Home
Syllabus
Lecture Notes
Assignments
Project 1

David Galles
Computer Science
Univerisity of San Francisco

Programming Assignment 1: Sparse Arrays and the Game of Life

- Sparse Array Competion Due Date: 2/23/2015
- Final Due Date: 2/27/2015

Large Matrices

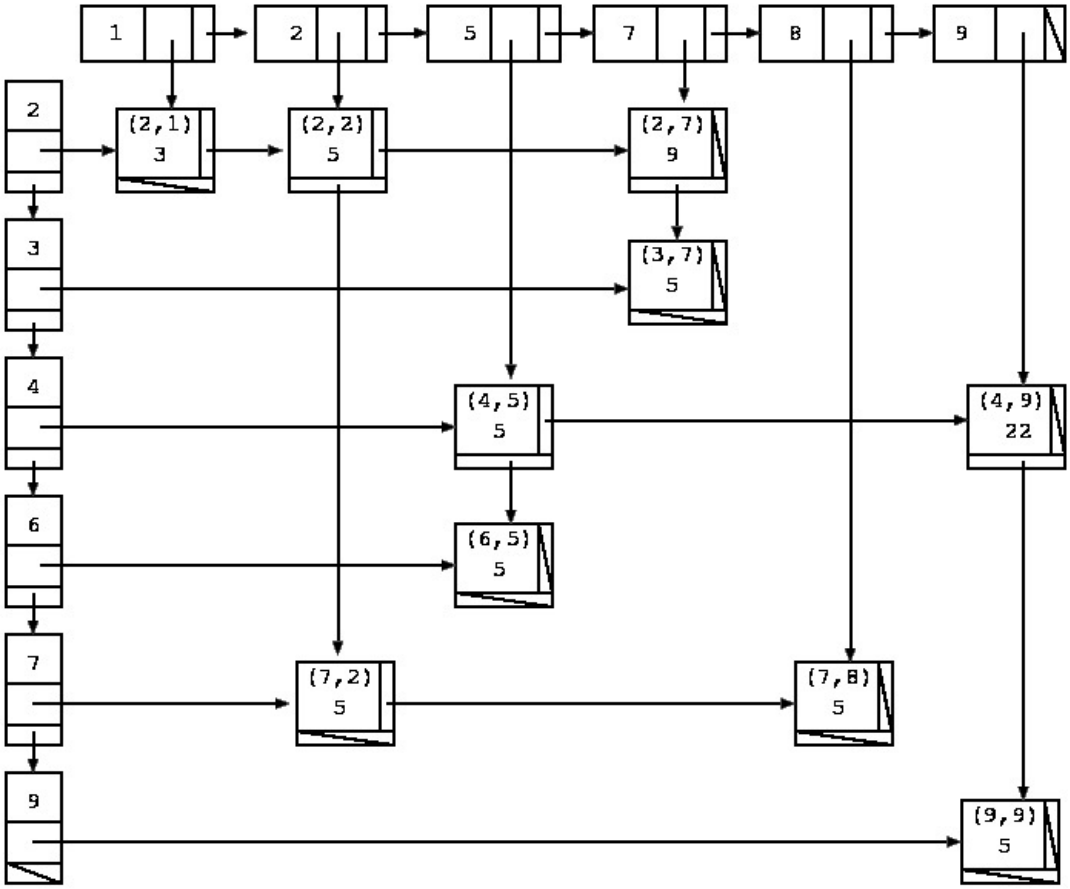
Consider representing a very large 2D matrix of values, where most of the values are 0. The most straightforward representation would be a very large two dimensional array of integers. This straightforward approach has a few limitations however. If the matrix we are reperesenting is very large, it requires a large quantity of memory to store, even if the number of non-zero elements is small. A better way of representing a sparse martrix would be to only store the values that are non-zero, in such a way that iterating through the rows or columns of non-empty elements is relatively fast.

