CSE2010 Term Project Spring 2019 Hangman Player

Team: “ `Fantastic for(int i=0; i<4; i++)`”
Sarah Arends, Alejandra Escobar, Michael Hon, Josias Moukpe

I. Goal and Motivation

THE project serves to simulate a game of Hangman. Our program will interface with an evaluation program that provides a list of unknown words and returns feedback on guessed letters. The ”Hangman Player” program will initialize data structures from a dictionary file, use known information to guess a subsequent letter, and perform additional operations in response to feedback from the evaluation program. This project tests our ability to efficiently match patterns in strings.

II. Initial Approach

A. Algorithms and Supporting Data Structures

B. Ideas Devised by Group

C. Ideas Discussed in Course and Material

D. Ideas from Other Sources

III. Final Approach

A. Changes in Supporting Data Structures

Rather than storing each word as an ordered path of characters in a trie structure, the information on each word is encoded in the following structure.

```
typedef struct {

    bool is_cand;
    letter_t* distinct_letters;
    byte_t  get_letter_t[ALPHABET_SIZE];

} word_t;
```

Listing 1. Word Struct in ”hangman.h”

The boolean `is_cand` indicates whether the given word is still a candidate, meaning it matches the known information about the hidden word thus far. The variable `distinct_letters` is a dynamically allocated array of `letter_t` types, containing information on each letter in the word. The array `get_letter_t` maps each letter in the word to a `get_letter_t` struct of related statistics on that letter. If a letter does not occur in the word, the element in `get_letter_t` has value `NONE = 255`.

Again, each distinct letter in a word corresponds to a `get_letter_t` struct, which is defined below.

```
typedef struct {
    byte_t freq;
    uint pos;

} letter_t;
```

Listing 2. Letter Struct in ”hangman.h”

The variable `freq` stores the number of occurrences of a letter in a word. The variable `pos` encodes the positions of a letter in a word as a sequence of ones and zeroes, indicating which positions contain that letter.

The word structs are stored in a series of doubly linked lists, declared as `DLList_t** words`, each containing words of the same length. Words that are still potential candidates for the hidden word are kept at the front of the list. In this way, no words are deleted from the lists for future use, but the candidates are still separated from the non-candidates.

B. Changes in Algorithms

For any given hidden word, the only linked list of interest is the one containing words of the same length as the hidden word. This significantly reduces the number of words being considered. All words in the list are initially marked as candidates, and all letters are initially marked as “not guessed”.

The next guess is determined by traversing the list and determining what letter occurs in the greatest number of candidate words. The frequency for each letter is incremented for every word in which it appears. The maximum is assigned to be the largest of these frequencies, neglecting those letters that have already been guessed. The letter corresponding to the maximum frequency will be the next guess, and it is marked as such using a boolean array.

After the evaluation provides feedback on a guess, the list of candidate words can be further refined using the `elimWords` function. We traverse the list from the front in order to assess the words that are still candidates. Once the latter half of the string, containing non-candidates, is reached, there is no need to check the candidacy of the strings. Therefore, we can break from this traversal once a non-candidate is reached.

In the case that the previous guess was correct, the frequency of the letter in the hidden word is first determined, using the updated string passed from the evaluation program. If the frequency in the hidden word does not match that in a candidate word, we know that word is no longer a candidate and mark it as such. Otherwise, if the words have the same frequency of the guessed letter, the encoded positions of the two strings are compared to determine if the letters occur at the same set of locations. If not, that word is no longer a candidate. Words that are still marked as candidates after this process are pushed to the front of the list, in order to maintain separation between candidates and non-candidates.

A different algorithm is performed with the same `elimWords` function in the case of an incorrect guess by passing the function `is_good = FALSE`. For this case, the candidate words are again traversed, but this time every word that contains one or more occurrences of the ”bad guess” is marked as a non-candidate. If a word is still marked as a candidate after this process, it is pushed to the front of the list.

C. Additional Input: Headers

D. Ideas Devised by Group

E. Ideas Discussed in Course and Material

F. Ideas from Other Sources

IV. Evaluation

A. Accuracy, Time, and Memory usage

B. Improvements from Initial to Final Submission

V. Analysis

A. Improvement in Accuracy

B. Improvement in Time Complexity

C. Improvements in Memory Usage

D. Possible Further Improvements