# CSC442: Intro to AI

# Project 2 Report: Automated Reasoning

Name: Soubhik Ghosh, Rich Magnotti

NetId: sghosh13, rmagnott

October 9, 2019

## Introduction

This project is about implementing at least 2 inference methods for Propositional Logic. It has been collaboratively executed by the following 2 members:

1. Soubhik Ghosh
2. Rich Magnotti

In this project we have implemented the following inference methods:

- Truth-Table Enumeration (Basic Model Checking)
- Resolution algorithm (Advanced Propositional Inference)
- The WalkSAT algorithm (Advanced Propositional Inference)

## Program design

We implemented the complete program in Haskell and Python on the vim editor.

## Code Structure and Problem Representation

Our implementation is organized into 6 Haskell source files and 3 Python Files

1. `Formula.hs`
2. `Reasoning.hs`
3. `Test.hs`
4. `Scanner.hs`
5. `Parser.hs`
6. `CNF_converter.hs`

1. `Well-Formed-Formula_Parser.py`
2. `Data_Structs.py`
3. `CNF_Converter.py`

**`Formula.hs`**

- This file contains the algebraic data type for a well formed formula called `Prop` and a type instance for writing a formula to std output for viewing purposes.
- The data type uses a recursive definition by defining the type `Prop` in terms of itself by creating logical connectives from meaningful symbols (See source code for more details)
- We use/assume the following symbols for each of the logical connectives in our implementation:
    - Negation: !
    - Conjunction: &
    - Disjunction: |
    - Implication: =>
    - Biconditional: <=>
- This recursive definition is internally represented as a complete binary tree in Haskell. Hence our formulas which are defined using `Prop` can be thought of as being represented using a binary tree data structure. This tree's leaves correspond to the atomic sentences (symbols) and the internal nodes correspond to logical connectives

**`Reasoning.hs`**

- This file contains the implementation of the 3 inference methods
    - Truth-Table Enumeration
    - Resolution algorithm
    - The WalkSAT algorithm
- Each of the above inference methods correspond to the following entry point functions in order:
    - `ttEntails`
    - `plResolution`
    - `walkSatEntails`
- Each one of these functions takes 2 input strings and returns True or False depending on whether the query can be entailed or not.

**Truth-Table Enumeration**

- `ttEntails` is the function that performs truth table enumeration for deciding propositional entailment.
- `ttEntails` makes use of another recursive function `ttCheckAll` that iterates over the list of symbols and exhaustively checks all models (symbol assignments) to find entailment.
- 2 other helper functions indirectly used by `ttEntails` are `symbols` and `plTrue`.

- `Symbols` returns the list of atomic sentences by performing DFS (Depth First Search) on the formula tree. As a matter of fact DFS has been used a lot throughout the implementation.
- `plTrue` returns the evaluation of a sentence on a particular model (Variable assignment). It computes the value recursively by performing DFS on the formula tree. After each child node is evaluated, the parent node's value is evaluated by operating on the children's value(s) by applying the parent's logical connective.

**Resolution algorithm**

- `plResolution` is the function that performs resolution for deciding propositional entailment.
- `plResolution` converts the strings into the tree representation defined by `Prop`. It then joins them into one tree (KB ∧ ¬α) and finally converts the tree into a CNF form.
- The CNF form is then broken down into a set of clauses, where each clause is a set of literals (atomic sentences and negated atomic sentences). Set has a literal meaning here which means we use Haskell's Data.Set module for the set data type. We use a set for storing clauses to remove any duplicate clause that could be generated during the resolution steps. We also use a set for storing literals in a clause to keep only one copy of a literal in a clause. The removal of multiple copies of literals is called **factoring.**
- Like the actual resolution pseudocode (AIMA Figure 7.12), we resolve clauses until we find an empty clause or no new clauses can be generated. But unlike the actual resolution pseudocode, we do a couple of things differently without changing the actual algorithm as explained below:
    - We replace the loop implementation with a recursive function `loopTillDone` where each function call is one loop iteration. We do this to respect Haskell's functional paradigm.
    - While generating new clauses by resolving existing ones, we discard a clause if its a tautology (`tautology`) or if its size is greater than or equal to the existing clauses (`longerClause`). This is done so that our `plResolution` terminates in a reasonable amount of time. (More on this later)
- Finally `plResolution` returns True or False depending on whether it found an empty clause or not. Empty clause is nothing but an empty set in our case.

**`Well-Formed-Formula_Parser.py & CNF_Converter.py`**

The intention here was to simulate the following grammar:

```
#    expression ::= letter
#    expression ::= '(' expression ')'
#    expression ::= expression '=>' expression
#    expression ::= expression '<=>' expression
#    expression ::= expression '&' expression
#    expression ::= expression '|' expression
#    expression ::= '!' expression
```

Input:

The input for the parser is a string delimited by spaces, written in infix format.

Output:
The output is the user's input in CNF form to then be passed to the Haskell Python function call.

**STEP 1: Infix -> Postfix**

The very first step was to convert the string to a list delimited by spaces, which we did with the simple Python *split(' ')* function.

Subsequently because the project rubric calls for the expression to be a tree structure, we decided to use an expression tree representation. Because it is much simpler to convert a postfix equation to an expression tree, we chose to first convert the input infix expression to a postfix expression.

Thus, from our main function we called *tokenizerPF(tokenList)*:

```
def main():
    tmp = input("enter your logical sentence\n")
    tokenList = tmp.split(" ")
    tokenizerPF(tokenList)
```

The function *tokenizerPF()* takes the list of infix tokens, and cleverly± reformats the list into postfix form. This algorithm is derivative of the Shunting Yard algorithm, but for logic. Most notably, this algorithm is also able to deal with negatives and their distributions over parenthesis via the following portion:

```
if tokenList[k] is '(' and tokenList[k - 1] is '!':
    numParenSets = 1
    j = k + 1
    while numParenSets > 0:
        if tokenList[j] is '(':
            s.push(tokenList[j])
            numParenSets = numParenSets + 1
        elif tokenList[j] is ')':
            numParenSets = numParenSets - 1
        elif tokenList[j] is '!':
            tokenList[j] = ' '
            break
        else:
            tokenList[j] = flipListToken(tokenList[j])
        j = j + 1
    tokenList[k - 1] = ' '
    s.pop()
```

Any time a left parenthesis is encountered along our enumeration of the total infix token string, we enter a while loop. The while loop will continue to negate every operator and proposition until even number of open/close parenthesis are found or until another negation is found (in which case we break the loop because everything after the subsequent negation will no longer be negated). The magic here is in the *flipListToken()* function which will add a 'marker' (really just a '*') that will be handled when constructing the tree, and each marked operator or atomic proposition will be changed to its negation down the pipeline.

**STEP 2: Construct Tree**

Once properly in postfix form, the list of tokens is passed to *constructExpTree()* to formulate the respective expression tree:

```
def main():
    tmp = input("enter your logical sentence\n")
    tokenList = tmp.split(" ")
    tokenizerPF(tokenList)
    eT = constructExpTree()
```

Here each token becomes a node in a binary tree which we call '*ExpTree*' (or simply '*tree*'). Each tree has three attributes: a left child, a right child, and a Boolean which ascertains whether the tree is marked to be negated or not.

While constructing the tree, we enumerate over all the tokens in the list and add each one to the tree as either an operator or a clause (specifically an atomic proposition or a proposition with a negation, e.g. *A* and *!A* are valid expressions based on our grammar). Operators will always have two children , and propositions will always be leaf nodes. As we enumerate, if any token has a negation marker (recall: '*') we toggle its negation before adding it to the tree.

**STEP 3: Postfix token list -> CNF**

Now that the *easy* part is over, from our current script we pass the tree to the WFF->CNF converter module *CNF_Converter.py:*

```
def main():
    tmp = input("enter your logical sentence\n")
    tokenList = tmp.split(" ")
    tokenizerPF(tokenList)
    eT = constructExpTree()
    toCNF(eT)
```

Once in the CNF converter module, our script steps through the following CNF conversion process:

1) Eliminate biconditionals
2) Eliminate implications
3) Move negations inwards
4) Distribute disjunction over conjunction

Eliminate Biconditionals

Input:
(A ⇔ B)
Output:
(A ⇒ B) ^ (B ⇒ A)

To accomplish this we used a DFS/in-order traversal of the expression tree (this will be a running theme), from the bottom up eliminating each biconditional. There was no need to re-traverse the tree once fully traversed because once an implication is eliminated, nothing is propagated inward unlike eliminating implications , for example, which pushed negations inward.

Eliminate Implications

Input:
(A ⇒ B)
Output:

(¬A ∨ B)

Similar to the process of eliminating biconditionals, this process traverses the expression tree DFS/in-order, and handles each implication, with the added work of now needing to deal with consequent negations. To deal with this, we push negations through the tree at every implication we find with the auxiliary function *makeNeg()*:

```
def impElim(tree):
    if tree is not None:
        impElim(tree.leftChild)
        if '=>' in tree.token:
            makeNeg(tree.leftChild)
            tree.token = '|'
        impElim(tree.rightChild)
```

```
def makeNeg(tree):
    if tree is not None:
        makeNeg(tree.leftChild)
        flipToken(tree)
        makeNeg(tree.rightChild)
```

The function *makeNeg()* takes the passed tree's left child and DFS/in-order explores each subsequent tree and toggles each tree's negation status, using the same *flipToken()* from when we were constructing the expression tree from the postfix token list.

**Because we propagated negations as we came across them, there was no need to make a separate step to do so. **

Distribute disjunction over conjunction

Input:

((A ^ B) ∨ C)

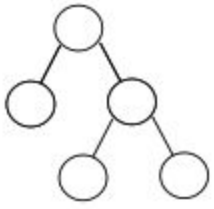
Output:

(A ∨ C) ^ (B ∨ C)

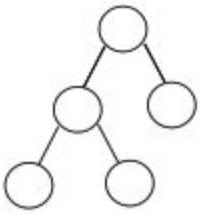
By far the most complex operation out of the bunch. The base case for this recursive function is a tree with the token '|' and one of its children with the token '&'.

There are three cases that are possible:

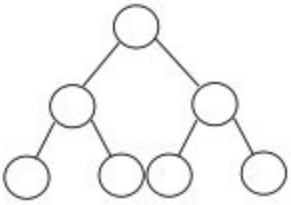
a) Singular proposition as left child with plural proposition as right child



b) Plural proposition as left child and singular proposition as right child



c) Plural proposition as left child and plural proposition as right child



In order to streamline the process of distribution of clauses, any time there was a singular clause child and plural clause as its counterpart sibling tree (e.g. figure a) and b)) we made its orientation into figure a).

Two ideas are at the heart of this algorithm: *deepcopy*s and mutual recursion. Utilizing Python *deepcopy* (which essentially copies the exact structure it is given) we were able to avoid needing to shuffle around references, and instead were able to cleanly edit copies of the trees we needed to. Because the essence of distributing disjunction over conjunction is simply pushing disjunctions downward (toward bottom of tree) while lifting conjunctions upward (toward top of tree), we needed to run DFS/in-order in an auxiliary function *topDownDFS()* each time we ran DFS in the primary function *DFSBottomUP()*.

The primary function *DFSBottomUP()*'s base case is that the current tree has 2 subsequent generations of children. Once recursed to the bottom of the tree, for each stack frame starting from the most recent call, the first step is to fix the current disjunction with the function *fixDisjunction()* (which uses those previously mentioned deepcopies) then call *topDownDFS()* on the current tree's children to ensure that the distribution did not result in another disjunction. Once in *topDownDFS()* it checks if any disjunctions are present in the tree's children; if yes, call *DFSBottomUp(),* if no, return to *DFSBottomUp()* from the initial call.

Results

Although a vast majority of test propositions will be converted flawlessly. Unfortunately a strange bug was encountered that was unexpected. Once we attempted to test the Horn clause we got the following result:

```
Input:
        ( !Y => ( !I & M ) ) & ( Y => I ) & ( ( ( I | M ) => H ) & ( H => G ) )
Output (actual):
        ['!!Y', '|', '!I', '&', 'M', '&', '!Y', '|', 'I', '&', '!I', '&', '!M', '|', 'H', '&', '!H', '|', 'G']
≡       ( (Y | !I ) & M & ( !Y | I ) & !I & ( !M | !H ) & ( !H | G ) )
```

This output is incorrect. The correct CNF form is:

(!Y | I) & (!I | Y) & (M | Y) & (!I | H) & (!M | H) & (!H | G)

This is especially strange because when provided the input:

```
Input:
        ( A | B ) & ( P => Q ) & ( C => D )
Output (actual):
        ['A', '|', 'B', '&', '!P', '|', 'Q', '&', '!C', '|', 'D']
≡       ( A | B ) & (!P | Q ) & ( !C | D )
```

This is the correct output. So we can rule out that the issue is multiple '&'s.

We hypothesize that the issue is specifically a singular clause that implies a plural clause with no further clauses in the parentheses. E.g. *( A => B ) & ( !C => ( !D & E ) ) & ( I | G )* did NOT WORK. However, *( A => B ) & ( !C => ( !D & E ) | Z ) & ( I | G )* DID work. Why this issue has occurred we are not positive, and further inspection might shed some light. However, that is for future work and outside the scope of this project due to time limitations.

We worked on the Python implementation of parser and CNF converter for cumulatively close to 40 hours. It was quite the herculean task, and although in the end our results were inconclusive, the amount of man hours that went into this attempt should cannot be overstated. However disheartened, we still had a solution up our sleeve. As a consequence of these issues, we have coded a working parser and CNF converter in Haskell. Haskell is a purely functional language that makes the task of parsing nearly trivial.

Feel free to test the standalone Python implementation. It will work nearly 90% of the time. However, for the sake of this project, please note that the fully functional parser and CNF converter is written in Haskell and can be found in the file "CNF_converter.hs" in the src folder.

**WalkSAT Satisfiability Checker (Extra Credit)**

- `walkSatEntails` is the function that uses the WalkSAT algorithm for deciding propositional entailment.
- `walkSatEntails` relies on the function `walkSat` which is the actual satisfiability checker. Hence `walkSatEntails` just negates the result of `walkSat` as proving unsatisfiability is required for deciding entailment.

- `walkSat` takes in a list of clauses, a probability P value of type `Float` and the number of max flips of type `Int`. It then creates a model of random True/False assignments to symbols using Haskell's Data.Map module.
- Unlike the pseudocode for WalkSAT, we use a recursive function called `applyFlip`. This function's execution corresponds to one iteration of the for loop.
- In each iteration we filter the clauses which are not satisfied by the current model. If there are no such clauses, we stop and return True implying satisfiability. Else we proceed to use the filtered clauses. Notice that we just return True instead of a model, as we are just concerned with the question of satisfiability.
- As per the algorithm, we randomly select an unsatisfied clause and then with probability P we flip the value in our model for a randomly selected symbol in our clause or with probability '1 - P' we flip the value of whichever symbol might maximize the number of clauses after a flip
- If we run out of max flips, we just return false implying unsatisfiability.

**`Test.hs`**

- This is the main program that runs and tests each of the inference methods on different examples.
- `runProblem` is the main driver function that tests each example. This function runs each of the 3 inference methods and generates 3 results. Each result can be one of the following values: TRUE, FALSE and MAYBE. TRUE implies that the query can be entailed from the knowledge base. FALSE implies that the query cannot be entailed from the knowledge base, but it's negation can be entailed. MAYBE implies that neither the query nor its negation could be entailed from the knowledge base. Hence notice that we run each inference method twice, the second one using the negated query.
- Finally, we print the results for each of the examples (More on this later).

