Assignment 2 - Puzzle Platform

1 Overview

Every day, thousands (millions?) of people play puzzle games like Sudoku or Candy Crush Saga. These puzzle games can be a fun way to sharpen your mental skills, or an addictive time sink. Either way, let's try to capitalize on this!

In this assignment, you will implement an interactive puzzle platform which supports two different features for users: make and undo moves to solve a puzzle, and give up and ask for a hint or an entire solution to the puzzle. Moreover, through an elegant use of inheritance we will to enable users to play different kinds of games on our single platform.

The program for this assignment uses a Model-View-Controller (MVC) design, which is commonly used for building interactive applications. Read this section carefully, along with the starter code, to make sure you understand how everything fits together before writing any code yourself.

Download the starter code and extract the files into your `assignments/a2` folder.

1.1 View

File(s): `view.py, game.html`

The View class is an abstract class that is the interface between the user and the rest of the puzzle application. It is responsible for displaying information about the state of the game and providing ways for the user to interact with the game. To demonstrate the power of inheritance, we have provided two different view implementations: a text view, which just uses the console to print and read in strings, and a web view, which displays the puzzle game as a very limited website.

You won't have to change any code for these files, but you will find at least the text view useful for testing your work for this assignment. You might just have some fun playing the games, too!

1.2 Controller

File(s): `controller.py`

The Controller class is a concrete class which is the connection between the view and the representation of the puzzles and automatic puzzle solver. It is responsible for understanding user inputs and performing the corresponding tasks, and then updating the view to display the results.

By default, the controller will only print out the current state of the puzzle (see the `act` method). You will add to this method on this assignment to enable the user to input moves, ask for hints, or even ask the computer to solve the puzzle.

1.3 Model: Puzzles

File(s): `puzzle.py, sudoku_puzzle.py, word_ladder_puzzle.py`

In the MVC framework, models are classes that represent the actual data and other entities in our application.

Read through the documentation of the abstract Puzzle class to understand the API all puzzles must implement. We have provided one almost-complete implementation for a Sudoku puzzle; in this assignment, you will complete it and then implement an entire other puzzle, Word Ladder.

Please note that the provided interface only fits a certain type of puzzle, which satisfy the following criterion:

- There is no randomness or hidden information. If we are at a state in the puzzle and make a move, we know with 100% certainty what the resulting state will be. This is not the case with Candy Crush, for example.

1.4 "Model": Solver

File(s): `solver.py`

We chose the particular interface in Puzzle to support generic solving algorithms which work on any type of puzzle that is compatible with this interface. Even though this generality is quite nice because it allows us to write one set of functions to work on a wide variety of puzzles, it comes with a cost. These algorithms are unable to use strategies tailored to a particular game, and so it is much slower than specialized solvers.

Your tasks here will be to implement the two recursive functions which solve a puzzle automatically, and then another, faster function which searches for a solution, but stops after exploring too many possibilities, and simply returns a plausible next step for the puzzle.

Side note: we put Model in quotation marks because we aren't using a class for this part, but instead writing a series of functions. However, this maintains the same spirit as "pure" MVC, by separating the main computation logic (puzzling solving) from both the controller and the view.

2 Part 1: Solving Puzzles

2.1 Basic Solver

Open `solver.py`. Your first task is to implement the `solve` and `solve_complete` functions according to their docstrings. Note that they are clients of the `Puzzle` interface, so you should only use methods defined in that class, and nothing special about a particular game like Sudoku.

Once you are done, you should be able to run `solver.py`, as we have given a simple example use of the `solve` function in the "main" block. You should also try replacing `solve` with `solve_complete` to make sure both work correctly, although do keep in mind that both will be fairly slow right now.

2.2 Improving SudokuPuzzle

The reason the solver is very slow right now is that the Sudoku puzzle has one particularly silly feature: in the `_possible_letters` helper method, which is used by `extensions` to generate new moves, it always returns all possible letters, even when it should be able to immediately rule out some letters based on the current letters on the board.

Modify the `_possible_letters` method so that it only returns a list of letters that do not conflict with the existing filled squares (e.g., do not try filling in a square with an 'A' if there is already an 'A' in the same column). After this, you should see a dramatic speed up in the solving of Sudoku puzzles.

2.3 Implementing Word Ladder

Now that you have experience working with the `Puzzle` interface, open `word_ladder_puzzle.py`. Your next task is to implement `WordLadderPuzzle` according to the game description in the module docstring, and the provided method docstrings. You must decide what private attributes to give this class so that you have all the necessary information to complete the all methods of the Puzzle API except `move` (which you'll do in Part 2). However, keep in mind that `WordLadderPuzzle` should implement the exact same interface as `SudokuPuzzle`, and so you should implement the four given methods plus a constructor, but not add any new public methods or attributes.

When you are finished, you should be able to run the solver functions with a Word Ladder puzzle instead of a Sudoku puzzle. Try it! (It will be very, very slow though.)

2.4 Connecting the main game

Your last task in this part is to take your work in this section and put it into the main application. Open `controller.py` and modify the `act` method so that:

- If the user command is `:SOLVE` then the program solves the puzzle and returns a string representation of the solution.
- If the user command is `:SOLVE-ALL` then the program returns string representations of all the possible solutions of the puzzle. The string representations should be concatenated together, separated by the newline character `\n`. You may find the built-in `map` and `join` useful here.

Both of these actions cause the game to end. When the game ends, the "goodbye" message is printed out (in the text view), and the program stops accepting user inputs.

