cse13s asgn7 DESIGN.pdf

Lucas Lee; CruzID: luclee

3/3/2022

1 Program Details

We will be using the k-nearest neighbors algorithm to identify the most likely author of an anonymous text.

2 Manhattan Distance

The Manhattan Distance is one of the 3 distance metrics we will use. The main concept for this is that you get the sum of the frequency/total of each word in one vector to those of another.

1. Comparing 2 vectors, get the fractional component of each, and get the absolute value of that number
2. Repeat this for each vector component and get the fractional sum of the values.
3. The value of the summation of these numbers is the Manhattan Distance

3 Euclidean Distance

The Euclidean Distance is the default of the 3 metrics we will use. This calculation subtracts the vector components and squares them before getting the total summation, then takes the square root of taht summation

1. Get the difference of two vector components
2. Square the difference from the step above
3. Repeat this for all vector components

4. Sum up all values from the calculation and take the square root of the number to get the Euclidean Distance

4 Cosine Distance

The Cosine Distance is the last of the 3 distance metrics. This calculation multiplies the vector components and subtracts the sum of these numbers from 1 to get the Cosine Distance.

1. Get the product of two vector components
2. Store the sum of each of the products
3. Repeat for each of the vector components
4. Subtract the counting sum of the products from 1 to get Cosine Distance.

5 Hash Table

We will need to make a Hash Table struct in order to store unique words in a text with the amount to times it is used. Hash tables allow for us to look for these words at a potentially O(1) time complexity.

1. HashTable htcreate(uint32t size)
   (a) constructs a hash table.
   (b) Set size equal to parameter size
   (c) Allocate memory for array of Nodes with capacity of parameter size
   (d) salts.h has the data for salt in the HashTable parameter
   (e) Initialize each slot in the hash table to NULL
   (f) Return the constructed hash table
2. void htdelete(HashTable ht)
   (a) destroys the hash table.
   (b) Free all nodes in the hash table
   (c) After freeing the nodes, set the pointer to ht to NULL

(d) NOTE: node.c contains a way to free the nodes inside the hash table

3. uint32t htsize(Hashtable ht)
   (a) return the total number of slots in the hash table
   (b) this is given by the size used when creating the hashtable
4. Node htlookup(HashTable ht, char word)
   (a) Searches given hash table for a node that contains the word given as a parameter
   (b) Return the pointer to the node if found, othewise return a NULL pointer
   (c) Hash the word that you want to look up and start from there.
   (d) Start looking through the hash table and if the word in the hash table index matches the word to look up then return that node
   (e) Stop looking when you have iterated through the whole hash table
   (f) NOTE: this may work with a counter to keep track of how many elements are looked at in the hash table.
5. Node htinsert(HashTable ht, char word)
   (a) Insert word into given hash table.
   (b) If the word is already in the table, increment its count by 1.
   (c) Use Linear Probing to traverse the hash table and find a location for the word.
   (d) If Linear Probing fails because hash table is full, Return a NULL pointer.
   (e) Use a counter and index to complete the linear probing.
   (f) If the Node is successfully inserted, return the pointer to that Node.
   (g) Insert the word into the hash table by creating a new node at a NULL index in the hash table
6. void htprint(HashTable ht)
   (a) debug function used to print the contents of the hash table.
   (b) write this immediately after htcreate() to help debug each function.

6 Hash Table Iterator

Create a struct to assist in iterating over the hash table. This should be created in the same file as the constructor and the other hash table functions.

1. HashTableIterator hticreate(HashTable ht)
   (a) Constructor for the hash table iterator.
   (b) Slot parameter will be initialized to 0.
   (c) HashTable parameter will be set to the given ht parameter
2. void htidelete(HashTableIterator hti)
   (a) Delete the current iterator.
   (b) Do not delete the hash table that corresponds with this iterator.
3. Node htiter(HashTableIterator hti)
   (a) Return the pointer to the node at the next entry in the hash table.
   (b) Incrementing the slot parameter of the iterator may assist in this.
   (c) If the next entry is not valid(there is nothing there) increment the slot parameter and dont return anything.
   (d) Return NULL if the iterator slot is at the max hash table size (iterated over entire table).
   (e) Use a while loop to iterate though the hash table and compare the size of the table to the slot parameter.

