2D Bit Arrays and Conway's Game of Life

Alex Fischer

03 October 2016

1 How Java Stores Boolean Arrays

Say that you want to store an array of `boolean`s in Java. How does Java store that array? Since a `boolean` requires only 1 bit of information to store its state, you might guess that Java would allocate n bits of memory to store an array of size n. When retrieving the ith element of the array, Java would just retrieve the ith bit of the memory that it allocated.

However logical that might seem, Java does not in fact store `boolean` arrays that way. Although it is implementation specific, the JVM you are using probably uses eight bits, or one byte for each `boolean` in the array. Why does Java do this? It turns out that it is faster to store them in this way because when processors retrieve data from memory, they cannot retrieve an individual bit from memory. Depending on the processor, they can retrieve in one instruction anywhere from one byte to eight bytes on a 64-bit processor. They cannot retrieve one bit from memory, but anywhere from eight to 64 at a time.

There is another inefficiency in the way Java stores arrays. Say you have a two-dimensional $n \times m$ array. You might think that Java would allocate one block of memory to store that array, and that the block of memory would be of size $m \times n \times s$, where s is the size of the individual units we are storing in the array. However, that is not what Java does. Java instead allocates memory for an array of n object references, and each of those references will point to an array that stores m of the individual units. This configuration allows one to use variable-width arrays, where one row of the array has a different width than another row. However, if we are using fixed-width two dimensional arrays, it is slower, because we have to manage multiple possibly separate blocks of memory instead of just one block of memory.

2 Memory Wasted

Although it is faster to store a `boolean` in one byte, it does waste memory. Exactly how much memory is wasted? If we are storing a large number of bits in our array, then we can ignore small constant memory usage intrinsic to objects or arrays and look at just the data being stored. For enough bits to fill a 1000×1000 grid (which we just might do in this project), Java will allocate at least $1000 \times 1000 \times 8$ bits for a standard two-dimensional array of `boolean`s. For out ideal bit array, we will only use 1000×1000 bits. That means we will save seven million bits, or slightly less than one megabyte. That's not a ton of

memory saved, but if we were using, say $1,000,000 \times 1,000,000$ bits (something we will not be doing in this project), then it could make a difference.

3 A Solution

To store bits more efficiently, we will create a class called `BitArray2d`. Part of this class is already implemented, but you will implement the rest. The way this class works is that it stores all of its bits in a `long[]` array, which puts it all in one accessible block of memory. Say you have a 20×20 array of bits that you want to store efficiently. You will put each 20 bit row in your array in order into the `long[]` array, wasting no memory in the process.

To illustrate this process, let's work with our hypothetical 20×20 bit array. We will store the first row of bits (ie, from `bitArray.get(0,0)` to `bitArray.get(19,0)`) in the first 20 bits of our long array's first long, or `longarray[0]`. The same will be true of the second row of the array (ie, from `bitArray.get(0,1)` to `bitArray.get(19,1)`). However, the third row of the bit array (ie, from `bitArray.get(0,2)` to `bitArray.get(19,2)`) will be stored in both `longarray[0]` and `longarray[1]`, as `longarray[0]` only has 64 bits. Bit arithmetic is used to store and retrieve the bits from the array.

4 Conway's Game of Life

Conway's Game of Life is one interesting way we can use bit arrays. The "game" is a zero player game that involves squares on a board evolving according to a set of rules. Many interesting patterns can evolve in this game depending on the initial conditions, some of which turn out to be quite aesthetically pleasing, and some of which turn out to be highly chaotic. For a fuller description of this game, see the Wikipedia page about this topic.

The game board consists of an infinite (although for practical purposes, we will limit it in size) square grid of cells, each of which can be alive or dead. When the board evolves into its next state, a live cell stays alive if and only if it has two or three live neighbors, else it dies. A dead cell becomes alive if it has three live neighbors, else it stays dead. "Playing" the game consists of continuously evolving the board. The board is typically visualized with dead cells being drawn as white squares, and live cells as black squares.

5 Methods You Will Implement

You will be completing the `BitArray2d` class in the `automata.conwaysgameoflife.src` package. There are four methods that you will have to complete. You will have to use bit arithmetic to complete some of these methods.

5.1 Constructor

This method should just store given dimensions of the array and allocate the necessary memory to store the bits by initializing the `bits` array.

5.2 Get

This method should retrieve the given bit from the `bits` array and return it as a `boolean`. It should also throw an `ArrayIndexOutOfBoundsException` if the point that was asked for is out of bounds.

5.3 Set

This method should set the given bit to the given value by modifying the `bits` array. It should also throw an `ArrayIndexOutOfBoundsException` if the point that was asked for is out of bounds.

5.4 Evolve

This method should evolve the first `BitArray2d` onto the second `BitArray2d` using the specified rules for Conway's Game of Life. It should not modify the first `BitArray2d` in any way.

To deal with the edge conditions, where the edge cells evolve, I just assume that the edge cells are always 0 and only evolve the cells that are on the inside. You can deal with the edge conditions another way if you'd like.

6 Testing

I have included two JUnit test files in the package `automata.conwaysgameoflife.test`. One tests only the `get` and `set` methods of the `BitArray2d`, while the other tests the `evolve` method.

Once all of those tests pass, run the `Main.java` class in the `automata.conwaysgameoflife.support` package. Once that runs, press Start, and you should see a collection of black cells changing shape and moving across the board slowly. Click around the moving object to turn some cells into live cells and see if you can get some strange patterns going. Or, pause the game, draw your own pattern, and start it again to see even stranger patterns evolve.