CSCI 3901 Assignment 4

Due date: 11:59pm Thursday, March 29, 2024 in git.cs.dal.ca at
https://git.cs.dal.ca/courses/2024-winter/csci-3901/assignment4/xxxx.git
where xxxx is your CSID (this repository already exists, so clone it and then add your code to it).

Problem 1

Goal

Work with exploring state space.

Background

Logic grid problems became popular in the 20th century to pass the time. The puzzles present you with a set of clues and require you to make logical deductions to solve the puzzle.

Figure 1 shows an example of a logic grid puzzle where people are being matched with female cats, where the cats have preferred activities and numbers of kittens. On the right, you find a set of clues to the puzzle. The grid on the left shows all the combinations of answers and allows someone to track their conclusions. For example, one might enter a checkmark in a square when we know that a match happens or an X in a square when we know that the combination of values cannot happen. The goal is to match a value from each category with answers from all other categories, thus filling in the table seen in Figure 2.

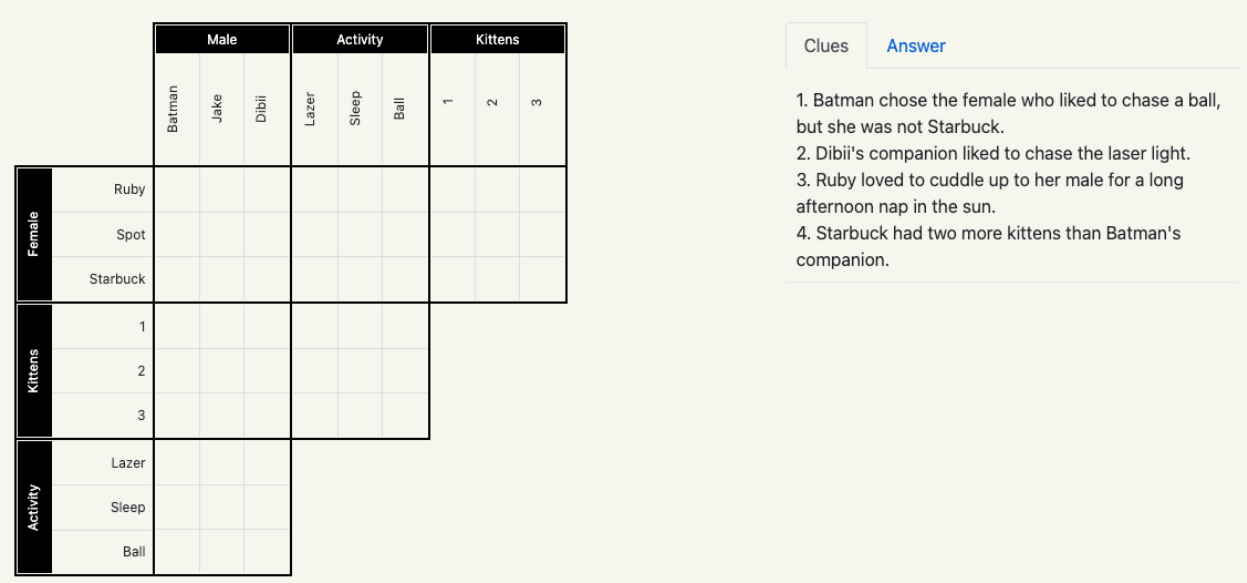

Figure 1 Sample logic grid puzzle from https://www.ahapuzzles.com/logic/logic-puzzles/cats-in-spring/

Some clues for a puzzle might give you direct information. Clues 2 and 3 Clue 1 in Figure 1, for example, tell us direct information: Dibii's cat chases the laser light and the cat Ruby sleeps. Other clues are slightly less diret: Clue 1 tells us diretly that Batman's cat chases a ball and is not Starbuck and infers that Starbuck is not the cat that likes to chase a ball. All these conclusions lead us to the solution grid shown in Figure 3.

Clues | Answer

Female	Male	Activity	Kittens
Ruby			
Spot			
Starbuck			

Figure 2 Solution table for the logic puzzle of Figure 1, from https://www.ahapuzzles.com/logic/logic-puzzles/cats-in-spring/

As we summarize information in the grid, we can then conclude additional information. In Figure 3, we can now see that Jake's cat must be the one who likes to sleep since that is the only option left in the Male / Activity area. After that, given that Jake's cat likes to sleep and, from earlier, Ruby is the cat who likes to sleep, then Jake's cat must be Ruby.

