University of Pittsburgh

Boggle Solver

CS 1699 – Final Deliverable
Performance Testing a Java Application

John Abraham & Yibo Cui
Bill Laboon
4/14/2015

TABLE OF CONTENTS

1. SUMMARY AND TESTING CONCERNS:

For this deliverable we decided to run a performance test on a Java program that we previously wrote for our Data Structures class. The program is called "Boggle Solver." Boggle is a word game where players race to find words hidden in a 4x4 (or larger) grid of letters. What this program does, however, is solve the boggle board for the user instead of the letting him/her play the game and find the words themselves. In the original version of the game, the program is given a dictionary of 172,822 words and a grid of letters; then, it loads the grid into a 2D array and loads the dictionary into an ArrayList. Then, using recursive backtracking it generates all the possible combinations of letters in the grid, and each combination gets compared against the ArrayList of all the dictionary words. On the first several runs, we noticed that as the grid size got bigger the execution time became slower and slower exponentially and not linearly. So we used Java's VisualVM profiler to determine which method was the most resource intensive in terms of CPU usage (i.e. CPU "hot-spot") in order to know what part of the code to refactor to potentially make the program run faster and in a more linear fashion.

The main focus was to measure and improve response time and speed of execution rather than availability. Response time measure how quickly the system responds to user input. If the user inputs large grid, 100x100 for example, they don't want to sit around waiting for half an hour or hours to see the results. Optimizing response time is challenging. We had to use data structure analysis and algorithmic analysis on what code we already had.

Section 2 of this report shows the VisualVM profiler results that we used to determine which method to refactor.

Section 3 shows what we did to optimize the original code.

Section 4 gives an overall assessment of quality of our refactored code, a simple graph showing a response time comparison based on different grid size inputs, and it shows how confident we feel users would be satisfied with the changes we made.

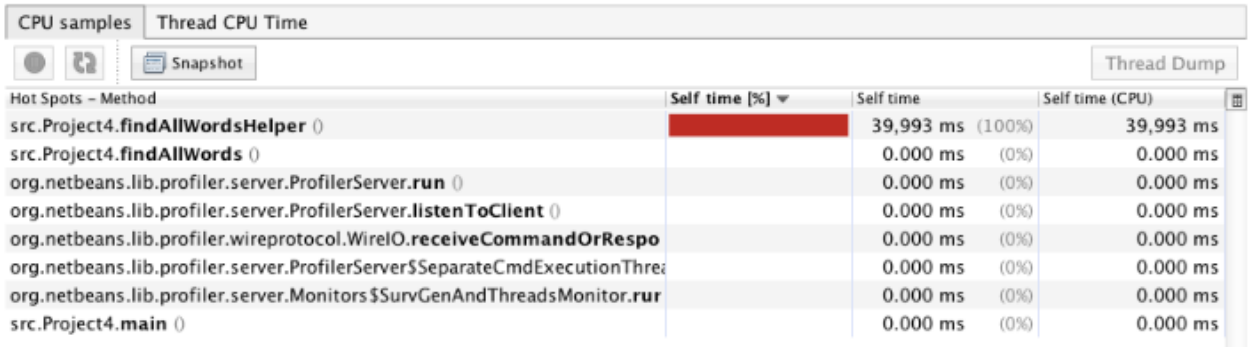
Section 5 shows the results of the executed JUnit tests that we performed to ensure consistency between between the outputs of the the original program and the refactored program.

Some concerns that we have going further is weighing speed efficiency against memory efficiency. In this particular exercise our refactored code gave better response time and better memory efficiency because we changed a data structure that we were using to store a relatively large dictionary; however, in other cases we may have to sacrifice memory for speed and we have to determine if it's worth it or not, but ideally we want to achieve both.

