Pseudo-3D Crossword

Relevant Updates/Design Decisions

The Algorithm for Placing Words

Initial Plan:

Our initial plan for the algorithm for placing words was simply to start with the largest possible word and place it onto the grid. Then, we would go through every word in the list and attempt to place it on the grid, placing each word on the first available spot. This was problematic for two reasons:

1. The puzzles we ended up with were of very low quality. Words were usually clustered together and many times adjacent to one another, which would make for a terrible Crossword puzzle.
2. The algorithm worked on a square/rectangular grid. However, since we were planning to use a plus-sign shape grid (in order to theoretically map different sections to faces of a cube), the algorithm did not make the words spread out at all, and they remained clustered at the center of the grid.

Our First Solution:

Our solution was to devise an algorithm that looked at every possible position a word can be placed, and assigned a score for each position. It would then place the word at the position with the highest score. Our initial scoring parameters were: number of adjacent characters(had to be low), number of words crossed(the more, the better), and distance from center of the grid(the further, the better. This solved our first problem: the puzzles become of a better quality, no longer with adjacent and clustered words.

However, spreading out in the plus-sized grid was still a problem. While we were prioritizing distance from the center, this caused the algorithm to over-prioritize whatever direction it ended up branching out to first, as it was always further and thus always considered superior. Therefore, it limited its spreading to one direction, remaining very clustered.

Our Final Solution:

To solve the clustering problem, we decided that we had to change the algorithm prioritize whichever direction had the lowest character density. Therefore, we began keeping track of the number of characters placed in each of the section of the grid. We then altered the algorithm to prioritize sections with lower amounts of characters, rather than distance from the center. This elegant solution ensured a relatively uniform spread throughout the grid, solving the clustering problem.

Displaying Different Sections of The Grid

Initial Plan:

Our initial plan for displaying different sections of the grid on the user interface was to have 5 different Character[ ][ ], that were a ninth of the size of the grid, each representing one of the five sections we wanted to display. This was problematic for two reasons:

1. We had no way to track the connections between grids - using the switch perspective buttons would probably entail having several different if statements for each button, which would be inefficient, a pain in the ass, and just terrible from a design perspective.
2. Even if we managed to switch perspectives, because each grid was individual, we would not be able to correctly display words that started in one grid and ended in another; if a user filled them in, it would be very hard to alter the correct chars in multiple grids.

Our Solution:

The LinkedBounds data structure, which held the solution to both problems.

Our solution to the first problem was the “Linked” part of this data structure, which enabled us to easily switch between grid sections and keep track of where we were. Each instance of LinkedBounds holds links to adjacent LinkedBounds, letting you easily switch back and forth between adjacent ones using the perspective buttons.

Our solution to the second problem was the “Bounds” part of the the data structure. Instead of storing smaller Character[ ][ ] and displaying those, we decided simply to store boundaries that defined which part of the grid was to be displayed. This allowed us to edit the user grid directly and have the changes be displayed graphically, without having to reflect them in each specific smaller grid, especially if words spanned across these grids.

Storing the Coordinates of the Words

Initial Plan:

Our initial plan was to have each Word object also store the position of it’s first character on the grid. We thought that this would make things easier since it would allow us to quickly find specific words within the grid.

Why it was a good decision:

Storing the initial position of each word was more helpful than we expected. Firstly, it allowed us to easily all find possible placed positions by checking if two Words share a char, and quickly finding the unplaced words possible placement from the placed words initial position. This is opposed to running through the whole grid for every char of a Word, and checking each position for that char, which would have been horribly inefficient. Furthermore, it was helpful when painting the grid, as it allowed a quick way for us to find where we should place each Word’s hint number in the grid. Finally, it was helpful for filling out the mirror grid containing user Inputs, as we knew exactly where to start filling for any selected word.

char[ ][ ] vs Character[ ][ ]

Initial Plan:

In our initial design, we were using a 2D array of primitive chars for our grid. This worked fine initially, when we were simply using rectangular grids. However, it became a problem when we decided to move on to a plus-shaped grid: we had no simple way of specifying that the four corners of the grid were not valid, other than manually making sure each word didn't cross into any of those areas.

Our Solution:

We simply slightly refactored out program so that the grid was now a 2D array of Character Objects, rather than primitives. This allowed us to make the corners of the grid null, while having Characters storing (char)0 in valid areas. This facilitated the process of checking whether a word placement was valid or not, as well the initial tests of printing out the grid to the console.

Saving Time (In the Real World)

Here are two examples of how we decided to use java library functionalities to save us time from writing methods ourselves.

The first of these occasions was that we decided to use an ArrayList<Word>, which automatically handled length when appending values, instead of writing our own method for appending a Word to a Word[ ]. We then just used ArrayList.toArray to convert it to the necessary array when we were done. While trivial, I assume this saved a reasonable minute or two of our time.

Another example was when we wanted to sort an array of Words in order of longest to shortest. We could have taken some minutes to write and insertion or merge sort method that did that for us, but a simpler solution was to have the Word class implement Comparable<Word>, writing a toCompare method that compared words by their length, and then using the Arrays.sort( ) method from java.util.Arrays to sort the array.