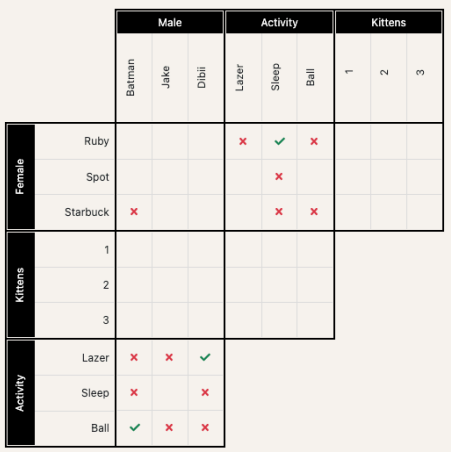

Figure 3 Logic puzzle from Figure 1 after we consider clues 1, 2, and 3.

With a bit more deduction from Clue 4, the logic puzzle in Figure 1 has the solution shown in Figure 4.

Typical clues can include information like

- Give you a pairing
- Tell you that some pairing can't happen
- Tell you that a pairing might be one of a couple of options
- Tell you that two pairings might be ordered relative to one another by a third category (like clue 4 in Figure 1). That ordering might be by some temporal ordering or one category (like a category of days of the week).

Clues can get more complex and tricky. This assignment will treat them as a search among possibilities rather than have you create logic for complex clues.

Female	Male	Activity	Kittens
Ruby	Jake	Sleep	2
Spot	Batman	Ball	1
Starbuck	Dibii	Lazer	3

Figure 4 Solution to the logic puzzle of Figure 1.

Most puzzles use simple logic steps. Common ones are

- If A and B are paired and if B and C are paired then A and C are also paired
- If A and B are paired and if A cannot be paired with C then B cannot be paired with C
- If A is bigger / later than B then A is not the smallest / earliest and B is not the biggest / latest

The puzzles also have the convention of not intentionally trying to mislead. If a clue gives to options for information then we can assume that those two options are distinct. We see this patter in clue 1 of Figure 1 where we could directly conclude that the cat who likes to play with

the ball isn't Starbuck since both those options are mentioned as potential options (or non-options, in the case of Starbuck).

Problem

Write a class called "LogicGridPuzzle" that accepts categories with values and accepts constraint clues. The class will then seek to solve the puzzle from the given set of clues.

The class has at least 5 methods:

- boolean setCategory(String categoryName, List<String> categoryValues) – Define a category or grouping of values for the puzzle clues and the solution. If one of the values is used in a clue that implies an order among the values then assume the order from this definition List.

  All categories in one puzzle must have the same number of values.

  Return true if this category can be used once the method is concluded. Return false if the category cannot be used.

- boolean valuePossibilities(String baseValue, Set<String> options, Set<String> exclusions) – Indicate that value "baseValue" is limited to one of the values in "options" and cannot take on any values in "exclusions". One of the sets can be a null value if there are no items to consider for it. For example, clue 3 from Figure 1 would have "Ruby" as the baseValue, would have a single item set of {"sleep"} for options and a null value for exclusions.

  Return true if this clue is accepted into the puzzle. Return false if the clue will not be used for some reason.

- boolean listOrder(String orderCategory, String firstValue, String secondValue, int orderGap) – Indicate that fistValue comes before secondValue according to their matchings in "orderCategory". If "orderGap" is 0 then the rule says that "firstValue" comes sometime before "secondValue". If "orderGap" is an integer then "firstValue" comes that number of order positions before "secondValue".

  For example, clue 4 of Figure 1 would say that Batman comes before Starbuck according to category "kitten" and the orderGap is 2…because the values of "kittens" differ by 1 so the order differences and the math differences are the same.

  Return true if this clue is accepted into the puzzle. Return false if the clue will not be used for some reason.

- Map<String, Map<String, String>> solve(String rowCategory) -- Solve the puzzle, either through logical deductions or by trying value pairings to see if they lead to a consistent final solution. The trying of values can be needed if the puzzle needs a clue that cannot

be encoded using one of the rules that we have implemented.

Report the solution or deductions as an analogy to Figure 4. The outer Map is indexed by the values of "rowCategory" and returns a Map of its own. That secondary Map is indexed by the remaining categories and its values are pairings, or null if nothing has been paired yet.

For example, Figure 4 would come from solve("Female"). If s = solve("Female") from Figure 4 then s.get("Ruby") would give us a map with keys of "Male", "Activity" and "Kittens" and then s.get("Ruby").get("Activity") would give the value of "Sleep".

Return null if there is no current solution.Return null if there is no solution.

- boolean check(String rowCategory, Map<String, Map<String, String> solution) – Determine if the solution presented by the "solution" parameter satisfies all of the constraints in the system. The rowCagetory and solution parameters match the descriptions of rowCategory and the method output from the solution() method.

