Introduction to Artificial Intelligence

COS30019

ASSIGNMENT 2 - INFERENCE ENGINE

Student Details

The team that created the inference engine program consisted of:

- Jaclyn Seychell (100585248)
- Dale de Silva (2146606)

Implemented Features

- Solving a knowledge base using Truthtables
- Solving a knowledge base using Forward Chaining
- Solving a knowledge base using Backward Chaining
- Propositional Logic Operators:
    - NEGATION
    - CONJUNCTION
    - DISJUNCTION
    - IMPLICATION
    - EQUIVALENCE

Enhancements

Backtracking in Chaining Methods

It was agreed that the chaining solution methods should be able to find a route to the query Literal even if initially led down a dead end. This lead to the idea of implementing backtracking within the backwards and forwards chaining.

If any sentence provided within the knowledge base resulted in the literal not been found within that sentence. In Backward Chaining the neededliterals stack would be cleared, as it would likely contain literals that contribute to a sentence that cannot be solved, and the previously found literal would be placed back onto the neededLiterals stack. This would result in research for another route. This happen repeatedly when the goal is not found until the neededLiterals stack is empty and the Query deemed unsolvable.

Unifying Literal References

Within the InferenceEngine class, a method called **unifyLiterals** was created that unifies all the literals within the given sentence. This feature ensures that all literals in every sentence will refer to the same object in memory thus making the literals objects directly comparable. In order to map each literal to an object of memory, within the method a test loop had to be created that compared the string name of the first Literal to the string name of the second Literal. If they matched the one within the sentence would be replaced with one that had already been found in the created literals array list, so they can reference the same object and thus location in memory.

This allows for far simpler syntax when comparing literals. It design to also allow changing the value of a Literal globally before or during calculations, however, nothing that utilized this second functionality was implemented.

Comprehensive Unit Testing

It was very important to the group that our program was well tested, as this ensured that we had the most accurate and efficient outcome as possible. This method of testing allows for less errors in the final product as we were debugging the system as we went.

While some tests were created during or after various aspects were created, test driven development also allowed us to develop tests *before* implementation of methods and classes in a way that helped clarify (between us and in general) the expected output of those methods and classes.

Many bugs in operators or solving logic were found through specific Unit Tests.

Bugs and Missing Features

No basic requirements of the task are missing.

Backtracking of chaining methods was initially implemented in Backward Chaining in a way that removed any chaining steps from the output that resulted in dead ends. It was later decided that this may have been a misinterpretation of the necessary output and was altered. All chaining steps are currently output even if they result in dead ends before being backtracked.

Design & Structure

Sentence Interpretation

In order to compartmentalize the necessary tasks and make linking the various functions easier, it interpreting each sentence string was offset to the Sentence class itself. It receives the string, parses it and internally creates Literal and Operator objects.

This meant that each Literal object created would be a different object in memory to the same Literal object in a different sentence. In order to unify this, the **unifyLiterals** method was created in InferenceEngine (describe previously). While this second pass over the literals is inefficient, it was chosen over the alternative of parsing all sentences in Inference Engine for better compartmentalization of the tasks and easier programming.

Operator Handling

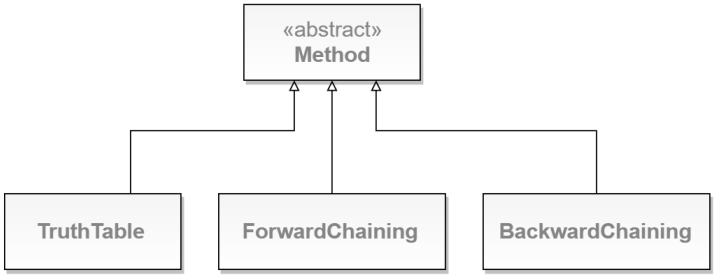
All strings representations of operators and the processes to evaluate using the operates were contained within the Operator class itself. This meant that throughout the program the static references in the class to operator strings could be utilized without the risk of errors rising from mismatched symbols or typos.

The inclusion of the **eval** function allows us to completely delegate each operator's function to itself. The eval function accepts reference to a stack of Booleans and applies the required operation on it. The caller doesn't need to know what the operation is or edit the stack itself.

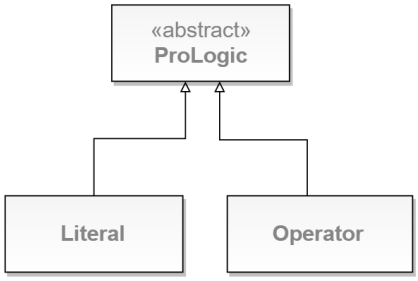
Inheritance and Polymorphism

In order to ensure each solution method implemented certain functions and features that the InferenceEngine was expecting from all of them, and abstract class of Method was created which each specific method then inherited from.

This also enabled the setup of one IMethod variable in InferenceEngine that would could confidently call the solve and other related functions on regardless of its actual object instantiation.



As each sentence would contain both Literals and Operators, and abstract class of ProLogic was created for them both to inherit from. This enabled an ArrayList of ProLogic objects to hold both Literals and Operators through Polymorphism. It also enabled various methods like **getName** to be implemented and used confidently across the two.



