CS2461 Project 5 Report
12/06/2018
Melody Lee

I. Implementation

I split my code into five different C files. The breakdown is described below.

hashmap.c: Implementation from HW7, with some minor alterations.

`struct llnode`

- Linked list node that contains char pointer to word, frequency of word for each doc (term frequency), pointer to next node in list.

`struct hashmap`

- The hashmap itself. It contains llnode** map, the number of buckets, and the number of elements.

`struct hashmap* hm_create(int num_buckets)`

- Allocates new hashmap with specified number of buckets.

`struct llnode* hm_get(struct hashmap* hm, char* word)`

- Returns a pointer to the llnode with the word passed in.

`void hm_put(struct hashmap* hm, char* word, int d1_count, int d2_count, int d3_count)`

- Puts the key and values into the hashmap.

`void hm_remove(struct hashmap* hm, char* word)`

- Based on the key passed in, removes that key along with its associated values.

`void hm_destroy(struct hashmap* hm)`

- Frees all the memory allocated for the hashmap and its components.

`void hm_print(struct hashmap* hm, int num_buckets)`

- Prints the contents of the hashmap.

`int hash(struct hashmap* hm, char* word)`

- Maps the given word to a bucket in the hashmap by adding the ASCII codes of all of the characters and mod-ing by the number of buckets.

training.c: Pre-processing phase

`void stop_word(struct hashmap* hm)`

- For every bucket in the hashmap, it traverses through the linked list and calculates the idf (using a function from calc.c) for every node and deletes it if the idf is zero.

`struct hashmap* training(int num_buckets)`

- Creates a hashmap and reads in each document's words one by one and puts them in the hashmap with the correct term frequency depending on which document is currently being read.
- Calls stop_word() to remove the stop words and thereby conclude the training phase.

search_retrieval.c: Phase in which search query words are processed and document rank scores are calculated to output the results in order of descending relevance.

`struct rank:`

- Keeps track of both the relevance score and document number.

`double* read_query(struct hashmap* doc_hm, double* scores)`

- Assumes search query of arbitrary length, so will continuously get an individual character from stdin until newline character is encountered.
- Once query_word is filled and getchar() is a whitespace, gets that word from the document hashmap, calculates its idf, and adds the tf-idf to the document ranks.
- Uses memset() to fill in the query_word with null terminators so that after the chars are placed in query_word, the string is null-terminated.
- After looping, fill in scores array with the ranks and return pointer to the array.

`void rank(double* scores)`

- From the passed in scores array pointer, takes all of the doubles and puts them into an array of structs as the score values. Then assigns each struct in the struct array the corresponding document number.
- Sorts the structs from highest to lowest score using insertion sort (which calls a helper function to swap two elements).
- Once struct array is sorted, loops through and prints the document number if the score is greater than zero (because we do not want to output a document that contained none of the search query words). If all scores are zero, no match.

`void sort(struct rank arr[])`

- Insertion sort in descending numeric order (highest to lowest). Used in rank()

`void swap(struct rank* a, struct rank* b)`

- Swaps two structs (must pass in addresses). Used in sort()

calc.c: Functions to compute needed values in stop_word() and in read_query()

`int calc_df(struct llnode* node)`

- Use the passed in llnode pointer to check each document's term frequency for that word. If it's greater than one, then we can increase the df.

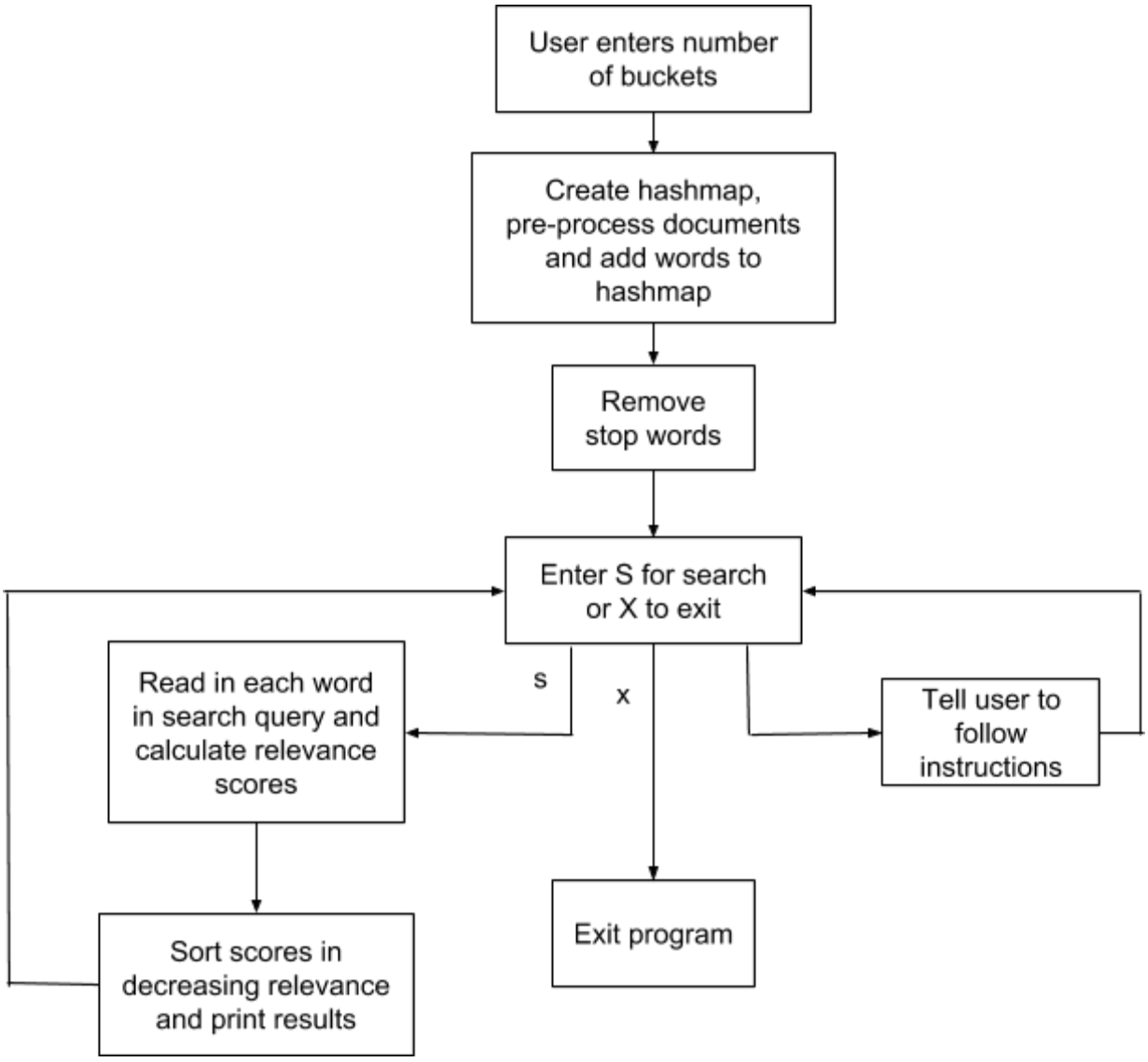
`double calc_idf(struct llnode* node)`

- Takes log base 10 of the number of documents over the document frequency (computed with calc_df).
- Handles the divide by zero case by adding 1 to the denominator.

main.c: Where it all comes together

- First prompts the user for the number of buckets and takes input in using scanf
- Then calls training() to create a hashmap with that number of buckets
- Continuously prompts for next query set or to exit
    - If s or S, read_query() and rank() to display search results
    - If x or X, stop the program
    - If not either of those, notify user

II. Flowchart



III. Questions

Question (a): If we have N documents, and document D_i consists of m_i words, then how long does this simple solution take to search for a query consisting of K words. Give your answer in big O notation.

After reading all of the documents and creating one large linked list, the linked list would have $\sum_{i=0}^{N} D_i * m_i$ number of elements, which is the total number of words in all of the documents. Searching through the linked list in the worst case is therefore $O(\sum_{i=0}^{N} D_i * m_i)$. Since we must search through the linked list for *each* query keyword, this naive solution has a time complexity of $O(\sum_{i=0}^{N} D_i * m_i * K)$.

Question (b): If we assume that the hash function maps words evenly across the buckets (i.e., each bucket gets same number of words), then what is the time complexity (big O notation) of this improved solution? Use the same parameters as Question (a) for the size of each document and size of the query set.

The improved solution's time complexity is
$O(((\sum_{i=0}^{N} D_i * m_i) / num\_buckets) * K)$
because for every query word, we must access the correct bucket (which is O(1) for a hashmap) and search the linked list within that bucket. Due to the assumption that words are evenly mapped across buckets, we know that the number of words in every bucket is the total number of words divided by the total number of buckets.

Question (c) why does this lead to a more efficient search process ?

By removing stop words with idf = 0, the size of the hashmap is reduced. That is, there are fewer elements we must search through. This is more efficient because time is not spent checking for nodes that do not need to be checked in the first place.

Question 1(d): Which of the two approaches is more efficient in terms of removing stop words and why. Which option did you choose to implement.

The second approach is more efficient for removing stop words because we only need to examine the “upper level” linked list to determine whether to remove a word (whose counts for each document are stored in a “lower level” linked list). I chose to implement a solution that is similar to the first option. It is slightly different because I reconfigured the llnode struct to contain the number of occurrences for each document rather than a single field for the total number of occurrences in all the documents. This way, I can calculate the df and idf, check if idf=0, and remove the node accordingly since each distinct word has its own singular node no matter which documents it appears in.

Note: Hash functions using a modulo N function typically use a prime number for N (the number of buckets). Why do you think this is the case ?

I think hash functions usually mod by a prime number because it may reduce the number of collisions depending on the distribution of the keys. If the keys share the same factors and also share a factor with N, then it is likely that many of the keys will be hashed to the same buckets. However, using a prime number can help minimize collisions because a prime number’s only factors are 1 and itself.