Bamphiane Annie Phongphouthai

Bp8qg

October 19, 2017

Postlab6.pdf

Big-Theta calculation:

The reading of the dictionary file and insertion of the words into the hash table are linear functions according the amount of words. The reading of the file will read it line by line. Then it inserts it into the spot of the table one by one. The run time of the vector implementation is also linear. The reading of the grid files is also linear. This is because we have to read according the rows and columns. Finding the word is also linear because it needs to search through the list of strings one by one. The number of directions we considered were a total of eight. Hence the run time is words*8*rows*columns. I am looking at the for loops that was used to during each search.

For the optimization I tried a number of things. The run time before any optimizations was 1.9293 seconds for the words2.txt 300x300.grid.txt. I used this dictionary and grid for all the test. I am running on a Microsoft Windows 10 with Intel® Core™ i3-3225 CPU using Linux by a virtual box. The optimizations are in order.

First, I added the *~output in the makefile. The *~output goes in the clean section. The clean section removes files ending in .o, ~, and outputfile. It isn't invoked unless the clean is called, I wasn't sure if it would do anything but I mainly wanted to try different things on the make file to see if it would do anything and to learn more about it. I thought that would decrease the time. It did by a small fraction. The run time after adding that portion reduced it down to 1.90706 seconds and speed up was 1.012 which was not too significant. I also added in a $(CXXFLAGS) in the CXX line of the make file to see if it would change. The time stayed the same.

Second, I tried to fix my hash function a little. I added a division by 2 when calculating the length. That reduced the time down to from the 1.90706 seconds to 1.74584, speed up of 1.092. I added a one to make the number I wanted an odd value representation instead because all evens are not prime since they are divisible by 2. My train of thought was if I only work with odd numbers it would be better for the program. The length, not only did I divide by 2, I also added one. I knew this would optimize my time because the length is being reduced. Therefore, the for loop wouldn't have to run as much for each key going into the hash function and then optimizing some time. This in turned reduced the time to from 1.74584 to 1.67675 seconds, speed up 1.0412.

I used a lot of if/else statements for the directions. I was rather curious to see if switch statements were faster so I used it instead. The difference is hardly noticeable. The time was reduced to 1.66257 seconds, speed up 1.008. A little faster than before but still the same. I learned that those two are very similar in time. For both situations you would have to check the conditions.

Another thing I tried was the string stream. Instead of using cout I added in a string stream that may be faster. This helped the time significantly. It reduced it down to 1.01446 seconds, speed up of 1.639. Although, there was some problem with this at the beginning. When I ran it with the string stream it stopped for a little at certain locations during the word search. I kept on looking at the code where I added the string stream and could not determine why it was doing that. It was because I had declared my variable for my string stream outside of the loop. In theory I thought that would save some time because I wouldn't have to keep declaring the variable. In practice, it slowed down my program if I didn't declare it inside the loop. Once I fixed that issue the time was down to 0.96858 seconds, speed up 1.047.

Lastly, I wanted to change the my getNextPrimeNumber implementation. I added two new variables an unsigned int and bool. The unsigned variable was declared to be what was getting passed it +1. The Boolean was used for the while loop to check if the unsigned number was a prime. It would keep adding to the number until you got a prime. Once you got a prime then it would return that. In turn, the time went from 0.96858 seconds to 0.824584 seconds, speed up 1.175.

The total speed up of my optimization was from 1.9293 to 0.824584 seconds with. The speed up was 2.34.

The bad hash function I decided to implement was to equate the hash value to the key. In turn, this caused for more collisions to take place. The more collisions the decrease in time. Going from my optimized code of 0.824584 to 3.92467 seconds with that hash function.

For the bad size I removed the prime number checker. Without that, it would increase the amount of collisions in the program. The check for the prime number would make the size smaller. The time for this was 0.94797 seconds.