Sandeep Heera

Technical Manual for Cell.java and Spreadsheet.java

Cell.java

Since a Cell object would represent a cell in the spreadsheet, it needed fields that stored its formula, its value, and for convenience its cell address on the grid as well as a reference to the containing 2-dimensional array of Cells (used in other classes). I was in charge of implementing the topological sort, so I decided the best way to implement each Cell object as a vertex on a directed graph would be to store its in-degree as a field as well. Another field, current in-degree was added to store the in-degree of the cell during the topological sort so we wouldn’t lose track of the original in-degree. The edges of the directed graph were stored as linked lists of type Cell token (which houses cell addresses) in the dependencies field and the dependentCells field. The reasoning behind using two linked lists was for saving time during the topological sort.

The dependencies linked list stores the cell addresses of cells that need to be evaluated before the containing Cell object. All Cell objects referenced in the dependencies linked list have a vector protruding from them (tail belongs to the cells referenced in the dependencies linked list) and pointing to the containing Cell object (head of the vector at the containing Cell object). In a directed graph, there are never two vectors of a vertex that point to the same vertex. Therefore, I had to ensure that there were no duplicates and accessory methods were implemented to respect this property of the directed graph. I tested this by manually adding cell addresses to Cell objects. I noticed that the equals() method was not properly returning true if two Cell tokens (objects that store cell addresses) were the same. I worked around this problem by creating a custom isEquals() method in the CellToken class. There are also other accessory methods which assist in the updating of the dependencies linked list.

The dependentCells linked list contains the cell addresses of Cell objects who are dependent on the containing cell. This comes in handy during the topological sort, as you have to reduce the current in-degree of adjacent cells (those that are dependent on the cell that has been removed from the graph during the algorithm) by one. Without this list, when a vertex (Cell object) is removed from the directed graph, you would have to iterate through the respective dependencies linked lists of every cell on the grid to see if the cell being removed is found there. With the dependentCells linked list, you can directly go to vertices which are being pointed to by the node (without going through the entire set of Cell objects) being removed and decrement their current in-degrees by one during the topological sort algorithm.

Since users will modify the contents of a cell, it was imperative to implement the resetCell() method which resets the values of the cell. It was important to ensure that the dependentCells linked list was not altered during the reset as changing the formula for a particular cell object will never cause other Cell objects currently being pointed to by the that object to lose the vector connecting the two. The dependencies will however change. To test this method and all other remaining methods, I manually added and reset several Cell objects and printed their contents using the debugString() method to ensure that they did what they were supposed to. I checked to see if lists were being updated properly and that no duplicate references were ever created.

Spreadsheet.java

I decided to have this class store the underlying 2-dimensional array of Cell objects that the user would interact with. The constructor allows the class to be of any size. To update any changes to user-entered input, I added the updateCell() method which parses the stack created by the formula entered by the user and updates the dependencies linked list of the current cell and the dependentCells linked list of the corresponding dependencies. The first time this was tested, I noticed that I had forgotten to reset the cell prior to updating it. This caused the dependencies linked list to still contain references to cells it no longer needed. This was easily remedied by calling resetCell() from the Cell class before the parsing of the stack.

We wanted to print out a message if the graph was cyclic, so I decided to make the topological sort method, isCyclic(), return true if the new graph corresponding to the current 2-dimensional array of Cell objects contained a cycle. The topological sort algorithm is the standard one: search for Cell objects in the graph (array of Cell objects) and find ones with current in-degree equal to 0. Put these into a queue. After this first pass, remove elements from the queue and decrement their respective adjacent cell’s current in-degrees by one. The values for the cells are also calculated here. Continue this until all cells have been evaluated (checked this with a counter). To detect a cycle, I added another counter which tracks the number of cells that have been evaluated before the start of a new pass over of the array. If this counter is still equal to the counter which tracks the total number of elements that have already been evaluated, then we know that there are currently no cell objects with current in-degree 0 meaning that there is a cycle. The method restores the current in-degrees to their actual in-degrees after a cycle has been detected or after each cell has been evaluated.

When testing this, I noticed that my initial method was not restoring the current in-degrees of cells when a cycle was found. This was remedied by adding the loop to restore the current in-degrees before the return statement. I tested several cases including purposely creating a cycle, cases where there was no cycle, and entering in a differing combination of cycle and non-cycle inputs to make sure that the graph structure was being restored after each pass.