**Nicholas Robishaw**

**INF-503 HW5**

**NAU ID: nr768**

**Student ID: 005841641**

**Zoom Recording:**

**Job script commands for running the program after compiling.**

Commands for Subproblem A:

Commands for Subproblem B:

**My Approach**

This homework assignment required the students to create a tire tree for storing and searching for nodes. Each node would correspond to the letters A, C, G, T, and N. Just like the past homework, our program will be searching for queries from the subject fragment. To start this homework, I first read through all the subjects, which were approximately 3 billion characters, into a character array. This file reader was used for the previous homework and needed no changes. Once the subject was read in, it was time to create the trie. Now to create the trie, I needed to first come up with a custom data structure for what the node will hold. I came up with a structure of just pointers. Each pointer in the struct (A, C, G, T, and N) will essentially be seen as a character depending on the previous pointer. Now the root node will essentially be the head of this trie, and it can point to up to five different pointers. Here is what the structure will look like:

A screen shot of a computer

Description automatically generated

When a pointer is not pointing to a nullptr, then the program will see that as a letter. This is a little confusing at first, but it's far more efficient than holding actual characters in the structure space or search complexity. So the function will first generate a random 50k-segment substring from the subject. Once this segment is created, the program will have a parameter passed in that matches the command line parameter for how many queries we want the program to generate. For this homework, we need to generate 5k, 50k, 100k, and 1M query fragments from the segment. These fragments are 36 characters long, which is 4 characters longer than the previous homework. Once a random query is created from the segment, it is passed into an add node to trie helper function, which will iterate through the trie and only add the node in the path if it does not exist. This means that the program will iterate through the fragment and the trie at the same time. It will first check if the character exists, and if it does, then it will go to that node; otherwise, it will assume that it does not exist, and it will create the node, set the pointer, and then go to that new node. This way, I create a nice, compact tree that we can iterate through and save space since we do not want duplicate paths. Now this process will continue for the number of random indexes wanted. Now part B adds another step to this process where the random indexes will have a 5% per-base error rate, meaning that the string can mutate before being added to the trie. I accomplish this by checking if we need to do subproblem B, and if that is true, I will pass the random index created to another function, which will modify the string if a 5% is hit. Moving on to the searching process, it will work very similar to how the program created the trie. The program will iterate through the 50k segment linearly and create 36 character-long fragments. The test fragment will be used to iterate through the trie. For this program, we are using fuzzy search to calculate the hits in the query, and we are allowed up to 1 mismatch. For this search function to work with fuzzy matching, I needed to incorporate a stack to do level-based traversal. This means that at every level, I will be trying every possible combination if it has up to one mismatch. So at each node checkpoint, the program will first check if the node exists to match the character, then add that node to the stack with the index and the mismatch counter. Under that check will be an option to explore the other child nodes, to which it will push another node to the stack, add the index, and update the mismatch score to +1. The next check will assume a mismatch and will also push the new path to the stack with the updated mismatch score of +1. At the top of each loop, the top of the stack will be pulled off for these checks. This way, the program will find as many hits as possible with fuzzy searching. Finally, once the traversal function is finished, the program will display the number of nodes in the trie along with the hit count and then call the deconstructor to free up all the memory allocated from the trie and subject.

**Subproblem A)**

5k Queries

50k Queries

100k Queries

1M Queries

**Subproblem B)**

5k Queries

50k Queries

100k Queries

1M Queries

**Describe the differences observed in the results between Part A and Part B? Additionally, explain why these results make sense and why these differences might have been observed (if any).**

**Describe the specific bugs and issues you encountered while solving this assignment. These bugs could be from any part of your code for this homework. Provide detailed explanations of these challenges, avoiding trivial errors such as "missing a semicolon in the code."**

The only bug that I ran into in this homework was during the trie creation process. I was running into a segfault when the program attempted to free the memory allocated to the trie. The root node would be created and would be the only node present in the tree when the program was completed. This was very strange, and it seemed as though the nodes were not being linked together properly. After doing some digging in the node creation, I luckily found the bug in the tree creation function where I redefined the root node pointer to a local variable instead of a global variable, which is what it needs to be. The compiler did not catch this redeclaration since it was legal, but it messed up the whole program. To fix this, I had to get rid of the “character\_node\*” that was in front of the local temp pointer that will be used to iterate through the trie. This was a very small bug and was the only bug that I ran into during development.

**Highlight at least one specific optimization you made to improve the code's efficiency or readability**

The only optimization for this program was with the node data structure. At first, I had planned on each node housing the actual character that the node represented. I went back to rewatch the lectures and saw that we can just get away with the pointers in the node and just add some logic to tell the program that when it’s determining the next node, it will just check if that pointer is null or not. This saves a large amount of memory the program needs to run and shrinks the number of lines just a bit. At first, it may be confusing to the developers, but the comments help explain how the data structure works.