Knowledge Representation and Reasoning

2017/2018

Name: Valerija Holomjova

Course Code: ICS1019-SEM2-A-1718

Table of Contents

Introduction

Both parts of the assignment were entirely completed and are fully functional. It is important to note that the user input has to be entered as specified in the description of each part as the results will not display otherwise. This is a bug I have not been able to fix and is the only problem that has been encountered when testing both parts of the assignment. The algorithms have been coded using JavaScript. HTML has also been used in the source code in order to acquire user input and apply the results.

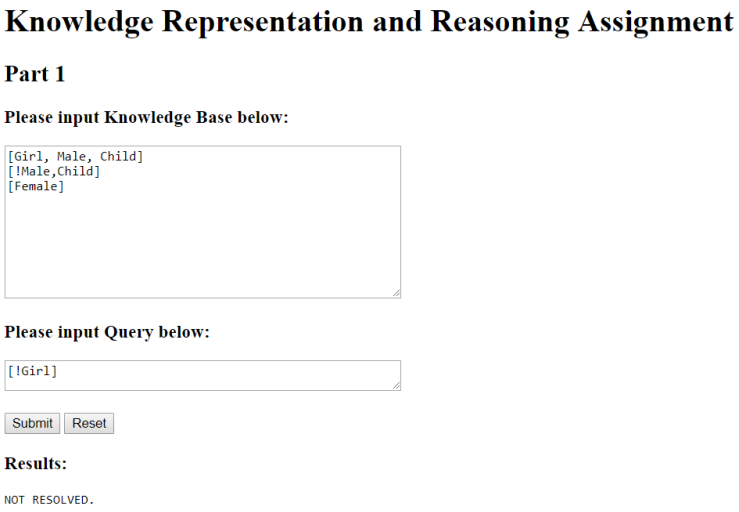
Part 1 – Parsing of Horn Clauses, and Reasoning using Back-Chaining

Description of User Interface

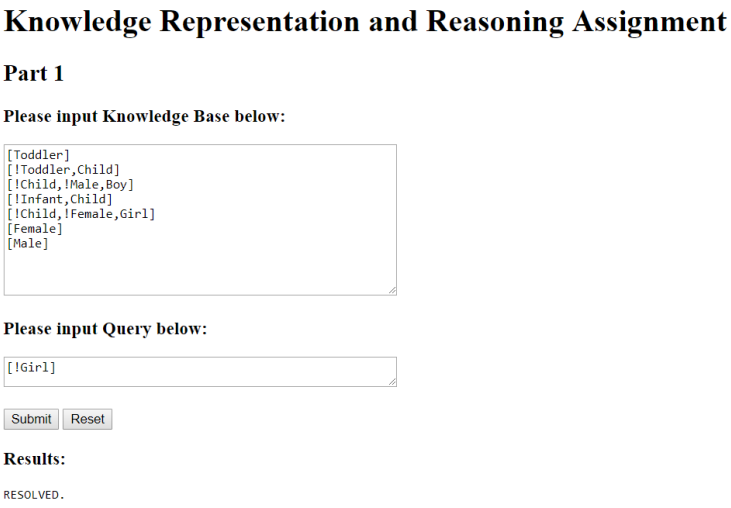
In order to use the algorithm, the user must input their knowledge base clauses in the first textbox and their query in the second text box. The knowledge base clauses must start and end with a square bracket and be separated from each other by a new line. Each literal must be separated by a comma and the '!' symbol has been used instead of the ' ¬' symbol to indicate a negative literal. Next, the user has to click on the 'Submit' button which will then display whether the query has been 'RESOLVED' or 'NOT RESOLVED' under the 'Results:' section. Please note that the 'Reset' button must be pressed if no results are displayed so that the algorithm can be used again. If no results are displayed, it could be for the reasons mentioned below.

It is important to note that if spaces are inputted in between the literals of a clause, or an extra new line is inputted at the end of the last clause, the algorithm could bug and not produce any results. Unfortunately, I have attempted many solutions to solve this problem but have not found a fix. It must be mentioned that I have also not found a solution against infinite loops. In other words, the algorithm will keep going until the query clause is empty or cannot find a clause in the knowledge base with opposing literals. If such a loop is encountered, no results will be displayed.

Please refer to the screenshots below, for an example of how the code has been tested.



(Example of Query Not being Solved)



(Example of Query being Solved)

User Input

The algorithm starts by reading the users input from the HTML form. The knowledge base clauses and query are stored in temporary string variables exactly as they were inputted.

