Jordan Carr

CS 1501

7/14/17

Solving the Anagrams Problem (and How to Do It Quickly)

Finding anagrams is as interesting a mathematical problem as it is a linguistic one. Computing the permutations of a set number of letters into English words involves both areas of study, and the algorithm presented is one means of deriving them. However, finding anagrams for an arbitrary number of letters efficiently is another problem entirely. Some implementations of the solution could take as little as seconds while others could take days to derive the same exact results. There are a variety of ways to solve the anagrams problem, but exploring how they are found in two different implementations gives insight on what makes an algorithm efficient.

The *Anagrams* class is a presented solution to solving the problem of finding all of the anagrams which can be made with a given string of arbitrary length. The first task the program performs is reading two text files, one containing a dictionary of words to reference from and the other containing a series of strings which serve as the arguments that anagrams will be derived from. The dictionary text file adds each word individually to a data structure which stores them and formats them in a way in which they can be searched and checked easily. This object can either be a *MyDictionary* or a *DLB*, both of which have the same functionality via the *DictInterface* interface and will be discussed in further detail later. User input determines which one will be used upon runtime. When the arguments file is read, the strings are copied to two lists, one which will convert them to mutable objects for ease of modification and use for the algorithm, and the other to immutable strings which will be stored for writing them to the output file. Modification of the mutable objectsincludes sorting all of their individual characters alphabetically using a variation of quick sort called *alphaSort* in order to make the derivation of the anagrams appear in alphabetical order, simply a formatting choice for writing to the output. In addition to this, any spaces that may appear in them are removed so they are not mistaken as letters by the algorithm using a method called *delSpaces*. These mutable objects will be referenced when building solutions.

After said changes are complete, the mutable objectsare ready to be passed to the *findAnagrams* method individually. The main problem encountered when developing the program was whether to use two different methods in order to find single-word and multiword anagrams. Both were developed separately but it was determined only the later was needed due to its flexibility in being able to support single words. The first was kept for the sake of increased functionality for the class as a whole.

Upon each iteration of the algorithm variables are reset to form the basis of building the anagram solutions. These include *build*, a modifiable object used to build the solutions; *available*, a Boolean array which corresponds to the individual characters of the argument string and stores whether a character is available to be added to *build*; *pos*, an integer indicating what position within *build* the program is building from; and *numSpaces*, an integer indicating how many spaces are used in currently *build* which also indicates how many words *build* is comprised of.

When the *findAnagrams* method is called it cycles through all of the characters of the argument string. After checking whether the character is available, it will be appended to *build* in order to start building a solution. If it is already in the solution it will try the next one until one is found or the end of the argument string is reached. *Build* will then be checked by the *searchPrefix* method from the *DictInterface* object. If the prefix is not found, the character will be removed and another will be added until it is valid, a form of pruning the solutions from the pool of possible permutations of letters and not bothering to build more solutions from invalid prefixes. If *build* is just a prefix it will recursively call itself, updating *pos* and looking for another character or deleting them and backtracking until a valid word is formed. If what is found is just a word, it will then check whether all of the letters were used using *available* and the *allUsed* method, which returns true if all of the letters in the argument string were used in *build*. As long as it returns true and the anagram has not already been found, it is added to *anagrams*, a list of liststhat stores the anagrams as strings.

Each of the sub-lists stores the answers based on how many words it is comprised of. This is determined by the number of spaces, or *numSpaces*, that the particular anagram has. Essentially, *numSpaces* acts as an index for which sub-list the anagram will be placed in. This is done to more easily format the anagrams for printing to the output file. If not all characters are used, a space is added to *build* and it will recursively call to try to find another word from the remaining characters. In the case that *build* is both a word and a prefix two things will happen. First, it will be treated as before and will be given another space if not all of the letters have been used. After the program eventually backtracks back to this point, the space will be deleted and *build* will be treated as a single word, with more letters being added on to it until it either backtracks fully or anagrams are formed from it. When the algorithm backtracks to where a certain letter was added, it will be removed and it will cycle through the letters until all possible anagrams have been found.

After the anagrams are formed the results are printed to a text file using the *writeToFile* method. The output is formatted and sorted in order of how many words the anagrams are comprised of. In other words, the anagrams are printed in order of which sub-list they came from. If a sub-list is empty it will be skipped until all of the anagrams from all of the lists have been written to the file. Once this is done, the cycle repeats and the anagrams for the next string are found, formatted, and printed to the output until all possible solutions from all argument strings have been derived.

To examine the differences between efficient and inefficient algorithms two implementations of *DictInteface*, *MyDictionary* and *DLB*, have been developed. *DictInterface* is an interface which lays out how the algorithm will store and search for dictionary terms when building anagrams. *MyDictionary* is one implementation which achieves these results. Data added to it is stored in an alphabetically sorted list. The *searchPrefix* method is implemented using nested loops and takes a mutableobject as an argument. The outer loop iterates through all of the words in the dictionary while the inner loop iterates through the individual letters of a given word.

