**Unit Name:** Introduction to AI

**Unit Code:** COS30019

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

**No of Group (ESP):** COS30019\_A02\_T033

**Student1:**

NAME: S M Ragib Rezwan

ID: 103172423

**Student2:**

NAME: Linh Vu

ID: 103519240

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:



++Need to update it with DPLL

***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 not 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). Furthermore, although DPLL has restriction of working only work Conjunctive Normal Form (CNF), currently in the software a resolution algorithm has been attached with it to ensure it can convert the information in the TELL into its needed CNF form before performing DPLL.

**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).

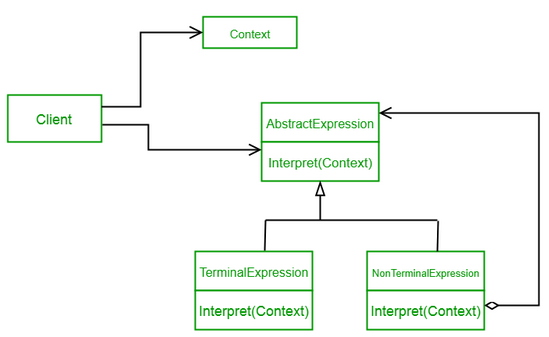
In our software, ANTLR has been utilized as it was python compatible and also because it enabled us to handle inputs of KB (Knowledge base) or TELL in multiple forms (ie for horned clause, normal prepositional clause form, etc.) in a structured manner, rather than hard-coding a parser. Furthermore, it also enabled us to develop the feature to read in and update KB and queries (whose details have been noted in Research 2 section)

Currently it works as follows:

Step 1: The Main function creates the PLAgent object, assigns the file to it and asks it to interpret the method

Step2: The PLAgent calls the PLparser file which takes in the preset grammar (in the PL.g4 file) using PLexer (i.e. the file that performs Lexical analysis which basically means breaking down the inputted strings into tokens <https://en.wikipedia.org/wiki/Lexical_analysis>) and creates the CST (Concrete syntax tree) representing them in a tree-like form.

Step3: This, in turn, is utilized by the PLInterpreter, using the various abstract expression files (i.e AtomExpression, ConjunctionExpression, NegationExpression, ImplicationExpression, BiconditionalExpression, DisjunctionExpression, HornClause, TellContext, ASKContext, HornTellContext, HornAskContext, etc) and the interpreter design pattern (FIG 1)) in order to create AST (Abstract Syntax Tree) which are syntactic representation of the initially inputted code (<https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/> ). These are then stored as KB and query objects respectively.



***Figure 2: Interpreter Design pattern***

**Note:** In the code, the grammer passed was for prepositional logic whilst the expression classes where for ASK, TELL, Atomic expression, horn clause expression, conjunction, disjunction, etc. Hence the AST produced in the end is basically the prepositional symbols and its clauses that were noted under TELL and ASK and thus will be passed to KB and query accordingly.

**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:

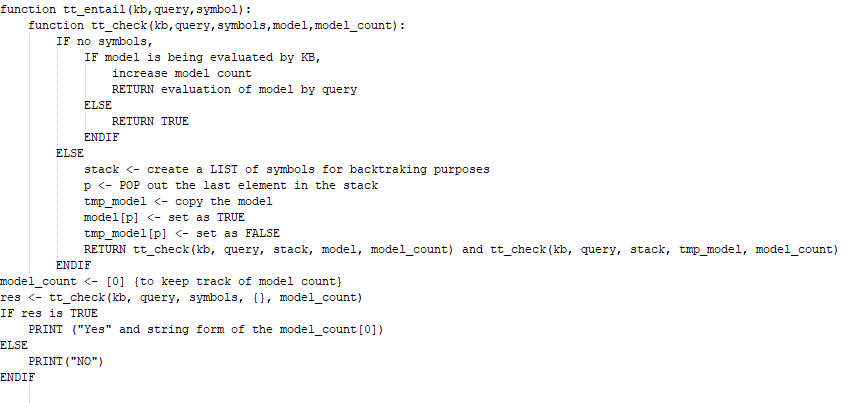
Step 1: The user calls upon the TT iengine by using the following command:

