Guillermo Alfaro
CSE13S Winter 2022

Author Identification

Description:

Author identification utilizes k-nearest neighbors to compare two different texts and determine how similar they are. Utilizing three different ways to mathematically compare the distance between two texts and determine how close they are to each other.

Files

1. bf.c & .h

   Contains the implementation of the Bloom filter ADT.

2. bv.c & .h

   Contains the implementation of the bit vector ADT.

3. ht.c & .h

   Contains the implementation of the hash table ADT and the hash table iterator ADT

4. identify.c

   Contains main() and the implementation of the author identification program.

5. metric.h

   Defines the enumeration for the distance metrics and their respective names stored in an array of strings

6. node.c & .h

   Contains the implementation of the node ADT.

7. parser.c & .h

   Contains the implementation of the regex parsing module.

8. pq.c & .h

   Contains the implementation of the priority queue ADT.

9. salts.h

   Defines the primary, secondary, and tertiary salts to be used in Bloom filter implementation. Also defines the salt used by the hast table implementation.

10. speck.c & .h

    Contains the implementation of the has function using the SPECK cipher.

11. text.c & .h

    Contains the implementation for the text ADT.

12. Makefile

    File that helps compile programs.

13. README.md

    Describes how to use the program and Makefile.

14. DESIGN.pdf

This file.

15. WRITEUP.pdf

    The observed behavior of the program.

Pseudocode:

bf.c

```
    BloomFilter *bf_create(size)
        Malloc space for bf, check if successful
        Assign all salts to their respective arrays
        Return bf

    Void bf_delete(BloomFilter **bf)
        bv_delete((*bf)->filter)
        free*bf

    Uint32_t bf_size(BloomFilter *bf)
        Return bv_;ength(bf->filter)

    Void bf_insert(BloomFilter *bf, char *word)
```

```
        setbit(filter, hash)

        setbit(filter, hash)

        setbit(filter, hash)


    Bool bf_probe(BloomFilter *bf, char *word)

        Bool first = setbit(filter, hash)

        Bool second = setbit(filter, hash)

        Bool third = setbit(filter, hash)

        If first || second || third

            Return true

        Else return false


    Void bf_print(BloomFilter *bf)

        bv_print(bf->bv)


bv.c

    BitVector *bv_create(uint32_t length)

        Malloc bv

        var->vector = malloc(length * size of uint8_t)

        Initialize array to 0
```

```
Void bv_delete(BitVector **bv)
    Reverse of create

Uint32_t bv_length(BitVector *bv)
    Return bv->length

Bool bv_set_bit(BitVector *bv, uint32_t i)
    bv->vector[i / 8] |= (1 << (i % 8));   // Code on how to set a bit using bitwise manipulation from Eugene

Bool bv_clr_bit(BitVector *bv, uint32_t i)
    bv->vector[i / 8] &= ~(1 << (i % 8)); // Code on how to clear a bit using bitwise manipulation from Eugene

Bool bv_get_bit(BitVector *bv, uint32_t i)
    Vector[byte] >> bit & 1U (Citation)
    Raw link
    https://stackoverflow.com/questions/47981/how-do-you-set-clear-and-toggle-a-single-bit
```

```
Void bv_print(BitVector *bv)
    If bv is true
        print
```

ht.c

```
HashTable *ht_create(uint32_t size)
    HastTable *ht = (HashTable *) malloc(sizeof(Hastable))
    ht->slots = (Node **) malloc…

void ht_delete(HashTable **ht)
    free(ht->slots)
    ht->slots = NULL
    free(ht)
    Ht = NULL
uint32_t ht_size(HashTable *ht)
    Return ht->size

Node *ht_lookup(HashTable *ht, char *word)
    For (i = 0; i < ht_size(ht); i++)
```

```
                If (ht->slots[i].word == word)
                        Return ht->slots[i]
        Return NULL

Node *ht_insert(HashTable *ht, char *word)
        Node temp = ht_lookup(ht, word)
        If (temp != NULL)
                temp->count += 1
        Node new_word = node_create(word)
        Else ht->slots[ht_size] = new_word
        If (ht->slots[ht_size] == NULL)
                Return NULL
        Else
                ht->size += 1
                Return new_word

void ht_print(HashTable *ht);
        While HTI != null
                Print nodes
```

```
    HashTableIterator *hti_create(HashTable *ht)

        Malloc HTI

        Assigns ht to table

        Return HTI


    void hti_delete(HashTableIterator **hti)

        Free HTI


    Node *ht_iter(HashTableIterator *hti)

        For hti->slot < hti->table->size

            If hti->table->slots[i] is not NULL increase slot


node.c

    Node *node_create(char *word)

        Malloc size of Node

        Word = strdup(word)

        Count = 0


    void node_delete(Node **n)

        free((*n)->word
        Word = NULL

        free(*n)


    void node_print(Node *n)


        Print n->word


Pq.c

    Using insertion sort

    PriorityQueue *pq_create(uint32_t capacity)

        Malloc size of PQ

        Malloc array with capacity

        ->Capacity = capacity

        Top = 0




    void pq_delete(PriorityQueue **q)

        Free array

        array=null

        Free q
```

```
bool pq_empty(PriorityQueue *q)

    If top is 0 return true else false


bool pq_full(PriorityQueue *q)

    If top is not capacity return false else true


uint32_t pq_size(PriorityQueue *q)

    Return capacitu


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

    If pq_full(q) = true

        Return false

    Else enqueue node

        Assign node to top space

        Increments top

        Return true



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

```
        If pq_empty(*q) = true
            Return false
        Else dequeue
            Pass the highest priority node to the **n
            Decrememnts top
            Return true


    void pq_print(PriorityQueue *q)
        Prints q
```

text.c

```
    Text *text_create(FILE *infile, Text *noise)
        Malloc size of text
        Ht = ht_create(1 << 19)
        Bf = bf_create(1 << 21)
        Wordcount = 0


        While next word != null
        For loop
            Lowercase word
        If noise
            Compare words to noise and only add if not in noise
        Else
            Add in all words if noty already in
        Increase word count
    Return


void text_delete(Text **text)
    ht_delete(ht)
    bf_delete(bf)
    Free rtext


double text_dist(Text *text1, Text *text2, Metric metric)
    Create two array with counts of word frequencies
    Of each word in combined unique words array


    Use the different simple math done in PDF if that metric
    Is selected and return
```

```
double text_frequency(Text *text, char *word)
    Return count / wordcount

bool text_contains(Text *text, char *word)
    Probe for word

void text_print(Text *text)
    Print each word count
```

Pq.c

```
Get opt switch like all other multiple inputs
Copy only l amount of words from noise and use new file to create
Text noise
Read first line of database
Create a PQ with first number of DB
While loop
    fgets(author from database)
    fgets(filepath from database)
    If feof then break
    Remove \n from author and filepath
    Open filepath
    If successful
        Create text with file
        Get distance compared to anon text provided from STDIN
        Enqueue author and text
    Close
Print Top (k) metric (metric used) noise limit (l)
For 0 < k
    Dequeue
    Print i, author distance


Free everything
```

Citations:

I used this source to be able to get an exact bit. It uses bit shift operators to achieve this feat. I claim no credit for these singular lines of code I partially used.

https://stackoverflow.com/questions/47981/how-do-you-set-clear-and-toggle-a-single-bit