Due Thursday, 2000 Feb 10, at the beginning of class.

Contents

1 Reading

Read chapter 2 of the textbook.

2 Homework Solution Tip

Solving this homework will probably require as much preparation time as time spent programming. (Our solution required writing approximately two pages of code.)

Here is one way to approach a homework that has more pages of explanation than lines of code.

1. In a first pass, skim through the homework, trying to understand how the search engine works and what you are required to do. Skim through the provided code,

particularly the comments.

2. In a second pass, read carefully, drawing pictures of the search engine and how information moves through it. Try to understand the math as much as possible. In your drawing, try to determine what code you need to write.

3. Now that you have identified what the task is, carefully read through that code, trying to understand what the variable types are, what code is provided, and what code needs to be written.

4. With your classmates, repeat the above process.

5. With your classmates, plan what functions you will write. Using English, describe what each function should do.

6. With your classmates, determine how you will test your code.

7. With your classmates, write throw-away programs using the STL containers described below.

8. On your own, write the necessary code. If syntax errors halt your progress, ask a classmate, working on another computer, of course, for help.

3 Programming Tip

When trying to use code you have never previously used, try testing it out in a small "throw-away" program. Doing so helps ensure that any difficulties come from the new code, not from other parts of a larger program. Writing small throw-away programs may seem like a waste of time, but, for me, they actually save time. Here is a code template for a throw-away program.

4 Some Useful STL Containers

We provide a short explanation for four STL containers that we use for the search engine:

`pair`: a group of two values

`vector`: an array that can grow and shrink

`string`: string of characters that can grow and shrink

hash table: table permitting quick lookup

4.1 `Pairs`

The C++ Standard Template Library uses `pairs` to group together two values. For example, one can group together a string and an integer using `pair<string,int>("hello",3)`. Here is some example code.

Constructors: There are at least three different ways to create pairs. To create an empty pair, use

```
pair<double,int>()
```

To specify the pair's contents upon creation, use

```
pair<double,int>(3.4,12)
```

If you do not want to explicitly write the pair's types, you can use

```
make_pair(3.4,12)
```

and the compiler will take its best guess about the pair's types.

Predicates: none

Selectors: To access a pair p's first item, use

```
p.first
```

To access a pair p's second item, use

```
p.second
```

Be sure to #include <utility> near the top of your file.

4.2 Vectors

vectors are arrays that can change their size. Anything one can do to an array, one can also do to vectors. Here is some example code.

Constructors: To create a vector of size zero that can hold elements with type T, use

```
vector<T> v
```

To create a vector of size n, use

```
vector<T> v(n)
```

To change a vector so it can hold n elements, use

```
v.resize(n);
```

After doing this, accessing positions 0, 1, ... , $n-1$ is legal.

Predicates: If a vector v is empty,

```
v.empty()
```

yields true. If

```
v.size()
```

yields s, then positions 0, 1, ... , $s-1$ may be accessed.

Selectors: To access or change the element at position i, assuming the vector has size at least $i+1$, use

```
v[i]
```

Alternative equivalent syntax for accessing, but not changing, the element is v.at(i). This has the nice feature that, if the vector does not have a position i, it kills the program rather than just returning a garbage value.

```
v.push_back(item)
```

enlarges the vector by one position, inserting item into the last position.

```
v.pop_back(item)
```

shrinks the vector by one position, eliminating item from the last position.

v.begin() and v.end() yield iterators for the vector's beginning and one past the end.

Be sure to #include <vector> near the top of your file.

4.3 Strings

strings are like C-style strings but with more operators and no maximum length. Anything one can do to C-strings, one can do to strings and more. Be sure to #include <string>.

code	meaning
string s;	creates a string with no characters
string t("hello");	creates a string containing "hello"
string t = "hello";	another way of doing the same thing
s = t;	makes s equal "hello"
cout << s[1];	prints the letter 'e'
s = "good";	changes s's contents
s = s + "bye";	changes s to "goodbye"
cout << s + t;	prints "goodbyehello"
s.empty();	yields false because s has characters
t.size();	yields 5 because it has five characters
t.push_back('s');	appends s to "hello"
t.clear();	shrinks t to the empty string
t.c_str();	converts t to a C-string.