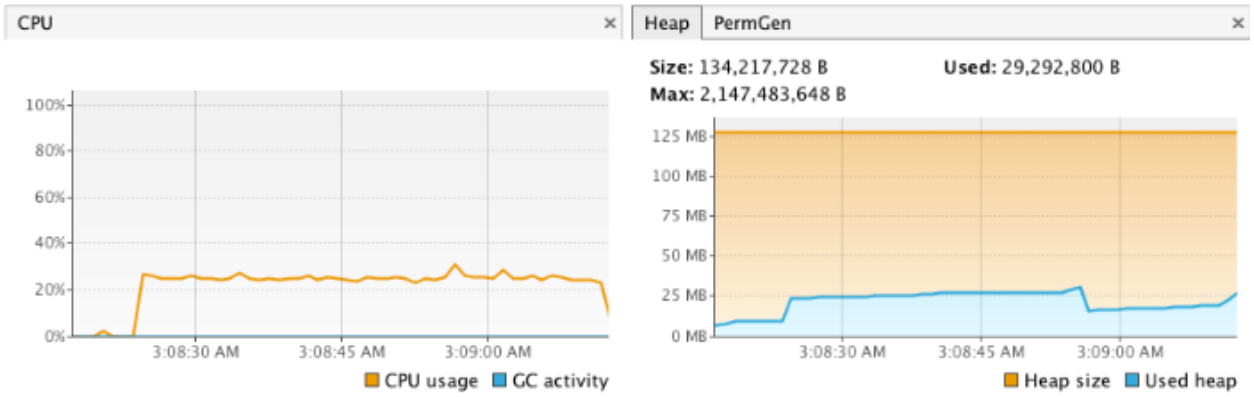
The original code as well as the refactored code can be both found on GitHub under this link: https://github.com/abrahamjj/CS1699/Deliverable5

2. PROFILER SCREENSHOTS BEFORE REFACTORING

The very first step that we took was running the program that we had using a medium-sized input grid of 10x10 through the VisualVM profiler to see which method was the CPU hot-spot. The following screenshot shows that compared to the other methods in the program the method called findAllWordsHelper() was consuming most if not all of the CPU resources throughout the execution.



The screenshot below shows that throughout the execution of the original program the CPU was running at between 20% and 40%. It also shows that the size of the used heap was at about an average of 25MB.



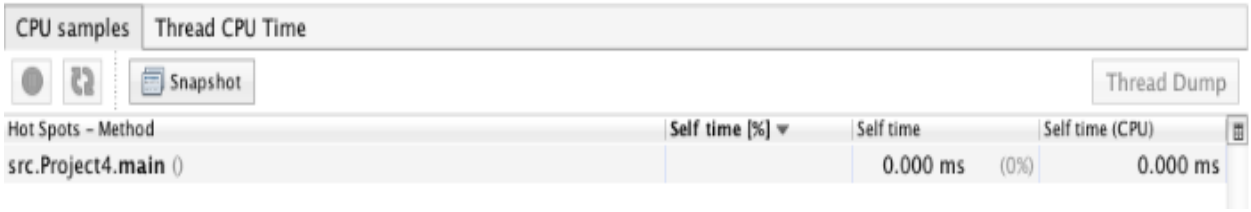
3. METHOD REFACTORING

These results gave us a starting point. We started analyzing the findAllWordsHelper() method to see if there's anything possible to refactor, and the biggest thing that stuck out was that the ArrayList method contains() was getting called on each iteration of the loop to make lookups in the dictionary stored in the ArrayList, and we both knew that ArrayLists have and average search time of O(1) and a worst case of O(n) which isn't the best. Since we're using a dictionary we decided that a Prefix Tree (Trie) data structure would be more appropriate to to store it in. A Prefix Tree has a worst case search time of O(m)
(where m is the length of a search string). This is much better than O(n) where n is the entire dictionary.

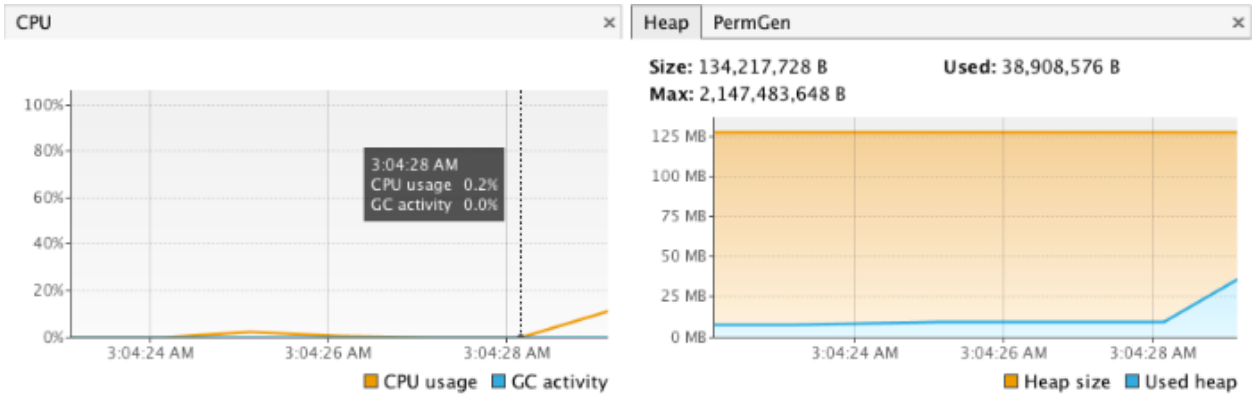
Since Java doesn't have a built-in Trie data structure class we had to build our own which was somewhat of a challenge but doable. Two classes were added, PrefixTree.java and TreeNode.java, to be used along with the main program to store the dictionary.

