PROJECT REPORT

December 16, 2021

Michael Humphrey
Colorado School of Mines
CSCI598 Program Verification and Synthesis

Introduction

Converting First Order Logic propositions to Conjunctive Normal Form (CNF) allows for easier solving of logical propositions and allows for faster evaluation of complex expressions by simplifying them to a more standardized form. For this reason, I built a CNF solver that converts complex First Order Logical expressions to CNF. This CNF solver handles many of the common connectives used in first order logic (Conjunctions, Disjunctions, Negations, Implications, and Biconditionals). These will be parsed in using standard representations of these connectives. For example:

- Conjunction: $\wedge, \&, AND$
- Disjunction: $\vee, |, OR$
- Negation: $\neg, !, NOT$
- Implication: $\Rightarrow, =>, IMPLIES$
- Biconditional: $\Leftrightarrow, <=>, IFF$

Each of these symbols may detected to make the solver more robust and able to operate on more propositions.

These connectives are expected to be in the following forms:

- Binary Connectives (Conjunction, Disjunction, Implication, Biconditional): $(\phi)f(\psi)$ where ϕ and ψ are the expressions of the connective, and f is the connective itself.
- Unary Connectives (Negation): $f(\phi)$ where f is the connective and ϕ is the operator.

Formally, the grammar that will be accepted by this CNF converter is:

$$\begin{array}{rcl}
\langle exp \rangle & \rightarrow & \langle term \rangle \\
 & | & \langle uop \rangle \langle exp \rangle \\
 & | & \langle exp \rangle \langle bop \rangle \langle exp \rangle \\
\langle uop \rangle & \rightarrow & \neg \\
\langle bop \rangle & \rightarrow & \wedge | \vee | \rightarrow | \Leftrightarrow
\end{array}$$

Additionally, for simplicity's sake, my CNF solver expects the inputs to be fully parenthesized, and outputs a simplified CNF version of this proposition. This CNF solver was written as a multi-threaded C program using best practices of C. This solver focuses on the performance and usability of the conversion to CNF.

Examples

As stated above, this CNF converter expects it's inputs to be fully parenthesized. Below are a few sample inputs that this program will accept:

- ((A) AND ((B) OR (C)))
- ((A) IFF ((B) IFF (C)))
- ((NOT (A)) OR ((B) AND (C)))
- ((A) $\rightarrow$ (B))
- $((A) \wedge ((B) \vee (\neg(C))))$

This converter will output the same expression in CNF in a succinct way that is easy to read, and an end-user will be able to easily decipher. Below are the above expressions converted to CNF.

- $(A) \wedge (B \vee C)$
- $(\neg C \vee \neg B \vee A) \wedge (\neg C \vee C \vee A) \wedge (B \vee \neg B \vee A) \wedge (B \vee C \vee A) \wedge (\neg A \vee \neg C \vee B) \wedge (\neg A \vee \neg B \vee C)$
- $(\neg A \vee C) \wedge (\neg A \vee B)$
- $(\neg A \vee B)$
- $(B \vee \neg C) \wedge (A)$

As can be seen above, the CNF propositions are outputted in a way that makes reading the propositions simple. Internally, the propositions are also stored in a data structure that would allow for very expansion into future work such as a full SAT solver.

Approach

This CNF solver uses a very basic algorithm with three major parts:

- Eliminate biconditional connectives from the expression.
- Eliminate implication connectives from the expression.
- Convert the expression to Negation Normal Form (NNF)
- Convert the NNF expression to CNF

Each of the algorithms to make these eliminations or conversions are implemented in pseudocode below.

Algorithm 1: ImpElim

```
Input: $E : \phi, \sigma, \psi$ ;                               // left, op, right
Output: $E' : \phi, \sigma, \psi$ ;                             // left, op, right
Function f($E$):
    if $term(E)$ then
        return $E$
    else if $\sigma = Implication$ then
        return $(f(\neg\phi) \vee f(\psi))$
    return $(f(\phi)\sigma f(\psi))$
return $f(E)$
```

Algorithm 2: BiconElim

```
Input: $E : \phi, \sigma, \psi$ ;                               // left, op, right
Output: $E' : \phi, \sigma, \psi$ ;                             // left, op, right
Function f($E$):
    if $term(E)$ then
        return $E$
    else if $\sigma = Biconditional$ then
        return $(f(ImpElim(\phi \Rightarrow \psi)) \wedge f(ImpElim(\psi \Rightarrow \phi)))$
    return $E(f(\phi)\sigma f(\psi))$
return $f(E)$
```

Algorithm 3: NNF

```
Input: $E : \phi, \sigma, \psi$ ;                               // left, op, right
Output: $E' : \phi, \sigma, \psi$ ;                             // left, op, right
Function f($E$):
    if $term(E)$ then
        return $E$
    else if $E = (\neg(\neg(E)))$ then
        return $f(E)$
    else if $E = (\neg(\phi \vee \psi))$ then
        return $f(\neg\phi) \wedge f(\neg\phi)$
    else if $E = (\neg(\phi \wedge \psi))$ then
        return $f(\neg\phi) \vee f(\neg\psi)$
    return $f(\phi\sigma\psi)$
return $f(E)$
```

Algorithm 4: CNF

```
Input: $E : \phi, \sigma, \psi$ ;                               // left, op, right
Output: $E' : \phi, \sigma, \psi$ ;                             // left, op, right
Function Dist($E_1, E_2$):
    if $E_1 = (\phi \wedge \psi), E_2 = \_\_$ then
        return $E(dist(\phi E_2) \wedge dist(\psi E_2))$
    else if $E_1 = \_\_, E_2 = (\phi \wedge \psi$ then
        return $(dist(E_1\phi) \wedge dist(E_1\psi))$
    return $(E_1 \vee E_2)$
```

Evaluation

This program was evaluated by timing it on a number of inputs, ranging from trivial computations to very complex expressions with up to 100,000 clauses in the finished CNF form. I decided to count the size of an expression as the number of clauses in CNF form as this was the most accurate way to measure it. the alternative, counting the number of expressions of the input expression may be inaccurate as certain expressions (biconditional) expand to two separate expressions. For this reason, the 'size' of an expression will refer to the number of clauses in the converted CNF expression.

The goal of the testing of this program was to determine primarily the runtime of the conversion to CNF. Beyond anecdotal measurement, a secondary goal was to determine if there was any correlation between the size of an expression and its runtime. The testing procedure for this evaluation was simple: measure the runtime and the input size of a number of different inputs run through the program, logging both the size and runtime.

The size of inputs was built to cover a wide range of expressions, ranging from a size of 1 clause to 100,000 clauses. This allowed for a large enough domain to accurately measure the performance of this CNF converter.

Below is a graph of the runtimes plotted against the size of the input for a sample of 20 inputs. The exact inputs used may be viewed in the input directory of the program.

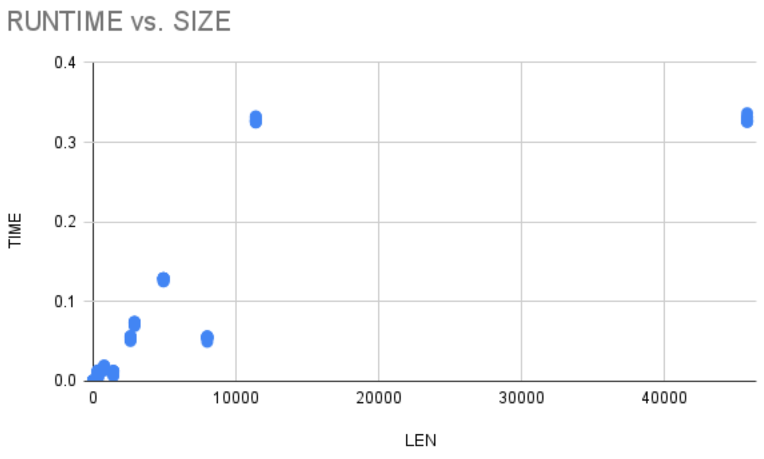

Figure 1: Runtime vs Size of Expression

As can be seen from the above graph, there is some correlation between the runtime of an expression and the size of the expression. Without outliers, this graph is roughly linear. However, adding in the outliers, the graph does not fit to a trendline for any simple complexity. As such, the results of the 180 points on the graph above are relatively inconclusive, but there are certainly parallels to draw between the size of an expression and the runtime to convert to CNF.

Another source of potential error in the graph above could be that some expressions are simpler to compute than others; they have fewer recursive calls to make despite being the same size of expression. As such, it is incredibly difficult to determine a plausible relationship between the runtime and the size of an expression.

Conclusion

This program may be used to convert a variety of propositional logic expressions to CNF quickly and efficiently, with little data cleaning from the user. It will always find a conversion to CNF given enough time and computation power. The source code and documentation for this project may be found on: Github.