**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

## Program architecture:

We built an agent that speaks a propositional logic-like language, whose grammar is given below, and is able to make inferences from its knowledge base. The agent, thus, has a parser to parse propositional formulas, an interpreter to map propositional formulas into its own knowledge representation language. The parser, performing syntactic analysis, is generated using ANTLR4. The interpreter is a visitor that walks the concrete syntax tree generated by the parser and generates the abstract syntax tree that represents the robot's internal interpretation.

Thus, we designed two things in support of our agent program:

- A propositional logic-like language
- Inference procedures for logical entailment

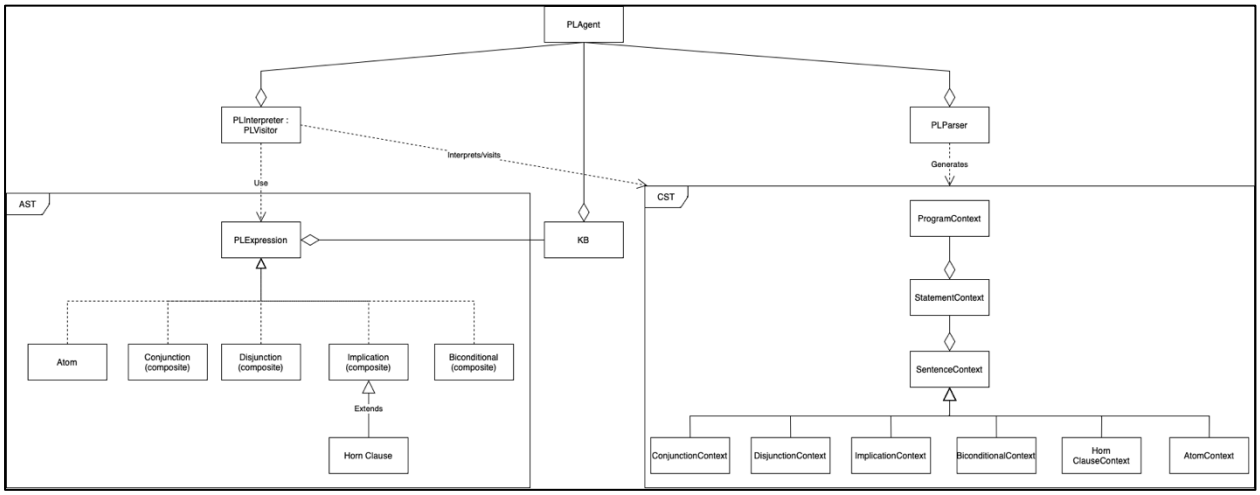



***Figure 1: Class design***

- All logical expression, except for Atom is a composite expression of other logical expressions.

**Note:** PLExpression is not needed, thanks to Python's duck typing. It is included only as illustration of "implicit" contract.

```
program: ( ( hornStatement | NEWLINE )* | ( statement | NEWLINE )* ) EOF;

statement: 'TELL' NEWLINE ( sentence ';' )* NEWLINE # Tell
         | 'ASK' NEWLINE sentence NEWLINE # Ask
         ;

hornStatement: 'TELL' NEWLINE ( hornClause ';' )* NEWLINE # HornTell
             | 'ASK' NEWLINE query=SYMBOL NEWLINE # HornAsk
             ;

hornClause: hornClauseHead IMPLICATION hornClauseTail # HornProper
          | atom=SYMBOL # HornSingle
          ;

hornClauseHead: SYMBOL ( CONJUNCTION SYMBOL )*;
hornClauseTail: SYMBOL;

sentence: '(' sentence ')' # Nested
        | NEGATION sentence # Negation
        | lhs=sentence CONJUNCTION rhs=sentence # Conjunction
        | lhs=sentence DISJUNCTION rhs=sentence # Disjunction
        | lhs=sentence IMPLICATION rhs=sentence # Implication
        | lhs=sentence BICONDITIONAL rhs=sentence # Biconditional
        | atom=SYMBOL # Atom
        ;

SYMBOL: 'True' | 'False' | [A-Za-z]+[0-9]*;
CONJUNCTION: '&';
DISJUNCTION: '||';
NEGATION: '~';
IMPLICATION: '=>';
BICONDITIONAL: '<=>';
NEWLINE: [\n];
WHITESPACE: [ \t\r]+ -> skip;
```

