

**Unit Name:** Introduction to AI

**Unit Code:** COS30019

**Title:** Assignment 2 (Inference Engine)

**No of Group (ESP):**

**Student1:**

NAME: S M Ragib Rezwan

ID: 103172423

**Student2:**

NAME: Linh Vu

ID:……………

Contents

[Features/Bugs/Missing: 2](#_Toc135199078)

[TestCases: 2](#_Toc135199079)

[Achknowledgements/Resources 2](#_Toc135199080)

[Notes: 2](#_Toc135199081)

[Research: 3](#_Toc135199082)

[Team Summary Report: 3](#_Toc135199083)

## 

## Features/Bugs/Missing:

This section will be utilized in order to discuss about the various functionalities that our program performs, alongside the bug that had been noticed during testing (along with the way we corrected it). The Overall structure of our program can be expressed using the following UML structure:



**Figure 1: UML structure of iengine program**

Furthermore, in order to run our program, one must the following command through command line interface:  
**iengine <method> <filename.txt>**

* where **iengine** is the name of the program in .exe format,
* **method** is the name of inference engine that will be run (i.e TT, FC, BC and DPLL) and,
* **filename** is the name of the file that would contain the TELL (ie the knowledge base or KB) and ASK (ie the query or preposition that our program will try to entail from the KB).

**Note:** The TELL can be in any format delimited by “;” and doesn’t necessarily need to be horned form. But if it is in horned form, then it will not be able to perform Forward-Chaining (FC) and Backward-Chaining (BC) as they only work with horned clause (either book or fc and bc information reference). Also, for DPLL, the input needs to be in Conjunctive Normal Form (CNF) where each space would delimit the disjunctions while each new line seperates the conjunction clause------[NEED LINH’s help to set this one into proper format]

**Feature1: Use of ANTLR:**

In order to develop this software, ANTLR (Another Tool for Language Recognition ) had been used as a parser generator(<https://en.wikipedia.org/wiki/ANTLR> ). A parser generator helps perform syntactic analysis ( ie analyzing the provided string of symbols with the given rules of grammar (<https://builtin.com/data-science/introduction-nlp>, <https://en.wikipedia.org/wiki/Parsing> ))… ) by taking in the grammar file in Backus-Naur form (BNF) and creating the course code of a parser that can be used for the programming language (<https://en.wikipedia.org/wiki/Compiler-compiler> ). The Parser file is basically the code that performs the syntactical analysis by taking the inputted information (ie the string of symbols) and assigning meaning or value to it (which, in turn, enables the rest of the software to properly process it).

In our software, ANTLR has been utilized as it was python compatible and also because it enabled us to handle inputs of KB (Knowledge base) or TELL in multiple forms (ie for horned clause, CNH, normal prepositional clause form, etc.) in a structured manner, rather than hard-coding a parser.

Currently it works as follows:……NEED LINH’s help to understand how parser works step by step:

Step 1:…..

Step2:….

(either steps or pseudocode…both will work….need to know for all the files…ie lexer, parser and tree parser)

**Feature2: Implimentation of Truth Table:**

In our software we have implemented the truth table (TT). TT is basically a mathematical table that uses logic to ensure whether or not a propositional expression is logically valid (true for all legitimate input values) (<https://en.wikipedia.org/wiki/Truth_table> ). In our program it is used to verify whether or not our ASKed proposition can indeed be entailed from the given KB in TELL and for how many models. This has been implemented in the following manner:

:……NEED LINH’s help to understand how it works step by step:

