CSCI 1321 (Principles of Algorithm Design II), Fall 2001

Homework 3[1]

Assigned: September 27, 2001.

Due: October 9, 2001, at 5pm. *Not accepted late.*

Credit: 40 points.

> *Note:* The HTML version of this document may contain hyperlinks. In this version, hyperlinks are represented by showing both the link text, formatted like this, and the full URL as a footnote.

Contents

1 Hints and tips **2**
- 1.1 How to approach this homework . . . 2
- 1.2 Programming tip . . . 2

2 Some useful STL containers and functions **2**
- 2.1 Pairs . . . 3
- 2.2 Vectors . . . 3
- 2.3 Strings . . . 5
- 2.4 Hash tables . . . 5
- 2.5 Sorting . . . 6

3 Problem statement **6**
- 3.1 The mathematical model . . . 6
- 3.2 Search engine algorithms . . . 7

4 Details **7**
- 4.1 Coding the search engine . . . 7
- 4.2 Types . . . 8
- 4.3 What files do I need? . . . 9
- 4.4 Sample document sets . . . 9

5 What to submit and how **10**

6 Miscellaneous tips and remarks **10**
- 6.1 How our search engine is simpler than commercial engines . . . 10
- 6.2 Using the `prepareDatabase` program . . . 10
- 6.3 Where to find out about STL classes and functions . . . 11

[1] © 2001 Jeffrey D. Oldham (`oldham@cs.stanford.edu`) and Berna L. Massingill (`bmassing@cs.trinity.edu`). All rights reserved. This document may not be redistributed in any form without the express permission of at least one of the authors.

1 Hints and tips

1.1 How to approach this homework

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.

- 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*. You should be able to understand what each of the provided functions does simply by reading the comments preceding its declaration; reading the implementation should be optional.
- 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.
- 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.
- Plan what functions you will write. Using English, describe what each function should do.
- Determine how you will test your code.
- Write throw-away programs using the STL containers described below.
- Write the necessary code.

Remember that discussions with classmates are allowed, up to the point at which you begin writing code.

1.2 Programming tip

When trying to use code (e.g., a library function or class) you have never previously used, try testing it out in a small "throw-away" program. Doing so helps ensure that, when you use the library function or class in a larger program, any problems are caused by new code in the larger program and not by your imperfect understanding of how to use the library code or class. Writing small throw-away programs may seem like a waste of time, but in fact in the long run it can save time. Examples of such throw-away programs are sample programs `vector-use-example.cpp`[2] and `pair-use-example.cpp`[3].

2 Some useful STL containers and functions

The C++ Standard Template Library (STL)[4] defines a number of useful container classes. This section describes four of these classes that we use for the search engine, namely:

- `pair`: a group of two values.

[2] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/vector-use-example.cpp`
[3] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/pair-use-example.cpp`
[4] `http://www.sgi.com/tech/stl/`

- `vector`: an array that can grow and shrink.
- `string`: a string of characters that can grow and shrink.
- hash table: a table permitting quick lookup.

It also describes the STL `sort()` function, which you may find useful.

2.1 Pairs

The STL provides a class `pair`[5] to group together two values. For example, one can group together a string and an integer using `pair<string, int>("hello", 3)`. See `pair-use-example.cpp`[6] for more simple examples of use. Be sure to `#include <utility>` in your program.

Creating pairs

There are at least three different ways to create a `pair`.

- 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.

Working with pairs

- To access a pair `p`'s first item, use

  ```
  p.first
  ```

- To access a pair `p`'s second item, use

  ```
  p.second
  ```

2.2 Vectors

An STL `vector`[7] is an array that can change size. Anything one can do to an array, one can also do to a `vector`. See `vector-use-example.cpp`[8] for more simple examples of use. Be sure to `#include <vector>` in your program.

[5] `http://www.sgi.com/tech/stl/pair.html`
[6] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/pair-use-example.cpp`
[7] `http://www.sgi.com/tech/stl/Vector.html`
[8] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/vector-use-example.cpp`

Creating vectors

- To create a vector of size zero that can hold elements of type `T`, use

  ```
  vector<T> v
  ```

- To create a vector of size n, use

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

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

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

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

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

Working with vectors

- If a vector `v` is empty,

  ```
  v.empty()
  ```

  yields `true`; otherwise it yields `false`.

- If

  ```
  v.size()
  ```

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

- 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.

- The code

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

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

- The code

  ```
  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.

2.3 Strings

An STL `string`[9] is like a C-style string (array of `char`) but with more operators and no maximum length. Anything one can do to a C-style string, one can do to a `string`, and more. Below are some examples of use; see `string-use-example.cpp`[10] for more examples. 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";`	Creates a string containing "hello".
`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-style 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 an `istream` or `ostream`, which requires as input a C-style string (array of `char`), not a `string`.

2.4 Hash tables

Note: You probably will not need to understand the material in this section to complete the homework, if you read the comments for the provided functions carefully. This material is provided to help those who want to understand more about the implementation of these functions.

Hash tables (described in chapter 12 of the textbook) 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:

```
hash_map<string,int> hm;        // create the hash table
// code for storing values in the hash table omitted
hash_map<string,int>::const_iterator pos = hm.find("bonjour");
                                // try to find "bonjour"
if (pos == hm.end())
  cout << "bonjour not in the hash table\n";
else
  cout << "bonjour's int is " << (*pos).second << ".\n";
```

[9] `http://www.sgi.com/tech/stl/basic_string.html`

[10] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/string-use-example.cpp`

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

2.5 Sorting

The STL `sort()`[11] function sorts all or some of the elements of a suitable container class into ascending order. Here is an example of using this function to sort a `vector`:

```
vector<int> v(10);
// code to put elements into v omitted
sort(v.begin(), v.end());
```

The two parameters passed to `sort()` are STL iterators; the first one points to the first element to be sorted, and the second points just past the last element to be sorted. `begin()` and `end()`, as the code suggests, are member functions of the `vector` class.

The function does require that there be a sensible `<` operator defined for the elements to be sorted. This will be the case for simple types (e.g., `int` or `double`) and some library classes (e.g., `string`). If there is no appropriate `<` operator, you can write your own comparison function and pass it to `sort()` as a third parameter. This function should behave like less-than, taking two parameters (the objects to be compared) and returning a `bool` whose value is `true` if the first object is "smaller" (should appear first in a sorted list) and `false` otherwise. See `print-sorted.cpp`[12] for an example of using this feature to do a case-insensitive sort of strings.

Be sure to `#include <algorithm>` in your program.

3 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.

3.1 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.

[11] `http://www.sgi.com/tech/stl/sort.html`

[12] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/SamplePrograms/print-sorted.cpp`

3.2 Search engine algorithms

The two parts of a search engine are:

- Preprocessing the documents to produce a vector for each Web document.
- 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")`. We then 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 $1/|A|$.

(*Aside:* Observe that in the above discussion "vector" means a mathematical vector, not C++ `vector`. A C++ `vector` is like an array, and its length is the number of elements. A mathematical vector is a sequence of numbers, and its length is as defined in the preceding paragraph. Its number of elements, in contrast, determines its dimensionality — vectors with two elements are two-dimensional, vectors with three elements are three-dimensional, etc.)

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, a 4 in component 20, and zeroes everywhere else. (The number of vector components is determined by the number of words in the hash table.) The normalized vector then has 0.6 in component 12, 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. 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.

(*Aside:* Observe that a user could type the same search word repeatedly. The effect will be to make the repeated words more significant in finding matches — for example, searching for "hello hello hello hello hello goodbye goodbye" specifies that "hello" is 2.5 times more important than "goodbye". It is not clear that this is a useful feature, and we do not know whether current search engines provide such a feature. We will agree to simply allow the user to repeat words and let the chips fall where they may.)

4 Details

4.1 Coding the search engine

We provide software for preprocessing sets of documents, the results of the preprocessing, and a few sets of example documents. We also provide a partially-written program, including the code

required to read in the database file and store its contents in a hash table and in a vector of document-vector pairs with one component per document. 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.

The incomplete program is search-engine.cpp[13]. This program takes one command-line argument (the name of the file containing the database information); it prompts the user to enter search words, computes how close each document in the database is to the search vector, and prints those that are closest. More specifically, this code is supposed to do the following:

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 and print 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, in the form of several functions with extensive comments describing their pre- and postconditions. Review these comments before starting to write your own code; our sample solution makes use of all of the provided functions. To see where you must, or may, add code, look for comments containing the word "ADD", for example

