ACE cw2 report

Junyu LIU 20216355

November 2021

1 Introduction

For ACE coursework 2, the algorithm designs and evaluations are explained in the following sections. Section 2 and 3 introduce the algorithm design ideas, and section 4, 5, and 6 present some basic evaluations.

2 Data Structure Choice

For the main data structure, I choose tree to store the tokens of literal strings. Since the whole LD formula could be divided into several recursive patterns, in which a propositional operator could be seen as a node and its left and right literal could be seen as left and right children node respectively. Thus, a recursive construction of the tree could be a appropriate data structure to store the LD formula.

3 Algorithm Design and Implementation

The algorithm contains 5 major steps, which are described in detail as follows.

3.1 Read input into a infix-represented list

The user input is handled by using a Scanner. After reading the input, since the return of Scanner is a string, I use StringTokenizer to separate the literals by "". The result is stored in a ArrayList, for its convenience to handle accessing, comparing, popping and adding elements.

3.2 Convert infix-represented list into suffix-represented list

Aiming to store the literals into a tree, I found that the current ArrayList is not optimal. This is because I want to use a stack to help creating the tree, but the infix-represented list (a literal + a propositional logic operator + a literal) cannot be directly traversed on the stack. Thus, I convert the infix-represented list into a suffix-represented list. Detailed process is presented as follows.

A new stack and an empty Arraylist (served as the final suffix-presented list) is created. Each time one literal is get from the infix-represented list.

- If the literal is one of "[", "&", "|", not, "$->$", push the literal onto the stack;
- Else if the literal is "]", pop out the literal stored in the stack and add into the final suffix-presented list one by one, until the popped out one is "[";
- Else add the literal into the final suffix-presented list.

After turning into the suffix-presented list, the list follows the (a literal + a literal + a propositional logic operator) pattern, which is useful for the following operation (putting into a stack).

3.3 Store the suffix-represented list into a tree

A new stack is created to help creating the tree. Each time one literal is get from the suffix-represented list.

- If the literal is neither one of the "&", "|", not, "$->$", create a new node which has the literal as its literal value, and push the newly created node onto the stack;
- Else if the literal is " ", create a new node which has the literal as its literal value; pop a node from the stack to set as the right child of the newly created node; set the left child of the newly created node as null; then push the newly created node onto the stack.
- Else, create a new node which has the literal as its literal value; pop a node from the stack to set as the right child of the newly created node; pop another node from the stack to set as the right child of the newly created node; then push the newly created node onto the stack.

After all literals are get from the suffix-represented list, the stack should only stores one node. Then pop this node out as the root of the tree; create and return the tree.

3.4 Traverse tree and change transferable nodes

The tree is traversed from top to bottom (root to leaf) because this could prevent both checking upwards and downwards, which might be messy to implement. If a node is checked as transferable, then change the corresponding nodes. However, traverse once might not be enough, since the afterwards changing might create new transferable nodes which previous check will miss. Thus, after traversing and changing the nodes, I traverse it again to check whether there is still transferable nodes. If there is, do the traverse again and check for the number of transferable nodes, until the number of transferable node is 0.

3.5 Traverse tree and output the final output

The tree is traversed from bottom up to merge into a whole string, as final output.

4 Correctness of Program

The correctness of this program could be evaluated by evaluating sections 3.2, 3.3, 3.4, and 3.5 respectively. For 3.2, 3,3 and 3.5, the algorithm follows specific rules, like converting into suffix-representation or preorder traversal, thus could be considered as correct. For 3.4, the correctness is confirmed by checking and changing all possible transferable patterns in the tree, and making sure that there are no transferable patterns in final. Overall, the correctness could be satisfied by the above reasons.

5 Time Complexity of Program

The time complexity is $O(n^2)$, since steps 3.2, 3.3, 3.5 are $O(n)$ and 3.4 has $O(n^2)$ time complexity. The reason why 3.4 has $O(n^2)$ time complexity is that each traversal of the tree has $O(n)$ time complexity and the loop checking for each traversal also has $O(n)$ time complexity.

6 Testing of the Program

For the test, I first test all the primary situations (all the elementary test cases provided in the example.txt) to make sure that the program satisfy all these basic situations.

Next, for more complex situations, I created several complicated cases which include combining and nesting logical operations. All of the basic situations are included in these complex situations to ensure the comprehensiveness of the test.

For example $[[E(a,b)]->[mE(c,d)]]->[[W(b,c)]->[E(c,d)]]$;
$[[E(a,b)]|[F(c,d)]]|[\ [[E(a,b)]|[F(c,d)]]]$;
$[[E(a,b)][[W(b,c)]|[N(c,d)]]]->[[[E(a,b)]|[W(b,c)]][N(c,d)]]$.