CS1632 – Deliverable 4: Performance Testing

Project: word\_finder

(repository link: https://github.com/yds725/D4)

Daesang Yoon (yds725) and Chris Mananghaya (chrsam)

Summary

This deliverable was very interesting. Making main program was more challenging than Deliverable 2. For this assignment, we actually had to build graph (or other kinds of data structures) and traverse graph to find paths starting from each vertex or node. This does not require very hard and complex algorithm but it was still challenging to implement graph structure in Ruby for Ruby beginners. However, Dr.Laboon’s graph example from exercise 2 was helpful. In fact, we would like Dr.Laboon to show his more efficient version of traversing graph (we used a simple depth search, inspired by example from internet). Anyway, most challenging part was finding path by traversing graph. Other than that, other parts were straightforward to implement.

We considered basic edge cases or failure modes such as unexpected number of arguments and non-existent file or wrong path. We let the program print out usage to user when it catches those exceptions.

Before improving the code, the longest or highest method on the stack was ‘compare\_two\_string\_arrays’ method. This method takes two arguments which are two string arrays. One is each list of permuted path (CAKE,CAT,etc.) and other is just one huge list of wordlist.txt, our dictionary file. For our initial implementation, we had double each loop (double for loop) that compares each word from permuted list with each word in dictionary file. This is very easy, simple and naïve brute force approach. Its time complexity is quadratic, O(n2) and the performance can be worse than n2 when the graph file is too huge and our dictionary file is already huge.

Time measured by Measure-Command was 0.513..sec for small graph but the executed time dramatically increased when testing medium-size graph. It took 85.5sec (more than 1 min). We were not surprised because the time complexity is quadratic so we already expected that it would be slow. The simple brute force will eventually give us correct output but you will have to wait for the program to be finished over Christmas break if you are going to run the program with ultra big graph file.

The change we made to improve this time complexity was very simple thanks to Ruby’s array built-in function. We used intersection (& operator). The & operator between two arrays simply returns an array of common elements between two arrays without duplicate. We looked over Ruby API library and it says that intersection operator compares elements using their hashes and equal methods for efficiency. After using this operator, execution time dramatically improved.

|small\_graph.txt| -> 0.354..sec

|medium\_size\_graph.txt| -> 0.653..sec.

It also took under 1 sec for big graph file. We do not know exact time complexity for intersection operator but it is surely better than quadratic. (We believe it may be close to linear time)

Finally, we could perhaps improve more to execute as fast as Dr.Laboon’s one for ultra big graph file. However, it is not simple task because we will have to approach in different way like using dynamic programming and recursion with memoization. But we both are not super genius in algorithm design so we will probably have to spend more time to research over it. Anyhow, this assignment was helpful for understanding performance testing and definition of performance in terms of system.

Final Times (for final version)

Small\_graph: 1)0.267sec 2) 0.254sec 3) 0.268sec Mean:0.263sec

Medium\_graph: 1)0.554sec 2) 0.560sec 3) 0.631sec Mean:0.581sec

Big\_graph: 1) 0.735sec 2)0.736sec 3)0.732sec Mean:0.734sec