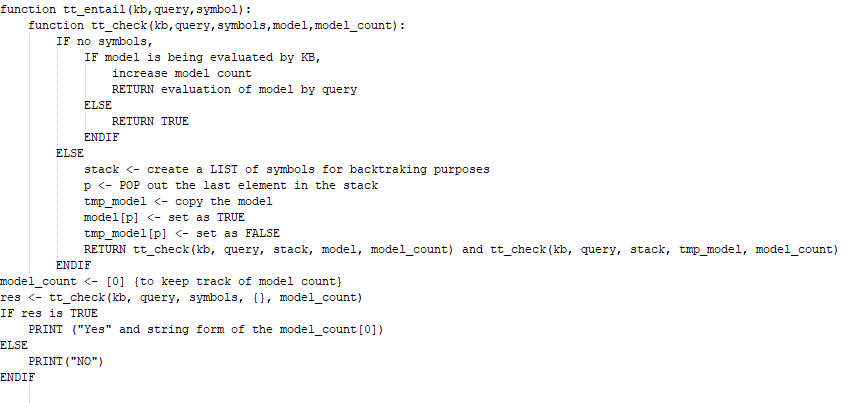
Step 1:…The users calls upon the TT iengine by using the following command:

iengine TT <filename.txt>..

Step2:….This calls the main function which creates the PLAgent object, assigns the file to it and asks it to interpret the method.

Step3: The PLAgent calls the PLInterpreter file which utilizes the PLParser file (alongside PLVisitor file, AtomExpression, ConjunctionExpression, NegationExpression, ImplicationExpression, BiconditionalExpression, DisjunctionExpression, HornCaluse, TellContext, ASKContext, HornTellContext, HornAskContext, etc) to analyse the input and assign each symbol and preposition their own meaning. Furthermore, the PLAgent file also calls the KB to set itself, (using the meaning that has been assigned to each symbol and their proposition), being calling up ttenum (ie the file that has the truth table logic).

Step4: When, ttenum is called, it basically goes through KB and checks whether the preposition noted in ASK does entail from KB or not and for how many model. The internal logic of the algorithm can be understood using the pseudocode noted below:



**Figure 2: Pseudocode for Truth table**

**Note:** Currently the truth table is also keeping track of time and memory it is using whilst running. This had been done in order to perform both testing and also to measure and analyse its performance (details about this is spoken in Research1). But these are not being mentioned in the above pseudocode as it is not needed for the Truth table to work.

Hence, it results in the two following outputs:

1. If it is possible to entail the ASKed preposition from the TELLed KB, it will say:

**“YES : N”** where N is number of models where KB entails the proposition.

1. If it is not possible, it says

**“NO”**

**Feature3: Implementatino of Forward chaining**

In our software we have implemented Forward chaining (FC). Forward chaining is basically a way of checking whether or not a proposition is entailed by a knowledge base (written using horned clause) by applying inference rules (reference: the book). It does this by going through the clauses provided in the KB (starting from the terminal node proposition), using inference rules to extract more data, until it reaches the proposition that has been asked (<https://en.wikipedia.org/wiki/Forward_chaining> ). In our program it is used to verify whether or not our ASKed proposition can indeed be entailed from the given KB in TELL, along with the list of propositional symbols (entailed from KB) that has been found during the execution of the algorithm. This has been implemented in the following manner:

:……NEED LINH’s help to understand how it works step by step:

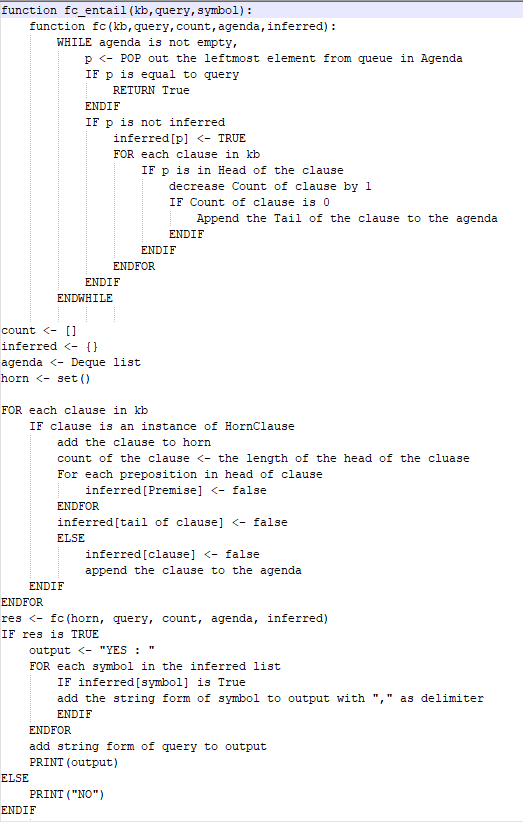
Step 1:…The users calls upon the TT iengine by using the following command:

iengine FC <filename.txt>..

