF: Formula) {
        val writer = new PrintWriter(new File(filename + ".in"))
        miniSAT = cNFToMiniSAT(cNF)
        var column = “"
##### miniSAT comment about mapping terminal and id #####
        ListMap(idMap.toSeq.sortBy(_._1):_*) foreach {case (key, value) => column =
column + "c | " + key + " -> " + value + "\n" }
        writer.write(column)

        writer.write("p cnf " + idMap.size + " " + miniSAT.size + "\n")
        for (lst <- miniSAT) {
            var column = ""
            for (el <- lst) column = column + " " + el
            writer.write(column + " 0\n")
        }
        writer.close()

    }

    def checkValid(cNF: Formula): Boolean = {...}
}
```

nonogram\Nono.scala

```
##### Nono object #####
object Nono {
##### def main(args: Array[String] = Array(“input.txt")) #####
##### Nono object’s main method #####
    def main(args: Array[String] = Array("input.txt")) {
##### get input as int list and parse #####
        val inputList = Source.fromFile(args(0)).toList.map(_.toInt - 48)
##### size parsing #####
        val sizeX = inputList(0)
        val sizeY = inputList(2)
##### conditions list for row and column as List[List[Int]] #####
        var rowList: List[List[Int]] = Nil
        var colList: List[List[Int]] = Nil
        val rowsAndColumns = inputList.tail.tail.tail.tail
        var rowcount = 0
        var lineList: List[Int] = List()
##### parsing conditions #####
        for(i <- rowsAndColumns) i match {
            case -38 => {
                if (rowcount < sizeX) rowList = rowList :+ lineList
                else colList = colList :+ lineList
                rowcount = rowcount + 1
                lineList = List()
            }
            case -16 => {}
            case n => lineList = lineList :+ n
        }

##### adding new terminals for CNF of Nonogram ####
##### m_(i,j)   := terminal representing matrix(i, j) is filled or empty #####
##### pr_(i,j,k) := terminal representing the pivot, starting position, of ith
condition block of jth row is (j, k) #####
##### for example, if 2nd row has condition(2, 1) and first block, which length is 2,
starts at (2,1) in matrix, pr_(1,2,1) = true #####
##### same as pc_(i,j,k) := pivot terminal for column condition block #####
################################################################################
       for(i <- 1 to sizeX){
            for(j <- 1 to sizeY) CNFmini.addNewId("m"+i+j)
        }
        for(i <- 1 to sizeX){
            for(j <- 1 to rowList(i-1).length){
                for(k <- 1 to sizeY) CNFmini.addNewId("pr"+j+i+k)
            }
        }
        for(i <- 1 to sizeY){
            for(j <- 1 to colList(i-1).length){
                for(k <- 1 to sizeX) CNFmini.addNewId("pc"+j+k+i)
            }
        }

        val formula: Formula = createFormula(sizeX, sizeY, rowList, colList)
        val cnf: Formula = CNF.formToCNF(formula)
        CNFmini.miniSATGen("Nono", cnf)

        "minisat -verb=0 Nono.in Nono.out".!

        val outputList = Source.fromFile("Nono.out").toList.tail.tail.tail.tail
##### create formula from parsed input #####
        val formula: Formula = createFormula(sizeX, sizeY, rowList, colList)
##### convert formula to CNF #####
        val cnf: Formula = CNF.formToCNF(formula)
##### create miniSAT input file, name “Nono.in” #####
        CNFmini.miniSATGen("Nono", cnf)
##### run process executing miniSAT program with input “Nono.in”, returns output
“Nono.out” #####
        "minisat -verb=0 Nono.in Nono.out".!
##### read “Nono.out”, visualize it. #####
        val outputList = Source.fromFile("Nono.out").toList.tail.tail.tail.tail
        visualize(outputList, sizeX, sizeY)
    }

