University of Toronto
csc228s— **File Structures and Data Management, Spring 2001**

Assignment 2 — A Graph of the Web

Deadline A: Thursday 15 February.
Deadline B: Thursday 08 March.
Total weight: 15% of your course grade. Part A is worth a small fraction of this total weight.

Suppose you work for a company that is developing a fabulous new web search engine. One of the problems with many search engines is that they return way too many matches for a query, and the useful ones are lost in the clutter. Your company needs a way to rank the matches so that the user can be presented with only the "best" ones in some sense.

One way to do this is to give a higher rank to pages that are popular (i.e., that are linked to by many other web pages) and to pages that are linked to by pages that *themselves* have a high rank. In fact this is how Google ranks pages[1].

In order to compute the rank of all web pages, one would need to store a graph that represents the link structure of the web: what pages link to what other pages. You are going to write a program that can build this sort of graph. You will test your program using relatively small test data, but of course a graph of the entire web would be enormous, so the program must store the graph in a file.

Input: A troll file

Your company has a tool that regularly trolls the internet following links from web page to web page. For each web page that it visits, it records the URLs of all the web pages that this page links to. (Notice that links are directional. Just because page x contains a link to page y, doesn't mean that page y has a link back to x.)

The trolling program logs its results as it goes in a **troll file**. A new, huge, troll file is produced each month. The troll file consists of variable-length records, each of which contains the following, in order:

- A vertical bar character. This makes it easy to identify the beginning of each record.
- The URL ("Uniform Resource Locator") for a web page, such as `http://www.cs.toronto.edu/~dianeh/pubs.html`. This is a variable-length field delimited by a newline character.
- A set of URLs to which this page links, in no particular order. Again the number of URLs logged varies by web page. These URLs are separated from each other by a newline character.

No URL in the troll file is longer than 80 characters. As far as your program is concerned, a URL is just a string; you will never have to pull URLs apart.

A small sample troll file appears in figure 1. Figure 2 shows a graphical representation of the web pages that generated this troll file.

[1] Page rank corresponds to an eigenvector. So pay attention in linear algebra!

```
|http://www.cs.uofa.edu/~fred
http://www.cs.uofa.edu/~toni
http://www.cs.uofa.edu/~tim
|http://www.cs.uofa.edu/~toni
http://www.cs.uofa.edu/~tim
http://www.cs.uofa.edu/~fred
http://www.cs.uofa.edu/~jan
|http://www.cs.uofa.edu/~jan
http://www.cs.uofa.edu/~toni
|http://www.cs.uofa.edu/~marsha
http://www.cs.uofa.edu/~toni
|http://www.cs.uofa.edu/~michelle
|http://www.cs.uofa.edu/~diane
http://www.cs.uofa.edu/~jim
http://www.cs.uofa.edu/~michelle
http://www.cs.uofa.edu/~marsha
```

Figure 1: Sample troll file.