Step2:…. This calls the main function which creates the PLAgent object, assigns the file to it and asks it to interpret the method.

Step3: The PLAgent calls the PLInterpreter file which utilizes the PLParser file (alongside PLVisitor file, AtomExpression, ConjunctionExpression, NegationExpression, ImplicationExpression, BiconditionalExpression, DisjunctionExpression, HornCaluse, TellContext, ASKContext, HornTellContext, HornAskContext, etc) to analyse the input and assign each symbol and preposition their own meaning. Furthermore, the PLAgent file also calls the KB to set itself, (using the meaning that has been assigned to each symbol and their proposition), being calling up fc (ie the file that has the forward chaining logic).

Step4: When, fc is called, it basically goes through KB and checks whether the preposition noted in ASK does entail from KB or not and notes the list of propositional symbols (entailed from the KB) that it has found during the execution. The internal logic of the algorithm can be understood using the pseudocode noted below:



**Figure 3: Pseudocode for Forward Chaining**

**Note:** Currently the Forward chaining is also keeping track of time and memory it is using whilst running. This had been done in order to perform both testing and also to measure and analyse its performance (details about this is spoken in Research1). But these are not being mentioned in the above pseudocode as it is not needed for the forward chaining to work.

Hence, it results in the two following outputs:

1. If it is possible to entail the ASKed preposition from the TELLed KB, it will say:

**“YES : a,b,c,d,….”** where a,b,c,d,… are the propositional symbols that had been entailed from KB during the execution of the algorithm.

1. If it is not possible, it says

**“NO”**

**Note:** During the initial testing of the algorithm, it had been noticed that the output of the algorithm did not match the one given in test1.txt (given in assignment 2 page (REF)). Hence in order to correct it, the list for the agenda had been sorted as deque and leftmost element was always popped out. Details regarding this aspect have been discussed in Bugs section.

**Feature4: Implementation of Backward chaining**

In our software we have implemented Backward chaining (BC). It is similar to FC in the fact that it is also restricted to horned clauses and that it checking whether or not a proposition is entailed by a knowledge base (written using horned clause) by applying inference rules (reference: the book). But it differs from FC as it works in the opposite direction. In BC, it at first tries to make the proposition that is being ASKed True. If it is not possible, it checks the implications that be used to infer the proposition. If all the premises, leading to that proposition can be proven as True, then the proposition is True. If not check the implications that can be used to infer the premises. This is continued recursively, until it reaches the terminal propositions. In our program, it is used to verify whether or not our ASKed proposition can indeed be entailed from the given KB in TELL, along with the list of propositional symbols (entailed from KB) that has been found during the execution of the algorithm. This has been implemented in the following manner:

:……NEED LINH’s help to understand how it works step by step:

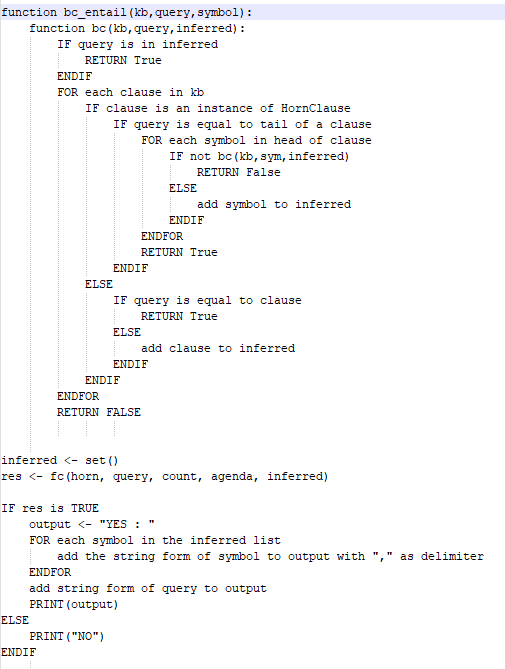
Step 1:…The users calls upon the TT iengine by using the following command:

iengine FC <filename.txt>..

