CS1632 – Deliverable 5

Performance Testing Conway’s Game of Life

Philip Ni

CS1632 Software Quality Assurance

University of Pittsburgh

Repository: https://github.com/phil-nye/CS1632/tree/master/Deliverable5-JavaLife

# Summary

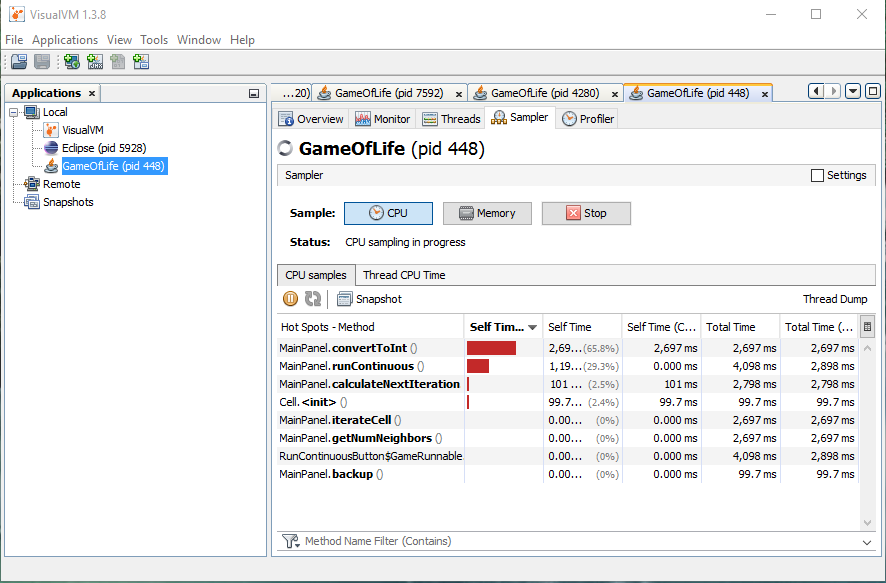
The purpose of this project was to determine trouble spots in a version of Conway’s Game of Life, and then make it run more efficiently. To verify and validate my changes, I wrote pinning tests to ensure that functionality did not change; only performance should change. I changed the names of the original methods and kept the in the code so that I could test them concurrently with the more performant version I modified.

To begin, I ran the original program files with VirtualVM in the background to determine exactly what functions and classes were the most CPU-intensive. Running the program step-by-step allowed me to push all buttons (sans the “Run Continuous” one). Doing this, I immediately identified that the Cell.toString() method was taking an unusually long time to run, followed by MainPanel.convertToInt(). Next, running continuously, I was able to confirm that MainPanel.convertToInt() was the bigger issue when running through many steps of the game.

Looking into the MainPanel.convertToInt() method, I found that it was called by the getNumNeighbors() method. In fact, getNumNeighbors() already calculates the number of adjacent living cells as a primitive int. So, the entire convertToInt() method is redundant. In fact, convertToInt() just makes the entire program take longer to run because all it does is loop unnecessarily, just to return an int as an int. The input and outputs are exactly the same. For the refactored version, I simply returned numNeighbors. After running the performance profiler again, I found that MainPanel’s runContinuous() method was now the least performant.

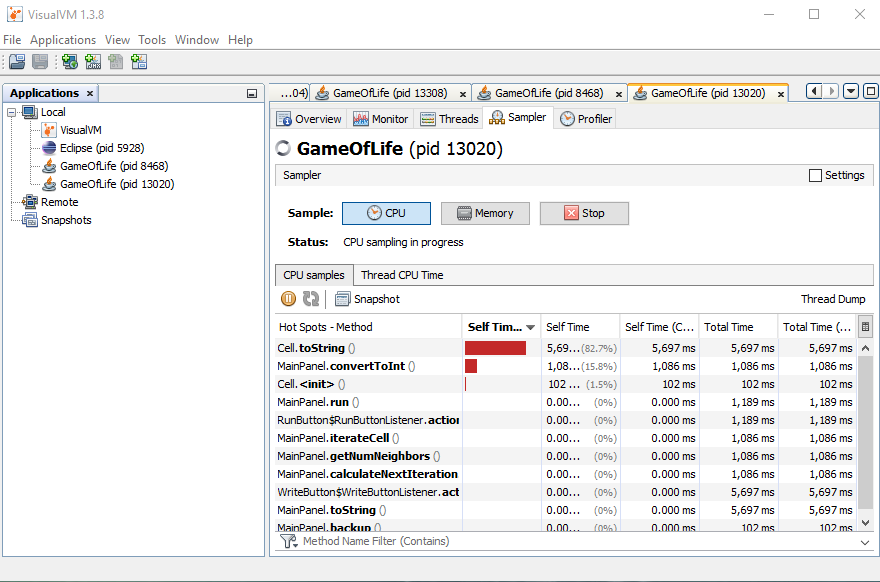
The runContinuous() method contains a for loop that simply takes the value of “\_r” (1000) and calculates a remainder ten thousand times or “\_maxCount” times. When the loop is complete, the value of “\_r” is just set back to its original value and never used again. Clearly, these lines are extraneous and does not do anything to compute the next iterations of the game. In the more performant version, I simply removed these lines.

