ICS1019 - Assignment

Daniel Magro
484497M

May 2017

Contents

Question 1

1.1 Code Explanation

The data structures used for storing the Knowledge Base were the following: The Knowledge Base was stored in an Array List (dynamic sized array which grows/shrinks as needed) of Horn Clauses.
A Horn Clause class was created, which includes an Array List of Literals.
A Literal class was created, which includes a String storing the value of the Literal, and a boolean value storing the polarity of the Literal (i.e. whether it is positive or negative).

When the program is launched, it first declares an Array List of Horn Clauses, which will serve as the Knowledge Base, and initializes it to the output of the **parseInput** method, to which the program arguments (the name of the text file the knowledge base will be read from) supplied by the user are passed.

The **parseInput** method first declares an array of Strings, where each String will store a single line (CNF horn clause) from the input text file, and sets the value of each string/reads each line using the **readInput** method.
Next, an Array List of Horn Clauses is declared (the knowledge base).
A loop then goes through every element in the array of Strings, textFile[ ], and parses each line as a Horn Clause using the **parseQuery** method and passing the String holding the line to be parsed.
After every line is parsed as a Horn Clause, it is added to the Knowledge Base, the Knowledge Base is returned, and execution of the main method continues.

The **parseQuery** method receives a line taken from the input text file/-knowledge base as an argument, and returns a Horn Clause.

First, a Horn Clause is declared to store the Horn Clause which will be parsed.
Then, the passed 'line' is processed by the method to turn it into a Horn Clause object.
First, the **'['** and **']'** are removed from the beginning and the end of each horn clause and stored in a String **cnf**.
Next, the String **cnf** is split by every '**,**' and each sub-string is stored in an array of Strings **literalsInClause**[ ]
Then, the program loops over every string in the array **literalsInClause**[ ], and for each string does the following:
First, a new Literal is created which will store each Literal inside the Horn Clause.
Then, the value of the Literal is set to the current sub-string (which was split by '**,**'), minus the whitespaces.
Next, the program checks whether the first character of that Literal is a - (negation) or not. If it is, it sets the polarity of that literal to **false** and the value of that literal to what it was, minus the first character. If it isn't, then it sets the polarity of that literal to **true**.
Finally, each literal is added to the current horn clause.
After the parser has gone through every Literal, it returns the Horn Clause.

When the program has finally parsed the entire inputted Knowledge Base, it then outputs it to the user so that it can be checked that it is indeed the intended Knowledge Base.

After that, the program asks the user to input the Negated Query he/she would like to have resolved with the Knowledge Base.
Similarly to what was done earlier, the Negated Query is passed to the **parseQuery** method so that it can be stored as a Horn Clause.

The program finally calls the **resolve** method and passes the Knowledge Base and the Negated Query to be resolved as arguments.

Each time it is called, the **resolve** method checks whether the query to be resolved is empty, in which case it is considered **SOLVED** and the program stops execution.
If the query is not empty, the program goes into the following process of resolution:
The outer-most loop goes through the literals in the query to be resolved.

The loop inside of that goes through the horn clauses in the knowledge base. The loop inside of that goes through the literals inside the current hornclause. Inside the nested loops, the program checks each literal in the query with every other literal inside every other horn clause for a matching value and an opposing polarity.
Once such a match is found, the literal that has been matched is removed from the query, then all the literals (except the one that has been matched) from the horn clause inside the knowledge base are added to the query being resolved.
After this is done, a recursive call happens to the same method, passing the same knowledge base and the updated query.

The recursive calls keep happening either until the base case is reached, i.e. an empty query([ ]) and thus being SOLVED so the program can terminate, or until all literals inside the query have been checked with every other literal in the knowledge base, in which case the program outputs NOT SOLVED and terminates.