**.\iengine.exe 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 and PLexer files (alongside PLVisitor file, PL.g4 file, AtomExpression, ConjunctionExpression, NegationExpression, ImplicationExpression, BiconditionalExpression, DisjunctionExpression, HornCaluse, TellContext, ASKContext, HornTellContext, HornAskContext, etc) to analyse the input and assign each symbol and preposition with their own meaning using the grammer and context files (See feature 1 for details). Furthermore, the PLAgent file also calls the KB to set itself, (using the meaning that has been assigned to each symbol and their proposition), before 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 3: 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:

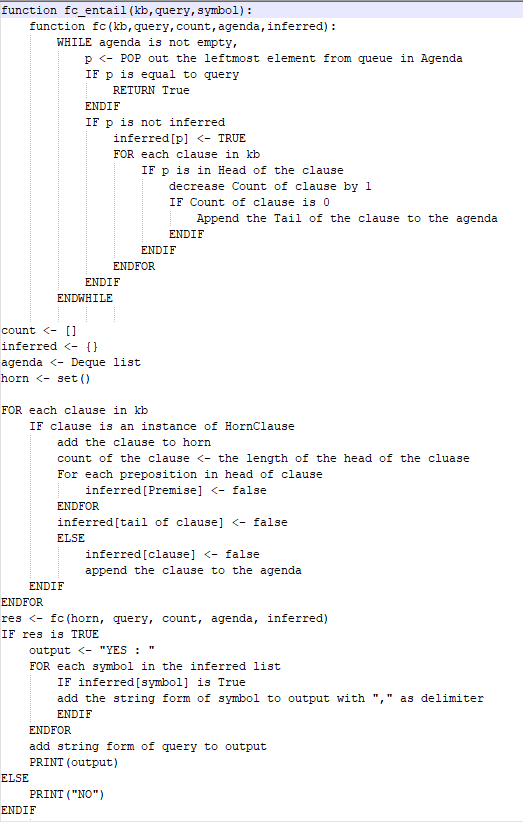
Step 1: The user calls upon the FC iengine by using the following command:

**.\iengine.exe 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 and PLexer files (alongside PLVisitor file, PL.g4 file, AtomExpression, ConjunctionExpression, NegationExpression, ImplicationExpression, BiconditionalExpression, DisjunctionExpression, HornCaluse, TellContext, ASKContext, HornTellContext, HornAskContext, etc) to analyse the input and assign each symbol and preposition with their own meaning using the grammer and context files (See feature 1 for details). Furthermore, the PLAgent file also calls the KB to set itself, (using the meaning that has been assigned to each symbol and their proposition), before calling up fc (i.e. 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 4: 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 analyze 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:

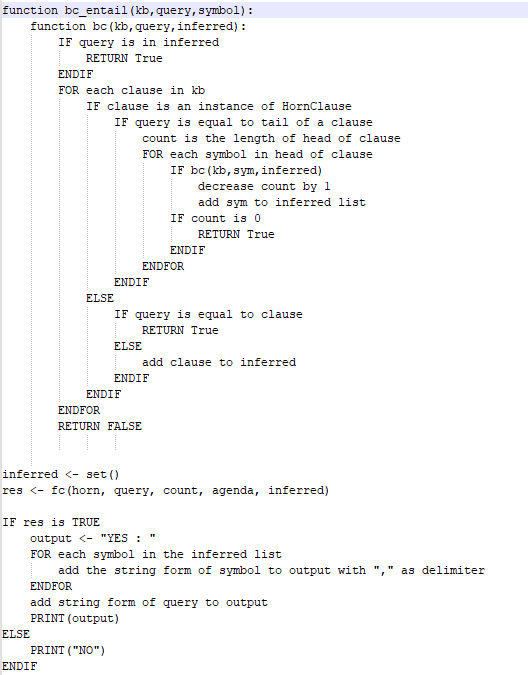
Step 1: The users calls upon the BC iengine by using the following command:

**.\iengine.exe bc <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 and PLexer files (alongside PLVisitor file, PL.g4 file, AtomExpression, ConjunctionExpression, NegationExpression, ImplicationExpression, BiconditionalExpression, DisjunctionExpression, HornCaluse, TellContext, ASKContext, HornTellContext, HornAskContext, etc) to analyse the input and assign each symbol and preposition with their own meaning using the grammer and context files (See feature 1 for details). Furthermore, the PLAgent file also calls the KB to set itself, (using the meaning that has been assigned to each symbol and their proposition), before 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 5: 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:

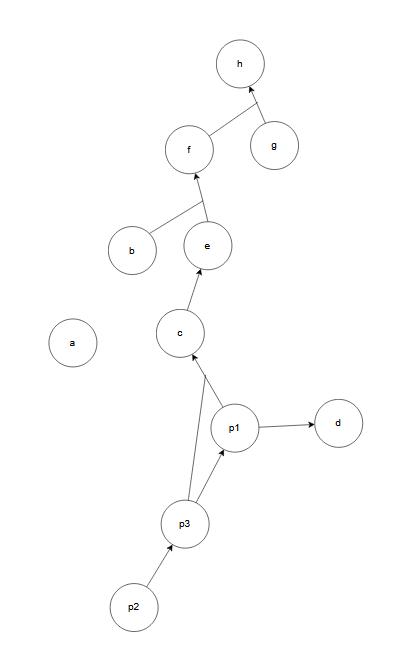
**“YES : a,b,c,d,….”** where a,b,c,d,… are the propositional symbols that had been entailed from KB during the execution of the algorithm.

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 6: 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 to only consider the single prepositions noted in the TELL as terminal nodes.

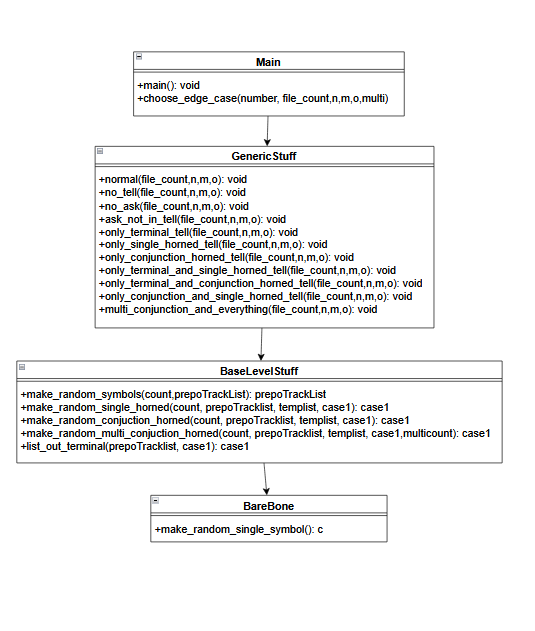
Even so, output still didn’t match the desired output that we were supposed to receive. So, we decided to further investigate the matter. That’s when we realized that our KB had been set as a set and not a list! And in python, a set is unordered (which basically means when you try to bring out the items, they will not come in the order they had been inserted into it <https://www.freecodecamp.org/news/python-set-vs-list/> )!

So, we changed it to store data as a list instead. This ended up providing the same result as the one noted in assignment 2 page (REF). But when we checked the time taken, it was almost twice slower than original! Hence, in the end we reverted it back to before as the assignment (REF) had stated that even different outputs would still be correct and in that point of view, using set for KB is actually optimizing the program!

## 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:



***Figure 7: UML for test case generator***

Overall, the program works in the following manner:

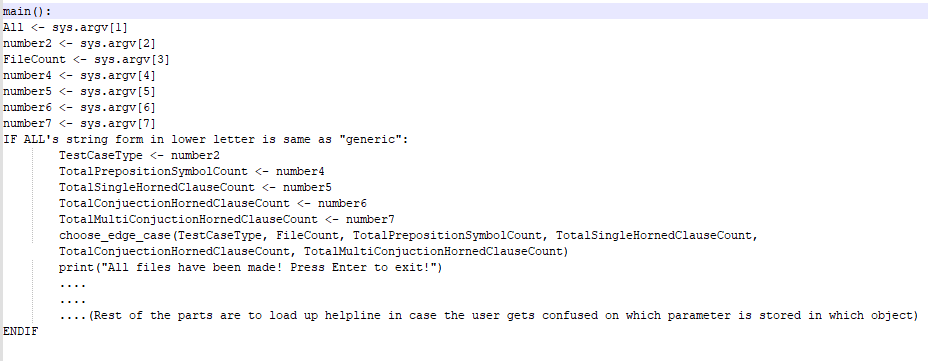
Step 1: It accepts inputs in the following format:

**<filename—ie .\InferenceEngineMapMaker.py> <TestCaseTypeNo> <TestCaseTotalFileCount> <TotalPrepositionSymbolCount> <TotalSingleHornedClauseCount> <TotalConjuectionHornedClauseCount> <TotalMultiConjuctionHornedClauseCount>**

In this format:

* 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

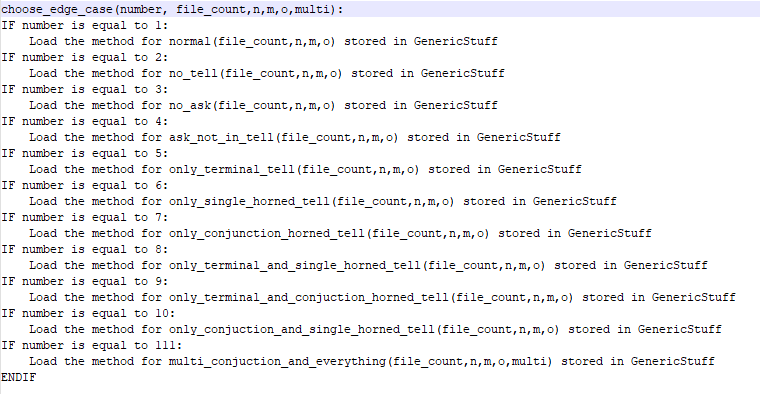
Step2: When it is inputted, it passed into main which assigns the parameters to their respective variables before passing it to choose\_edge\_case



***Figure 8: Main method in Main file***

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

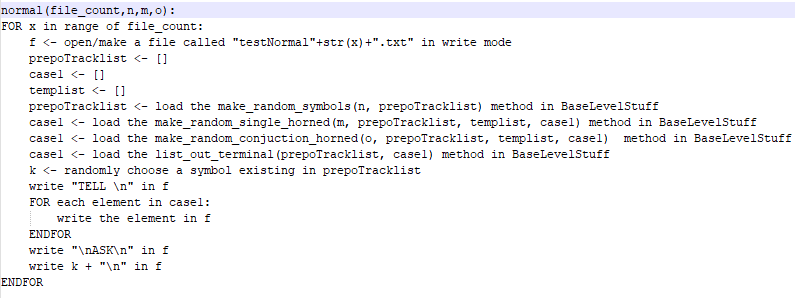
**Note:** for normal case or case1, it will call the GenericStuff.normal(file\_count,n,m,o) method located in chose\_edge\_case



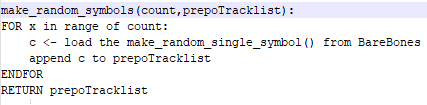
***Figure 9: choose\_edge\_case method in Main file***

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

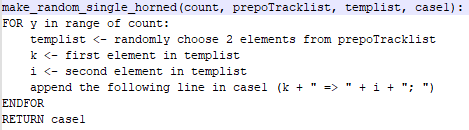
**Note:** for the normal case, it will create a file called testNormalN (where N is no of the file) and empty list to track preposition (prepoTrackList), empty list to track current KB (case1), empty list to temporarily hold prepositions or KB(tempList). Then it will populate prepoTrackList with random 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



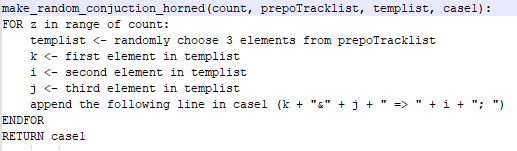
***Figure 10: normal method in Generic Stuff file***



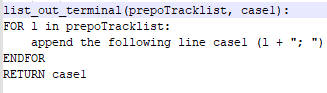
***Figure 11: make\_random\_symbol method in Base Level Stuff file***



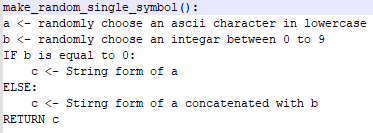
***Figure 12: make\_random\_single\_horned method in Base Level Stuff file***



