Sejal Dua
COMP 15
Matias Korman
FALL 2018

The Biologist's Grep part 1

Student: Sejal Dua (utln: sdua01)
Advising TA: Eli Rosmarin (utlin: erosma01)

Date: 22nd October 2018
Deadline: 24th of October (23:59)

(1) What data structures will you be using for this project, and why were those choices made? If using more than one data structure, how will they interact? If you are modifying a known data structure, what changes will you be making and why?

I will be using a tries as my data structure for this project because it is the most logical implementation of a tree-like structure for the purposes of the task at hand. When working with letters, instead of making a 26-ary tree or something of that sort, tries are useful because they place a heavy emphasis on prefixes and suffixes. Since we will be working with matching / comparing sequences, we want to be able to traverse a tree and get to a point where we can then decide which path to pursue.

Unlike the last homework, during which we implemented a simple binary search tree containing nodes with data and a next pointer, the nodes of our trie are going to be a bit more complex for this project. I am going to implement a struct trieNode which is going to not only contain a pointer, but it is going to contain an array of pointers. Each child in the trie should have at least 4 branches for A, C, T, and G. The array should also be able to handle while card characters, such as * and ?. I may allocate a pointer specifically for when we arrive at an asterisk, but I will probably handle ? with a separate function. I plan on denoting the end of a sequence /pathway in a trie with a $ pointer.

To answer the questions more directly, I am using multiple data structures: tries, arrays, and linked lists. The data structures interact with one another quite cohesively. Linked lists make the trie traversable and the array within the struct for a trieNode helps us use indexing to get specific pointers when we arrive at each child.

(2) How do you plan on solving insert, query and remove requests?

I plan on solving insert, query, and remove requests recursively. For an insert, I will take in a string and ensure that a path that condenses to that string exists in the trie. I will recurse through the existing trie until I am either able to formulate (via traversal of the trie) the whole DNA sequence I have passed in OR once I arrive at a leaf node (a child that points to null) that lacks the next node I want the trie to have, I will insert it. In the case of a ?, I will have no choice but to insert all 4 options: A, C, T, and G. In the case of a *, I will point have the previous node point to the * position in my struct array.

For the query function, I will recursively traverse my trie with the goal to get as far down as possible such that the prefix matches perfectly with the passed in string. Once the matching trend cannot be upheld any longer, then length gets priority. If two sequences are the same length, then alphabetical order takes precedent. The query function will have a base case such that when we reach the end of a sequence that we have followed all the way down to its $ symbol, we return the string concatenation of all the nodes we traveled through. For the recursive part, we will follow pointers down character by character, in accordance with the passed in string. In the event of a ?, we need to call query 4 times to see if the A path, the C path, the G path, or the T path gives us a sequence with highest resemblance. I anticipate needing some kind of "evaluate success rate" helper function, which will tell us how closely the returned string matches the passed in string.

For the remove function, we will have to consider three cases:

1. We must first check to see if the string we are trying to delete exists in the trie. If we are unable to find an exact match after querying the sequence, we will return false. If it does exist in the trie with a 100% match, we can then check the other two cases.
2. If we are trying to delete a sequence that exists in the trie (with a 100% match), then we have to check if any other sequences are dependent on the nodes we would potentially be deleting. For this, I am going to recursively "bubble" up from the last character of the sequence I am trying to remove and check my struct node array to see if the total amount of pointers that the node has is less than or equal to 1. If that boolean evaluation is false, we cannot safely delete the node. If it is true, we can walk up the trie and check to see if we can remove the next node and so on.
3. If we are trying to delete a sequence that is a subsequence of another sequence that exists in the tree, then all we have to do is set the node's $ pointer to null. Therefore, we will have effectively "removed it" by just making it part of whatever larger sequences depend on it, but not a complete string in and of itself.

(3) Why do you think your solution is a good solution to this project? How are space and time concerns addressed?

The worst case runtime for creating a traditional trie structure and populating it is dependent on how many strings the trie contains, and how long they might potentially be. We can express this runtime as $O(m \cdot n)$, where m is the longest sequence and n is the total number of sequences stored in the trie. The runtime for searching, inserting, and deleting from a traditional trie depends on the length of the passed in string a and the total number of string. Thus, it is $O(a \cdot n)$.

However, this project involves a bit of a unique implementation of a trie. The runtime for our functions are going to be exponential because the addition of the ? wildcard leaves us no choice but to check all 4 possible bases. There is no easy way to avoid this, but it is important to be aware that we are implementing a data structure with a slow runtime, but we know why that has to happen.

As far as space concerns go, tries do take up a lot of space in memory because they involve a lot of nodes. To engineer around that not-so-optimal reality, I considered using a compressed trie, which uses strings as keys instead of chars. I am still undecided on if I am going to go through with that. Compressed tries could get problematic if there are many insertions / deletions- they work best for a static data storage structure that is primarily used for queries only.

One thing worth mentioning is that I am using an array of pointers within my node struct. Arrays take up more space in memory, thus making each of my nodes bigger, but I am justifying this choice because I think the ability to index into my array and grab whichever pointer I want will make my life a lot easier. I am still pondering how to optimize space, but I think I have a good handle on advantages and disadvantages of tries.

(4) In a general sense, how will your code be structured? What classes will you write? What are some of the most important functions you must implement?

I will definitely have an insert function, a query function, and a remove function, but the name of the game is modularization! I will probably write functions called tree_height, node_total, evaluate_match, pointer_count, unknown_wildcard, and more that may come to mind while I am coding. As with any assignment, I will need a constructor, a copy constructor, and assignment overload to help me with my driver cpp.

(5) Any code you have written thus far.

I have done outlining of my .cpp and .h files. My files kind of look like skeletons or, in other words, what the files would look like after cp-ing them from the hw server into my hw directory. I basically have some starter code with //TO DO in each function and a strong understanding of the tasks I must accomplish. I have made the Makefile, but I have not written any major functions yet because I have a midterm tomorrow. :/