Overview

The implementation of this program follows the method descriptions in the assignment system. Refer to that document for those method requirements.

The main goal of this program is to load a fill-in puzzle from a file with words that can solve the puzzle. It then uses those words to solve the puzzle with state space exploration.

Files and External Data:

FillinInPuzzle.java: Holds reference to puzzle and second puzzle. It's methods include: loadPuzzle, solve, print and choices.

Slot.java: Captures an empty space of puzzle grid. It maintains a list of possible words that same the same lengths as it's size. It also keeps track of the words it placed on the puzzle grid.

Puzzle.java: Captures the puzzle grid. It is responsible for placing and removing words to the grid.

SecondPuzzle.java: Creates a second puzzle with same dimensions. The purpose of this class is to keep track of connected slots.

Orientation.java: Creates a type to know if the slot is horizontal or vertical.

Data structures and their relations to each other

FillinPuzzle

slotMap: Hashmap of slots. Key contains size of Slot and Value is the slot object.

slots: ArrayList containing Slot objects.

visitedWord: Set of slots that are solved

slotStack: Stack holding all unsolved Slots.

visitedSlot: Stack holding solved slots.

Slot

connectedSlots: Set containing that slot's connected slots.

possibleWords: ArrayList containing words that are the same length as the slot.

wordsAttempted: Words that the slot have placed in puzzle at any point of time.

Puzzle

Grid: 2-d character array representation of the puzzle

SecondPuzzle

Grid: 2-d array of type ArrayList<Slot>, used to hold references of slots

Assumptions

The program requirements were specific enough that no additional assumptions had to be made.

Choices

I chose to represent the empty spaces to be solved as Slot objects. A slot class will maintain possible words that could be used to solve that puzzle.

Strategy and Algorithm for solution:

Strategy for solving slots:

Each slot will try to solve itself by using a word from it's possible words list that are not in the visited slot set. If successful, it will add the word to wordsAttempted set to move on to the next slot.

If another slot down in the decision tree could not be solved, it will move one way up to the previous slot solved, and try to solve it with another word. If it sees it already used all possible words, it will reset wordsAttempted set and backtrack to the previous slot in the decision tree.

Strategy for efficiciency:

1. I chose to solve slots that have distinct sizes first. (A puzzle with 3 two letter words, 4 three letter words and 1 five letter word. It will fill the five letter word first.
   This way, we will reduce the size of the decision tree by removing these words and slots from the pool.
2. Every slot object (empty space of a puzzle) maintains a possibleWords list. Those words contain words of same length. For example, a slot of size five might have the words "array, frail ,plush" in it's possible words list. Each slot will only attempt those words when generating a solution. This further reduces the size of the decision tree.
3. Lastly, in my initial design, I chose to write a separate secondPuzzle class. That would have the same grid as the main puzzle of type ArrayList<Slot>[][].

   When reading a slot line from the file, it will place that slots reference in the second puzzle cell. Using the second cell, we can then derive which slot is connecting with

eachother, and their connection cell. I did this because my initial plan was to solve this with a breadth first approach, first solving the slot with most connected cells, then making recurisive calls to solve the neighbours. This would let us know if a word was placed incorrectly before we move too far down in the decision tree.

I later decided to abandon this approach because of time constraints and solve it by selecting slots randomly, but I still left the code there, because I intend to implement it after the course ends.

Solve Algorithm:

- Solve slots that have distinct size. For example, For a puzzle with 3 two letter words, 4 three letter words and 1 five letter word, It will solve the five letter word first.
- Create a stack of unsolved slots and send them to be solved in Solution(Stack<Slot> slotStack, Set<String> visitedWord, Stack<Slot>visitedSlot).

Solution algorithm:

- Pop stack to get last inserted slot.
- If, slot can insert a word from it's possibleWords list,
    - Add word to visitedWord set.
    - Push Slot into visitedSlot stack.
    - Make recursive call to solution with slot stack, visited word set, and visited slot stack.
- Else, if slot can't place any of the words in it's possibleWords list,
    - Increment choice
    - Push the last slot that wasn't able to place a word in the slot stack
    - If all possible words were explored, still could not place word successfully
        - Pop from visitedSlot stack and push it into slot stack
        - Remove the word placed from the visited word set
        - Remove the word from the grid
        - make recursive call to solution with slot stack, visited word set, and visited slot stack.
    - Else, make recursive call to solution with slot stack, visited word set, and visited slot stack.