Assignment-3

The Puzzle Problem

Date March 8, 2024

Name: Shrey Patel

Email id: shrey.patel@dal.ca

Dalhousie Id: B00960433

Overview:

In the puzzle problem, we are trying to compare the difficulty levels of various puzzles. As this topic is subjective and it can vary from person to person, we take multiple inputs sometimes and check for the support level that decides what population's votes are needed to be considered. Understanding the difficulty levels of wooden and wire puzzles can be subjective, as individuals may have different perceptions of what makes a puzzle easy or hard. Rather than asking people to assign a difficulty level to each puzzle, we can gather feedback by asking them to compare puzzles and determine which one is harder or easier relative to another. This approach leverages people's intuitive sense of comparing puzzles rather than ranking them individually.

However, aggregating evaluations from a crowd based on comparisons poses challenges. It's common to encounter scenarios where certain puzzles are not compared at all, or where individual opinions seem inconsistent. For example, one person may perceive puzzle A as harder than puzzle B, while another person may have the opposite opinion. These discrepancies arise due to the subjective nature of comparisons, where each person's perception may differ. To address such challenges, we introduce the concept of "support." Support represents the proportion of votes that favor a particular comparison outcome. By establishing a threshold for support (e.g., 5%), the system can filter out low-support comparisons and focus on those with sufficient agreement among participants.

The task involves designing a class called "PuzzleLibrary", which takes input from a stream that has comparisons of various puzzle two at a time. The solution here gives the user flexibility to get various comparisons of the puzzles that are given in the input. In this problem one has flexibility to change the support that is required for the comparisons to be called as valid whenever the user wants. The data in the comparisons list will be added or deleted accordingly. The support has impact on all other functions as well. Apart from this there is a functionality to check the equivalent puzzles for a given puzzle input, which checks for transitivity as well.

The problem then is further divided into 3 main parts. The first phase gets the input from the input streams and adds to a temporary arraylist. The second phase includes filtering out the data on the basis of the given support after calculating support for every comparison. The third step then includes getting harder, hardest and equivalent puzzle for the given input string. It also includes finding the puzzle group of puzzles that have been compared at least once.

In this Java class, we aim to develop a tool that helps analyze crowd consensus on puzzle difficulties, taking into account the concepts of support and equivalence. By implementing this class, we can gain valuable insights into the perceived difficulty levels of wooden and wire puzzles based on user comparisons.

Files and External data:

The implementation of the code mainly requires 2 files; one is the main PuzzleLibrary file where the logic for the comparisons like harder, equivalent and hardest is given. The other file can the input file from which the input stream is read which consists of the list of the puzzle comparisons. Every comparison is stored in this file by a tab space in between where the first puzzle is harder then the second puzzle.

Below three figures show the data flow of the implementation of the puzzle problem:

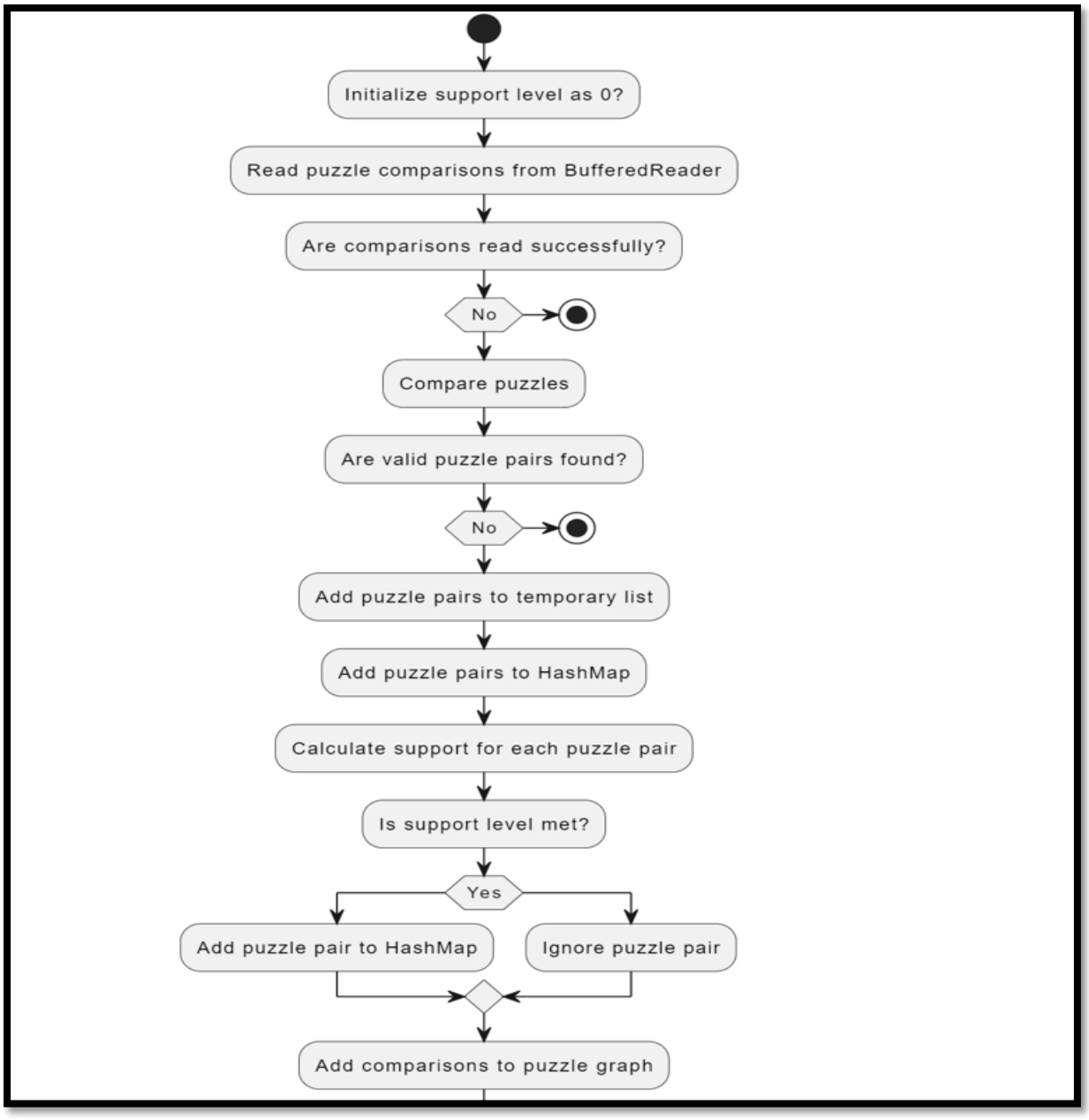

Figure 1. Data Flow chart of adding comparisons to a Hash Map

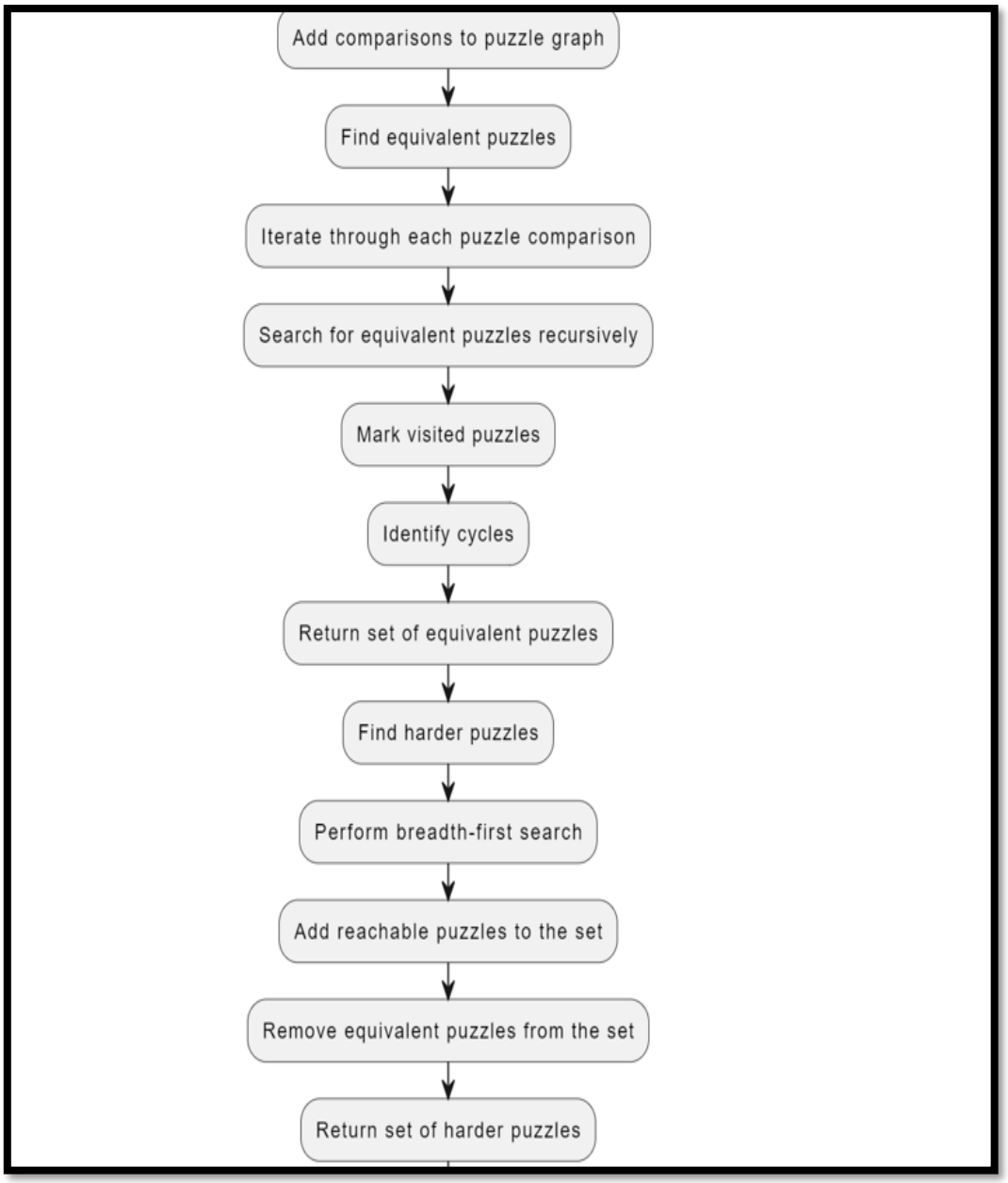

Figure 2. Flow Chart of Equivalent and Harder puzzle

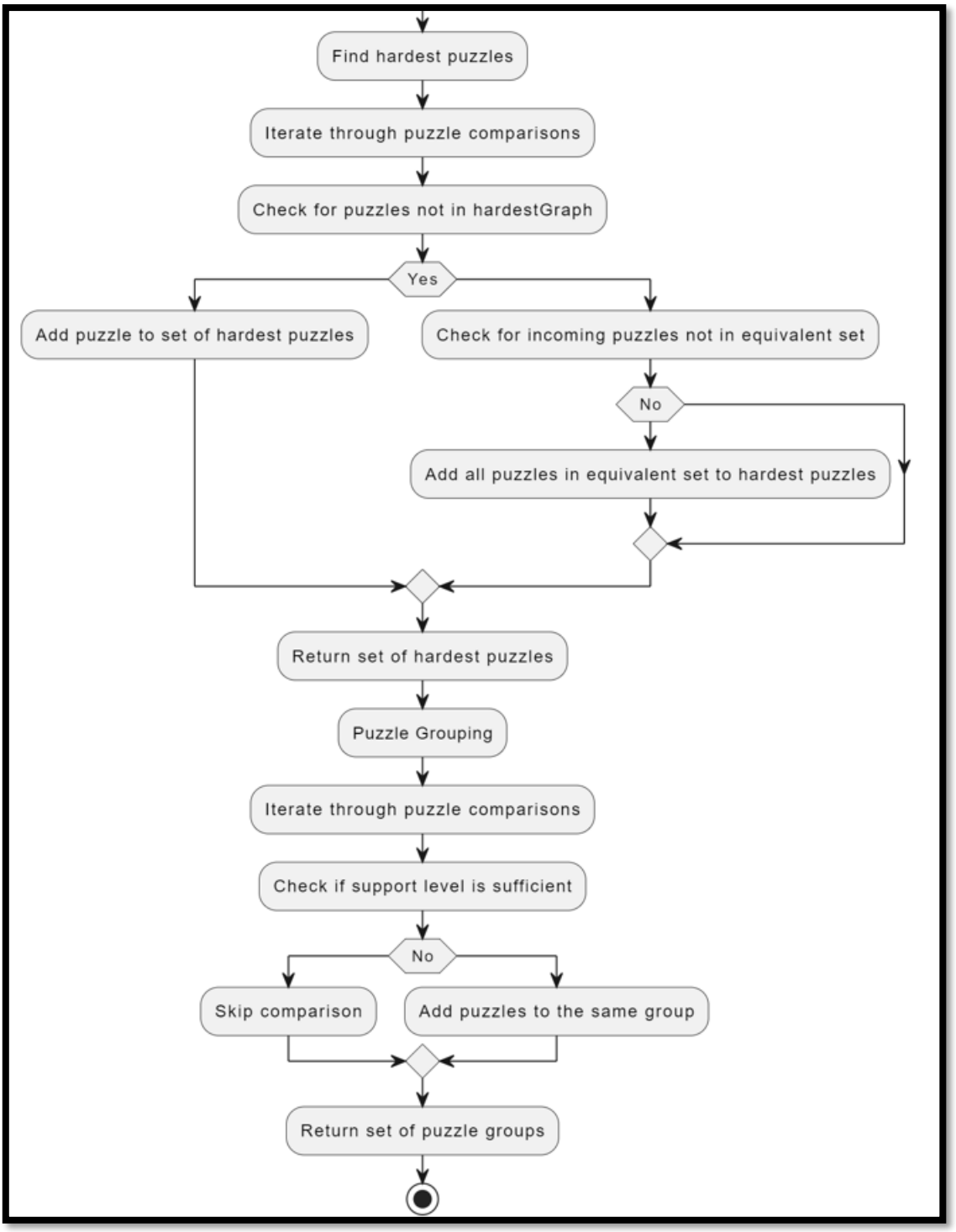

Figure 3. Flow Chart of Hardest puzzle and Puzzle group

The puzzleLibrary class implements the class which defines seven different methods which have different operations to the tree:

- Constructor PuzzleLibrary(Int Support): Creates a new PuzzleLibrary object with the specified support level for comparisons. The support level indicates the minimum

level of support needed for comparisons to be considered. The support parameter is an integer representing the support percentage divided by 100.

- setSupport(int support): Sets a new level of support needed for the PuzzleLibrary object. The support parameter follows the same format as in the constructor. Returns true if the new support level is set successfully, otherwise returns false.
- comparePuzzle(Bufferedreader streamOfComparisons): Reads a set of comparisons between puzzles from the provided BufferedReader stream. Each line of the stream represents a comparison between two puzzles, separated by a tab character. The first puzzle is considered harder than the second puzzle in each comparison. Returns true if all comparisons are successfully read into the library, otherwise returns false.
- equivalentPuzzle(String puzzle): Returns the set of all puzzles that are deemed equivalent in difficulty level with the given puzzle. Equivalence is transitive, meaning if puzzle A is equivalent to puzzle B and puzzle B is equivalent to puzzle C, then puzzle A is also equivalent to puzzle C. The returned set includes the given puzzle itself.
- harderPuzzle(String puzzle): Returns the set of all puzzles that are deemed harder in difficulty level compared to the given puzzle. If puzzle A and puzzle B are equivalent, and puzzle C is harder than puzzle B, then puzzle C is considered harder than puzzle A. The returned set does not include the given puzzle itself.
- hardestPuzzle(): Returns the set of all puzzles that have no puzzle listed as harder than them. This includes puzzles that are not directly compared to others but are deemed hardest due to transitive relationships with equivalent puzzles.
- puzzleGroup(): Groups the puzzles of the same type together based on comparisons with sufficient support. Returns a set of sets, where each inner set contains puzzles of the same type that have comparisons with sufficient support.

Data Structure and their relation to each other:

For PuzzleLibrary:

As per the guidelines the efficiency of the code did not matter, the output only mattered. So, I preferred to use only simple data structures for my code. In the whole logic for all the functions mainly 1 data structure that is array. I have used this data structure as I had good knowledge of both it and have used it in the previous projects as well.

In this problem our main goal was to achieve all the required functionalities using any data structure that we want, making the code robust such that it can handle any type of input. Here I have used data structures like array, hash map, hash set, arary list and linked list for my implementation of the code.

- **Arrays:** Arrays are used as the primary data structure for storing puzzle comparisons, puzzle pairs, tempPuzzleList Array List: This ArrayList of type String[] is used to temporarily store puzzle pairs read from the input stream. Each element of this ArrayList is an array containing two strings representing a puzzle pair. String[] Arrays: Puzzle pairs are represented as arrays of strings. For example, String[] puzzleArray represents a puzzle pair with two elements, where puzzleArray[0] is one puzzle and puzzleArray[1] is another puzzle. Also arrays is used as the key in the key value pair of the Hash Map that stores the puzzle comparisons and their support.

- **HashMap:** A HashMap<String[], Double> is used to store puzzle comparisons along with their support level. The key of the HashMap is an array of strings representing a puzzle pair, and the value is the support level for that comparison.
- **ArrayList:** An ArrayList of string arrays (ArrayList<String[]>) is used as a temporary list to store puzzle pairs before adding them to the HashMap. This ArrayList is used in the comparePuzzles() method to read puzzle comparisons from a BufferedReader and add them to the HashMap.
- **HashSet:** HashSet is used to store equivalent puzzles, harder puzzles, and hardest puzzles. In the equivalentPuzzles(), harderPuzzles(), and hardestPuzzles() methods, HashSet is employed to store sets of equivalent puzzles, harder puzzles, and hardest puzzles, respectively. HashSet ensures that duplicate puzzles are not included and provides efficient membership testing.
- **LinkedList:** LinkedList is used as a queue in the harderPuzzles() method for breadth-first search traversal. It is utilized to store puzzles to be processed in a first-in-first-out manner during the traversal.

Overall, these data structures work together to efficiently store and manipulate puzzle comparisons, allowing for operations such as finding equivalent puzzles, harder puzzles, hardest puzzles, and grouping puzzles based on their comparisons and support level.

Assumptions:

- Support for the elements is calculated by counting that comparisons and dividing it by total comparisons.
- The support level provided during initialization determines the threshold for considering puzzle comparisons. Puzzle pairs with a support level above this threshold are considered valid for further processing.
- Puzzle comparisons are expected to be in a specific format, where each line contains two puzzle identifiers separated by a tab character. Any line that does not adhere to this format is considered invalid and skipped during processing.
- The puzzle graph (puzzleGraph) is constructed to store all puzzles with edges indicating relationships from harder to easier puzzles. This graph representation facilitates efficient traversal and manipulation of puzzle relationships.
- Another graph (hardestGraph) is maintained to store all puzzles with edges representing incoming puzzles for each puzzle. This graph provides additional information on the relationships between puzzles and supports certain operations such as finding the hardest puzzles.
- Puzzle comparisons are assumed to be consistent, meaning that if puzzle A is compared to puzzle B, and puzzle B is compared to puzzle C, then puzzle A is automatically compared to puzzle C. Inconsistent comparisons might lead to unexpected behavior in certain operations.
- The code assumes a reasonable limit on the number of puzzles that can be processed efficiently. This limit might influence the design decisions and algorithms used for puzzle comparison and analysis.

Choices:

- I have used HashMap to store the unique comparisons and their support. This created a key value pair for me which helped me to get easy access of those elements when needed. Also hash map does not allow duplicate entries in the map.
- I have created two graphs using hash map for equivalent and hardest puzzle. This graph stored the nodes from where there were either incoming or outgoing edges, which helped me to solve the hardest and equivalent puzzle function easily.
- Also I have used array to store the data as key inside the Hash map, so that I can access the first and second element very easily.
- I have tried to use stack in equivalent puzzle whenever I could not find a path after reaching a node, I use the pop functionality of the stack which deleted the node from the current path.
- I have selected to use DFS for traversal as it gives me flexibility to select one node and check all available paths for it at one go.
- I have used arrayList to temporarily store the input stream of puzzles in form of the string. I then use support to filter out the data from this and store into a HashMap.

Key algorithms and design elements:

Constructor PuzzleLibrary(int support):

Input: Integer value representing the support level for comparisons.

Output: None.

Algorithm:

- Check if the support level is within the range [0,100]. If not, throw an IllegalArgumentException.
- Initialize the class variables support, comparisonsOfPuzzle, tempPuzzleList, puzzleGraph, and hardestGraph with the provided values.

setSupport(int support):

Input: Integer value representing the new support level.

Output: Boolean value indicating whether the support level is set successfully.

Algorithm:

- Check if the provided support level is within the range [0, 100].
- If it is, update the support variable with the new value
- Clear the comparisonsOfPuzzle map, and add puzzle pairs to the map again based on the new support level.
- Return true if the support level is set successfully, otherwise return false.

comparePuzzles(BufferedReader streamOfComparisons):

Input: BufferedReader containing puzzle comparisons.

Output: Boolean value indicating whether the comparisons are read successfully.

Algorithm:

- Read each line from the BufferedReader until the end of the stream.
- Parse each line to extract puzzle pairs.
- Add valid puzzle pairs to the tempPuzzleList.
- After reading all lines, add puzzle pairs from tempPuzzleList to the comparisonsOfPuzzle map.
- Return true if the comparisons are read successfully, otherwise return false.

addtoHashMap(ArrayList<String[]> puzzles):

Input: ArrayList of puzzle pairs.

Output: HashMap containing puzzle pairs with their support levels.

Algorithm:

- Clear existing data in comparisonsOfPuzzle, puzzleGraph, and hardestGraph.
- Iterate through each puzzle pair in the input list.
- Calculate the support level for each puzzle pair.
- If the support level meets the specified threshold, add the puzzle pair to the comparisonsOfPuzzle map.
- Update the puzzleGraph and hardestGraph based on the puzzle pairs.
- Return the updated comparisonsOfPuzzle map.

countforMax(String[] array1):

Input: String array representing a puzzle pair.

Output: Double value representing the count of the maximum support for the puzzle pair.

Algorithm:

- Count the occurrences of the given puzzle pair in the tempPuzzleList.
- Return the count.

equivalentPuzzles(String puzzle):

Input: String representing a puzzle.

Output: Set of strings representing equivalent puzzles to the given puzzle.

Algorithm:

- Initialize sets and a stack for tracking visited puzzles and cycles.
- Perform depth-first search (DFS)[1] to find equivalent puzzles recursively.
- Handle cycles[2] in the equivalence relationship.
- Return the set of equivalent puzzles.

harderPuzzles(String puzzle):

Input: String representing a puzzle.

Output: Set of strings representing harder puzzles than the given puzzle.

Algorithm:

- Perform breadth-first search (BFS) to find all puzzles reachable from the given puzzle.
- Exclude equivalent puzzles from the result set.
- Return the set of harder puzzles.

hardestPuzzles():

Output: Set of strings representing the hardest puzzles.

Algorithm:

- Iterate through the puzzle comparisons.
- Add equivalent puzzles of each puzzle in the comparison to a set.
- Identify the hardest puzzles based on the comparisons and their relationships.
- Return the set of hardest puzzles.

puzzleGroup():

Output: Set of sets where each set contains puzzles belonging to the same group.

Algorithm:

- Group puzzles based on their comparisons and support level.
- Return the set of sets representing puzzle groups.

Limitations:

- Here we assume that support remains constant after new graph is created, but if we add value to the input then there is no provision of dynamically changing it.
- There is a possibility that the program does not handle large input streams because there are few nested loops inside it which needs to be optimized.
- The data structure that I have used provide me the functionality that I need, but it uses high memory, this could harm us when we give a very large data set.
- Better efficient data structures can be used for creating puzzle graphs as the implementation relies on the nested loops which cannot be optimized for larger data set.
- Recursive traversal and stack manipulation are used in the method to find similar puzzles; this approach might not be effective for huge datasets or deeply nested equivalency relationships. There might be ways to increase the algorithm's effectiveness.
- For some cases, we can enter the puzzles of two different types which can never be compared, there is not validation for the same.

Test Cases:

PuzzleLibrary(int support) constructor

Input validation

- Null input support
- Empty input support.

Boundary cases

- Try to add negative input for support.
- Try to add support greater than 100.
- Try to enter maximum integer value of support.
- Try to enter minimum value of the support.
- Enter 0 support.
- Try to enter double value as input.

Control flow

- Try to call the function when there is no input of comparisons.
- Read input that is between 0 and 100.
- Read input that has support exactly as 0 or 100.

Data flow

- Try to call the function when there is not input comparisons.
- Call the function before the input file is read.
- Call the function before compare puzzle function.

boolean setSupport(int support)

Input validation

- Enter null or empty value as input.

Boundary cases

- Try to add negative input for support.
- Try to add support greater than 100.
- Try to enter maximum integer value of support.
- Try to enter minimum value of the support.
- Enter 0 support.
- Try to enter double value as input.

Control flow

- Try to call the function when there is no input of comparisons.
- Read input that is between 0 and 100.

- Read input that has support exactly as 0 or 100.
- Set support to a data set that has only one comparison.
- Set support to a data set that has only same comparisons in whole input file.
- Set support to a data where there are many groups in the comparisons.

Data flow

- Try to call the function when there is not input comparisons.
- Call the function before the input file is read.
- Call the function twice.
- Set support before compare puzzle function.
- Call the function before the constructor is created.
- Call the function before and after equivalentPuzzle function.
- Call the function before and after harderPuzzle function.
- Call the function before and after hardestPuzzle function.
- Call the function before and after puzzleGroup function.

boolean comparePuzzles(BufferedReader streamOfComparisons)

Input validation

- Enter null or empty value as input.
- Empty comparisons list.

Boundary cases

- Try to read the input with only 1 puzzle comparison.
- Try to read input with duplicate comparisons.
- Try to read input with puzzle compared with itself.
- Try to read input with puzzle compared with null or empty string.
- Try to read input with more than two puzzles in one line.
- Try to read input which is not separated by tab space.

Control flow

- Enter only one comparison in the input.
- Try to enter more than 100 comparisons.
- Compare puzzle with itself.
- Compare puzzle with null or empty string.
- Read the input with trailing blank lines.
- Read the input with leading blank lines.
- Read the input which is not a string.
- Read input where there is uneven spacing in between the puzzle comparison (not tab space).

- Read input where there is no space in between the puzzles in same line.
- Compare more than 2 puzzles in same line.

Data flow

- Try to call the function when there is not input comparisons.
- Call the function twice.
- Try to read two files that has valid input streams.
- Try to read two files where second file has invalid input while the first has valid input.
- Set the new support before this function.
- Call the function after setting a new support that is greater than the old support.
- Call the function after setting a new support that is lesser than the old support.
- Call the function before the constructor is created.
- Call the function before and after equivalentPuzzle function.
- Call the function before and after harderPuzzle function.
- Call the function before and after hardestPuzzle function.
- Call the function before and after setting a new support.
- Call the function before and after puzzleGroup function.

Set<String> equivalentPuzzles(String puzzle)

Input validation

- Enter null or empty value as input.

Boundary cases

- Try to get equivalent puzzle when there is only one comparison.
- Try to get equivalent puzzle for empty string.
- Try to get equivalent puzzle for null value.
- Try to get when there is no equivalent puzzle in the comparison.
- Try to get when there are more than 1 equivalent puzzles.
- Try to get for transitive property like if A is equivalent to B, and B is equivalent to C then A and C must be equivalent.
- Try to get equivalent puzzles for value that is in input file but not filtered after given support.

Control flow

- Enter only one comparison in the input.
- Try to enter more than 100 comparisons.
- Get equivalent puzzle which is not in comparison.
- Get equivalent puzzle for null or empty string.

- Call for a puzzle that is present in different puzzle groups.
- Call for puzzle that is present in one cycle of equivalence.
- Call for puzzle that is present in more than one cycle of equivalence.
- Call the function when the whole input is equivalent to one another.
- Call for puzzle that is nested in one cycle in other.
- Call for puzzle that has no equivalent puzzle other than itself.

Data flow

- Try to call the function when there is not input comparisons.
- Call the function twice.
- Call the function after setting a new support that is greater than the old support.
- Call the function after setting a new support that is lesser than the old support.
- Call the function before the file is read.
- Call the function when the file is empty.
- Try to call the function when two files that has valid input streams.
- Call the function before the constructor is created.
- Call the function before and after setting a new support.
- Call the function before and after harderPuzzle function.
- Call the function before and after hardestPuzzle function.
- Call the function before and after puzzleGroup function.

Set<String> harderPuzzles(String puzzle)

Input validation

- Enter null or empty value as input.

Boundary cases

- Try to get harder puzzle when the input is empty or null.
- Try to get harder puzzle when there is only one comparison.
- Try to get harder puzzle when there are all puzzles harder than a given puzzle.
- Try to get harder puzzle when all puzzles are equivalent or easier then the given puzzle.
- Try to get harder puzzle in case of transitivity.
- Try to get harder puzzle for value that is in input file but not filtered after given support.

Control flow

- Enter only one comparison in the input.
- Try to enter more than 100 comparisons.
- Get harder puzzle which is not in comparison.

- Get harder puzzle for null or empty string.
- Call for a puzzle that is present in different puzzle groups.
- Call for puzzle that is present in one cycle of equivalence.
- Call for puzzle that is present in more than one cycle of equivalence.
- Call the function when the whole input is equivalent to one another.
- Call the function where there is no puzzle harder then the given puzzle.
- Call for puzzle that is nested in one cycle in other.
- Call for the puzzle where the whole cycle is harder than a given puzzle; here cycle means the set of equivalent puzzles.

Data flow

- Try to call the function when there is not input comparisons.
- Call the function twice.
- Call the function after setting a new support that is greater than the old support.
- Call the function after setting a new support that is lesser than the old support.
- Call the function before the file is read.
- Call the function when the file is empty.
- Try to call the function when two files that has valid input streams.
- Call the function before the constructor is created.
- Call the function before and after setting a new support.
- Call the function before and after hardestPuzzle function.
- Call the function before and after equivalenPuzzle function.
- Call the function before and after puzzleGroup function.

Set<String> hardestPuzzles( )

Input validation

- No input validation needed.

Boundary cases

- Try to get hardest puzzle when the input is empty or null.
- Try to get hardest puzzle when there is only one comparison.
- Try to get hardest puzzle when there are all puzzle comparisons unique in the input stream.
- Try to get hardest puzzle when all puzzles are equivalent to each other.
- Try to get hardest puzzle in case of transitivity.
- Try to get hardest puzzle and check of the value that is in input file but not filtered after given support.

Control flow

- Enter only one comparison in the input.

- Try to enter more than 100 comparisons.
- Get hardest puzzle which is not in comparison.
- Get hardest puzzle for null or empty puzzle comparison.
- Call when there are different puzzle groups.
- Call when a particular puzzle is present in one cycle of equivalence.
- Call for puzzle that is present in more than one cycle of equivalence.
- Call the function when the whole input is equivalent to one another.
- Call the function where there are all puzzles equivalent.
- Call when there are nested cycles on inside other.

Data flow

- Try to call the function when there is not input comparisons.
- Call the function twice.
- Call the function after setting a new support that is greater than the old support.
- Call the function after setting a new support that is lesser than the old support.
- Call the function before the file is read.
- Call the function when the file is empty.
- Try to call the function when two files that has valid input streams.
- Call the function before the constructor is created.
- Call the function before and after setting a new support.
- Call the function before and after harderPuzzle function.
- Call the function before and after equivalenPuzzle function.
- Call the function before and after puzzleGroup function.

Set<Set<String>> puzzleGroups( )

Input validation

- No input in this function; so does not need validation.

Boundary cases

- Try to get puzzle group when there is only one comparison.
- Try to get puzzle group for empty string.
- Try to get puzzle group for null value.
- Call function when there are only 2 puzzles in every group, all comparisons different in input stream.
- Try to get puzzle group when all puzzles in one group.
- Try to get puzzles for one, two and multiple groups.
- Get puzzle groups from two different input files.

Control flow

- Check the group for only one comparison.
- Check the group when then are all unique comparisons in the input stream.
- Check the group when there is transitivity.

- Check the group when the input stream is null or empty.
- Check for the group when there are many cycles in the comparisons.
- Call the function when puzzle has only compared to a particular puzzle.

Data flow

- Try to call the function when there is not input comparisons.
- Call the function twice.
- Call the function after setting a new support that is greater than the old support.
- Call the function after setting a new support that is lesser than the old support.
- Call the function before the file is read.
- Call the function when the file is empty.
- Try to call the function when two files that has valid input streams.
- Call the function before and after the constructor is created.
- Call the function before and after setting a new support.
- Call the function before and after harderPuzzle function.
- Call the function before and after hardestPuzzle function.

References:

1. *GeeksforGeeks. (n.d.). Java Program for Depth First Search (DFS) for a Graph. Retrieved from* https://www.geeksforgeeks.org/java-program-for-depth-first-search-or-dfs-for-a-graph/
2. *Baeldung. (n.d.). Detecting a Cycle in a Directed Graph in Java. Baeldung.* https://www.baeldung.com/java-graph-has-a-cycle