AG2411 – GIS Architecture and Algorithms
Takeshi Shirabe
Geoinformatics, KTH
2021-11-19

Exercise 3
Implementing map algebraic operations

This is the last exercise on raster data structures and algorithms. The **Layer** class you have developed so far has methods for reading, writing, and visualizing raster data. Now you will complete this class by adding to it methods for performing three map algebraic operations, LocalSum, FocalVariety, and ZonalMinimum.

PREPARATION

Stay in the same directory that you designated as your working directory in the previous exercises.

TASKS

Note that there is no self-study guide, step-by-step instruction, or sample code for this exercise. Come to the lab and use all kinds of learning resources including your teaching staff, reference books, and past handouts to get ideas on how to approach to this assignment.

1. Add to the **Layer** class three new methods called **localSum**, **focalVariety**, and **zonalMinium**. The **Layer** class is now structured as below.

	Layer
+	**name**: String
+	**nRows**: int
+	**nCols**: int
+	**origin**: double[]
+	**resolution**: double
+	**values**: double[]
+	**nullValule**: double
+	**Layer(String, String)**
+	**Layer (String, int, int, double[], double, double)**
+	**print(): void**
+	**save(String): void**
+	**toImage(): BufferedImage**
+	**toImage(double[]): BufferedImage**
+	**localSum(Layer, String): Layer**
+	**focalVariety(int, boolean, String): Layer**
+	**zonalMinimum(Layer, String): Layer**
–	**getNeighborhood(int, int, boolean): int[]**

YOU ARE ENCOURAGED TO IMPLEMENT ADDITIONAL MAP ALGEBRAIC OPERATIONS, which could be used in the term project!

<Methods>

- The second **Layer** method is another constructor. It takes six parameters to initialize the **name**, **nRows**, **nCols**, **origin**, **resolution**, and **nullValue** of a new Layer object. You will need this constructor to create an empty layer, which stores the output of a map algebraic operation.
- The **localSum** method takes as input a Layer and a String representing the name of an output layer, applies *LocalSum* on **this** Layer and the input Layer, and returns the resulting Layer as output.
- The **focalVariety** method takes as input an integer value representing the radius of each neighborhood, a boolean value indicating if the neighborhood's shape is a square (if true) or a circle (if false), and a String representing the name of an output layer, applies *FocalVariety* on **this** Layer using the specified neighborhood, and returns the resulting Layer as output.
- The **zonalMinimum** method takes as input a Layer and a String representing the name of an output layer, applies *ZonalMinimum* with **this** Layer as the value Layer and the input Layer as the zone Layer, and returns the resulting Layer as output.
- The **returnNeighborhood** method takes as input an integer value representing the index of a cell, an integer value representing the radius of that cell's neighborhood in terms of the number of cells, and a boolean value indicating if the neighborhood's shape is a square (if true) or a circle (if false), and returns as output an array of integer values representing the indices of the cells that belong to the neighborhood. This operation may be **private** and used only to facilitate the implementation of focal operations.
    - Note that you may consider a list (rather than an array) of Integer objects as output.
    - Note also that if you implement the **values** attribute as a 2D array, you may want this method to return an array (or list) of arrays of two integer values—the two values indicating the location of a neighbor cell.

