

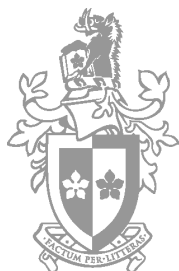
Inference Engine

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 acknowledge 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).

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 language 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 language, 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	10

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 (\Rightarrow) and for conjunction ($\&$), a **biconditional** can be used using the \Leftrightarrow syntax, a **disjunction** can be used using \vee or a pipe $|$, and **negation** can be used using a tilde, \sim . 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

$$(P \mid (Q \ \& \ (\sim S \Rightarrow ((T \mid W) \ \& \ P)))) \Leftrightarrow Q$$

(b) Parsed representation of input

Figure 1.1: Representation of full token parsing

As shown in Figure 1.1b, the correct operator precedence 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 overridden 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 4), and
- `inNegationNormalForm` (Listing 3)

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 2). 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 2.

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).

For example, the following output shown in Listing 1 is given when the knowledge base provided in 1 is provided running the `cnf` method. Each of these can be verified using an alternate inference engine such as WolframAlpha (<https://www.wolframalpha.com/>), which was used to ensure that conversions of forms were indeed correct.

Another example used is that shown in ??, which shows the correct output of a non-representable

$$KB = \{ \tag{1}$$

$$(\neg p \vee q \vee r) \wedge (\neg q \vee r) \wedge (\neg r), \tag{2}$$

$$(\neg p \vee q \vee r) \wedge \neg(\neg q \vee r) \wedge (\neg r), \tag{3}$$

$$(\neg p \vee q \vee r) \wedge ((p \wedge \neg q) \vee r) \wedge (\neg r) \tag{4}$$

$$\} \tag{5}$$

$$\alpha = (p \wedge q) \Rightarrow p \vee q \wedge \neg p \tag{6}$$

Listing 1: Output of CNF conversion from 1

```

YES:
p in CNF is p
;
q in CNF is q
;
~(p & q) in CNF is ~p | ~q
;
~p | ~q in CNF is ~p | ~q
;
(~(p & q)) <=> (~p | ~q) in CNF is ((q | (~p | ~q)) & (p | (~p | ~q))) & ((q | (~p | ~q)) & (p | (~p | ~q)))
;
(p & q) => (p | (q & ~p)) in CNF is ((~p | ~q) | (p | ~p)) & ((~p | ~q) | (p | q))

```

Listing 2: 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 3: 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 4: 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: rhs,
                                connective: resultAsComplex.connective,
                                rightSentence: lhs)
    } 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.

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:

$$KB = \{((Smoke \wedge Heat) \Rightarrow Fire) \Leftrightarrow ((Smoke \Rightarrow Fire) \vee (Heat \Rightarrow Fire))\}$$
$$\alpha = Smoke$$

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 aforementioned 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

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

<i>Smoke</i>	<i>Heat</i>	<i>Fire</i>	r_0 $\text{Smoke} \wedge \text{Heat}$	r_1 $r_0 \Rightarrow \text{Fire}$	r_2 $\text{Smoke} \Rightarrow \text{Fire}$	r_3 $\text{Heat} \Rightarrow \text{Fire}$	r_4 $r_2 \vee r_4$	r_5 $r_1 \Leftrightarrow r_4$
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>
<u><i>true</i></u>	<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>
<u><i>true</i></u>	<i>true</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>
<u><i>true</i></u>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>

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 connective 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`, there are tests for checking resolve works for tautologies (i.e., $A \vee \neg A$) to the False propositional. 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 appropriate. 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 `Sentences`, which can be seen at the bottom of the `Sentence.swift` source.

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.

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].