##### visualize(out: List[Char], sizeX: Int, sizeY: Int). #####
##### using out, result of miniSAT, visualize it with file name “output.txt” #####
    def visualize(out: List[Char], sizeX: Int, sizeY: Int) {
        val t: String = "0 "
        val f: String = ". "
        val writer = new PrintWriter(new File("output.txt"))
        var counter: Int = 0
        var curr: Boolean = true
        var line: String = ""
        val iter = out.iterator
        while(counter < sizeX*sizeY){
            iter.next match {
                case ' ' => {
                    line = line + (if(curr) t else f)
                    counter = counter + 1
                    curr = true
                    if (counter % sizeY == 0) line = line+"\n"
                }
                case '-' => curr = false
                case _ => ()
            }
        }
        writer.write(line)
        writer.close()
    }
```

```
##### create formula from conditions of nonogram #####
def createFormula(sizeX: Int, sizeY: Int, rowList: List[List[Int]], colList:
List[List[Int]]): Formula = {
        var wholeFormulaList: List[Formula] = List()
        var rowNumber = 1
        var colNumber = 1

##### for all blocks pivot only exist one in each row. #####
##### (for 2nd condition block in 1st row) add constraint as
##### -pr(2,1,1) and -pr(2,1,2) and … and -pr(2,1,4) and pr(2,1,5) and -pr(2,1,6) and
... and -pr(2,1,10) #####
        //add one pivot for one block
        var block3: List[Formula] = List()
        for(i <- 1 to sizeX){
            if (rowList(i-1)(0) != 0){
                var block2: List[Formula] = List()
                for(j <- 1 to rowList(i-1).length){
                    var block1: List[Formula] = List()
                    for(k <- 1 to sizeY) block1 = NonTerminal(Nothing, Not,
Terminal("pr"+j+i+k)) :: block1
                    for(k <- 1 to sizeY) block2 = wrapAnd(block1.updated(k-1,
block1(k-1).asInstanceOf[NonTerminal].right)) :: block2
                }
                block3 = wrapOr(block2) :: block3
            }
        }
        val rowblocks: Formula = wrapAnd(block3)
##### do same for each column. #####
        block3 = List()
        for(i <- 1 to sizeY){
            if (colList(i-1)(0) != 0){
                var block2: List[Formula] = List()
                for(j <- 1 to colList(i-1).length){

                    var block1: List[Formula] = List()
                    for(k <- 1 to sizeX) block1 = NonTerminal(Nothing, Not,
Terminal("pc"+j+k+i)) :: block1
                    for(k <- 1 to sizeX) block2 = wrapAnd(block1.updated(k-1,
block1(k-1).asInstanceOf[NonTerminal].right)) :: block2
                }
                block3 = wrapOr(block2) :: block3
            }
        }
        val colblocks: Formula = wrapAnd(block3)

##### create formula for each row and it's condition using 'createRowFormula method
        for (rL <- rowList) {
            if(rL(0) != 0) wholeFormulaList = createRowFormula(rowNumber, sizeY,
rL) :: wholeFormulaList
            rowNumber = rowNumber + 1
        }
        for (cL <- colList) {
            if(cL(0) != 0) wholeFormulaList = createColFormula(colNumber, sizeX,
cL) :: wholeFormulaList
            colNumber = colNumber + 1
        }
##### wrap all formula with '&' and return it #####
        wrapAnd(List(rowblocks, colblocks, wrapAnd(wholeFormulaList)))
    }
```

```
##### createRowFormula(...): Formula. Return formula of one row condition #####
def createRowFormula(rowNumber: Int, rowSize: Int, condList: List[Int]): Formula = {

##### block pivot should be ordered by condition, and every pivot should not be
adjoint. create formula for available pivot index #####
##### createPivotIndex(5, 2) = {(1,3),(1,4),(1,5),(2,4),…(3,5)} #####
##### wrap up conditions with or #####

        val pivotCondList = (wrapOr(createPivotIndex(rowSize, condList.length)
map(_.zipWithIndex map({ case (a, b) => Terminal("pr" + (b + 1) + rowNumber + a)}))
map(wrapAnd(_))))