7 SPECK Cipher

Using a given SPECK Cipher, we are given a hash function that takes a salt(2 uint64ts) and a key to hash. The hash() function returns a uint32t which is the index where the key is mapped.

8 Nodes

We will need to create nodes that contain a word and its frequency(count).

1. Node nodecreate(char word)

(a) Create a copy of the word that is passed into this function as a parameter using strdup()

(b) Allocate memory for the size of the word and copy the word into char parameter

(c) Edit: strdup() allocates memory for that character, so just remember to free that character

2. void nodedelete(Node n)

   (a) Destroy a node by freeing the memory for word, and then free the node.

   (b) After memory is freed, set the node pointer to NULL

3. void nodeprint(Node n)

   (a) Debug function that prints out the contents of the node

   (b) Do this function quickly to test if node is functioning properly

9 Bloom Filters

We will be using Bloom Filters to determine if a word is most likely in the hashtable. The struct for bloom filter will be using 3 different salts to give an estimate of the word.

1. BloomFilter bfcreate(uint32t size)

   (a) Set each of the salts in the struct to the given corresponding salts in salts.h.

   (b) Allocate memory for the bit vector filter that should have the necessary amount of bits for the Bloom filter.

   (c) Edit: memory can be allocated for the bit vector using the corresponding constructor

2. void bfdelete(BloomFilter bf)

   (a) Destructor for the Bloom filter, freeing memory allocated for the bitvector and any other memory allocated in the constructor.

   (b) Use bit vector destructor to free the bitvector.

   (c) Set pointer to NULL after freeing all necessary memory.

3. uint32t bfsize(BloomFilter bf)

(a) Return the number of bits that BloomFilter bf can access.

(b) This is also the length of bit vector

4. void bfinsert(BloomFilter bf, char word)

   (a) Inserts word into the bloom filter.

   (b) Uses hash() on the word for each salt in the bloom filter (hash the word 3 different times)

   (c) Sets the bit at the indices of the bit vector

5. bool bfprobe(BloomFilter bf, char word)

   (a) Run hash() on the word for each salt in the bloom filter similar to bfinsert().

   (b) Check if the bits at the indices are set, and return true if they are all set.

   (c) If not all the bits at those indices are set, return false.

6. void bfprint(BloomFilter bf)

   (a) Debug function that prints out the bits of Bloom filter.

   (b) This may use the print function for debugging in bit vector code.

10 Bit Vectors

We need to make a Bit Vector ADT that represents a 1D array of bits. This will be used to indicate true and false through checking if a given bit within a byte is a 1 or 0. Bit shifting logic from code.c in assignment 6 can applicable here.

1. BitVector brcreate(uint32t length)

   (a) Constructor for the bit vector.

   (b) Return NULL in the event that sufficient memory cannot be allocated.

   (c) Otherwise, return a BitVector $\star$ or a pointer to a BitVector that has allocated space.

   (d) Initialize each bit of the vector parameter to 0 (calloc).

   (e) Use ceiling division on the given length to get the space to allocate for the bitvector.

2. void bvdelete(BitVector bv)
    (a) Destructor for the bit vector.
    (b) Free memory allocated with the BitVector bv passed in as a parameter.
    (c) After freeing the memory, set the pointer of bv to NULL.
3. uint32t bvlength(BitVector bv)
    (a) Return the length of the bit vector.
    (b) This is given by the value in the struct length
4. bool bvSetBit(BitVector bv, uint32t i)
    (a) Sets the bit at index i to 1.
    (b) Return false if I is out of range and true if the bit is successfully set.
    (c) Use i mod 8 to get how many bits to shift a temporary number
    (d) Use bitwise or to change a specific bit to 1
5. bool bvClrBit(BitVector bv, uint32t i)
    (a) Clear bit at index i by setting it to 0.
    (b) If I is out of range, return false.
    (c) Return true to indicate successful clearing of a bit.
    (d) Clear bit by subtracting value at bit location from 255(8 bits set to 1)
    (e) Use bitwise and to change specific bit to 0
6. bool bvGetBit(BitVector bv, uint32t i)
    (a) Gets the bit at index i.
    (b) Return false if i is out of range.
    (c) Return true if the value at index i is 1.
    (d) Return false if the value at index i is 0.
    (e) Use bitshifting to shift bits to the right.
    (f) Try to shift the value by as many bits to the right as you need to get the desired bit at index 0

(g) Check if the number is even or odd to determine the value of the leftmost bit

7. void bvprint(BitVector bv)
   (a) Debug function that prints out the bits in the bit vector.
   (b) Iterates over each byte in the bitvector and individually prints each bit.

