COS30019
Introduction to Artificial Intelligence

ASSIGNMENT 2

INFERENCE ENGINE

REPORT

James Kilby | 102101141
Le Bao Duy Nguyen | 102449993

Bao Vo | Wed 12:30 PM
Qinyuan Li | Tue 2:30 PM

Table of Contents

(1) Student Details

Student 1
Name: James Kilby
ID: 102101141

Student 2
Name: Le Bao Duy Nguyen
ID: 102449993

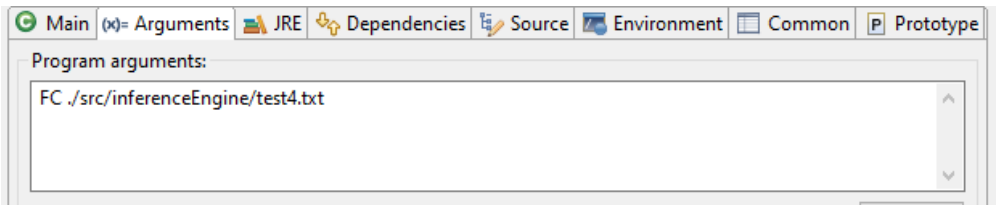
(2) Instructions

To start this program type the following:
iengine <method> <filename>
Where <method> is the algorithm used to implement the Truth Table, Backward Chaining or Forward Chaining.
1. TT: Truth Table
2. BC: Backwards Chaining
3. FC: Forward Chaining
<filename>: is the name of the text file with arguments of a horn-form and a query (e.g. test1.txt)

The output is:
Truth Table
Yes: n (number of entailments), or
No: meaning query does not entail the KB.

Forward and Backward Chaining output the propositional variables entailed from the KB found during the algorithm's execution period instead of the number of entailments.
e.g.
Yes: a, b, p2, p3, p1, d

Or pass the arguments into the IDE manually



(3) Introduction

This report is an inference engine for propositional logic in software based on the Truth Table (TT)
checking, and Backward Chaining (BC) and Forward Chaining (FC) algorithms. Your inference engine will take as arguments a Horn-form Knowledge Base KB and a query q which is a proposition symbol and

determine whether q can be entailed from KB.

(4) Implementation

Example
Tell
p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p1=>d; p1&p3 => c; a; b; p2;

Ask

d

Where p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p1=>d; p1&p3 => c; are CLAUSES

c, a, b, p2 are FACTS

d is QUERY

=> is Implication SYMBOL

& is Conjunction SYMBOL

(5) Features

Knowledge Base (KB)
Split a text file into Clause, Fact, Query and Symbol. Clause is the character with the statement type (e.g. '=>' , '<=>' , '&' , '/' , '~', etc.) and another character. Fact is the standalone characters at the end of the TELL phase. Query is the last character after the ASK. Symbol is the statement type (e.g. '=>' , '<=>' , '&' , '/' , '~', etc.). The more complex and lengthy KB, the more the run time of the algorithms. However for this assignment, only implication and and (i.e. =>, &) are used.

```
reader.readLine();

String tell = reader.readLine();

tell = tell.replaceAll(regex: "\\s", replacement: "");

String[] sentence = tell.split(regex: ";");

for (int i = 0; i < sentence.length; i++) {
  if (sentence[i].contains(s: "=>")) {
    clauses.add(new HornClause(sentence[i]));
  } else {
    facts.add(sentence[i]);
  }
}
```

Truth Table
The Truth Table method follows the mathematical Propositional Logic of KB $\vdash \alpha$ which means KB entails sentence Alpha if and only if alpha is true in cases that KB is true.

A model of the Truth Table is designed to include all symbols within the text file and is 2^n deep and the number of symbols long. The depth of a truth table grows exponentially with the number of symbols.

Generate a grid with columns and rows using a maths function.

```
columns = items.size();

rows = ( ( int ) Math.pow ( a: 2, ( items.size() ) ) );

grid = new boolean[ rows ][ columns ];
```

The Knowledge Base is developed and tested by using the developed Truth Table and using some logical methods that test the clauses and return a value to put into the KB. The depth of the KB is the same as the developed Truth Table and to map each point correctly the vector of symbols is used for ordering.