For the user's inputted knowledge base clauses, an array called 'clauses' is defined and this will represent the entire knowledge base. It will consist of an array of clauses where each clause is an array of objects representing the literals. The user's inputted clauses are split using '\n' as a delimiter and stored in a temporary array of clauses. Subsequently, each clause is accessed and the square brackets are removed from the start and end of each clause. Each clause is then split using ',' as a delimiter and stored in a temporary array of literals. Each literal of the current clause is then accessed, and checked for whether it contains the '!' symbol. If the literal contains the '!' symbol, it will be considered as a literal with negative polarity. Thus, the '!' symbol will be removed from the literal string to be used as a name, and an object with the name and a negative polarity (false) will be pushed onto the current clauses array of literals. If the '!' symbol is not found, it will be considered a literal with positive polarity and pushed with the name of the literal and a positive polarity (true). This process repeats for every clause. In the end, each element of the 'clauses array' represents a clause which is an array of objects where each object is a literal with a name and polarity.

For the user's inputted query, the square brackets are first removed from the query string. An array called 'queryclause' is defined which will represent the query clause before and after resolution is performed. If the '!' symbol is found, the '!' is removed from the query string to be used as a name, and an object with the name and negative polarity is added onto to the query clause. Otherwise, it is added as a positive literal onto the query clause. Thus, before resolution, the 'queryclause' should be an array of just one literal as an object having a name and polarity which is the user's inputted query.

Resolution

The recursive function 'is_resolved' will attempt to resolve the query and will return true or false based on whether the resolution is successful. It starts by storing the number of literals in the query in an integer called 'n'. If the value of 'n' is 0, signifying that the query clause is empty, 'true' is

returned, signifying that the resolution was successful. This is the stopping condition. If the value of 'n' is not 0, each literal in the current query clause is accessed using a 'for' loop. A function called 'find_opp_literal' is used to check whether there is clause in the knowledge bases that has an opposite literal to the current one being checked. It does this by checking each clause in the knowledge base, and each literal, then checking if its name is identical but the polarity is not the same. If an opposing literal is found, the clause with the opposing literal is returned, otherwise null is returned.

If a clause with an opposing literal is found, the query clause will be modified using the function 'replace'. It then calls the function 'is_resolved' recursively and returns true if true is returned by the recursive function. If a clause with an opposing literal is not found, false is returned as the query clause can no longer be modified.

The function 'replace' handles the modification of the query clause. First it adds new literals to the query clause from the other clause. It does this by checking if there are any literal names in the other clause that the query clause does not have yet using the 'has_name' function. This simply scans each literal in the query clause and checks if it has an identical name with a given literal name of the other clause and return true or false based on whether it is found or not. If the query clause doesn't have the name of a literal in the other clause, it is pushed onto the query clause as an object with the correct polarity and name.

Next, the function removes repeated literals by accessing each literal of both the query clause and other clause with two 'for' loops. If a literal is found in the other clause with the same name but different polarity, it is removed from the query clause. If the query clause becomes empty, it is returned to avoid errors. After the 'for' loops terminate, the newly modified query clause is returned where new literals from the other clause have been added, and opposing literals have been removed.

Thus, if the 'is_resolved' function reaches an empty query clause, it will return true and 'RESOLVED' will be displayed under results. Otherwise, it will keep search for a solution until there the query clause has no opposing literals found. If there are no opposing literals found in the knowledge base, the function will return false and 'NOT RESOLVED' will be displayed under the results. It is important to note that infinite loops can be caused if a literal in the query clause can be constantly replaced with another from the knowledge base.