**`Scanner.hs` (Extra Credit)**

- This file contains the types and implementation for a string tokenizer of well formed formulas.
- The main function in this file is `tokenize` which takes in a string and returns a list of tokens.
- Each token can be of 3 types:
  - A variable (Atomic sentence) of alphanumeric type
  - A logical connective: `['!', '&', '|', '=>', '<=>']`
  - A round opening or closing parenthesis: `['(', ')']`
- Failure to match any of the above type will raise an error and terminate the program

**`Parser.hs` (Extra Credit)**

- This file contains the implementation for parsing a list of tokens to form a tree defined by the `Prop` data type
- The main function to be of concern here is `parse` which takes in a string and returns a tree representation of the WFF. This function first tokenizes the string using the

`tokenize` function and then calls `buildTree` on the list of tokens to return a tree representation.

- The algorithm for creating the 'parse tree' was completely handwritten without any external references though it was revealed later to be very similar to this algorithm.
- `buildTree` reads the tokens one by one and uses 2 stacks for storing partial trees and operators.
- We also take into account the operator precedence as given in AIMA (¬, ∧, ∨, ⇒, ⇔) in cases where there are no parentheses.
- The error handling is not as robust as it raises a parsing error for only some cases where we read an unexpected token. If the input list of tokens does not have well formed parentheses our parser fails to detect that. Hence some care has to be taken when feeding formulas to our parsers such that the brackets our well formed

Eg.

Calling `parse "(P => Q) ^"` will raise an error

Calling `parse "(P => Q ^ R"` will NOT raise an error and will simply parse it while giving higher precedence to the conjunction.

**`Makefile`**

- This file compiles the Haskell source files to create a binary executable file `Test`
- We run this binary file to run our experiments.

## Experiments and Results

The program starts by showing the following output:

```
************* Welcome to Automated Reasoning *************


Propositional theorem proving tests


Press Enter to continue...
```

(Note: Our program doesn't listen to any input apart from waiting for the user to press enter)

After the user presses the enter key, the program runs the first problem: **`Modus ponens`**

The expected output should be:

```
Modus Ponens test.

Knowledge base: (P => Q) & P

Query: Q

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: TRUE

Ans with resolution: TRUE

Ans with SAT checking: TRUE

Press Enter to continue...
```

In the first problem we get TRUE for all the 3 inference methods as expected which means

$(P \Rightarrow Q) \ \& \ P \models Q$

Press enter for the next test: **Wumpus World test (Simple)**

The expected output should be:

```
Wumpus World test.

Knowledge base: !P11 & (B11 <=> (P12 | P21)) & (B21 <=> (P11 |
P22 | P31)) & !B11 & B21

Query: P12

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: FALSE

Ans with resolution: FALSE

Ans with SAT checking (WalkSAT): FALSE

Press Enter to continue…
```

In the wumpus world problem we get FALSE for all the 3 inference methods as expected which means 2 things:

1. `!P11 & (B11 <=> (P12 | P21)) & (B21 <=> (P11 | P22 | P31)) & !B11 & B21 ⊭ P12`
2. `!P11 & (B11 <=> (P12 | P21)) & (B21 <=> (P11 | P22 | P31)) & !B11 & B21 ⊨ !P12`

Hence from the wumpus world problem we prove that there is no pit at location [1, 2]

Press enter for the next test: **`Horn Clauses test.`**

The expected output should be:

```
Horn Clauses test.

Knowledge base: (Y => I) & (!Y => (!I & M)) & ((I | M) => H) &
(H => G)

Query: Y

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: MAYBE

Ans with resolution: MAYBE

Ans with SAT checking (WalkSAT): MAYBE


Horn Clauses test.

Knowledge base: (Y => I) & (!Y => (!I & M)) & ((I | M) => H) &
(H => G)

Query: G

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: TRUE

Ans with resolution: TRUE

Ans with SAT checking (WalkSAT): TRUE
```

```
Horn Clauses test.

Knowledge base: (Y => I) & (!Y => (!I & M)) & ((I | M) => H) &
(H => G)

Query: H

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: TRUE

Ans with resolution: TRUE

Ans with SAT checking (WalkSAT): TRUE


Press Enter to continue...
```

In this problem we define the following symbols:

```
-- Mythical: Y

-- Immortal: I

-- Mammal: M

-- Horned: H

-- Magical: G
```

Hence from the above output we conclude that the unicorn is definitely magical and horned but maybe mythical. We can also see that each of 3 inference methods agrees on the same result for each of the 3 queries.