##### From blockList0~4, blockCondList, create formula list of each block #####
##### for example, if 2X1 block(1st block) is placed in (2,3),(2,4), then
##### ‘p(1,2,3) and m(2,3) and m(2,4) and -m(2,2) and -m(2,5)’ #####

        val blockList0 = condList map ((1 to rowSize).toList.sliding(_).toList map (_
map ("m" + rowNumber + _)))
        val blockList1 = blockList0.zipWithIndex map ({case (block, index) => block
map ({case (m) =>
            val withoutNeg = "pr" + (index + 1) + m(0).tail :: m
            var withNeg: List[String] = withoutNeg
            if (withoutNeg.tail.head.last.toInt-48 != 1) withNeg = ("-" +
withoutNeg.tail.head.substring(0, 2) + (withoutNeg.tail.head.last.toInt-48 - 1)) ::
withNeg
            if (withoutNeg.last.last.toInt-48 != rowSize) withNeg = ("-" +
withoutNeg.last.substring(0, 2) + (withoutNeg.last.last.toInt-48 + 1)) :: withNeg
            withNeg})})
        val blockList2 = blockList1 map (_ map (_ map ({case (s) =>
            if(s.head != '-') Terminal(s)
            else NonTerminal(Nothing, Not, Terminal(s.tail))})))
        val blockList3 = blockList2 map (_ map (wrapAnd(_)))
        val blockList4 = blockList3 map (wrapOr(_))
        val blockCondList = wrapAnd(blockList4)

##### wrap up all conditions with and #####
        wrapAnd(List(blockCondList, pivotCondList))
    }

##### method doing same for one column #####
    def createColFormula(colNumber: Int, colSize: Int, condList: List[Int])
       : Formula = {
        ...
        wrapAnd(List(blockCondList, pivotCondList))
    }
```

```
##### createPivotIndex(size: Int, cnum: Int): List[List[Int]]. #####
##### Method for creating available pivot index #####
    def createPivotIndex(size: Int, cnum: Int): List[List[Int]] = {
        var ret: List[List[Int]] = List()
        if (size < cnum*2-1) System.err.println("UNSATISFIABLE")
        var pivotIndex: List[Int] = (1 to cnum).toList map (_ * 2 - 1)
        var i = cnum-1
        while(pivotIndex(0) <= size - 2*(cnum-1)){
            if (pivotIndex(cnum-1) <= size){
                ret = pivotIndex :: ret
                i = cnum-1
            }
            pivotIndex = pivotIndex.updated(i, pivotIndex(i) + 1)
            for(j <- (i+1) to cnum-1) {
                pivotIndex = pivotIndex.updated(j, pivotIndex(j-1) + 2)
            }
            i = i-1
        }
        ret
    }

##### utility function warpOr. warp up formulas in list with or #####
    def wrapOr(formulaList: List[Formula]): Formula = {
        formulaList.length match {
            case 1 => formulaList(0)
            case _ => {
                val splitList = formulaList splitAt formulaList.length/2
                NonTerminal(wrapOr(splitList._1), Or, wrapOr(splitList._2))
            }
        }
    }

##### utility function warpAnd. warp up formulas in list with and #####
    def wrapAnd(formulaList: List[Formula]): Formula = {
        formulaList.length match {
            case 1 => formulaList(0)
            case _ => {
                val splitList = formulaList splitAt formulaList.length/2
                NonTerminal(wrapAnd(splitList._1), And, wrapAnd(splitList._2))
            }
        }
    }
}
```

3) Algorithm limitation

Above algorithm has problem. Since constructed conditions are
(1) one pivot for each block
(2) pivot + filled block + adjoint cell should be empty,
it only can handle nonograms such as every space between two block is 1 or 2. If there are several empty cell between two block, then constructed formula doesn't have any information about those empty cells, and may fill those cells.

Also, "one pivot for each block" condition has a lot of conjunction, this algorithm cannot hold with non (n >~ 5, 6) with condition that has several block in one row or column, since on making cnf algorithm, those conjunctions will flood by method 'distr', and will cause "OutOfMemoryError".

First problem can be solved by adding 'pr(i, j, k) and m(j, k)' and 'pc(i, j, k) and m(j, k)' to link all pivot and cells. But it will make more flooding while converted to CNF.