COS-30019
Introduction to Artificial Intelligence

Assignment 2
Inference Engine for Propositional Logic

Sehajpreet Singh
104211068

Pulkit Pannu
104093910

Table of Contents

Instructions

To run the Propositional Logic Inference Engine, follow these steps:

1. Open the command line interface (CLI) on a Windows 10 operating system
2. All the files are supposed to be in the same directory.
3. Navigate to the directory where the "main.py" file is located using the "cd" command followed by the path to the directory.
4. Once you are in the correct directory, use the following command format to run the program:

python main.py <filename> <method>

- Replace <filename> with the name of the text file containing the knowledge base and query. The file should be located in the same directory as the "main.py" file.
- The file can contain either a horn sentence or a general sentence.
- Replace <method> with one of the following inference methods:
    - TT: Truth Table checking
    - FC: Forward Chaining
    - BC: Backward Chaining

For example, to run the program with the "test1.txt" file using the Forward Chaining method, use the following command:

python main.py test1.txt FC

Introduction

The Propositional Logic Inference Engine is a software tool created to figure out if a question can be answered using a given set of information. It uses three different ways to find the answer: Truth Table Checking (TT), Forward Chaining (FC), and Backward Chaining (BC). Propositional logic is a type of logic that focuses on statements and how they are connected. In this report, we look into how these methods are used, talk about the ideas behind them, and share the outcomes of our tests.

Propositional Logic

Propositional logic, also called Boolean logic, is a branch of logic that deals with statements that can be either true or false [8]. It is important in various areas of computer science, such as formal verification, automated reasoning, and artificial intelligence [9]. The main components of propositional logic are:

1. Propositions: These are statements that can be true or false [8].
2. Logical Connectives: These are symbols that connect propositions to create more complex logical statements [10].

3. Some key logical connectives include:

    a. Negation (~): This changes the truth value of a proposition [8].

    b. Conjunction (&): This is true only if both propositions are true [8].

    c. Disjunction (||): This is true if at least one of the propositions is true [8].

    d. Implication (=>): This is true if the left-hand side (antecedent) implies the right-hand side (consequent) [10].

    e. Biconditional (<=>): This is true if both propositions have the same truth value [10].

For this assignment, we will focus on Horn-form knowledge bases, which are a subset of propositional logic. “Horn clauses are clauses in normal form that have one or more positive literals” [14]. Horn clauses are particularly important in computer science because they are simple and efficient for automated reasoning.

Inference Engine

The Inference Engine is a key part of intelligent systems that uses logical rules to draw new conclusions or make decisions based on a knowledge base [1]. In this project, we've implemented three different inference algorithms to show various ways of logical reasoning:

1. Truth Table Checking (TT):

Truth Table Checking involves evaluating all possible truth value combinations for the propositions in the knowledge base. By checking each model systematically, it determines if the query is true in every model where the knowledge base is true [2]. While this method is exhaustive and accurate, it can be computationally intensive for large knowledge bases [3].

2. Forward Chaining (FC):

Forward Chaining is a method that starts with known facts in the knowledge base and uses inference rules to generate new facts. This process continues until the query is derived or no more inferences can be made [4]. Forward Chaining is efficient for certain types of knowledge bases and is commonly used in expert systems [5].

3. Backward Chaining (BC):

Backward Chaining is a method that starts with the query and works backward, looking for rules that can lead to the query. It recursively checks the premises of these rules, trying to

establish each one as true through further backward chaining [6]. This method is useful when the goal is specific and the knowledge base is large, as it avoids exploring irrelevant parts of the knowledge base [7].

Efficiency

Efficiency is an important factor to consider when it comes to different methods of reasoning. The TT method, for example, is quite computationally expensive because it has to generate and evaluate all possible models. This becomes even more challenging as the number of propositional symbols increases, making it impractical for large knowledge bases. TT generates 2^n models where n is the number of symbols in our knowledge base. For example, if we have 10 symbols in our knowledge base then we get 1024 models.