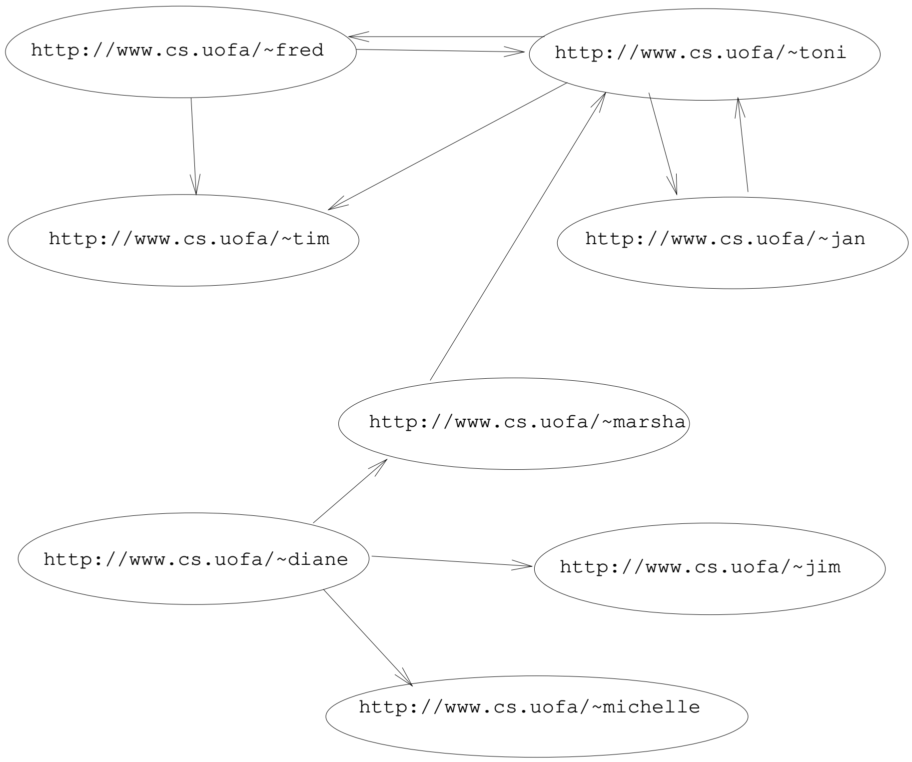

Figure 2: Web pages that were trolled to produce this troll file.

Output: Graph file and index file

The troll file contains all the information needed to compute page rank, but in a very inconvenient form. The team writing the page rank code needs to be able to quickly find the record for a given URL and then look at the records for all the the web pages to which it links. Your job on this assignment is to write a program that transforms the information in the troll file into a form that makes such searches much more efficient. Specifically, your company wants two files to be produced from a given troll file.

Webpage graph file

This file contains a graph of the web pages mentioned in the troll file. Each node in the graph will represent a web page. The node for page p will contain of the URL for page p and a list of pointers to the nodes for the web pages that p has links to. Instead of storing the URLs of these neighbouring web pages, you will store a pointer to their records in the graph file. (Think about why this might be preferred.) These pointers need not be in any particular order.

You cannot make any assumption about the maximum number of neighbouring web pages that the troll file will contain for a given web page. However, your company has decided to limit the records in its web graph file to at most 1000 bytes. If the information in the troll file would cause a graph node to exceed 1000 bytes, just cut off the excess information[2] (but not in the middle of a URL).

1000 bytes is an upper limit on the size of a record, but many records will be much smaller. However when you first create a record you may not know about all the links from that page. This means you may want to expand a record. For this reason your company has decided to use fixed width records of 1000 bytes. You know that the maximum URL length is 80 characters, but many URL's will be much smaller. Therefore, you should design your file layout to use variable-width fields within the records. Notice that for some URL lengths, a record may still be less than 1000 bytes, but there won't be enough room for another pointer in the record. Just drop the pointers which overflow the 1000 byte limit and pad the record to the full size. Don't put half a pointer inside a record.

Index file

This file is an index on the URL file, in the form of a sorted array. Each node in the array stores a URL and a pointer into the graph file to the record for that node. The index is kept in a file; it will not fit into memory. It can be searched using binary search.

In lecture, you will learn about some more realistic file structures for a large index, and you will implement some of them for your project.

Your task

Write a program that takes a troll file and produces both a graph file and an index file. Your program must also be able to take a troll file along with an *existing* graph file and its associated index file, and *revise* the graph and index files by adding new information. As before, if revising a graph node would make it exceed 1000 bytes, just cut off the excess information.

[2] This is not realistic. To compute page rank properly, we would need to store all the links. Think about how this might be done.

Notice that your program can only add nodes to the the graph and index files. When you read a trolling record for a web page which is already in your graph, you may add or remove links from that graph node but you should not remove the URL itself even if it no longer has any links from it to other pages. Think about why we don't want you to do this.

Since web pages are constantly disappearing, we ultimately will need a facility for removing a page from the graph; however someone else at your company will deal with that.

