By: Nicholas Soucy

For: Dr. Roy Turner

Boyer-Moore & Knuth-Morris-Pratt Algorithms

Boyer-Moore Algorithm

My Boyer-Moore Algorithm is by far the most interesting out of the two algorithms. From the pseudocode in Dr. Turner's slides, I was able to quickly write the code and it worked smoothly. However, I needed to write the lastOccureneceFunction from scratch as no pseudocode was given.

Originally, I had my lastOccureneceFunction create a dictionary that was the size of the given alphabet. It would iterate through each letter in the alphabet, and if the letter was in the pattern string, it would set the value to be the index of the last occurrence of that letter in the text. If the letter isn't in the pattern, it would set it to -1. This is inefficient because we have a lot of wasted space and time, however, as the pseudocode was given for the BoyerMooreMatch function, we needed this to correctly jump if there was no match.

Therefore, Dr. Turner and I discussed possible solutions, we wanted to create a dictionary that only had the letters that were in the pattern, so we cut down the space and time complexity of the lastOccureneceFunction. That was quite easy to code, but we needed to change the BoyerMooreMatch function as well. Instead of having 'l = lastOccureneceFunction_dictionary[text[i]]' we had to see if the character was in the dictionary first, therefore, we used a try/except block. If the letter was in the dictionary, we were able to set 'l' to be the index of the last occurrence. If the letter wasn't in the dictionary, it would be an error, so we go to the except block and set 'l' to be -1. This lets us create a dictionary with better time and space complexity.

This solution allows the algorithm to be completely independent of an alphabet! The original algorithm needs to have the alphabet because we used a try/except block, which probably was not considered in the algorithm's creation. If we didn't have this try/except block, we would need to create a dictionary the size of the alphabet, for it to be complete.

Knuth-Morris-Pratt Algorithm

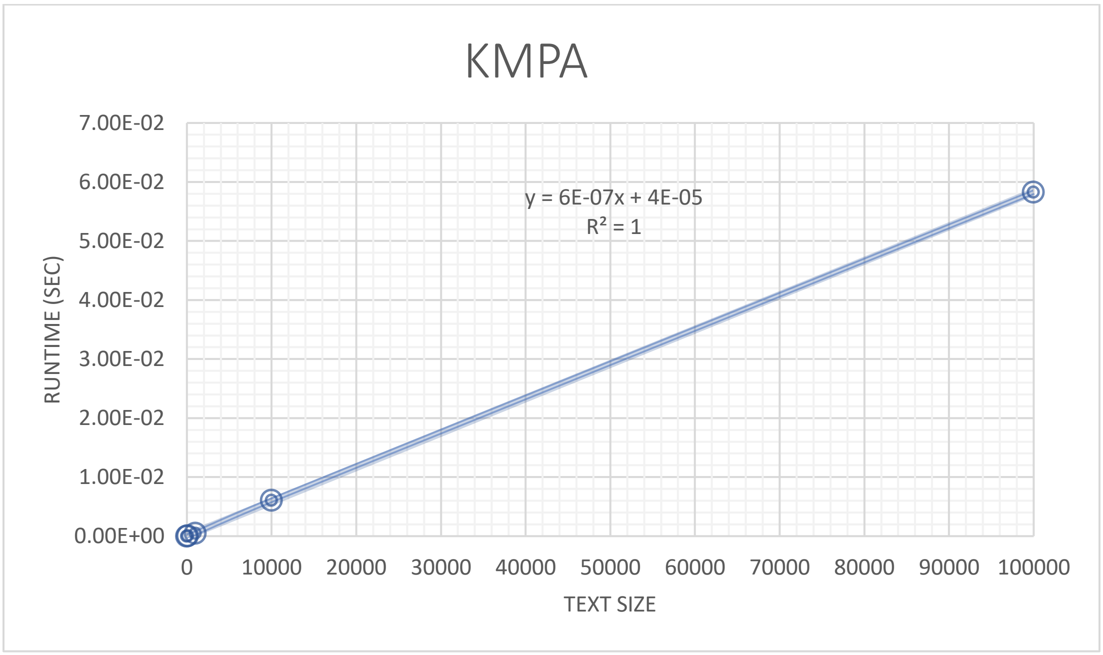
Writing the Knuth-Morris-Pratt Algorithm was quite straight-forward. I used the pseudocode that was in Turner's lecture slides and it was a direct transfer. I had some issues with the 'CreateFailureFunction' as I couldn't originally find it in the slides and issued an incompatible function that was in the book. After Dr. Turner pointed this out, I was able to use the function in the slides and everything worked with no issues.

Time Comparisons

For the time comparison, I ran text sizes 10 – 100,000 characters in length. In the interest of timing the worst case, I let the pattern be equal to the text. We expect KMPA to grow linearly and BMA to grow n-squared time. Each point is a combination of 1000 trials. (Yes, I had it running a lot over the day! Except for BMA at 100,000. I had to run that for 10 trials so it's not a great average, it just took too long!)

Textsize	BMA (sec)	KMPA (sec)
10	1.15E-05	4.86E-06
100	0.000672875	4.13E-05
1000	0.069264574	0.000509064
10000	7.035249472	0.006081926
100000	711.1788385	0.0582812

KMPA:



KMPA is a clear linear line, with perfect R-squared value. Therefore, our experimental runtime for KMPA matches the theoretical exactly.

The BMA doesn't look very nice on a graph due to the very large differences in numbers. Therefore, let's do our analysis using the table. We can see that as we increase the text size by 10 each time, our runtime increases by 100 each time, which is 10 squared. Therefore, our BMA's experimental runtime matches our theoretical runtime.