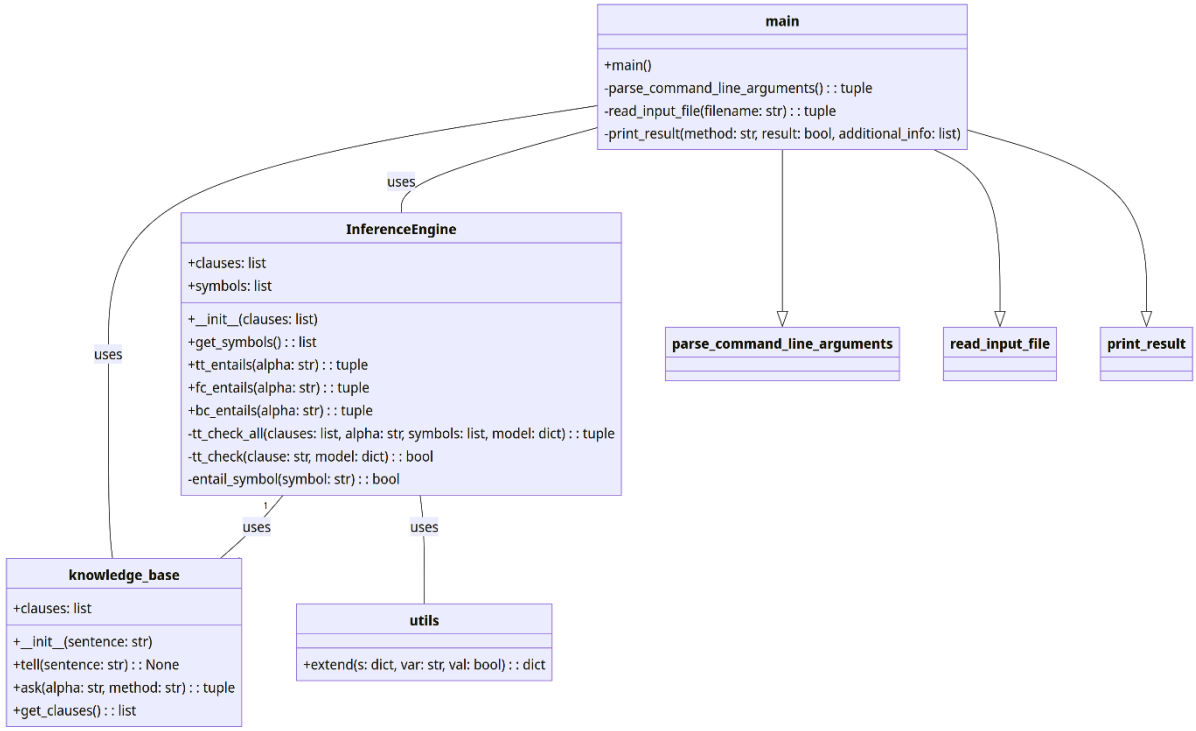
On the other hand, the FC method is generally considered more efficient than TT. It doesn't need to generate all possible models but instead incrementally derives new facts. It stops when the query is proven or when no more facts can be derived. However, one drawback is that it may end up deriving irrelevant facts during the inference process.

BC, or backward chaining, is often even more efficient than FC, especially when the query is closely related to the facts in the knowledge base. It focuses on the relevant subgoals and avoids deriving unnecessary facts. However, if the query is not well-structured, BC may explore irrelevant subgoals, which can be a drawback.

In summary, efficiency plays a crucial role in reasoning methods. While TT is computationally expensive, FC and BC offer more efficient alternatives, each with their own advantages and limitations.

The selection of the inference method relies on the unique features of the knowledge base, the query, and the desired balance between completeness and efficiency. Here are a few pointers: - When dealing with a small knowledge base and prioritizing completeness, the Truth Table method is a dependable option. - If the knowledge base consists mainly of facts and rules, and the query pertains to the initial facts, Forward Chaining is a suitable choice for efficient inference. - In cases where the knowledge base is well-structured and the query can be broken down into subgoals easily, Backward Chaining proves to be an effective and goal-oriented strategy.

Implementation



The inference engine is capable of utilizing three different inference methods: Truth Table (TT), Forward Chaining (FC), and Backward Chaining (BC). It requires a knowledge base consisting of propositional logic clauses and a query to ascertain if the query can be deduced from the knowledge base through the chosen inference method.

The Python implementation of the inference engine consists of five main components: utils.py, extra.py, iengine.py, knowledge_base.py, and main.py where utils and extra are helper files.

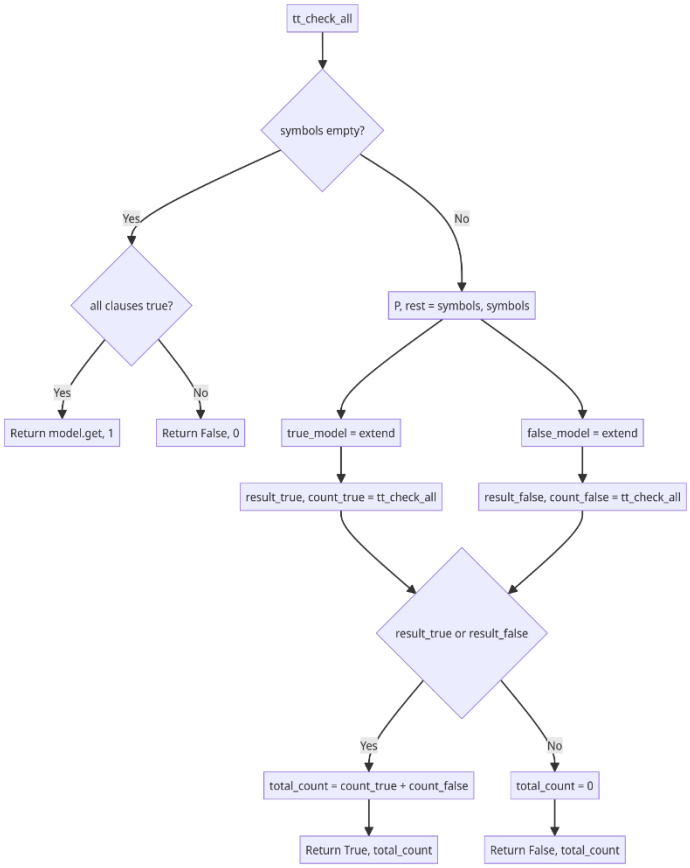
In terms of code structure, the utils.py file contains utility functions that are utilized by the inference engine. One of these functions is "extend," which allows for the extension of a model with a new symbol-value pair. On the other hand, the iengine.py file defines the InferenceEngine class, which encompasses the implementation of the three inference methods. Additionally, the knowledge_base.py file defines the knowledge_base class, which is responsible for storing and querying the knowledge base. Lastly, the main.py file serves as the main entry point of the program. It handles tasks such as parsing command-line

arguments, loading the knowledge base from a file, and executing the specified inference method.

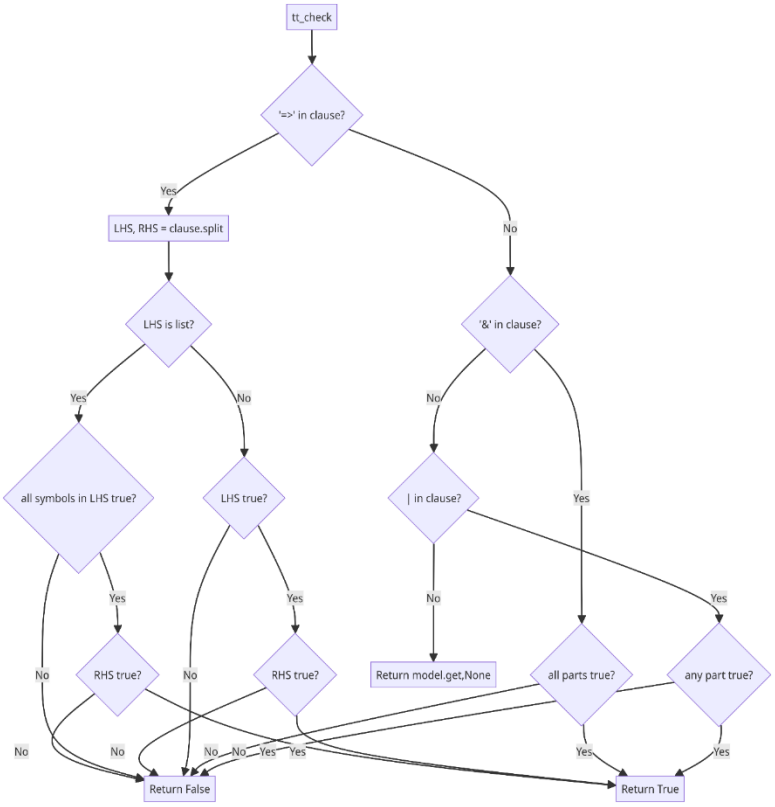
TRUTH TABLE

One of the implemented inference methods is the Truth Table (TT) method. This method is implemented within the tt_entails function of the InferenceEngine class. It relies on two helper functions: tt_check_all and tt_check.

The tt_check_all function recursively generates all possible models for the given symbols and clauses. It takes the clauses, the query alpha, the list of symbols, and the current model as arguments. The function returns a tuple that consists of a boolean value indicating whether alpha is entailed by the knowledge base and the count of models that satisfy the knowledge base.



The tt_check function evaluates a single clause under a given model. It handles different types of clauses (implications, conjunctions, disjunctions, and single literals) and returns the truth value of the clause under the model.



FORWARD CHAINING

The FC method is utilized in the fc_entails function within the InferenceEngine class.

The function initializes two empty lists: symbols_entailed and agenda. It creates an inner function called entail_symbol, which checks if a symbol can be entailed from the knowledge base. If the symbol is in symbols_entailed or if the query alpha is already in symbols_entailed, it returns True. Otherwise, it goes through the clauses in the knowledge base to see if the symbol is on the right-hand side of an implication clause. If it is, it recursively checks if all symbols on the left-hand side of the implication can be entailed. If they can, the symbol is added to symbols_entailed and agenda, and True is returned. If not, it returns False.

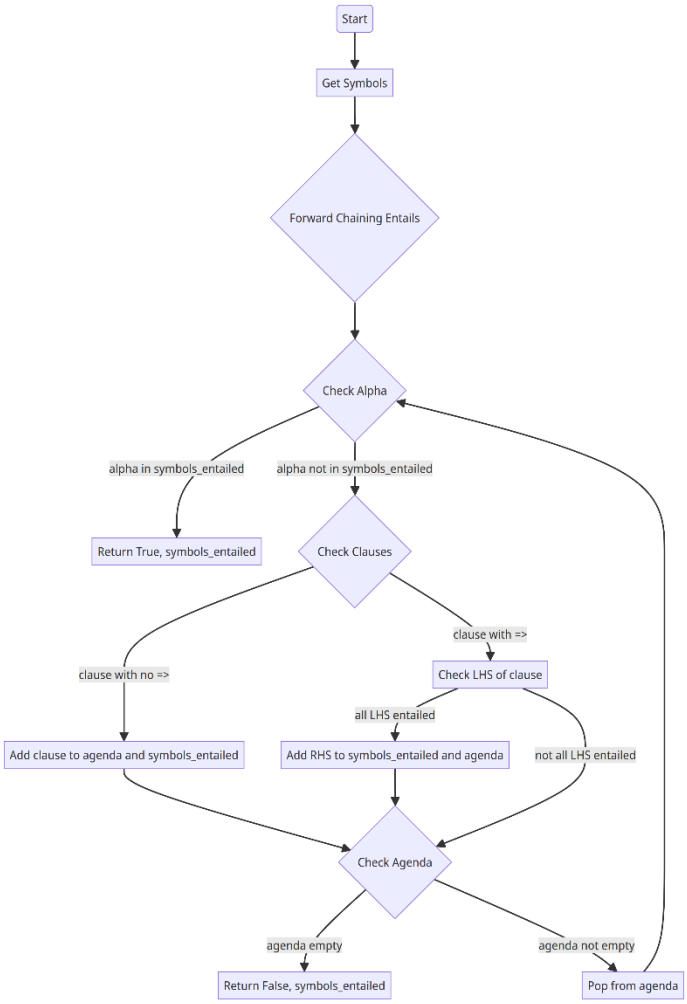
The algorithm starts by going through the clauses in the knowledge base. If a clause is a fact (no implication), it is added to symbols_entailed and agenda. If the fact is the same as the query alpha, True is immediately returned along with symbols_entailed.

After processing the facts, the algorithm enters a loop that continues until the agenda is empty. In each iteration, it removes the first symbol p from the agenda. If p is equal to the query alpha, the function immediately returns True along with symbols_entailed.

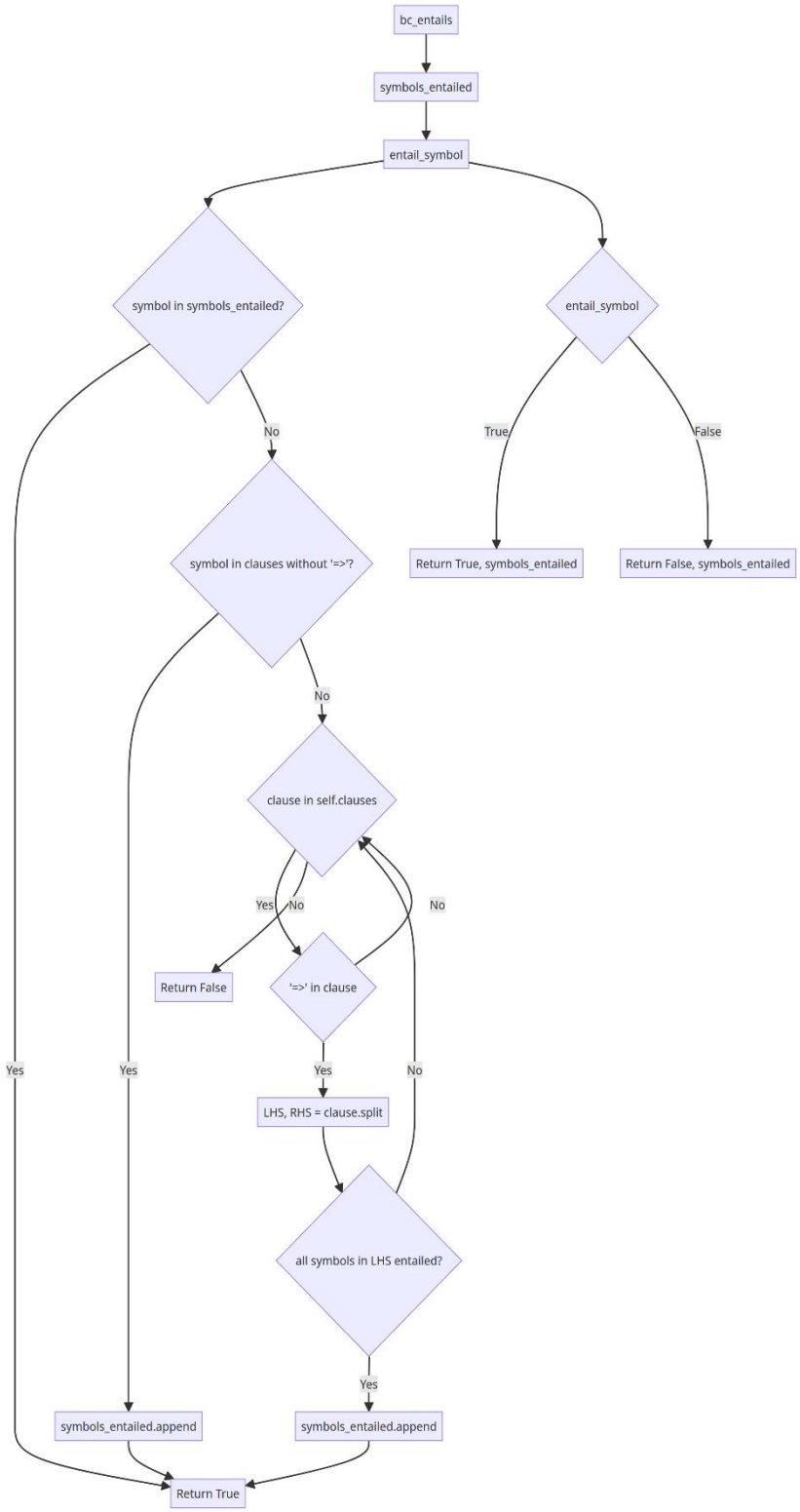
For each symbol p, the algorithm goes through the clauses in the knowledge base again. If a clause is an implication and all symbols on the left-hand side can be entailed, the RHS

symbol is checked. If the query alpha is already in symbols_entailed, the function returns True along with symbols_entailed. If the RHS symbol is not in symbols_entailed, it's added to both symbols_entailed and agenda. If the RHS symbol is the same as the query alpha, True is returned along with symbols_entailed.

If the loop finishes without finding the query alpha, the function returns a tuple indicating whether alpha is in symbols_entailed and the list of symbols_entailed.



BACKWARD CHAINING



The BC method is utilized in the bc_entails function within the InferenceEngine class. It employs a recursive entail_symbol function to verify if a specific symbol can be inferred from the knowledge base.

Initially, the entail_symbol function examines whether the symbol is already in symbols_entailed or if it is a fact in the knowledge base. If not, it proceeds to search for a clause where the symbol is present on the right-hand side of an implication and recursively verifies if all symbols on the left-hand side can be inferred.

Subsequently, the bc_entails function invokes entail_symbol with the query alpha and provides a tuple indicating whether alpha is inferred and the list of symbols_entailed throughout the process.

KB REPRESENTATIONS

The representation of the knowledge base is done through the utilization of the knowledge_base class. It stores the clauses in the form of a string list. To add new sentences to the knowledge base, the tell method is employed, which takes in a horn sentence and adds it to the list of clauses. On the other hand, the ask method is utilized to inquire about the knowledge base using a designated inference method. It generates an InferenceEngine object with the stored clauses and invokes the suitable entailment function based on the method used.

Testing

We created a total of 16 test cases where 8 of them are for horn clauses and 8 for generic sentences.

By far the best algorithm is BC then FC and then TT, based on the number of symbols entailed and time taken to return the result.

Below is a table that summarizes all the test case scenarios:

Test number	KB type	TT result	FC result	BC result	Test description
1	Horn	YES:1	YES: p, x, y, q, v, s, t, u, w	YES: x, t, u, p, q, v, w	General test where FC entails more symbols then BC
2	Horn	YES:1	YES: p1, p3, d, p2, p4, a, b, c, e	YES: p1, p2, a, b, p3, p4, c, d, e	FC and BC entail same symbols but in different order
3	Horn	YES:1	YES: a, c, e, g, i, k, m, b, f, h, d, l, n, j	YES: i, k, l, m, n, j	FC entails way more symbols then BC
4	Horn	NO	NO	NO	KB doesn’t entail alpha
5	Horn	YES:1	YES: a1, b1, c1, d1, e1, f1	YES: f1	Alpha is a known fact but FC entails unnecessary symbols.
6	Horn	YES:2	NO	NO	FC and BC cant entail alpha because the LHS of the sentence the entails alpha cant be completely entailed with the current KB
7	Horn	YES:6	NO	NO	FC and BC cant entail alpha because the LHS of the sentence the entails alpha cant be completely entailed with the current KB

8	Horn	YES:1	YES: a, c, e, g, i, b, d, f, h, j, k, l, m, n	YES: e, f, g, h, l, i, j, a, b, c, d, k, m, n	FC and BC entail same symbols but in different order
9	Generic	YES:30	YES: p, q, s, r, u, v	YES: s, v	Successful conversion from general sentence to horn form.
10	Generic	YES:4	NO	NO	Conversion to horn form successful. Not all the symbols in LHS for the sentence that entails alpha can be entailed
11	Generic	YES:2	YES: a, b, e, g, d	YES:b, a, d	Successful conversion from general sentence to horn form.
12	Generic	YES:4	YES: m, p, q, t, u, o, s	YES: p, q, s	Not all the general sentences could be converted to horn form because its CNF form had clauses with disjunctions of more than one positive literal. But this did not affect the output of the KB to entail alpha because the algorithms were able to entail alpha with the horn form of the other sentences.
13	Generic	NO	NO	NO	One of the general sentences could not be converted to horn form because it contained more than one positive literal and alpha could not be entailed on the basis of the rest of the sentences in KB.
14	Generic	YES:2	YES: a, b, e, h, i, c	YES: a, b, c	Successful conversion to horn form
15	Generic(cnf)	YES:1	YES: q, r, s, u, x, p, v, w, z, y	YES: x, u, s, q, r, p, v, w, z, y	Conversion from CNF to horn was successful. The number of symbols entailed by FC and BC are the same but in different order.
16	Generic(cnf)	YES:1	YES: a, b, e, c, d	YES: a, b, c, d	Successful conversion from cnf to horn form