Some advice

About file layouts

You need to design the file structure layouts for the URL file and index file. It may seem that everything has been decided, but the specifications above only describe the *logical* structure of the files. There are a number of design decisions left, including the following:

- For "pointers" into a file, will you use a byte offset or relative record number?
- Will there be any delimiters between the fields of a node, or at the end of a node (to separate it from the one that falls next in the file)?

Whatever choices you make, try to picture each node as a sequence of bytes, and to picture the nodes as falling one after another in the file. In fact, draw yourself a picture of this.

Representing a node in memory

How do you store the contents of a graph node in memory? You simply need to:

- Define one or more variables that are appropriate to hold the information in a node. You can use a character array with a fixed size of 1000, and a counter to show how much of it is currently in use.
- Write a function for writing such a node to a file. Write the *whole* array, not just the used part, because we want room for expansion. Think about how you will determine how much of the node is in use when you read it from the file. Remember that your 1000 bytes of space per record includes everything you write to the file for that graph node. The output function will have to produce output that conforms to your design for the file layout.
- Write a function for reading such a node from a file. This will have to expect data in the file to be in the format produced by the output function. As a result, it will be able to read any node that the output function wrote.

Now that you have some variables and some functions that are all related, they are an obvious candidate to glue together in a class. You might call it GraphFileNode. Now a GraphFileNode knows how to do things to itself, and any code that uses such nodes doesn't have to think about the grungy details.

Note that appendix F of your text contains code for handling record input and output. It is very general code, and as a result is quite complex. You are welcome to use it for this assignment. It takes some time to figure out how to use, but this investment may really pay off if you use the code again on your project, which will also be permitted.

About building the graph

As you go through the troll file (whether you are building a graph from scratch or adding to an existing graph), what has to happen when you read that web page x has a link to web page y? You should already have the graph file node for web page x in memory. Now you need to find out where the graph file node for web page y is, so you can store this in the node for web page x (unless it's already in there). How do you find out where the graph file node for web page y is? Easy: use the index.

But what if there is not yet any graph file node for web page y? You'll have to add one (and put it into the index) so that you can point to it from the graph file node for web page x.

If you design your objects well, the paragraphs above will convert easily into code that is short and elegant.

About the index

The code presented in tutorial and available on the web page may help you get started. Note that it stores the index in memory, which you may not do. You may also take advantage of the tutorial code for binary search in a file. Because you are going to do binary search on the index, it must contain fixed-width records.

Miscellaneous

At any one time, there should be only one node from the graph file in memory (or maybe a small constant number of them). You should absolutely not read the whole graph into memory – it will not fit. This means that there is no need for any field that is a "`GraphFileNode *`" in your GraphFileNode class, for example.

Similarly, you should only have one entry from the index in memory at a time, or perhaps a small constant number of them.

Although you will be jumping all over the files, which is very slow, you are not required to do any buffering in an attempt to speed things up.

About tackling the assignment

Take time to design the code structure before you start writing. This means much more than a rough idea of what the classes are. For each class, you should design its interface precisely, giving careful specifications of what each public function does, including preconditions.

Also take time to plan the *process* of developing your code. In what order will you do it? What sort of testing can you do along the way? Plan to write and test the code incrementally, and to keep a record of such testing.

What to hand in for deadline A

For deadline A, you should have some code running. Exactly what is up to you, but it should be something non-trivial that is along the way to completing the assignment. Don't hand in a report with part A. Just hand a printout of your code (including any driver programs), and some test runs that demonstrate what works. Explain your test results with brief, hand-written comments on your printout. Try to make your submission as brief as possible.

What to hand in for deadline B

For deadline B, hand in your assignment in the form of a report, as described in the 228 Course Guide. Your report must include all of your code and all relevant input and output test files. Remember to justify the decisions you made in designing your program. This is a very important part of your report.

Your report should be thorough, well-written, concise, and honest. It should be no more than about five pages long, not including code and tests. Make sure that you run your report through a spelling checker. You will also be required to hand in your assignment electronically. Please check the website for instructions.