4. PROFILER SCREENSHOTS AFTER REFACTORING

After the refactoring was done we used the same input size that we used before, a 10x10 grid to run the program. The following screenshot shows the CPU hot-spot on the refactored program. The program ran so fast that the profiler showed 0% CPU usage.



The screenshot below shows that throughout the execution of the refactored program the CPU was running at mostly between 0% and 1% which is a big difference compared to between 20% and 40% in the original code. It also shows that the size of the used heap was at about an average of less than 10MB which is also better compared to 25MB in the original.

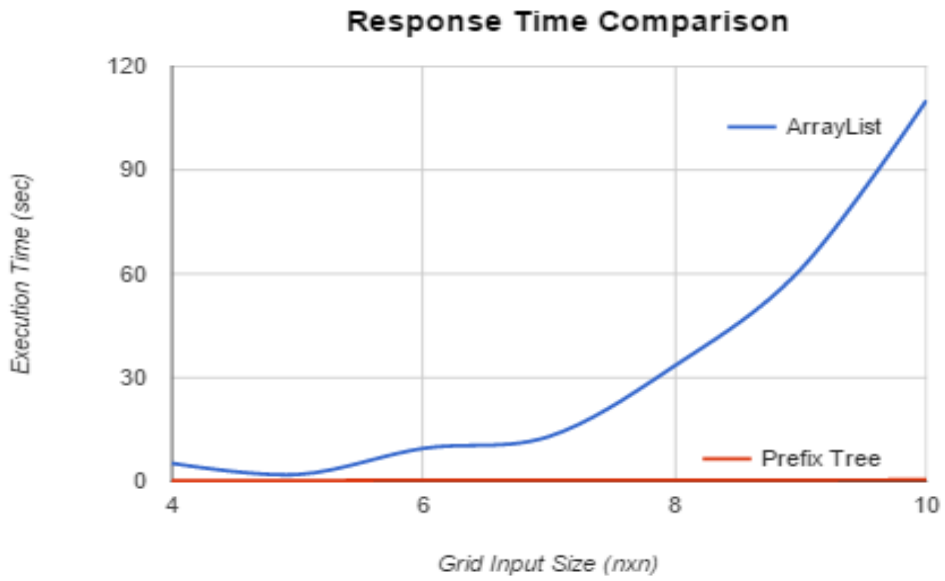


5. ASSESSMENT OF QUALITY

To measure response time we varied the input sizes, ran it through the original and the refactored programs several times, and took an average for each input size:

grid size	4x4	5x5	6x6	7x7	8x8	9x9	10x10	20x20
ArrayList (s)	5.279	1.996	9.562	12.983	33.431	61.256	48.095	211.372
Prefix Tree (s)	0.187	0.262	0.377	0.43	0.342	0.498	0.463	0.566

The following graph shows the linear versus exponential increases in execution time according to increases in grid input size:



As an overall assessment of quality, looking at the profiler results and the response time results we feel confident that users would be satisfied with the changes we've made because the graph shows that the difference in execution time is very noticeable. If we were doing this performance test for someone we would recommend that the product is ready to be released.

6. JUINIT TESTS

The only thing that remains a concern is to ensure that the refactored code produces the same output as the original code because otherwise our work would be useless. To do this we ran a 3x3 grid input with the original code which produced a list of the words found on the grid. We wrote two unit tests for the refactored code: one to check that the number of words found is the same and one to check that the actual words produced are the same. The unit tests can be found in the same repo mentioned in the summary under the "JUnit_Tests" directory. Also some minor changes were made the refactored code under test to make it testable using unit testing. The change was adding a constructor to be able to instantiate the main class and start the program. The following is a screenshot of the executed unit tests: