Assignment 7 DESIGN.pdf

Victor Nguyen

March 12, 2022

1 Description of Programs:

In this assignment we will be making an author identification of a given anonymous text, and see how closely related the text is compared to very known authors.

2 Files to be included in directory "asgn7":

1. bf.h
    - Contains the interface for the bloom filter ADT.
2. bf.c
    - Source file that contains the implementations of the bloom filter ADT.
3. bv.h
    - Contains the interface of the bit vector ADT.
4. bv.c
    - Source file that contains the implementations of the bit vector ADT.
5. ht.h
    - Contains the interface of the hash table ADT.
6. ht.c
    - Source file that contains the implementations of the hash table ADT.
7. node.h
    - Contains the node ADT interface.
8. node.c
    - Source file that contains the implementations of the node ADT.
9. pq.h

- Contains the priority queue ADT interface.

10. pq.c
    - Source file that contains the implementations of the priority queue ADT.
11. parser.h
    - Contains the parser ADT interface.
12. parser.c
    - Source file that contains the implementations of the parser ADT.
    - Parsing implementations was given by the professor.
13. metric.h
    - Contains the necessary enumerations for the distance metrics.
    - Metric calculations was given by the professor.
14. identify.c
    - Contains the implementations of the author identification program.
15. salts.h
    - Contains necessary definitions for the hash table sizes, and bloom filters..
16. speck.h
    - Contains the speck ADT interface.
17. speck.c
    - Contains the implementations of the hash function using the SPECK cipher.
    - Given to by the professor.
18. text.h
    - Contains the interface for the text ADT.
19. text.c
    - Contains the implementations for the text ADT..
20. Makefile
    - File that formats the program into clang-format.
    - File also compiles the above .c programs.
    - Cleans files that were generated during the compilation process.
    - File also contains scan-build and valgrind to check for any errors/memory leaks.

21. README.md
    - A text file that's in Markdown format that describes how you would build/run the program.
22. DESIGN.pdf
    - Describes the design and thought process of the main program.
23. WRITEUP.pdf
    - Observations during the implementation of this Author Identification program.

3 Structure and Pseudocode for node.c:

The hash table will require a type of struct to store the words in. Our struct for this will be a node.

Node *node_create(char *word)
Creates a node containing the word onto the heap.

void node_delete(Node **n)
Delete the node. This is done by freeing the memory used for the node. Make sure to set the pointer to NULL after freeing.

void node_print(Node *n)
Presumably used for debugging our node ADT. Use this to print out whatever pertaining to the Node struct.

3.1 Note about the pseudocode:

- I used my asgn 6 as reference to make the node ADT.
- If creating the node was unsuccessful, free the node.
- When deleting a node, make sure to see if the node that was passed as an argument is even a node in the first place, don't try to clear something that isn't a node.

4 Structure and Pseudocode for pq.c:

The author identification will also require a priority queue. We need a way to enqueue and dequeue things in a certain way.

void heap_sort(PriorityQueue *q, Node *n)
Used to sort the priority queue in the correct sequence. Order is determined by author distance. Will need to implement a heap sort fixer and queuer for this assignment.

struct PriorityQueue
Create a struct that contains the things that make up a Priority queue.

PriorityQueue *pq_create(uint32_t capacity)
Create a priority queue to hold the nodes. If the priority queue was unsuccessful in creating, free the priority queue and set the pointer to NULL. Return the pointer to the priority queue.

void pq_delete(PriorityQueue **q)
Delete the Priority queue by freeing the memory used for the priority queue. Make sure to set the pointers to NULL.

bool pq_empty(PriorityQueue *q)
Check to see if the queue is empty. Return true if it's empty, false otherwise.

bool pq_full(PriorityQueue *q)
Check if the queue is full. If it's full return true, otherwise false.

uint32_t pq_size(PriorityQueue *q)
Return the queue size.

bool enqueue(PriorityQueue *q, char *author, double dist)
Enqueue an author and it's distance into the priority queue. Return true if successful, false if the priority queue was full.

bool dequeue(PriorityQueue *q, char **author, double *dist)
Dequeues an author and its distance from the priority queue. Return true if successful, false if the priority queue was empty.

void pq_print(PriorityQueue *p)
Presumably used for debugging our priority queue ADT. Use this to print out whatever pertaining to the priority queue struct.

4.1 Note about the pseudocode:

- The "node" that takes the highest priority are author distances that have the smallest distance.
- Will need to make a heap sort for this assignment.
- For deleting, enqueueing, or dequeuing, always make sure to check if the priority queue exists before attempting the aforementioned. Same should be checked when checking if the queue is empty, full, and the size.
- When enqueueing an author/distance pair, make sure the queue isn't full. The same should be done for dequeuing but check to see if the queue isn't empty.

5 Structure and Pseudocode for ht.c

struct HashTable
Create a struct that contains the things needed for a hash table. This includes:

- uint64_t salt[2]
- uint32_t size
- Node **slots

HashTable *ht_create(uint32_t size)
Create the hash table with the size defined by the argument size.

void ht_delete(HashTable **ht)
Deletes the hash table on the heap. Set the pointer to NULL.

uint32_t ht_size(HashTable *ht)
Returns the size of the hash table.

Node *ht_lookup(HashTable *ht, char *word)
Looks up a specific node that contains the word that we are looking for. Return a pointer to that node when we found one. Otherwise return a NULL pointer to indicate that we couldn't find it.

Node *ht_insert(HashTable *ht, char *word)
Inserts a word into the hash table. If the word exists already, we should instead increment the word by 1. If it doesn't exists, we should make a node for it and set the count to equal 1. Return the pointer to the newly made node in the hash table, otherwise return a null pointer to indicate failure.

void ht_print(HashTable *ht)
Prints out the hash table for debugging purposes.

5.1 Note about the pseudocode

- Hash table should be on the heap.
- I used the speck.h file that was given to hash files.

5.2 Iterating Over Hash Tables

We need a way to actually iterate over Hash Tables. We should also put these functions in our ht.c.

struct HashTableIterator
Create a struct that contains the things needed for a hash table iterator. This includes:

- HashTable *table
- uint32_t slot

HashTableIterator *hti_create(HashTable *ht)
Create the hash table iterator given a hash table. Slot should be initialized to 0.

void ht_delete(HashTable **ht)
Deletes the hash table iterator on the heap. Don't delete the table. Set the pointer to NULL.

Node *ht_iter(HashTableIterator *hti)
Returns the pointer to the next valid Node in the hash table. Return NULL if we iterated through the entire hash table.

6 Pseudocode for bv.c

We need a way to manipulate bits in a byte for our bloom filter.

BitVector *bv_create(uint32_t length)
Create a bit vector that holds length bits. Return the pointer to the bit vector.

void bv_delete(BitVector **bv)
Deletes the bit vector on the heap. Set the pointer back to NULL.

uint32_t bv_length(BitVector *bv)
Return the total length of the bit vector.

bool bv_set_bit(BitVector *bv, uint32_t i)
Sets a bit at a specific index. Return true to indicate success, otherwise return false, due to out of range, or the bv doesn't exist.

bool bv_clr_bit(BitVector *bv, uint32_t i)
Clears a bit at a specific index. Return true to indicate success, otherwise return false, due to out of range, or the bv doesn't exist.

bool bv_get_bit(BitVector *bv, uint32_t i)
Returns true if the bit specified is a 1. Otherwise return false.

void bv_print(BitVector *bv)
Prints out whatever necessary to see if the bit vector was implemented properly.

6.1 Notes about pseudocode

- Follow what I did in asgn6 for code.
- Make sure to clear all the bits before returning the newly made bit vector.
- Make sure to check if the bit vector exists before performing any of the functions mentioned.

7 Pseudocode for bf.c

We need a quick an "easy" way of iterating through the hash table to find something fast. This is what the bloom filter will be used for.

struct BloomFilter
Create a bloom filter struct that contains:

- uint64_t primary[2]
- uint64_t secondary[2]
- uint64_t tertiary[2]
- BitVector *filter

BloomFilter *bf_create(uint32_t size)
Creates a bloom filter on the heap. Should contain the 3 salts given to us in salts.h.

void bf_delete(BloomFilter **bf)
Deletes the bloom filter on the heap. Make sure to set the pointer to NULL.

uint32_t bf_size(BloomFilter *bf)
Returns the size of the bloom filter, this can be obtained through the bv size.

void bf_insert(BloomFilter *bf, char *word)
Inserts the word into all three salts and sets the bits to those indices.

bool bf_probe(BloomFilter *bf, char *word)
Searches to see if a word is contained in the salts. If true is given back from all three salts, this indicates that the word could be in the bloom filter. Return false if the word isn't present in all three salts.

void bf_print(BloomFilter *bf)
Prints out whatever necessary for a bloom filter.

7.1 Notes about pseudocode

- I tried to follow the code given from the assignment pdf as closely as possible.
- The bloom filter will let us check whether or not a word is definitely not in the hash table, or it possibly could be in the hash table.

8 Pseudocode for text.c

Adt that will parse through the text.

Text *text_create(FILE *infile, Text *noise)
Creates a text on the heap. If Text *noise is null, that means the text file that we are creating is the noise text. Otherwise we will need to filter out the words from the noise to create the text.

void text_delete(Text **text)
Deletes a text on the heap.

double text_dist(Text *text1, Text *text2, Metric metric)
Calculate the distance using either Manhattan distance, Euclidean distance, or cosine distance.

double text_frequency(Text *text, char* word)
Return the normalize frequency of a given word in the text.

bool text_contains(Text *text, char* word)
See if a word is contained in the text.

void text_print(Text *text)
Prints out the text.

8.1 Notes about pseudocode

- Computing the normalize frequencies of a given word requires dividing the count of words for that specific word, and the total number of words that shows up inside the text.
- There are three different distance calculations that our program will allow to be used. These are:
    1. Manhattan Calculations:
        - After getting the normalize frequencies of a text, we need to get the difference between the words and getting the absolute value of it. We should sum all the normalized frequencies of the words contained in the text.
        - In mathematical terms: $MD = \sum_{i \in n} |u_i - v_i|$
    2. Euclidean Calculations:
        - Euclidean requires the same as a Manhattan calculation but with an additional step of squaring the sum and square rooting the total sum.
        - $ED = \sqrt{\sum_{i \in n} (u_i - v_i)^2}$
    3. Cosine Calculations:
        - Requires a multiplication of the sums to get the cosine similarity. We minus 1 from the sum to get the cosine distance.
        - $CS = \sum_{i \in n} (u_i * v_i)$
        - CD = 1 - CS

9 Pseudocode identify.c

This program will identify and compare an anonymous sample text with a library of texts. The things you'll need to do is:

- Create a noise file.
- Read in from stdin and create a text for this file. This will be our anonymous file.
- Open up the data base library that contains the authors and text file pairs. The first line will and should be the number of authors/location pairs.
- Create a priority queue that can hold the number of elements specified in the data base.
- Process the rest of the lines in the data base in pairs.
  - Use fgets() to read in the rest of the file.
  - Open up the author's file and use that to create a new text.
  - Compute the distance between the author's text and the anonymous text.
  - Enqueue the author's name and the distance we computed into the priority queue.
- Dequeue the number of matches specified by k (or until we've ran out of things to dequeue).
- Print out the dequeued author and computed distance.

9.1 Notes on identify.c

- Our identify.c should allow these command line options:
  - -d: Specify the path to the data base of authors and texts. Default is lib.db.
  - -n: Specify the path to the file that contains the noise words to filter. Default is noise.txt.
  - -k: Specify the number of authors to print out. Default is 5.
  - -l: Specify the number of words to filter out of the noise text. Default is 100.
  - -e: Use the Euclidean distance metric to calculate the text distances. This is the default.
  - -m: Use the Manhattan distance metric to calculate the text distances.
  - -c: Use the Cosine distance metric to calculate the text distances.
  - -h: Prints out the program help page.
- If the file couldn't be open in the data base, continue on.
- Make sure to lowercase the words when reading in the files.
- Filter out any words that are in the noise text.
- Make sure to get rid of the newline that is grabbed during the usage of fgets().

10 Error Handling:

Some problems and solutions I found while coding.

1. If the noise file only contains 50 words, and we tried to specify that we should filter out 100 words. We should only still filter out 50 words.
2. If the noise file only contains 50 words, and the specified that we should only filter out 20 words. We should only filter out 20 words.

11 Credit:

- I've tried to follow the information provided in asgn7 to implement my programs.
- I've reused a lot of code that I've written in previous assignments this quarter.
- I attended Eugene's section on March 4th. He helped me understand the basics of the assignment.
- I attended Brian's section on March 3rd. He helped me understand how to get started on the assignment, and also gave me ideas on what I should start working on first.
- Audrey's example design pdf helped me structure my own design pdf. This can be found on this quarters cse13s discord server in the asgn-1-faq-n-tips channel.
- The C Programming Language book may have influenced some of the code I've written.
- I've used my own Makefile from asgn6 as reference to build the makefile for this assignment.
- I tried following the assignment pdf specifications as closely as possible.
- I used Elmer#1515 code on lower casing words. This was provided in the cse13s channel.
- The user gecko10000#7137 on the discord cse13s server helped me figure out how to remove newlines from my fgets() argument.
- I referred to my asgn3 max heap implementations to create a min heap for this assignment.