Yash Sharma
CSE13S, Winter 2021

Assignment 7 - WRITEUP.pdf

Introduction:

This assignment requires us to implement a program that attempts to identify the most likely authors for an anonymous sample of text when given a large database of texts with known authors. In other words, we will be using an algorithm that's commonly used in machine learning to identify authors of anonymous samples of text. This algorithm is also known as the k-nearest neighbor's algorithm. By utilizing this algorithm, we will classify samples of texts and determine which authors are most likely to have authored an anonymous sample of text.

To begin, we first created the hash table function in order to retrieve unique words in a sample of text along with its count for each unique word. A hash table in this case maps the keys to values and provides fast constant-time lookups. This is done by taking a key and hashing it with some hash function and placing the key's value in an array through a set index. This way we can store all the unique words in a sample of text along with their respective counts. Within this function, we created ht_create, ht_delete, ht_size, ht_lookup, ht_insert, and ht_print functions. Each of these functions served its own purpose: ht_create was used to create a hash table given a certain size (the number of slots in the hash table), ht_delete was used to free any remaining nodes in the hash table, ht_size was used to return the number of slots a hash table can index up to, ht_lookup was used to search for a node in the hash table that contains a word, ht_insert was used to insert a specific word into the hash table, and the ht_print function was used to simply print out a hash table. Within this function, I came across a few errors when creating the hash table as I was not pointing the right nodes in the right place.

Next, we need to store each hash table entry in its own array index. In this case, we need to create a hash table iterator that keeps track of which slot has iterated up to in the hash table. The hti_create, hti_delete, and hti_iter functions are used for this implementation. In order to create a hash table iterator, we need to iterate over the hash table and set the slot of the iterator field to zero. For hti_delete, we need to simply set the table field of the iterator equal to null and free the hash table, however, for hti_iter we need to return the pointer to the next valid entry in the hash table. This function was quite simple to implement and is used heavily when reviewing large bodies of text within the samples of text.

Next, we need to create the nodes that will contain a word and count for each function. Within this function, we created the node_create, node_delete, and node print functions. Each of these functions serves its own purpose: node_create is used to make a copy of the word that is passed in and allows for a node to be created, node_delete is used to delete a node, and node_print is utilized when we need to print out the contents of a node. The node.c file is one of the most important ones because, without it, the hash table and bloom filters cannot be created.

After this, we proceed to implement the bloom filters. The bloom filter is used to test whether an element is a member of a set and returns false positives if possible. A bloom filter is also represented with an array of bits using the bit vector and maps over a set element. In other words, this implementation will allow us to add each word of a sample of text into the bloom filter. This is a much more efficient solution because it will be simply querying the Bloom Filter of the other sample of text for whether or not a word has been located, rather than looking in the hash table. Within this function, we created the bf_create, bf_delete, bf_size, bf_insert, bf_probe, and bf_print functions. Each of these functions serves its own purpose: bf_create is utilized for creating the bloom filter, bf_delete is used to delete the bloom filter, bf_size is utilized to return the size of the bloom filter, bf_insert takes a word and inserts it into the bloom filter, bf_probe takes the word and searches for it within the hash table, and bf_print is used to print out the bits of the bloom filter.

Next, we implemented the bit vector: a one-dimensional array of bits that are used to denote if something is true or false. Within this function, we created the bv_create, bv_delete, bv_length, bv_set_bit, bv_clr_bit, bv_get_bit, and bv_print functions. Each of these functions serves its own purpose: bv_create is utilized to create the bit vector with memory allocated for it, bv_delete frees and deletes the bit vector when called, bv_length is used to return the length of the bit vector, bv_set_bit is used to set the bit in a bit vector, bv_clr_bit is used to clear the bit in the bit vector, bv_get_bit is used to return the bit vector acquired, and bv_print is used to simply print out the bits of a bit vector.

In terms of the Text and Priority Queue, they both work hand in hand when implementing this program. The text file is used to encapsulate the parsing of a text and serves as the in-memory representation for the distribution of words in the file. Within this function, we created the text_create, text_delete, text_dist, text_frequency, text_contains, and text_print functions. Each of these functions serves its own purpose: text_create is utilized to create a text given a certain file with words and a text structure, text_delete simply deletes a text from the text file, text_dist returns the distance between the two texts depending on the metric being used, text_frequency returns the frequency of the word in the text, text_contains returns whether or not a word is in the text, and text print just prints out the contents of a text. Within the priority queue, we are storing the names of each author along with the distance calculated between text authored by the author of the anonymous sample of text. Along with this, the priority queue will enqueue and dequeue pairs of the author and the corresponding distance as different parameters. Within the priority queue, we created pq_create, pq_delete, pq_empty, pq_full, pq_size, enqueue, dequeue, and pq_print. Each of these functions serves its own purpose: pq_create is utilized to create a priority queue for the program, pq_delete is used as a destructor for the priority queue, pq_empty simply checks if the priority queue is empty, pq_full checks if the priority queue is full, pq_size checks the number of elements in the queue, enqueue enqueues the author & distance pair into the priority queue, dequeue dequeues the author & distance pair into the priority queue, and pq_print simply prints the queue.

For the identify program, my program accurately identifies the author for a small passage of text by taking in all of the noise words and separating them from the rest of the text when outputting the distance levels for each text. In terms of a large passage of text, my program is a bit slower when completing this however, it still sorts through the rest of the text and updates it based on the text. The different metrics: Euclidean, Manhattan, Cosine are different based on the fact that Euclidean is the fastest type of sorting the passage of text. Manhattan also has a quick response time with Cosine displaying the slowest output out of the three.