Introduction to Artificial Intelligence

Assignment 2 - Inference Engine

Student Details

The team that created the inference engine program consisted of:

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

Features/Bugs/Missing

During the implementation of the Inference Engine program, numerous features were created to extend the project to be able to handle more complex sentences being implemented through the knowledge base. In order to extend this program we were able to implement an operator class that is able to work with sentences that contain the equivalence operator, as well as parenthesizes. These extra operators allowed us to test our backwards and forwards chaining method to a higher degree, as even with a harder input, it should still be able to solve for the query (that is, if it is a solvable knowledge base) using the same logic as for the more simpler inputs.

Within the inference engine class, a method has been 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 become uniform within the sentence making it much easier for the sentence to interpret the sentence string internally as they have the same memory object each time it is run. In order to map each literal to an object of memory, within the method a test loop had to be created that compared the name of the literal to the current object to make sure that the same literal in different sentences is in a spot of memory that is different object. When a literal was found, it would be replaced within that sentence with one that had already been found in the created literals array list, so they can reference the same object. This continues for the length of the literals within the sentence declared. This feature allows for the program to be much more efficient as it uses less memory to run as the same literal will be mapped to the same memory object for every sentence inputted.

While making this solution it was agreed that the program should be as efficient as possible. This lead to the idea of implementing backtracking within the backwards and forwards chaining. The reason this was implemented was to check if any sentence provided within the knowledge base was able to be solved for the given query, if the literal had not been found within that sentence, the needed literals stack would be cleared as it contains literals that contribute to a sentence that cannot be solved. From here, the literal would be placed back onto the needed literal stack, in search for a different route. This sentence is removed as it is an unsolvable sentence, and removes the last added item to the path array, as it does not get you to your desired solution. By backtracking the program is able to reuse the memory created as objects are constantly being popped off the stack or cleared when they are no longer needed.

Test cases

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. Most tests classes were created after the implementation of the class, before moving on to create the next section of the program. This method of testing allows for less errors in the final product as you are debugging the system as you go. This was you will find more errors or bugs that are easier to fix as the whole program has not been implemented yet. The test classes that were created were:

- **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. The test consist of both simple and complex sentences that should fail their tests as the given input is not a horn sentence. Another test was created that allowed for multiple impactions to be implemented into the test sentence such as "`p2=>p3&g=>y`" but as this sentence is not a horn sentence the test should fail. 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.

- **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_General**:- The general test class created, creates a simple test to see if the forward chaining query and knowledge base will succeed or fail when implemented.

- **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.

- **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.

- **Test_Operators** :- This test class demonstrates how each operator that was created was tested to ensure that it is pushed onto the stack as "false" to indicate it is not a literal, and that the operator is able to identify itself.

- **Test_Sentence**:- To ensure that the sentences are all readable, 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 identify operator's negation and brackets when reading the sentence.

- **Test_TruthTable**:- The truth table test demonstrates that a truth table can be created and be populated by the number of literals that would be called to the createTruthTable method.

Acknowledgement / Resources

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

- RegEx reference for splitting the sentence
  - http://stackoverflow.com/questions/9856916/java-string-split-regex
  - https://www.cheatography.com/davechild/cheat-sheets/regular-expressions/
  - http://www.2ality.com/2013/08/regexp-g.htm
  - https://docs.racket-lang.org/guide/Looking_Ahead_and_Behind.html
  - https://regex101.com/r/jH4nC4/2
  - 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

Example of use within the final code:

```
//test
Int getBit(int aNum, int aPos) {
   Return (aNum >> aPos)&1;
}
//for printing test
Integer.toBinaryString(int i);
```

