Program 4: Bloomin’ ’Mazement

Owen Kaser

November 9, 2019

Due: Dec. 6, 11:59 pm. May not be submitted late.

1 Introduction

In this program, you will go back to the maze-generation problem and work with $40 \times 40$ mazes. In that previous program, you did not actually print out the generated mazes. This time, you want to do this, and you will use a number of different mechanisms, including

- 2-D array(s) of booleans
- `java.util.BitSet`
- `java.util.HashSet`
- a Bloom filter (see below)
- a direct approach that “erases” things in graphics (this has been done for you)

Except for the last (direct) approach, you will record the set of walls removed during the maze generation process. When it is finished, you will draw a wall at every possible position, except for positions contained in the set of removed walls. The different alternatives (BitSets, HashSets, 2-D arrays of booleans, Bloom filters) offer different approaches to implementing the set of removed walls.

Write your program so that it takes a command-line argument, a value from 1 to 4, which specifies the set implementation to be used. (1 means arrays of booleans, 2 means `BitSet`, 3 means `HashSet` and 4 means to use a Bloom filter.)

These mechanisms will not be taught in lectures, though an upcoming tutorial will probably deal with `BitSet` and `HashSet`. It does not make sense to learn about the details of Bloom filters until we have covered textbook Section 3.4.

A Bloom filter provides an approximate set. Like any `java.util.Set` you can add new items to it, and you can check whether the set contains a given item. An ordinary set implementation, like `java.util.HashSet`, gives you an answer that is always correct. A Bloom filter gives you an answer that has some
probability of being wrong. But it is a “1-sided error”: If the Bloom filter says that an element is not in the set, it is never wrong. If it says the item is in the set, it probably is (but might not be). “No means no, and yes means . . . maybe”. There is a trade-off between the amount of memory that you let a Bloom filter use and the probability that its “yes” answers are incorrect. Bloom filters that have a fairly small probability of error can use much less memory than a comparable exact set implementation, such as a `HashSet`.

The errors produced by your Bloom filters will cause you to erase a few walls that you should not. We want to answer a question about the effect of this — on average, how much shorter is the path with the Bloom filter’s accidentally erased walls than the path that does not use the accidentally erased walls?

2 Details

Some details follow on displaying a maze, implementing your sets of removed walls, finding an actual path that solves your maze, and doing the experiments about the effect of the Bloom filter’s errors.

2.1 Displaying a Maze

We want to display a maze both as a graphical image and as an old-school piece of ASCII art (e.g., [https://www.asciiart.eu/mythology/dragons](https://www.asciiart.eu/mythology/dragons)).

My sample solution, scaled down to a smaller grid, gives the screen shot in Figure 1.

![Screenshot of maze, graphics and ASCII “graphics”](image)

Figure 1: Screenshot of maze, graphics and ASCII “graphics”

Graphically: (This was done for you already.) First make a grid (like graph paper) by drawing a lot of horizontal and vertical line segments, say in blue on a white background. Then, when the maze generator has decided to knock down
a wall between two cells, draw a small white segment on top of the existing grid, effectively erasing the blue.

I used the textbook’s StdDraw (see pages 42–44), which seemed easier than using the standard Java graphics libraries.

ASCII Art: To draw a maze with ordinary text characters, we first have to record the set of walls that the maze generator has knocked down between the cells in the maze. Then, for each row of cells in the maze, you will print out two rows of characters.

Represent each cell in the maze as a $3 \times 2$ grid of characters that shows the “walls” present on the south and east side of the cell. (We don’t worry about the north wall, because it is handled as the south wall for the neighbouring cell above us. Likewise for the west wall.)

The rules for what to draw are technically known as “hairy” (and will probably result in a horror of nested if statements when coded).

- For a cell that still has walls on both the south and east sides, draw

  \[
  \text{SS|} \\
  \text{---+}
  \]

  (Here, we show spaces explicitly with S. Your program should show spaces as blanks.)

- A cell with a wall on the east side but not on the south can be drawn as

  \[
  \text{SS|} \\
  \text{SS|}
  \]

  or

  \[
  \text{SS|} \\
  \text{SS+}
  \]

  depending whether the cell east of this one has a wall to its south.

- A cell with a wall on the south but not on the east can be drawn as

  \[
  \text{SSS} \\
  \text{---}
  \]

  or as

  \[
  \text{SSS} \\
  \text{---+}
  \]
depending on whether the cell south of this one has a wall to its east.

