<https://github.com/evanmce/D4>

@evanmce @RyanGuild

CS 1632 – DELIVERABLE 4: Performance Testing

McEllhenney, Evan Charles and guild, ryan david

Some of the challenges we faced while making this deliverable were creating the correct regex strings in order to properly parse the nodes. After getting the regex working, we realized that in order to properly create the graph, that we would need to initialize all of the nodes beforehand and then have them point to the specified nodes. We did this by reading the total number of lines in the file first, and then reading each individual line afterwards.

Another challenge we faced was figuring out how to best represent wordlist.txt as an object in order to search and compare against our found string permutations. We settled on storing the words as a hash where all words can be alphabetized, so the keys of the hash maps are those alphabetized words. By alphabetizing any string, you can find all of its valid permutations in O(1).

The edge cases / failure modes we considered were graphs where the nodes point to nodes that don’t exist on the graph. This was relatively easy to implement, but was important so that disjointed graphs won’t cause the program to stop working.

We built a couple of different word finders, we did not directly understand that it had to be a permutation of the string found in the graph, so our first iteration of the project did not work as specified in deliverable4.md (this was changed later to meet the requirements).

When using flame graph, we discovered that the method taking up the most CPU time was the population of the word hash, followed by the loading in of the wordlist.

The changes that we made based on the flame graph data was that we serialized the word list, so rather than creating the hash every time the program is run, we just load up the serialized word hash from a premade file. What was most surprising is that when we serialized the word list beforehand, it actually tripled the run time of the program rather than making it more efficient. The reasoning behind this is that reading in the markup of the serialized word hash takes longer then generating the serialized word hash itself.

We thought that we were going to have to generate another hash map containing all valid prefixes of a word, by using this map we thought we would be able to clip paths of the graph search. In actuality, generating this hash map, due to the number of string modification, was by far the most time-consuming part of the program.

We created a simple node.js script to generate variable sized graphs to use for testing. We found that even with a graph with 100 nodes with each node having between 0 and 6 links each that it was still able to find the longest valid word/s in that graph in under a second (500 nodes in 10 seconds with 0-7 links).

Our next step was to remove all of the outputs to the terminal from displaying the generated graph and all of the possible permutations from that graph, which improved our performance time slightly by eliminating unnecessary syscalls.

We took the logic from our *find\_word* function and converted it to a single word function to see if it was a word. Then, we wrapped that function in its own thread and used our string collector object to feed it substrings found from the graph. This allowed for utilization of resources by the jit compiler to garbage collect less. We had a bunch of big static arrays that were generated in the process of finding words, and now we only store finished words. This produced a nominal speed up in the traversal of the *pretty\_ultra\_big\_graph.txt (i.e. 500 nodes)*, but was less obvious for some of the provided smaller graphs.

We were surprised that this was the direction necessary to optimize the runtime of the program, but were happy with the results nonetheless.

