Description

The program take an input of TREC format document, and build an inverted index table, along with a lexicon file and doc table in a I/O and memory efficient manner.

After a successful build, you can query words in the established index database.

Example TREC file: https://github.com/microsoft/TREC-2019-Deep-Learning.

Usage

Make

```
$ make # use this to compile the project
```

The executable will be created with the name 'main' under root directory. Run by ./main [options]

```
$ make clean # You can use this to clear all the output artifacts if necessary.
```

Usage

```
"-b <TREC file>  build inverted index table, lexicon and doc table from input
file(without <>). Output to output/... \n"
"-p <1~10> Load from 10 to 100 percent of data, mapping to 1 to 10 to 10% to
100%. Default to 10(100% data) "
"-m <400~8000> Memory Limit 400MB to 8GB, mapping to 400 to 8000\n . Default to
1000(1G)";
"-q <word1> <word2> query list of word(without <>).\n"

$ ./main -b msmarco-docs.trec -m 2000 # build and set memory limit to 2G (not
accurate)
$ ./main -b msmarco-docs.trec -p 1 # build upon only load 10% of data if you
like
$ ./main -q american people # must after successful built
```

Output:

Temporary files will be created under tmp/ during program and **deleted automatically** during building process.

When successful built, the final output will go to output/ with an inverted table, a lexicon and a doc table.

How it works

Building process consists of three phases: parse --> sorting --> merging.

Parse phase (main.c)

Input: Trec file

Output: N intermediate temporary file(to the location **tmp/intermediate-x**) (determined by program setting) and a term id <--> term map file (**tmp/tmp_term**)

Step:

1. Read in document with a buffer and parse like stream.
2. Parse the document between <TEXT> tags. First line is URL. Parse the rest to extract out words. Use a hashmap(term string as key) and a counter to assign each distinct word(uncapitalized during parsing) a term id, and store its doc-ids and frees in the hashmap in the form of linked-list node structure as value:

```
/*  model.h */
/*  posting model  */
//  TE(term_id) -> TC(doc_id)

// Term Entry Chunk, by doc_id
typedef struct TC{
    int doc_id; // doc id
    int freq;   // frequency of words in this doc
    struct TC* next; // next TC Node
}term_chunk;

// Term Entry
typedef struct TE{
    int total_size;
  // total size of this TE node = sizeof(term_id) + sizeof(total_sizes) +
sizeof(tc_length) + sum(full size of each TC node)
    int term_id;   // term id
    int tc_length; // TC list length
    struct TC* chunk_list_head; // TC list head
    struct TC* chunk_list_tail; // TC list tail
}term_entry
```

Whenever encountered a word, query if is already in the hashmap. If not, create a TE node and its child TC and child PN for it and put the TE node in the map. If yes, retrieve the existing TE node and check the doc-id of its TC list tail(chunk_list_tail), because doc-id is sequentially assigned, so if the tail TC has same doc-id as now --> create a PN node and append to the TC, if the tail TC has a different(must be smaller) doc-id --> create a new TC node and child PN, append it to the TE.

You can see the above structure does not contain the term itself. The actual entry node stored in hashmap is as follows:

```
/*  main.c  */
/*  Hash Map  */
/* store mapping between term_id and term during parsing, to ensure same string
map to the same id */
struct hashmap *term_map;

typedef struct
{
   char key_string[KEY_MAX_LENGTH];
   int number; // term id
   term_entry* te;
} map_entry;
```

3. The hashmap will grow very large over time. So set a limit for the memory of hashmap. If exceeded, writing the TE node information of every node in hashmap to a **separate intermediate file** while keeping map_entry node because we still need it for distinct-word purpose, yet we can free the TE node after writing out.

   Intermediate posting file format:

   (Length{4 bytes}, Term ID{4}, Doc List Length{4}, [(Doc ID{4}, Freq{4}) , (Doc ID{4}, Freq{4}) ,...]) (...) ...

   - **Length:** the total length of this unit.
   - **Doc List Length:** number of doc IDs in its TC list
   - **Freq:** frequencies of the word in this document

   So it is just a little-compressed format of TE node structure. The space efficiency is not good because every word will at least take 4 bytes in this format so in general the result inverted table will be as large as original file. It is something I need to compress in the next assignment.

So in the end, there will be many intermediate files(0 - n). If i > j, doc-ids in inter-i are larger than doc-ids in inter-j.

4. Term <--> Term ID map:

   This mapping can be output(to a buffer, then flush to a file if full) every time encounter a new word.

   Term id map file format:

   Total Terms{4}, (Term ID{4}, Term Length{4}, Term{determined by Term Length}) (...) ...

   - **Total**: total number of terms. This is for convenience of reading because the reader end can know the length of array it needed to store all the records.
   - **Term Length:** Length of the term string, which determines the length of the Term field.

5. Doc table

   Doc table is output during document parsing with a standalone output buffer.

   Doc table format

   Total Size{4}, (Unit Length{4}, Doc_ID{4}, Doc_Size{4}, URL{determined by Unit Length}) (...) ...

   Doc_Size: length of original document

Sort phase (sort_phase.c)

Input: N intermediate files, tmp_term file(containing the map between term id and term)

Output: N sorted intermediate files

Read in each intermediate file and transform the binary into struct model described earlier for convenience of sorting. Read in term id map into an array instead of a hashmap, because only id -> term is needed during sorting. Also in this step I assume the memory can fit in one intermediate file.