When called, it first checks whether the length of the argument is longer than that of dictionary term. If it is then they do not match and another word is tried. It then checks whether the letter in the argument string comes before the one in the dictionary term. If it does not, then that word is not in the dictionary at all. This condition is reached if after all of the words that precede the argument alphabetically have been checked and none of them matched, stopping the algorithm from checking dictionary terms all together after that point. If the string passes both of these tests, then the individual letters of both the term and the string are checked. If they are not equal, then they do not match, another dictionary term is tried, and the process above is repeated. If the program has reached the point where all of the letters of the argument have been checked and all of the letters matched, it checks whether all letters have been for the dictionary term have been checked as well. If they have, then that term matches the argument completely. If not, then it is a prefix to a word. In the case that it passes both tests, then argument is both a prefix and a word. The *Anagrams* class handles the return values for each case as needed.

The *DLB* implementation does not use a list to store the data but instead stores each letter in a series of nodes. Each node stores a letter; a pointer to a sibling node, which points to another letter which could be made to form a word at that level; and a pointer to a child node, which is the next letter which would build a word from the current one. If a path of nodes from the root level all the way down to some child node with one letter per level forms a complete word, then the final node in the chain of nodes will point to a designated end string character or will have a child which will have a sibling who points to said end line character, which in this implementation is ‘\*’ since no word in the English language contains it. This allows for larger words to be built from smaller words while reusing the same chain of linked nodes. For example, “lot” and “lots” could be built from the same chain of nodes. ‘l’ would be at the root level, ‘o’ at the second level, ‘t’ at the third level, and both ‘s’ and ‘\*’ at the final level of nodes while ‘s’ would have a child point to ‘\*’.

This data structure makes the *searchPrefix* method far different. To find a word one must examine each individual letter of the argument string and compare it with that at the appropriate level and node of the data structure. Starting at the root, which contains the first letter of a string, one will move through each node and examine if it contains the letter needed or not. If it does not, it moves to its sibling until a null pointer is reached or it finds the letter. If null is reached, the word being searched for is not found. If the letter is found, then it will move to that node’s child and repeat the process until all of the letters of the argument string have been checked. If this is done without reaching a null pointer, the argument is valid in some way. To determine what exactly it is, it checks what the child of the final node in the sequence is and moves through its siblings. If it finds a ‘\*’, then it is a word. If it finds anything else, not including null, then it is a prefix. As with the previous implementation, the argument string could be both a prefix and a word if it finds both ‘\*’ and another letter.

Although both implementations are valid solutions to the same problem, the difference between the implementations of *DictInterface* exemplify the gap between efficient and inefficient algorithms in terms of runtime. In all test cases the runtime of *MyDictionary* is shown to be far slower than that of *DLB*. This is primarily due to the organization of the data structure and the means by which it searches for prefixes. Because it loops through the dictionary and searches through every item regardless of where the argument could be located in the dictionary alphabetically its runtime is greatly hindered. Not only that, it could potentially search through each individual letter of each dictionary entry. Therefore, in the worst case it is a *O*(*NK*) algorithm where *N* is the number of entries in the dictionary and *K* is the number of letters in the argument. Larger *K* values means more individual letters to search through and more permutations of those letters, decreasing the runtime performance substantially. Further, the number of permutations increases by a factor of *K*!. The number of solutions in a potential solution exacerbates this problem. Since anagrams can be formed from multiple words, this means that each individual word would have to be put through the algorithm, regardless if it leads to a valid anagram or not.

On the other hand, *DLB* is far more efficient at finding solutions. This is because the number of entries in the dictionary does not affect the runtime of the algorithm. It will have to search through *K* levels of nodes and *S* entries in each level of nodes, giving it a *O*(*KS*) runtime in the worst case. However, this case is unlikely to happen and on average has a runtime of *O*(*K*). This means that its runtime is completely independent of the number of entries in the dictionary. The primary factor that does affect it is the number of letters in a potential solution, which is far fewer than the number of dictionary entries. This implementation is so much faster that it is not nearly as affected by the possible number of solutions that could be found from an anagram or the number of words in it. It is so much efficient in terms of saving time that it allows the algorithm to take much greater advantage of the pruning of the *K*! permutations of the characters of the argument. This huge increase in efficiency makes the DLB a superior solution in storing and searching for dictionary terms.

|  |  |  |
| --- | --- | --- |
| Test File Runtimes | | |
| Test | MyDictionary Runtime | DLB Runtime |
| Test 1 | ~7 seconds | Less than 1 second |
| Test 2 | ~24 seconds | Less than 1 second |
| Test 3 | ~12 minutes | Less than 1 second |
| Test 4 | ~20 minutes | Less than 1 second |
| Test 5 | Hours | ~11 seconds |
| Test 6 | Days | ~20 minutes |