1.2 How to Run the Program

1) Open the folder **KRR1_jar**
2) Enter your Knowledge Base inside **input.txt**
NOTE: Instead of the symbol '¬' for negation, please use '**-**', Both in the Knowledge Base (input.txt) and the query inside the program.
Clauses should be separated by new lines, and literals by commas, an example is already included in input.txt
3) Launch the program by double-clicking on **run.bat**
4) The knowledge base you inputted is displayed for verification. You can enter the query you wish to resolve and press enter to start resolution.
eg: [-x, -y, z]

1.3 Source Code Listing

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Scanner;
import static java.lang.System.exit;

public class Main {

    // Literal class - contains value String and polarity Boolean, indicating whether a literal is +ve or -ve
    public static class Literal {
        String value;
        boolean polarity;
    }

    // HornClause class - contains an Array List of Literals
    public static class HornClause {
        ArrayList<Literal> clause = new ArrayList<Literal>();
    }

    // Main Method
    public static void main(String[] args) throws IOException {
        // Declaring knowledge base as an Array List of Horn Clauses
        // And calling the method to parse the text file
        //ArrayList<HornClause> knowledgeBase = parseInput("input.txt");
```

```java
        ArrayList<HornClause> knowledgeBase = parseInput(args[0]);
        System.out.println("The inputted knowledge base is the following:");
        printKB(knowledgeBase);
        System.out.println("Please enter the negated query you would like to resolve with the Knowledge Base in CNF");
        Scanner scan = new Scanner(System.in);
        String queryInput = scan.nextLine();
        // The negated query inputted by the user is apssed to the parseQuery method to be parsed into
        // a Horn Clause (Array List of Literals) and stored in 'query'
        ArrayList<Literal> query = parseQuery(queryInput);
        System.out.print("Resolving with the Knowledge Base: "); printHC(query);
        resolve(knowledgeBase, query);
    }

    // File IO method, retrieves all the lines in the text file and stores each Clause
    // as an element in an array of Strings
    private static String[] readInput(String inputFile) throws IOException {
        String[] data = Files.readAllLines(Paths.get(inputFile)).toArray(new String[]{});
        return data;
    }

    private static ArrayList<HornClause> parseInput(String inputFile) throws IOException {
        // Calling method to retrieve input from the text file of clauses
        String textFile[] = readInput(inputFile);
        // Declaring kb
        ArrayList<HornClause> kb = new ArrayList<HornClause>();

        // Parser
        // for loop goes through each element in the array of unparsed clauses taken from the text file
        for (int i = 0; i < textFile.length; i++) {
            // Declaring horn clause
            HornClause hc = new HornClause();

            hc.clause = parseQuery(textFile[i]);
            kb.add(hc);
        }
```

```java
        return kb;
    }

    private static ArrayList<Literal> parseQuery(String queryText) {
        // Declaring horn clause
        HornClause hc = new HornClause();

        // removing [ and ]
        String cnf = queryText.replaceAll("\\[|\\]", "");
        // splitting by ,
        String[] literalsInClause = cnf.split(",");
        // looping over every literal in the cnf
        for (String s : literalsInClause) {
            // Declare literal
            Literal lit = new Literal();
            // remove white space
            lit.value = s.trim();
            // if the literal starts with -, set the polarity to false and remove the first character
            if (lit.value.charAt(0) == '-') {
                lit.polarity = false;
                lit.value = lit.value.substring(1);
            } else {
                lit.polarity = true;
            }
            hc.clause.add(lit);
        }

        return hc.clause;
    }

    private static void printKB(ArrayList<HornClause> knowledgeBase) {
        for (int i = 0; i < knowledgeBase.size(); i++) {
            System.out.print("[");
            for (int j = 0; j < knowledgeBase.get(i).clause.size(); j++) {
                if (knowledgeBase.get(i).clause.get(j).polarity)
                    System.out.print(knowledgeBase.get(i).clause.get(j).value);
                else System.out.print("-" + knowledgeBase.get(i).clause.get(j).value);
                if(j+1 != knowledgeBase.get(i).clause.size()) {
                    System.out.print(", ");
                }
            }
```

```
            System.out.println("]");
        }
    }

    private static void printHC(ArrayList<Literal> hornClause) {
        System.out.print("[");
        for (int j = 0; j < hornClause.size(); j++) {
            if (hornClause.get(j).polarity)
                System.out.print(hornClause.get(j).value);
            else System.out.print("-" + hornClause.get(j).value);
            if(j+1 != hornClause.size()){
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }

    // method to resolve a query and the knowledge base to the empty clause
    private static void resolve(ArrayList<HornClause> knowledgeBase, ArrayList<Literal> query) {
        // if a query is empty, this means that it has been SOLVED
        if (query.size() == 0) {
            System.out.println("SOLVED");
            exit(0);
        }

        // traversing kb for a matching literal with opposite polarity
        for (int i = 0; i < query.size(); i++) {
            for (int j = 0; j < knowledgeBase.size(); j++) {
                for (int k = 0; k < knowledgeBase.get(j).clause.size(); k++) {
                    if (knowledgeBase.get(j).clause.get(k).value.equals(query.get(i).value)
                            && knowledgeBase.get(j).clause.get(k).polarity != query.get(i).polarity) {
                        System.out.print("Resolving with: ");
                        printHC(knowledgeBase.get(j).clause);
                        // remove the matched literal from the query
                        query.remove(i);
                        // add the literals inside the clause the query was resolved with
                        // (except the resolvent of the original query) to the query being resolved
```

```
                                for (int l = 0; l < knowledgeBase.get(j).clause.size(); l++) {
                                    if (l != k) {
                                        query.add(knowledgeBase.get(j).clause.get(l));
                                    }
                                }
                                System.out.print("Query: ");
                                printHC(query);
                                resolve(knowledgeBase, query);
                            }
                        }
                    }
                }
                System.out.println("NOT SOLVED");
                exit(0);
            }
}
```

Question 2

2.1 Code Explanation

The data structures used for storing the Inheritance Network were the following:
The Inheritance Network (Graph) was stored in an Array List (dynamic sized array which grows/shrinks as needed) of Concepts (Nodes).
A Node class was created, which includes a String storing the name of the Concept/Node and an Array List of Edges (Links) that start from that node.
An Edge class was created, which includes 2 Nodes storing the start node and the end node of the link (start and end are explicit since it is a directed edge) and a boolean value storing the polarity of the link, where true implies IS-A and false implies IS-NOT-A.
A Path class was also created, which includes an ArrayList of Nodes, and a boolean value storing polarity, i.e. whether the last link in the path is an IS-A or IS-NOT-A link (true and false, respectively).

When the program is launched, it first declares a new Graph, which will serve as the Inheritance Network, and initializes it to the output of the **parseInput** method, to which the program arguments (the name of the text file the knowledge base will be read from) supplied by the user are passed.

The **parseInput** method first declares an array of Strings, where each String will store a single line (subConcept IS-A/IS-NOT-A superConcept) from the input text file, and sets the value of each string/reads each line using the **readInput** method.
Next, a Graph is declared (the inheritance network).
A loop then goes through every element in the array of Strings, textFile[ ], and parses each line as a two Nodes and an Edge. In order to parse a line, the

program first calls the **checkExistence** method in order to check whether the subConcept is already in the Array List of Nodes, in order not to create a new, duplicate node. The same is done for the superConcept. A new Edge is declared and is initialised with the subConcept (start), superConcept (end) and the polarity (true=IS-A or false=IS-NOT-A). When that is done, the Edge is added to the Array List of Edges held inside the subConcept.
After every line is parsed, the Inheritance Network is returned, as a Graph, and execution of the main method continues.

Next, the Inheritance Network which has just been parsed is printed out to the user using the **printIN** method. This is done so that the user can verify that the Inheritance Network has been parsed correctly and that the correct IN was saved inside input.txt.

Next, the user is prompted to enter a Query. The query is split by every white space and the first word is the subConcept and the third is the superConcept (the second word is ignored since it will always be 'IS-A').

Once the program has parsed the query entered by the user, it can start looking for all the paths between the subConcept and the superConcept. An Array List of Paths is created, which will store all the paths that are found. A Path is created named visited, which will keep tracks of which nodes the DFS has visited, and the subConcept is added to the start of the path. The **findPaths** method is called, passing the Inheritance Network, subConcept, superConcept, visited Path and resultant array of Paths as arguments. When the method completes, all the Paths are printed out.

The **findPaths** method is a modified recursive Depth First Search.
The Base Case can only be reached when the current Node (start) equals the end Node and all the links in the path are positive. If these conditions are met, the polarity of the path is set to true, and the Path (visited) is added to the Array List of Paths. After this is done the program returns execution to wherever it was called from.
The General Case does the following each time it is accessed:
First, an Array List of Nodes, adjacentNodes, is declared, which as the name implies, will store all of the Nodes which are adjacent to the current (start) Node. A similar Array List of Boolean is declared, which corresponds to adjacentNodes, which will store the polarity of every link between the current Node and every adjacent Node. A for loop initialises these 2 array lists.
Then, for every adjacent Node, the program does the following:
If an adjacent Node is already in the list of visited Nodes, then it is skipped.

A new temporary Path is declared. If the Edge is an 'IS-NOT-A' edge, we check if we have reached the destination (since an IS-NOT-A edge can only be at the end of a path). If we have reached the destination, we add the list of visited Nodes to the temp path, we add the last Node (current adjacent Node) to the path too, the polarity of the path is set to false (meaning there is an IS-NOT-A edge at the end) and the path is stored inside the Array List of Paths, paths. If the destination was not reached, the program goes onto the next adjacent node.
If the Edge is an 'IS-A' edge, we add the visited Nodes to the temp path, we add the current adjacent Node to the temp path, and we make a recursive call, this time with the start Node set to the current adjacent Node.
After all Paths have been found, execution returns to the main method, and all the paths found are printed out using the **printPaths** method (which takes an Array List of Paths as an argument).

After the program has all the Paths between the subConcept and superConcept as queried by the user, it calls the **shortestDistance** method and passes the Array List of Paths as an argument.
The method **shortestDistance** first goes through every Path and determines what the shortest Path Distance is. Then another loop goes through every Path and adds those Paths that have a length equal to the minimum path distance to an Array List of Paths, shortestPaths. The method then prints the Array List shortestPaths. Execution is then returned to the main method

Finally, the **inferentialDistance** method is called, which will find all the admissible Paths and print them out. The Inheritance Network graph, superConcept Node and the Array List of all the Paths are passed as arguments.
The **inferentialDistance** accomplishes 2 things, it looks for **redundant** and **pre empted** edges in Paths, and filters them out since they make the Path inadmissible.
First the program looks for preempted edges in each path, and if they are found, the path is considered inadmissible. First a boolean value named preempted is declared, which will help optimise the program, by skipping over certain checks if a path is deemed inadmissible in an earlier iteration. A for loop goes through every Path that has a positive polarity (i.e. ends with an IS-A edge) within the inferentialPaths ArrayList, and another loop goes through every Node inside each Path inside the Array List. The method **findPaths** is called, with the starting point set to the preceding node, and the end node set as the superConcept of the current Node. When all the Paths are found, it is checked whether any one of them has a negative edge to the superConcept. If so that path is declared pre empted. When a Path is

declared pre empted, it is removed from the Array List of admissible paths, inferentialPaths, the counter is decremented to adjust for the shifting of the elements of the array, and the inner loop breaks as an optimisation.
Next, the program looks for redundant edges in each path, and if they are found, the path is considered inadmissible. First a boolean value named redundant is declared, which will help optimise the program, by skipping over certain checks if a path is deemed inadmissible in an earlier iteration. A for loop goes through every Path within the inferentialPaths ArrayList, and another loop goes through every Node inside each Path inside the Array List. The method **findPaths** is called, with the starting point set to the current node, and the end node set to the successive node. When all the Paths are found, another loop goes through each path found, and finds the highest path length of all the positive paths (i.e. paths that end with IS-A). If the max path length is greater than 1, then a longer, more detailed path was found so the edge can be declared redundant. Due to this, the path is removed from the Array List inferentialPaths, the coutner, i, is adjusted, and the inner loop breaks so the next path is checked. When all inadmissible paths have been filtered out from inferentialDistance (Array List of admissible paths) the Array List is printed out using the 'printPaths' method.

2.2 Problematic Feature:

The Inferential Distance does not function correctly. This is because findPaths is being used to find any path/number of edges which can make an edge redundant/pre empted, and findPaths does not factor out paths which have already been deemed inadmissible. If not for time constraints, this could be easily fixed by changing 'findPaths' for a more specific method which looks for sub-paths based only on the paths which are still deemed admissible (i.e. still inside the inferentialDistance Array List of Paths).

2.3 How to Run the Program

1) Open the folder **KRR2_jar**
2) Enter your Inheritance Network inside **input.txt**
Links should be separated by new lines, an example is already included in input.txt
3) Launch the program by double-clicking on **run.bat**
4) The Inheritance Network you inputted is displayed for verification. You can enter the query you wish to resolve and press enter to start resolution.
eg: Clyde IS-A Gray

2.4 Source Code Listing

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.Scanner;

public class Main {

    public static class Node {
        String name;
        ArrayList<Edge> connections = new ArrayList<Edge>();
    }

    public static class Edge {
        Node start;
        Node end;
        boolean polarity;
    }

    public static class Graph {
        ArrayList<Node> nodes = new ArrayList<Node>();
    }

    public static class Path {
        ArrayList<Node> path = new ArrayList<Node>();
        boolean polarity;
    }

    // Main Method
    public static void main(String[] args) throws IOException
    {
```

```java

        // Declaring inheritance network as a Graph - an array list of Nodes
        Graph IN = new Graph();
        // Calling method to parse inputted inheritance network
        IN = parseInput(args[0]);

        // printing the IN to make sure it was parsed correctly / user selected the correct IN
        System.out.println();   //inserting a new line
        printIN(IN);
        System.out.println();   //inserting a new line

        // User Input - Query
        System.out.println("Please enter your query:");
        Scanner scan = new Scanner(System.in);
        String query = scan.nextLine();
        // parsing the user's query:
        String[] queryParts = query.split(" ");
        String start = queryParts[0];
        String end = queryParts[2];

        // Find all paths from subConcept to superConcept

        // Find which nodes are the subConcept and the superConcept
        Node subConcept = new Node();
        for (int i = 0; i < IN.nodes.size(); i++) {
            if (IN.nodes.get(i).name.equals(start)) {
                subConcept = IN.nodes.get(i);
            }
        }
        Node superConcept = new Node();
        for (int i = 0; i < IN.nodes.size(); i++) {
            if (IN.nodes.get(i).name.equals(end)) {
                superConcept = IN.nodes.get(i);
            }
        }
        // ArrayList of paths which will store all the paths found
        ArrayList<Path> paths = new ArrayList<Path>();
        // Path object which will store all the visited nodes in an ArrayList of Nodes
        Path visited = new Path();
        // setting the first visited node to the starting node (subConcept)
        visited.path.add(subConcept);
```

```
        findPaths(IN, subConcept, superConcept, visited, paths);
        System.out.println("\nAll Paths found are:");
        printPaths(paths);

        // Shortest distance
        shortestDistance(paths);

        // Inferential distance
        inferentialDistance(IN, superConcept, paths);
    }

    // File IO method, retrieves all the lines in the text file and stores each connection
    // as an element in an array of Strings
    private static String[] readInput(String inputFile) throws IOException {
        String[] data = Files.readAllLines(Paths.get(inputFile)).toArray(new String[]{});
        return data;
    }

    private static Graph parseInput(String inputFile) throws IOException {
        String textFile[] = readInput(inputFile);

        Graph IN = new Graph();

        for (int i = 0; i < textFile.length; i++) {
            // splitting by whitespace
            String[] statementParts = textFile[i].split(" ");

            // declaring a new node since the current concept is not already in the list of nodes
            Node subConcept = new Node();
            // checking whether the sub-concept is already in the list of nodes
            if (!checkExistence(IN, statementParts[0])) {
                // setting the name of the node to the first part of the statement (the subConcept)
                subConcept.name = statementParts[0];
                // adding the node to the list of nodes
                IN.nodes.add(subConcept);
                /*
                // storing the current connection within the list of connections of that node
                IN.nodes.get( IN.nodes.size()-1 ).connections.add(edge);
                */
```

```
            } else {
                for (int j = 0; j < IN.nodes.size(); j++) {
                    if (IN.nodes.get(j).name.equals(statementParts[0])) {
                        subConcept = IN.nodes.get(j);
                        break;
                    }
                }
            }

            Node superConcept = new Node();
            // checking whether the super-concept is already in the list of nodes
            if (!checkExistence(IN, statementParts[2])) {
                superConcept.name = statementParts[2];
                IN.nodes.add(superConcept);
            } else {
                for (int j = 0; j < IN.nodes.size(); j++) {
                    if (IN.nodes.get(j).name.equals(statementParts[2])) {
                        superConcept = IN.nodes.get(j);
                        break;
                    }
                }
            }

            // declaring a new edge to store the current statement
            Edge edge = new Edge();
            edge.start = subConcept;
            edge.end = superConcept;

            if (statementParts[1].equals("IS-A")) {
                edge.polarity = true;
            } else {
                edge.polarity = false;
            }

            // storing the edge inside the list of connections of the subConcept
            subConcept.connections.add(edge);
        }
        return IN;
    }

    // method which checks if a certain concept is already in the list of nodes
    private static boolean checkExistence(Graph IN, String conceptName) {
```

```java
        for (int j = 0; j < IN.nodes.size(); j++) {
            if (IN.nodes.get(j).name.equals(conceptName)) {
                return true;
            }
        }
        return false;
    }

    private static void printIN(Graph IN) {
        for (int i = 0; i < IN.nodes.size(); i++) {
            for (int j = 0; j < IN.nodes.get(i).connections.size(); j++) {
                System.out.print(IN.nodes.get(i).name);
                if (IN.nodes.get(i).connections.get(j).polarity) {
                    System.out.print(" IS-A ");
                } else {
                    System.out.print(" IS-NOT-A ");
                }
                System.out.println(IN.nodes.get(i).connections.get(j).end.name);
            }
        }
    }

    // recursive method which finds all the paths between two provided nodes in a graph
    private static void findPaths(Graph IN, Node start, Node end, Path visited, ArrayList<Path> paths) {
        // if the current start Node is equal to the end Node
        if (start.equals(end)) {
            // add the current Path to the ArrayList of Paths
            visited.polarity = true;
            paths.add(visited);
            return;
        } else {
            // find all the nodes adjacent to the current one
            ArrayList<Node> adjacentNodes = new ArrayList<Node>();
            ArrayList<Boolean> polarities = new ArrayList<Boolean>();
            for (int i = 0; i < start.connections.size(); i++) {
                adjacentNodes.add(start.connections.get(i).end);
                polarities.add(start.connections.get(i).polarity);
            }
            // for every node adjacent to the current one
```

```
            for (int i = 0; i < adjacentNodes.size(); i++) {
                // if an adjacent node has already been visited, ignore it
                if (visited.path.contains(adjacentNodes.get(i))) {
                    continue;
                }
                Path temp = new Path();
                // if the connection is 'IS-NOT-A'
                if (!polarities.get(i)) {
                    // if the node at the end of the connection is the destination, we accept it as a path
                    if (adjacentNodes.get(i) == end) {
                        temp.path.addAll(visited.path);
                        temp.path.add(adjacentNodes.get(i));
                        temp.polarity = false;
                        paths.add(temp);
                        continue;
                    }
                    // if the node after the connection (IS-NOT-A) is not the destination, we ignore that path since
                    // IS-NOT-A can only be at the end of the path
                    else {
                        continue;
                    }
                } else {
                    temp.path.addAll(visited.path);
                    temp.path.add(adjacentNodes.get(i));
                    // recursive call, this time starts search for paths from the adjacent node instead of the original starting node
                    findPaths(IN, adjacentNodes.get(i), end, temp, paths);
                }
            }
        }
    }

    // method which prints paths
    private static void printPaths(ArrayList<Path> paths) {
        for (int i = 0; i < paths.size(); i++) {
            for (int j = 0; j < paths.get(i).path.size(); j++) {
                System.out.print(paths.get(i).path.get(j).name);
                if (j < paths.get(i).path.size() - 2) {
                    System.out.print(" IS-A ");
                }
```

```java
                if (j == paths.get(i).path.size() - 2) {
                    if (paths.get(i).polarity) {
                        System.out.print(" IS-A ");
                    } else {
                        System.out.print(" IS-NOT-A ");
                    }
                }
            }
            System.out.println();
        }
        System.out.println();
    }

    // Method which determines the Shortest Distance Path
    private static void shortestDistance(ArrayList<Path> paths) {
        ArrayList<Path> shortestPaths = new ArrayList<Path>();

        int minSize = paths.get(0).path.size();
        for (int i = 1; i < paths.size(); i++) {
            if (paths.get(i).path.size() < minSize) {
                minSize = paths.get(i).path.size();
            }
        }

        for (int i = 0; i < paths.size(); i++) {
            if (paths.get(i).path.size() == minSize) {
                shortestPaths.add(paths.get(i));
            }
        }

        System.out.println("Preferred Path/s (Shortest Distance):");
        printPaths(shortestPaths);
    }

    // Method which determines the Shortest Inferential Distance Path
    private static void inferentialDistance(Graph IN, Node superConcept, ArrayList<Path> paths) {
        // ArrayList of Paths which will hold all the Preffered Path/s (Inferential Distance)
        ArrayList<Path> inferentialPaths = paths;

        // Checking for Preempted Edges:

        // boolean value storing whether a certain edge has already been shown to be preempted, thus avoiding having
```

```
to
        // search the rest of the edges in the path
        boolean preempted;
        // Iterating through every path
        for (int i = 0; i < inferentialPaths.size(); i++) {
            preempted = false;
            if (inferentialPaths.get(i).polarity) {
                // Iterating through every Node in the Path (starting from the second node => j=1)
                for (int j = 1; j < inferentialPaths.get(i).path.size(); j++) {
                    // ArrayList of Paths which will store every path between the previous node and the superConcept of the
                    // current node
                    ArrayList<Path> preemptedEdges = new ArrayList<Path>();
                    Path visited = new Path();
                    visited.path.add(inferentialPaths.get(i).path.get(j - 1));
                    findPaths(IN, inferentialPaths.get(i).path.get(j - 1), superConcept, visited, preemptedEdges);
                    // finding if any of the paths lead to the same superConcept but have a negative edge
                    for (int k = 0; k < preemptedEdges.size(); k++) {
                        // if the alternate path found ends with a Negative Edge
                        if (!preemptedEdges.get(k).polarity) {
                            // we can say that the edge is pre empted
                            preempted = true;
                        }
                    }

                    // if an edge is found to be pre empted
                    if (preempted) {
                        // the Path is removed from the Array List of admissible Paths
                        inferentialPaths.remove(i);
                        // the counter is adjusted to account for the left shift of the Array List
                        i--;
                        // the inner loop breaks to check the next Path
                        break;
                    }
                }
```

```java
            }
        }

        // Checking for Redundant Edges:

        // boolean value storing whether a certain edge has already been shown to be redundant, thus avoiding having to
        // search the rest of the edges in the path
        boolean redundant;
        // Iterating through every path
        for (int i = 0; i < inferentialPaths.size(); i++) {
            redundant = false;
            // Iterating through every node (except the last => -1)
            for (int j = 0; j < inferentialPaths.get(i).path.size() - 1; j++) {
                // ArrayList of Paths which will store every path between 2 consecutive nodes along a path
                ArrayList<Path> redundantEdges = new ArrayList<Path>();
                Path visited = new Path();
                visited.path.add(inferentialPaths.get(i).path.get(j));
                findPaths(IN, inferentialPaths.get(i).path.get(j), inferentialPaths.get(i).path.get(j + 1), visited, redundantEdges);
                // finding the longest path between the 2 consecutive nodes
                int maxPathLength = 1;
                for (int k = 0; k < redundantEdges.size(); k++) {
                    // minus 1 since a Path of 2 nodes only has 1 Edge
                    if ((redundantEdges.get(k).path.size() - 1) > maxPathLength && redundantEdges.get(k).polarity) {
                        maxPathLength = redundantEdges.get(k).path.size() - 1;
                    }
                }
                // if the longest path is longer than 1 edge, therefore that edge is redundant and that path is inadmissible
                if (maxPathLength > 1) {
                    redundant = true;
                }

                // if an edge is found to be redundant, the loop through nodes breaks as an optimisation
```

```
                if (redundant) {
                    inferentialPaths.remove(i);
                    i--;
                    break;
                }
            }
        }

        System.out.println("Preferred Path/s (Inferential Distance):");
        printPaths(inferentialPaths);
    }
}
```