CSCI 3901 Assignment-4

The Logic Grid Problem

Date April 4, 2024

Name: Shrey Patel

Email id: shrey.patel@dal.ca

Banner Id: B00960433

Overview:

The domain of logic grid problems calls for an intellectual journey: a quest to find one's way across an artfully and carefully designed state space. It was during the 20th century that these puzzles grew in popularity as a leisurely pastime, challenging participants with logical reasoning from a series of given clues towards making approachable deductions aimed at finding a coherent solution.

The essence of the logic puzzle is to put up a scenario with the people aligning to the female cats, each of them having different preferences and attributes highlighted through a logic grid. This grid helps in visual assistance, allowing solvers to note their deductions of whether they confirm a match with a checkmark or negate possibilities with an X. The idea at the end is to draw logical connections across different categories to complete the puzzle tableau. A classic example of the matching type includes matching people to cats with definite preferred activities and kitten counts. Cljson is a type of puzzle which gives a set of clues useful to the solver in making deductions. Some are giving direct information, like what some cats must have been doing or their activities, while others are giving an implication, like a guess of what some cat must not have after elimination.

In this way of solving the problem, it suggests to look into a well-defined set of possibilities, treating the problem as a search space of looking for viable options, rather than playing around with a playground of free guesses. The logical steps are propelled by clear, deductive reasoning, taking to the side the inclination for intended misguidance. Basically, a logic grid puzzle presents a rather organized but yet very loose framework of intellectual exploration. It requires a combination of careful analysis and creative derivation of conclusions in order to solve the riddles laid out in the grid.

To solve this problem, I have used combination of various data structures like hash map, set, and arrays in order to make sure that the goal is achieved with optimized approach.

Files and External data:

The implementation of the code mainly requires 1 main class called LogicGridPuzzle and 4 other classes called listOrder, possibleValues, logicalDeduction and addCategory. The main class calls many other classes that serve different functionalities. A class called addCategory makes sure that the category and its values are added perfectly to the map which can be further used. Also, listOrder and possibleValues class makes sure that the required functionality to pass the clues of the puzzle is satisfied and mapping is done on that moment only with optimized approach.

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

- **boolean setCategory**(String categoryName, List<String> categoryValues): This function creates a new category for puzzle clues and solutions, filled with relevant values. If the clues suggest a certain order among these values, that order is established right from the start in this list. Each category within a single puzzle needs to have an equal number of values. If everything is set up correctly and the category can be used, the method returns true. Otherwise, if there's a hitch and the category can't be established, you'll get a false.
- **boolean valuePossibilities**(String baseValue, Set<String> options, Set<String> exclusions): Here, we specify that the "baseValue" is restricted to being one of the values in "options" and it's not allowed to assume any value in "exclusions." It's okay for either set to be empty if there's nothing to note. For example, if we have a clue with "Ruby" as the baseValue, we might say it can only be {"sleep"} for options, and have no exclusions. This will return true if the puzzle can work with this clue, or false if it decides to skip it for any reason.
- **boolean listOrder**(String orderCategory, String firstValue, String secondValue, int orderGap): This method sets up a sequence within a category, saying that "firstValue" comes before "secondValue." If the "orderGap" is zero, it just means "firstValue" has to be sometime before "secondValue" without specifying when. If "orderGap" is a number, then it's that exact number of spots before. It returns true if this clue fits into the puzzle framework, or false if it's left out.
- **Map<String, Map<String, String>> solve**(String rowCategory): This method is all about cracking the puzzle. It uses logical deductions or tries different value pairings to find a consistent solution. The solution structure is like a map within a map, with the first layer sorted by "rowCategory" and the second by the other categories, showing pairings or nulls if pairings aren't figured out yet. If the puzzle's unsolvable at this point, you get null back.
- **boolean check**(String rowCategory, Map<String, Map<String, String>> solution): Lastly, this function checks if a proposed solution ticks all the boxes of the puzzle's constraints. It uses the same "rowCategory" and solution structure as the solve method. If everything aligns perfectly, it returns true. But if something's amiss, or if it's too early to say due to insufficient information (especially about the order of values), you'll get a false.

Data Structure and their relation to each other:

For LogicPuzzleGrid:

As per the guidelines the efficiency of the code did matter. 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 hash map, hash set, and linked list for my implementation of the code.

- **HashMap:** A HashMap<String, Map<String,String>> is used to store the cross relationships of the clues and their values. The key of the HashMap is an array of strings representing all the values across different categories.
- **LinkedList:** LinkedList is used as for storing the values of categories in the setCategory() method for maintaining order of storing the values which is further used in the listOrder functions.