Unit Tests

- **Test_BackwardChaining**:- Within the backwards chaining test class numerous test cases were created. The main purpose of this test class was to check that the system was able to identify if the given sentence is a horn sentence. We tested both simple and complex sentences would fail if the given input is not a Horn Clause. Another test was created confirmed that multiple implications existing in the sentence "`p2=>p3&g=>y`" would still be interpreted as not a horn sentence.
  The backwards chaining test also tested for the standard solving method using two different routes, straight forward route, and deceptive route. The deceptive route implements some decoys literals in order to try and manipulate the route to taking the wrong path in order to find the solution. This checks that the program is able to backtrack itself in order to find the correct route in order to solve the given query.
  - NonHornSentence_Simple
  - NonHornSentence_Complex
  - NonHornSentence_MultipleImplications
  - StraightForwardRoute
  - DeceptiveRoute

- **Test_ForwardChaining** :- The forward chaining test class is very similar to the backwards chaining test class, as it does the same tests for checking if program is able to check if a sentence would be considered a horn class or not. It checks against simple and complex sentences as well as using multiple implications to solve the query.

- **Test_HornSentence**:- Within the horn sentence test class, a recognition test is created to test if the sentence declared would be considered a horn clause. This test ensures that the horn clause method is able to recognize itself before proceeding to continue on with the rest of the implementation. Another test is created to setup the literals by getting the correct right hand side literal from the horn sentence and ensure that it is separated from the left hand side of the sentence correctly.
  - Recognition
  - SetupLiterals

- **Test_InferenceEngine**:- The inference engine test class ensures that the program is able to read in the knowledge base as a string, test the literals to ensure that it is able to extract the appropriate literals from the knowledge base, and test that the literals called are the same objects of those in the inference engine's literals list (unified).
  General tests for correct success and failure instances were also created as well as tests against the specific examples given in the assignment sheet.
  - KnowledgeBaseInterpretation
  - LiteralInterpretation
  - LiteralUnifying

    - ForwardChainingSuccess
    - ForwardChainingFailure
    - BackwardChainingSuccess
    - BackwardChainingFailure
    - TruthTableSuccess
    - TruthTableFailure

    - TruthTableExampleSolve
    - ForwardChainingExampleSolve
    - BackwardChainingExampleSolve

- **Test_Operators** :- This test class demonstrates how each operator that was created was tested to ensure that it evaluates the Booleans correctly and manipulates the passed in stack correctly.
    - Negation
    - Conjunction
    - Disjunction
    - Implication
    - Equivalence

    - StackManipulation1
    - StackManipulation2

- **Test_Sentence**:- To ensure that the sentences strings passed in were always interpreted correctly, a test was created to test that all spaces were removed from the given knowledge base, and regardless if a sentence contains spaces or not, it will still produce the same result. Other tests check that the knowledge base is able to order sentences with negation and brackets correctly into Reverse Polish Notation.
    - Spaces
    - Literals
    - Basic
    - Negation
    - Brackets

- **Test_TruthTable**:- The truth table tests demonstrate that a truth table can be created and be populated by the number of literals that would be called to the createTruthTable method. As wells as the ability to access those values from the TruthTable.
    - CreateTruthTable
    - Basic1
    - Basic2

Acknowledgement / Resources

There was a number of resources that we used in order to create this final project. Those included:

- Reverse Polish Notation Starter Code
  - Steven Morris (Blackboard)

- RegEx reference for splitting the sentence
  - http://stackoverflow.com/questions/9856916/java-string-split-regex
  - https://www.cheatography.com/davechild/cheat-sheets/regular-expressions/
  - https://docs.racket-lang.org/guide/Looking_Ahead_and_Behind.html
  - http://www.regular-expressions.info/captureall.html

- jUnit Testing:
  - http://www.cavdar.net/2008/07/21/junit-4-in-60-seconds/

- Operator Precedence:
  - http://academics.triton.edu/faculty/ebell/2%20-%20Propositional%20Logic.pdf

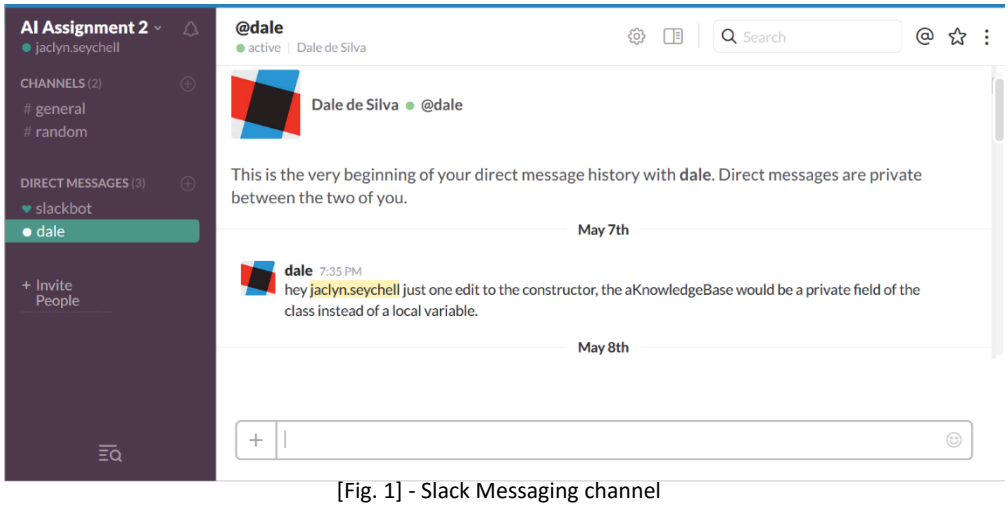
- Bitwise operations in Java
  - http://stackoverflow.com/questions/14145733/how-can-one-read-an-integer-bit-by-bit-in-java/14145767
- Recursion for originally populating the the TruthTable
  - https://sites.google.com/site/spaceofjameschen/home/recursion/truth-table-implementation-microsoft---recursion