Step2:…. This calls the main function which creates the PLAgent object, assigns the file to it and asks it to interpret the method.

Step3: The PLAgent calls the PLInterpreter file which utilizes the PLParser file (alongside PLVisitor file, AtomExpression, ConjunctionExpression, NegationExpression, ImplicationExpression, BiconditionalExpression, DisjunctionExpression, HornCaluse, TellContext, ASKContext, HornTellContext, HornAskContext, etc) to analyse the input and assign each symbol and preposition their own meaning. Furthermore, the PLAgent file also calls the KB to set itself, (using the meaning that has been assigned to each symbol and their proposition), being calling up bc (ie the file that has the backward chaining logic).

Step4: When, bc is called, it basically goes through KB and checks whether the preposition noted in ASK does entail from KB or not and notes the list of propositional symbols (entailed from the KB) that it has found during the execution. The internal logic of the algorithm can be understood using the pseudocode noted below:



**Figure 4: Pseudocode for Backward Chaining**

**Note:** Currently the Backward chaining is also keeping track of time and memory it is using whilst running. This had been done in order to perform both testing and also to measure and analyse its performance (details about this is spoken in Research1). But these are not being mentioned in the above pseudocode as it is not needed for the forward chaining to work.

Hence, it results in the two following outputs:

1. If it is possible to entail the ASKed preposition from the TELLed KB, it will say:

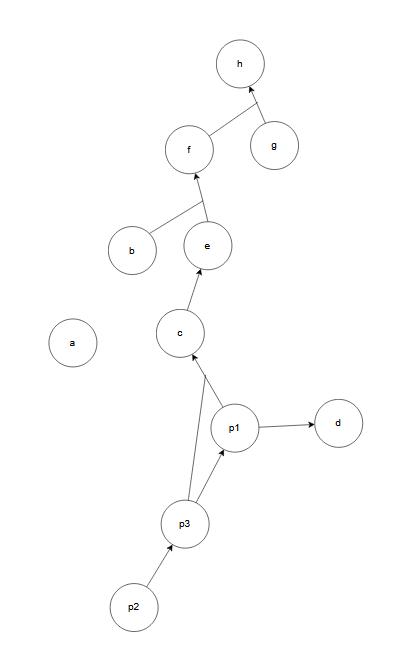
**“YES : a,b,c,d,….”** where a,b,c,d,… are the propositional symbols that had been entailed from KB during the execution of the algorithm.

1. If it is not possible, it says

**“NO”**

Bug1: Storage Order issue

This had been noticed when testing forward chaining algorithm for the test1.txt that had been provided to us in the assignment 2 module (REF). There we had noticed that although our algorithm was being able to correctly assess that the proposition is entailed by KB, it was giving us the wrong list of proposition. So, we decided to draw and resolve the algorithm by hand to see what was actually going on:



**Figure 5: Diagram created to check Forward chaining for text1.txt file**

In the diagram, we had noticed the terminal nodes were a, b, p2 and g. So according to the algorithm’s logic these will be checked first.

Then it will discard a, decreasing path value from b and e to f by 1 (so now it will only have 1), decreasing path from p2 to p3 by 1 (so now it will have 0) and decreasing the path from g and f to h by 1 (so now it will only have 1).

Then it will see p3 and go through it. This will result in path between p3 to p1 decreasing by 1 (so now it will be 0) and path between p3 and p1 to c decreasing by 1 (so now it will be 1).

Then it will see p1. This will cause the path from p1 and p2 to c to decrease by 1 (so now it will be 0) and path from p1 to d to decrease by 1 (so now it will be 0).

Here it will see that d has been detected and would stop the algorithm. Thus it will result in the following propositions being listed:

a,b,g,p2,p3,p1 and d

But in the assignment 2 module ((REF)), it can be seen that g is not included!

Then we realized that although g was a terminal node in the diagram, according to the syntax of the information in file, only a,b and p2 were terminal nodes. So we updated the code accordingly.

