Discrete Mathematics | Homework 2

21800053 Kim Gyurim

1. Problem Definition

1-1. DNF converter

A Disjunctive Normal Form is the propositional formula that is consist of disjunction of conjunctions. The DNF converter must receive propositional logic which is consist of "propositional value: a_n", "and", "or", and "not". The received input must be stored as tree-like structure and if it is not a propositional formula or, the input has wrong syntax, it has to print out error message.

The input that is stored in tree-like structure must be converted into NNF. When the stored values are fully converted to NNF from, they must be changed again to DNF through Distributive Law. The output is DNF that is equivalent as input propositional formula. It represents output as "n" that is a_n (n > 0), "-n" that is not a_n (n > 0), "space" that is "and", and "Enter" that is or.

1-2. DNF Solver

A DNF solver must find solution of DNF and represent it. If the formula is unsatisfiable, the program should print "UNSAT". If the formula is satisfiable, then the program should print out only one solution according to the following rules:

If a_i is True for the solution, print "i". If a_i is False for the solution, print "-i" ("i" is an integer > 0).

2. Solving Problem

2-1. DNF converter

(1) Receiving input & Converting formula into NNF $\rightarrow$ Same as PA2

(2) Converting to DNF: To convert NNF to DNF, the program should downgrade all conjunction recursively as possible. By using "Distributive Law *(P and (Q or R))* $\leftrightarrow$ *((P and Q) or (P and R))*", the program convert conjunction to disjunction.

```
//convert to DNF
Tree DNF (Tree tree){
  switch(tree -> type){
    case Atom, Not;
        return tree;
    case Or:
        return (DNF(tree.left t) or DNF(tree.left t));
    case And: Distribute(DNF(tree.left t), DNF(tree.right)); }}
```

```
//Distributive Law
Tree Distribute(t1, t2){
  if(t1.type = Or)
    return Distribute(t1.left t, t2) or Distribute(t1.right t, t2);
  if(t2.type = Or)
    return Distribute(t1.left t, t2) or Distribute(t1.right t, t2);
  return t1 and t2;
  }
```

2-2. DNF Solver

(1) Receiving DNF propositional logic

The program receives DNF as string and store by line. Each line is handed over to function that checks satisfiability. If the line is satisfiable, then the program stores the propositional value as satisfiable value ,else the program stores it as unsatisfiable value. If all lines are unsatisfiable, then program prints "UNSAT" and end.

(2) Print solutions

The program checks stored satisfiable value first. If there are duplicate values, then program removes the rest, leaving only one number. After that, the program checks stored unsatisfiable value that there are no duplicate numbers in both unsatisfiable values and satisfiable values. Then prints all remain value.

3.Prove Validation

input	output		input	output		input	output
(and a2 (not a2))	2 -2 0 UNSAT		(or (and a1 a2 a3 (not a1)) (and a4 (not a4) a5 (not a6)) (and a7 a9 (not a3)))	1 2 3 -1 4 -4 5 -6 7 9 -3 0 7 9 -3 1 2 4 5 -6		(or (and a1 a2 a3 a1) (and a4 a4 a5 a6) (and a7 a8 a9 a3))	1 2 3 1 4 4 5 6 7 8 9 3 0 1 2 3 4 5 6 7 8 9
(or a2 (not a2))	2 -2 0 2					(or a1 (not (or (and (or a2 a3)) a4)))	ERROR: Not right propositional logi

4. Discussion

It took a lot of time complexity to find duplicate numbers. The program uses multiple for loop too find duplicate number $\rightarrow$ n^2. To reduce unnecessary calculations to a minimum, the *i-index* checks from *0* to *n* and the *j-index* checks from *n* to *i* when calculating with a for-loop. However, it still seems to cost too much unnecessary calculation. Also, my solution prints unsorted values. The sorted values would have required more computations, but it would have been easier to check. Or, it might be able to save space, save time, if I sort the values at first time. Anyway, I think my solution lacks many things and there will be better ways.