When using the .c_str() function to convert from a string to a C-style string, be sure to use the result immediately. It may "magically" disappear by the time the next statement is executed. This function is seldom needed, but it is useful when using the .open(*filename*) function for istreams and ostreams, which takes only const char [].

4.4 Hash Tables

Hash tables (textbook, ch. 12) permit quickly finding a pair by specifying the pair's first component. In our search engine, we use a hash table to map from a word to its position in a vector. For example, suppose our hash table is called ht and vector position 12 corresponds to "bonjour", i.e., the pair ("bonjour",12). Here is C++ code to determine "bonjour"'s vector position.

The .size() member function yields the number of elements in the hash table. Be sure to #include <hash_map> near the top of your file.

5 Problem Statement

The problem is to finish writing a simple web search engine. Before describing what code needs to be written, we present the model and the algorithmic ideas.

5.1 The Mathematical Model

We call two web documents similar if they contain many of the same words used with similar frequency. Each web document is modeled using a very long vector, with each vector component representing the occurrence frequency of a particular word in the document. For example, if the word "molasses" occurs twice as frequently as the word "jam," the molasses component will be twice as large as the jam component.

Technically, two web documents are *similar* if the angle between them is small. To understand what this means, first consider the dot product of two vectors. The dot product of two vectors is the sum of the pairwise multiplication of vector components. For example, $(3,4,5) \cdot (6,7,8) = 3 * 6 + 4 * 7 + 5 * 8$. The dot product is large if the two documents have many of the same words. For the computation, we actually use the relative frequency of words within a document, e.g., "molasses" forms 20% of the document's words while "jam" forms 10%. Using the formula for dot product $A \cdot B = |A||B| \cos\theta$, we see that the angle θ is small if $(A/|A|) \cdot (B/|B|)$ is large.

5.2 Search Engine Algorithms

The two parts of a search engine are:

1. Preprocessing the documents to produce a vector for each web document.
2. Given a list of search words, finding the closest web documents.

Preprocessing the documents requires collecting the documents, extracting the documents' words for use as vector components, and then computing each document's vector. For this homework, we just used the `wget` command to snarf a collection of documents. We then collected all of the documents' words into a hash table and finally converted each document into a vector.

To convert a document into a vector, for every word we read from a document, we increment the word's component in the vector. To determine the component number, we ask the hash table for the word's component. For example, if the hash table is called `ht`, we can determine `hello`'s component (an integer) using `ht.find("hello")`. Then, we normalize the vector A by scaling by the reciprocal of its length $|A| = \sqrt{A \cdot A}$. That is, we multiply every component of A by $1/|A|$.

For example, if a document contains only the words "bonjour" (three times) and "hello" (four times) and the components of "bonjour" and "hello" are 12 and 20, respectively, then the unnormalized document vector will have a 3 in component 12 and a 4 in component 20 and zeroes everywhere else. (The number of vector components depends on the number of words in the hash table.) The normalized vector then has 0.6 in component 12 and 0.8 in component 20 and zeroes everywhere else.

Given a list L of search words, we wish to determine the closest web documents. To do so, we first construct a search vector from L. For each word w in L, we use the hash table to increase w's component by one.[1] Then, we normalize by scaling by the reciprocal of its length. A document is similar if its dot product with the search vector is large. Search words not in the hash table can be ignored.

[1] Although I had not previously considered this, I suppose a user could type the same search word repeatedly. For example, searching for "hello hello hello hello hello goodbye goodbye" specifies that "hello" is 2.5 times more important than "goodbye". Is adding the importance of each word to a search query a useful feature? Do any current search engines provide this feature?

5.3 Coding the Search Engine

We will provide software for preprocessing sets of documents, the results of the preprocessing, and a few sets of example documents. Your job is to finish writing the code that queries the user for search words, determines which documents are most similar, and prints the results.

For each set of documents we provide, we will provide a file containing the set's words and, for each document, the vector components. We will also provide code to read in the file, storing its contents in a hash table and in a vector of document-vector pairs with one component per document.

Although you probably do not need to know how to preprocess the documents to complete the assignment, we describe it here for completeness and so you can process your own set of documents if you desire. The `prepareDatabase` program takes one or two command-line arguments:

1. the name of a file listing all the documents to be included in the database. Documents are specified by filename.

2. optionally, a prefix to affix to the beginning of each document filename. For example, if the document's filename is "zmrcs10.txt" and the prefix is "ftp://metalab.unc.edu/-pub/docs/books/gutenberg/etext99/" then the combined name is "ftp://metalab.-unc.edu/pub/docs/books/gutenberg/etext99/zmrcs10.txt," which looks a lot like a web address.

This program preprocesses the documents, sending the keywords it selects and each document's vector to the standard output. Save this output in a file to give to your search engine.

Your job is to finish the search engine code that queries the user for search words, compute how close each document is to the search vector, and prints the closest documents. More specifically,

1. Read the document database information into the `KeywordMapping` hash table and the `vector<DocVec>` collection of documents and their vectors.

2. Prompt the user for any positive number of search words (allowing repeats) as well as the desired number n of similar documents.

3. Find the n closest documents, printing them in order from most similar to least similar. It would also be nice to print their scores.

Code is provided for some portions of these tasks.

5.3.1 Types

Using STL containers can easily lead to very long names for types. For example, `hash_map <const string, vector<double>::size_type>` is the type of the hash table translating words to vector components. Instead of typing this forty-eight character type name, we say

```
typedef hash_map<const string, vector<double>::size_type> KeywordMapping;
```

creating a new equivalent type named `KeywordMapping`. Thus, declaring variables with a type of `KeywordMapping` is the same as using the forty-eight character type.

(The syntax for the type definition statement `typedef` is `typedef` *`type new-synonym`*`;`

)

types.h contains several type definitions:

`DocVec:` is a `pair` of a document's name, which is a `string`, and its `vector`. A document vector contains `doubles`. We guarantee that every component has a value.

`DocScore:` is a `pair` of a document's name, which is a `string`, and its dot product with the search vector, a `double`.

`KeywordMapping:` is a hash table translating a word, i.e., a `string`, to its component in a vector if the word is in the hash table.

5.4 What Files Do I Need?

Your job is to finish writing the search engine code in search-engine.cc. You will also need the type declaration file types.h. Be sure it is called "types.h" and is in the current directory. To compile, use a command similar to `g++ -Wall -pedantic search-engine.cc -o search-engine`.

If you want to process your own set of documents, download prepareDatabase.cc and types.h. Compile using a command similar to `g++ -Wall -pedantic prepareDatabase.cc -o prepareDatabase`.

To ease compilation, use the Makefile. To create an executable called `search-engine`, use

```
make search-engine
```

5.4.1 Sample Document Sets

You will need to generate your own data to test your code, but we have provided three sets of documents and one program to generate random sets.

- The hello-goodbye set has three documents, mostly consisting of the words "hello" and "goodbye."
- The etext90 and etext91 sets are etexts from the Project Gutenberg.
- The generate program produces a set of randomly generated documents. Read the use comments at the beginning of the program.

Each set's database file for use with `search-engine` has a ".db" suffix. Compressed archive files ending with a ".tgz" suffix are provided in case you want to copy a set of documents to your *own* computer. To extract the files, use the command `tar xzvf` *`filename`*.

Please do not copy the large set of documents to your home directory on the computer science computers. If any significant fraction of students do so, the computer science disk will quickly fill up. Instead copy just the database file to your home directory. If you really do want all the files, please store them in the directory called /tmp so that 1) they will not fill the computer science disk and 2) they will automatically be erased when the machine is rebooted.

6 How Our Search Engine is Simpler than Commercial Engines

Our search engine uses one approach to solve the most important and most difficult task performed by search engines: determining which web documents are closely related to each other. Our code, however, uses a very simple ranking scheme similar to what Altavista probably used to use. More complicated ranking schemes can yield more usable results such as those returned by Google. Also, our preprocessor makes a very limited attempt to filter out uninteresting words by omitting all words of three or fewer characters but does not stem suffixes from words such as "played" and "playing" so that they match. It makes a heuristic attempt to remove punctuation and ignore case. We also do not filter unacceptable web documents from the document pool.

Commercial search engines usually must accept more complicated input syntax such as boolean operators, must have at least 99.9% uptime, be able to handle large number of simultaneous queries, and deal with network issues.

7 Submission Rules

Please submit only your completed search-engine.cc. You need not send any other files. Please send only text documents, do *not* send Microsoft Word documents, PDF documents, HTML documents, etc.

We will test your code on our sample data. Please be sure it compiles without warning when using `g++ -Wall -pedantic`.

See the submission details for information how to email the programs.