Jack McVeigh

CS 260

8 October 2021

Programming Assignment 1: Autocomplete Report

Introduction

Autocompletion is a well-known feature of many applications, whether it be searches on google, word shortcuts while texting, or tab completion when writing code, the convenience and speed that autocomplete is unparalleled. Not only is autocompletion useful, but it's also quite easy to implement in its rawest of forms. In this report the autocompletion algorithm described in the below flowchart will be analyzed.

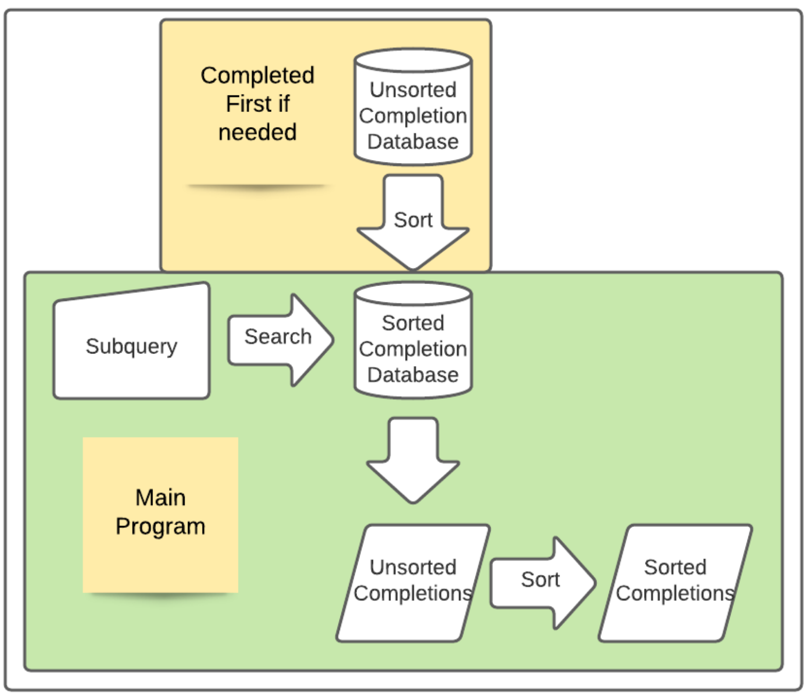

Figure 1. AutoComplete Program Execution

Loading the Dictionary

The dictionary file is provided by the user as a command line argument. The first action of the program is to load the dictionary into a data structure that allows for easily searching for a provided subquery. The structure chosen to store the dictionary is an array of entry struct pointers. Below is the definition of the entry struct. The entry struct consists of two members, a string (character pointer), word, and a float, weight.

```
typedef struct entry_s entry;
struct entry_s {
    char *word;
    float weight;
};
```

Figure 2. The entry struct definition.

This entry struct pointer array is set to a maximum size of 50,000, the size of the provided dictionary. This size assignment could have been dynamic (and should be for a larger project) by reading the length of the but was implemented as a static size to decrease complexity of the assignment (dynamic size assignment did not seem necessary). Storing the entry struct pointers as an array is beneficial as this allows for O(n) data access, which is required for the sorting algorithms used later.

Sorting the Dictionary

One of the requirements of this assignment was to sort the provided dictionary by word alphabetically so that future searches could be faster. However, when running the program, only one query can be processed, as the example execution shows in the project description. To make use of this dictionary sort process, after sorting the dictionary, the dictionary (array of entry struct pointers) is written to a new file with the extension ".sorted" in the data directory. When the user runs a query the program checks for the supplied dictionary file with the ".sorted" extension. If the file exists, the program uses that pre-sorted file, skipping the sort process. The ability to skip sorts if the dictionary has been sorted before provides incredible speed up to the program (but does require twice as much hard disk space for storing the dictionary).

The dictionary sort used is a bubble sort. Bubble sort provides decent time complexity in its O(n^2) performance. This sorting algorithm was chosen for this project as its simple to implement, making code easier, and only needs to be run once for each supplied dictionary file.

```
void sort_dictionary(entry **dictionary)
{
    int i, j;
    entry *x, *y, temp;

    for (i=0; i < DICTIONARY_SIZE-1; i++) {
        for (j=0; j < DICTIONARY_SIZE-i-1; j++) {
            if (strcmp(dictionary[j]->word, dictionary[j+1]->word) > 0) {
                temp = *dictionary[j];
                *dictionary[j] = *dictionary[j+1];
                *dictionary[j+1] = temp;
            }
        }
    }
}
```

Figure 3. Dictionary Bubble Sort.

Searching the Dictionary for Autocompletions

Searching the dictionary using the provided query requires extra overhead for the best performance. The selected search algorithm was Binary search. Binary search provides a very fast time complexity of O(logn). However, there is an issue with this style of search that a linear search would not encounter. For example, see the below excerpt from a sorted dictionary file.

```
9227 complains 1101.000000
9228 complaint 9162.000000
9229 complaints 5661.000000
9230 complement 648.000000
9231 complementary 216.000000
9232 complete 40210.000000
9233 completed 8357.000000
9234 completely 72777.000000
9235 completes 750.000000
9236 completing 1058.000000
9237 completion 1314.000000
9238 complex 11993.000000
9239 complexes 396.000000
9240 complexion 1363.000000
9241 complexities 259.000000
```

Figure 4. An excerpt from a sorted dictionary file.

If a user supplied the query “complet” to the autocomplete engine, they would expect to see the highlighted values in the output. With a linear search, that is exactly what they would see, since the search would return the position of the first match, here that position is 9,232. However, with a binary search, it is possible that the search will find a match other than the first alphabetical match. Due to this, after a match is found, the program must then do a linear search backwards until the query is not matched, then returning the position of the last matched entry.

Once the position of the first matched entry is found, the program adds each match to a linked list, the implementation of which can be found below.

```
typedef struct list_s list;
struct list_s {
    entry *data;
    list *next;
};
```

Figure 5. Linked list implementation.

A linked list is chosen, despite its time complexity of O(n), since it's expected that the user will have a complex enough query that only a few completions are found.

Sorting the Autocompletions

The completion linked list is sorted using a Bubble sort. Bubble sort was chosen as it can be implemented easily by swapping nodes in the list.

```c
void sort_completion_list(list **completions)
{
    int swapped;
    list *cur, *last;
    entry *temp;

    last = NULL;

    do
    {
        swapped = 0;
        cur = *completions;

        while (cur->next != last) {
            if (cur->data->weight > cur->next->data->weight) {
                temp = cur->data;
                cur->data = cur->next->data;
                cur->next->data = temp;
                swapped = 1;
            }
            cur = cur->next;
        }
        last = cur;
    }
    while (swapped);

}
```

Figure 6. Completion list bubble sort implementation.

Bubble sort, as mentioned earlier for the dictionary sort, has O(n^2) time complexity. This is acceptable for this application as the program, again, expects few completions to be found. In the event that many competions are found, this time complexity still performs well.

Conclusion

The overall time complexity of this program when run with a new dictionary file O(n^2) for the dictionary sort, O(logn) for the Binary search for completions, and O(n^2) for the Bubble sort of the completion list. However, for subsequent runs with the dictionary file, the time complexity is only O(logn) for the Binary search for completions and O(n^2) for the Bubble sort of the completion list since the Bubble sort of the dictionary is skipped.

Two outputs of the program are shown below to demonstrate the performance of the program. The first is a completely unacceptable initial run which keeps the user waiting for almost 15 seconds before giving an output. This is due to the intense Bubble search taking place in the beginning on the program since the “moveieScripts.txt” file has not been seen before.

```
jackmcveigh@jacks-mbp assn1 % time ./auto movieScripts.txt complet
completely,72777.000000
complete,40210.000000
completed,8357.000000
completion,1314.000000
completing,1058.000000
completes,750.000000
./auto movieScripts.txt complet  14.91s user 0.07s system 89% cpu 16.682 total
```

Figure 7. Initial run with moveScripts.txt.

The second is a lightning-fast run search using the pre-sorted file generated in the first run. This subsequent run is 744.5x faster than the initial run, using the same search query.

```
jackmcveigh@jacks-mbp assn1 % time ./auto movieScripts.txt complet
completely,72777.000000
complete,40210.000000
completed,8357.000000
completion,1314.000000
completing,1058.000000
completes,750.000000
./auto movieScripts.txt complet  0.02s user 0.00s system 85% cpu 0.028 total
```

Figure 8. Subsequent run with moveScripts.txt.

This sped up search would provide great user experience and executes the autocomplete algorithm well in an efficient manner. In the future a better implementation of this algorithm could use a more efficient dictionary searching algorithm such as insertion sort. For the applications of this project, however, the observed performance is acceptable.