Arisa Chue

Mr. Eckel

AI 1 Period 2

11 December 2020

Modeling Challenge III: Calibron

I began by handling the first test case of calculating whether the rectangles matches the total area. I incremented a variable with the areas of each rectangle from the list and checked if it was equal to the product of the board's dimensions. My run code would print out "Containing rectangle incorrectly sized." and end the program to prevent my code from trying to solve the input statement.

Next, I wanted to sort the rectangles in decreasing size so I would try to fit the largest piece on the board first and then go to smaller, easier pieces. Sorting the rectangles list was a bit more complicated than I expected because I had to calculate each area and remember each area with its original index. I implemented this by first creating a dictionary, which stores the index as the key and the area as the value. I would also store the area into a list and use the sort() method to arrange the areas in the list in decreasing order (I set reverse=True as well). Next, I would traverse this sorted list and obtain the key of the dictionary from each area. I created another method that would return the key from a specified value, and I added the tuple that was at the index in rectangles (the original unsorted list) to a new list. After testing this part of the code, I realized that I also had to set the dictionary value to 0 after I return the key so I could add rectangles with different dimensions that had the same area.

I used a list to represent the board, which took in a tuple containing the x and y coordinate of the top left corner and the height and width of the rectangle.

My idea to solve the puzzle was to use the backtracking algorithm we used in Sudoku and N-Queens. The method would take in a board and a list of rectangles as the parameter and would recursively call a new board that has a newly inserted piece. For my base case, I would check how many rectangles are remaining. If zero, I assume that the board is solved. Next, I call the get_next_unassigned_var() method to find the next available x, y coordinate. I do this by popping off the last rectangle in the board and checking its x and y coordinate. If the board is empty, the x and y coordinate would be the top left corner, or (0, 0). Next, I would find all the rectangles that can fit next to the rectangle found from get_next_unassigned_var(). For each of these rectangles, I would try to insert it into the board and also remove the rectangle from the list of rectangles in the parameter. If the board is not None (if the rectangle fits), I recursively call my backtracking method with this new board. If the result of the backtracking method does not return None, I return this solved board as well. If not, my method determines that it is unsolvable and returns None.

Currently, my backtracking algorithm returns None for solvable puzzles. The reason why my backtracking method doesn't work is that my get_next_unassigned_var() method can only check if a rectangle fits next to the last rectangle on the board. Instead, I need to not only look horizontally but vertically as well to successfully return the top left most available corner on any location of the board. In addition, I need to take into consideration that a rectangle can be laid out horizontally or vertically on the board. If I return the correct upper left-hand corner that a rectangle can be placed and also check different rotations of the rectangle when inserting it into the board, my recursive backtracking might work. Lastly, I think I need to consider a better data structure to store the board. Finding the next unassigned variable with a list would be complicated, and probably inefficient. I will need to brainstorm a better way to represent the board that can easily traverse through the placed rectangles since the next unassigned variable will not always be the last rectangle added to the list.