COS30019 – Introduction to AI

Assignment 2 Research Report

TEAM COS30019_A02_T026: DANVERS FLETT (100501422), DYLAN FORSTER (2058979), ADAM RICHARDS (7634765)

Contents

1 Introduction

Truth Table processing of knowledge bases (KBs) is the most basic, “brute force” way for an inference engine to determine if a query is entailed by the KB. It lacks the elegance and efficiency of using Forward- or Backward-Chaining to prove entailment of a Horn-form KB but it has the advantage of not being limited to Horn-form clauses. Truth Tables can be used to evaluate any Boolean expression, no matter how complex or whatever form it is expressed as. In this assignment, we have found that Binary Expression Trees are an effective data structure for storing and evaluating Boolean expressions. By converting the Boolean expressions that are represented in infix notation to postfix, it is then straightforward to use a variation of the postfix evaluation algorithm to convert a postfix expression into a Binary Expression Tree. Binary Expression Trees can be evaluated in linear time, requiring no further nonlinear-time computation to produce a result. This is vital in Truth Table inference engines where the size of the truth table is exponential to the number of literals in the KB.

2 Binary Expression Tree

This general knowledge base solving application is based around a Binary Expression Tree data structure. The nodes in the tree consist of ***sentenceClass*** objects. This is an abstract class that is implemented with concrete types that are either literals or operators. The literals are the symbols that store a Boolean true or false value. The operators take one or two sentence objects and return a result according to the truth table for the values of the sentence objects it contains.

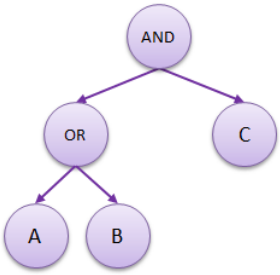

Figure 1 Example of a Boolean Binary Expression Tree (1)

When the ***eval()*** method of a ***sentenceClass*** object is called, if it is an operator it recursively calls the ***eval()*** methods of the objects it contains until a literal is found. The Boolean value returned by the node is the result of the full binary algebraic evaluation of all the operators and literals contained within the tree structure below the node.

2.1 Infix To Postfix Converter

The Horn clauses provided in the *test1.txt* file are in infix notation. This is the familiar mathematical notation that places the operator between the two literals upon which the operator is to be applied to obtain a result. This is easy for humans to read but difficult for a machine to automatically solve. Postifx notation, also known as Reverse Polish notation is much easier to automate as the order of operations is always from left to right and there are no parentheses (2). The use of a stack data structure allows a computer to store the running result and operands to be processed in the next operation.

In this application, the sentences of the knowledgebase are passed to the Infix To Postfix converter as a string containing all the symbols of that clause. The postfix expression is returned as a List of Strings, one string per symbol (literal or operator).

The following is a description of the Infix to Postfix algorithm (3)

1. Print operands as they arrive.

2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.

3. If the incoming symbol is a left parenthesis, push it on the stack.

4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.

5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.

6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.

7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.

8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

2.2 Creating a Binary Expression Tree

The algorithm for creating a Binary Expression Tree from clauses in postfix notation is basically the same as actually evaluating postfix clauses.

This is an algorithm for evaluating postfix expressions (4):

```
Initialize(Stack S)
x = ReadToken();  // Read Token
while(x)
{
     if ( x is Operand )
         Push ( x ) Onto Stack S.

     if ( x is Operator )
            {
            Operand2 = Pop(Stack S);
            Operand2 = Pop(Stack S);
            Evaluate (Operand1,Operand2,Operator x);
            }
     x = ReadNextToken();  // Read Token
}
```

Instead of evaluating the literals and operators, when an operator symbol is encountered, an operator node object is created containing the objects stored on the stack. Instead of keeping a “running result”, the subsequent objects created contain the objects previously created, building up a tree structure. Each expression object is created using sentence objects from the stack. After creation, the expression object is pushed to the stack and the procedure is repeated, until there are no more symbols to be processed in the original postfix expression.

The ***makeSentenceTree*** method in the ***truthTable*** class takes in a clause in infix notation, uses the Infix To Postfix converter to obtain a list of symbols (as Strings) in postfix order, and then runs the postfix evaluation algorithm on this list to create the a Binary Expression Tree.