Even so, output still didn’t match the desired output that we were supposed to receive

[ASK LINH whether she has a copy of the previous version of FC where the results were different for iengine when compared to one given in assignment page and put it here as figure]

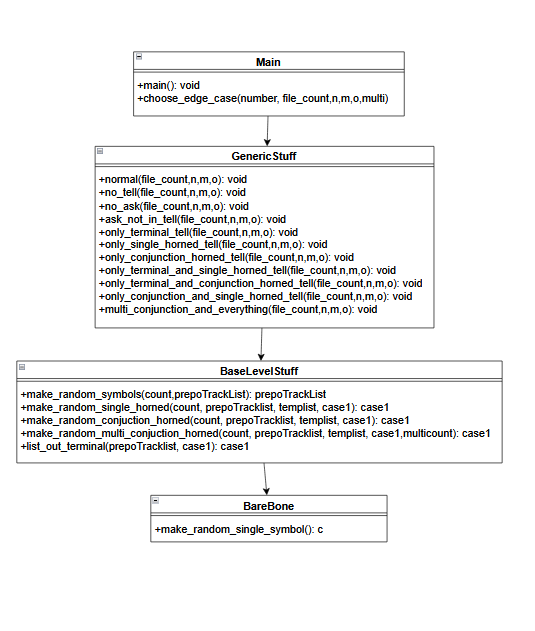
So, we decided to further investigate the matter. That’s when we realized that …..ordering issue….say how it initially was…how it is now… REF [NEED LINH’s help to see how the program was before and the change that we had made to ensure it gives same output….it was basically ordering of one of the collections as initiall we had set as random or smth but later we set a fixed order I guess? Neeed to reconfirm about the matter with her]

## TestCases:

Whilst developing the program, we were uncertain on whether or not the parser files and the algorithms were indeed working as intended. Furthermore, we also had the plan of measuring the performance aspects of the algorithms (i.e. time and memory usage of the algorithms during runtime) to ascertain the theories spoken about it.

Hence in order to perform that in a dynamic manner, we created the InferenceEngineMapMaker program. The UML of the program is as follows:

[CURRENTLY ONLY DREW UML AND EXPLAINED FOR ALL VERSION OF HORNED CLAUSE. WASN’T ABLE TO PROPERLY CODE IT TO WORK FOR NORMAL CLAUSE DUE TO LACK OF TIME. IF NOT POSSIBLE TO FINISH UP NORMAL CLAUSE FOR TRUTH TABLE TESTING< THEN JUST GO TO THAT FILE AND COMMENT OUT THE IMPLICATION RELATED STUFF] -----for Future Ragib to deal with



**Figure 6: UML for test case generator**

Overall, the program works in the following manner:

Step 1: It accepts inputs in the following format:

<filename> <TestCaseTypeNo> <TestCaseTotalFileCount> <TotalPrepositionSymbolCount> <TotalSingleHornedClauseCount> <TotalConjuectionHornedClauseCount> <TotalMultiConjuctionHornedClauseCount>

In this fomat:

* Filename is basically generator file that is being called,
* TestCaseTypeNo determines which type of test case will be created. These can be:
  1. Standard case [This creates file where TELL’s KB has horned clause with conjunction (ie a &b => c format), single horned clause (ie a => b format), terminal nodes (ie a) and ASK has a preposition that exists in the KB noted in TELL
  2. No TELL case [This creates file where only ASK with a preposition exists ]
  3. No ASK case [This creates file where ASK section doesn’t have any preposition or symbol ]
  4. ASK Not In Tell [ This creates file where the ASKed preposition doesn’t exist in TELL]
  5. Only Terminal Tell[This create file where TELL only has terminal nodes and ASK has preposition that exists in TELL]
  6. Only Single Horned tell [This create file where TELL only has single horned clause and ASK has preposition that exists in TELL]
  7. Only Conjunction Horned tell [This create file where TELL only has Conjunction horned clause and ASK has preposition that exists in TELL]
  8. Only Terminal and Single Horned tell [This create file where TELL only has single horned clause and Terminal nodes and ASK has preposition that exists in TELL]
  9. Only Terminal and Conjunction Horned tell [This create file where TELL only has conjunction horned nodes and Terminal nodes and ASK has preposition that exists in TELL]
  10. Only Conjunction and Single Horned tell [This create file where TELL only has conjunction horned clause and single horned clause and ASK has preposition that exists in TELL]
  11. Multi conjunction and everything[This creates file where TELL’s KB has horned clause with multiple conjunction (ie a &b&c&d&e&f&…. => z format), single horned clause, terminal nodes and ASK has a preposition that exists in TELL
* TestCaseTotalFileCount determines number of files that will be generated
* TotalPrepositionSymbolCount determines the total number of prepositional symbols that will be used to create the KB
* TotalSingleHornedClauseCount determines the total number of single horned clauses that will created
* TotalConjuectionHornedClauseCount determines the total number of conjunction horned clause that will be created
* TotalMultiConjuctionHornedClauseCount determines the total number of multi conjunction horned clause that will be created

