COS30019 ASSIGNMENT 2

Nadelin Nop- 103487208
Mohammad Tousif Shariar - 103517862

GROUP: COS30019_A02_T058

Features and Test Cases

Parser class implementation

Overall the purpose of this class is to take the input knowledge and split them according to clauses and facts. Each clause will be a horn object and it will be stored in a list for usage throughout the program, the same applies for facts instead it will be stored as a string. Furthermore the required query will be stored in this class too. Various voids were created to access all the parsed information from the knowledge base.

- read_line(Scanner file) – Reads all the lines from the text file, the second line which is the knowledge base will get their whitespace characters removed first. The knowledge base will be split by the delimiter ';' and it will be stored in a string array named 'split'. All strings in the 'split' will be looped through, if the string contains '=>' it will be added to the clauses list as a horn object. If not the string will be added to the facts list as a string.
- returnClauses() – It will return the clauses list
- returnFacts() – It will return the facts list
- returnQuery() – It will return the query which was read from the text file

Horn class implementation

Every clause will be a horn object and the purpose of this class is mainly to split a clause where the literals of the clause will be stored in a list and the entailment will be stored as a string. Various voids were created to access key information and update the list of literals of a clause.

- horn(String clause) – The constructor will take the clause and split it by '=>' and store the strings in the 'splitArrow' array. First element of the 'splitArrow' will be split by the delimiter '&' and the literals will be stored in the 'splitAnd' array. All the elements in the splitAnd list will be added to the firstList and the implication will be defined as a string
- getfirstList() – returns the 'getfirstList' list
- getListArrow() – returns the implication element
- updatefirstList(String c) – removes a specific element from the 'firstList'
- firstListCount() – returns the size of the 'firstList'
- returnfirstListIndex(int index) – returns a literal at specified index

Shunting Yard class implementation

The Shunting Yard algorithm was implemented as a class, which allows for the encapsulation of an algorithm so that the code within can be organised and modularized better. The whole point of implementing this algorithm is to convert the expression, which is an infix format, to a postfix expression, grouping all the operands together and operators together based on the value of its precedence as well. By converting it to a postfix notation it allows the truth table

checking algorithm to handle both general knowledge bases and horn form knowledge bases.[3]

- The method infixToPostfix() – takes an infix expression, which is a string expression given by the parser class, and converts it to postfix by iterating over each character, determining its type and handling it accordingly. It uses a StringBuilder to construct the postfix expression and a Deque to temporarily store operators. During the iteration, operands are appended directly to the postfix string, while operators are pushed onto the operator stack based on their precedence. When a closing parenthesis is encountered, operators are popped from the stack and appended to the postfix string until the corresponding opening parenthesis is reached. Finally, any remaining operators in the stack are appended to the postfix string. The resulting postfix expression is then returned after trimming any whitespace and removing semicolons.
- getPrecedence(String operator) – essentially just determines the value of precedence of each operator.

Truth Table Checking class implementation

The TruthTableChecking class is implemented to evaluate and check the truth table of a logical expression in postfix notation. Its purpose is to determine the truth values of the expression for all possible combinations of variable assignments and then evaluate a given query against the truth table. It is able to handle knowledge bases in both horn form and generic form.

- evaluate(String query) – is a method used to evaluate a query against the truth table for the expression. It iterates over each row of the truth table. For each row, it creates a HashMap called model to store the variable assignments for that particular row. It assigns the variable values by using bitwise operations on the row index, essentially just assigning true false values to all the characters. Next, it evaluates the postfix expression using the evaluateExpression method, passing the expression and the model as parameters. The result of the expression evaluation is stored in the result variable. After that, it evaluates the query in the same way, using the evaluateExpression method with the query and the model as arguments. The result of the query evaluation is stored in the queryResult variable.

  If both the query and the expression are true (queryResult and result are both true), it increments the trueCount variable. After evaluating all rows, it checks the value of trueCount. If it is greater than 0, it means there is at least one row where the query and the expression are both true.