The algorithm tests the KB's entailment to find a solution. It maps the query location and tests each KB clause against each one and then tests if the developed Truth Table query symbol is true. The count increases if it returns true and continuously loop through the next line on the KB and Truth Table until the end.

```
@Override
public boolean resultPassed()
{
    for ( int i = 0; i < rows; i++ )
    {
        for ( int j = 0; j < factIndex.length; j++ )
        {
            if ( formulaResult[ i ] )
            {
                if ( ! grid[ i ][ queryIndex ] )
                {
                    formulaResult[ i ] = false;

                    queryResult[ i ] = false;

                    break;
                }
                else
                {
                    queryResult[ i ] = true;
                }

                formulaResult[ i ] = grid[ i ][  factIndex [ j ]  ];
            }
            else
            {
                break;
            }
        }
    }

    for ( int i = 0; i < rows; i++ )
    {
        if ( formulaResult[ i ] )
        {
            for ( int j = 0; j < literalIndex.length; j++ )
            {
                if( clauses.get(j).literalCount() == 2 )
                {
                    if ( ( grid[ i ][  literalIndex[ j ][ 0 ]  ] == true )
                    &&   ( grid[ i ][  literalIndex[ j ][ 1 ]  ] == true )
                    &&   ( grid[ i ][  entailed[ j ]  ] == false ) )
                    {
                        formulaResult[ i ] = false;
                    }
                }
                else
                {
```

```
                else
                {
                    if ( ( grid[ i ][  literalIndex[ j ][ 0 ]  ] == true )
                    &&   ( grid[ i ][  entailed[ j ]  ] == false ) )
                    {
                        formulaResult[ i ] = false;
                    }
                }
            }
        }
    }

    for ( int i = 0; i < rows; i++ )
    {
        if ( formulaResult[ i ] )
        {
            count++;
        }

        if ( queryResult[ i ] == false && formulaResult[ i ] == true )
        {
            return false;
        }
    }

    return true;
}
```

```
public void createGrid()
{
    for ( int i = 0; i < clauses.size(); i++ )
    {
        for ( int j = 0; j < clauses.get( i ).literalCount(); j++ )
        {
            items.add( clauses.get( i ).getLiteralsAtIndex( j ) );
        }
        items.add( clauses.get( i ).getEntailedLiteral() );
    }

    Set<String> hashset = new HashSet<>();
    hashset.addAll(items);
    items.clear();
    items.addAll(hashset);

    columns = items.size();

    rows = ( ( int ) Math.pow ( a: 2, ( items.size() ) ) );

    grid = new boolean[ rows ][ columns ];

    formulaResult = new boolean[ rows ];

    for ( int i = 0; i < rows; i++ )
    {
        formulaResult[ i ] = true;
    }

    literalIndex = new int[ clauses.size() ][ 2 ];

    factIndex = new int[ facts.size() ];

    entailed = new int[ clauses.size() ];

    queryResult = new boolean[ rows ];

    queryIndex = 0;

    count = 0;
```

```
public void getFactsColumnIdx()
{
    for ( int i = 0; i < facts.size(); i++ )
    {
        for ( int j = 0; j < items.size(); j++ )
        {
            if ( facts.get( i ).equals( items.get( j ) ) )
            {
                factIndex[ i ] = j;
            }

            if ( query.equals( items.get( j ) ) )
            {
                queryIndex = j;
            }

        }
    }
}

public void getLiteralsColumnIdx()
{
    for ( int i = 0; i < items.size(); i++ )
    {
        for ( int j = 0; j < clauses.size(); j++ )
        {
            for ( int k = 0; k < clauses.get( j ).literalCount(); k++ )
            {
                if ( clauses.get( j ).getLiteralsAtIndex( k ).equals( items.get( i ) ) )
                {
                    literalIndex[ j ][ k ] = i;
                }
            }

            if ( clauses.get( j ).getEntailedLiteral().equals( items.get( i ) ) )
            {
                entailed[ j ] = i;
            }
        }
    }
}
```

Forward Chaining

The Forward Chaining (FC) algorithm works by drawing conclusions based on known facts. Known facts represent the initial state and the target state (i.e. query). FC an automated process, not knowing whether the clauses investigated will contribute to the goal state and keep searching through the clause. The algorithm keeps on repeating the same inference process to achieve data. As a result, the FC may find that it is performing a number of additional operations that are unrelated to the destination. Consequently, the FC might find a number of additional redundant operations unrelated to the goal. Therefore, BC is considered to be a more efficient algorithm than FC.

```
@Override
public boolean resultPassed(){

    while ( !facts.isEmpty() ){

        String aFact = facts.remove(index: 0);

        outputFacts.add( aFact );

        if ( aFact.equals( query ) )
        {
            return true;
        }

        for ( int i = 0; i < clauses.size(); i++ )
        {
            for ( int j = 0; j < clauses.get(i).literalCount(); j++ )
            {
                if ( aFact.equals( clauses.get(i).getLiteralsAtIndex(j) ) )
                {
                    clauses.get(i).removeLiteral(aFact);
                }
            }
        }

        for ( int i = 0; i < clauses.size(); i++ )
        {
            if ( clauses.get(i).literalCount() == 0 )
            {
                facts.add( clauses.get(i).getEntailedLiteral() );

                clauses.remove(i);
            }
        }
    }
    return false;
}
```