In mathematical notation for logical entailment we have the following:

- `(Y => I) & (!Y => (!I & M)) & ((I | M) => H) & (H => G)` $\nvDash$ `Y`
- `(Y => I) & (!Y => (!I & M)) & ((I | M) => H) & (H => G)` $\nvDash$ `Y`
- `(Y => I) & (!Y => (!I & M)) & ((I | M) => H) & (H => G)` $\vDash$ `G`
- `(Y => I) & (!Y => (!I & M)) & ((I | M) => H) & (H => G)` $\vDash$ `H`

Press enter for the next test: **`The Labyrinth Test.`**

This is an extra 5th test we incorporated to test our implementations. We were introduced to this test in class and were asked to solve it by hand using resolution. Hence we ran this test using our problem solver to confirm correctness with our hand derived results and vice versa

The expected output should be:

```
The Labyrinth Test.

Knowledge base: !(G & (S => M)) & !(!G & !S) & !(G & !M)

Query: G

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: FALSE

Ans with resolution: FALSE

Ans with SAT checking (WalkSAT): FALSE


The Labyrinth Test.

Knowledge base: !(G & (S => M)) & !(!G & !S) & !(G & !M)

Query: M

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: MAYBE

Ans with resolution: MAYBE

Ans with SAT checking (WalkSAT): MAYBE


The Labyrinth Test.

Knowledge base: !(G & (S => M)) & !(!G & !S) & !(G & !M)

Query: S

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: TRUE

Ans with resolution: TRUE
```

```
Ans with SAT checking (WalkSAT): TRUE


Press Enter to continue...
```

In this problem we use the following interpretations:

```
   -- Gold road will get to the center: G

   -- Marble road will get to the center: M

   -- Stones road will get to the center: S
```

Hence from the above output we see that only the Road made of stones is guaranteed to lead us to the center of the Labyrinth.

(We skip writing down the mathematical notation as this is not a selected problem)

Press enter for the next test: **The Doors of Enlightenment. Smullyan's problem**

The expected output should be:

```
The Doors of Enlightenment. Smullyan's problem

Knowledge base: (A <=> X) & (B <=> (Y | Z)) & (C <=> (A & B)) &
(D <=> (X & Y)) & (E <=> (X & Z)) & (F <=> (D | E)) & (G <=> (C
=> F)) & (H <=> ((G & H) => A))

Query: X

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: TRUE

Ans with resolution: TRUE

Ans with SAT checking (WalkSAT): TRUE


The Doors of Enlightenment. Smullyan's problem

Knowledge base: (A <=> X) & (B <=> (Y | Z)) & (C <=> (A & B)) &
(D <=> (X & Y)) & (E <=> (X & Z)) & (F <=> (D | E)) & (G <=> (C
=> F)) & (H <=> ((G & H) => A))
```

```
Query: Y

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: MAYBE

Ans with resolution: MAYBE

Ans with SAT checking (WalkSAT): MAYBE


The Doors of Enlightenment. Smullyan's problem

Knowledge base: (A <=> X) & (B <=> (Y | Z)) & (C <=> (A & B)) &
(D <=> (X & Y)) & (E <=> (X & Z)) & (F <=> (D | E)) & (G <=> (C
=> F)) & (H <=> ((G & H) => A))

Query: Z

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: MAYBE

Ans with resolution: MAYBE

Ans with SAT checking (WalkSAT): MAYBE


Press Enter to continue...
```

From the output we conclude that X is a good door which the philosopher should choose.