Sparse Arrays

An alternate representation for a very large 2D matrix of integers is a sparse array Instead of a 2D array (which is just an array of arrays), we will store list of lists, and only include non-zero elements in our lists. Since we are no longer storing every element, we will also need to store a row/column index for each element in our list. Also, so that we can traverse the matrix in a row-major or column-major fashion, we will actually have two lists-of-lists, which share elements. This is easiest to see with an example. Consider the following 2D marix, which contains only 10 non-zero elements, and 90 zero valued elements:

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	3	5	0	0	0	0	9	0	0
0	0	0	0	0	0	0	5	0	0
0	0	0	0	0	5	0	0	0	22
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	5	0	0	0	0
0	0	5	0	0	0	0	0	5	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	5

We can represent this matrix with a sparse array, as follows:



For your first project, you will implement sparse arrays.

Implementation

You will implement the following interfaces:
SparseArray.java

```
public interface SparseArray
{
   public Object defaultValue();
   public RowIterator iterateRows();
   public ColumnIterator iterateColumns();
   public Object elementAt(int row, int col);
   public void setValue(int row, int col, Object value);
}
```

Where:

- **defaultValue()**: Returns the default value for the sparse array. In our integer array version above, this would be 0. Note that it is an object, so we could really have any default value that we wanted. The default value for your sparse array should be set by the constructor
- **iterateRows()**: Returns an iterator that can be used to iterate through the rows of the array. More on this below.
- **iterateColumns()**: Returns an iterator that can be used to iterate through the columns of the array. More on this below.
- **elementAt(int row, int col)** Returns the object stored at (row,col), if such an element exists, or the default value, if not such element exists.
- **setValue(int row, int col, Object value)** Sets the value of the matrix

at position (row,col), adding new linked list element(s) as necessary. For instance, the first time you add an element to the sparse array, you will need to add 3 linked list elements -- one for the element itself, one for the row in which it is added, and one for the column in which it is added. Note that if you use setValue to set the value to the defaultValue (as determed by the .equals method), you should *remove*the element from your lists. Only non-default values should be stored.

To iterate through the sparse array, you can either iterate through the rows or the columns, giving you a RowIterator or ColumnIterator:

RowIterator.java

```
abstract class RowIterator implements java.util.Iterator
{
   public abstract ElemIterator next();
   public abstract boolean hasNext();
   public void remove()
   {
      throw new UnsupportedOperationException();
   }
}
```

ColumnIterator.java

```
abstract class ColumnIterator implements java.util.Iterator
{
   public abstract ElemIterator next();
   public abstract boolean hasNext();
   public void remove()
   {
      throw new UnsupportedOperationException();
   }
}
```

For either of these iterators, a call to **next()** does not return an element, but another iterator, that allows you to traverse every element in that row or column:

ElemIterator.java

```
abstract class ElemIterator implements java.util.Iterator
{
   public abstract boolean iteratingRow();
   public abstract boolean iteratingCol();
   public abstract int nonIteratingIndex();
   public abstract MatrixElem next();
   public abstract boolean hasNext();
   public void remove()
   {
      throw new UnsupportedOperationException();
   }
}
```

- **iteratingRow()** Returns true if this iterator is iterating through a row (that is, if this iterator was obtained from a call to next from a ColumnIterator)

- **iteratingColumn()** Returns true if this iterator is iterating through a column (that is, if this iterator was obtained from a call to next from a RowIterator)
- **nonIteratingIndex()** If we are iterating through a row, return the index of the row we are traversing. If we are iterating through a column, return the index of the column we are traversing
- **next()** Returns the next element in the row (or column) we are traversing
- **hasNext()** Returns true if there are more elements in this row/column
- **remove()** Not supported. (We have to include it, since we implement java.util.Iterator. We'll just throw an exception.)

Finally, the element returned by the iterator is:

MatrixElem.java

```
public interface MatrixElem
{
   public abstract int rowIndex();
   public abstract int columnIndex();
   public abstract Object value();
}
```

Example Use

Once we have created a sparse array, we could print out all of the non-default values in the array as follows:

```
SparseArray a;

//  Create a new sparse array, fill with values

RowIterator r = s.iterateRows();
while (r.hasNext())
{
   ElemIterator elmItr = r.next();
   while (elmItr.hasNext())
   {
      MatrixElem me = elmItr.next();
      System.out.print("row:" + me.rowIndex() +
                       "col:" + me.columnIndex() +
                       "val:" + me.value() + " ");
   }
   System.out.println();
}
```

Implementation Variations

As long as you correclty implement the interface, you have some lattitude as to how you manage your data structures. You can, for instance, use doubly-linked lists and dummy elements to make your coding easier, if you prefer.