Backward Chaining

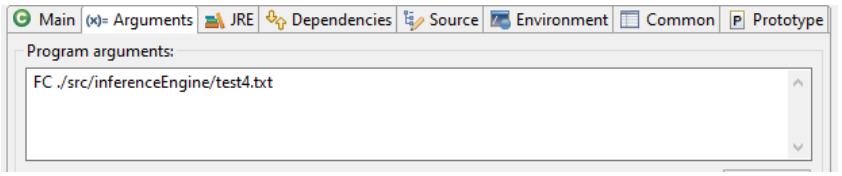
The backward chaining (BC) algorithm starts from the target state (i.e. query). Then it repeats all the clauses from the target state in the opposite direction to prove the query through sub goals. This process checks if the query can be derived from the KB.

```
@Override
public boolean resultPassed(){

    queries.add( query );

    while ( queries.size() > 0 )
    {
        String currentQuery = queries.remove(index: 0);

        outputFacts.add( currentQuery );

        if( ! compareFacts( currentQuery ) )
        {
            if( ! compareClauses( currentQuery ) )
            {
                return false;
            }
        }
    }
    return true;
}
```

```
public boolean compareClauses( String aQuery )
{
    boolean result = false;

    for ( int i = 0; i < clauses.size(); i++ )
    {
        if ( aQuery.equals( clauses.get(i).getEntailedLiteral() ) )
        {
            result = true;

            for ( int j = 0; j < clauses.get(i).literalCount(); j++ )
            {
                queries.add( clauses.get(i).getLiteralsAtIndex(j) );
            }
        }
    }

    for ( int i = 0; i < outputFacts.size(); i++ )
    {
        for ( int j = 0; j < queries.size(); j++ )
        {
            if ( outputFacts.get(i).equals( queries.get(j) ) )
            {
                queries.remove(j);
            }
        }
    }

    Set<String> hashset = new HashSet<>();
    hashset.addAll(queries);
    queries.clear();
    queries.addAll(hashset);

    return result;
}

public boolean compareFacts( String aQuery )
{
    for ( int i = 0; i < facts.size(); i++ )
    {
        if ( aQuery.equals( facts.get(i) ) )
        {
            return true;
        }
    }
    return false;
}
```

(6) Bugs

All necessary features have been implemented (i.e. Truth Table, Forward Chaining and Backward Chaining).
Bug #1: iengine.exe file was compiled successfully using Launch4J with a JAR file, however our team was unable to run it. By using an IDE like Eclipse, manual entering of arguments is a substitute method to run the project. If the marker can't run the exe file, please use manual entering method instead.
E.g.



(7) Missing

This assignment has not done the complex symbols that work with the Truth Table scenario. Right now it only supports implication and conjunction. However the algorithms should run smoothly with the current set up. Additional symbols are part of the extra research which our team did not have enough time to implement.

~ for negation (¬)
& for conjunction (∧)
|| for disjunction (∨)
=> for implication (⇒)
<=> for biconditional (⇔)

(8) Test Cases

Test case 1 (test_HornKB.txt)
Tell
p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p1=>d; p1&p3 => c; a; b; p2;
Ask
d

```
Truth Table
YES: 3

Forward Chaining
YES: a, b, p2, p3, p1, d

Backward Chaining
YES: p2, p3, p1, d
```

Test case 2 (test2.txt)

TELL

a&b=>d; b=>c; c&a=>d; b&d=>e; c&e=>f; d=>b; a; c;

ASK

f

```
Truth Table
YES: 1
```

```
Forward Chaining
YES: a, c, d, b, d, e, c, f
```

```
Backward Chaining
YES: a, d, b, e, c, f
```

Test case 3 (test3.txt)

TELL

a&z=> b; a => b; a;

ASK

b

```
Truth Table
YES: 2
```

```
Forward Chaining
YES: a, b
```

```
Backward Chaining
NO
```