Your code may look like…

```
public class Layer {

       //Attributes
       public String name;         // name of this layer
       public int nRows;           // number of rows
       public int nCols;           // number of columns
       public double[] origin = new double[2]; // x,y-coordinates of lower-left corner
       public double resolution;  // cell size
       public double [] values;   // raster data
       public double nullValue;   // value designated as "No data"

       //Constructors
       public Layer(String name, String inputFileName) {
              // has been implemented in the previous exercise
       }
       public Layer(String name, int nRows, int nCols, double[] origin,
                     double resolution, double nullValue) {

              // construct a new layer by assigning a value to each of its attributes
              this.name = outLayerName;  // on the left hand side are the attributes of
              this.nRows = nRows;        // the new layer;
              this.nCols = nCols;        // on the right hand side are the parameters.
              // to be continued...
       }
       //Other methods
       public void print(){
              // print this layer on the display
       }
       public void save(String outputFileName) {
              // save this layer as an ASCII file that can be imported to ArcGIS
       }
```

```
        public BufferedImage toImage() {
              // create the layer in grayscale
        }
        public BufferedImage toImage (double[]) {
              // visualize the layer by assigning random colors to values of interest
        }
        // This method is complete.
        public Layer localSum(Layer inLayer, String outLayerName){
              Layer outLayer = new Layer(outLayerName, nRows, nCols, origin,
              resolution, nullValue);
              for(int i=0; i<(nRows*nCols); i++){
                    outLayer.values[i] = values[i] + inLayer.values[i];
              }
              return outLayer;
        }
        public Layer focalVariety(int r, boolean IsSquare, String outLayerName) {
        }
        public Layer zonalMinimum(Layer zoneLayer, String outLayerName) {
        }
        private int[] getNeighborhood(int i, int r, boolean isSquare) {
        }
}
```

2. Compile the **Layer** class.

3. Create a class called **Ex03** that consists only of the **main** method that takes several arguments indicating the name of a map algebraic operation (e.g. localSum, focalVariety, or zonalSum), the name of an input file, the name of another input file (if necessary), the name of an output file, etc., applies the map algebraic operation on the input layer(s), visualize the output layer, and saves it in the output file.

Your code may look like…

```
public class Ex03 {
      public static void main(String[] args) {
            String operation = args[0];
            if (operation.equals("localSum")) {
                  Layer inLayer1 = new Layer("", args[1]);
                  Layer inLayer2 = new Layer("", args[2]);
                  Layer outLayer = inLayer1.localSum(inLayer2, "");

                  // save and visualize the output layer

            }
            else if (operation.equals("focalVariety")) {
                  // perform focalVariety and save & visualize the output layer
            }
            else if (operation.equals("zonalMinimum")) {
                  // perform zonalMinimum and save & visualize the output layer
            }
            else {
                  System.out.print(operation + " is not currently available.");
            }
      }
}
```

4. Compile and test the **Ex03** class.

HELPS

1. Using a HashMap

When you implement the **zonalMinimum** method, you may want to make a kind of table that records zones in one column and their minimum values in another column. **HashMap** class (in **java.util**) will help with this. The following code illustrates how to use a HashMap.

```
// Create a HashMap
HashMap<Integer, Layer> hm = new HashMap<Integer, Layer>();

// Suppose these are IDs of layers
int i1 = 1, i2 = 2, i3 = 3;

// Create Integer objects from int values
Integer i1Obj = new Integer(i1);
Integer i2Obj = new Integer(i2);
Integer i3Obj = new Integer(i3);

// Create Layer objects
Layer l1 = new Layer("development", "./data/development.txt");
Layer l2 = new Layer("hydrology", "./data/development.txt");
Layer l3 = new Layer("vegetation", "./data/vegetation.txt");

// Map Integer objects (as keys) to Layer objects (as values)
hm.put(i1Obj, l1);
hm.put(i2Obj, l2);
hm.put(i3Obj, l3);

// You can map an Integer to a Layer in a more efficient manner (which may look
// busy, though).
hm.put(new Integer(4), new Layer("elevation", "./data/elevation.txt"));

// Retrieve a Layer object
int i = 1;
Integer iObj = new Integer(i);
Layer l = hm.get(iObj);
System.out.println("Layer " + l.name + " has been retrieved.");

// This works, too
System.out.println("Layer " + hm.get(new Integer(2)).name + " has been
retrieved.");
```

2. Converting an index in 1D array to an index pair in 2D array.

In the instructor's design of the Layer class, the location of each cell is identified by a single integer. This should be fine in most cases, but you may want to know the row and column of each cell in order to find which cells are in a cell's neighborhood. If you have chosen to use a single array to store raster data, you need to <u>convert the index of each cell in the 1D array to the index pair (i.e. row index and column index) of that cell in the corresponding 2D array.</u>

Take a look at the layer illustrated below. Cell 15 has Row # 2 and Column # 3. Cell 18 has Row # 3 and Column # 0. Try to find a rule for the aforesaid conversion. If you find one, then can you tell the Row # and Column # of, say, Cell 269, assuming that the layer has more rows?