Test Files

We have provided the file TestSparseArray.java to help you test your Sparse Array implementation.

Assignment

For your first assignemnt, you will

- Implement a class named **MySparseArray** that implements the **SparseArray** interface. The constructor for your sparse array should take as an input parameter the default element for the sparse array.
- Create a class called **Life** which contains a main program that uses **MySparseArray** to play the game of life.

The Game of Life

The ``Game of Life'' is a cellular automaton, consisting of a two-dimensional grid of cells. Each cell is either alive or dead. Cells either die, stay alive, or are born using the following rules:

- For cells that are alive:
  - If the cell has 0 or 1 neighbors, it dies from loneliness
  - If the cell has 2 or 3 neighbors, it lives on to the next round
  - If the cell has 4 or more neighbors, it dies from overpopuation
- For cells that are not alive
  - If the cell has 3 neighbors, a new cell is born in this location
  - If the cell has eithr more than 3 or fewer than 3 neighbors, it remains dead

The game is ``played'' in generatrions. An initial setup says which cells are alive. The rules specify which cells will exists in the next generation. All cells ``off the gird'' (that is, with a row or column of -1) are perminantly ``dead''. For more details (and a cool ~~applet~~ javascript / canvas application), see: pmav.eu/stuff/javascript-game-of-life-v3.1.1/

Command Line Parameters and File Format

Your program should take three command line parameters: the filename for the initial conditions, the filename to ouput, and the number of generations to simulate. Your program should read in the initial conditions from the specified file, simulate for the specified number of generations, and then write the result to the output file. The format for the input and output files are the same (so that you could use an output file as an input file for further testing)

File Format:

Life files consist of a list of zero or more row,column positions of live cells. A comma should separate the row and column, and a newline should separate each live cell. The list should be sorted first by rows, and then by columns. For instance, a file represeting live cells at locations (1, 4), (100, 43), (10, 8), and (10, 4) would be:

```
1,4
10,4
10,8
100,43
```

For example, if your input file was:

```
100,100
100,101
100,102
100,103
100,104
```

and you ran the simulation for 4 generations, the output file would be:

```
98,101
98,102
98,103
99,100
99,104
100,100
100,104
101,100
101,104
102,101
102,102
102,103
```

Note that even if you do not use all of the functionality in the interface for your game of life, you still must implement the entire SparseArray interface properly. We will be testing both the sparse matrix itself and the game of life!

Life hints

Probably the easiest way to code the game of life is to maintain 3 sparse arrays:

- Current Generation
- Number of neighbors
- Next Generation

Load the initial contitions into the ``Current Generation'', and then repeatedly:

- Count the neighbors in the ``Current Generation'' to get the ``Number of Neighbors''
- Use the ``Number of Neighbors'' and the ``Current Generation'' to get the ``Next Generation''
  Replace the ``Current Generation'' with the ``Next Generation''

Due Dates

Your sparse array class should be checked into subversion by Monday, Feb. 23th. All files required for Life should be checked into subversion by Friday, Feb. 27th.

Submission

Submit your files using subversion. Your files should be stored in the subversion In fact, I recommmend that you don't wait until you program is done to get it into subversion, start right away to protect yourself. The subverion directory you should use for this proect is

https://www.cs.usfca.edu/svn/*username*/cs245/project1/, where *username* is your cs username. You need to submit **all** files necessary to run your project, including the files that are provided.

Collaboration

It is OK for you to discuss solutions to this program with your classmates. However, **no** collaboration should **ever** involve looking at one of your classmate's source programs! It is usually extremely easy to determine that someone has copied a program, even when the individual doing the copying has changed identifier names and comments.

Provided Files

The following files are provided. Note that you need to use the interfaces as they are given, so that you program will work correctly with the testing code that we will provide.

- SparseArray.java Your Class MySparseArray needs to implement this interface
- RowIterator.java
- ColumnIterator.java
- ElemIterator.java
- MatrixElem.java
- TestSparseArray.java Some testing code for your SparseArray