***Figure 2: Grammar (in src/grammar/PL.g4) of agent's language***

Data structures utilized to represent the robot's knowledge:

- **KB** (defined in **KB.py**) provides a representation of the robot's knowledge base. It can be evaluated, reduced to conjunctive normal form (CNF), etc. as if it is a conjunction expression. Its underlying container is a set.
- **Conjunction, Disjunction, Atom, Negation, Implication and BiconditionalExpression** are recursive data structures that represents their respective propositional formulas. They can evaluate themselves given a model (or partial model using cnfeval() if in CNF), reduce themselves of CNF.
- **HornClause** is an Implication Expression that can be constructed in "Horn form", by passing in a clause's head and tail. It's evaluation and CNF reduction inherits from the Implication Expression.
- All logic expressions are defined in **plast.py**.

The agent employs these methods to perform inference:

- Truth table enumeration: **tt_entail()**
- Forward chaining: **fc_entail()**
- Backward chaining: **bc_entail()**
- DPLL-based proof of unsatifiability: **dpll_entail()**
- All algorithms are defined in **algorithms.py**

Upon entry, a **PLAgent** object (the agent) is created, the file's path is passed in and the agent will perform syntactic analysis. By calling **agent.interpret()**, the agent will perform semantic analysis and appropriate inferences according to the statements in the file.

## User manual:

To run the program, call:

**iengine <method> <filename>**

- 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' – method name is case-insensitive) 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).

The file's format must conform to the given grammar. Some major points include (note: the test files on canvas should parse successfully):

- The grammar is case sensitive, whitespaces are ignored, except for new line characters.
- You can only 'TELL' or 'ASK' (case-sensitivity applies) the agent.
    - Each tell or ask statement ends with a new line:
    - 'TELL' and 'ASK' keywords must be delimited from following sentence(s) by **1** new line. If you receive message: "No viable input at ...", most likely your tell statement ends with EOF instead of \n
    - Sentences in the ASK's statement must end in ';'
    - Tell statement accepts only one sentence.
    - Ask statement cannot be empty. You cannot ask an empty question.
- You can put multiple ask and tell statements in a file. The agent will add sentences to its working knowledge base each time its encounter an ask statement, and execute inference on the fly, with the knowledge base it has at that point, each time it encounters a tell statement. The knowledge base can only be "reset" once the program stops.
    - **Bugs:** the agent has a know() function, which is called by the interpreter whenever an atom expression is created. This is utilized during the inference procedure, and contains query's symbols. Query's symbols should, ideally, be deleted upon completion of inference.
- Truth value literals are 'True' and 'False'; they are evaluated as-is. 'TRUE' and 'FALSE' are interpreted as propositional symbols.
- A program must be entirely, or not at all, in horn form. Tell statements that pass in horn form clauses must be followed by ask statements with horn formed queries. Otherwise, the file is interpreted as passing in a generic knowledge base and generic query. Backward and forward chaining, used in these cases, will exhibit undefined behavior.

# Implementation of inference algorithms:

## 1. Truth table enumeration

Implemented in algorithms.tt_entail, the function is a faithful implementation of AIMA's pseudocode. Given a knowledge base, query and symbols, the procedure constructs full models of all symbols, and evaluate the knowledge base and query in each model. Logical entailment is proven if in all models where kb evaluates to True, the query also evaluates to True. The inference process also keeps track of the number of models in which the knowledge base is satisfiable.

```
function tt_entail(kb,query,symbol):
Global model count <- 0
IF tt_check(kb, query, symbols, {}, model_count) is TRUE
       PRINT ("Yes" and model count)
ELSE
       PRINT("NO")
ENDIF
       function tt_check(kb,query,symbols,model,model_count):
              IF no symbols,
                     IF KB evaluates to True in model,
                            increase model count
                            RETURN evaluation of query in model
                     ELSE RETURN TRUE
                     ENDIF
              ELSE
                     symbols stack <- create a LIST of symbols for backtraking
purposes
                     p <- top element in the stack
                     rest <- remaining elements in the stack
                     tmp_model <- copy the model
                     model[p] <- set as TRUE
                     tmp_model[p] <- set as FALSE
                     RETURN tt_check(kb, query, rest, model, model_count) and
tt_check(kb, query, rest, tmp_model, model_count)
              ENDIF
```

***Figure 3: Pseudocode for tt entail***

The result of the inference process is:

a) If the agent's knowledge base entails query:

   **"YES : N"** where N is number of models wherein KB is satisfiable.

b) If the agent's knowledge base does not entail query:
   **"NO"**

## 2. Forward chaining

Implemented in algorithms.fc_entail, the function is a faithful implementation of AIMA's pseudocode. Given a knowledge base and query (the symbols are provided for uniform calling of all inference procedures), the procedure fires each clause in the knowledge base, if all premises of the

fired clause are inferred True, the consequent of the clause is inferred. This process is repeatedly carried out until the query is inferred.

```
function fc_entail(kb,query):
count <- table (hash map)
inferred <- table (hash map)
agenda <- queue
horn <- set
# Preprocess
FOR each clause in kb
      IF clause is an atom:
            inferred[clause] <- false
            add clause to agenda
      else:
            add clause to horn set
            count[clause] = count of clause's premises
            For each premise in head of clause
                  inferred[premise] <- false
            ENDFOR
            inferred[tail of clause] <- false
            ELSE
                  inferred[clause] <- false
                  add clause to agenda
      ENDIF
ENDFOR

print output of fc(horn, query, count, agenda, inferred)

function fc(kb,query,count,agenda,inferred):
            WHILE agenda is not empty,
                  p <- front of agenda
                  IF p is query
                        RETURN True
                  ENDIF
                  IF p is not inferred
                        inferred[p] <- TRUE
                        FOR each clause in kb
                              IF p is in head of the clause
                                    decrease count of clause by 1
                                    IF count of clause is 0
                                          add tail to the agenda
                                    ENDIF
                              ENDIF
                        ENDFOR
                  ENDIF
            ENDWHILE
```

***Figure 4: Pseudocode for Forward Chaining***

The output of forward chaining is: **"YES : a,b,c,d,…."** where a,b,c,d,… are the propositional symbols that had been inferred from KB during the execution of the algorithm, if the query can be entailed from the knowledge base. Otherwise: the result is "No".

**Note:** The ordering of the output symbols list is not deterministic. Running the procedure multiple times result in different set or ordering of the set of symbols. This is due to the knowledge base and horn set being an unordered set. Testing showed that using hash for efficient access speeds up the inference process by two times compared with using a list. This is a calculated decision that should have no bearing on the result of the inference procedure.

In fact, due to the knowledge base being unordered, the ordering of models enumerated in truth table enumeration checking is also not deterministic. This doesn't matter and is not noticeable at all since tt entail enumerates every model possible.

## 3. Backward chaining

```
function bc_entail(kb,query,symbol):
      inferred <- set
            IF query is in inferred RETURN True
            FOR each clause in kb
                  If clause is an atom and clause == query Return True
                  Else add clause to inferred
                  If clause.tail() == query
                        keep count of number of premises in clause
                        if for all premises, bc(kb, premise, symbols)is True
return True
                        if for one premise, bc(kb, premise, symbols) is True,
add premise to inferred
```

***Figure 5: Pseudocode for Backward Chaining***

Both backward and forward chaining maintain an inferred set as their "working memory". Backward chaining traverses the knowledge base. On encountering an atom, it is either the goal (query) or inferred. On encountering a clause, if the tail of the clause is a goal, the premises are subgoals. If all subgoals are inferred, the goal is inferred. Otherwise, move on to next subgoal. This subgoal inference process a recursive procedure. Our implementation uses a recursive function to carry out the process.

The procedure ouptuts **"YES : a,b,c,d,…."** where a,b,c,d,… are the propositional symbols inferred from KB during the execution of the algorithm, if the query is entailed in the knowledge base; otherwise, **"NO".**

**Note:** Like forward chaining, the ordering of the output is not deterministic, for similar reasons.

## 4. Generic knowledge base entailment

### 4.1. Inference on generic knowledge base

To allow for inference on generic knowledge base, the only problem is :

- Our grammar allows for nested, conjunction, disjunction, negation, biconditional, implication, atomic with appropriate order precedence:

+ Nest > Negation > Conjunction > Disjunction > Implication > Biconditional > Atom

(Horn clauses are unambiguous.)