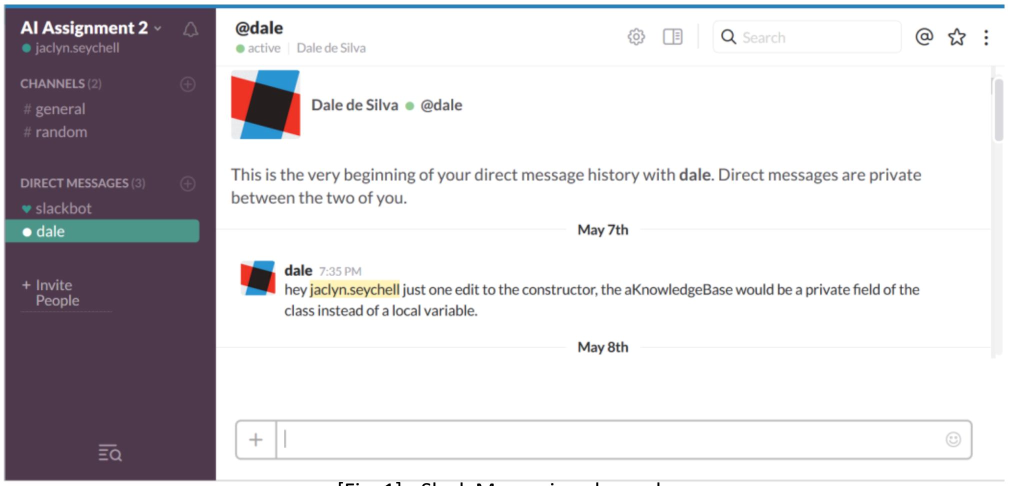
- Truth Table using recursion for populating the table
  - https://sites.google.com/site/spaceofjameschen/home/recursion/truth-table-implementation-microsoft---recursion

- Using modulo operator for mathematical functions (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.

Team work Summary

In order to achieve 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.



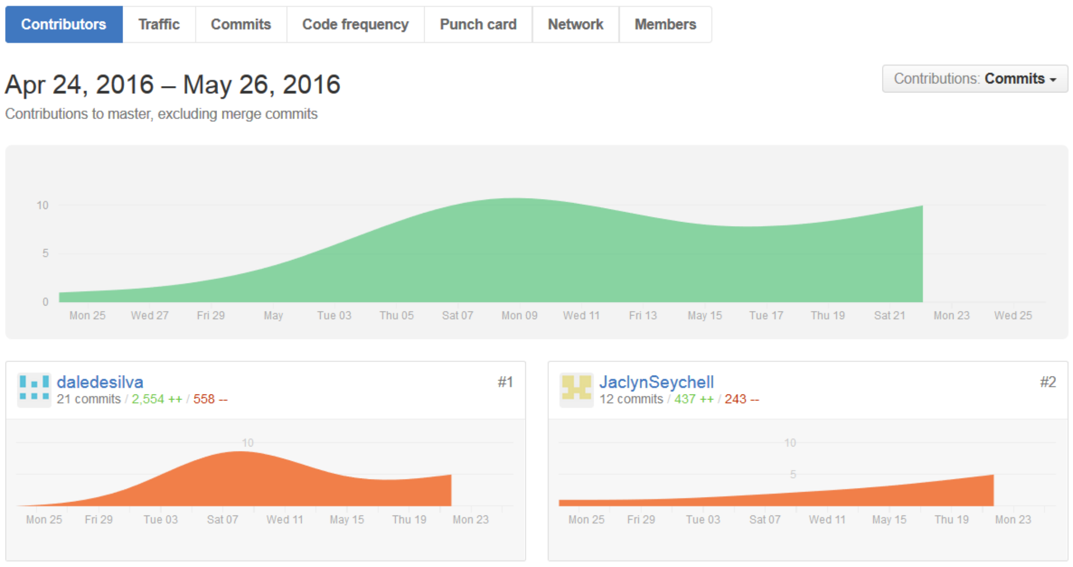
This is the very beginning of your direct message history with **dale**. Direct messages are private between the two of you.

May 7th

dale 7:35 PM
hey jaclyn.seychell just one edit to the constructor, the aKnowledgeBase would be a private field of the class instead of a local variable.

May 8th

[Fig. 1] - Slack Messaging channel



Apr 24, 2016 – May 26, 2016

Contributions to master, excluding merge commits

daledesilva #1
21 commits / 2,554 ++ / 558 --

JaclynSeychell #2
12 commits / 437 ++ / 243 --

[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 had grasped the concept and knowledge to complete this project quite 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 and 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 at Swinburne University 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 member to understand what the code was exactly doing and gives time for team members to ask about any issues they are 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 a concept.

Overall the team worked well together and was able to produce a finished inference engine program, due to good team work and communication. 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.