3 Part 2: User Moves

Your goal for this part is to allow users to interactively solve the puzzle by typing actions into the view representing moves to make for the puzzle. Here is the required format of user inputs for each of the puzzles:

- Sudoku: `(<row>, <col>) -> <letter>`. For example, the user would type in `(2, 3) -> A` to put the letter 'A' into the spot in row 2 and column 3. Rows and columns start their index at 0.
- Word ladder: `<word>`. The user inputs the next word to go in the word ladder.

3.1 Extend the models

We have given you an abstract method `move` in the `Puzzle` class, which should take a string representation of a move and return a new puzzle state which is result of making that move. Your first task is to implement this method for both `SudokuPuzzle` and `WordLadderPuzzle`, following the input formats specified above. Note that you should raise a ValueError when the move is invalid, for any reason.

3.2 Extend the game

After that's done, you should extend the `act` method of the controller to allow the user to input moves in the view, and then, if the move is valid, return (a string representation of) the new state of the puzzle after that move.

If the user inputs an invalid move, including a string in the wrong format, you should return an appropriate error message from the `act` method.

Moreover, after the user solves the puzzle, either by typing in `:SOLVE`, `:SOLVE-ALL`, or by inputting a correct sequence of moves, the game should end, and the goodbye message should be printed.

In summary, after you are done with this task, your program should be able to:

- Show the user a starting configuration for a puzzle (this can be hard-coded to a fixed Sudoku or Word Ladder puzzle)
- Let the user type in moves, and either show the new puzzle state after a valid move, or display an error message after an invalid move
- Allow the user to "give up" at any point and type in `:SOLVE` or `:SOLVE-ALL` to immediately end the game, and display a solution (or multiple solutions) to the puzzle consistent with the user's sequence of moves. Or, display an error message if the puzzle is unsolvable from the current state.

4 Part 3: Saving moves

When working on a puzzle, the user might make a mistake and want to undo a move, and then try again from the previous state with a different move. Of course, the user can undo multiple times in a row to revert

consecutive moves. While you can implement this behaviour using a simple stack, we're going to ask for something more complex: we want you to save all the puzzle states visited by the user over the course of the game in a **tree data structure**.

Each value in the tree should consist of two parts: a **puzzle state**, and the **user input move** which was used to get to that state from its parent. The root of the tree should be the starting puzzle state and have an empty "user input" string, and each user move over the course of the game will create a new value in the tree which is a child of the current one.

If the user never undoes a move, then the generated tree looks like a list; but if the user undoes a move and then chooses a new move, and repeats this several times, the actual tree of moves can become quite complex.

Also, in order to not waste space the tree should not have duplicate states: if the user is at a state, makes a move, then undoes that move, then makes that same move, then the second time that move is made there should not be a new value added to the tree.

Change the implementation of `Controller` to incorporate this tree-like data structure, and use it to enable the following user commands:

- `:UNDO` – Revert the puzzle state to the previous one (i.e., the state of the puzzle from which the current state was reached), and return (a string representation of) the new current state. Return a descriptive error message if there is no "previous state."'
- `:ATTEMPTS` – Print out all of the puzzle states resulting from moves the user made at the current state, along with the string the user typed in to make the corresponding move. The states and moves should be printed out in the order the user made them. If the user hasn't made any moves from the current state (i.e., they just reached the state for the first time), print a message saying that.

Note that you shouldn't print out states reached in subsequent moves after the current state. For example, suppose the user is at state **A**, then makes a move to get to state **B**, and then makes another move to get to state **C**, and then does an `:UNDO` twice, returning to state **A**. If the user then enters `:ATTEMPTS`, only state **B** is printed out.

Make sure that your work is consistent with all your previous work; in other words, these changes should not affect the behaviour of any of the previous actions the user could take. Note that the `:SOLVE` and `:SOLVE-ALL` commands should always find a solution from the current puzzle state.

You may (and probably should) change the starter code we have given you to support this change.

5 Part 4: Hints

Sometime the user solving the puzzle might get stuck, but not want to give up entirely and ask for a solution. Your final task on this assignment is to enable the user to ask for a hint at their current state. A "hint" for a puzzle state is **the string representation of a valid move from the current state** that brings the user one step closer to solving the puzzle.

The natural way to do this is to solve the puzzle from the current state, and then return the first move taken to get to a solution. However, this might take a long time, depending on the complexity of the game and how close the current state is to a solution.

Instead, you will write a function which takes a puzzle state and a positive integer n, and does the following:

- Searches for a solution which can be reached from the input state in at most n moves.
- If it finds such a solution, return the puzzle state obtained from the input state after taking one move towards this solution.
- Otherwise, if it found a valid state after making n moves in sequence, return the puzzle state obtained by taking the first move. (This move could lead to a solution after more than n moves, but the solver hasn't verified that.)

- Otherwise, the input puzzle state is doomed (no solutions within n moves, and no valid states after n moves).

Here is a docstring you should use to define a function - put this in the `solver.py` file.

```
def hint_by_depth(puzzle, n):
    """Return a hint for the given puzzle state.

    Precondition: n >= 1.

    If <puzzle> is already solved, return the string 'Already at a solution!'
    If <puzzle> cannot lead to a solution or other valid state within <n> moves,
    return the string 'No possible extensions!'

    @type puzzle: Puzzle
    @type n: int
    @rtype: str
    """
```

Implement this function, and then use it in the controller to enable the user command `:HINT` , which prints a hint to the user which and leads to either a solution or valid state within `<n>` moves (including the hint itself). Note that the user should be able to type in exactly what is returned by `hint_by_depth` (when it returns a valid hint) to make a move. If `hint_by_depth` returns an error message, print that message to the user instead.

For example, if the user types in `:HINT 5` then you should call `hint_by_depth` on 5, and return its output to the user. If the user doesn't type in a positive number after `:HINT`, print an error message to prompt the user to try again.