- Our design uses different data structures for different logical expressions and consider horn clauses specialized implications expression. The expressions implemented are recursive in nature, they can be nested to form a tree of expressions.

- Each expression evaluates itself by performing model checking and Boolean logical deduction:

e.g., Model[a] and Model[b] = True and False = False

**Note:** Sound and complete inference on generic knowledge base can only be carried out using tt entail or DPLL. Inferring using forward and backing chaining on a generic knowledge base will result in undefined behavior. There is currently no checking mechanism for this, and the process of checking if a clause is in horn form is done during syntactic, rather than semantic, analysis.

### 4.2. CNF Conversion and DPLL

DPLL entail leverages DPLL satisfiability check algorithms to perform logical entailment using the deduction theorem. Given a knowledge base, query and list of symbols, it outputs the result of the entailment inference procedure: 'YES' or 'NO' and the flattened CNF of (KB & ~query) (a conjunction of disjunctions in set form).

To accommodate DPLL, CNF conversion is carried out according to the standard procedures, with workload distributed over the expression hierarchy. The procedure that performs CNF conversion is called **reduce()**:

- BiconditonalExpression reduces itself to a conjunction expression of two implication expressions by biconditional elimination.
- ImplicationExpression reduces itself to a disjunction expression of a negation and another expression by implication elimination.
- DisjunctionExpression reduces itself to a conjunction expression of disjunctions by distributivity.
- NegationExpression reduces itself to the inner expression of its inner expression by double negation, by double negation or a conjunction or disjunction of negations by De Morgan's law.
- ConjunctionExpression reduces itself to a conjunction expression of its left- and right-hand side's reduced form.
- AtomExpression is in CNF.
- KB reduces itself to chained conjunction of its clauses.

By recursively calling each other's reduce(), we obtain the CNF of a complex propositional formula.

To deduce logical entailment, dpll_entail, given a knowledge base, query and symbols, attempts to perform DPLL satifisfiability checking on (KB & ~query). If (KB.reduce() & ~query) is unsat, KB entails query by means of deduction theorem.

DPLL satisfiability checking is perform optimizations on backtracking:

- We call flatten() on the given sentence, which returns a set of sets of literals (atom and negation of atom). The naïve implementation we attempted in the past does not flatten the hierarchy, thus, whenever performing early exit checking, pure symbols or unit clause heuristics, we will walk the entire tree (expensive heuristics), leading to runtime worse than tt entail. By flattening the hierarchy, runtime is reduced by a factor of 100.
- Early exit by walking the 2-level hierarchy and calling each's logical expression data structure cnfeval() function. If one item in a conjunction (the first level) is False, exit as False, if one item in a conjunction (the second level) is True, exit (return to level one) as True.
- Pure symbols heuristics by checking for all symbol, if while walking the set, only its atom or the negation of its atom is found, assign a value that makes its occurrences' evaluation True.
- Unit clause heuristics by walking the set, if in one second-level set, one and only one literal is not contained in the model, assign a value that makes the disjunction True. The Negation and Atom Expression's unit is used to find the value and the symbol of the unit clause.

## Test Cases:

The InferenceEngineMapMaker program generates test cases automatically:

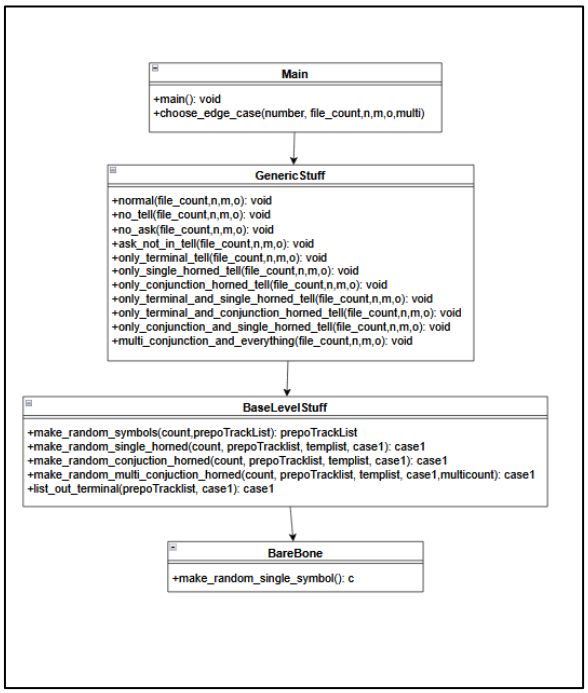



***Figure 7: UML for test case generator***

Overall, the program works in the following manner:

Step 1: It accepts inputs in the following format:

**<path to python interpreter> <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 (i.e. a &b => c format), single horned clause (i.e. a => b format), terminal nodes (i.e. 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**

```
main():
All <- sys.argv[1]
number2 <- sys.argv[2]
FileCount <- sys.argv[3]
number4 <- sys.argv[4]
number5 <- sys.argv[5]
number6 <- sys.argv[6]
number7 <- sys.argv[7]
IF ALL's string form in lower letter is same as "generic":
        TestCaseType <- number2
        TotalPrepositionSymbolCount <- number4
        TotalSingleHornedClauseCount <- number5
        TotalConjuectionHornedClauseCount <- number6
        TotalMultiConjuctionHornedClauseCount <- number7
        choose_edge_case(TestCaseType, FileCount, TotalPrepositionSymbolCount, TotalSingleHornedClauseCount,
        TotalConjuectionHornedClauseCount, TotalMultiConjuctionHornedClauseCount)
        print("All files have been made! Press Enter to exit!")
        ....
        ....
        ....(Rest of the parts are to load up helpline in case the user gets confused on which parameter is stored in which object)
ENDIF
```

***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**

```
choose_edge_case(number, file_count,n,m,o,multi):
IF number is equal to 1:
    Load the method for normal(file_count,n,m,o) stored in GenericStuff
IF number is equal to 2:
    Load the method for no_tell(file_count,n,m,o) stored in GenericStuff
IF number is equal to 3:
    Load the method for no_ask(file_count,n,m,o) stored in GenericStuff
IF number is equal to 4:
    Load the method for ask_not_in_tell(file_count,n,m,o) stored in GenericStuff
IF number is equal to 5:
    Load the method for only_terminal_tell(file_count,n,m,o) stored in GenericStuff
IF number is equal to 6:
    Load the method for only_single_horned_tell(file_count,n,m,o) stored in GenericStuff
IF number is equal to 7:
    Load the method for only_conjunction_horned_tell(file_count,n,m,o) stored in GenericStuff
IF number is equal to 8:
    Load the method for only_terminal_and_single_horned_tell(file_count,n,m,o) stored in GenericStuff
IF number is equal to 9:
    Load the method for only_terminal_and_conjuction_horned_tell(file_count,n,m,o) stored in GenericStuff
IF number is equal to 10:
    Load the method for only_conjuction_and_single_horned_tell(file_count,n,m,o) stored in GenericStuff
IF number is equal to 111:
    Load the method for multi_conjuction_and_everything(file_count,n,m,o,multi) stored in GenericStuff
ENDIF
```

***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

```
normal(file_count,n,m,o):
FOR x in range of file_count:
    f <- open/make a file called "testNormal"+str(x)+".txt" in write mode
    prepoTracklist <- []
    case1 <- []
    templist <- []
    prepoTracklist <- load the make_random_symbols(n, prepoTracklist) method in BaseLevelStuff
    case1 <- load the make_random_single_horned(m, prepoTracklist, templist, case1) method in BaseLevelStuff
    case1 <- load the make_random_conjuction_horned(o, prepoTracklist, templist, case1)  method in BaseLevelStuff
    case1 <- load the list_out_terminal(prepoTracklist, case1) method in BaseLevelStuff
    k <- randomly choose a symbol existing in prepoTracklist
    write "TELL \n" in f
    FOR each element in case1:
        write the element in f
    ENDFOR
    write "\nASK\n" in f
    write k + "\n" in f
ENDFOR
```

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

```
make_random_symbols(count,prepoTracklist):
FOR x in range of count:
    c <- load the make_random_single_symbol() from BareBones
    append c to prepoTracklist
ENDFOR
RETURN prepoTracklist
```

***Figure 11: make_random_symbol method in Base Level Stuff file***

```
make_random_single_horned(count, prepoTracklist, templist, casel):
FOR y in range of count:
    templist <- randomly choose 2 elements from prepoTracklist
    k <- first element in templist
    i <- second element in templist
    append the following line in casel (k + " => " + i + "; ")
ENDFOR
RETURN casel
```

***Figure 12: make_random_single_horned method in Base Level Stuff file***

```
make_random_conjuction_horned(count, prepoTracklist, templist, casel):
FOR z in range of count:
    templist <- randomly choose 3 elements from prepoTracklist
    k <- first element in templist
    i <- second element in templist
    j <- third element in templist
    append the following line in casel (k + "&" + j + " => " + i + "; ")
ENDFOR
RETURN casel
```

***Figure 13: make_random_conjunction_horned method in Base Level Stuff file***

```
list_out_terminal(prepoTracklist, casel):
FOR l in prepoTracklist:
    append the following line casel (l + "; ")
ENDFOR
RETURN casel
```

***Figure 14: list_out_terminal method in Base Level Stuff file***

```
make_random_single_symbol():
a <- randomly choose an ascii character in lowercase
b <- randomly choose an integar between 0 to 9
IF b is equal to 0:
    c <- String form of a
ELSE:
    c <- Stirng form of a concatenated with b
RETURN c
```

***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 ( i.e. 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)

**Note:** Whilst coding for BC, we had also noticed that it initially kept failing for cases where a single preposition was repeated multiple times. Thus, in order to correct that, we had created various test cases (using the test case generator) where the total preposition was kept very small (i.e. 5 and 10) and total number of single, conjunction and multi conjunction horned clauses were kept very large (i.e. 50 and 100). This ensured that the test cases generated will make horn clauses from a limited pool of prepositions (ensuring single preposition was repeated multiple times) and thus enabled us to ensure that BC was working as intended.

```
TELL
m3 => z6; z6 => p; m3 => q8; i4 => m3; z6 => p; t9 => d6; p => z6; z6 => t9; d6 => u9; t9 => f1; z6 => i4; d6 => f1; p =>
t9; u9 => p; u9 => d6; u9 => d6; i4 => u9; f1 => t9; u9 => j1; f1 => m3; d6 => u9; q8 => i4; m3 => i4; t9 => q8; m3 => d6;
f1 => j1; j1 => p; i4 => u9; f1 => m3; p => i4; m3 => j1; u9 => m3; p => m3; q8 => u9; t9 => p; z6 => t9; z6 => t9; f1 =>
q8; d6 => i4; d6 => t9; f1 => z6; f1 => i4; j1 => t9; p => f1; j1 => q8; z6 => m3; q8 => f1; m3 => i4; p => z6; u9 => j1;
z6&i4 => j1; t9&z6 => f1; j1&d6 => i4; z6&j1 => d6; t9&j1 => p; p&u9 => i4; q8&t9 => f1; m3&p => t9; z6&i4 => p; m3&q8 =>
f1; t9&i4 => j1; u9&d6 => i4; m3&q8 => u9; m3&q8 => t9; z6&j1 => t9; d6&u9 => z6; m3&t9 => j1; t9&q8 => j1; m3&t9 => d6;
i4&j1 => m3; m3&t9 => z6; z6&p => i4; q8&z6 => d6; d6&z6 => q8; t9&i4 => p; m3&f1 => p; t9&j1 => u9; p&i4 => t9; u9&q8 =>
j1; u9&d6 => i4; m3&q8 => j1; t9&d6 => m3; i4&q8 => z6; m3&t9 => z6; p&f1 => i4; i4&m3 => j1; m3&q8 => d6; u9&i4 => f1;
t9&q8 => m3; q8&f1 => j1; z6&j1 => d6; z6&j1 => u9; q8&f1 => z6; d6&q8 => i4; f1&p => d6; d6&p => q8; q8&f1 => m3; t9&z6 =>
f1; u9&q8 => z6; u9&d6 => z6; m3; f1; d6; z6; t9; p; u9; q8; j1; i4;
ASK
m3
```

***Figure 16: One of the test cases that had been generated and tested against BC***

According to information provided in the slides and the book, forward chaining is generally slower than backward chaining. That's because backward chaining will be starting from goal case (i.e. 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**

