Abdullah Almaayoof
CSCI 2461
Professor Narahari

Project 5 Report

I chose both option 2's, option 2 (hashmap with list of lists) and option 2 (i.e, reading arbitrary files from a directory).

The hashmap is keyed to the word using a prime of 37 to elevate early characters in an unsigned long (preventing negative numbers and adding a bit) before being reduced to the modulus of the bucket count, ensuring a random bucket placement. While the user can choose a low bucket count, for large files or many files, a very large bucket count makes sense, as a bucket has only an 8 byte overhead on a 64 bit compile. Since the hashmap should be doing the heavy listing, no attempt was made to optimize the word or file lists to place more frequent words or files earlier. During training, the current file will be first for all duplicated words, which is optimal. Hash map functions were added to support the three level (hash-list-list) structure.

The main calls supporting functions for many behaviors, which are all in main.c except the hash map generic functions. Some hash map functions are done directly in main.c functions. I made extensive use of realloc() to create and grow variable size and unlimited structures, like the query line, query word list and result set. The result set was implemented with 2 parallel arrays of names and scores, anchored in a struct with the current length. I used a simple bubble sort on the results.

The following are the functions of my program:

hashmap.c

struct hashmap* hm_create(int num_buckets):
Function hm_create creates empty hashmap with the specified number of buckets.

struct llnode* hm_find_word(struct hashmap* hm, char* word):
Function hm_find_word returns the node supporting a word or null if the word is not in the hashmap.

Struct fnode* hm_find(struct hashmap* hm, char* word, char* document_id):
Function hm_find returns the node supporting a word and file name or null if there is no such word or file.

int hm_get(struct hashmap* hm, char* word, char* document_id):
Function hm_get returns the count of occurrences of a word in a file or -1 if the word is not in that file.

void hm_put(struct hashmap* hm, char* word, char* document_id, int num_occurrences):
Function hm_put finds or creates an entry in the hashmap for the word and file name, and sets the number of occurrences of that word in that file. It may update the number of files a word is in.

void hm_destroy(struct hashmap* hm):
Function hm_destroy frees all the storage of the hashmap.

hash(struct hashmap* hm, char* word):
Function hash creates a pseudo-random number from the characters of a word, using prime number 37 to record the value and position of every character, then reduces it by modulus of the bucket count.

int printHash(struct hashmap* hm):
Function printHash prints the contents of the hashmap in bucket order, one line for each word followed by a line for each file.

main.c

void stop_word(struct hashmap* hm, int n):
Function stop_word finds words in the hashmap that are in every file, and removes them.

int training(struct hashmap* hm):
Function training will find every word in every file in folder p5docs that matches *.txt, counting the files that sucessfully open and recording in the hashmap the words, number of files containing the word, names of files containing the word, and how many times each file contains the word. Finally, it calls stop_word to remove words in all files from the hashmap.

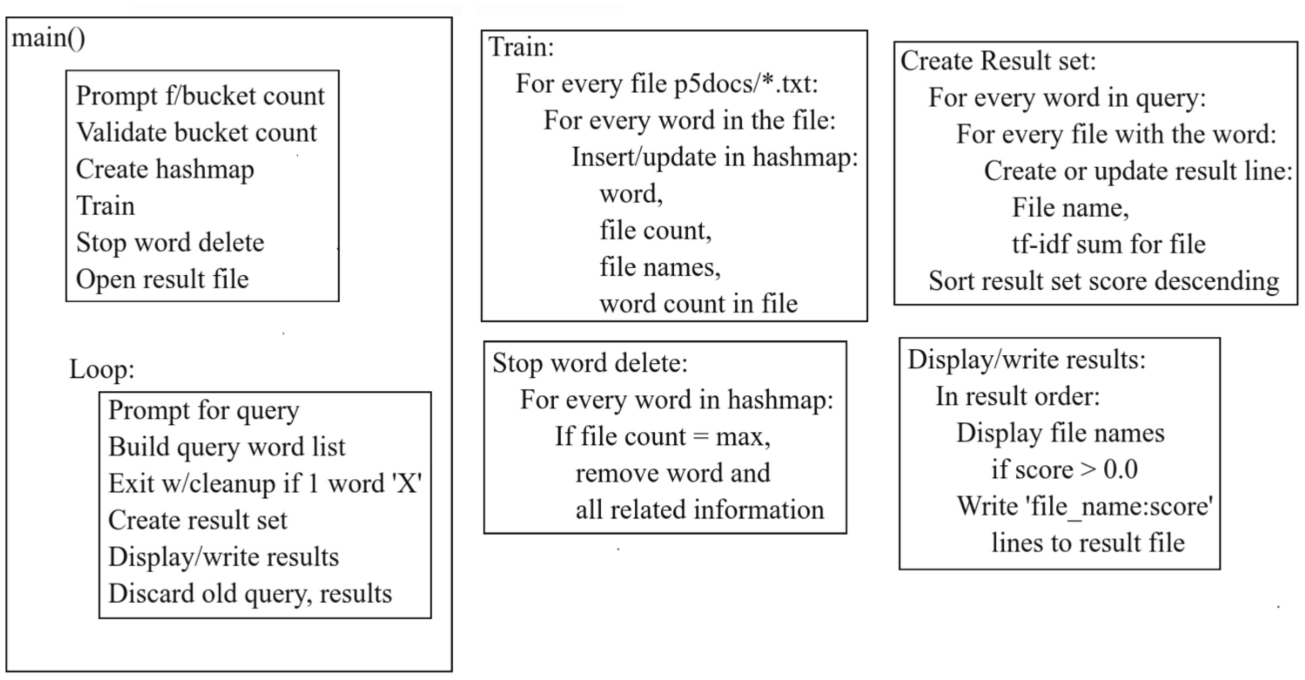
char read_query(int *len):**
Function read_query will read one line of input of any length into an expanding buffer. Then, it will parse the words in the line, copying them onto an expanding list. It returns the list and writes the length by reference.

void free_query(char q, int len):**
Free query words and the array that holds them.

double tf_idf(int tf, int n, int df):
Calculate the tf_idf.

struct ranked_files rank(struct hashmap* hm, char list, int len, int n):**
Function rank will look up every word in the list, and for every file, create or update a ranking score. Then sort the list by decreasing score. Output is a structure of parallel arrays of names and scores, and the array length.

Flow chart



```
main()
    Prompt f/bucket count
    Validate bucket count
    Create hashmap
    Train
    Stop word delete
    Open result file

    Loop:
        Prompt for query
        Build query word list
        Exit w/cleanup if 1 word 'X'
        Create result set
        Display/write results
        Discard old query, results
```

```
Train:
    For every file p5docs/*.txt:
        For every word in the file:
            Insert/update in hashmap:
                word,
                file count,
                file names,
                word count in file
```

```
Stop word delete:
    For every word in hashmap:
        If file count = max,
            remove word and
            all related information
```

```
Create Result set:
    For every word in query:
        For every file with the word:
            Create or update result line:
                File name,
                tf-idf sum for file
    Sort result set score descending
```

```
Display/write results:
    In result order:
        Display file names
            if score > 0.0
        Write 'file_name:score'
            lines to result file
```