Part 2 – Construction and Reasoning with Inheritance Networks

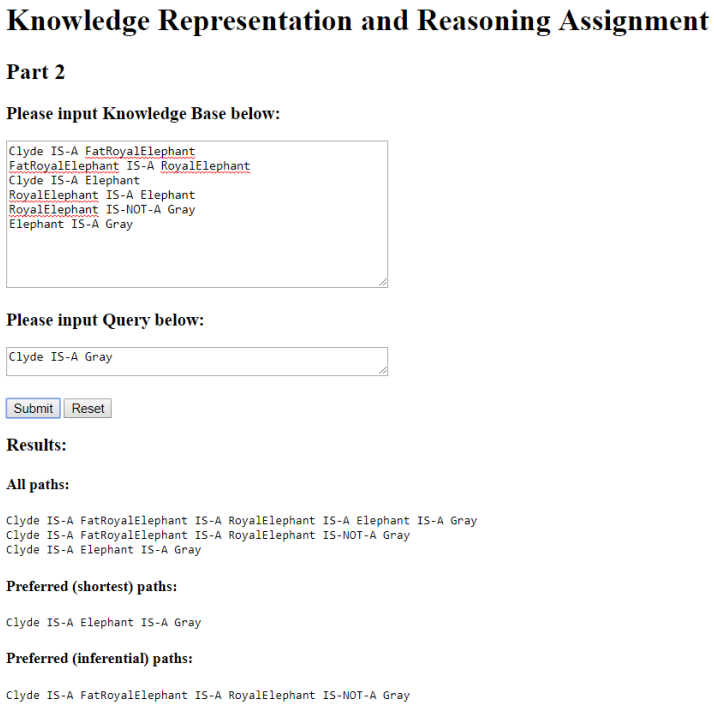
Description of User Interface

In order to use the algorithm, the user must input their inheritance network as a series of concepts in the first textbox and their query in the second text box. The concepts must be separated from each other by a new line and spaces should only be used to separate the sub-concept and super-concept from the link. Next, the user has to click on the 'Submit' button which will then display the different types of paths under each path section. Please note that the 'Reset' button must be pressed if no results are displayed so that the algorithm can be used again. If no results are displayed, it could be for the reasons mentioned below.

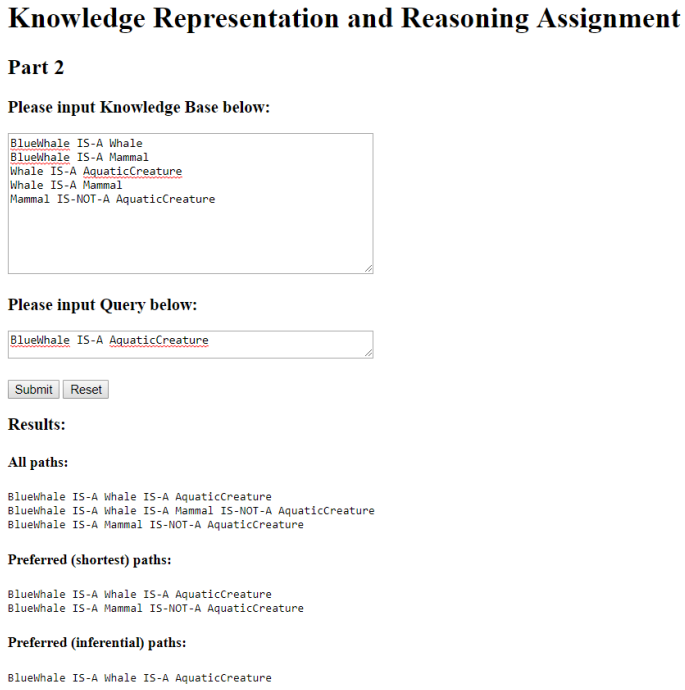
It is important to note that if excess spaces are inputted in the concepts, or an extra new line is inputted at the end of the last concept, the algorithm could bug and not produce any results.

Despite trying many solutions to solve this problem, a fix has not been found. If a new line is entered after the last clause, or excess spaces are entered within a clause, there could be no paths displayed.

Please refer to the screenshots below, for an example of how the code has been tested.



(Example of Part 2)



(Another Example of Part 2)

User Input

The algorithm starts by reading the users input from the HTML form. The concepts and query are stored in temporary string variables exactly as they were inputted.

First, the user's concepts are split using '\n' as a delimiter and stored in array called 'concepts'. Next, an array called 'nodes' is declared which will store a list of objects. These objects will represent nodes and have a name variable, as well as array of edges. Each edge is an object itself with three variables which are the names of the sub-concept, super-concept and the polarity of the edge. A 'for' loop is used to access each concept and the sub-concept is extracted from each concept by extracting the first token element using the space character as a delimiter. A function called 'contains node' checks whether the node name already exists in the 'nodes' array. If the node name already exists, a new edge is created using the 'new_edge' function and added into the nodes array of edges using the 'push_edge' function. If the node does not exist, a new node is created using the 'new_node' function and pushed into the 'nodes' array and the new edge is created using the 'new_edge' function and pushed into the nodes array of edges using the 'push_edge' function.

- The 'new_node' function is used to return an object with a given name as a variable and an empty array of edges.
- The 'push_edge' function finds the appropriate node by searching for a nodeby its name in the 'nodes' array. When the node is found, the edge is added onto its array of edges by using the '.push()' library function.
- The 'new_edge' function returns an edge object, by splitting the concept using the space character as a delimiter, then extracting the first token as the sub-concept, the second as the relation and the third as a super-concept. If the relation includes the word 'NOT' in it, it will set the polarity of the edge to 0 (negative). Elsewise, the polarity of the edge is 1 (positive). The function will then return an object with the sub-concept, super-concept and polarity as object variables.

The user's inputted query is split using the space character as a delimiter. Then, the sub-concept and super-concept is extracted by taking the first and third token of the split. The sub-concept will represent the start and the super-concept will represent the end of the paths to be found.

Next, the algorithm uses three functions to determine all the possible paths, all the shortest paths, and all the inferential paths. These are displayed in HTML in their appropriate categories. 'For' loops are used so that multiple paths can be displayed. Now the report will proceed by describing each path finding function separately.

Finding all Possible Paths

The global variable 'paths' will be used to store an array of paths where each path is an array of strings. These paths are found using the 'find_paths' function and works like the following. The function is recursive and has three arguments:

- 'start' represents the current node and does not remain fixed.
- 'end' represents the last node that has to be reached and remains fixed.
- 'path' represents the string holding the current path, and does not remain fixed.

The function starts by setting the 'current_node' and 'current_edges' by locating the information of the start node in the inheritance network. This is done by checking the node's array for a node that

has the same name as the start node and then simply setting the variables ‘current_node’ to the node in the ‘nodes’ array and ‘current_edges’ to the nodes array of edges.

If the current node has no edges, the function will not execute any commands and return with the ‘path’ variable unchanged. Thus, if a dead end is ever encountered while searching for paths, the function will backtrack and try another edge. Elsewise, if the current node has edges, each edge in the current node will be accessed. Next, the ‘path’ variable is duplicated to prevent it from being changed, and the variable ‘next_node’ is set by accessing the super-node object variable of the current edge. Subsequently, an ‘if’ statement checks whether the ‘next_node’ variable is the same as the ‘end’ variable. If this statement is true, the current node is added to the new path. Then the relation is added, ‘IS-NOT-A’ or ‘IS-A’ depending on whether the edges polarity is negative or positive respectively. The ‘next_node’ is added to the ‘new_path’ as well. The ‘new_path’ is now a possible path represented as an array of strings and is added to the ‘paths’ array as a possible path. The algorithm will proceed by checking the next edge of the current node.

It is important to note that if a negative edge was encountered and the ‘next_node’ is not the ‘end’ node, then the algorithm will not execute any commands and continue by searching the next edge of the current node.

If neither of these statements are satisfied then the current node and relation will be added onto the current path as strings and the function will recursively call itself. This function will stop once every possible path from the start node is checked to see whether it satisfies the query.

Finding all Shortest Paths

The function ‘find_shortest_path’ returns an array of the shortest paths where each path is an array strings. The variable ‘smaller’ keeps track of the smallest path in the ‘paths’ array. An array called ‘found_paths’ is defined and will store the shortest paths and return them at the end of the function. Initially, the smallest path is set to the first path in the path’s array. Next, the function accesses each path in ‘paths’ array and updates the ‘smaller’ variable if a path is shorter than the current smallest path. The length of the path is determined by ‘.length()’ function since each path is an array of strings, hence, the ‘.length’ function will return the number of elements (or strings) in each path. After the shortest path is found, the function proceeds by adding all paths with the same length size to the ‘found_paths’ array to ensure that multiple paths are considered. The ‘found_paths’ array is the returned as an array of the shortest paths.

Finding all Inferential Paths

The function ‘find_inferential_path’ returns an array containing all the inferential paths found where each path is an array strings. An array called ‘found_paths’ is defined and will store the inferential paths and then return them. A ‘for’ loop is used to access each path in the ‘paths’ array which are then passed through the ‘is_redundant’ and ‘is_preempted’ functions. If both functions return false for the given path, the path is considered inferential and is added to the ‘found_paths’ array. The array is returned after each path is checked in the ‘paths’ array.

The ‘is_preempted’ function returns true or false depending on whether a given path is pre-empted or not respectively. The function was implemented based on the statement that an edge x->y is pre-empted if there exits another path a->v->x->y and a->v->¬y. It starts by accessing each path in the

‘paths’ array and an if statement ensures that the same path isn’t being considered. It then uses an index ‘j’ to traverse through every node in the given path (since it increments by two) until the end of the given path. When the nodes of the given path and accessed path are no longer the same, the pre-emptor node is found ‘v’ by being the last node in which the paths were the same. The function checks whether the endings of both paths are the same and whether the final links are of opposite polarity. It then checks if the node where both paths are no longer the same equivalent to the other paths end node. If these statements are true, the given path is considered to have pre-empted edges, and returns true. If all of these conditions are not true, it will search the next path in the ‘paths’ array until true is returned or until all paths have been searched and false is then returned.

The ‘is_redundant’ function returns true or false depending on whether a given path is redundant or not respectively. The function was implemented based on the statement that an edge x->y is redundant if we another path x->…->y. It starts by accessing each path in the ‘paths’ array and ensures that the same path as the given path isn’t considered in the function. Subsequently, copies are created of both paths and an index is used to access the nodes of both paths (since it increments by 2). When the nodes of both path are no longer the same, all previous elements of both paths are removed and an if statement checks whether the entire remainder of the given path can be found in the rest of the accessed path. If this condition is true then the function returns ‘true’ indicating that the given path is redundant. Elsewise, the function will keep searching all the paths until ‘true’ is returned or all paths have been searched and then ‘false’ is returned.