```
// ====> ADD code here <====
```

4.2 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 48-character name, we can say

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

and create a new equivalent type named `KeywordMapping`. Thus, declaring variables with a type of `KeywordMapping` is the same as using the 48-character type.

The syntax for the type definition statement `typedef` is

`typedef` *type new-synonym*;

types.h[14] contains several type definitions, including the following:

- `DocVec`: A `pair` consisting of a document's name (a `string`) and its vector (a `vector<double>`). We guarantee that every element in the vector has been assigned a value.

[13] http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Homeworks/HW03/Problems/search-engine.cpp
[14] http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Homeworks/HW03/Problems/types.h

- `DocScore`: A `pair` consisting of a document's name (a `string`) and its dot product with the search vector (a `double`).
- `KeywordMapping`: A hash table that translates a word (a `string`) into its component in a vector, assuming the word is in the hash table.

4.3 What files do I need?

First, you will need the incomplete search-engine program `search-engine.cpp`[15]. You will also need the type declaration file `types.h`[16]. Be sure it is called `types.h` and is in the same directory as `search-engine.cpp`. 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.cpp`[17]. and `types.h`[18]. Compile using a command similar to

```
g++ -Wall -pedantic prepareDatabase.cc -o prepareDatabase
```

and refer to the last section of this document for instructions on how to use the program.

To ease compilation, you can use a makefile, as described in the preceding homework.

4.4 Sample document sets

To help you test your code, we provide three sets of documents and one program to generate random sets. For each set of documents we provide, we provide a file containing the set's words and, for each document, the vector components. I.e., each set contains a database file (suffix `.db`) for use with `search-engine`, plus the original text files used to create it, and a compressed archive file (suffix `.tgz`) in case you want to copy a set of documents to your *own* computer (to extract the files, use the command `tar xzvf` *filename*).

For each set, the only file you need to copy into your own directory is the one ending `.db`. ***Please*** *do not copy the entire set(s) of documents into your home directory on the department's computers; this wastes space on the file server's disk and on the daily backup tapes.* If you really do want all the files, please store them in the directory called `/tmp` so that (1) they do not take up space on the file server's disk and (2) they will automatically be erased when the machine is rebooted.

Here are the document sets and program:

- The `hello-goodbye`[19] set has three documents, mostly consisting of the words "hello" and "goodbye".

[15] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Homeworks/HW03/Problems/search-engine.cpp`
[16] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Homeworks/HW03/Problems/types.h`
[17] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Homeworks/HW03/Problems/prepareDatabase.cpp`
[18] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Homeworks/HW03/Problems/types.h`
[19] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Homeworks/HW03/Problems/SampleData/hello-goodbye`

- The `etext90`[20] and `etext91`[21] sets are etexts from Project Gutenberg[22].
- The `generate` program (source `generate.cpp`[23]) produces a set of randomly generated documents. Read the use comments at the beginning of the program to see how to use it.

5 What to submit and how

Submit your source code as described in the Guidelines for Programming Assignments[24]. For this assignment, use a subject header of "cs1321 hw3", and submit a single file containing your revised version of `search-engine.cpp`. You do not need to send `types.h` or any of the database files; I will provide these files when I test your code.

6 Miscellaneous tips and remarks

6.1 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[25] probably used at one time. More complicated ranking schemes can yield more usable results such as those returned by Google[26]. 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 must accept more complicated input syntax such as boolean operators (usually), have at least 99.9% uptime, be able to handle large number of simultaneous queries, and deal with network issues.

6.2 Using the `prepareDatabase` program

You probably do not need to know how to preprocess the documents to complete the assignment, but 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:

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

[20] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Homeworks/HW03/Problems/SampleData/etext90`
[21] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Homeworks/HW03/Problems/SampleData/etext91`
[22] `http://www.gutenberg.net/`
[23] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Homeworks/HW03/Problems/generate.cpp`
[24] `http://www.cs.trinity.edu/~bmassing/CS1321_2001fall/Notes/pgmguidelines/index.html`
[25] `http://www.altavista.com`
[26] `http://www.google.com/`

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. (The `.db` files we provide were generated in this way.)

6.3 Where to find out about STL classes and functions

Appendix H of our textbook briefly documents many of the STL classes and functions we will use in this course. For more details, see SGI's online Standard Template Library Programmer's Guide[27].

[27] `http://www.sgi.com/tech/stl/`