  Return true if all constraints are met. Return false if one or more constraint is not met. If a constraint speaks to an order of values and there isn't enough information in the solution yet to test that order then consider the constraint met…for now.

You get to choose how you will represent the puzzle in your program and how you will proceed to solve the puzzle.

You are required to implement the following optimizations to searching the space of solutions in solve(). These relate to the rules from valuePossibilities() or relate to general deductions. You are responsible for making these optimizations as effective as possible in your search.

1. If valuePossiblities says to exclude values for some option then your state search does not try solutions that use these values.
2. If valuePossiblities says that some option only has a set of included values all from the same category then your state search in that category only tries solutions that use those values.
3. Given three categories, if option A in the first category matches option B in the second category and if option B in the second category matches option C in the third category then you will automatically match option A with option C in a solution.

The marking scheme also asks you to implement some additional optimizations of your own choosing. You need to decide on what those optimizations might be. Grading of your optimization is not only on accomplishing the optimization, but on how effectively your choice of optimization reduces the search space and how efficiently (by running time) it runs.

Example

We can create the puzzle of Figure 1 with the following calls (correcting "lazer" spelling of that web site to "laser"):

- possibleValues("Batman", {"Ball"}, {"Starbuck"})
- possibleValues("Dibii", {"Laser"}, null)
- possibleValues("Ruby", {"Sleep"}, null)
- ~~either of the following (both not needed)~~
    - listOrder("Kitten", "Batman", "Starbuck", 2)
    - ~~numericOrder("Kitten", "Batman", "Starbuck", 2, true)~~

When solved, solution("Female") would return the following Map:
"Ruby" -> { "Male" -> "Jake", "Activity" -> "Sleep", "Kittens" -> "2" }
"Spot" -> { "Male" -> "Batman", "Activity" -> "Ball", "Kittens" -> "1" }
"Starbuck" -> { "Male" -> "Dibii", "Activity" -> "Laser", "Kittens" -> "3" }

If we call solution("Activity") then we would get the following Map instead:
"Sleep" -> { "Male" -> "Jake", "Female" -> "Ruby", "Kittens" -> "2" }
"Ball" -> { "Male" -> "Batman", "Female" -> "Spot", "Kittens" -> "1" }
"Laser" -> { "Male" -> "Dibii", "Female" -> "Starbuck", "Kittens" -> "3" }

Assumptions

You may assume that

- All category names and category values will be unique across the whole puzzle.
- The category names and values are case-sensitive. If two different by any character then treat them as different.

Constraints

- You may use any data structures from the Java Collection Framework.
- You may not use an existing library that already solves this puzzle
- If in doubt for testing, I will be running your program on timberlea.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Notes

- **Start with solving the puzzle with brute force: trying different combinations of pairings until one works.** While that may seem inefficient, it gets you a solution to refine. Starting with having your code make logical deductions is longer to get bits right, even though the logical deductions lead you to solutions with smaller running times.

  I make this recommendation from experience: my solution started with the logical deductions first and the coding was bogged down with lots of details quickly….worse tracking of pieces than you had in assignment 2. **Were I to start again, I would start with exhaustively trying out combinations of values first.**

- You might consider creating a method that prints out the logic grid (like Figure 3) for your puzzle to help with debugging. It may help you see if you are missing logical deductions.
- You do not need to create full puzzles to do most of your testing. Instead, you can create partial puzzles that only use one rule (won't be fully solvable) and see what deduce() concludes from your data. You can also set up potential solutions yourself and run check() with them to see if your code is checking a rule properly without having had to solve using that rule yet.
- Develop a strategy on how you will solve the puzzle before you finalize and start coding your data structure(s) for the puzzle.
- Work incrementally. First write the code to get the categories and maybe print a solution. Test that. Next, write the code to directly specify valuePossibilities. Test that. Write the code to infer pairings from valuePossiblities. Test that. Then, move on to a next clue type. Tackle the trial-and-error work of solve() first, using check() to see if you have a solution. Keep your optimizations to the end.

Marking scheme

- External documentation – 2 marks
- Program design – 2 marks
- One of the following (doing both is good, but doesn't give extra marks) – 2 marks
    - List of test cases for the problem
    - A description of up to 3 additional optimizations that you could make, prioritized by which you expect to be more effective (and why). The description should be at a level that a peer in the class could implement in your code, given your external documentation.
- Ability to accept categories and clues / constraints – 4 marks
- Ability to check a solution – 6 marks
- Ability to reach a solution somehow – 5 marks
- Implementation of the required optimizations – 5 marks
- Implementation of one additional optimization of your choice – 3 marks