Finally, Cell.toString() took the next longest amount of time, after modifying the MainPanel class. I found that the original toString() method contains a for loop that continues to append the same string to the end of the current string. This loop is extraneous because the if statement contains a substring() call that just takes the first character out of the string again. To make this method more performant, I simply removed the for loop to eliminate the extra work. After this, the substring call is no longer needed. All you have to do is get the text directly from the GUI state by using getText().



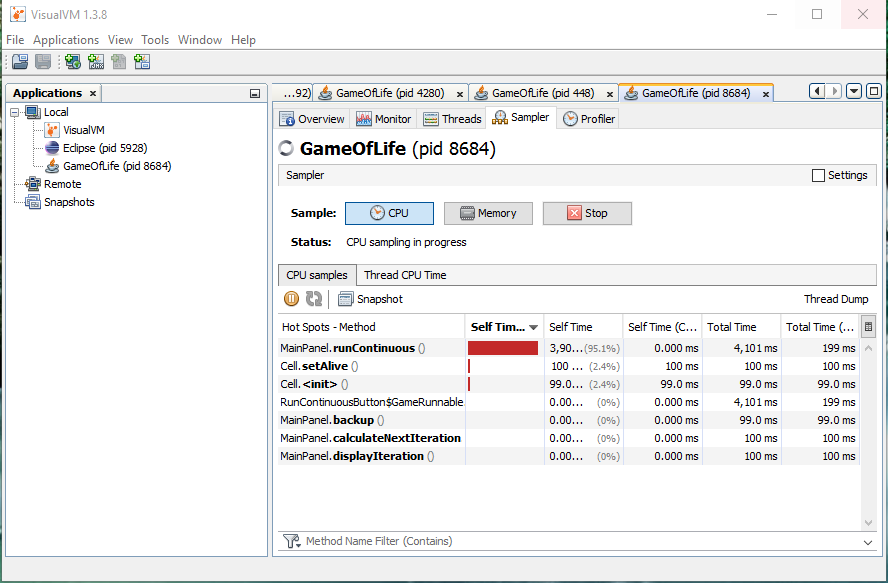
Profiler Results BEFORE Modification

(15x15 map; continuous; 5 sec run)



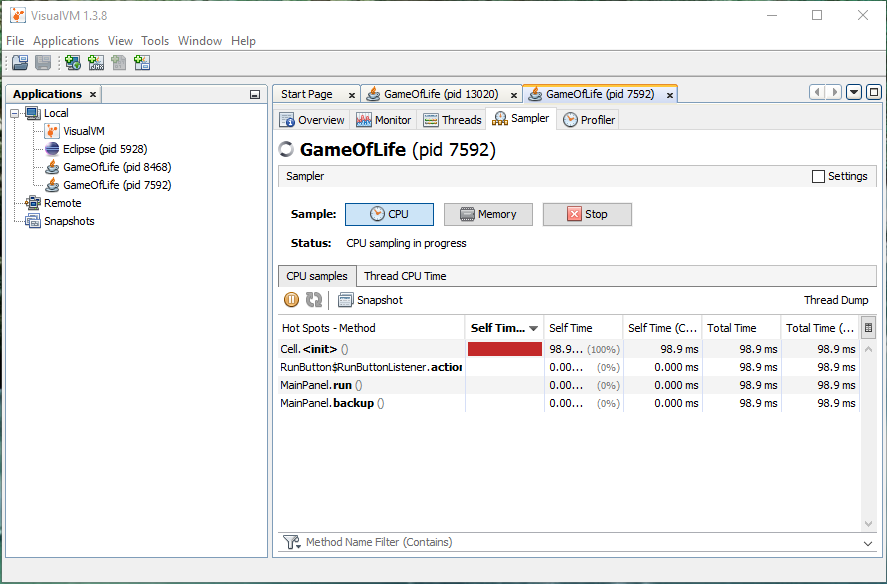
Profiler Results BEFORE Modification

(15x15 map; running step-by-step)