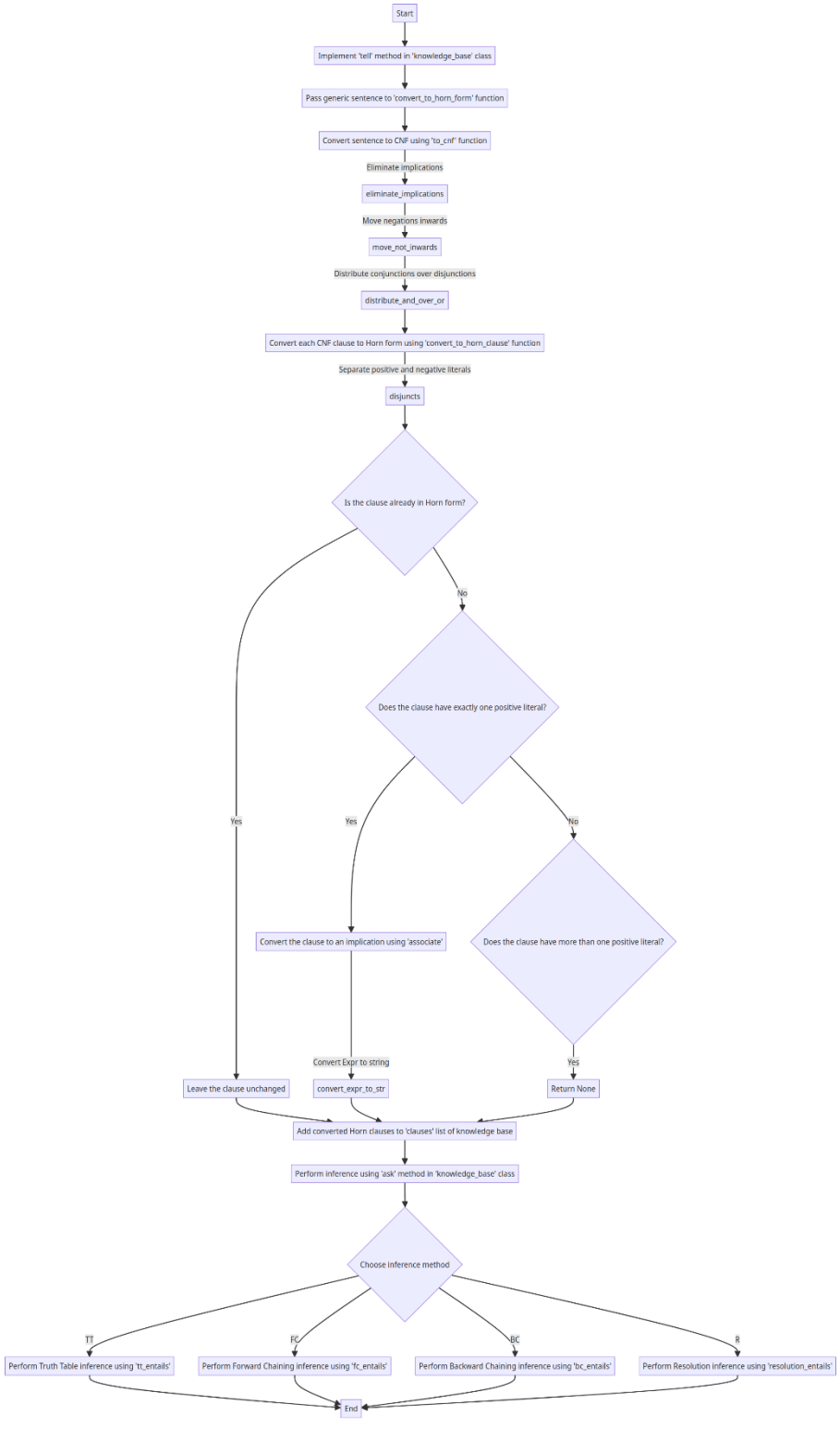
Research

To convert a generic knowledge base to Horn form, we used the following approach:

First, we implemented the tell method in the knowledge_base class. This method takes a generic sentence and sends it to the convert_to_horn_form function from the extra module for conversion [12].

Inside the convert_to_horn_form function, we first convert the sentence to Conjunctive Normal Form (CNF). To do this, we use the to_cnf function, which applies various techniques like eliminating implications, moving negations inwards, distributing conjunctions over disjunctions and flattening conjunctions and disjunctions [12]. We found the notes from the University of Iowa [13] really helpful in understanding and implementing the CNF conversion process.

1. After getting the CNF form, we convert each clause to Horn form using the convert_to_horn_clause function. In this function, we separate the positive and negative literals. Depending on the number of positive literals, we either leave the clause as it is (if it's already in Horn form), convert it to an implication (if there's exactly one positive literal), or return None (if there's more than one positive literal) [12]. The notes from Simon Fraser University [14] were a great resource for understanding the Horn clause conversion process.
2. After converting each sentence to horn form, the data type of the sentence is Expr (this data type is defined in utils.py). To change it from Expr to string, we call the method convert_expr_to_str(expr) which is defined in extra.py. This method returns the horn form as a string.
3. Finally, we add the converted Horn clauses to the clauses list of the knowledge base [12].


Start
Implement 'tell' method in 'knowledge_base' class
Pass generic sentence to 'convert_to_horn_form' function
Convert sentence to CNF using 'to_cnf' function
Eliminate implications
eliminate_implications
Move negations inwards
move_not_inwards
Distribute conjunctions over disjunctions
distribute_and_over_or
Convert each CNF clause to Horn form using 'convert_to_horn_clause' function
Separate positive and negative literals
disjuncts
Is the clause already in Horn form?
No
Does the clause have exactly one positive literal?
Yes
Yes
No
Convert the clause to an implication using 'associate'
Does the clause have more than one positive literal?
Convert Expr to string
Yes
Leave the clause unchanged
convert_expr_to_str
Return None
Add converted Horn clauses to 'clauses' list of knowledge base
Perform inference using 'ask' method in 'knowledge_base' class
Choose inference method
TT
FC
BC
R
Perform Truth Table inference using 'tt_entails'
Perform Forward Chaining inference using 'fc_entails'
Perform Backward Chaining inference using 'bc_entails'
Perform Resolution inference using 'resolution_entails'
End

Team Summary Report

Task Distribution:

1. Research and Planning:
    - We both researched propositional logic, inference methods, and the assignment requirements.
    - We met regularly to discuss our findings, ask questions, and plan how to implement the project.
2. Implementation:
    - Sehaj focused on coding the Truth Table (TT) and Forward Chaining (FC) methods.
    - Pulkit worked on coding the Backward Chaining (BC) method and handling input/output.
    - Pulkit worked on the code on converting a general sentence to conjunctive normal form and Sehaj improved on it so that it could convert a cnf sentence to horn form.
    - We often checked each other's code, gave feedback, and made improvements.
3. Testing and Debugging:
    - We both created test cases to cover different scenarios and check if our methods worked correctly.
    - We worked together on finding and fixing bugs in the code during testing.
    - We took help of chatgpt for debugging purposes and creation of test cases[15].
4. Documentation:
    - Sehaj mainly wrote the introduction, inference methods, and implementation parts of the report.
    - Pulkit focused on writing the testing, features/bugs/missing, and conclusion parts.
    - We proofread and edited each other's writing to make sure the report was consistent and easy to understand.

Feedback and Collaboration: Throughout the assignment, we maintained open communication channels, regularly discussing our progress, challenges, and ideas. We provided constructive feedback to each other, helping improve the quality of our code and report. We also adapted to each other's working styles and schedules, ensuring a smooth and efficient collaboration.

