Krisha Sharma

CSE13s

Winter 2022 Long

03/04/2022

Assignment 7 DESIGN.pdf

Description of Program:

Assignment 7: Author Identification has the overall task of creating a program that attempts to identify the most likely authors for an anonymous sample of text given a large database of texts with known authors. Modern-day stylometry usually is performed using machine learning, achieving high identification accuracies by learning over time, but implementing this from scratch would take an extraordinary effort. Instead during this assignment, we will be using an algorithm that's commonly used in machine learning to identify authors of anonymous samples of text, albeit less accurately.

Files to be included in directory "asgn6":

- `bf.h`
    - Defines the interface for the Bloom filter ADT. Do not modify this.
- `bf.c`
    - Contains the implementation of the Bloom filter ADT
- `bv.h`
    - : Defines the interface for the bit vector ADT. Do not modify this.
- `bv.c`

    - Contains the implementation of the bit vector ADT.
- `ht.h`
    - Defines the interface for the hash table ADT and the hash table iterator ADT. Do not modify this.
- `ht.c`
    - Contains the implementation of the hash table ADT and the hash table iterator ADT.
- `identify.c`
    - Contains `main()` and the implementation of the author identification program.
- `metric.h`
    - Defines the enumeration for the distance metrics and their respective names stored in an array of strings. Do not modify this.
- `node.h`
    - Defines the interface for the node ADT. Do not modify this.
- `node.c`
    - Contains the implementation of the node ADT.
- `parser.h`
    - Defines the interface for the regex parsing module. Do not modify this.
- `parser.c`
    - Contains the implementation of the regex parsing module.
- `pq.h`
    - Defines the interface for the priority queue ADT. Do not modify this.
- `pq.c`

    - Contains the implementation for the priority queue ADT.
- `salts.h`
    - Defines the primary, secondary, and tertiary salts to be used in your Bloom filter implementation. Also defines the salt used by the hash table in your hash table implementation.
- `speck.h`
    - Defines the interface for the hash function using the SPECK cipher. Do not modify this.
- `speck.c`
    - Contains the implementation of the hash function using the SPECK cipher. Do not modify this.
- `text.h`
    - Defines the interface for the text ADT. Do not modify this.
- `text.c`
    - Contains the implementation for the text ADT.
- Makefile
    - `CC = clang` must be specified
    - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified
    - `pkg-config` to locate compilation and include flags for the GMP library must be used.
    - `make` must build the encoder and the decoder, as should `make all`.
    - `make encode` should build only the `encode` program.
    - `make decode` should build only the `decode` program.

    - `make clean` must remove all files that are compiler generated.
    - `make format` should format all the source code, including the header files
- README.md
    - Text file in Markdown format that describes how to build and run the program, how the program handles erroneous inputs, and any problems encountered while developing the program.
- DESIGN.pdf
    - This must be a PDF. The design document should answer the pre-lab questions, describe the purpose of your program, and communicate its overall design with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program verbatim. You should instead describe how your program works with supporting pseudocode. C code is not considered pseudocode.
- WRITEUP.pdf
    - This document must be a PDF. The write up must discuss what you observe about your program's behavior as you tune the number of noise words that are filtered out and the amount of text that you feed your program. Does your program accurately identify the author for a small passage of text? What about a large passage of text? How do the different metrics (Euclidean, Manhattan, Cosine) compare against each other?

Functions for Hash Tables:

HashTable *ht_create(uint32_t size)

The constructor for a hash table. The size parameter denotes the number of slots that the hash table can index up to. The salt for the hash table is provided in salts.h.

```
pseudocode here
HashTable *ht_create() {
     allocate memory for the HashTable;
     if the hash table is NULL
          return NULL;
     allocate memory for the slots array;
     ht->slots = slot;
     ht->size = size;
     set the high salt value;
     set the low salt value;
     return the hash table ;
}
```

void ht_delete(HashTable **ht)

The destructor for a hash table. This should free any remaining nodes left in the hash table. Remember to set the pointer to the freed hash table to NULL.

```
pseudocode here
void ht_delete() {
     free the remaining nodes in the hash table;
}
```

uint32_t ht_size(HashTable *ht)

Returns the hash table's size, the number of slots it can index up to.

```
pseudocode here
```

```
uint32_t ht_size() {
     return ht->size;
}
```

Node *ht_lookup(HashTable *ht, char *word)

Searches for an entry, a node, in the hash table that contains word. If the node is found, the pointer to the node is returned. Otherwise, a NULL pointer is returned.

```
pseudocode here
Node *ht_lookup() {
     initialize count at 0;
     initialize an index;
     set index equal to hash(ht->salt, word);
     mod the value in index by the hash table size to
     account for the 32 bits;
     while (count < ht->size)
          Node **slots = ht->slots[index];
          if (slots && ht->slots[index]->word == 0)
               return the pointer to the node;
          index = index + 1 % the size of the hash table
          increment the count by 1;
     return NULL;
}
```

Node *ht_insert(HashTable *ht, char *word)

Inserts the specified word into the hash table. If the word already exists in the hash table, increment its count by 1. Otherwise, insert a new node containing word with its count set to 1.

Again, since we're using open addressing, it's possible that an insertion fails if the hash table is filled. To indicate this, return a pointer to the inserted node if the insertion was successful, and return NULL if unsuccessful.

```
pseudocode here
Node *ht_insert() {
     initialize count at 0;
     initialize an index;
     set index equal to hash(ht->salt, word);
     mod the value in index by the hash table size to
     account for 32_bits;
     if (count < hash table size)
          if the slot is empty
               create a node and place it at the slot;
               set the count of the word to 0;
               return the pointer to the node;
          if the word at the hash table index is the same
          as the word being inserted
               increment the count, keeping track of the
               amount of times the word occurs;
               return the pointer to the node;
          index = index + 1 % the size of the hash table
          increment the count by 1;
     return NULL;
}
```

void ht_print(HashTable *ht)

A debug function to print out the contents of a hash table. Write this immediately after the constructor.

```
pseudocode here
void ht_print() {
     initialize the index at 0;
     for (index = 0, index < the size of the hash table,
     index += 1)
          if ht->slots[index] == NULL
               print(ht->slots[index]->word);
}
```

Functions for Iterating over Hash Tables:

HashTableIterator *hti_create(HashTable *ht)

Creates a new hash table iterator. This iterator should iterate over the ht. The slot field of the iterator should be initialized to 0.

```
pseudocode here
HashTableIterator *hti_create() {
     allocate memory for the hash table iterator;
     hti->slots = ht;
     hti->slot = 0;
     return hti;
}
```

void hti_delete(HashTableIterator **hti)

Deletes a hash table iterator. You should not delete the table field of the iterator, as you may need to iterate over that hash table at a later time.

```
pseudocode here
void hti_delete() {
     free the hash table;
     set the pointer to the hash table to NULL;
}
```

Node *ht_iter(HashTableIterator *hti)

Returns the pointer to the next valid entry in the hash table. This may require incrementing the slot field of the iterator multiple times to get to the next valid entry. Return NULL if the iterator has iterated over the entire hash table.

```
pseudocode here
Node *hti_iter() {
     while (hti->slot < hti->table->size)
          if (hti->table->slots[hti->slots] != NULL)
               increment hti->slot by one;
               return hti->table->slots[hti->slot-1];
          increment hti->slot by one;
     return NULL;
}
```

Functions for Nodes:

Node *node_create(char *word)

The constructor for a node. You will want to make a copy of the word that is passed in. This will require allocating memory and copying over the characters for the word. You may find strdup() useful.

```
pseudocode here
Node *node_create() {
    allocate memory for the node;
    set n->word equal to a pointer to the string
    declaration of word;
    set n->count equal to 0;
    return n;
}
```

void node_delete(Node **n)

The destructor for a node. Since you have allocated memory for word, remember to free the memory allocated to that as well. The pointer to the node should be set to NULL.

```
pseudocode here
void node_delete() {
    free n->word;
    set n->word equal to NULL;
    free n;
    set n equal to NULL;
}
```

void node_print(Node *n)