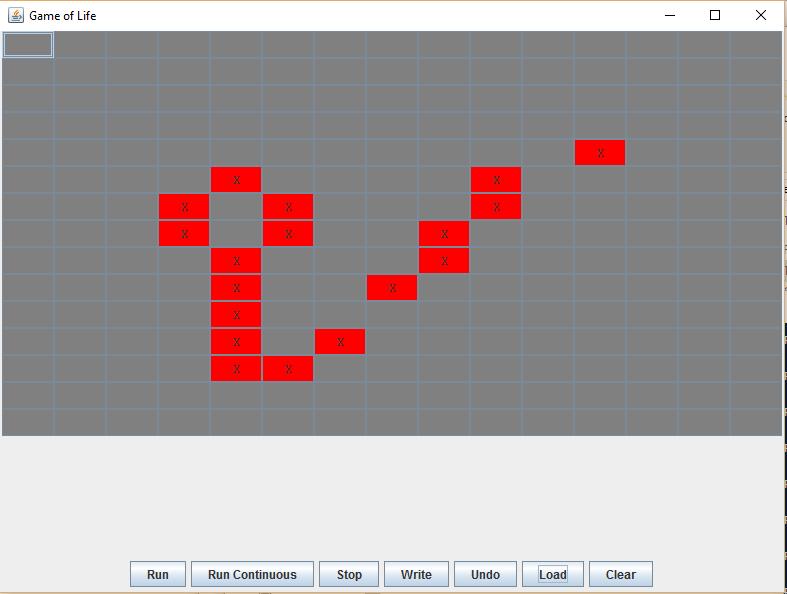
Profiler Results AFTER Modification

(15x15 map; continuous; 5 sec run)

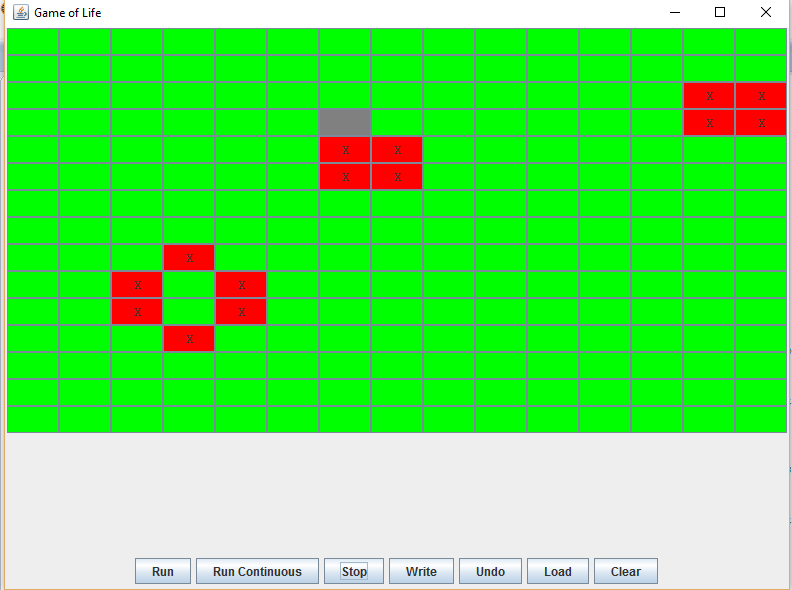


Profiler Results AFTER Modification

(15x15 map; running step-by-step)

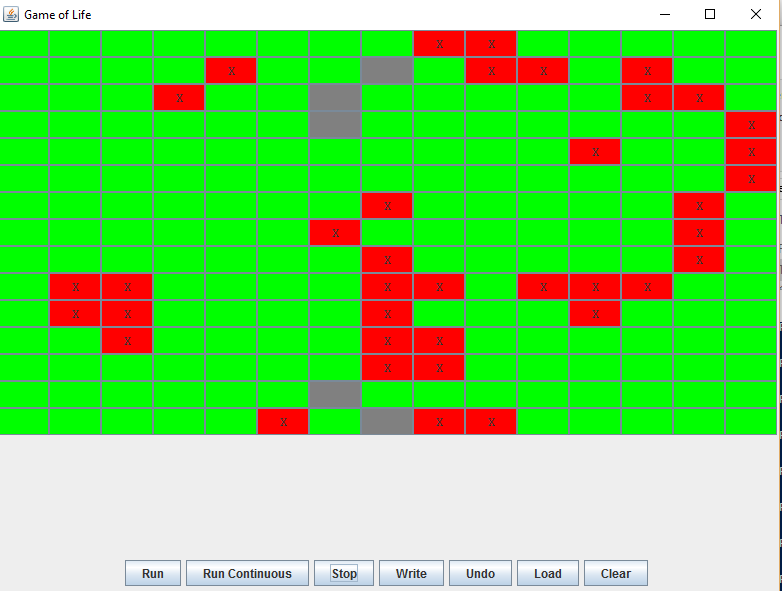


Original Game State Start



Game State AFTER Modification

(15x15 map; continuous; 5 sec run; steady state)



Game State BEFORE Modification

(15x15 map; continuous; 5 sec run; still running)