• A cell with neither a south nor an east wall can be drawn as

SSS

SSS

These rules are “good enough” rather than perfect. In particular, they result in a bit of gap in the northwest corners of walls, sometime. If you want to play around and get something prettier, go ahead. I have the impression that using underscore characters rather than minus signs might mean that we don’t need to use plus signs.

You will need to draw a top and left boundary for the entire grid using | and - characters.

2.2 Sets to Replace 2D Arrays

Suppose you want to record that a cell at position \((x, y)\) is in some set. With a 2D array of booleans, you could write something like

\[
isInSet[x][y] = true
\]

Storing things like a pair of values \((x \text{ and } y)\) into a Set or BitSet may seem to be a problem, because Set requires a single object (not two), and BitSet is even more restricted: it can only store a set of integers.

However, you can turn a cell’s coordinates into a single string or a single integer with a little work: You can do much of the same thing as earlier (with some slowdown) using a `Set<String>`: use something like `mySet.add(x+";"+y)` The code for generating mazes already has to handle the fact the Union/Find data structure can only work with single integers (see `xyToId()`).

2.3 Bloom Filters

Refer to [https://llimllib.github.io/bloomfilter-tutorial/](https://llimllib.github.io/bloomfilter-tutorial/) for an introduction to Bloom filters. Your Bloom filters should implement approximate sets of strings. Your program should use \(k = 2\) and should have a false-positive rate of about 1% when an average number of walls have been knocked out. Use an underlying `BitSet` to record the hash codes that have been seen.

The noted page gives you a formula to approximate the false-positive rate, assuming you know the number of elements in the set. This number will vary some between different runs of the maze-generation program, but you should have average numbers from Program 2. (Or if you didn’t get these numbers from Program 2, a classmate’s Program 2 numbers can be used.)

One of your two hash functions can be `String.hashCode()`.

You need at least 4 JUnit tests for your Bloom filters. In particular, one of the tests should confirm that in 10 attempts, at least 5 result in a false-positive rate between 0.5% and 2.0%, when the number of items in the set is the expected
number of walls removed and you choose the same size table as when building the maze.

Other tests should confirm that when the filter says “No”, it is always correct.

2.4 Solving a Maze

Once we get to Section 4.1 of the textbook, you will learn about undirected graphs and an algorithm called breadth-first search. After you know about these things, you can build a graph from the maze and start a breadth-first search at the maze entrance. You should be able to use BreadthFirstPaths.pathTo() (see page 540) to record the set of cells on the shortest path that solves the maze. Use this to improve your drawings of the maze (both the graphical one and the ASCII-Art one, inserting dots that mark the cells that are on the path through the maze.

2.5 False Positive Effects

When the Bloom filter option (#4) is chosen from the command line, not only must you generate and display one maze, but you also do some computational experiments on 100 more mazes (that you don’t display) to determine the average effect of the false positives on the length of the path that solves the maze. For a particular maze, compute the length of the shortest path that solves the maze that we intended to generate (using an error-free Set). Then compute the length of the shortest path that solves that same maze, as it would be output when you use a Bloom filter (with its \(\approx 1\%\) false positives removing extra walls). What is their ratio? Get 100 ratios from 100 random mazes, and report on the average ratio. (Use the usual “arithmetic mean”. For ratios it is often perilous, but it feels intuitively ok to me here. Your statistics professor would know whether the requested calculation is actually sensible, but do it regardless.)

Put your result as a comment at the bottom of your main source-code file.

3 More Requirements

Use the Maven skeleton given in the GitHub classroom link. The grader should be able to type “mvn package” to build your software. Your README file should say what he then needs to do, in order to run your program.

As usual, while you are actively coding or debugging, you must do a GitHub commit and push every 15 minutes or so. Don’t forget to git add any new .java files you create while you work.

\(^1\)This is risky with probability. For example, see the Wikipedia article on Simpson’s paradox.
4 Grading

<table>
<thead>
<tr>
<th>Item</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text drawing with removed-walls set</td>
<td>25</td>
</tr>
<tr>
<td>Sets are 2D boolean arrays</td>
<td>5</td>
</tr>
<tr>
<td>Sets are BitSet</td>
<td>5</td>
</tr>
<tr>
<td>Sets are HashSet</td>
<td>5</td>
</tr>
<tr>
<td>Sets are Bloom filters</td>
<td>20</td>
</tr>
<tr>
<td>Shortest path maze solution</td>
<td>20</td>
</tr>
<tr>
<td>Results of errors on path lengths</td>
<td>5</td>
</tr>
<tr>
<td>JUnit tests</td>
<td>15</td>
</tr>
</tbody>
</table>