A debug function to print out the contents of a node.

```
pseudocode here
```

```
void node_print() {
     printf(n->word);
}
```

Functions for Bloom Filters:

BloomFilter *bf_create(uint32_t size)

The constructor for a Bloom filter. The primary, secondary, and tertiary salts that should be used are provided in salts.h. Note that you will also have to implement the bit vector ADT for your Bloom filter, as it will serve as the array of bits necessary for a proper Bloom filter.

```
pseudocode here
BloomFilter *bf_create() {
     allocate memory for the bloom filter;
     if the bloom filter equals NULL
          return NULL;
     initialize a bit vector pointer using bv_create;
     if the bit vector pointer equals NULL
          free the bloom filter;
          set the bloom filter equal to NULL;
          return NULL;
     bf->filter = filter;
     set bf->primary[0] equal to the salt primary low;
     set bf->primary[1] equal to the salt primary high;
     set bf->secondary[0] equal to the salt secondary low;
     set bf->secondary[1] equal to the salt secondary high;
```

```
        set bf->tertiary[0] equal to the salt tertiary low;
        set bf->tertiary[1] equal to the salt tertiary high;
    }
```

void bf_delete(BloomFilter **bf)

The destructor for a Bloom filter. As with all other destructors, it should free any memory allocated by the constructor and null out the pointer that was passed in.

```
    pseudocode here
    void bf_delete() {
        call bv_delete on bf->filter;
        free the bloom filter;
        set the bloom filter to NULL;
    }
```

uint32_t bf_size(BloomFilter *bf)

Returns the size of the Bloom filter. In other words, the number of bits that the Bloom filter can access. Hint: this is the length of the underlying bit vector.

```
    pseudocode here
    uint32_t bf_size() {
        return bv_length(bf->filter);
    }
```

void bf_insert(BloomFilter *bf, char *word)

Takes word and inserts it into the Bloom filter. This entails hashing word with each of the three salts for three indices, and setting the bits at those indices in the underlying bit vector.

```
    pseudocode here
```

```
void bf_insert() {
     initialize a variable called index1;
     initialize a variable for index2;
     initialize a variable for index3;
     set index1 equal to hash(bf->primary, word);
     index1 = index1 % the size of the bloom filter;
     set index2 equal to hash(bf->secondary, word);
     index2 = index2 % the size of the bloom filter;
     set index3 equal to hash(bf->tertiary, word);
     index3 = index3 % the size of the bloom filter;
     set the bits at index1 by calling bv_set_bit on
     bf->filter and index1;
     set the bits at index2 by calling bv_set_bit on
     bf->filter and index2;
     set the bits at index3 by calling bv_set_bit on
     bf->filter and index3;
}
```

bool bf_probe(BloomFilter *bf, char *word)

Probes the Bloom filter for word. Like with bf_insert(), word is hashed with each of the three salts for three indices. If all the bits at those indices are set, return true to signify that word was most likely added to the Bloom filter. Else, return false.

```
pseudocode here
bool bf_probe() {
     initialize a variable for count and set it equal to 0;
```

```
        initialize a variable called index1;
        initialize a variable for index2;
        initialize a variable for index3;
        set index1 equal to hash(bf->primary, word);
        index1 = index1 % the size of the bloom filter;
        set index2 equal to hash(bf->secondary, word);
        index2 = index2 % the size of the bloom filter;
        set index3 equal to hash(bf->tertiary, word);
        index3 = index3 % the size of the bloom filter;
        if the value of the bit at index1 is 1
             increment the count by 1;
        if the value of the bit at index2 is 1
             increment the count by 1;
        if the value of the bit at index3 is 1
             increment the count by 1;
        if the count equals 3
             return true;
        if the count is greater than 3
             print an error message;
        return false;
    }
```

void bf_print(BloomFilter *bf)

A debug function to print out the bits of a Bloom filter. This will ideally utilize the debug print function you implement for your bit vector.

```
pseudocode here
void bf_print() {
     call bv_print to print out the value of bf->filter;
}
```