- Using modulo operator for mathematical functions (originally used within the truth table to calculate the amount of rows.
  - http://www.dreamincode.net/forums/topic/273783-the-use-of-the-modulo-operator/

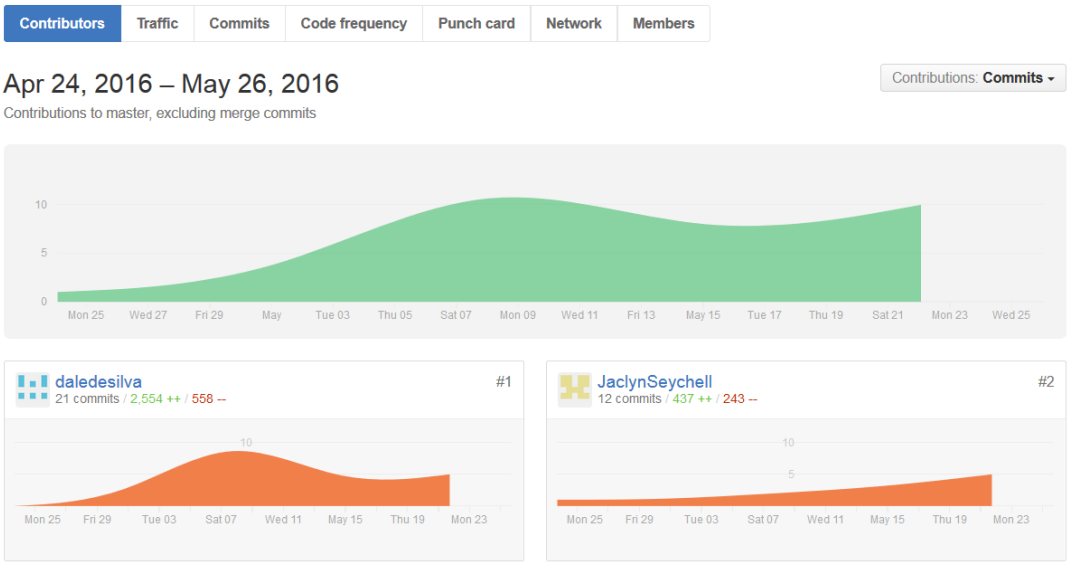
All resources used have been listed and described to the extent they have been used. Many were used as quick reference in order to understand the concept in order to create that section of code.

Teamwork Summary

In this project, it was extremely important to have good team work. In order to ensure that both members were able to effectively communicate with each other and have someone to contribute their work to, a GitHub repository and slack channel was created by Jaclyn for this assignment as demonstrated below by the provided screenshots.



[Fig. 1] - Slack Messaging channel



[Fig. 2] – GitHub repository and commitment history

As demonstrated by figure 2, both team members contributed to the assignment, and tried their best to help one another out. The tasks were divided into Task 1: backwards chaining and forwards chaining, and Task 2: Truth Table checking and report. Task 1 was implemented by Dale de Silva, while Jaclyn Seychell completed task 2. Between the two people, the split of tasks was tried to keep as even as possible. Throughout the project, issues arise where team members did not understanding certain concepts or plans that were to be implemented, and to ensure that everything stays on track, group members sought guidance from one another. As shown above, Dale was a high contributor to the final

product as he also implemented the structure of the program and thus grasped the concepts well. Jaclyn did however struggle creating the truth table but sought guidance from her tutor and team member for ideas and assistant. This was her first time creating a project in Java, and found it quite a challenge, but never less with the good team work allowed for a final implementation to be created.

To stay on top of the project, physical team meetings were held once or twice a week to work together on any issues that had arisen, and give a quick explanation to the new section that had been implemented since the last meeting. This allowed for both team members to understand what the new code was doing and gave time for team members to ask about any issues they were having. Instant messaging services such as Slack, allowed for team members to communicate at any time of the day, and was best used for asking questions about the functionality of specific code.

Overall the team worked well together and good communication was vital to producing a finished inference engine program. Both team members feel as though they have learnt a lot from this assignment and now have a good grasp of how forward and backwards chaining works and how truth tables can be implemented.