CSE13S Winter Quarter 2022
Assignment 7: Author Identification

Description of the program:

- This assignment attempts to identify the author of a document by looking at their diction and comparing their word usage to other authors

Files to be included in the Asgn7 directory

- bf.h
- bf.c
- bv.h
- bv.c
- ht.h
- ht.c
- identify.c
- metric.h
- node.h
- node.c
- parser.h
- parser.c
- pq.h
- pq.c
- salts.h
- speck.h
- speck.c
- text.h
- text.c
- Makefile
- DESIGN.pdf
- WRITEUP.pdf

Purpose:

- node
    - The node is the abstract data type that we use the most throughout the program
    - The node contains two pieces of information
        - The word and the number of times it has occurred
- hashtable

  - This is the program that is responsible for taking in information and being able to store and access it quickly
  - Because we are using a good hash function we should hope that it is in O(1) time at the very best
- bit vector
  - The bit vector is used in the bloom filter mostly as just a way to store more data more efficiently
  - The bit vector, rather than having information stored in bytes, stores it in bits, effectively increasing the effectivity of the memory by 8.
- bloom filter
  - The bloom filter looks to find where the hashed word could have gone.
  - This is useful in order to later determine if a word is in a hashtable and resolve any collisions
- salt
  - The hashtable and bloom filter both use the pre-determined salts to hash the words
- speck
  - This is the hash function we are given to use
- metric
  - This is just a more simple way for us to call different algorithms whether it be Euclidean, Manhattan, or Cosine
- parser
  - This is another piece of code that is given to us and helps us look at each piece of text word by word, using regular expressions
- text
  - This is where the bulk of what we are actually doing takes place
  - Here we create the text abstract data type, in which we include a hashtable and a bloom filter
  - Using both we are able to quickly add words to the hashtable and bloomfilter using the parser function
  - Eventually, we use the data to compute the frequency of a given word and the distance between two pieces of text using the aforementioned algorithms
- main
  - This is where we use all the functions we just created to print out the outputs, take in user input and whatnot into our final product.

Pseudocode:

- HashTable *ht_create(uint32_t size)
  - malloc hashtable

        - initialize size as size
        - allocate memory for nodes
        - return hashtable
- void ht_delete(HashTable **ht)
    - for s in range(size)
        - free memory allocated to words
    - free hashtable
- uint32_t ht_size(HashTable *ht)
    - return size of hashtable
- Node *ht_lookup(HashTable *ht, char *word)
    - index = hash of the word
    - if hashtable has a node at the index return true
    - else:
        - return false
- Node *ht_instert(HashTable *ht, char *word)
    - index = hash of the word
    - if hashtable has a node at the index and the node's word is the word:
        - increment the count for the word
    - if not:
        - return null pointer of Node
    - if hashtable returns NULL:
        - create a node with the word
        - insert a node into the hashtable
    - return node
- HashTableIterator *hti_create(HashTable *ht)
    - allocate memory for hashtable hti
    - set hti as ht
    - initialize table and slto
- void hti_delete(HashTableIterator **hti)
    - set pointer as 0
- Node *ht_iter(HashTableIterator *hti)
    - iterate throug the hashtable
        - return Node if you find one
- Node create functions can be taken from the previous assignment
- BloomFilter *bf_create(uint32_t size)
    - allocate memory
    - run through all three hash iterations
- void bf_delete(BloomFilter **bf)
    - free memory
- uint32_t bf_size(BloomFilter *bf)

    - return size
- void bf_insert(BloomFilter *bf, char *word)
    - set bitvectors for all three iterations
- bool bf_prove(BloomFilter *bf, char *word)
    - if all the indices of bf_insert are not 0 return true
- BitVector *bv_create(uint32_t length)
    - allocate memory
- void bv_delete(BitVector **bv)
    - free memory
- uint32_t bv_length(BitVector *bv)
    - return bit vector length
- bool bv_set_bit(BitVector *bv, uint32_t i)
    - bit twiddle to set bitvector
    - return true
    - if i is greater than bv_length return false
- bool bv_get_bit(BitVector *bv, uint32_t i)
    - same as bv_set_bit but get bit using temp variable and bit twiddle
- Recycle priority queue
- Text *text_create(FILE *infile, Text *noise)
    - allocate memory for the text
    - use ht_create
    - use bf_create
    - initialize word_count
    - regcomp
    - if noise == NULL:
        - parse through the infile and lower each word
        - insert to bloom filter and hashtable
    - else:
        - parse through the infile and lower each word
        - if the word is not in the noise text file:
            - insert to bloom filter and hashtable
            - increment the word_count
    - regfree
    - return text
- void text_delete(Text **text)
    - delete hashtable and bloom filter
    - free allocated memory
- double text_dist(text1, text2, metric)
    - create hashtable iterators for both text1 and text2
    - iterate through text 1

        - if the word is in text 2:
            - subtract frequencies and take the absolute value
            - add multiplication of frequencies to cosine sum
        - frequency is just the frequency of text1
        - add to manhattan sum
        - add square to euclidean sum
    - iterate through text2
        - if word is not in text1
            - frequency is just the frequency of text2
            - add to manhattan sum
            - add square to euclidean sum
    - return each sum based on metric input
- double text_frequency
    - check if the text contains the word
    - if it does look it up and find the count of the word
    - float division count of the word and the overall text word_count
    - return the quotient
- bool text_contains(Text *text, char *word)
    - probe through the bloom filter if true:
        - look it up in the hashtable
        - if the node is not NULL:
            - return true
        - else:
            - return false
    - else:
        - return true
- identify.c
    - run through getopt loop
    - get the filesize from the first scan of the database
    - create a text file with the stdin
    - for loop through filesize
        - get a .txt file and author
        - create a text file
        - calculate the distance between the text file and the stdin text file
    - look through the list and pick out the K authors with the smallest distance
    - print out results
    - destroy all the used abstract data types
    - close all open files

Credit:

- Eugene’s lab section gave me a good preliminary idea of how I should tackle the lab
- I went to Brian’s office hours and got little ideas even though I didn’t ask many questions
- The entirety of the bit vector adt was from the code comments folder