Conclusion

The Propositional Logic Inference Engine shows how various inference methods work. The TT method is thorough but not as fast with big knowledge bases as the FC and BC methods, which are better for Horn-form KBs. To make it better, we could optimize the algorithms more and add support for other logical connectives and more complicated knowledge bases in the future.

Acknowledgements/Resources

Our tutor Hy Nyugen's classes significantly helped us in achieving our goal. Hy was quick to reply when we had doubts and always provided us with great insight about certain topics.

We used the book "in Artificial Intelligence: A Modern Approach, 3rd ed" to take reference and understand the core concepts of this assignment

https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf

We used the following code library for which the link was provided on canvas to take reference and use a couple of methods.

https://github.com/aimacode/aima-python

Bugs/missing/notes

Bugs

- Our Algorithm doesn't take into account where the query to be entailed is a sentence and not just a propositional symbol or a known fact.
- Our algorithm does not take into account knowledge base sentences which can lead to infinite loop. If encountered with such a KB, our algorithm will produce errors. We didn't have enough time to implement a safety net for this situation.
- According to the project brief the symbol used for disjunctions is "||" whereas in the implementation of our algorithms, we mistook it for "|" and it was too late when we realised this bug so instead of changing our algorithm we hardcoded this implementation in out tell method of our knowledge base class where we simply replace "||" with "|" for each sentence.

Missing

- Although the conversion from a generic sentence to cnf works perfectly to the best of out knowledge, the pl resolution algorithm is not implemented as our version of the algorithm had errors, hence we removed it from the final iteration of our project.
- We couldn't implement the time module into our implementation because of the time constraints.

Notes

- The test_genericKB.txt and test_genericKB_1.txt provided with the project brief cannot be converted to horn form because the sentence "(a <=> (c => ~d)) & b & (b => a)" when converted to cnf is of the form "(d | a) & (c | a) & (~d | ~c | ~a) & b & (a | ~b)". Here "(d | a) & (c | a)" consists of more than one positive literal and hence cannot be converted to horn form. This is because the conversion from a clause in cnf with atmost one positive literal to a horn clause is done by using the implication property[14]:
    - ~P v Q ----> P => Q[14]

  Therefore if a clause consists more than one positive literal, we cannot use this property to convert it into a horn form

References

[1] S. Russell and P. Norvig, "Logical Agents," in Artificial Intelligence: A Modern Approach, 3rd ed., Upper Saddle River, NJ, USA: Prentice Hall, 2010, ch. 7, pp. 235-275.

[2] "Truth Table," Wikipedia, Mar. 25, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Truth_table

[3] "Inference Engine," Techopedia, [Online]. Available: https://www.techopedia.com/definition/3199/inference-engine

[4] "Forward Chaining," Geeks for Geeks, Mar. 16, 2023. [Online]. Available: https://www.geeksforgeeks.org/forward-chaining-in-artificial-intelligence/

[5] "Expert System," Wikipedia, Apr. 13, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Expert_system

[6] "Backward Chaining," JavaTpoint, 2023. [Online]. Available: https://www.javatpoint.com/backward-chaining-in-ai

[7] "Backward Chaining," Artificial Intelligence: Foundations of Computational Agents, 2nd ed., Cambridge University Press, 2017. [Online]. Available: https://artint.info/html/ArtInt_267.html

[8] S. Russel and P. Norvig, "Propositional Logic," in Artificial Intelligence: A Modern Approach, 3rd ed., Upper Saddle River, NJ, USA: Prentice Hall, 2010, ch. 7, sec. 7.4, pp. 204-210.

[9] "Propositional Logic," Stanford Encyclopedia of Philosophy, Mar. 19, 2018. [Online]. Available: https://plato.stanford.edu/entries/logic-propositional/

[10] "Propositional Logic," OpenTextBC, Apr. 6, 2021. [Online]. Available: https://opentextbc.ca/mathematicalreasoning/chapter/propositional-logic/

[11] "Horn Clause," Wikipedia, Apr. 3, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Horn_clause

[12] N. Nandanwar and P. Patel, "Propositional Logic Inference Engine," Swinburne University of Technology, Melbourne, VIC, Australia, May 2024.

[13] C. Tinelli, "CNF Conversion," Department of Computer Science, The University of Iowa, Iowa City, IA, USA, Fall 2010. [Online]. Available: https://homepage.cs.uiowa.edu/~tinelli/classes/188/Fall10/notes/cnf-conversion.pdf

[14] D. Mitchell, "Logic in Clause Form and Horn Clauses," School of Computing Science, Simon Fraser University, Burnaby, BC, Canada. [Online]. Available: https://www2.cs.sfu.ca/CourseCentral/383/dma/notes/LS_CF_HC.pdf

[15] Sam Altman, “ChatGPT”, OpenAI, https://chatgpt.com/?oai-dm=1