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	6	7	8	9	10	11
2	12	13	14	15	16	17
3	18	19	20	21	22	23
4	24	25	26	27	28	29

One idea is to divide the index (in the 1D array) by the number of columns (6 in the present example). For Cell 15, if you divide 15 by 6, you will get 2 as the *quotient* and 3 as the *remainder*. For Cell 18, if you divide 18 by 6, you will get 3 as the quotient and 0 as the remainder. Do you see some pattern?

In Java, this conversion can be done in the following manner.

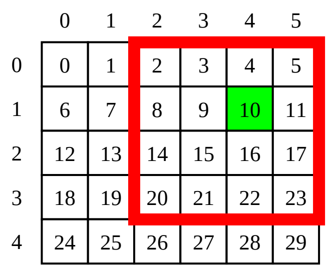
```java
// Convert i in 1D array to (r,c) in 2D array
int r = i/nCols; // the row number of cell i
int c = i%nCols; // the column number of cell i
```

% is called **modulo**.

3. Adding a rectangular group of cells to a list.

Perhaps the most difficult task in the implementation of a focal operation is to define the neighborhood of each cell, especially if that cell is located near the edge or corner of the layer. This difficulty may be overcome by a routine that adds a rectangular arrangement of cells to a list (**ArrayList** in Java).

As a concrete example, suppose that Cell 10 has a neighborhood defined by the rectangle drawn by a red line in the layer below. Note that we understand that this shape of a neighborhood may not be of your interest, as you may be interested only in square and circular shapes of neighborhoods.



	0	1	2	3	4	5
0	0	1	2	3	4	5
1	6	7	8	9	10	11
2	12	13	14	15	16	17
3	18	19	20	21	22	23
4	24	25	26	27	28	29

Then the following code adds those cells bounded by this rectangle to an ArrayList.

```
ArrayList<Integer> l = new ArrayList<Integer>();       // With <Integer>, only
int neighbor;                                          // Integer objects can be
Integer neighborObj;                                   // added to ArrayList.
for (int r = 0; r <= 3; r++) {
      for (int c = 2; c <= 5; c++) {
            neighbor = r*nCols + c; // converting back to the index in 1D array
            neighborObj = new Integer(neighbor);
            l.add(neighborObj);
      }
}
```

4. Converting a list into an array

If necessary, you can convert a list into an array by iterating through the list. For example, the following code converts a list of Integer objects to an array of them.

```
// Create a list of Integer objects
ArrayList<Integer> intObjList = new ArrayList<>();
Integer intObj1 = new Integer(1);
Integer intObj2 = new Integer(2);
Integer intObj3 = new Integer(3);
intObjList.add(intObj1);
intObjList.add(intObj2);
intObjList.add(intObj3);


// Convert the list to an array
Integer[] intObjArray= new Integer[intObjList.size()];
int counter = 0;
for (Integer intObj: intObjList) {
      intObjArray[counter] = intObj;
      counter++;
}
```

If you think the above code looks too much for such a simple task, you may appreciate one of the methods of **ArrayList** called **toArray**().

```
// Alternatively, the above conversion can be done by the following method.
Integer[] intObjArray= new Integer[intObjList.size()];
intObjArray = intObjList.toArray(intObjArray);
```

The toArray() method works only if both the list and the array consist of objects of the same class. Otherwise, you need to take the first, more involved approach. For example, the following code converts a list of Integer objects to an array of integer values.

```
// Convert the list to an array of integer values
Integer[] intArray= new Integer[intObjList.size()];
int counter = 0;
for (Integer intObj: intObjList) {
      intArray[counter] = intObj.intValue();
      counter++;
}
```

SUBMISSION

Submit the following.

- Source codes of **Layer** class and of **Ex03** class.

DUE DATE

To avoid penalty points, your submission must be made through **CANVAS** no later than **11:59 PM, November 25, 2021.**