- evaluateExpression(String postfixExpression, HashMap<String, Boolean> model) – is responsible for evaluating a postfix expression based on a given model of variable assignments. It starts by creating an empty stack of booleans to hold the intermediate results of the expression evaluation. The postfix expression is split into tokens using whitespace as a delimiter. If the token is not an operator, it retrieves the corresponding

boolean value from the model based on the token (variable name) and pushes it onto the stack.

If the token is an operator, a series of operations are performed based on the specific operator. The switch statement handles each operator case. After evaluating the operator, the resulting boolean value (result) is pushed back onto the stack. Once all tokens have been processed, the method checks the final value on the stack. If it is false, it means the expression evaluates to false. In this case, it returns false immediately. If the final value on the stack is true, it means the expression evaluates to true. In this case, it returns true.

Truth Table Checking Tests

Test Case	**Program Output**	**Expected Output**
TELL c=>q;l&b=>c;b&l=>m;a&p=>l;a&b=>l;a;b; ASK q	YES:2	YES:2
TELL p1=> p2; p2=> p3; p3=> p4; p4=> p5; p5=> p7; p9=> p14; p1; ASK p14	YES:2	YES:2
TELL p => q; l&m => p; a&b&l => m; a&p => l; a&b => l; a; b; ASK q	YES:1	YES:1
TELL p1 => p2; p2 => p3; p3 => p4; p1; p3; ASK p2	YES:1	YES:1
TELL c=>q;l&b=>c;b&l=>m;a&p=>l;a&b=>l;a;b;q; ASK	YES:2	YES:2

m		
TELL p1 => p2; p2 => p3; p3 => p4; p4 => p5; p5 => p7; p9 => p14; p1; p5; ASK p5	YES:3	YES:3
TELL p => q; l&m => p; a&b&l => m; a&p => l; a&b => l; a; b; p; ASK q	YES:1	YES:1
TELL p1 => p2; p2 => p3; p3 => p4; p1; p3; ASK p4	YES:1	YES:1
TELL p => q; l&m => p; a&b&l => m; a&p => l; a&b => l; a; b; p; ASK f	NO	NO
TELL p => q; l&m => p; a&b => m; a&p => l; a&q => l; a; b; p; ASK m	YES: 1	YES: 1
TELL p1 => p2; p2 => p3; p3 => p4; p4 => p5; p5 => p7; p8 => p6; p9 => p14; p1; p8 ASK p6	YES:3	YES:3
TELL c&p => q; l&m => p; a&b&l => m; a&p => l; a; b; p; c; ASK q	YES:1	YES:1
TELL p1 => p2; p2 => p3; p3 => p4; p5&p6 => p7; p1; p3; p5; ASK p7	YES:2	YES:2

TELL c&q=>f;l&b=>c;b&l=>m;a&p=>l;a;b ;f; ASK f	YES:12	YES:12
TELL p1 => p2; p2 => p3; p3 => p4; p4 => p5; p5 => p7; p9&p12 => p14; p1; p5; p12; ASK p9	YES:1	YES:1

Forward Chaining

One of the algorithms we have implemented in our program is the forward chaining algorithm. Basically it is an inference engine that is a part of the expert system that implements logical rules on the knowledge base for the purpose of finding out new data.[1] So the process usually starts with initial facts, rules will be identified that have conditions and then each rule will be evaluated. If the rule's conditions are met, actions will be executed according to the rule and new facts will be derived. All these steps will be repeated until the specific goal has been reached.

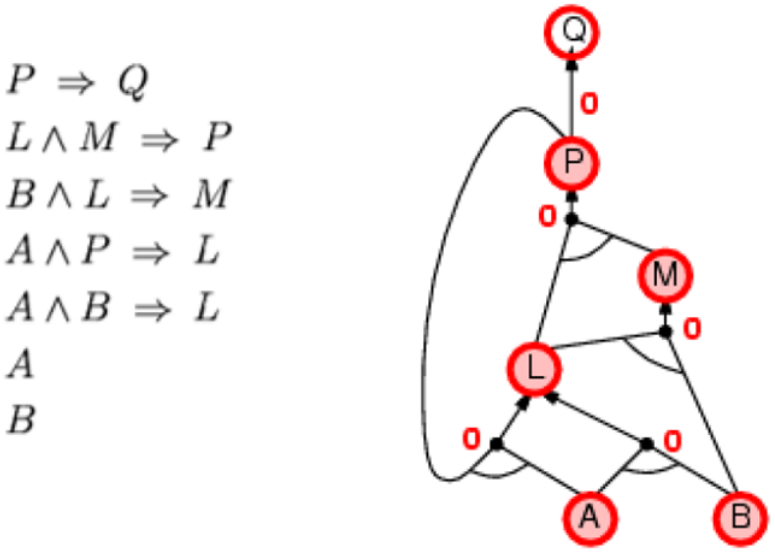


Figure 1: Illustration of forward chaining

Below is a detailed explanation of how this algorithm was implemented in our program:

- checkFacts() – Firstly a fact will be removed from the facts list and it will be added to another list. Then it will be checked if that fact is equal to the query or not. If not all the clause will be looped through to see if it matches with a fact or not. If there is a match, that element will be removed from the clause. Clauses will be looped through again to see whether the length of a clause is 0 or not, if it is zero hence the entailment is true and that clause will be removed from the clause list.
- test() – If the checkFacts void returns true, all the unique facts identified will be displayed, else query was not proven.

Forward chaining tests

Test case	**Program output**	**Expected output**
TELL c=>q;l&b=>c;b&l=>m;a&p=>l ;a&b=>l;a;b; ASK q	YES: a, b, l, c, q	YES: a, b, l, c, q

TELL
p1=> p2; p2=> p3; p3=> p4;
p4=> p5; p5=> p7; p9=> p14;
p1;
ASK
p14
	NO: QUERY cannot be proven	NO: QUERY cannot be proven

TELL p => q; l&m => p; a&b&l => m; a&p => l; a&b => l; a; b; ASK q	YES: a, b, l, m, p, q	YES: a, b, l, m, p, q
TELL p1 => p2; p2 => p3; p3 => p4; p1; p3; ASK p2	YES: p1, p3, p2	YES: p1, p3, p2
TELL	YES: a, b, q, l, c, m	YES: a, b, q, l, c, m

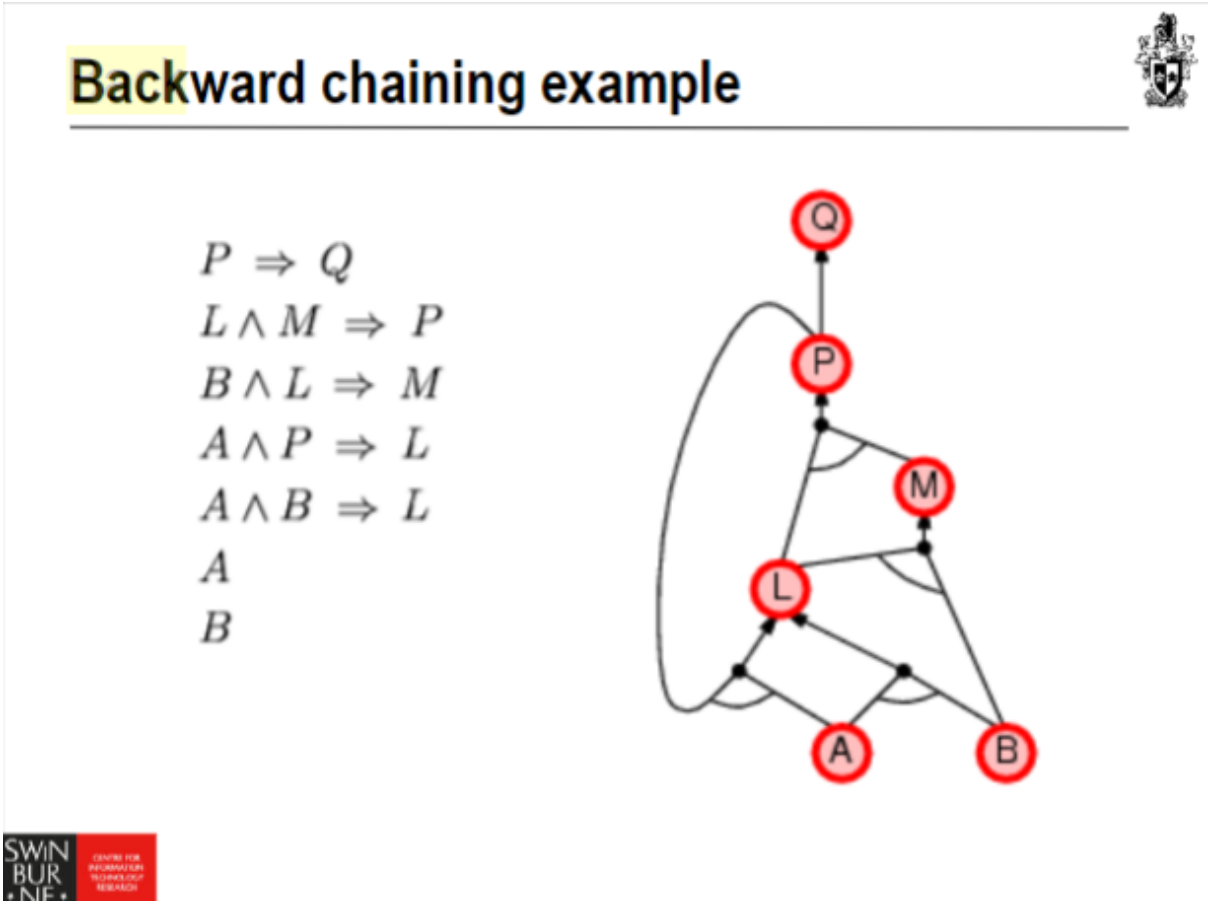
c=>q;l&b=>c;b&l=>m;a&p=>l;a&b=>l;a;b;q; ASK m		
TELL p1 => p2; p2 => p3; p3 => p4; p4 => p5; p5 => p7; p9 => p14; p1; p5; ASK p5	YES: p1, p5	YES: p1, p5
TELL p => q; l&m => p; a&b&l => m; a&p => l; a&b => l; a; b; p; ASK q	YES: a, b, p, l, q	YES: a, b, p, l, q
TELL p1 => p2; p2 => p3; p3 => p4; p1; p3; ASK p4	YES: p1, p3, p2, p4	YES: p1, p3, p2, p4
TELL p => q; l&m => p; a&b&l => m; a&p => l; a&b => l; a; b; p; ASK f	NO: QUERY cannot be proven	NO: QUERY cannot be proven
TELL p => q; l&m => p; a&b => m; a&p => l; a&q => l; a; b; p; ASK m	YES: a, b, p, m	YES: a, b, p, m

TELL p1 => p2; p2 => p3; p3 => p4; p4 => p5; p5 => p7; p8 => p6; p9 => p14; p1; p8 ASK p6	YES: p1, p8, p2, p6	YES: p1, p8, p2, p6
TELL c&p => q; l&m => p; a&b&l => m; a&p => l; a; b; p; c; ASK q	YES: a, b, p, c, l, q	YES: a, b, p, c, l, q
TELL p1 => p2; p2 => p3; p3 => p4; p5&p6 => p7; p1; p3; p5; ASK p7	NO: QUERY cannot be proven	NO: QUERY cannot be proven
TELL c&q=>f;l&b=>c;b&l=>m;a&p=>l;a;b;f; ASK f	YES: a, b, f	YES: a, b, f
TELL p1 => p2; p2 => p3; p3 => p4; p4 => p5; p5 => p7; p9&p12 => p14; p1; p5; p12; ASK p9	NO: QUERY cannot be proven	NO: QUERY cannot be proven

Backward Chaining

Backward chaining is an inference algorithm used in expert systems to determine the conditions or facts necessary to satisfy a given goal or query. It works by starting with the goal and recursively working backward to find the conditions that lead to the goal. The

algorithm searches for rules or statements that have the goal as a conclusion and examines the sub-goals or conditions required to satisfy the conclusion. It continues to apply the algorithm to each sub-goal, searching for rules that satisfy them. The process continues until the necessary conditions are found or no further derivation is possible. It allows the system to determine if the required facts are already present or prompts the system to search for rules that can derive those conditions.



Below is a detailed explanation of how this algorithm was implemented in our program:

- prove(string q) – The prove method aims to determine the validity of a given query within a set of facts and clauses. It first checks if the query is present in the facts. If it is, the query is considered true and added to the list of returned facts. The method then returns true.

  If the query is not found in the facts, the method loops through all the clauses and compares the query with the arrow part of each clause. If a match is found, the method adds the query to the list of returned facts and evaluates the individual facts or sub-queries within the clause. It recursively calls the prove method for each item in the first list of the clause. If any of the recursive calls return true, it means that the facts or sub-queries have been proven, and the method returns true.

If none of the facts or sub-queries can be proven for any clause, the method returns false, indicating that the query could not be proven based on the given facts and clauses.

- checkFacts – is a helper method that is used to determine whether a given query can be proven based on the available facts and clauses.
- test – method is responsible for generating a string that displays the result of proving the query. It first calls the checkFacts() method to determine whether the query can be proven.

Backward chaining tests

Test case	**Program output**	**Expected output**
TELL c=>q;l&b=>c;b&l=>m;a&p=>l;a&b=>l;a;b; ASK q	YES: b,a, l, c, q	YES: a, b, l, c, q
TELL p1=> p2; p2=> p3; p3=> p4; p4=> p5; p5=> p7; p9=> p14; p1; ASK p14	NO	NO
TELL p => q; l&m => p; a&b&l => m; a&p => l; a&b => l; a; b; ASK q	YES: b, m, a, l, p, q	YES: a, b, l, m, p ,q
TELL p1 => p2; p2 => p3; p3 => p4; p1; p3; ASK p2	YES: p1, p2	YES: p1, p2
TELL c=>q;l&b=>c;b&l=>m;a&p=>l;a&b=>l;a;b;q; ASK	YES: a,l,b,m	YES: a,l,b,m

m		
TELL p1 => p2; p2 => p3; p3 => p4; p4 => p5; p5 => p7; p9 => p14; p1; p5; ASK p5	YES: p5	YES:p5
TELL p => q; l&m => p; a&b&l => m; a&p => l; a&b => l; a; b; p; ASK q	YES: p,q	YES: p,q
TELL p1 => p2; p2 => p3; p3 => p4; p1; p3; ASK p4	YES:p3,p4	YES:p3,p4
TELL p => q; l&m => p; a&b&l => m; a&p => l; a&b => l; a; b; p; ASK f	NO	NO
TELL p => q; l&m => p; a&b => m; a&p => l; a&q => l; a; b; p; ASK m	YES: b,a,m	YES: b,a,m
TELL p1 => p2; p2 => p3; p3 => p4; p4 => p5; p5 => p7; p8 => p6; p9 => p14; p1; p8 ASK p6	YES: p8, p6	YES: p8, p6
TELL c&p => q; l&m => p; a&b&l => m; a&p => l; a; b; p; c; ASK q	YES: p,c,q	YES: p,c,q
TELL p1 => p2; p2 => p3; p3 => p4; p5&p6 => p7; p1; p3; p5; ASK p7	NO	NO

TELL c&q=>f;l&b=>c;b&l=>m;a&p=>l;a;b ;f; ASK f	YES:f	YES:f
TELL p1 => p2; p2 => p3; p3 => p4; p4 => p5; p5 => p7; p9&p12 => p14; p1; p5; p12; ASK p9	NO	NO

Bugs

BC is not able to properly handle circular dependencies as there are some lacking conditional statements when it comes to keeping track of visited clauses and proven facts and clauses. As for generic parsing it isn't made to handle more than one negation and this was mainly implemented based on similar formats to given general knowledge text files and horn text files.

Notes

The Shunting Yard algorithm is implemented to also handle more than one characters as an operand such as p2, p14. The parser is also able to handle more than two operands between & operator, so BC and FC are also able to handle that as well.

Research

For one aspect of the research component, we've implemented the Shunting Yard algorithm that will convert the expression into a postfix expression, which will allow the Truth Table checking algorithm to be able to handle both horn form knowledge bases as well as generic knowledge bases. Another thing worth mentioning is that the Shunting Yard algorithm is implemented based on the precedence value that the unit convenor mentioned. The truth table class was also changed to evaluate the postfix expression using primarily switch case statements. Furthermore we also attempted to implement the resolution algorithm however we were not successful as we found the conversion of the generic-clause to the CNF difficult. This was mainly due to the fact that we struggled to follow the proper steps to simplify the generic-clause in terms of coding it.

Team Summary

Overall the work allocation was split approximately equally. Tousif worked on parsing the Horn-form clauses (20%) and implementing the forward chaining algorithm (20%). Moving to Nadelin's contribution, she worked on implementing the truth table algorithm for

Horn-form clauses (20%), backward chaining algorithm (20%) and the truth table algorithm for generic-form clauses (20%). Lastly both of us have an equal amount of contribution for the report, where we each wrote the implementations of the classes we were assigned to. Nadelin wrote the Research, Notes and Bugs, whereas Tousif made the test cases, wrote the Acknowledgement, as well as cite the references and summary.

Acknowledgement

- https://www.youtube.com/watch?v=EZJs6w2YFRM&t=711s&pp=ygVBZm9yd2FyZCBjaGFpbmluZyBhbmQgYmFja3dhcmQgY2hhaW5pbmcgaW4gYXJ0aWZpY2lhbCBpbnRlbGxpZ2VuY2U%3D - This YouTube video allowed us to get a greater understanding of the forward and backward chaining algorithms in greater detail since they've provided many examples to allow us to have a better understanding. This enabled us to implement the algorithms smoothly in our program.
- https://swinburne.instructure.com/courses/49155/pages/assignment-helper?module_item_id=3421270- - This video provided by the tutor has been extremely helpful in explaining the requirements of the assignment as well as helping us get started on it. It gave us some insight on the number of classes required for this particular assignment. It also greatly helped with truth table checking algorithm as we were initially struggling with the process of evaluating all the clauses and facts against the query since there were so many.
- https://mathworld.wolfram.com/HornClause.html- This helped with identifying what a horn form knowledge base is and showed a list of its features. This helped with parsing the text file and creating the appropriate horn class to store each horn clause object.
- https://www.youtube.com/watch?v=SWxpkZ_SzaA&pp=ygUDY25m - This YouTube helped me to understand the process of converting a generic-clause to a CNF.

References

[1]https://www.engati.com/glossary/forward-chaining [Online][Accessed 20 April 2023]
[2]https://www.javatpoint.com/forward-chaining-and-backward-chaining-in-ai[Online] [Accessed 15 May 2023]
[3]https://mathcenter.oxford.emory.edu/site/cs171/shuntingYardAlgorithm/[Online][Accessed 16 May 2023]