Let KB = `(A <=> X) & (B <=> (Y | Z)) & (C <=> (A & B)) & (D <=> (X & Y)) & (E <=> (X & Z)) & (F <=> (D | E)) & (G <=> (C => F)) & (H <=> ((G & H) => A))`

In the KB each priest makes a statement which can be true or false depending on the priest type. Hence there is a biconditional between a priest and his/her statement and conjunction between all the biconditionals.

In mathematical notation we get:

- KB ⊨ X
- KB ⊭ Y
- KB ⊭ !Y
- KB ⊭ Z

- KB ⊭ !Z

Press enter for the next test: **The Doors of Enlightenment. Liu's problem**

The expected output should be:

```
The Doors of Enlightenment. Liu's problem

Knowledge base: (A <=> X) & (H <=> ((G & H) => A)) & (C <=> (A &
(B | C | D | E | F | G | H))) & (G <=> (C => (Anything)))

Query: X

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: TRUE

Ans with resolution: TRUE+

Ans with SAT checking (WalkSAT): TRUE


The Doors of Enlightenment. Liu's problem

Knowledge base: (A <=> X) & (H <=> ((G & H) => A)) & (C <=> (A &
(B | C | D | E | F | G | H))) & (G <=> (C => (Anything)))

Query: Y

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: MAYBE

Ans with resolution: MAYBE

Ans with SAT checking (WalkSAT): MAYBE


The Doors of Enlightenment. Liu's problem

Knowledge base: (A <=> X) & (H <=> ((G & H) => A)) & (C <=> (A &
(B | C | D | E | F | G | H))) & (G <=> (C => (Anything)))

Query: Z

Cases: 1  Tried: 1  Errors: 0  Failures: 0

Ans with model checking: MAYBE
```

```
Ans with resolution: MAYBE

Ans with SAT checking (WalkSAT): MAYBE
```

(end)

From the above we can conclude that X is a good door, and the philosopher should choose door X even when he lacked concentration.

Let KB = `(A <=> X) & (H <=> ((G & H) => A)) & (C <=> (A & (B | C | D | E | F | G | H))) & (G <=> (C => (Anything)))`

In the KB each priest makes a statement which can be true or false depending on the priest type. Hence there is a biconditional between a priest and his/her statement and conjunction between all the biconditionals. But the philosopher just correctly heard Priest A and H.

In the case of Priest C, we can assume that the second priest could be any one of the remaining priests, hence our biconditional will be: `(C <=> (A & (B | C | D | E | F | G | H)))`.

In the case of Priest G, we assume the conclusion in his statement to be anything, which means we introduce a new variable called “Anything”. Hence our formula will be

(G <=> (C => (Anything))).

Hence our final KB = `(A <=> X) & (H <=> ((G & H) => A)) & (C <=> (A & (B | C | D | E | F | G | H))) & (G <=> (C => (Anything)))`

In mathematical notation for logical entailment our results will be:

- KB ⊨ X
- KB ⊭ Y
- KB ⊭ !Y
- KB ⊭ Z
- KB ⊭ !Z

This proves that the philosopher had heard enough to make the correct decision of choosing door X.

To add any other tests add the following piece of code near the end of `Test.hs`:

```
runProblem <testName:String> <Knowledge Base:String> <Query:String>

Eg. runProblem “Modus Ponens Test.” “(P => Q) & P” “Q”
```

## Member Contributions

**Soubhik Ghosh:**

Implementation in Haskell

Experiments for Selected problem 1,2 and 3

**Richard Magnotti:**

Implementation in Python

Experiments for Selected problem 4 and Labyrinth

## References


- http://learnyouahaskell.com/
- http://hackage.haskell.org/
- Artificial Intelligence: A Modern Approach (3rd Edition) Textbook by Peter Norvig and Stuart J. Russell
- ±: heavy inspiration for this algorithm was taken from the Shunting Yard algorithm Python implementation from: https://runestone.academy/runestone/books/published/pythonds/BasicDS/InfixPrefixandPostfixExpressions.html