***Figure 13: make\_random\_conjunction\_horned method in Base Level Stuff file***



***Figure 14: list\_out\_terminal method in Base Level Stuff file***



***Figure 15: make\_random\_single\_symbol method in Bare Bones 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)

**Note:** For bug testing, we had only tested for cases where there was 20 total numbers of prepositions with 5,10,15 and 20 cases of single horned, conjunction horned and multi horned cases (along with just normal prepositions for Truth Table testing). That’s because we were not able to get reliable algorithms online to verify our answers and thus had to check them by hand (and on excel sheet) to ensure they were correct. Thus, although the program had accurately solved for those small cases, it’s accuracy (or correctness) has not been tested for large cases (like 100 or 1000 prepositional symbols)

## Achknowledgements/Resources

++Mention all the resources++find and include DPLL youtube and dpll tutorial website link

++put int all the references here!

## Notes:

++Mention outcome of tests performed and their performance has been automatically and verbosely noted in the following files:…..

## Research:

**Research1:** Implementation of time counter (in nano seconds <https://docs.python.org/3/library/time.html#time.time_ns> …) and memory usage (in byte <https://psutil.readthedocs.io/en/latest/> ) for performance measure for forward and backward chaining algorithm

**Note:** Initially, the performance measure was planned to be tested on truth table, forward chaining and backward chaining algorithms. But for truth table it would take far too long (as for n number of prepositional symbols, time complexity will be take 2n (REF---slide on week 7)) and thus it had been decided to only compare between Forward chaining and Backward chaining instead.

According to information provided in the slides (REF) and the book (REF), forward chaining is generally slower than backward chaining. That’s because backward chaining will be starting from goal case (ie ASKed proposition being implied by the Knowledge Base in TELL) and will go up to the terminal nodes to prove or disprove it and thus, would look through less number of nodes and clauses compared to forward chaining!

But, they were not clear on how much difference would actually exist in time and memory usage for those algorithms. Hence an experiment has been set up where data (of 100 cases each) were collected for:

* 1000 total symbols and 10 horned and single clauses
* 1000 total symbols and 25 horned and single clauses
* 1000 total symbols and 50 horned and single clauses

**Note:** This had been done by generating those files using the Inference Engine Map generator Program where parameter of total symbol was set to 1000 and horned and single clauses were both set to be 10, 25 and 50 accordingly. These have been made by running the following command through command line:

**python .\InferenceEngineMapMaker.py generic 1 100 1000 <10 or 25 or 50> <10 or 25 or 50> 0**

Also, the following command line script has been used to run the files (whose results have been stored excel sheets):

* **Get-ChildItem testNormal\*.txt | foreach-object{.\iengine.exe fc $\_}**
* **Get-ChildItem testNormal\*.txt | foreach-object{.\iengine.exe bc $\_}**

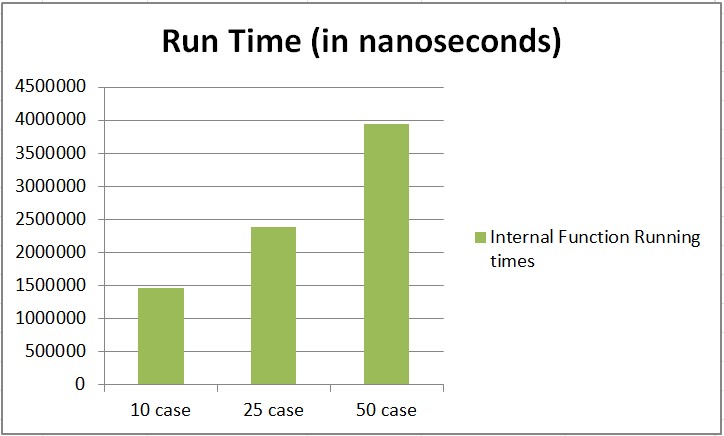
**Note:** Here the **Get-ChildItem** will call all the files that start with **testNormal** and end with **.txt** extension. Then the **foreach-object** will gather all those files and basically run the following command

**.\iengine.exe <method> <filename>.txt**(REF)

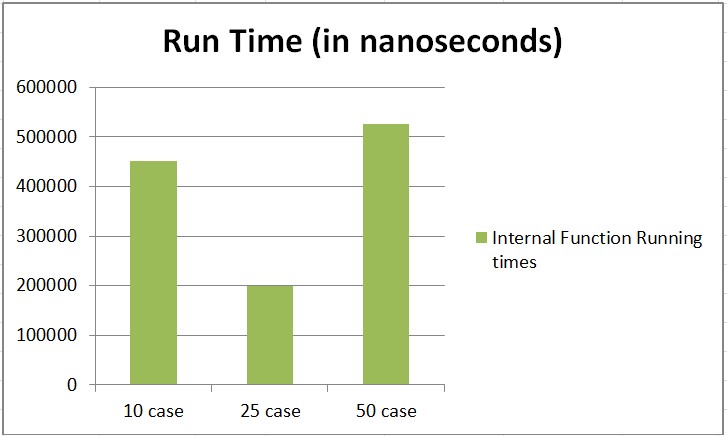
Microsoft, “Get-ChildItem (Microsoft.PowerShell.Management) - PowerShell,” *learn.microsoft.com*. <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-childitem?view=powershell-7.3> (accessed Apr. 13, 2023).

[17] Microsoft, “about Automatic Variables - PowerShell,” *learn.microsoft.com*. <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view=powershell-7.3> (accessed Apr. 13, 2023).

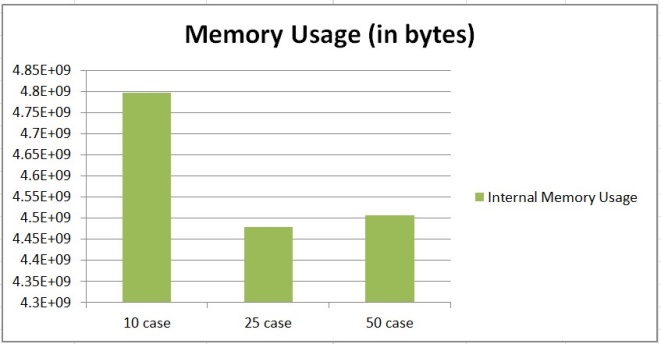
This experiment has resulted in the following outcome:

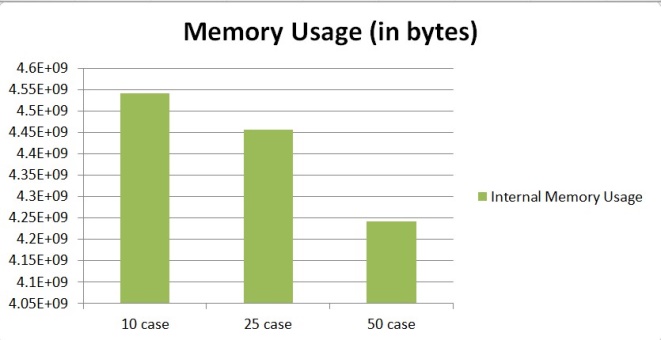
**

***Graph 1(A): Average Time Used by Forward chaining for those cases***

**

***Graph 1(B): Average Time Used by Backward chaining for those cases***

***Graph 2 (A): Average Memory Used by Forward chaining for those cases***

******

***Graph 2 (B): Average Memory Used by Backward chaining for those cases***

From the graphs, it can be seen that for forward chaining, as the number of horned clauses (both single and conjunction) increased, the the run time increased in an almost linear manner (Graph 1 (A)), whilst memory usage decreased, but in a fluctuating manner (Graph 2 (A)). Furthermore, for backward chaining, as the number of horned clauses increased, the run time fluctuated a lot (decreasing from 10 to 25, before increasing to new height for 50's case in Graph 1 (B)), whilst memory usage decreased in an almost linear manner (Graph 2 (B)).

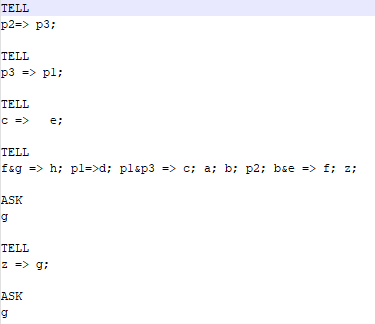
Moreover, when comparing forward and backward chaining, it can be seen that backward chaining always took less time (69%less for 10 case, 92% less for 25 case and 87% less for 50 case) but used more amount of memory (93%more for 10 case, 91% more for 25 case and 90% more for 50 case). Hence, if one has a large KB in horned form and needs a quick answer on a powerful machine (ie machine that can support large amount of memory) on whether a preposition in indeed entailed by KB, he or she should use backward chaining. But, if they want to perform it in a regular machine (ie machine that doesn't support large amount of memory) and doesn't need an immediate response, it would be better for them to use forward chaining instead.

**Note:** But one should keep in mind that the time noted here is in nano seconds and memory used noted in bytes. Thus, if someone is trying to find out whether the preposition is entailed by KB for small KBs (or even for KB with 1000 preposition), he or she can go for either algorithms and they will still get the answer almost instantly (in terms of human perception of 100 mili seconds or 100,000,000 Nano seconds (https://www.pubnub.com/blog/how-fast-is-realtime-human-perception-and-technology/ )) with relatively little amount of memory being used (when compared with current 8 GB RAM that can be found on average computers (<https://www.businessinsider.com/guides/tech/how-much-ram-do-i-need)>).

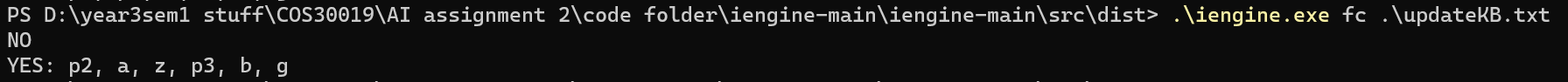
**Research2:** Implementation of updating of Knowledge base for Truth table, Forward chaining, and backward chaining

We have also implemented the feature of updating of KB for the algorithms. We have done this by modifying the ASK and TELL contexts (for both normal and horned clause cases) that we had created for the parser file. This feature allowed it to accept input files containing multiple TELL and ASK sections and parse them accordingly, allowing the stored KB to update itself.

The easiest way to see this in action is by passing a file containing multiple ASK and TELL (as seen in Figure 16).



***Figure 16: Text file for activating the updating of KB***

******

***Figure 17: outcome for that shows the updating of KB***

**Note:** The above image had been taken before the conversion to .exe format for Iengine, and thus the input method is slightly different from the standard one

In the figure 16, it can be seen that initially f and g implied h, but g was not noted as terminal node. Hence in forward chaining, it was not able to find it and thus stated that ASKed g was not implied in KB.

But in the second case, after the z implied g had been added, it was able to find it (as z was already noted as terminal nodes before) and thus realize that the ASKed g was indeed implied in KB.

**Research3:** Implementation of DPLL

DPLL (David-Putnam-Logemann-Loveland) is basically a backing-tracking based complete algorithm (as all prepositional logic can be converted to and from CNF REF <https://en.wikipedia.org/wiki/Conjunctive_normal_form> ) that decides whether or not the given propositional logic (analysed in CNF or Conjunctive Normal Form) is satisfiable (i.e the sentence of KB implying query is true in some models of KB (Lecture slide week 7)).https://en.wikipedia.org/wiki/DPLL\_algorithm. It does this in the following way:

At first, it chooses a literal, assigning it as True and simplifies the formula before recursively checking whether the simplified formula is satifiable. If it is, then original one is also satisfyable.

If it’s not, then the recursive check is performed again, but assuming the opposite truth value following splitting rule (ie splitting main problem into two simpler sub problem).

Moreover, in DPLL, in each step unit propagation and pure literal elimination is also performed. Unit propagation basically removes all clauses containing unit clause’s literal (ie clause that contains a single unassigned literal) and its complementary form and Pure literal elimination basically removes cases where pure literals occurs (ie clause that contains a literal that can always be assigned in such a way that it makes all clauses that contains it, true).

Hence, it is another algorithm (alongside TT,FC and BC) that can be used to ascertain whether or not the given query is implied by the KB in a more efficient way (when compared to naïve resolution method).

**Note:** Unfortunately, whilst initially coding for DPLL, our team had assumed that the input that will be provided to the DPLL algorithm will be in CNF form and thus had created the code with that in mind. Later on, in the final week, we had been informed that we would also need to create the algorithm that will convert the given TELL and ASK into the CNF form, before initiating the DPLL algorithm. Although this was a big issue (especially considering the deadline), we were still able to complete and implement the DPLL below in the following manner:

+Insert pseudocode for DPLL

***Figure 18: Pseudocode for DPLL***

## Team Summary Report: