Inference Engine

COS30019–Introduction to AI

Swinburne University of Technology

Alex Cummaudo 1744070

Semester 1, 2016

Abstract

This assignment implements a basic inference engine using propositional logic to make inferences. In addition to the required truth table, forward and backward chaining methods, extensions to the inference engine were made. These extensions include a full propositional logic parser, making the truth table method more complete, as well as an additional entailment method—the resolution method. Since the resolution method was made, the inference engine is able to translate input sentences into conjunctive normal form, and thus also negation normal form.

Acknowledgements

I would like to acknolwedge Russell and Norvig (2009) as it was referenced for all implementations of each entailment method, chiefly in their provided psuedo-code for the algorithms. In addition, the lecture on converting to conjunctive-normal form by van der Meyden (2000) helped significantly, as well as the attached reference by Huth and Ryan (2004).

The logic inference engine developed by Sorensen (2014) was used as an alternate inference engine to validate my conversions of conjunctive and normal forms. It was helpful to have this secondary source to assert that my inference engine was working as expected.

Lastly, to implement the tokeniser used in the implementation of parsing the sentences from file, the article provided by Swift Studies (2014) on creating a generic tokeniser in the Swift programming langauge was useful.

Code History

The history for the source code provided is version controlled via git and can be found on GitHub. Refer to `http://github.com/alexcu/inference-engine/commits/master`.

Prerequisites

The implementation was written using the Swift 2.2 programming langauge, which you can download from at `https://swift.org/download/`.

Teamwork

The unit convenor has granted special permission allowing me to complete the assignment individually.

Contents

1 Features **4**
1.1 Search Algorithms 4
1.2 Extended Propositional Logic Token Parser 4
1.3 Conjunctive and Negation Normal Form 5

2 Testing **10**
2.1 Unit Testing 10
2.2 Extended Parsing Entailment 10
2.2.1 Truth Table Entailment 10
2.2.2 Resolution Tests 11

1 Features

A more extensive list of search features can be shown using the `--help` switch.

1.1 Search Algorithms

The solver implements the following entailment methods:

- Truth Table Method, `TT`,
- Forward Chaining Method, `FC`,
- Backward Chaining Method, `BC`, and
- Resolution Method, `RE`

1.2 Extended Propositional Logic Token Parser

Full token parsing is used when parsing the knowledge base from the source file. Thus, in addition to the horn-clause connectives for implicate (`=>`) and for conjunction (`&`), a a **biconditional** can be used using the `<=>` syntax, a **disjunction** can be used using `\/` or a pipe `|`, and **negation** can be used using a tilde, `~`. In addition, the tokeniser supports parentheses to associate precedence. Refer to the example shown in Figure 1.1.

$$P \vee Q \wedge (\neg S \Rightarrow ((T \vee W) \wedge P)) \Leftrightarrow Q$$

(a) Input logic

```
(P | (Q & (~S => ((T | W) & P)))) <=> Q
```

(b) Parsed representation of input

Figure 1.1: Representation of full token parsing

As shown in Figure 1.1b, the correct operator precdence has been used—the parser associates $Q \wedge (\neg S \Rightarrow ((T \vee W) \wedge P))$ over $P \vee Q$ shown by the output parentheses. However, $T \vee W$ takes precedence over $W \wedge P$ as it is in a set of parentheses unless overriden by braces.

1.3 Conjunctive and Negation Normal Form

To properly implement the Resolution Method, sentences must be resolved to their conjunctive normal form and, thus, their negation normal form. As such, two computed properties exist on all sentences:

- `inConjunctiveNormalForm` (Listing 5), and
- `inNegationNormalForm` (Listing 4)

When converting to conjunctive normal form, it is also required that all biconditionals and implications are eliminated. This is implemented using the `withoutImplications` computed property (Listing 3). Note that the listings shown below are only for *complex* sentences; *atomic* sentences have a default implementation that just return the sentence unchanged. Custom operators defined in the code make it easier to interoperate sentences together; refer to the implication and biconditional eliminations in Listing 3.

Users can choose to convert their input files to either negation or conjunctive normal forms by providing the inference engine with the `cnf` method or `nnf` methods (refer to README.txt or use the `--help` switch).

Consider the example shown below. The knowledge base consists of a sentence in CNF and thus NNF (1), a sentence in NNF but not CNF (2)[1] and a sentence not in NNF and thus not in CNF (3)[2]. Lastly, the query α is in CNF and thus NNF (4).

$$
\begin{aligned}
KB = \{ & \\
& (\neg p \lor q \lor r) \land (\neg q \lor r) \land (\neg r), && (1)\\
& (\neg p \lor q \lor r) \land ((p \land \neg q) \lor r) \land (\neg r) && (2)\\
& (\neg p \lor q \lor r) \land \neg(\neg q \lor r) \land (\neg r), && (3)\\
\} & \\
\alpha = p \land q & && (4)
\end{aligned}
$$

It can be seen that converting the above to CNF and NNF, as shown in Listings 1 and 2, stays static for (1) and (4), as per (5) and (6), successfuly converts (2) to CNF form, as per

[1]There is a conjunction in the second disjunction, i.e. $((p \land \neg q) \lor r)$

[2]Only atoms can be negated in CNF, but there is a not before the second negation, i.e., $\neg(\neg q \lor r)$

(8), but makes no changes when converting to NNF as it is already in NNF, as per (7), and lastly (3) can be converted to NNF, and thus is representable in CNF, as per (9) and (10).

$$\text{NNF}((\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg r)) = (\neg p \vee q \vee r) \wedge q \wedge \neg r \wedge \neg r \tag{5}$$
$$\text{CNF}((\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg r)) = (\neg p \vee q \vee r) \wedge q \wedge \neg r \wedge \neg r \tag{6}$$

$$\text{NNF}((\neg p \vee q \vee r) \wedge ((p \wedge \neg q) \vee r) \wedge (\neg r)) = (\neg p \vee q \vee r) \wedge (p \wedge \neg q \vee r) \wedge \neg r \tag{7}$$
$$\text{CNF}((\neg p \vee q \vee r) \wedge ((p \wedge \neg q) \vee r) \wedge (\neg r)) = (\neg p \vee q \vee r) \wedge (p \vee r) \wedge (\neg q \vee r) \wedge \neg r \tag{8}$$

$$\text{NNF}((\neg p \vee q \vee r) \wedge \neg(\neg q \vee r) \wedge (\neg r)) = (\neg p \vee q \vee r) \wedge q \wedge \neg r \wedge \neg r \tag{9}$$
$$\text{CNF}((\neg p \vee q \vee r) \wedge \neg(\neg q \vee r) \wedge (\neg r)) = (\neg p \vee q \vee r) \wedge q \wedge \neg r \wedge \neg r \tag{10}$$

Listing 1: Output of NNF conversion using input defined above

```
YES:
(((~p | q) | r) & (~q | r)) & ~r in NNF is (((~p | q) | r) & (~q | r)) & ~r
;
(((~p | q) | r) & ((p & ~q) | r)) & ~r in NNF is (((~p | q) | r) & ((p & ~q) | r)) &
    ~r
;
(((~p | q) | r) & (~(~q | r))) & ~r in NNF is (((~p | q) | r) & (q & ~r)) & ~r
;
p & q in NNF is p & q
```

Listing 2: Output of CNF conversion using input defined above

```
YES:
(((~p | q) | r) & (~q | r)) & ~r in CNF is (((~p | q) | r) & (~q | r)) & ~r
;
(((~p | q) | r) & ((p & ~q) | r)) & ~r in CNF is (((~p | q) | r) & ((p | r) & (~q |
    r))) & ~r
;
(((~p | q) | r) & (~(~q | r))) & ~r in CNF is (((~p | q) | r) & (q & ~r)) & ~r
;
p & q in CNF is p & q
```

Listing 3: Eliminating all implications and biconditionals in a complex sentence

```
var withoutImplications: Sentence {
    // non-binary sentences are just self
    if self.isUnary {
        return self
    }
    let lhs = self.sentences.left!.withoutImplications
    let rhs = self.sentences.right.withoutImplications
    if self.isSentenceKind(.Implicate) {
        // Implication elimination
        return ~lhs | rhs
    }
    else if self.isSentenceKind(.Biconditional) {
        // Biconditional elimination
        return (lhs => rhs) & (rhs => lhs)
    } else {
        // Return the lhs and rhs sentence without their implications using
        // the same connective
        return ComplexSentence(leftSentence: lhs,
                               connective: self.connective,
                               rightSentence: rhs)
    }
}
```

Listing 4: Converting a complex sentence to NNF

```
var inNegationNormalForm: Sentence {
    // Eliminate implications
    var result: Sentence = self.withoutImplications
    guard let resultAsComplex = (result as? ComplexSentence) else {
        return self
    }
    // Apply DeMorgan's Law to ~(A) to move .Negate inwards
    if resultAsComplex.isUnary {
        // Assume A is not atomic, else result is assigned
        if let negated = resultAsComplex.sentences.right as? ComplexSentence {
            // A == (~P)
            let rhsIsNegated =
                negated.isSentenceKind(.Negate)
            // A == (P & Q) or (P | Q)
            let rhsIsConjoinOrDisjoin =
                negated.isSentenceKind(.Conjoin) ||
```

```
                negated.isSentenceKind(.Disjoin)
            // ~A = ~(~P) = P
            if rhsIsNegated {
                // Hence result is just P
                result = (negated.sentences.right).inNegationNormalForm
            }
            // ~(P & Q) or ~(P | Q)
            else if rhsIsConjoinOrDisjoin {
                // Convert P & Q in NNF and support double negation (hence why
                // we negate at the start)
                let lhs = (~(negated.sentences.left!)).inNegationNormalForm
                let rhs = (~(negated.sentences.right)).inNegationNormalForm
                if negated.isSentenceKind(.Conjoin) {
                    // ~(P & Q) == ~P | ~Q
                    result = lhs | rhs
                } else {
                    // ~(P | Q) == ~P & ~Q
                    result = lhs & rhs
                }
            }
        }
    } else {
        let lhs = resultAsComplex.sentences.left!.inNegationNormalForm
        let rhs = resultAsComplex.sentences.right.inNegationNormalForm
        result = ComplexSentence(leftSentence: lhs,
                                 connective: resultAsComplex.connective,
                                 rightSentence: rhs)
    }
    return result
}
```

Listing 5: Converting a complex sentence to CNF

```
var inConjunctiveNormalForm: Sentence {
    // First convert to NNF
    var result = self.inNegationNormalForm
    // Recursively convert non-disjoint sentences
    guard let resultAsComplex = result as? ComplexSentence else {
        // If cannot represent as complex, then nothing else to do
        return result
    }
    // (P | (Q & R)) == (P | Q) & (P & R)
    if result.isSentenceKind(.Disjoin) {
```

```
            let lhs = resultAsComplex.sentences.left!.inConjunctiveNormalForm
            let rhs = resultAsComplex.sentences.right.inConjunctiveNormalForm
            // Either side is an Conjoin
            if lhs.isSentenceKind(.Conjoin) || rhs.isSentenceKind(.Conjoin) {
                // if rhs is conjoin then P | (Q & R) == p  | qr
                // if lhs is conjoin then (Q & R) | P == qr | p
                let p  =  rhs.isSentenceKind(.Conjoin) ? lhs : rhs
                let qr =
                    (rhs.isSentenceKind(.Conjoin) ? rhs : lhs) as! ComplexSentence
                let q  = qr.sentences.left!
                let r  = qr.sentences.right

                // (p | q) & (p | r)
                if rhs.isSentenceKind(.Conjoin) {
                    result =
                        (p | q).inConjunctiveNormalForm &
                        (p | r).inConjunctiveNormalForm
                // (q | p) & (r | p)
                } else {
                    result =
                        (q | p).inConjunctiveNormalForm &
                        (r | p).inConjunctiveNormalForm
                }
            } else {
                result = lhs.inConjunctiveNormalForm | rhs.inConjunctiveNormalForm
            }
        } else {
            let rhs = resultAsComplex.sentences.right.inConjunctiveNormalForm
            if resultAsComplex.isBinary {
                let lhs = resultAsComplex.sentences.left!.inConjunctiveNormalForm
                result = ComplexSentence(leftSentence: lhs,
                                         connective: resultAsComplex.connective,
                                         rightSentence: rhs)
            } else {
                result = ComplexSentence(connective: resultAsComplex.connective,
                                         sentences: rhs)

            }
        }
        return result
    }
```

2 Testing

2.1 Unit Testing

The codebase was developed using a test-driven develop strategy. Test coverage for the majority of the codebase was factored in. All tests are located within the the `test` subdirectory. These tests help with the confidence of complex implementations of equivelence forms, such as De Morgan's laws, implication and bidirectional elimination, NNF and CNF conversion using both implications, negations and bidirectionals. Refer to the tests shown in `ComplexSentenceTests.swift`.

Each of these tests can be verified using an alternate inference engine such as WolframAlpha (`https://www.wolframalpha.com/`), which was used to ensure that assertions of tests were indeed correct.

2.2 Extended Parsing Entailment

2.2.1 Truth Table Entailment

Extended propositional logic parsing was initially tested using the truth table parser. An that was used in the testing strategy was the *Smoke, Heat and Fire* example that was provided in a previous tutorial. For example:

$$\begin{aligned} KB &= \{((Smoke \land Heat) \Rightarrow Fire) \Leftrightarrow ((Smoke \Rightarrow Fire) \lor (Heat \Rightarrow Fire))\} \\ \alpha &= Smoke \end{aligned}$$

This would produce a truth table similar to that shown in Table 2.1. As r_5 shows that the all models in the knowledge base holds, there would be only four models in this entailment (the underlined *true*s in the *Smoke* column). Hence, it is shown that $KB \models \alpha$, and therefore output of the program is:

```
YES: 4
```

This is tested under the unit tests which are described in the `TruthTableTests.swift` and `XCTestCase+EntailmentTest.swift` files under the `test` directory.

While this test only contains one sentence in the knowledge base, it is to be noted that this test aims to only test the extended parsing entailment using the truth table method, and not the truth table method itself. There already exists test which are to test the logic for the truth table, which are described in the aformentioned files.

2.2.2 Resolution Tests

Testing the resolution method identified several bugs with the NNF and CNF implementations. Initially, NNF was not removing implications and biconditionals. It was found that the NNF, CNF and `withoutImplications` methods were not recursively recalling themselves if *no* special conditions were true. Thus, rather than just returning the result, the result is recreated using the same connecitve but applying NNF or CNF to the lefthand and righthand sentences, when applicable. This key missing implementation would cause errors in creating CNF, especially for the addition of new cases.

The resolution method also uses tests to verify that the resolve method works as intended. As seen in `ResolutionTests.swift`, tests assert that the resolution of two complementaries resolves to the $False$ propositional (i.e., `Resolve`$(\{A, \neg A\}) \mapsto False$). This was also identified to not work initially in the tests, as it was found that a `Sentence`'s `join` method did not support the creation of the false or true atom when appopriate. Since "the empty clause—a disjunction of no disjuncts—is equivalent to False because a disjunction is true only if at least one of its disjuncts is true" (Russell and Norvig, 2009, p.254) it is required to be calculated correctly, in order to verify that resolution method is complete, as shown in `Resolution.swift`. This is implemented in the `_ArrayType` extension applied to `Sentence`s, which can be seen at the bottom of the `Sentence.swift` source.

Table 2.1: Truth table describe the *Smoke, Heat and Fire* example

$Smoke$	$Heat$	$Fire$	r_0	r_1	r_2	r_3	r_4	r_5
			$Smoke \land Heat$	$r_0 \Rightarrow Fire$	$Smoke \Rightarrow Fire$	$Heat \Rightarrow Fire$	$r_2 \lor r_4$	$r_1 \Leftrightarrow r_4$
$false$	$false$	$false$	$false$	$true$	$true$	$true$	$true$	$true$
$false$	$false$	$true$	$false$	$true$	$true$	$true$	$true$	$true$
$false$	$true$	$false$	$false$	$true$	$true$	$false$	$true$	$true$
$false$	$true$	$true$	$false$	$true$	$true$	$true$	$true$	$true$
$\underline{true}$	$false$	$false$	$false$	$true$	$false$	$true$	$true$	$true$
$\underline{true}$	$false$	$true$	$false$	$true$	$true$	$true$	$true$	$true$
$\underline{true}$	$true$	$false$	$true$	$false$	$false$	$false$	$false$	$true$
$\underline{true}$	$true$	$true$	$true$	$true$	$true$	$true$	$true$	$true$

References

Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004. ISBN 052154310X.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, third edition, 2009. ISBN 0136042597.

Tyler Sorensen. PBL Package. `http://formal.cs.utah.edu:8080/pbl/PBL.php`, 2014. [Online; accessed 24 April 2016].

Swift Studies. Building a Swift Parser with an Improved Tokenizer. `http://www.swift-studies.com/blog/2014/7/1/building-a-swift-parser-with-an-improved-tokenizer`, 2014. [Online; accessed 24 April 2016].

Ron van der Meyden. Lecture 6: Conjunctive normal form. `http://www.cse.unsw.edu.au/~meyden/teaching/cs2411/`, 2000. [Online; accessed 10 May 2016].