The values stored in the list are retrieved and a one to one mapping is done amongst all the values of different categories except the same category in order to manipulate the Hash Map when clues are passed from valuePossibilities an listOrder functions. Overall, these data structures work together to efficiently store and manipulate the clues that are given for different values different categories. This eventually helps efficiently to solve the puzzle by mapping Yes or No infront of the values of the clue that are provided.

Assumptions:

- The puzzles of same category are not compared with each other.
- The entries that are entered in various categories are case sensitive.
- Also, duplicate entries are not allowed in same or different categories else it would be difficult to give the clue.
- There are uniform number of values in each category.
- There is only one option for each unique category value to other category value. No value likes two other values from different category.
- Also, there is a limit of maximum categories and maximum number of values that can be given as input.
- The clues can be given in any order, at the end the answer remains the same.
- If A likes B, and B likes C then automatically A likes C, similarly for exclusions.

Choices:

- I have used map inside a hash map to store one to one mapping of all the values across all categories. This helps me to put yes, no or null for each mapping. In virtual terms I am making the system play the game.
- Also I am doing deductions after every value possibility or list order is given, this helps me to keep the mapping updated. So that at the end I can retrieve data that has yes entry for the final solution.
- For the list order, I am treating the gap and the two values as input for the value possibility's function. Here I prefer to put no in front of the values that cannot be yes for the given input of the gap. This make logical deduction at the end.
- I create one to one mapping after every new category and values are added to the system. I fill initial values of mapping as null which I further update with yes or no according to the options and exclusions.
- I use list inside the map to store the categories and its values. Where key is category name and list stores the values of it. This helps me to get the index of then values stored which I can use in the listOrder function.

Key algorithms and design elements:

setCategory(String categoryName, List<String> categoryValues):

Input: CategoryName is the one which is unique and has list of values under same category.

Output: Boolean true if category is successfully added.

Algorithm:

- Check whether the category or values are null.
- Check whether the category already exists or not.
- Check whether the number of values provided in category are uniform across all categories.
- Add the values of the category to the map, where key is the category name and values are stored in form of list.
- After that create a one to one mapping of values of different categories with one another and give null as default input to every pair.

valuePossibilities(String baseValue, Set<String> options, Set<String> exclusions):

Input: baseValue that is any value of the categories. Options are those which are in favour for the base value and exclusions are not in favour or liked by the base value.

Output: Boolean value if the mapping is done properly and clues are added.

Algorithm:

- Check whether basevalue, options and exclusions are null or not.
- Then check if they exists in the given puzzle grid.
- Check whether the baseValue and options are of same category or not.
- After that put Yes in the hash map of one to one mapping for options and No in the same map for exclusions.
- After that deduce the further relationships according to constraints.

listOrder(String orderCategory, String firstValue, String secondValue, int orderGap):

Input: ordercategory for whose values ordergap is given. OrderGap is the exact difference between the index of firstValue and secondValue.

Output: Boolean value if clues are mapped successfully.

Algorithm:

- Check whether each of the input parameters are null or not.
- Validate the orderGap whether it exceeds the size of the values of categories.
- Then check whether the given orderCategory exists or not.
- After that use mathematics to find the probable index of the first value and second value from the order Gap provided.
- Do changes in the mapping for either yes or no.

solve(String rowCategory):

Input: String of the row category for which we are finding solutions

Output: Map<String, Map<String, String>> which contains the category name other than the input and respective unique solution value from all the remaining categories.

Algorithm:

- Check whether the input parameter is null or not.
- Traverse the whole hash map of one to one relationship.
- Check for the values of the given row category, go through their respective hash map and store the values from other categories where there is yes.
- Return the new map where the answer is stored.

check(String rowCategory, Map<String, Map<String, String>> solution):

Input: String rowCategory and the solution for that particular category.

Output: Boolean true if the solution that solve functions gives and the solution provided matches properly.

Algorithm:

- Check whether the given row category is null or not.
- Pass the rowCategory in the solve function mentioned above and check whether the given solution matches solution that is received from the solve function.

Optimization:

- I have used deduction methods after every value possibility is added. This makes the code optimized as at the end when solve method is called, every possible logical deductions are present in the hash map stored.
- Apart from this, I have used a hash map to traverse the options and exclusions for each value of the categories. It is more efficient to use hash map as it gets easy to traverse through its values.
- I have used break statements in the for loops to avoid unnecessary execution of the loops.
- I have created separate classes for different functionalities, which increases modularity and more ease can be achieved to search for different values.
- As I have not used state space so no more optimization could be done.

Limitations:

- The code has many loops that are used to put yes or not as per the options and exclusions.
- The nested approach of putting yes or no might be time consuming and can affect the overall performance if the input is large.
- Also, the deductions are done at various steps, there is chance that one particular deduction is done more than twice. As I have not solved the questions using state space.

- For the check method, I have assumed that the given row category solution is to be compared with the solution that we get from solve method on passing that row category. This might not be true always, as it depends on the order of the clues given.
- 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 grid graphs as the implementation relies on the nested loops which cannot be optimized for larger data set.
- There is no method of updating the values of the categories at any point of time.

Test Cases:

setCategory():

Input validation

- Null input category and its values.
- Empty input.
- Number of values not consistent in all categories.

Boundary cases

- Try to add same value twice in the values for same category.
- Try to add same category twice.
- Try to add different number of values in each category entered.
- Try to enter same values in different categories.
- Try to add more than 10 values in one category.

Control flow

- Try to call the function twice for same category and same values.
- Try to edit the category values after they are put.
- Try to add same set of category values for different categories.

Data flow

- Try to call the function for empty values.
- Try to call the function for after doing value possibilities.
- Try to call function after list order.
- Try to call the function after solving the puzzle.

valuePossibilities():

Input validation

- Enter null or empty value as input for base value or options or exclusions.
- Enter all null values at once.

Boundary cases

- Try to give more exclusions or options than the size of the category values.
- Try to give both exclusions and options from same category.
- Try to give input of options and exclusions from same category as the base value.
- Try to enter as many options and exclusions as possible.
- Try to enter null as options and many exclusions and null as exclusion and many options.
- Try to enter same options for same base value twice.
- Try to enter same exclusions for same base value twice.
- Try to enter value possibilities in random order.

Control flow

- Enter only one option or one exclusion in the input.
- Try to add the maximum given number of options and exclusions that is 7.
- Add options and exclusions in random order.
- Read the input of options and base value that is not present in any category.
- Add options after modifying the set category.
- Check whether logical deductions are maintained or not.

Data flow

- Try to call the function when there is not base value or options.
- Try to call the function before and after adding categories.
- Try to call the function before and after list order.
- Try to call function before and after solve.
- Try to call the function for 2 or more options of same category.
- Try to call function for 2 or more options of different category.
- Repeat it for exclusions.

listOrder():

Input validation

- Enter null or empty value as input.
- Enter only positive values for order gap.
- Enter row category other than those present in the categories.

Boundary cases

- Try to enter first value and second value of different categories.
- Try to enter first and second value of same categories.
- Try to add order gap in negative.
- Try to add order gap more than total values of the category values.
- Try to add row category and first and second value of same categories.

Control flow

- Try to add null values for either order gap or two values.
- Try to give order gap 0, 1 and maximum size of category values.
- Try to give row category as input which is not present in the categories.
- Try to give different order gaps for same values two in a row.

Data flow

- Try to call the function when there is not input values for categories.
- Try to call the function before and after adding categories.
- Try to call the function before and after value possibilities.
- Try to call function before and after solve.
- Try to call the function before and after creating a new value set for different category.

solve():

Input validation

- Enter a null or empty string in input for the row category.

Boundary cases

- Try to enter category value which is not added previously.
- Try to get solution without passing any clues.
- Try to call the function after only one clue passing.
- Try to call the function for different categories one by one.

Control flow

- Try to call for a puzzle that has many values in categories. Check whether logical deduction occurs or not.
- Try to give row category which is not present in the categories.
- Try to call function when there is combination of clues passed in list order and value possibilities.
- Try to call the function after giving clues in different order.

Data flow

- Try to call the function when there is not input values for categories.
- Try to call the function before and after adding categories.
- Try to call the function before and after value possibilities.
- Try to call function before and after check.
- Try to call function before and after listOrder.
- Try to call the function before and after creating a new value set for different category.

check():

Input validation

- Enter a null or empty string in input for the row category and for the solution as well.

Boundary cases

- Try to call the function after only one clue passing.
- Try to call the function for different categories one by one.
- Try to enter wrong solution for the given row category.
- Try to enter row category which is not in the category list mentioned earlier.
- Try to enter solution after entering every value possibility either option or exclusion.

Control flow

- Try to call the function after giving clues in different order.
- Try to call the function after every list order or value possibility is added.
- Try to check the output when the solution consists of no values for the given category.
- Try to check for the null input solution

Data flow

- Try to call the function when there is not input values for categories.
- Try to call the function before and after adding categories.
- Try to call the function before and after value possibilities.
- Try to call function before and after solve.
- Try to call function before and after listOrder.
- Try to call the function before and after creating a new value set for different category.