Step2: When it is inputted, it passed into main when assigns the parameters to their respective variables before passing it to choose\_edge\_case

Step3: choose\_edge\_case decides sees which case type it is and calls the relevant methods in generic stuff file.

**Note:** for normal case or case1, it will call the +normal(file\_count,n,m,o) method

Step4: Once inside the relevant method in the generic file, it will create a txt file with relevant name, call upon relevant methods in Base level stuff file and BareBones file in order to create various prepositions and clause that will be inserted into the file and outputs it

**Note:** for the normal case, it will create a file called testNormalN (where N is no of the file) and empty list to track preposition (prepoTrackList), empty list to track current KB (case1), empty list to temporarily hold prepositions or KB(tempList). Then it will populate prepoTrackList with random prepositional symbols using Base Level Stuff. After this, it will append the list case1 with random single horned from the prepoTrackList using Base Level Stuff, and then also append the list case1 with random conjunction horned from the prepoTrackList using Base Level Stuff. Later on it will also append the list case1 with all terminal nodes from the prepoTrackList using Base Level Stuff and randomly choose a preposition from prepotracklist and store it in k. Once all of this is done, it will write “TELL”, all elements in case1, “ASK”, and k as new lines in the file

**Note:** The base level stuff contains the logic that will be used to make random symbols, random single horned and random conjunction horned cases, whilst the Bare Bones contains the logic to make single random symbols (used by random symbol function in base level)

This allows us to generate not only files of various edge cases, but also create desired number of files with desired numbers of preposition ( ie the terminal nodes), single horned clause, conjunction horned clause and multi horned clause. Hence, it provides us with the flexibility that we need in both testing for bugs and also in performing research on performance aspects of the algorithms (noted in details in research 1)

## Achknowledgements/Resources

## Notes:

## Research:

**Research1:**

Implementation of time counter and memory usage for performance measure for all the algorithms implemented

What is it?

Why did you do it…..Fc data driven…bc goal driven…truth table exponential enum……wanted to find out whether it was indeed the case + wanted to see the the true amount of difference as…..”say a quote or smth about seeings is beliving….of being abel to understand the time and memory complexity ++ add reference that says how long it should usually take relative to one another and why…

How did you implement it? ---mention a bout test case generator +++ say that detaisl about generator mentioned in test case section

Pseudocodes for automated scripts

…. 100 test cases…. Compared performance measure for all of them using normal case….vs…only horned conjuction…vs…only conjuction….vs only terminal…..to see which one would be solved fastest and would take the least amount of memory….. + also checked for each each using

Total number of symbol 1000

+ 1)10 clauses in each case

+ 2) 25 clauses in each case

+ 3) 50 clauses in each case

Input…output

----Need ot have output in graphical format

**Research2:**

Implimentation of updating of Knowledge base for Truth table, Forward chaining, and backward chaining

What is it?

How did you implement it? Why?

Pseudocodes ?

Input…output

**Research3:**

+ say about looking up implementation of DPLL….realised it only worked with CNF forms…so currently set it up as follows…. + pseudo codes

## Team Summary Report: