Project 4

Chromosome Search

Kekoa Riggin - Ling 473 - August 30, 2017

My Approach

I had to look up how to implement a trie structure online before I could move forward with this project. At first, I used a generic trie implementation that I found online, but it took 4 hours just to check if the tree could find one dna sequence in the database. However, using this implementation gave me the confidence I needed to move forward on my own and I implemented a simple graph that limited each node to 4 children in an array that must be `A`, `C`, `G`, or `T` and each array saved an empty place for non-existent children.

To add a word to the trie, I ran a for loop that took in one word at a time and read it one letter at a time. For each letter, if the letter already had a node in the current nodes children, then I simply moved to that node and looked at the next letter. If the letter did not have a node in the current node's children, then I broke from the current loop and started a new loop that stated at the current code and with the current letter in the string. This loop created a new node for each letter and put the nodes sequentially in the current node's children. The head node on the tree was not a node that contained a letter but served as a starting place. I checked that each input was correct by looping though all the targets provided in the assignment and following the sequence down the tree to see in the path existed.

One note, is that at the end node of any input word, I set a boolean value to `True` that indicated that this node in the trie is a terminal node (but not necessarily a leaf). This feature ensures that both words `CAT` and `CATS` would be recognized as words in the tree as opposed to just `CATS` because the `T` node is not a leaf node.

After the trie is complete, my script runs a search function. This search loops through each of the chromosome files. In each chromosome file, the function moves one-by-one through each of the characters in the file. As it moves, a pointer keeps track of the position of the file. Another pointer moves ahead to see if the current position matches any of the strings in the trie. As the second pointer moves forward, a pointer follows the sequence in the trie to see if the path exists. If the path ever breaks, the first pointer is moved forward and the second pointer is returned to the same position as the first. Lastly, the pointer for the trie is returned to the head node. If the trie pointer ever touches a node that is marked as final, the string defined by the characters from the position of the first pointer to the position of the second pointer is printed to the console along with the hexadecimal zero-based offset of the position of the first pointer. This, however, does not reset the pointers; they continue until the sequence in the trie does not exist.

All output is formatted according to the specifications of the assignment.

Extra-credit

I completed the extra credit portion of the assignment by saving all the standard output to a string. This allowed to me develop the extra credit portion using the `output` file from the assignment locally (which was important given that my program ran in 2.5 hours).

I didn't worry too much about space or time complexity in my extra-credit portion. I probably could have reused my trie implementation, but juggling the different pieces of data I needed to reorganize was more complicated than I cared to do with a new data structure. So I opted for an array of dictionaries. Each index in the array was assigned to the coordinating chromosome, with index 23 and 24 belonging to the X and Y chromosomes respectively.

I read the string one line at a time, and as I encountered a line that contained the file path, I pared away the path and kept only the file name. From here, I updated a counter to the corresponding index of the array and stored the following lines (containing the DNA sequence and hex number) in the dictionary at that index.

Once the array was complete, I looped through the indices of the array and accessed each of the string/hex number pairs from the dictionary at that index. I added each pair to a new dictionary that used the nucleotide string as the key and had a string that contained the hex number and the file name (based on the index of the first array) as the value. If the nucleotide string was already in the dictionary, the value was updated to append the new match for the nucleotide string.

Finally, All the items in the dictionary were written to the file `extra-credit`, according to the specifications.

This is not the most efficient method to solving this problem. There are time and space complexities here, as well as some ugly code in the extra credit function. Regardless, this function completes the extra-credit task and in a matter of seconds.

Reflection

1. **Trie**: As mentioned above, I am not very familiar with the Trie data structure. I am not sure if my implementation is a typical trie, but it does complete the tasks in the assignment. When I had implemented the generic trie I found on the internet, my script found one 12-character string in the genome with correct results in 4 hours. After implementing my own graph implementation, the same feat was accomplished in 35 minutes. I was happy with these results.

2. **Run times**: As mentioned above, the run time of my script is fairly long. Given that I am dealing with approximately 3 billion characters, I am happy with the results. However, the fastest implementations of this assignment are reported as taking only 4 minutes total. It seems that this implementation uses a different programming language, which may account for some (possibly all, but I don't consider myself to be an expert programmer). I wrote my script in Python 3.x. With my final implementation (which included the extra-credit assignment), my script ran in 2.5 hours.

3. **Hexadecimal Offset**: One of the biggest issues I had with this project was generating the hexadecimal offset. I had no idea what this was, and after a considerable amount of

research, I only had a vague understanding of how to create a hexadecimal number from a decimal. I posted to the discussion board and received several responses from my classmates in a matter of hours. Within five minutes, I was able to solve the hexadecimal problem of the assignment.

4. **Extra-credit**: The problem with the extra credit assignment was the run time of the assignment task. As mentioned above, I developed the extra-credit portion of the assignment locally by using the output file as the source. This worked great since the extra-credit function runs almost instantly. Once I had the extra-credit correct, I had to introduce the code into my assignment code and run that code a couple times. This meant that any mistakes would take at least 2.5 hours to find.