KNOWLEDGE REPRESENTATION AND REASONING

UNIT ASSIGNMENT REPORT

Russell Sammut-Bonnici

ICS1019

April 27, 2018

Contents

Note

The two programs “task1.py” and “task2.py” were written using Python 2.7, in order to execute make use of the Python Interpreter specifically for version 2.7.

Task 1 – Horn Clauses (task1.py)

In this task, a program was required to parse and reason with an input collection of horn clauses. Using a text file to store these input horn clauses, the program made sense of the stream of characters and horn clauses were generated and stored in a knowledge base using objects.

The user is then prompted to query the knowledge base. The program recognizes the input query as a clause and resolves it with the knowledge base using SLD Backchaining, returning whether it has been "SOLVED" or "NOT SOLVED".

1.1 Object Oriented Logic

Class "Literal" was defined to be able to initialize literal objects which have their own string name (self.name) and boolean polarity (self.polarity). Literal objects would then be stored in object lists inside the "HornClause" objects.

Class "HornClause" was defined for an object that stores an object list (literalList) consisting of literal objects. A method addLiteral() was also made to be able to append literal objects to the object list self.literalList with ease.

1.2 Input Clauses

The input clauses were stored in the textfile "clauseStatements.txt". In the main the program calls **textToKnowledgeBase()** with the specified textfile name as the argument. A list of horn clauses are returned by this method and stored within the list variable **knowledgeBase** in the main.

In the method the program firstly initializes an empty clause list "formula". It then parses through the textfile character by character and from which it creates literals and horn clauses storing the literals into their appropriate horn clauses.

Through the use of conditional statements for the predicted characters '**[**' , '**!**' , '**,**' , '**]**' , '**\n**' and alphanumeric characters the program constructs the knowledge base.

'**[**' initialises a new HornClause object and a new Literal object.

'**!**' sets the current literal object's polarity to **False**.

Alphanumeric characters are appended to the current literal object's name

'**,**' adds the finished literal object to the current clause object's literalList and initializes a new literal.

'**]**' adds the finished literal to the hornClause and it also appends the finished clause to formula.

'**\n**' just prints out the current clause's literals along with their polarity before the next clause.

1.3 User Query

The method **requestQuery()** is called with no parameters, and the return value, that is the query horn clause, is stored in the variable "query" in main. When called the user is asked to input a query clause through the console. (An example of input would be [!Girl])

It then parses through this user input String using the same conditional logic used in the previous method for scanning the textfile. Except this time the ' **\n** ' condition is removed as for the query input there is only one line to input.

1.4 Resolution by SLD Backchaining

The recursive method **_resolve()** is called in the main passing the knowledge base and query clause as arguments. The method prints all of its steps to resolving in the console. When solved, the method prints "SOLVED" as well as returning the boolean value **True**. When unsolved, the method prints "NOT SOLVED" as well as returning **False**. The return value is stored in the variable "isSolved" in main.

The base case

The base case condition in the recursive method is when the literalList of the query clause becomes empty. This is detected by testing whether the literalList's length is equal to 0. This is accessed after matching and eliminating all the query clauses with clauses from the knowledge base. Upon reaching this stage the program prints "SOLVED" and returns **True**.

Traversing

Whilst the base case isn't as of yet reached, the program traverses through every literal of every clause in the knowledgeBase using 2 for loops. It then stays comparing the knowledge base literal with the literal at the end of the query clause literalList.

When a match is found

When a match between the literal names is found and opposite polarities are detected, it prints out this detection which is then followed by two conditions catering for two possibilities.

The first possibility, when the query clause has more than one literal. The variable "clauseClone" is set to the query clause. This is because the current query is still as of yet not empty, so it keeps eliminating the literals inside it from last to first until it is. Eventually, when it is it accesses the other condition. The variable clauseClone alongside knowledgeBase are passed into a recursive call of **_resolve()**. When backtracking the line after the recursive call returns **True** to go back up a depth.

The second possibility, when the query clause only has one literal. The variable "clauseClone" is set to the clause of the knowledge base. This is because the current query clause after removing its last

literal becomes empty, therefore it starts to then work with the not empty clause.literalList from the knowledgeBase as the next query for the next depth. The variable clauseClone alongside knowledgeBase are passed into a recursive call of **_resolve()**. When backtracking the line after the recursive call also returns **True** to go back up a depth.

When a match is not found

If the program goes through all the clauses in the knowledgeBase without finding a match in literal name and opposite polarity it then prints “NOT SOLVED” and returns **False**. This is placed outside the two for loops and is only ever accessed after scanning the whole knowledgeBase with no matches found.

1.5 Tested Data

In the textfile “clauseStatements.txt”

[Toddler]

[!Toddler,Child]

[!Child,!Male,Boy]

[!Infant,Child]

[!Child,!Female,Girl]

[Female]

[Male]

Tested Queries and their Returned Output

Query inputted: ***[!Girl]***

Match found : [!Child !Female Girl] with Query [!Girl]

Match found : [Female] with Query [!Child !Female]

Match found : [!Toddler Child] with Query [!Child]

Match found : [Toddler] with Query [!Toddler]

Empty List: []

SOLVED

Query inputted: ***[!Boy]***

Match found : [!Child !Male Boy] with Query [!Boy]

Match found : [Male] with Query [!Child !Male]

Match found : [!Toddler Child] with Query [!Child]

Match found : [Toddler] with Query [!Toddler]

Empty List: []

SOLVED

Query inputted: ***[!Elephant]***

NOT SOLVED

Query inputted: ***[Toddler]***

Match found : [!Toddler Child] with Query [Toddler]

Match found : [Toddler] with Query [!Toddler]

Empty List: []

SOLVED

Task 2 - Inheritance Networks (task2.py)

In this task, a program was required to construct and reason with an inheritance network based on the input statements provided. Using a textfile to store the the statements the program makes sense of the stream of characters and constructs a collection of edges from it using objects, resulting to a constructed inheritance network.

After constructing the network the program then accepts a user query from the console and from which it outputs all the possible paths and the preferred paths according to the short distance metric and the inferential distance metric.

2.1 Object Oriented Logic

Class “Node” was defined to be stores# Node objects with their own String name characteristic. Each node in the inferential base would then be referenced by at least one edge.

Class “Edge” was defined to store information regarding the edge. This includes whether the edge has a positive “IS-A” or negative “IS-NOT-A” polarity for inheritance, and it stores nodeA and nodeB which are to be used as pointers to the appropriate Node objects describing the edge. It has setter methods add_A(), add_B() and polarity() for setting each variable after initialization.

Class “Path” was defined for storing a list of edges. This list of edges is stored as pathList. The boolean type of the path was included, **True** for describing “IS-A” all throughout paths and **False** for describing paths with “IS-NOT-A” at the end. The length of the path was also included, which is calculated by getting the amount of edges stored in self.pathList.

2.2 Input Statements

The input statements were stored in the textfile “inheritanceNetwork.txt”. In the main the program calls **textToKnowledgeBase()** with the specified textfile name as the argument. A list of edges are returned by this method and stored within the list variable **knowledgeBase** in the main.

In the method the program firstly initializes an empty edge list “edgeList”. It then parses through the textfile character by character and from which it creates edge objects (pointing to node objects) with the appropriate edge polarity information, it adds all these edges to edgeList.

Through the use of conditional statements for the predicted characters ‘**\n**‘ ,‘**<**’ , ‘**>**’ the program constructs the knowledge base. The program makes use of the variable “flag” for deciding whether to appending other characters to node name (when flag = 1) or to relation (when flag = 0).

‘ **\n** ‘ initialises a new edge

‘ **<** ‘ initializes a new node and sets the flag to 1 for reading nodeName characters in the **else**.

‘ **>** ‘ if nodeA isn’t as of yet filled add Node tempN as nodeA, else when already filled add as nodeB

In the ‘ **>** ‘ condition when nodeB is added also store information on the edge’s relation by setting the edge’s polarity to **True** or **False** depending on whether tempR is “ **IS-A** “ or “ **IS-NOT-A** ” respectively.

else: When flag=1 append characters to tempN.name, when flag=0 append to tempR

Important to Note when Inputting Edges:

1. When inputting statements in the text file start the text with an empty line before the statements.
2. Encase the nodes in each statement using ‘<’ at the beginning and ‘>’ at the end.
3. Make sure the relations between the nodes have a space before and a space after.

Example input for a statement: “<Clyde> IS-A <FatRoyalElephant>”

2.3 User Query

The method **requestQuery()** is called with no parameters, and the return value, that is the query edge, is stored in the variable “query” in main. When called the user is asked to input a query edge through the console. (An example of input would be “<Clyde> IS-A <Gray>”)

It then parses through this user input String using the same conditional logic used in the previous method for scanning the textfile. Except this time the ‘ **\n** ‘ condition is removed and the edge object is initialized straight away before even scanning the characters in the input String.

2.4 Searching for All Possible Paths

searchAll() is called in the main passing knowledgeBase and query as arguments. The return value is all the possible paths posed by the query on the knowledge base and it is stored in the variable **pathObjList** which after has its paths printed with a for loop. The **searchAll()** method is just used as a starting point for initializing all the variables to be passed through the recursive method **_searchAll()**.

The method **_searchAll()** works by going through all the nodes in the knowledge base starting from the starting node defined by the first node nodeA from the query until it reaches endNode defined by nodeB from the query. It uses the variable currNode to set as the current Node, when it sees currNode is equal to the endNode it saves the path as successful and appends it to pathObjList.

The program finds all possible permutations for a successful path by dynamically adding edges to

tempPath when going in a depth and deleting edges from tempPath when going back up a depth. Flag logic is also used to accurately deduce whether the path has a **True** or a **False** type.

2.5 Preferred by Shortest Distance

In the main the program calls **shortestPath()** by passing all the list of all possible paths **pathObjList**, it then returns all the shortest path objects in a list and stores it to **shortestPathList.** The contents of **shortestPathList** are printed out using a for loop.

shortestPath() works by calling a quicksort algorithm implementation **sortByLength()**, passing pathObjList as an argument. **sortByLength()** works recursively and sorts path objects by their length which is stored in the object itself as variable self.len. The sorted paths in ascending order are returned and stored to the list **sortedPaths.**

The first path in **sortedPaths** (which is the shortest path) is appended to the initialized list **shortestPaths**. A for loop is then used to store any other paths at the same length as the first shortest path (making them also the shortest path). **shortestPaths** is then returned.

2.6 Preferred by Inferential Distance

In the main the program calls **inferentialPath()** by passing all the list of all possible paths **pathObjList**, it then returns all the inferential path objects that survived through the algorithm and stores it in list **infPathList.** The contents of **infPathList** are printed out using a for loop.

inferentialPath() works by firstly copying all the possible paths from **pathObjList** to a new variable **infPaths**. The method goes through two phases. The first one is the redundancy check. Here all the redundant paths are removed from **infPaths.** After this the second phase is the preemption check. In this check all the preempted paths are removed from **infPaths**. At the end of both of these checks the list **infPaths** is then returned with its surviving path objects.

Redundancy Check

In order to find the redundant paths to eliminate, the program, using **searchAll(),** queries for all possible sub-paths from the beginning of each path to the next node after it. This next node nodeB keeps on incrementing by each node, checking for longer subPaths each time until the endNode of the path is reached (or the path stops existing).

The path stops existing when a sub-path is found to have another more informative alternative, deeming the current path as redundant. Information is measured using **shortestPath()**. When more than one possible sub-path is found for the query the shortest sub-path is found and the path the shortest sub-path belongs to is found and is deleted.

Preemption Check

To find preempted paths to delete the program firstly finds paths with the type **False** (indicating it has IS-NOT-A at the end). Using **searchAll()** it then searches for all possible alternative paths from the node before the IS-NOT-A to the node after the IS-NOT-A. When alternative sub paths with type **True** are found it then removes the main paths that they belong to as they are deemed as preempted.

2.7 Tested Data

Test 1: In the textfile "clauseStatements.txt" (notice that the first line is blank)

```
<Clyde> IS-A <FatRoyalElephant>
<FatRoyalElephant> IS-A <RoyalElephant>
<Clyde> IS-A <Elephant>
<RoyalElephant> IS-A <Elephant>
<RoyalElephant> IS-NOT-A <Gray>
<Elephant> IS-A <Gray>
```

Output for query "<Clyde> IS-A <Gray>"

```
All possible paths:
-------
Clyde IS-A FatRoyalElephant IS-A RoyalElephant IS-A Elephant IS-A Gray
Clyde IS-A FatRoyalElephant IS-A RoyalElephant IS-NOT-A Gray
Clyde IS-A Elephant IS-A Gray
```

Preferred by shortest distance metric:	Preferred by inferential distance metric:
-------	-------
Clyde IS-A Elephant IS-A Gray	Clyde IS-A FatRoyalElephant IS-A RoyalElephant IS-NOT-A Gray

Output for query "<Clyde> IS-A <Elephant>"

```
All possible paths:
-------
Clyde IS-A FatRoyalElephant IS-A RoyalElephant IS-A Elephant
Clyde IS-A Elephant
```

Preferred by shortest distance metric:	Preferred by inferential distance metric:
-------	-------
Clyde IS-A Elephant	Clyde IS-A FatRoyalElephant IS-A RoyalElephant IS-A

	Elephant

Test 2: In the textfile “clauseStatements.txt” (added other redundant edge at the end)

```
<Clyde> IS-A <FatRoyalElephant>
<FatRoyalElephant> IS-A <RoyalElephant>
<Clyde> IS-A <Elephant>
<RoyalElephant> IS-A <Elephant>
<RoyalElephant> IS-NOT-A <Gray>
<Elephant> IS-A <Gray>
<Clyde> IS-A <RoyalElephant>
```

Output for query “<Clyde> IS-A <Gray>”

```
All possible paths:
-------
Clyde IS-A FatRoyalElephant IS-A RoyalElephant IS-A Elephant IS-A Gray
Clyde IS-A FatRoyalElephant IS-A RoyalElephant IS-NOT-A Gray
Clyde IS-A Elephant IS-A Gray
Clyde IS-A RoyalElephant IS-A Elephant IS-A Gray
Clyde IS-A RoyalElephant IS-NOT-A Gray
```

Preferred by shortest distance metric:	Preferred by inferential distance metric:
-------	-------
Clyde IS-A Elephant IS-A Gray Clyde IS-A RoyalElephant IS-NOT-A Gray	Clyde IS-A FatRoyalElephant IS-A RoyalElephant IS-NOT-A Gray

2.8 Minor Limitations

- Program crashes when a node that doesn’t exist in the knowledge base is inputted as a query. This could be solved by validating the input query and recognising when a requested node doesn’t exist in the knowledge base, then returning an error to the user and stopping the program from proceeding to calculate all possible paths and preferred paths.

- When the ‘<’ or ‘>’ tags aren’t used in user input the program isn’t able to recognize and construct nodes for the inheritance network. An alternative would be to redesign the logic in parsing entirely.