Knowledge Representation and Reasoning Assignment

by Jesmar Farrugia
0237196M
CIS1019

Task 1:

Parsing of horn clauses, and reasoning using back-chaining.

Solution:

A single class was used, which holds an array of ‘clauses’. Clauses are just an array of strings entered by the user. The user enters a clause, made up of strings, separated by ‘, ’ and using ‘!’ to indicate a negative literal. The method ‘add()’ is then used to append the clause to the array of clauses within the class KnowledgeBase, using a simple append function available in python.

The method backchain() takes two parameters: the knowledge base itself and the query (which is a clause). As a base case, if the query is an empty list then the method returns True, otherwise it checks if the query contains one literal or multiple. If the query has one literal, its polarity is switched (positive to negative or vice versa) by searching the string itself for a ‘!’, if it exists, it’s removed, if not then it is added. Then the method uses a two loops, one the size of the knowledge base and the other the size of each clause, to compare literals. Once a match is found, the clause is put into a temporary array, and the query is removed from it, then the method is called once again recursively, using the temporary array as the new query. If the query is not found within the knowledge base then it returns False.

If the query has more than one literal, then a loop the size of the query is used to recursively call backchain() for every literal within the query, and if any of them returns False, then an if statement terminates the loop and returns a False. Else if none of them return False then the method returns True.

Basically (as a dry run), the user queries a clause with three elements: [!boy, !child, !woman]. Since there is more than one literal the method first take ‘!boy’ as a query, switched the query to ‘boy’ and searches the knowledge base. It finds the clause [!male, boy], removes boy from array and searches for ‘!male’, finds [male], gets [] and returns true. So the loop continues, and now checks ‘!child’. Method finds [!adult, !girl, child], so once again ‘child’ is removed and the method checks whether both ‘!adult’ and ‘!girl’ holds true. The method searches for ‘adult’ but does not find a clause, so it breaks out of the loop and returns False. So the query ‘!child’ returns False and the loop is broken again and return False. The output should return the words “Not Solved” and “KB does not |= [!boy, !child, !woman]”.

Output Listing:

Testing was made assuming that all inputs are correct.

Input (Terminal)	Output
python Task1.py	Enter clauses line by line. Use ! for negation and type end (case sensitive) to close the knowledge base. Make sure that literals are separated with a , Example: !Mammal, Human Enter a clause in knowledge base
Toddler	Enter a clause in knowledge base
!Toddler, Child	Enter a clause in knowledge base
!Child, !Male, Boy	Enter a clause in knowledge base
!Infant, Child	Enter a clause in knowledge base
!Child, !Female, Girl	Enter a clause in knowledge base
Female	Enter a clause in knowledge base
Male	Enter a clause in knowledge base
end	Enter your negated (!) query
!Girl	Found ['Girl'] in ['!Child', '!Female', 'Girl'] Found ['Child'] in ['!Toddler', 'Child'] Found ['Toddler'] in ['Toddler'] Found ['Female'] in ['Female'] SOLVED KB |= ['Girl']

Limitations:

-Assumes all inputs are entered correctly, thus any data can be entered (no exception handling).

-Requires user to input knowledge base line by line.

-Not user friendly.

Task 2:

Construction and reasoning with inheritance networks.

Solution:

Two classes were used, one for statements and the other for the inheritance network. The class Statements hold three variables, two strings, one for the sub-concept and the other for the super-concept, and a boolean variable which is True when the statement has a 'IS-A' and is False when it has a 'IS-NOT-A'. The class InherNet contains an array of statements.

Statements has one method: create(), which takes an array of three strings, takes the first string as the sub-concept, the third as the super-concept and the second is passed through an if statement the change the polarity to True or False.

On th other hand, the class InherNet has nine methods: add(), _buildpaths(), _splitpaths(), _polpaths(), _redunpaths(), _dispaths(), _shorthpath(), _inferpath(), buildpath(). All paths that start with an '_' are private methods to be used only by the class.

The first method, add(), takes an array of three strings as a parameter, initializes a variable as a Statement, uses the method create(), passing the same array as a parameter and then appends the statement.

_buildpaths() has three parameters: a string, the query's super-concept (string), and an empty array. As a base case, if the first string parameter is the same as the query's subconcept then recursion is stopped (returns nothing). If not, then it searches through all sub-concepts within the inheritance network to find the query, once found, the method appends the whole statement to the empty array and recursively calls the method again using the statement's super-concept as the first parameter, (the second and third parameter stay constant). The method does not return anything but once the method ends, the empty array should contain all possible paths as a single array.

The array obtained from the previous method, if more than one path is present, will not make sense on its own. Example of what will be present in said array: [[Man IS-A Human],[Human IS-A Mammal], [Mammal IS-NOT-A Bipedal], [Human IS-A Bipedal]]. So the next two methods were made to make sense of such an array.

The method _splitpaths(), takes two parameters: the goal (query's super-concept) and the array mentioned before. The method goes through the entire array until it finds a super-concept that is equal to the goal. Once found, appends all previous statements to a temporary array, and removes all statements appended from starting array. This will go on until the starting array is empty, then it will return the temporary array with all the separate as a single element of an array (2d).Once the 2d array is obtained, _polpaths()'s job is to add the missing statements (if any) to the incomplete paths.

_polpaths(), has two parameters: the previous 2d array, and the query's sub-concept. The method goes through the array and stops once the first statement's sub-concept does not match the query's sub-concept. The method then matches the incomplete path with the previous path, adds the missing statements to to a temporary array, adds the incomplete path to the same array, and then replaces the incomplete path with the temporary array. So now all that's left is to eliminate redundant paths, find the shortest path(s) and the inferential path(s).

The next method: _redunpath(), uses two parameters, one is the array of (now complete) paths, and the goal (query's super-concept). The method searches through the array, and if it find a path that has a statement where its polarity is false and its super-concept is not the goal, removes the entire path from the array.

_shortpath() takes the array of paths as a parameter. The method compares the size of each path with each other, every time storing the shortest path in a temporary array. Once the shortest path is found another loop through the array of paths is done to check if there is more than one short path, and if so, it appends them to the temporary array. Finally the temporary array is returned.

_inferpath() takes the same parameter and uses the same algorithm as the previous method, at first, to find the longest (most detailed) path, and then uses that path to find other paths that contradict it. The method takes the last statement of the longest path, goes through all other paths in the network, and compares it with their last statement. If the polarities are different, the path is appended to a temporary array, to be returned at the end of the method.

To be able to display paths in an orderly fashion, _dispaths() was used. The method takes an array of paths as a parameter. It runs a loop the size of the array, and a nested loop the size of the path. If it's the first statement, it prints the sub-concept, 'IS-A' or 'IS-NOT-A' depending on the polarity, and the super-concept, otherwise it will only print the polarity and the super-concept.

Finally the method buildpaths() is used to simplify input, where it just takes the user's query statement and calls all other eight methods accordingly.

Output Listing:

Input (Terminal)	Output
python Task2.py	Type end to close the inheritance network. Use space to separate every concept and use IS-A and IS-NOT-A (case sensitive) Input example: Penguin IS-A Bird Enter statement
Clyde IS-A FatRoyalElephant	Enter statement
FatRoyalElephant IS-A RoyalElephant	Enter statement
Clyde IS-A Elephant	Enter statement
RoyalElephant IS-A Elephant	Enter statement
RoyalElephant IS-NOT-A Gray	Enter statement
Elephant IS-A Gray	Enter statement
end	Enter query
Clyde IS-A Gray	All 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 Shortest paths: Clyde IS-A Elephant IS-A Gray Inferential paths: Clyde IS-A FatRoyalElephant IS-A RoyalElephant IS-NOT-A Gray

Limitations:

-In limitations similar to the task before.

-If there are more than one longest path, the inferential path will most likely be incorrect.