COMP3010 Assignment 1 Document

Ameer Karas,

44948956

Problem

Input: a text file to be parsed containing values W (width), H (height) and N (number of unique letters).

Output: A grid with W columns and H rows, filled by N rectangles that fill the grid.

Structure of the Code

The main methods of the program are readData, solveBaseCase, possibleRectangle, bestRectangle and solve.

ReadData ():

The readData method is a java scanner class used to populate a grid from an input file using a standard try and catch method. The method is used to populate the fields:

public static int *W* = 0;
public static int *H* = 0;
public static int *N* = 0;
With the grid's width, height, and the number of unique letters, respectively. Code iterated by a while loop (at line 456) reads each letter and its corresponding value into a HashMap of String, Integer pairs. This is representative of a letter and its corresponding rectangle size. Further, I made a 'letter' class, and instances of this class are created and added to a List of 'letter' values. This list is referred to as 'letterSet'.

solveBaseCase ():

The solveBaseCase method was my first attempt at a solution. I modified it and called it in the 'solve' method for values of N that were equal to 1. The method takes a Letter value, as well as the y and x coordinates for that letter, given by the 'Cell' class. This method has a number of conditionals that check that the given coordinates are within the boundaries of the grid, or checks that the given letter is in the expected position (which is completed by the helper method checkLocation()). If neither of these conditions are met, then the method recursively solves the rectangle by shifting around the letter's location in the grid.

possibleRectangle ():

The possibleRectangle method determines all possible formations that a rectangle can make based on the number of cells that the letter has to occupy. For example, if we have a letter 'A' that must take up 4 cells in a 4 x 4 grid, then the potential formations are:

A A A A

A
A
A
A

A A
A A

These potential rectangles would be stored in a 2D 'tmp' array that is defined by a vertical value and a horizontal value.

We then also consider the case where we are given a 1 x 1 rectangle, and all of the potential rectangles are added to an array list.

bestRect ():

From the array list of potential rectangles, we then chose the best rectangle. The best rectangle is defined as a potential rectangle that does not breach the boundary of the grid, nor overlap with another letter's rectangle. This method three helper methods:

- zeroFill: this method populates a given grid with 0's, i.e. empty cells.
- insert: this method populates a given grid with instances of a given letter.
- subjectLetter: this method checks that the sub-grid we are operating on contains the letter that is the subject of the sub-grid.

solve ():

This method is the main method that attempts to solve the given input. When called, this method calls:

- readData
- possibleRectangle
- bestRect
- solveBaseCase
- zeroFill
- CheckGrid
- addGrid: adds a sub-grid to a grid, if it:
    - has no 0's.
    - is not currently stored in the 'areas' HashMap.
- complete: this method checks whether a grid has any remaining 0's.
- printGrid: this method prints the grids in the console.

Main variables/class attributes

The main variables/class attributes in the program and the values they store are:

- **public static int *W*:** stores the value for the width of a given grid.
- **public static int *H* = 0:** stores the value for the height of a given grid.
- **public static int *N* = 0:** the number of unique letters in a given grid.
- **public static HashMap<String, Integer> *areas*:** a HashMap that stores a letter and the number of cells that its rectangle must occupy.
- **public static String[][] *arr*:** the grid returned by readData, represented as a 2D array.
- **public static List<Letter> *letterSet*:** a list of instances of type 'letter'.
- **public static ArrayList<String[][]> *rectangleList:***
- **ArrayList<Cell> loc:** an array list that stores the location of a cell that holds a unique letter.
- **private ArrayList<String[][]> potRectangles:** the list of all possible rectangles.
- **private ArrayList<String[][]> bestRectangle:** the best fit for purpose of the potential rectangles.

Algorithm Explanation

1st example case, Base case:

W = 2, H = 2, N = 1

A 4

0 0

A 0

For the base case, as N = 1, line 126 of the solve() method will take us into recursively solving the base case with the solveBaseCase() method. With the given values, the program will pass the first 2

conditionals (as the cell we are observing is within the boundaries and does not contain a letter), so we would add the current cell to the location array list, and add the current letter to the blank cell. This would also lead to the ‘num’ value decrementing (as we initially needed to fill 4 cells, and having filled one in, have reduced the number of cells to fill by one). Recursing on this process will give us our solution.

2nd example case:

W = 3, H = 3, N = 3

A 4

B 3

C 2

A 0 0

0 0 B

C 0 0

As with this scenario N is not equal to 1, we cannot rely on the solveBaseCase() method.

We call the possibleRectangle() method and the bestRect() method for each letter, one at a time.

For A, our possible rectangles are:

4/1:

A A A A

0 0 B

C 0 0

This is invalid as it exceeds the boundary,

2/2:

A A 0

A A 0

0 0 0

1/4:

A 0 0

A 0 B

A 0 0

A

This is also invalid as it exceeds the boundary.

As we only have one possible rectangle for A, it automatically becomes the best rectangle. So, we add it to the grid in the position that does not exceed the boundary and includes the base value of A. At this point, our grid looks like this:

A A 0

A A B

C 0 0

We now iterate for B this time:

Possible rectangles are:

1/3:

A A B

A A B

C 0 B

As this is the only valid rectangle, we can follow the same steps as previous for A, and have the grid:

A A B

A A B

C 0 B

Now we iterate for C. Clearly there is only one possible rectangle, making it the best rectangle by default, and we then have:

A A B

A A B

C C B

As our completed grid.