Functions for Bit Vectors:

BitVector *bv_create(uint32_t length)

The constructor for a bit vector that holds length bits. In the even that sufficient memory cannot be allocated, the function must return NULL. Else, it must return a BitVector *), or a pointer to an allocated BitVector. Each bit of the bit vector should be initialized to 0.

```
pseudocode here
BitVector *bv_create() {
     allocate memory for one bit vector of length bits;
     if the bit vector is equal to NULL
          return NULL;
     allocate memory for the vector;
     if the vector is equal to NULL
          free the bit vector;
          set the bit vector equal to NULL;
          return NULL;
     set bv->vector equal to vector;
     set bv->length equal to length;
     return bv;
}
```

void bv_delete(BitVector **bv)

The destructor for a bit vector. Remember to set the pointer to NULL after the memory associated with the bit vector is freed.

```
pseudocode here
void bv_delete() {
     free bv->vector;
     set bv->vector equal to NULL;
     free the bit vector;
     set the bit vector equal to NULL;
}
```

uint32_t bv_length(BitVector *bv)

Returns the length of a bit vector.

```
pseudocode here
uint32_t bv_length() {
     return bv->length;
}
```

bool bv_set_bit(BitVector *bv, uint32_t i)

Sets the i th bit in a bit vector. If i is out of range, return false. Otherwise, return true to indicate success.

```
pseudocode here
bool bv_set_bit() {
     if i is out of range, i > bv->length
          return false;
```

```
        else
            set the bit at index i in the bit vector to 1
            using bit shifting;
            return true;
    }
```

bool bv_clr_bit(BitVector *bv, uint32_t i)

Clears the i th bit in the bit vector. If i is out of range, return false. Otherwise, return true to indicate success.

```
    pseudocode here
    bool bv_clr_bit() {
        if i is out of range, i > bv->length
            return false;
        else
            set the bit at index i in the bit vector to 0
            using bit shifting in order to clear the bit;
            return true;
    }
```

bool bv_get_bit(BitVector *bv, uint32_t i)

Returns the i th bit in the bit vector. If i is out of range, return false. Otherwise, return false if the value of bit i is 0 and return true if the value of bit i is 1.

```
    pseudocode here
    bool bv_get_bit() {
        if i is out of range, i > bv->length
            return false;
```

```
        if the bit is equal to 1 / if the bit is set
            return true;
        return false;
    }
```

void bv_print(BitVector *bv)

A debug function to print the bits of a bit vector. That is, iterate over each of the bits of the bit vector. Print out either 0 or 1 depending on whether each bit is set. You should write this immediately after the constructor.

```
    pseudocode here
    void bv_print() {
        initialize a variable to track the current byte and
        set it equal to 0;
        for (current byte = 0, current byte < the length of
        the bit vector, current byte += 1)
            if the bit is set to 1
                printf("1 value [%u], current byte");
            else
                printf("0 value [%u], current byte");
    }
```

Functions for Texts:

Text *text_create(FILE *infile, Text *noise)

The constructor for a text. Using the regex-parsing module, get each word of infile and convert it to lowercase. The noise parameter is a Text that contains noise words to filter out. That is, each

parsed, lowercase word is only added to the created Text if and only if the word doesn't appear in the noise Text. Why are we ignoring certain words? As you can imagine, certain words of the English language occur quite frequently in writing, words like "a", "the", and "of". These words aren't great indicators of an author's unique diction and thus add additional noise to the computed distance. Hence, they should be ignored. If noise is NULL, then the Text that is being created is the noise text itself. If sufficient memory cannot be allocated, the function must return NULL. Else, it must return a Text *, or a pointer to an allocated Text. The hash table should be created with a size of 219 and the Bloom filter should be created with a size of 2^21.

```
pseudocode here
Text *text_create() {
     allocate memory for the text;
     if the text == NULL
          return NULL;
     allocate memory for the hash table using ht_create;
     if the hash table equals NULL
          free the text;
          set the text equal to NULL;
          return NULL;
     allocate memory for the bloom filter using bf_create;
     if the bloom filter equals NULL;
          free the text;
          set the text equal to NULL;
          using ht_delete, delete the hash table;
          return NULL;
```

```
call regex_t re;
free the regex;
set text->ht equal to ht;
set text->bf equal to bf;
set the word_count equal to 0;
initialize char *word as NULL;
while word = next_word(infilem &re) does not equal
NULL
     for (i = 0, i < strlen(word), i ++)
          change the word at index i to lowercase;
     if there is no noise text
          use ht_insert to insert the word into the
          hash table;
          use bf_insert to insert the word into the
          bloom filter;
     if there is noise text
          if the word is not in the text
               use ht_insert to insert the word into
               the hash table;
               use bf_insert to insert the word into
               the bloom filter;
               increment word_count by 1;
use regfree to free re;
return text;
```

```
}
```

void text_delete(Text **text)

Deletes a text. Remember to free both the hash table and the Bloom filter in the text before freeing the text itself. Remember to set the pointer to NULL after the memory associated with the text is freed.

```
pseudocode here
void text_delete() {
     use ht_delete to delete the hash table;
     use bf_delete to delete the bloom filter;
     free the text;
     set the text to NULL;
}
```

double text_dist(Text *text1, Text *text2, Metric metric)

This function returns the distance between the two texts depending on the metric being used. This can be either the Euclidean distance, the Manhattan distance, or the cosine distance. The Metric enumeration is provided to you in metric.h and will be mentioned as well in §12. Remember that the nodes contain the counts for their respective words and still need to be normalized with the total word count from the text.

```
pseudocode here
double text_dist() {
     initialize a variable m and set it equal to 0;
     initialize a variable normt1 for the normalized value
     for one of the current text elements and set it equal
     to 0;
```

```
        initialize a variable normt2 for the normalized value
        of the second current text element and set it equal to
        0;
        if the metric is equal to manhattan
             loop until the end of the smallest text;
                  normalize the vector pairs;
                  compute the manhattan calculations;
        if the metric is equal to euclidean
             loop until the end of the smallest text;
                  normalize the vector pairs;
                  compute the euclidean calculations;
        if the metric is equal to cosine
             loop until the end of the smallest text;
                  normalize the vector pairs;
                  compute the cosine calculations;
        return m;
    }
```

double text_frequency(Text *text, char* word)

Returns the frequency of the word in the text. If the word is not in the text, then this must return 0. Otherwise, this must return the normalized frequency of the word.

```
    pseudocode here
    double text_frequency() {
        if the node in the hash table contains a word
```

```
            initialize a variable for the node word and set
            it equal to ht_lookup(text->ht, word)->count;
            divide node word by the text word_count;
            return node word;
        return 0;
    }
```

bool text_contains(Text *text, char* word)

Returns whether or not a word is in the text. This should return true if word is in the text and false otherwise.

```
    pseudocode here
    bool text_contains() {
        if the word was added to the bloom filter
            return that the node in the hash table contains a
            word;
        return false;
    }
```

void text_print(Text *text)

A debug function to print the contents of a text. You may want to just call the respective functions of the component parts of the text.

```
    pseudocode here
    void text_print() {
        print out the value of word_count;
    }
```

Functions for Priority Queue:

PriorityQueue *pq_create(uint32_t capacity)

The constructor for a priority queue that holds up to capacity elements. In the event that sufficient Memory cannot be allocated, the function must return NULL. Else, it must return a PriorityQueue *, or a pointer to an allocated PriorityQueue. The priority queue should initially contain no elements.

```
pseudocode here
PriorityQueue *pq_create() {
    allocate memory for the priority queue q;
    if the priority queue is equal to NULL
        return NULL;
    else
        set q->head equal to 0;
        set q->tail equal to 0;
        set q->capacity equal to capacity;
        allocate memory for q->author;
        allocate memory for q->distance;
        if q->author and q->distance
            return q;
    return NULL;
}
```

void pq_delete(PriorityQueue **q)

The destructor for a priority queue. Remember to set the pointer to NULL after the memory associated with the priority queue is freed. Anything left in the priority queue that hasn't been dequeued should be freed as well.

```
pseudocode here
void pq_delete() {
     if the priority queue exists
          free q->author;
          free q->distance;
          free the priority queue;
          set the priority queue to NULL;
}
```

bool pq_empty(PriorityQueue *q)

Returns true if the priority queue is empty and false otherwise.

```
pseudocode here
bool pq_empty() {
     if q->head is equal to 0
          return true;
     return false;
}
```

bool pq_full(PriorityQueue *q)

Returns true if the priority queue is full and false otherwise.

```
pseudocode here
bool pq_full() {
```

```
        if q->capacity equals q->head
            return true;
        return false;
    }
```

uint32_t pq_size(PriorityQueue *q)

Returns the number of elements in the priority queue.

```
    pseudocode here
    uint32_t pq_size() {
        return q->head;
    }
```

bool enqueue(PriorityQueue *q, char *author, double dist)

Enqueue the author, dist pair into the priority queue. If the priority queue is full, return false. Otherwise, return true to indicate success.

```
    pseudocode here
    bool enqueue() {
        if the priority queue exists
            if the priority queue is full
                return false;
            set author equal to q->author[q->head];
            set dist equal to q->distance[q->head];
            initialize a variable for the queue size and set
            it equal to pq_size(q);
            if the queue size is 0
                increment q->head by 1;
```

```
                    return true;
               for (i = pq_size(q), i > 0, i --)
                    if the q->distance[i] < q->distance[i - 1]
                         initialize a variable for the temp
                         distance and set it equal to
                         q->distance[i];
                         set q->distance[i] equal to
                         q->distance[i - 1];
                         set q->distance[i - 1] equal to the
                         temp distance;
                         initialize char *tempstring and set it
                         equal to q->author[i];
                         set q->author[i] equal to q->author[i -
                         1];
                         q->author[i - 1] equal to tempstring;
               set q->head equal to q->head + 1;
               return true;
          return false;
     }
```

bool dequeue(PriorityQueue *q, char **author, double *dist)

Dequeue the author, dist pair from the priority queue. The pointer to the author string is passed back with the author double pointer. The distance metric value is passed back with the dist pointer. If the priority queue is empty, return false. Otherwise, return true to indicate success.

```
     pseudocode here
```

```
bool dequeue() {
     if the priority queue exists
          if the priority queue is empty
               return false;
          *author = q->author[q->tail];
          *dist = q->distance[q->tail];
          initialize a variable for the queue size and set
          it equal to pq_size(q);
          for (i = 0, i is less than the queue size, i ++)
               set q->distance[i - 1] equal to
               q->distance[1];
               set q->author[i - 1] equal q->author[i];
          set q->head equal to q->head + 1;
          return true;
     return false;
}
```

void pq_print(PriorityQueue *q)

A debug function to print the priority queue.

```
pseudocode here
void pq_print() {
     print out the size of the priority queue;
}
```

Error Handling:

- I ran into an infinite loop while trying to test the code for bloom filter. Initially I thought that the error was in my bf.c; however, it was instead in my test. When I was making an if statement to test bf_probe I forgot to evaluate it out to true, when I changed this, the error was fixed.
-

Citations:

- Professor Long is cited throughout this assignment for the following:
    - Pseudocode in the assignment 7 PDF document
    - Walter Sobchack (Elmer) on discord in the CSE13s - Winter 2022 - Professor Long server
- Throughout this assignment, I had high-level pseudocode collaboration with my sister Twisha Sharma (tvsharma).
- Eugene is cited throughout this assignment for this help with explaining how to properly begin and correctly implement this assignment during his section on 03/04/2022.
- Ben is cited for his tutoring session on 03/08/2022 when he helped me understand that to access the salt array we should treat it as a pointer because we want to access the whole array, not a specific part of the array.
- Audrey is cited for her tutoring session on 03/09/2022 when she helped me understand how to properly delete the hash table as my destructor function was running errors.