3 Truth Table Processing

The application checks all sentences in the knowledgebase (KB) against all possible combinations of true and false assigned to every literal in the knowledgebase. Each possible combination of literal values is called a “model”. A knowledgebase has 2 ^ n possible models where n is the number of literals in the knowledgebase. The application iterates through every model, from zero to n-1 and at each iteration converts the model number to a binary string. Each “bit” of this string is assigned as a true (for “1”) or a false (for “0”) to each literal present in the KB.

3.1 Optimisations

Before checking the truth tables, a count of the number of true models is set to the maximum possible models for this KB (2 ^ n literals).

At each iteration of a possible model, the “ASK” query is first tested for truth. If it is false, the model is considered false and the count of possible true models is decremented by one. No further processing needs to take place for this model.

If the query is true, the program begins to evaluate sentences in the KB. Only one sentence in the KB needs to be false to falsify a model, so the program terminates checking for that model when it finds a false sentence. The count of possible true models is decremented by one and no further processing of this model occurs.

If all the sentences are true, the count of true models remains as-is.

In this way, the false models and false queries are eliminated and what is left is a count of all true models where the query is also true.

3.2 Debug Output

The program has a debug verison of the Truth Table inference method which displays all data relating to the checking of each model. It is invoked by using the “TTd” option at the command-line instead of the usual “TT”. It shows the input sentence strings, the ask query and uses the debug outputs of the sentence tree objects to verify that the tree has been created correctly and that the clauses are being evaluated correctly.

Below is an edited example of its output:

```
Model: 4  a:1 b:0 c:0
Query is  true
 {A1:  (A2: a:true & B2: b:false ):false => B1: c:false }:true
 (A1: ~b:false ):true
 <A1: a:true | B1:  (A2:  (A3: ~c:false ):true & B2: b:false ):false >:true
 [A1: b:false <=> B1: c:false ]:true
 Model is  true

...

Model: 7  a:1 b:1 c:1
Query is  true
 {A1:  (A2: a:true & B2: b:true ):true => B1: c:true }:true
 (A1: ~b:true ):false
 <A1: a:true | B1:  (A2:  (A3: ~c:true ):false & B2: b:true ):false >:true
 [A1: b:true <=> B1: c:true ]:true
 Model is false

Sentence Strings: [a&b=>c, ~b, a|~c&b, b<=>c]

Sentence Trees:
        [ ( a & b ) => c ]
        ~b
        ( a | ( ~c & b ) )
        [ b <=> c ]

Literals: 3 Sentences: 4

ask = a|~a
           YES: 1
```

4 Conclusion

Binary Expression Trees offer many options for optimisation in the processing of inference engine truth tables. A simple postfix evaluation acting on symbols represented as strings or simple objects would not scale well for KBs with large numbers of literals. Because the number of models of a KB is exponential to the number of literals, the processing of each model needs to be as efficient as possible. Algorithms that need to look up or assign literal values at the evaluation of each sentence will not have a linear execution time per model – this is what would occur with a conventional postfix evaluation were the expression represented as a string. Likewise, having to run the postfix evaluation on each sentence for each model check would be similarly inefficient.

The Binary Expression Tree solves this – each leaf node of the tree holds the value of a literal, and by asking the root node of the tree to return the result, the expression is evaluated “in place” with no searching of arrays or lists, and no execution of a postfix evaluation algorithm. The postfix evaluation only has to be run once for each sentence - when it is first added to the KB and before the actual truth table is processed.

In this program, some further optimisations were added to eliminate unnecessary evaluation – when a query or sentence in the KB is found to be false for a model, further processing of that model is terminated. An improved program may take this further, perhaps by re-ordering the list of sentences so that the simplest sentences are evaluated first. If any of these are false, the more complex sentences don't need to be checked.

5 References

1. **101 Computing.** Binary Expression Trees. *101 Computing.net.* [Online] 25 April 2017. [Cited: 25 May 2017.] http://www.101computing.net/binary-expression-trees/.

2. **Stone, Anthony.** Reverse Polish Notation. *Anthony Stone.* [Online] 14 March 2014. [Cited: 25 May 2017.] http://www-stone.ch.cam.ac.uk/documentation/rrf/rpn.html.

3. **Wolf, Carol E.** Infix to postfix conversion algorithm. *Carol E. Wolf.* [Online] [Cited: 25 May 2017.] http://csis.pace.edu/~wolf/CS122/infix-postfix.htm.

4. **c4learn.com.** Algorithm for Evaluation of Postfix Expression. *c4learn.com.* [Online] 2015. [Cited: 25 May 2017.] http://www.c4learn.com/data-structure/algorithm-evaluation-of-postfix-expression/.