11 Regular Expressions, Parser.c, regex

We will have files that allow us to use Regular Expressions to find words, contractions and hyphenations.

1. The given parser.h and parser.c have functions to look for words using nextword().
2. This function takes in a file to read words, and a regular expression.
3. Use regcomp() to compile the regular expression to pass it into the funciton.

12 Text

We will use a Text struct ADT to keep track of how many words are in each text.

1. Text textcreate(FILE infile, Text noise)
   (a) Constructor for text.
   (b) Get each word using functions in parser.c and regex. Convert all words to lowercase. Use tolower() in a loop to make each character in the word lowercase
   (c) Each lowercased word that is not in the noise parameter is added to the created Text.
   (d) If the Text noise is NULL, then the Text that is created in this function will be the noise.
   (e) If you cannot allocate sufficient memory then the function returns NULL.
   (f) Otherwise, Return a Text pointer or pointer to an allocated Text.

(g) Hash table should be created with a size of 2^{19}, and the Bloom Filter should be created with a size of 2^{21}.

(h) If you are creating a noise text, don't ignore any words, and stop reading and adding new words to the file if the wordcount if the text is the same as the noiselimit

(i) Otherwise, use the same loop to make words lowercased, and then skip over the words that are inside the noise text

2. void textdelete(Text text)

   (a) Delete a text by freeing hash table, bloom filter, then the text itself.

   (b) After frees, set pointer to NULL.

3. double textdist(Text text1, Text text2, Metric metric)

   (a) Returns the distance between text1 and text2 using one of the 3 metrics specified at the beginning of the document.

   (b) Metric.h provides the metric parameter.

   (c) NOTE: nodes contain counts for their words, and need to normalize it with the total word count in the text

   (d) Iterate through each text using hash table iterators to read the valid words in each text

   (e) Follow the formulas at the beginning of this document to calculate each distance based on the given metric

   (f) Delete hash table iterators after usage

4. double textfrequency(Text text, char word)

   (a) Return frequency of parameter word in parameter text.

   (b) If word is not in the text, return 0.

   (c) Otherwise return the normalized frequency of word.

   (d) Look up the word in the hash table and store the node with that word inside of a variable

   (e) Use the node's frequency and the text's word count to normalize the count and return that frequency

5. bool textcontains(Text text, char word)

(a) Return true or false based on if the word is inside the given text or not.

(b) True if word is in text, false if not.

(c) Probe the bloomfilter of the text to see if the word is most likely there

(d) If the word is there, perform a hash table lookup to check for false positives

(e) otherwise, return false

6. void textprint(Text text)

   (a) Debug function that prints the contents of parameter text.

   (b) Possible to call other debug functions of the other components of text to debug here.

13 Priority Queue

We will need to implement a priority queue similar to assignment 6's priority queue struct. This function will most likely contain a local struct in order to keep track of the author and the distance. Memory should not need to be allocated for this struct, so we will not need a destructor for it.

1. PriorityQueue pqcreate(uint32t capacity)

   (a) Constructor for priority queue.

   (b) Priority queue should have an author and the corresponding distance associated with that author as struct parameters.

   (c) Parameter capacity sets the total capacity of the priority queue.

   (d) If memory cannot be allocated return NULL from this function.

   (e) Otherwise, return PriorityQueue pointer or pointer to allocated PriorityQueue.

   (f) Priority queue should be initialized to have 0 elements inside the queue.

   (g) Create an array of structs that contain authors and distances

2. void pqdelete(PriorityQueue q)

   (a) Destructor for PriorityQueue q.

(b) Free all memory associated with PriorityQueue q.

(c) All remaining objects in the queue that are leftover must be removed/freed.

(d) Set the q pointer to NULL after freeing all memory.

3. bool pqempty(PriorityQueue q)

   (a) Return true if PriorityQueue q is empty, and false if there is more than 1 element in the queue.

   (b) It may help and make it easy to do this if a size parameter is added to the struct.

4. bool pqfull(PriorityQueue q)

   (a) Return true if PriorityQueue q is full, and false if there are 0 elements in the queue.

   (b) If a size parameter is added, this will be if the size is equal to the capacity.

5. uint32t pqsize(PriorityQueue q)

   (a) Return the number of elements in PriorityQueue q.

   (b) If a size parameter is added, this will just be the value of the size parameter.

6. bool enqueue(PriorityQueue q, char author, double dist)

   (a) Return false if the PriorityQueue q is full.

   (b) Enqueue the pair of an author and the distance dist corresponding to the author to the queue.

   (c) Return true to indicate the successful enqueue of the author, distance pair.

   (d) Queue should be sorted with highest priority going to the author with the lowest value for dist.

   (e) Edit: Enqueue the struct that contains the author and the distance for that author by setting the element at the index to the constructor function for the struct

7. bool dequeue(PriorityQueue q, char author, double dist)

   (a) Return false if the PriorityQueue q is empty.

(b) Dequeue an author/dist pair from the queue, shifting every element in the queue over by 1 to accomodate.

(c) Pass the char author back into the double pointer for author, and pass the distance back with the pointer to dist.

(d) Return true to indicate the successful dequeue of a pair.

8. pqprint(PriorityQueue q)

   (a) Debug function for checking if the Priority queue is implemented correctly.

14 Identify(Main function)

We will make an identify.c function that allows us to find the most likely author of a given text passed into stdin.

1. Handle and account for command line options:

   (a) -d = Specify path to database of authors and texts(default = lib.db).

   (b) -n = Specify path to file of noise words (default = noise.txt).

   (c) -k = Specify number of authors to match to the given text (default = 5).

   (d) -l = Specify number of noise words (default = 100).

   (e) -e = Set metric to use as Euclidean distance (default metric).

   (f) -m = Set metric to use as Manhattan distance.

   (g) -c = Set metric to use as Cosine distance.

   (h) -h = Displays help and usage.

   (i) NOTE: for setting the metrics, it should disable the other metrics if selected.

2. Specifics for main function:

   (a) Create a new Text from stdin. This will be the text we will try to identify the author for.

   (b) Create Text for the noise file. This file is specified in the command line inputs. Make all words in noise lowercased.

(c) Open database for authors and texts specified in command line -d. The first line in the file specifies the number of author/text pairs, we will call this value n.

(d) Use fscanf to read a single line in from the database and store that value inside n.

(e) Create a PQ that holds n elements. This makes it so that we can look through all the authors in the database.

(f) Read in the rest of the file to get the authors and their texts. Use a loop with a counting variable inside that counts until you reach n pairs.

   i. Use fgets() once to get the author, and use it again to get the path to the text for that author.

   ii. The call to fgets() keeps the newline characters, so we will need to remove it.

   iii. NOTE: fgets() stores string into a buffer, and the second to last element in that buffer will be a newline, since the last element in the buffer is a null pointer.

   iv. NOTE 2: duplicate the string that you are trying to store with the author to enqueue, or it could overwrite the other authors in the priority queue after enqueuing.

   v. Open the author's text and create a new Text with the file.

   vi. If the file fails to open, then skip over it and continue reading the database.

   vii. If the file opens, make sure that the parsed words are lowercased before adding the word to Text

   viii. Use the author's text and the anonymous text from stdin to compute a distance after filtering noise words.

   ix. Enqueue that author's name and the corresponding distance to the PQ.

   x. Delete the text created from the author's text to make room to create another.

(g) After reading through the entire database, we should have a PQ of the most likely authors for the text.

(h) Dequeue from PQ k number of times to get the most likely authors.

(i) For each element dequeued from the priority queue, free the character pointer for the author to avoid leaks with the string duplicate.

(j) Close all opened files, delete all created texts, and delete the priority queue.