Test case 4 (test4.txt)

TELL

a => b;

ASK

c

```
Truth Table
NO
```

```
Forward Chaining
NO
```

```
Backward Chaining
NO
```

(9) Acknowledgements / Resources

---

https://en.wikipedia.org/wiki/List_of_logic_symbols was used to help develop a Truth Table.
https://careercup.com/question?id=17632666 was used to understand how to populate a Truth Table with T/F combination.
http://aima.cs.berkeley.edu/algorithms.pdf was used to research about logical agents and the algorithms in this project.
https://youtu.be/EZJs6w2YFRM was used to help understand Forward and Backward Chaining better.
https://www.cs.cmu.edu/afs/cs/academic/class/15381-s07/www/slides/022707reasoning.pdf was used to learn about logic and reasoning.
Pseudo codes were used to understand the algorithms.

Truth Table

```
function TT-ENTAILS?(KB, α) returns true or false
    symbols ← a list of the proposition symbols in KB and α
    return TT-CHECK-ALL(KB, α, symbols, [])

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
    if EMPTY?(symbols) then
        if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
        else return true
    else do
        P ← FIRST(symbols); rest ← REST(symbols)
        return TT-CHECK-ALL(KB, α, rest, EXTEND(P, true, model)) and
               TT-CHECK-ALL(KB, α, rest, EXTEND(P, false, model))
```

Forward Chaining

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known to be true

  while agenda is not empty do
      p ← POP(agenda)
      unless inferred[p] do
          inferred[p] ← true
          for each Horn clause c in whose premise p appears do
              decrement count[c]
              if count[c] = 0 then do
                  if HEAD[c] = q then return true
                  PUSH(HEAD[c], agenda)
  return false
```

\

Backward Chaining

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

function FOL-BC-OR(KB, goal, θ) returns a substitution
  for each rule in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs ⇒ rhs) ← STANDARDIZE-VARIABLES(rule)
    for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ'

function FOL-BC-AND(KB, goals, θ) returns a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else
    first, rest ← FIRST(goals), REST(goals)
    for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ'' in FOL-BC-AND(KB, rest, θ') do
        yield θ''
```

(10) Research

By using the blow formula, we can estimate the runtime of a Truth Table that has a large size.

$$a_{t_1} = a_1 + (2 \times n)$$

$$a_{t_2} = a_1 \times 2 + (2 \times n)$$

$$a_{t_3} = a_2 \times 2 + (2 \times n) \ldots\ldots a_{t_n} = a_{n-1} \times 2 + (2 \times n)$$

Note: $a_1 = 40$ *with the formula starting at* 2^{18}

When a Truth Table is 2^n in size, the size and run time correlate with each other exponentially. That said, a 2^n when n is 30,40 etc would take hours to run and require a more efficient algorithm since the current ones are not guaranteed to successfully handle the run time.

That being said, Truth Table is not always the most efficient method of checking the facts for a query. In the event of large databases, Backward Chaining would be more preferably used as it will require less action and run time in the algorithm to execute and search for the solution.

Size (2^n)	10	14	16	18	20	30
Time taken (s)	<1	~2	~11	~95	~218	~201600

(11) Summary Report

This project was done swiftly and in a timely manner. Both members had no trouble communicating with each other via Discord and Github. Regular weekly meetings were conducted to ensure the project was done in checkpoints (2-3 meetings per week).

Team Member	Contribution	Contribution Percentage (%)
James Kilby	Report cover page, contents, formatting, research, cleaning up. Main, class, and Truth Table programming in C++ (original code version).	45%
Le Bao Duy Nguyen	Coded 3 algorithms in Java. Report documenting and testing test cases. Initially the project was in C++ but switched language due to not having enough time to learn C++. James helped a lot by showing me how to split a text file into KB and how Truth Table works.	55%

(12) Conclusion

This report details the required steps to use our Java program, specify the desired inference engine algorithm, and how to format text files to properly signify facts, clauses, and queries. It details our group's understanding of inference engines, propositional logic, and how to program a knowledge base.

Truth Table, Forward Chaining, and Backward Chaining each have their advantages and disadvantages. Truth Tables tend to be exponentially resource-heavy compared to the Chaining methods. Forward Chaining has the potential to waste processing time on unnecessary clauses that don't contribute to answering a posed query, whereas Backward Chaining limits itself to necessary clauses. The best algorithm for the job will depend on the context in which it is used.