SWINBURNE UNIVERSITY OF TECHNOLOGY

HO CHI MINH CAMPUS

COS30019 – Introduction to AI

Assignment 2 – Inference Engine

Student Name: Mai Ngoc Anh Doan

Student ID: 104220776

Instructor: Dr. Huy Truong

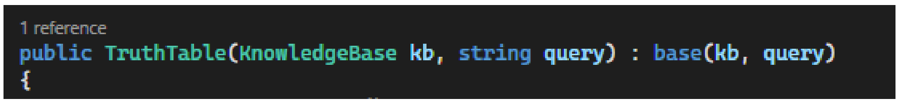
Table of Contents

1. Features/Bugs/Missing: 3
   1.1. Knowledge Base & Query: 3
   1.2. Horn Clause: 4
   1.3. Forward Chaining: 6
   1.4. Backward Chaining: 7
   1.5. Truth Table: 9
2. Test Cases: 12
   2.1. Test Case #1 – Simple set of HornClause 12
   2.2. Test Case #2 – Query can not be proven 12
   2.3. Test Case #3 – A query is a fact 13
   2.4. Test Case #4 – Irrelevant HornClause 13
   2.5. Test Case #5 – Entailed literal is entailed by itself 13
   2.6. Test Case #6 – Query is not in the KB 13
   2.7. Test Case #7 – Test file from the assignment outline 14
3. Acknowledgements/Resources: 14

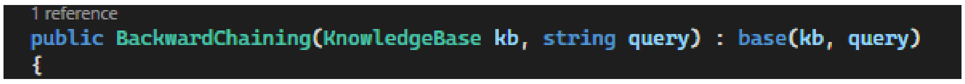
1. Features/Bugs/Missing:

1.1. Knowledge Base & Query:

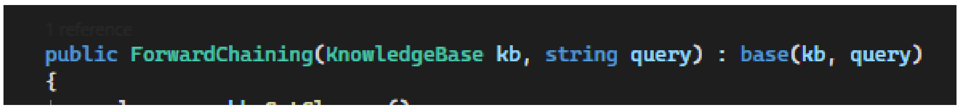
First of all, my three algorithms all take ‘KnowledgeBase kb’ and ‘string query’ as parameters so both of them will be introduced first.



```
1 reference
public TruthTable(KnowledgeBase kb, string query) : base(kb, query)
{
```



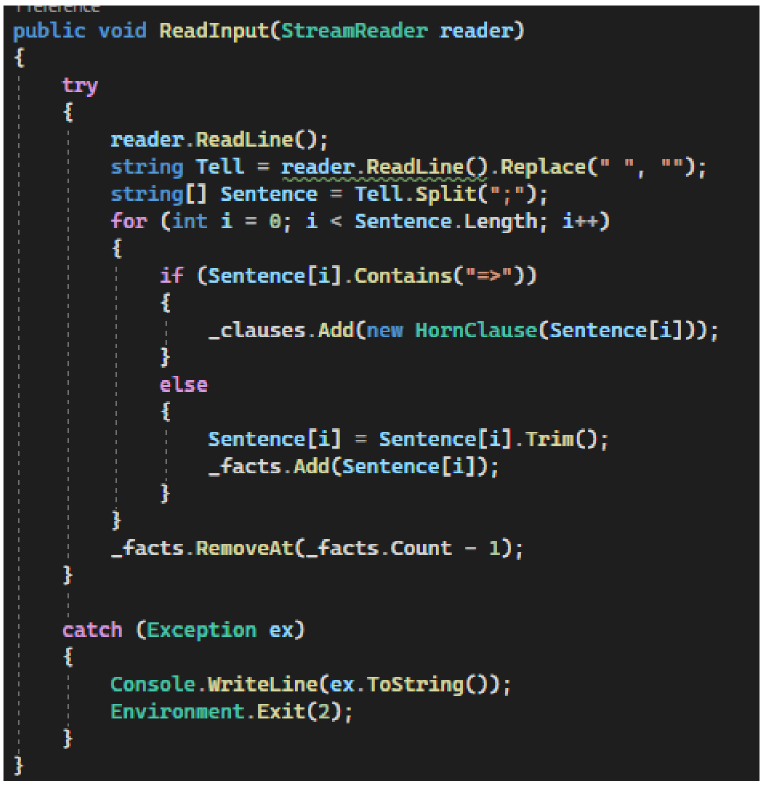
```
1 reference
public BackwardChaining(KnowledgeBase kb, string query) : base(kb, query)
{
```



```
1 reference
public ForwardChaining(KnowledgeBase kb, string query) : base(kb, query)
{
```

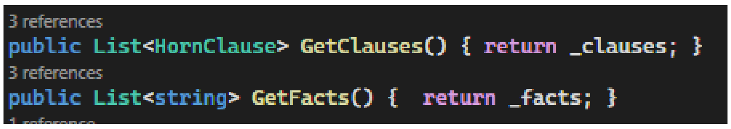
KnowledgeBase is where all the Horn clauses and facts are stored to check for entailment, all data after TELL line will be stored here. ‘string query’ is the sentence that this program need to prove whether it is entailed or not, this sentence located under ASK line in text file.

As I said above, the KnowledgeBase class will read the TELL statement and stored these clauses and facts in KB.



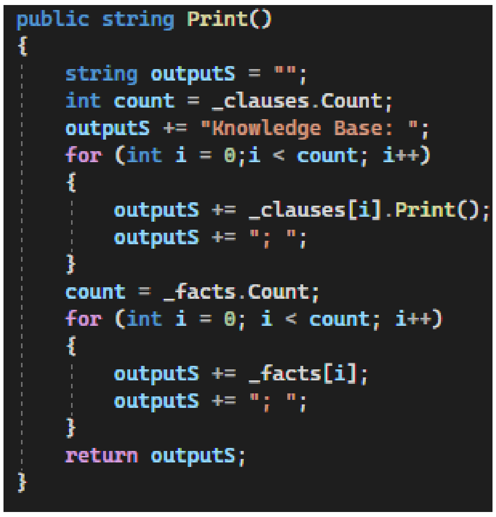
```
public void ReadInput(StreamReader reader)
{
    try
    {
        reader.ReadLine();
        string Tell = reader.ReadLine().Replace(" ", "");
        string[] Sentence = Tell.Split(";");
        for (int i = 0; i < Sentence.Length; i++)
        {
            if (Sentence[i].Contains("=>"))
            {
                _clauses.Add(new HornClause(Sentence[i]));
            }
            else
            {
                Sentence[i] = Sentence[i].Trim();
                _facts.Add(Sentence[i]);
            }
        }
        _facts.RemoveAt(_facts.Count - 1);
    }

    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
        Environment.Exit(2);
    }
}
```

The ReadInput procedure will be used to read text file into KnowledgeBase. The first line in text file is ‘TELL’ statement, there are nothing to do with this information so the program will disregard it. The second line is where the Horn clauses and facts located so those data is stored in ‘string Tell’ to handle. When reading the text file, there are many random whitespaces so this data need to be sanitized. By replacing all whitespaces with ‘nothing’, the whitespaces all deleted leaving the clean knowledge base. Sentences in KB is splited by the semi-colon. The program checks if the sentence contains ‘=>’ or not, if there is implication symbol in it, it is Horn clause and will be add to HornClause list to distinguish between literals and entailed literals. On the other hand, if it does not contain that symbol, the sentence will be added to fact list. The reason for '_facts.RemoveAt(_facts.Count – 1);' is that when splitting 'string Tell' with the semi-colons, the program automatically know that after a semi-colon is an element and is added to Sentence array. Because of that, there is a '/n' element after the last semi-colon resulting in an empty fact. This case appeared everytime so that I use that line of code to always delete the last empty element of the fact list.



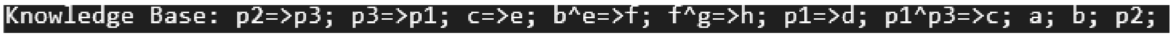
```
3 references
public List<HornClause> GetClauses() { return _clauses; }
3 references
public List<string> GetFacts() {  return _facts; }
```

In order to access clauses and facts in other algorithms' classes, there are 2 functions to get the Horn clauses and facts. The 'GetClauses()' function returns a List of HornClauses stored in the KnowledgeBase. The 'GetFacts()' function returns a List of string, containing facts stored in KnowledgeBase.



```
public string Print()
{
    string outputS = "";
    int count = _clauses.Count;
    outputS += "Knowledge Base: ";
    for (int i = 0;i < count; i++)
    {
        outputS += _clauses[i].Print();
        outputS += "; ";
    }
    count = _facts.Count;
    for (int i = 0; i < count; i++)
    {
        outputS += _facts[i];
        outputS += "; ";
    }
    return outputS;
}
```

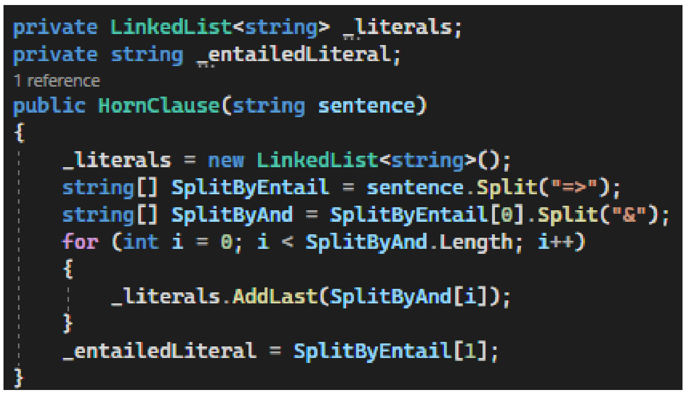
The last function in KnowledgeBase is 'Print()' function, this one will return the KnowledgeBase that has been sanitized and use '^' symbol for 'AND' instead of '&' as the text file. The image under will show how the KnowledgeBase is handled.



```
Knowledge Base: p2=>p3; p3=>p1; c=>e; b^e=>f; f^g=>h; p1=>d; p1^p3=>c; a; b; p2;
```

1.2. Horn Clause:

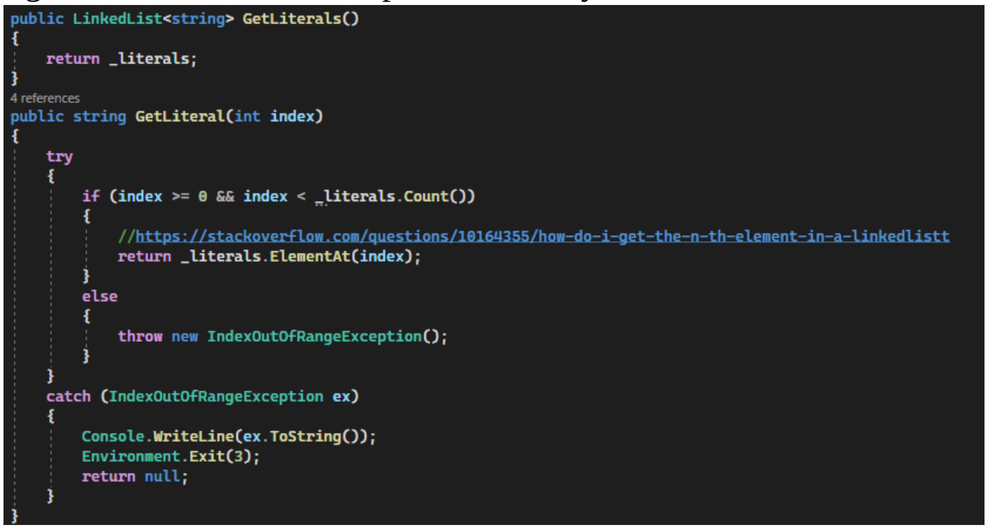
To store HornClause in my KnowledgeBase, the program uses HornClause class to handle all HornClauses that given by the text file.



```
private LinkedList<string> _literals;
private string _entailedLiteral;
1 reference
public HornClause(string sentence)
{
    _literals = new LinkedList<string>();
    string[] SplitByEntail = sentence.Split("=>");
    string[] SplitByAnd = SplitByEntail[0].Split("&");
    for (int i = 0; i < SplitByAnd.Length; i++)
    {
        _literals.AddLast(SplitByAnd[i]);
    }
    _entailedLiteral = SplitByEntail[1];
}
```

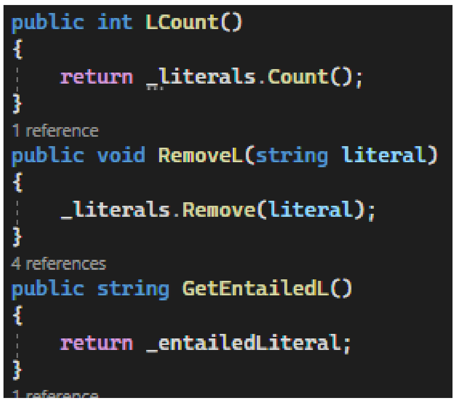
In HornClause class, I distinguish between literals which are sentences on the left side of '=>' and entailedLiteral which is on the right side of the implication symbol. In this program, the list that is used to store literals is LinkedList (represent as string) because in a HornClause, the number of literals is ≤ 2. In the case of 2 literals, both the literals need to know their next or previous literal in order to infer the entailedLiteral. In order to do that, the LinkedList<>() is preferred over List<>() since List<>() is not the best choice for this circumstance.

A new HornClause is constructed based on the string provided in the 'string sentence' parameter. At first, the sentence will be splitted by the '=>' into the left side symbolized for literals and the right side symbolize for entailedLiteral, both sides are put into an array called SplitByEntail. After that, the first argument of SplitByEntail array is splitted one more time by the '&' if this argument contains 2 literals and these 2 literals will be added to SplitByAnd array. There is a 'for' loop to loop through SplitByAnd array then store all the literals in the 'LinkedList<string> _literals'. Back to the SplitByEntail array, since the first argument contains literals, the second argument with index 1 symbolized for the right side of implication symbol will be entailedLiteral.



```
public LinkedList<string> GetLiterals()
{
    return _literals;
}
4 references
public string GetLiteral(int index)
{
    try
    {
        if (index >= 0 && index < _literals.Count())
        {
            //https://stackoverflow.com/questions/10164355/how-do-i-get-the-n-th-element-in-a-linkedlistt
            return _literals.ElementAt(index);
        }
        else
        {
            throw new IndexOutOfRangeException();
        }
    }
    catch (IndexOutOfRangeException ex)
    {
        Console.WriteLine(ex.ToString());
        Environment.Exit(3);
        return null;
    }
}
```

The 'GetLiterals()' function returns the LinkedList of string that contains the literals are referred to by the current HornClause (excluding the entailedLiteral). The next function is GetLiteral(int index), this one is used to get element at index. In this function, the program need to make sure that the index of the literal that needed is smaller than the number of literals had in current HornClause. Since this is the first time that I work with LinkedList on C#, I have trouble access element by index but I can figure it out and use ElementAt(index) to get the element at index.

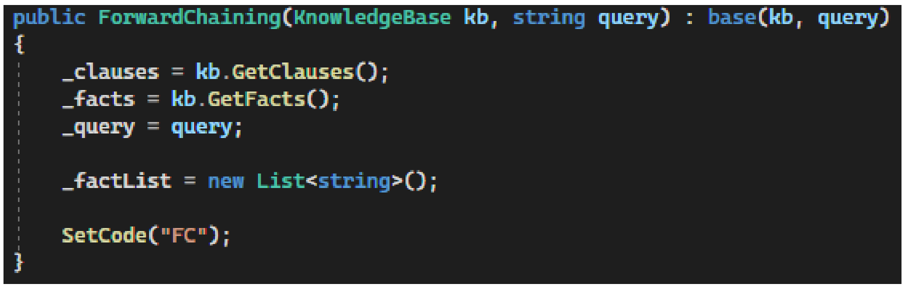


```
public int LCount()
{
    return _literals.Count();
}
1 reference
public void RemoveL(string literal)
{
    _literals.Remove(literal);
}
4 references
public string GetEntailedL()
{
    return _entailedLiteral;
}
1 reference
```

These function respectively counting literals, removing literal and get entailedLiteral of the current HornClause.

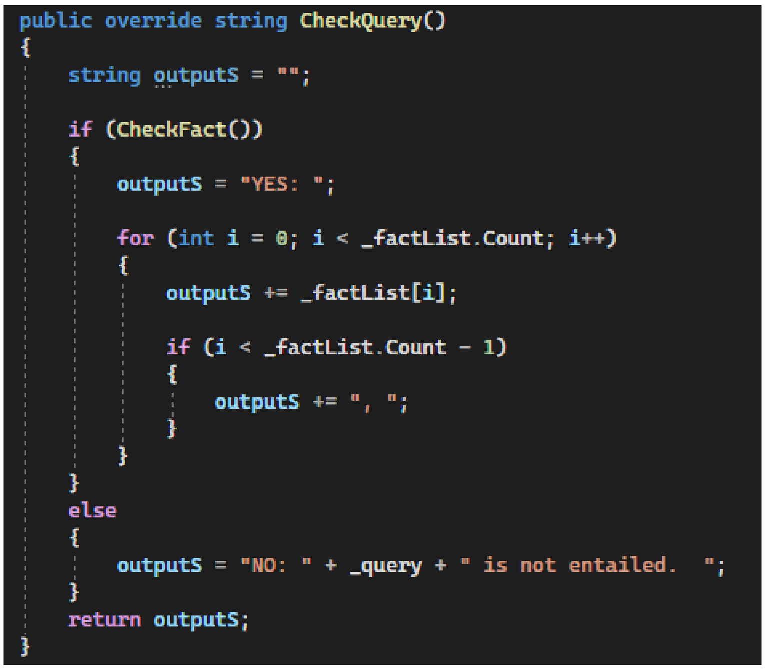
1.3. Forward Chaining:

The first algorithm is introduced in this report is ForwardChaining that takes 2 parameters which are KnowledgeBase kb and ‘string query’.



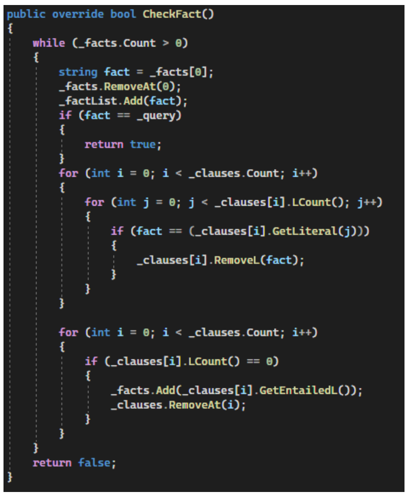
```
public ForwardChaining(KnowledgeBase kb, string query) : base(kb, query)
{
    _clauses = kb.GetClauses();
    _facts = kb.GetFacts();
    _query = query;

    _factList = new List<string>();

    SetCode("FC");
}
```

HornClause and facts are parsed from the KnowledgeBase with ‘GetClauses()’ and ‘GetFacts()’ function. The List of string name ‘_factList’ is to store each fact that has been discovered. In my three algorithms, there are code to call the algorithm with batch file.



```
public override string CheckQuery()
{
    string outputS = "";

    if (CheckFact())
    {
        outputS = "YES: ";

        for (int i = 0; i < _factList.Count; i++)
        {
            outputS += _factList[i];

            if (i < _factList.Count - 1)
            {
                outputS += ", ";
            }
        }
    }
    else
    {
        outputS = "NO: " + _query + " is not entailed.  ";
    }
    return outputS;
}
```

In my Algorithm class, there are 2 abstract function which are ‘CheckQuery()’ and ‘CheckFact()’. The ‘CheckQuery()’ function is used to call the ‘CheckFact()’ and if it returns ‘true’ then ‘YES:’ will be printed with the ‘_factList’ that contains all the facts have been discovered. On the other hand, if the ‘CheckFact()’ function returns ‘false’, the output string will be ‘NO:’.



```
public override bool CheckFact()
{
    while (_facts.Count > 0)
    {
        string fact = _facts[0];
        _facts.RemoveAt(0);
        _factList.Add(fact);
        if (fact == _query)
        {
            return true;
        }
        for (int i = 0; i < _clauses.Count; i++)
        {
            for (int j = 0; j < _clauses[i].LCount(); j++)
            {
                if (fact == (_clauses[i].GetLiteral(j)))
                {
                    _clauses[i].RemoveL(fact);
                }
            }
        }

        for (int i = 0; i < _clauses.Count; i++)
        {
            if (_clauses[i].LCount() == 0)
            {
                _facts.Add(_clauses[i].GetEntailedL());
                _clauses.RemoveAt(i);
            }
        }
    }
    return false;
}
```

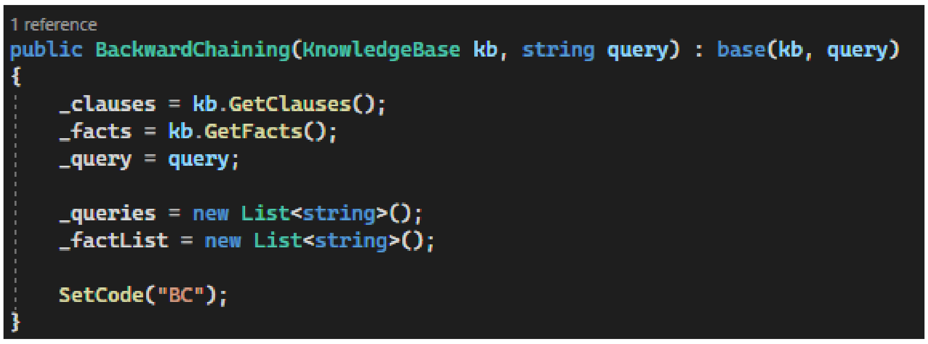
In every algorithms that this program provided, the ‘CheckFact()’ always plays a vital role in infering entailment, it is responsible for checking the KnowledgeBase to know whether the KB entails the query or not. Firstly, the function will search through each fact until there are no fact left in ‘_facts’. In this big loop, the first fact will be pop off to explore and is added to ‘_factList’. If the current exploring fact is equal to the query, the function will return ‘true’ immeadiately. Still in the loop but now, the facts are checked so the function moves on to check the

HornClause. The first ‘for’ loop, the program runs through each clause. With the current clause, it loops through the number of literals and if the current exploring fact is the same as the literal, the current clause will remove it as the literal from that HornClause. Move to the next ‘for’ loop, this loop will get the literal count of each clause and if it equals 0, the entailedLiteral will be added to the end of the list of facts and the clause also is removed from the list of clauses to stop the program from discovering the same fact over and over. If there are no facts to discover but the program still can not prove the entailment, then the ‘CheckFact()’ function will return ‘false’.

Overall, the ForwardChaining algorithm works perfectly and meet the requirements of the assignment without any bugs.

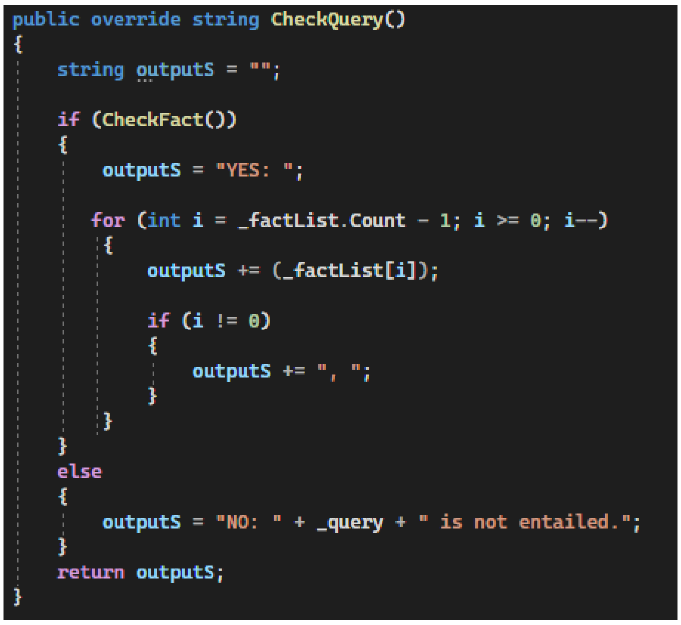
1.4. Backward Chaining:

On the contrary, if the ForwarchChaining algorithm begins with looping through facts and clauses to find the sentence that is the same as the query and return result, the BackwardChaining will begin with the query and make all his way back to infer entailment.



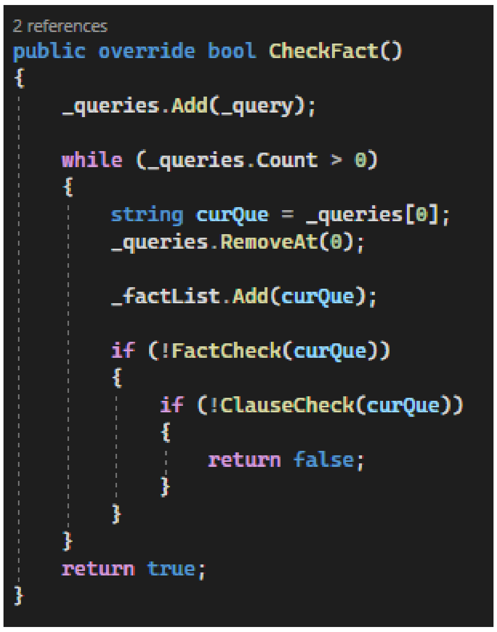
```
1 reference
public BackwardChaining(KnowledgeBase kb, string query) : base(kb, query)
{
    _clauses = kb.GetClauses();
    _facts = kb.GetFacts();
    _query = query;

    _queries = new List<string>();
    _factList = new List<string>();

    SetCode("BC");
}
```

In BackwardChaining algorithm, there is a new list of string that store queries that need to be proven.



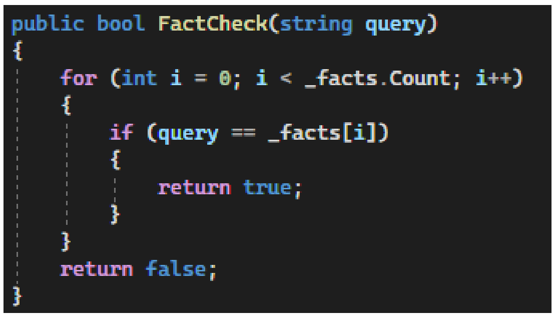
```
public override string CheckQuery()
{
    string outputS = "";

    if (CheckFact())
    {
        outputS = "YES: ";

        for (int i = _factList.Count - 1; i >= 0; i--)
        {
            outputS += (_factList[i]);

            if (i != 0)
            {
                outputS += ", ";
            }
        }
    }
    else
    {
        outputS = "NO: " + _query + " is not entailed.";
    }
    return outputS;
}
```

Different from the ForwarChaining, the ‘_factList’ will be printed reversely because the BC goes from the asked query, not the KnowledgeBase like FC.



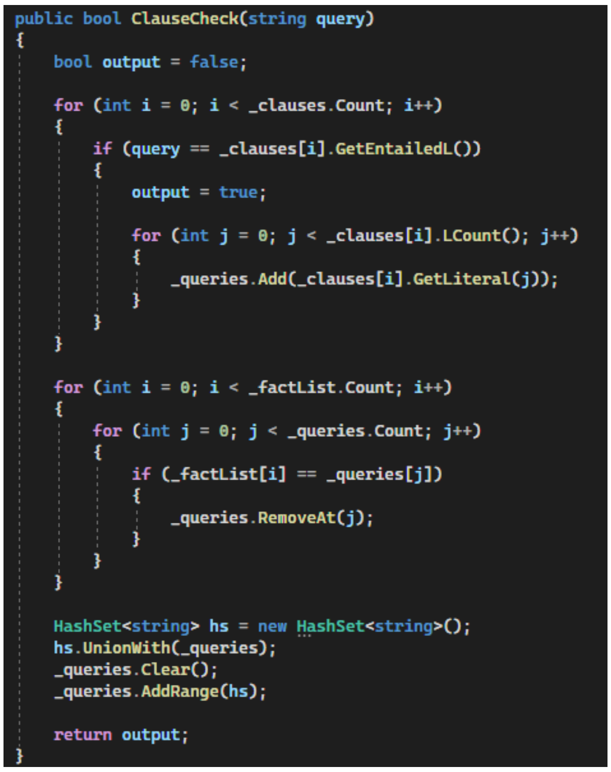
```
2 references
public override bool CheckFact()
{
    _queries.Add(_query);

    while (_queries.Count > 0)
    {
        string curQue = _queries[0];
        _queries.RemoveAt(0);

        _factList.Add(curQue);

        if (!FactCheck(curQue))
        {
            if (!ClauseCheck(curQue))
            {
                return false;
            }
        }
    }
    return true;
}
```

As been said above, '_queries' is the list of query that need to be proven so the program will put the asked query on top of this list. While there is still query that need to be proven, this function will pop off the first query and check it. That current query will be added to the list of queries that are used to infer output. On the next step, the current query will be checked with all the fact, if there is no match, the current query will move on to be checked with HornClause. If it matches, the literal(s) on the left side of the clause will be added to the '_queries'. If the current query still can not find the match until this step, the function will return 'false' and the search has failed. When the queries is empty, the program jumps out of the loop meaning that there is no query left to prove, the seach has successed.



```
public bool FactCheck(string query)
{
    for (int i = 0; i < _facts.Count; i++)
    {
        if (query == _facts[i])
        {
            return true;
        }
    }
    return false;
}
```

This function takes query as a parameter. It will loop through all facts of the KnowledgeBase and find a match. If a match is found, this function will return 'true'. On the other hand, it will return false and the query will be brought to check with HornClauses.



```
public bool ClauseCheck(string query)
{
    bool output = false;

    for (int i = 0; i < _clauses.Count; i++)
    {
        if (query == _clauses[i].GetEntailedL())
        {
            output = true;

            for (int j = 0; j < _clauses[i].LCount(); j++)
            {
                _queries.Add(_clauses[i].GetLiteral(j));
            }
        }
    }

    for (int i = 0; i < _factList.Count; i++)
    {
        for (int j = 0; j < _queries.Count; j++)
        {
            if (_factList[i] == _queries[j])
            {
                _queries.RemoveAt(j);
            }
        }
    }

    HashSet<string> hs = new HashSet<string>();
    hs.UnionWith(_queries);
    _queries.Clear();
    _queries.AddRange(hs);

    return output;
}
```

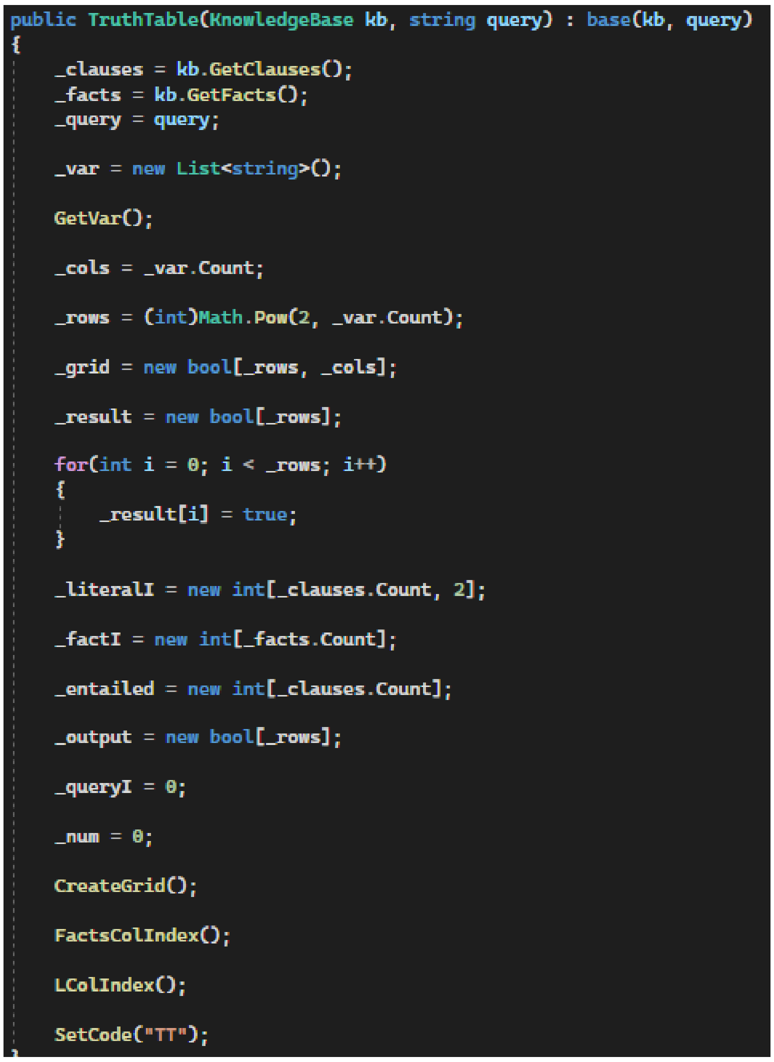
After being checked with all the facts, the query will be checked with HornClauses in KnowledgeBase. At the beginning, the output is set to 'false'. The function will loop through all the clauses to get its entailedLiteral and if there is a match of query and the entailedLiteral, the output will be set to 'true' and the literal(s) that on the left side of that clause will also be added to '_queries' to continue to prove.

The next method is used to prevent the BackwardChaining to reaccessing the literals that have been explored by checking whether there is any duplicate elements in the '_factList' and '_queries'. If there is a match, that element in the '_queries' will be removed. There is another safeguard to make sure that literals are not explored multiple time by using HashSet to clean the List of duplicate values.

All in all, the BackwardChaining works perfectly and can find the shortest way to infer entailment. It meets the requirement of the assignment without any bugs.

1.5. Truth Table:

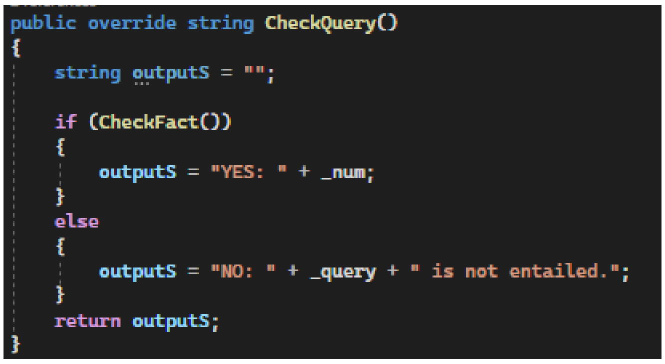
The last algorithm in my program is TruthTable, this is also the easiest and toughest algorithm to work with in my opinion. It is easy because I can understand the theory right away when first learnt about it but when comes to practice like this, I just can not put the theory into code which cost me an enormous amount of time for this algorithm and still not achieve perfection.



```
public TruthTable(KnowledgeBase kb, string query) : base(kb, query)
{
    _clauses = kb.GetClauses();
    _facts = kb.GetFacts();
    _query = query;

    _var = new List<string>();

    GetVar();

    _cols = _var.Count;

    _rows = (int)Math.Pow(2, _var.Count);

    _grid = new bool[_rows, _cols];

    _result = new bool[_rows];

    for(int i = 0; i < _rows; i++)
    {
        _result[i] = true;
    }

    _literalI = new int[_clauses.Count, 2];

    _factI = new int[_facts.Count];

    _entailed = new int[_clauses.Count];

    _output = new bool[_rows];

    _queryI = 0;

    _num = 0;

    CreateGrid();

    FactsColIndex();

    LColIndex();

    SetCode("TT");
```

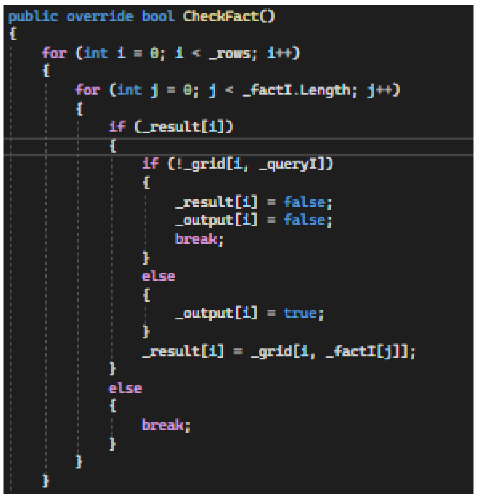
In TruthTable algorithm, there are many new fields to work with. The first one is '_var' which is the list of string that store variables for this algorithm. The variables are got from the 'GetVar()' function that this report will explain later. In this class, I will create a table and check rows by rows with the number of columns equals the number of variables and the number of rows equals 2^n with n is the number of columns. The table is constructed by the '_grid' field with the number of rows and columns above. Because this is a TruthTable so that this table will hold boolean value. The '_result' field will be used to determine the result of each row as a final column. At the beginning, each row will result in 'true' value. In this program, the facts and HornClauses need to be assigned with an index so that the program can access it easier in TruthTable. An array that holds entailedLiteral is '_entailed' and has '_clauses.Count' elements. The '_output' field is the query result in this TruthTable.

And the final field is '_num' that stands for number of model that contains the query. The 'CreateGrid()' function is responsible for populating grid to construct a table. The column index for every fact in the TruthTable grid is got from 'FactsColIndex()' function and those for every literals is got from 'LColIndex()' function.



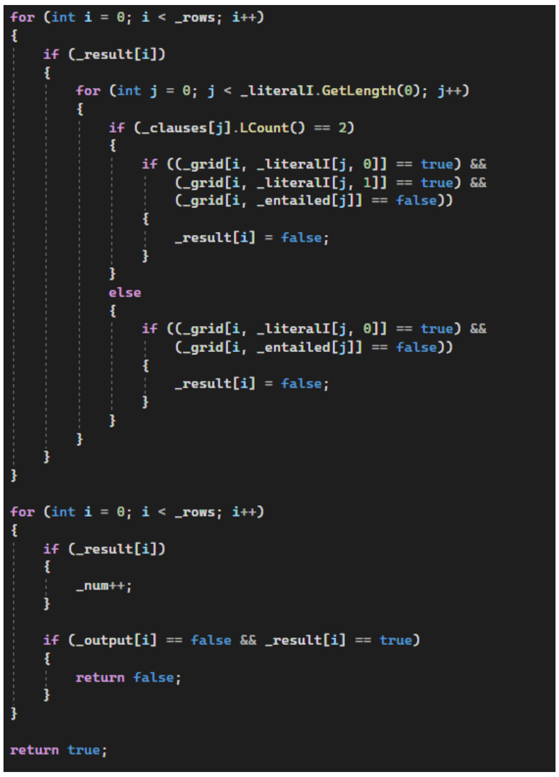
```
public override string CheckQuery()
{
    string outputS = "";

    if (CheckFact())
    {
        outputS = "YES: " + _num;
    }
    else
    {
        outputS = "NO: " + _query + " is not entailed.";
    }
    return outputS;
}
```

In lieu of printing '_factList' as Forward Chaining or Backward Chaining, the Truth Table algorithm prints the number of models that contains the query.



```
public override bool CheckFact()
{
    for (int i = 0; i < _rows; i++)
    {
        for (int j = 0; j < _factI.Length; j++)
        {
            if (_result[i])
            {
                if (!_grid[i, _queryI])
                {
                    _result[i] = false;
                    _output[i] = false;
                    break;
                }
                else
                {
                    _output[i] = true;
                }
                _result[i] = _grid[i, _factI[j]];
            }
            else
            {
                break;
            }
        }
    }
```

In the first 'for' loop of this function, this loop checking the state of each fact for every row. If any of facts in the row is false, the whole row will automatically false. The 'if' condition is used to prevent any false rows to be rewritten. The second 'if' stands for if the query is false, the whole row is false. On the other hand, the query is 'true'. After that, the value of row result will equal to the value of current row and fact index column.

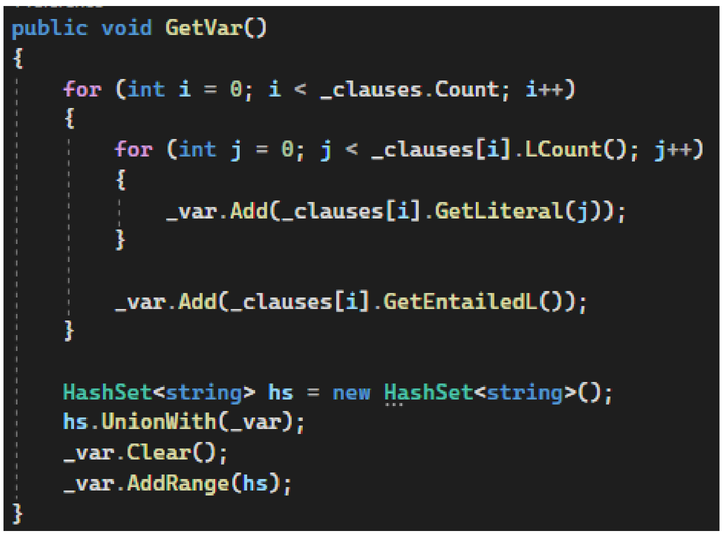


```
for (int i = 0; i < _rows; i++)
{
    if (_result[i])
    {
        for (int j = 0; j < _literalI.GetLength(0); j++)
        {
            if (_clauses[j].LCount() == 2)
            {
                if ((_grid[i, _literalI[j, 0]] == true) &&
                    (_grid[i, _literalI[j, 1]] == true) &&
                    (_grid[i, _entailed[j]] == false))
                {
                    _result[i] = false;
                }
            }
            else
            {
                if ((_grid[i, _literalI[j, 0]] == true) &&
                    (_grid[i, _entailed[j]] == false))
                {
                    _result[i] = false;
                }
            }
        }
    }
}

for (int i = 0; i < _rows; i++)
{
    if (_result[i])
    {
        _num++;
    }

    if (_output[i] == false && _result[i] == true)
    {
        return false;
    }
}

return true;
```

After checking the state of each facts, the function moves on to check the state of each literal for every row. One more time, the first 'if' condition is used to make sure no false row get rewritten. The function loop through each column of literals and if the clause that has 2 literals, the 2 literals are true but the entailedLiteral is false leading to the result of that row is false. In addition, if the clause has only one literal and it is true but the entailedLiteral still false, the result of that row is false too.

The last 'for' loop will count the number of rows that are true. The loop runs through every row, where the result is true, the number of models will plus 1.

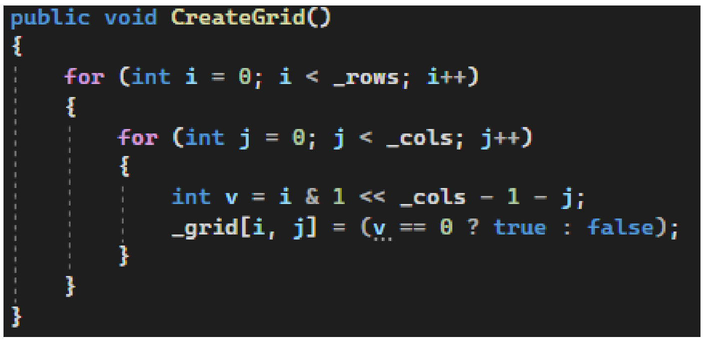
The only way that the search can return false is if KB ⊨ α if and only if (KB ^ ¬α) is unsatisfiable. After looping through all the checking, the function will return 'true' and output the number of models.



```
public void GetVar()
{
    for (int i = 0; i < _clauses.Count; i++)
    {
        for (int j = 0; j < _clauses[i].LCount(); j++)
        {
            _var.Add(_clauses[i].GetLiteral(j));
        }

        _var.Add(_clauses[i].GetEntailedL());
    }

    HashSet<string> hs = new HashSet<string>();
    hs.UnionWith(_var);
    _var.Clear();
    _var.AddRange(hs);
}
```

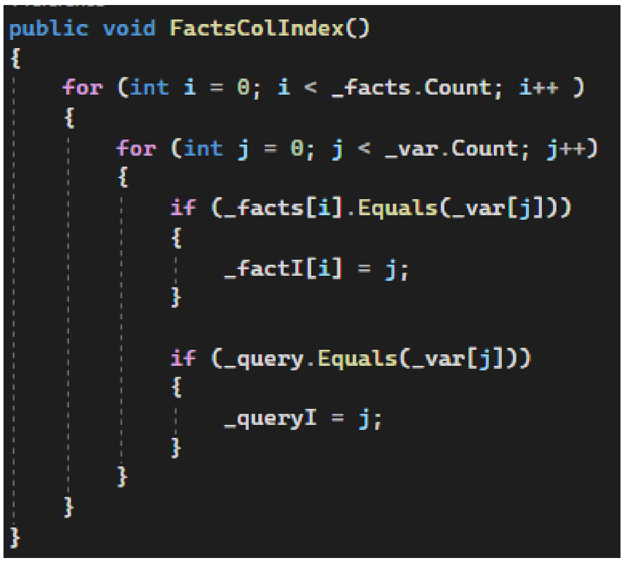
This function adds every literal of every HornClause into the list of variables. It loops through all the clause and its literal(s). The variable list first adds literals from left of the entailment, then add literal from the right side of impliccation symbol.

Just like the BC algorithm, the HashSet is used to remove repeated elements in a List.



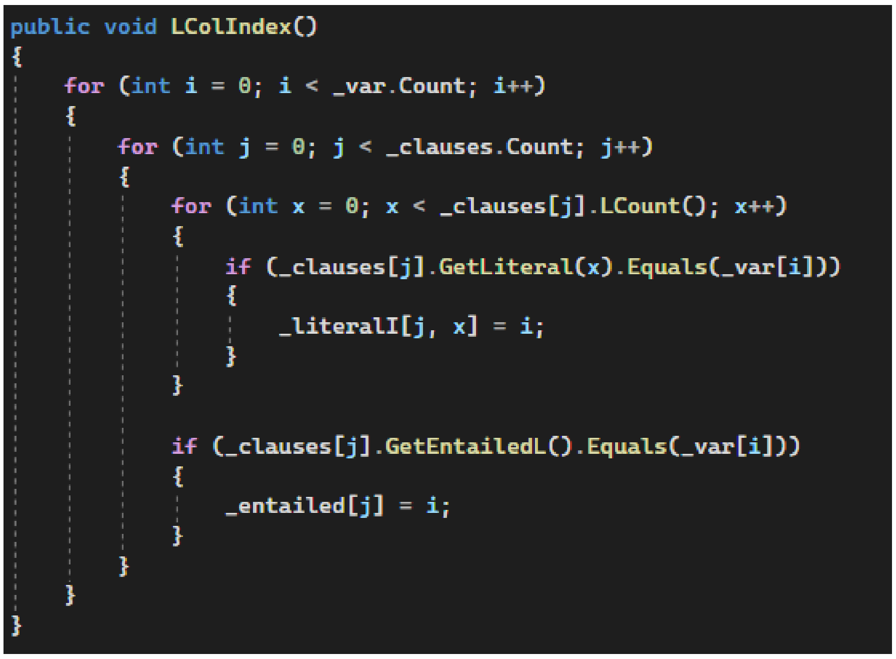
```
public void CreateGrid()
{
    for (int i = 0; i < _rows; i++)
    {
        for (int j = 0; j < _cols; j++)
        {
            int v = i & 1 << _cols - 1 - j;
            _grid[i, j] = (v == 0 ? true : false);
        }
    }
}
```

The use of this function is to populate grid for TruthTable with the number of rows and columns that have been calculated above. This table contains boolean value only.



```
public void FactsColIndex()
{
    for (int i = 0; i < _facts.Count; i++ )
    {
        for (int j = 0; j < _var.Count; j++)
        {
            if (_facts[i].Equals(_var[j]))
            {
                _factI[i] = j;
            }

            if (_query.Equals(_var[j]))
            {
                _queryI = j;
            }
        }
    }
}
```

The FactsColIndex() function is used to get the index from all the facts. This function loops through the facts and the variables and check if there is a match of fact and variable. If there is a match, the fact index of current fact is the index of the variable. It also check if the query equal to the variable or not. If it is equal, the query index is the index of the current variable.



```
public void LColIndex()
{
    for (int i = 0; i < _var.Count; i++)
    {
        for (int j = 0; j < _clauses.Count; j++)
        {
            for (int x = 0; x < _clauses[j].LCount(); x++)
            {
                if (_clauses[j].GetLiteral(x).Equals(_var[i]))
                {
                    _literalI[j, x] = i;
                }
            }

            if (_clauses[j].GetEntailedL().Equals(_var[i]))
            {
                _entailed[j] = i;
            }
        }
    }
}
```

After getting index from the list of facts, the index of literals of HornClause also necessary. If the literal(s) of the current HornClause equals to the current variable and then the index of the current clause with current literal will be the index of current variable. Moreover, if the entailedLiteral equals the variable then the index of entailedLiteral is the index of current variable too.

In conclusion, my TruthTable algorithm meets the requirement of the assignmnet with a minor bug that it can not prove entailment of an irrelevant query.

2. Test Cases:

To run the program, the basic command line is used with this form:

> iengine method filename

2.1. Test Case #1 – Simple set of HornClause

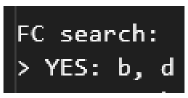
TELL

a => b; b=> d; d => c; b;

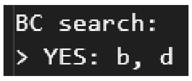
ASK

d

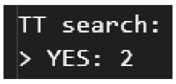
OUTPUT:



```
FC search:
> YES: b, d
```



```
BC search:
> YES: b, d
```



```
TT search:
> YES: 2
```

2.2. Test Case #2 – Query can not be proven

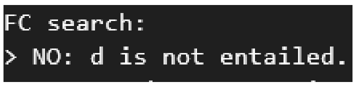
TELL

a => b; b=> d; d => c;

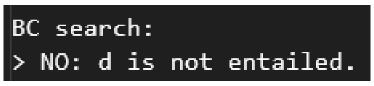
ASK

d

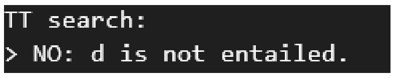
OUTPUT:



```
FC search:
> NO: d is not entailed.
```



```
BC search:
> NO: d is not entailed.
```



```
TT search:
> NO: d is not entailed.
```

2.3. Test Case #3 – A query is a fact

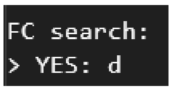
TELL

a => b; b=> d; d => c; d;

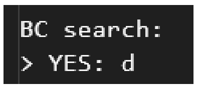
ASK

d

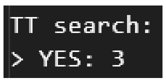
OUTPUT:



```
FC search:
> YES: d
```



```
BC search:
> YES: d
```



```
TT search:
> YES: 3
```

2.4. Test Case #4 – Irrelevant HornClause

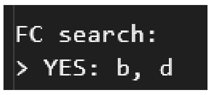
TELL

a => b; b=> d; d => c; m&n => o; b;

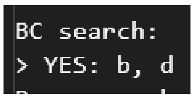
ASK

d

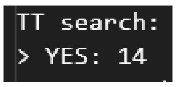
OUTPUT:



```
FC search:
> YES: b, d
```



```
BC search:
> YES: b, d
```



```
TT search:
> YES: 14
```

2.5. Test Case #5 – Entailed literal is entailed by itself

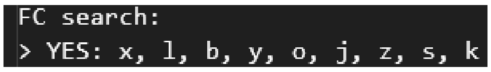
TELL

x&y => z; z&o = > s; s&j => k; l&d => d; d =>g; g=>o; b => j; x; l; b; y; o;

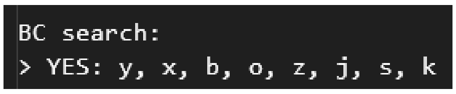
ASK

k

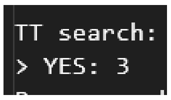
OUTPUT:



```
FC search:
> YES: x, l, b, y, o, j, z, s, k
```



```
BC search:
> YES: y, x, b, o, z, j, s, k
```



```
TT search:
> YES: 3
```

2.6. Test Case #6 – Query is not in the KB

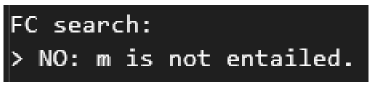
TELL

x&y => z; z&o = > s; s&j => k; l&d => d; d =>g; g=>o; b => j; x; l; b; y; o;

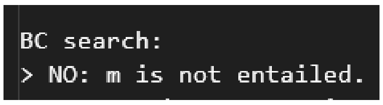
ASK

m

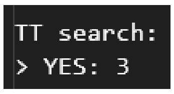
OUTPUT:



```
FC search:
> NO: m is not entailed.
```



```
BC search:
> NO: m is not entailed.
```



```
TT search:
> YES: 3
```

A bug is discovered, the result of TT search must be not entailed.

2.7. Test Case #7 – Test file from the assignment outline

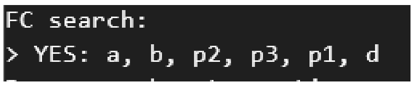
TELL

p2=> p3; p3 => p1; c => e; b&e => f; f&g => h; p1=>d; p1&p3 => c; a; b; p2;

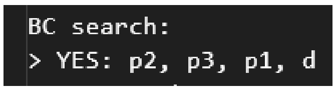
ASK

d

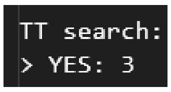
OUTPUT:



```
FC search:
> YES: a, b, p2, p3, p1, d
```



```
BC search:
> YES: p2, p3, p1, d
```



```
TT search:
> YES: 3
```

3. Acknowledgements/Resources:

- The Difference between List and LinkedList

This is my first time working with LinkedList so that I have trouble about how it different from the List. Thanks to the answer of Jon Skeet that helps me to understand clearly List and LinkedList so that I can briefly explain why I use LinkedList in this assignment.

- How do I get the n-th element in a LinkedList?

One more time, I have to say thank you to Jon Skeet answer in this topic. As I had said above, this is my first time working with LinkedList so I get trouble how to use the correct method. This topic help me get the element at index by using ‘ElementAt()’ method.

- Remove duplicates from a List in C#

Thanks to Even Mien for his answer in this topic. At the beginning, I intend to hardcode ‘if-else’ to deal with repeated element in a list but I soon realize that this is a tough way to deal with these elements so I did some research. Looking through all the answer in this topic, I choose using HashSet as recommendation of Even Mien because it is a cleanest way to deal with duplicates.

- Creat Truth Table

In this topic, I want to thank Dhass by recommending how to populate grid in code. Truly that my TruthTable algorithm may be very messy and unorganizing if I do not know how to populate grid in an efficient way.