Merge phase (merge_phase.c)

Input: N sorted files, term id mapping already readed during sort phase

Output: 1 merged file containing sorted and no duplicate term-id record

Use merge sorted lists algorithm w/ heap, but improves the I/O efficiency by assigning a buffer for each input files and final output.

Also, in this step we can easily merge every two record with the same term id which were dispersed across different intermediate files in the first phase because of the sorted nature of each input file and the usage of heap. So aftter this step, we have 1 merged and sorted file with no duplicate term-id record.

Build phase (final_build.c)

Input: 1 merged file, term id mapping

Output: 1 inverted index file, 1 lexicon , 1 doc table(already created in phase 1)

Final inverted index is compressed by **simple var bytes compression** and **difference compression**.

Final inverted index file format:

(Doc List Length{**var**}, [Doc ID{**diff+var**}, Freq{**var**}) , (Doc ID{**diff+var**}, Freq{**var**}]) (...) ...

Lexicon file format:

(Term Length{4}, Term{determined by Term Length}, Offset{8}) (...)

- **Offset:** the offset is the record offset in the inverted list of this term.

Doc table format:

Total Size{4}, (Unit Length{4}, Doc_ID{4}, Doc_Size{4}, URL{determined by Unit Length}) (...) ...

Query(query.c + lexicon.c)

1. Read in the whole lexicon file to a hashmap with structures below.

```
/*  query.h  */
/*  Hash Map  */
// read from lexicon to a map in memory , used for query
struct hashmap * query_map;

typedef struct
{
   char key_string[MAX_KEY_LENGTH+1];
   int offset;
} lexicon_el;
```

2. Read in the doc table to an array structure.

```
doc_entry** doc_table; //doc_entry array
int doc_table_L = 0;

bool read_doc_file(struct hashmap* query_map){
    FILE* f_doc = fopen("output/doc_table","r");
    file_buffer* fb = init_dynamic_buffer(1000000);
    fb -> f = f_doc;

    if(f_doc == NULL)
        return false;
    int L = 0;
    fread(&L, sizeof(int), 1, f_doc);

    doc_table = (doc_entry **)malloc(L*sizeof(doc_entry*));
    for(int i = 0; i < L; i++){
        doc_entry* de = (doc_entry*)malloc(sizeof(doc_entry));
        next_doc(fb,de);
        doc_table[de->doc_id] = de;
    }
    doc_table_L = L;
```

```
    free_file_buffer(fb);
    fclose(f_doc);
    return true;
}
```

3. Get the offset of the query word from the hashmap if exists. And use **fseek**() to directly access record in inverted list.

```
/* model_support.c */
void read_inverted_list(FILE* f, int offset, int length, term_entry* te);
```

Buffer

```
/*  model.h */
/*  Dynamic Buffer  */
typedef struct{
    FILE* f;
    char* buffer;
    int size;
    int curr;
    int max;
}file_buffer;

/*  Operate the term model by dynamic buffer */
file_buffer* init_dynamic_buffer(int size);
void free_file_buffer(file_buffer* fb);
bool next_record_from_file_buffer(file_buffer* fb, term_entry* te);
bool write_record(file_buffer* fb, term_entry* te, bool flush);
void flush_buffer_to_file(file_buffer* fb);
```

Designed a structured dynamic buffer because there are many cases that need buffer. So bind buffer to the file pointer.

Compression

```
#include <stdlib.h>
#include <stdio.h>

unsigned char* vb_encode(int n, int* L ){
    int m = n;
    int d = 0;
    if( m == 0 )
```

```
        d = 1;
    while(m != 0){ m/=128; d++;}
    unsigned char* bytes = (unsigned char*)malloc(d);
    for(int i = 0; i < d; i++, n /= 128)
        bytes[i] = n % 128;

    bytes[d-1] += 128;
    *L = d;
    return bytes;
}

int vb_decode_stream(FILE* f){
    int r = 0,n = 0, k = 1;
    unsigned char c = 0;
    do{
        fread(&c, 1, 1, f);
        n = c % 128;
        r = n*k + r;
        k = k*128;
    }while(c < 128 && !feof(f));
    return r;
}
```

Implemented simple var-byte compression.

Result

On input of 22GB TREC file

Option:

```
-m 1000 : approxi~ limit memory to 1GB ~ 1.5GB
```

Word length limit setting in lexicon.h :

```
#define WORD_LENGTH_MIN 1
#define WORD_LENGTH_MAX 15
```

CPU: 2.7 GHz Intel Core i7

Parse phase speed: ~40 minutes

Intermediate Files Sizes: 9.55 GB

Distinct words: 9962720

Sort phase : 5 minutes

Merge phase : 4 minutes

Build phase: 8 minutes

Inverted Index File Size: 2.45 GB

Lexicon File Size: 170.5MB

Doc Table Size: 247.7MB

Query speed: 3 seconds (Due to urgency, only tested on -p 1 for query function now)

Memory Check

```
==54418== LEAK SUMMARY:
==54418==    definitely lost: 0 bytes in 0 blocks
==54418==    indirectly lost: 0 bytes in 0 blocks
==54418==      possibly lost: 8,840 bytes in 7 blocks
==54418==    still reachable: 14,880 bytes in 162 blocks
==54418==         suppressed: 0 bytes in 0 blocks
```

Limitation

Time: Building is still slow. Should combine sort with first step. Query speed is not optimized.

Space: Simple var byte compression is not good enough. Should seek better compression. Lexicon and Doc table are not well compressed.

Index maintainability: Insert/Delete operations on index table are not considered.

Query: Does not support complex query now. Current structure are not good enough for query. Would be better to use fixed length block.