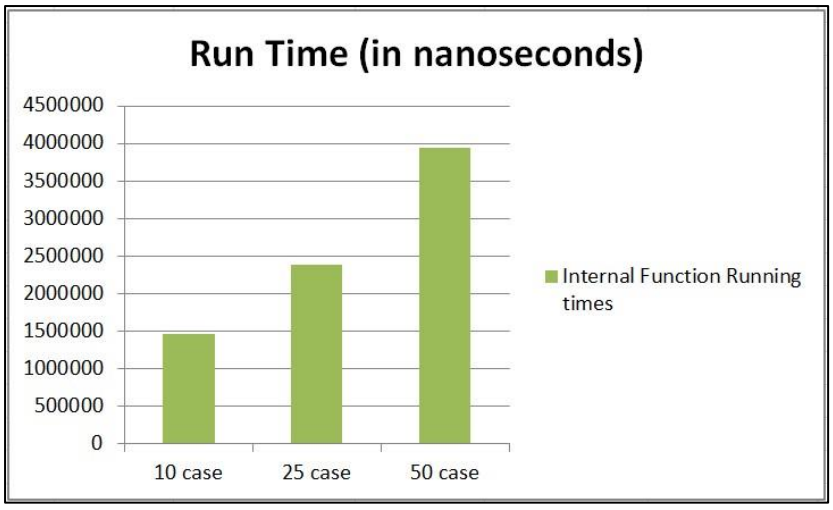
Also, the following command line script has been used to run the files (whose results have been stored excel sheets):

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

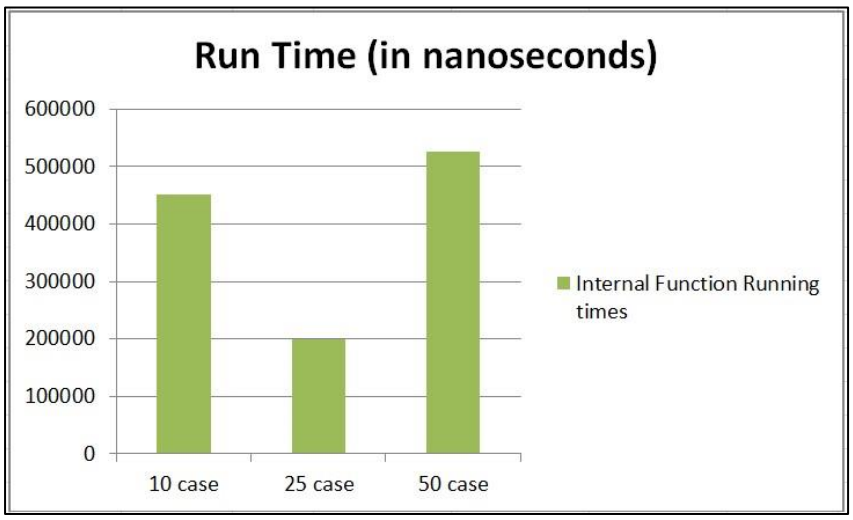
**Note:** Here the **Get-ChildItem** will call all the files that start with **testNormal** and end with **.txt** extension. Then the **foreach-object** will gather all those files and basically run the following command

**.\iengine.exe <method> <filename>.txt**

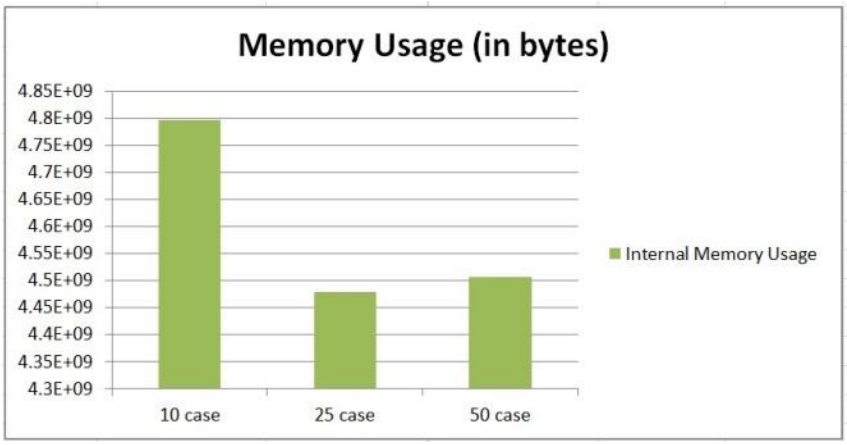
This experiment has resulted in the following outcome:



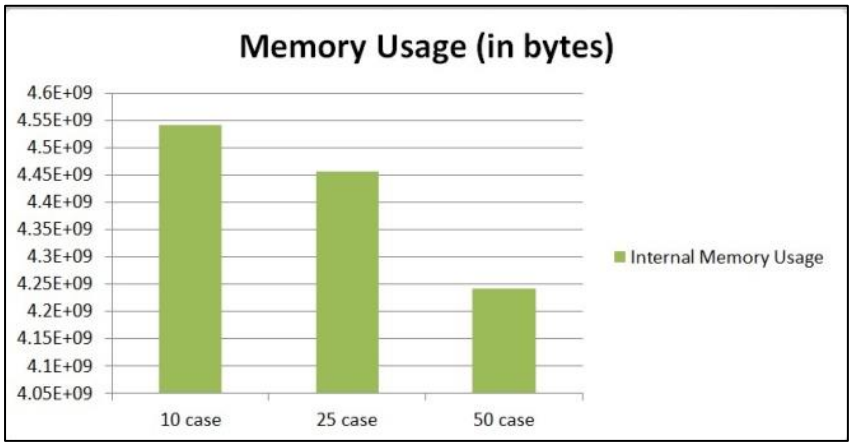

***Graph 1(A): Average Time Used by Forward chaining for those cases***




***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 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 (i.e. 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 (i.e. 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 proposition 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) with relatively little amount of memory being used (when compared with current 8 GB RAM that can be found on average computers).

## Acknowledgements

[AIMA] Chapter 7, Artificial Intelligence: A Modern Approach 3rd Edition, Stuart J. Russell, Peter Norvig, for the pseudocode and explanation for tt entail, forward chaining, backward chaining, DPLL.

[ArtInt] Artificial Intelligence: Foundations of Computational Agents, 2nd Edition, David L. Poole and Alan K. Mackworth, for the pseudocode and explanation of forward reasoning and backward reasoning.

ANTLR4 (Terrence Parr, contributors and StackOverflow's community) for a robust parser generation solution and the Definitive ANTLR4 Reference for the nice introduction to ANTLR4.

Brown University's CS1950Y unit's lab 4 SAT Solver tutorial sheet and the video by Prasenth Nattey on DPLL algorithm for Sat solving for inspiring us and helping us understand how to efficiently create the DPLL algorithm.

## Team Summary Report:

In this assignment, the work has been divided in the following manner:

S M Ragib Rezwan:

- Research and documentation of the algorithms and features used in program, consisting of:
    - Parser generator ANTLR4 and simplification on its functionality (including use of CST and AST for KB)

    - Truth table, Forward chaining, Backward chaining and DPLL algorithms
- Programming aspect:
    - Creation of initial DPLL (version that didn't have CNF resolution algorithm coded inside and thus could only work if input was in CNF form) algorithm
    - Creation of a program for test case generation for performance, testing and debugging of iengine program algorithms (including collection and analysis of data)
    - Creation of command line scripts for automated test case run
    - Modification of Iengine program's python codes (and libraries used in the code) to be windows compatible, alongside input detection (pre-parser)
    - Creation of .exe file using pyinstaller
- Report sections:
    - Wrote the initial version of all the sections of the report following the structure provided in the assignment 2 handout (but later they were updated into the new format by Linh)
    - Modification, proofreading and formatting of the final report and its images
    - In the current version of the report, Test case generation is fully written by me, (alongside partial parts and notes in the other sections in the current report)

Linh Vu:

- Design, implementation and optimization of agent program, consisting of:
    - Design and implementation of EBNF grammar (PL.g4) for parser generation using ANTLR4.
    - Design and implementation of abstract syntax tree data structures that accommodates generic knowledge base and CNF conversion.
    - Design and implementation of truth table enumeration, forward chaining, backward chaining .
    - Redesign and implementation of DPLL entailment, including CNF conversion, early termination, pure symbol and unit clause heuristics.
- Report sections:
    - Program architecture
    - User manual
    - Implementation details of tt entail, fc, bc, dpll