Overview

We have implemented a set of classes to receive puzzle comparisons and perform operations to obtain relations, such as which puzzles are related to each other and which are the hardest among those groups. We have also implemented methods to determine which puzzles are harder or equivalent to a given puzzle. Moreover, all these relations are performed based on the support level. Please refer to the assignment specification for details on the required methods.

The problem decomposes into four main components: handling puzzle comparisons, setting support levels for those comparisons, creating a graph from existing comparisons, and performing operations on that graph.

A PuzzleLibrary contains three types of data: the permanent input, the graph of that input, and the support level. In the permanent input, whenever comparePuzzles() is called, the upcoming data is appended to the existing data present in the permanent input. Then, whenever EquivalentPuzzles(), HardestPuzzle(), or HarderPuzzle() is called, we first create a graph from the permanent input based on the current support level. Whenever the support level is changed using setSupport(), we clear that graph so that the next time we perform any of the above operations, we get a new graph based on the new support level.

Overall, the program facilitates the analysis of puzzle comparisons based on a certain support level.

Files and External Data

The implementation divides into 3 classes (Figure 1):

- PuzzleLibrary - It primarily facilitates puzzle comparison and analysis through graph-based algorithm to determine puzzle relationships such as equivalence, harder, hardest and groups.

- Graph - It provides methods for performing operations on a graph data structure, such as depth-first search traversal, finding disjoint graphs, identifying equivalent, harder, and hardest puzzles, and detecting cycles within a graph, and also have method to merge the list containing the atleast one common puzzle.

- Utils - It provides methods for creating unique elements from a graph, converting between 2D ArrayList and 2D Set, calculating data size, and checking support thresholds for frequency analysis.

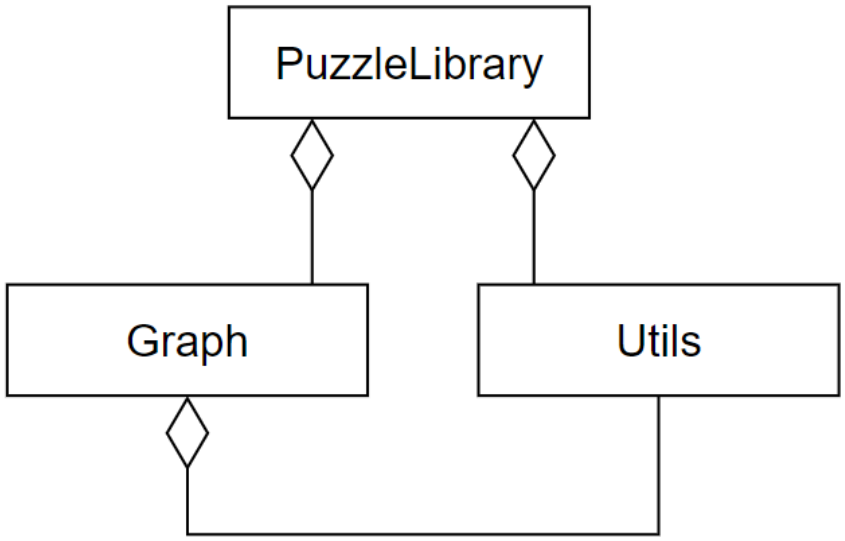

Figure 1 Class interactions

Data structures and their relations to each other

PuzzleLibrary:
- Contains the main method and all the methods required for the assignment.
- Manages the main data structures for storing puzzle data.
- Uses a Map to store input data, as it is easy to manipulate and allows tracking the frequency of each comparison occurrence. (Map<String, Integer>)
- Also stores the graph created from the input using another Map for representation of each puzzle and a set of related puzzles. (Map<String, Set<String>>)

Graph:
- Handles graph-related operations such as traversal using depth-first search (DFS), retrieving harder puzzles, finding the hardest puzzle, identifying equivalent puzzles, getting puzzle groups, identifying cycles present in the graph, and merging lists.
- Uses a 2D ArrayList to store puzzle groups for efficient access and to group puzzles within the same graph, minimizing space usage. (ArrayList<ArrayList<String>>)
- Utilizes a Map to track visited puzzles during graph traversal using DFS, with puzzle names as keys and their visited status (true or false) as values. (Map<String, Boolean>)

Utils:
- Provides utility methods for basic operations needed in the puzzle library.
- Includes methods like `convert2DSetTo2DArrayList` for converting a 2D Set to a 2D ArrayList, `convert2DArrayListTo2DSet` for the reverse operation, and a 1D set to store unique elements from the graph.

Assumptions

- When the PuzzleLibrary object is created, it must be initialized with a valid support value. Here, I've implemented it in such a way that if the support value is not valid, it throws an IllegalArgumentException.

Choices

- To illustrate the relationship between puzzles, I chose a graph because, for this problem, it best fits compared to a tree or any other data structure [1]. Moreover, using a graph, we can easily get the direction of dependency and also detect cycles, which helps in determining the answer to equivalent puzzles [3]. Furthermore, it helps in identifying puzzle groups, as in a graph, there are concepts of connected components [2].
- Here, I choose a map to represent the graph. If I were to choose a 2D matrix, the amortized time and space complexity would be higher in comparison.
- Additionally, the worst-case searching time complexity is O(N) in a map, where N is the size of the map, while in a 2D matrix it is O(N*M), where N is the number of rows and M is the number of columns.
- For storing puzzle comparisons, I again used a map as it is easier to handle the frequency of those particular comparisons within the map.
- For any methods from EquivalentPuzzles(), HarderPuzzles(), HardestPuzzles(), and PuzzleGroup(), if any error occurs or there is no answer for that particular puzzle or data, I return an empty set instead of returning null.

Key algorithms and design elements

<u>EquivalentPuzzle(String puzzle):</u>

1. Retrieve all cycles from the graph using getAllCycles(graph) and store them in cycles.
2. Merge cycles to obtain unique cycles using mergeList(cycles) and store in uniqueCycles.
3. Iterate through each cycle in uniqueCycles:
   For each cycle, iterate through each element:
   If the current element is equal to the input puzzle:
   Return entire cycle in which current element was found.

HarderPuzzles(String puzzle):

1. Initialize an empty ArrayList ans
2. Perform DFS traversal starting from the input puzzle
3. Iterate through the composite graphs:
    If the puzzle is found in a cluster:
        Add all puzzles in the cluster to ans
        Break the loop
4. Remove puzzles that are present in the DFS path from ans
5. Initialize an empty ArrayList harderPuzzles as a copy of ans
6. For each puzzle str in ans:
    Perform DFS for the puzzle str
    If the input puzzle is not present in the DFS path for str:
        Remove the puzzle str from harderPuzzles
7. Return harderPuzzles

HardestPuzzles():

1. Create a set uniqueElements containing unique puzzles from the graph
2. Iterate through each puzzle str in uniqueElements:
    Get the list of harder puzzles for str.
    If no harder puzzles exist or the list is empty:
        Add the puzzle str to hardestPuzzles
3. Return hardestPuzzles.

PuzzleGroup():

1. Iterate through the vertices of the graph:
    For each vertex:
        If the vertex is visited:
            Continue to the next vertex
        Else:
            Perform DFS for the current vertex:
                dfs(graph, vertex, tempVisited, currDfsPath)
                Add currDfsPath to dfsPaths
2. Merge the lists from dfsPaths in which atleast one element is common.
3. Return remaining dfsPaths.

DFS:

dfs(graph, currVertex, visited, dfsPath):

1. if currVertex is already visited:
        return
2. Mark currVertex as visited
3. Add currVertex to dfsPath
4. Get neighbors of currVertex
        For each neighbor of currVertex:
            dfs(graph, neighbor, visited, dfsPath)

MergeLists: (Merger those lists which have atleast one common element)
mergeList(ArrayList<ArrayList<String>> list):

1. Initialize a flag isMergePerform to false
2. while list is not empty:
        if size of list is 1:
            Add the only list to mergedList and break
        Initialize an tempList
        Initialize an empty set index
        Get the first list (currList) from list and remove it
        For each character currChar in currList:
            For each i in list:
                For each character in the i:
                    If currChar matches any character in the i:
                        Add the i to tempList
                        Add the index of this i to index
                        Set isMergePerform to true
                        Break out of inner loop
        Sort index in descending order
        Remove paths from list at indices specified by index
        If a merge operation was performed:
            Add the last list from mergedList into list
            Remove the last added list from mergedList
            Reset isMergePerform to false
3. Return mergedList

Limitations

- This program is designed only for the string data type.
- It can accept only comparisons that contain only two puzzles.
- Support cannot be set to float or double; it must be an integer.

Test cases for PuzzleLibrary class

puzzleLibrary(int Support)

Input validation
- Null parameter.
- Empty support value.

Boundary cases
- Read positive support value.
- Read negative support value.
- Read support value as zero.
- Read support value as the maximum integer support value.
- Read support value as the minimum integer support value.

Control Flow:
- Create an object with an appropriate value of support between 0 and 100.
- Create an object with an invalid value of support which throws IllegalArgumentException().

Data Flow:
- This is the constructor of PuzzleLibrary(), so first, we have to create an object with initial support. Only then can we perform other operations like EquivalentPuzzles(), SetSupport(), and HardestPuzzles().

setSupport(int support)

Input validation:
- Null parameter.
- Empty support value.

Boundary cases:
- Read positive support value.
- Read negative support value.
- Read support value as zero.
- Read support value as the maximum integer support value.
- Read support value as the minimum integer support value.

Control flow:
- Set the support value immediately after comparePuzzle().
- Set the support value before comparePuzzle().

- Set the support value after calling comparePuzzle() twice.
- Set the support value after some operations like EquivalentPuzzles() and HarderPuzzle(String Puzzle).

Data flow:
- Call before ComparePuzzle().
- Call after ComparePuzzle().
- Call after operations such as HardestPuzzle().

ComparePuzzles(BufferedReader streamOfComparisons)

Input validation:
- Null parameter for input stream.
- empty input stream.

Boundary cases:
- Testing a single comparison.
- Ensuring the method can handle long inputs.

Control flow:
- Testing presence of blank lines in the input.
- Checking inappropriate spaces between puzzles in one line.
- Testing three puzzles in one line.
- Ensuring both puzzles are not the same.
- Testing calling the method twice with different inputs.

Data flow:
- Calling before setSupport().
- Calling after setSupport().
- Calling before operations like EquivalentPuzzles() and HardestPuzzles().
- Calling after operations like EquivalentPuzzles() and HardestPuzzles().

puzzleGroup()

Input validation:
- Not required.

Boundary cases:
- Call where only one puzzle group exists.
- Call where with multiple puzzle groups.

Control flow:
- Call when the data set is empty or null
- call when the data is present but graph is empty or null.
- Call with single puzzle groups.
- Callwith multiple puzzle groups
- Call after changing of support values.
- Checking behavior when input data doesn't meet the minimum support threshold.

Data flow:
- Call after ComparePuzzle().

EquivalentPuzzles(String puzzle)

Input validation:
- Null puzzle
- Empty puzzle

Boundary cases:
- Call with an empty data set.
- Call when the the graph that we make from input stream is empty or null.
- Call with puzzle for which there are no equivalent puzzle found
- Call where entire input forms a cycle so entire graph wil be the answer.

Control flow:
- Call before input stream.
- Call for puzzle not present in the input stream.
- Call where only one cycle in the input stream.
- Call where multiple cycles in the input stream.
- Call where nested cycles within the input stream.
- Call after set support set to 0%, resulting in no equivalent puzzles.
- Setting support to 100% and checking for equivalent puzzle.
- Call multiple times on same input stream but for different puzzles.
- Call multiple times on same input stream while changing the support each time.
- Call multiple times on different input stream.

Data flow:
- Call after ComparePuzzle().

HarderPuzzles(String puzzle)

Input validation:
- Null puzzle.
- Empty puzzle.

Boundary cases:
- Call with an empty data set.
- Call when the the graph that we make from input stream is empty or null.
- Call with puzzle for which there are no harder puzzles.
- Call where entire input forms a cycle so no harder puzzles are found.

Control flow:
- Call before input stream.
- Call for the puzzle not present in the input stream.
- Call where only one cycle in the input stream.
- Call where multiple cycles in the input stream.
- Call where nested cycles within the input stream.
- Support set to 0%, resulting in no harder puzzles.
- Setting support to 100% and checking for harder puzzles.
- Call multiple times on same input stream but for different puzzles.
- Call multiple times on same input stream while changing the support each time.
- Call multiple times on different input stream.

Data flow:
- Call after ComparePuzzle().

HardestPuzzles()

Input validation:
- Not required.

Boundary cases:
- Call with an empty data set.
- Call when the the graph that we make from input stream is empty or null.
- Call where entire input forms a cycle so all the puzzles are hardest puzzle.

Control flow:
- Call before input stream.
- Call where only one cycle in the input stream.

- Call where multiple cycles in the input stream.
- Call where nested cycles within the input stream.
- Support set to 0%, resulting in no hardest puzzles.
- Setting support to 100% and checking for harder puzzles.
- Call multiple times on same input stream but for different puzzles.
- Call multiple times on same input stream while changing the support each time.
- Call multiple times on different input stream.

Data flow:
- Call after ComparePuzzle().

References

[1] Romin Vaghani, “Difference between graph and tree”, geeksforgeeks.org. Available: https://www.geeksforgeeks.org/difference-between-graph-and-tree/ [Accessed Mar. 08, 2024]

[2] “G-4. What are Connected Components ?,” Youtube, Aug. 06, 2022. Available:https://www.youtube.com/watch?v=lea-Wl_uWXY&list=PLgUwDviBIf0rGEWe64KWas0Nryn7SCRWw&index=4 [Accessed Mar. 08, 2024].

[3] “6.8 Detect Cycle in Directed Graph | Data Structures and Algorithms Tutorials”, Youtube, March 09, 2019. Available:https://www.youtube.com/watch?v=AK7BuT5MgU0 [Accessed Mar. 08, 2024]