

COMP 250 - Homework #5

Due on December 3rd 2004, in class

Question 1: Web search-engine or Sudoku (45 points)

(to be turned in on webCT before 13:35 on December 2nd)

For this question, you have the choice between two possible projects. You should do only *one* of the two.

Project #1: Web search-engine

For this final programming homework, we are going to write a mini-Google program that will (i) explore a mini-internet, (ii) build an index of the words contained in each web site of the mini-internet, (iii) analyze the structure of the internet graph to establish which web sites should be considered authorities, (iv) use this analysis to answer simple Google searches where a query word is entered by the user and you return the most relevant web sites available. Although we are only going to be dealing with a very small subset of the whole internet (we will only consider a subset of the web sites on the www.cs.mcgill.ca domain), the tools you will develop are essentially the ones used at Google to answer actual queries.

Needless to say that this programming is somewhat more substantial than those you had to do in the first four assignments. You will NOT be able to complete this project if you start too late! Still, your final solution will still be relatively simple, and I will guide you step by step.

Start by downloading and unzipping <http://www.mcb.mcgill.ca/~blanchem/250/hw5.zip> (on Linux, after downloading the file, type “unzip hw5.zip”).

The tools:

directedGraph class

I am providing you with a directedGraph class that represents directed graphs using the adjacency-list data structure seen in class. You will use this class to represent the mini-internet we will work with. The vertices of the graph will be labeled with Strings corresponding to the URLs of the web pages. A hash table is used to store a set of pairs (vertex, list-of-adjacent-vertices), where list-of-adjacent-vertices is a LinkedList of Strings. The class allows the following operations on the graph:

<code>void addVertex(String v)</code>	<code>// Adds a new vertex to the graph.</code>
<code>void addEdge(String v, String w)</code>	<code>// Adds an edge between two existing vertices</code>
<code>LinkedList getVertices()</code>	<code>// Returns a LinkedList of all vertices in the graph</code>
<code>LinkedList getNeighbors(String v)</code>	<code>// Returns a LinkedList containing the neighbors of vertex v</code>

```

LinkedList getEdgesInto(String v)    // Returns the list of vertices that have edges
                                     toward vertex v
int getOutDegree(String v)          // Returns the number of edges going out of v
boolean getVisited(String v)        // Returns true if the vertex v has been visited
void setVisited(String v, boolean b) // Sets the "labeled" field of vertex v to value b
double getPageRank(String v)        // Returns the pageRank of v
void setPageRank(String v, double pr) // Sets the pageRank of v to pr

```

A complete description of each method is available in the `directedGraph.java` file. You should not need to modify this class at all (but you can if you want).

htmlParsing class

This class has static methods to read the content of the web page at a given URL and to extract information from it. It has two static methods that you will need to use:

`getContent(String url)` returns a `LinkedList` of `Strings` corresponding to the set of words in the web page located at the given URL. You will need to use this method during your graph traversal to build your word index. `getLinks(String url)` returns a `LinkedList` of `Strings` containing all the hyperlinks going out of the given URL.

The class also has other methods that you will most likely not need. You should not need to modify this class at all.

searchEngine class

The `searchEngine` class is the one you will need to work on. It contains two important members: “`directedGraph internet`” is the internet graph we will be working on. The graph is initially empty, and it will be your job to add vertices and edges to represent an internet graph. “`HashMap wordIndex`” will be used to store the list of URLs containing a particular word. It will contain a set of pairs (`String word`, `LinkedList urls`).

HashMap class (part of the standard Java distribution)

To keep an index of which web sites contain which words, we will use the `java HashMap` class. A `HashMap` is an implementation of a dictionary. You will use words as keys in that dictionary. The data associated with a given word will be the list of web sites that contain that word. You will find a complete description of the `HashMap` class at

<http://java.sun.com/j2se/1.4.2/docs/api/>

Notice that the `get(String key)` method returns the information associated with a given key. The type of the object returned is `Object`, so you will need to cast it into the type it actually is (for example, `LinkedList =(LinkedList) wordIndex.get("hello")`). `HashMap` are extensively used in the `directedGraph` class, so you can have a look at that code to figure how things work.

LinkedList class (part of the standard java distribution)

You’ve already used this one. See <http://java.sun.com/j2se/1.4.2/docs/api/>.

Iterator class

You will often need to step through a `LinkedList` to do some processing on every piece of data in the list. A good way to do that is to use an object of the class `Iterator`, as follows:

```

LinkedList content = htmlParsing.getContent("http://www.cs.mcgill.ca/index.html");
Iterator i = content.iterator();
while ( i.hasNext() ) {
    String s = (String) i.next();
    // Do some processing on s
    System.out.println(s);
}

```

You will need to implement the following methods of the searchEngine class:

void traverseInternet(String url): This method will do a graph traversal of the internet, starting at the given url. You can use either a depth-first search or a breadth-first search, and you can make it either recursive or non-recursive. For each new url visited, you will need to (i) update the internet graph by adding the appropriate vertex and edges and (ii) update the wordIndex to add every word in the url to the index.

void computePageRanks(): This method will compute the pageRank of each vertex of the graph, using the technique seen in class and described below. This method will only be called after the traverseInternet method has been called.

String getBestURL(String query): This method will have to return the most relevant web site for a particular single-word query. The URL returned must be the one with the highest pageRank score containing the query. This method is only going to be called after the computePageRanks method has been called.

Calculating pageRanks

We calculate pageRank for each vertex as described in class. Let $C(v)$ be the out-degree of vertex v and let $w_1 \dots w_n$ be the vertices that have edges pointing toward v . Let $d = 0.5$. Then

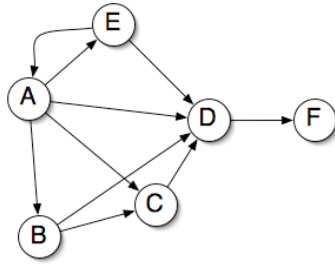
$$PR(v) = (1-d) + d * (PR(w_1)/C(w_1) + PR(w_2)/C(w_2) + \dots + PR(w_n)/C(w_n))$$

For a graph with n vertices, this gives rise to system of n linear equations that can easily be solved iteratively:

- 1) Initialize $PR(v) = 1$ for all vertices v
- 2) Repeat until convergence // 100 iterations should be enough
 - For each vertex v do
 - $PR(v) =$ formula above

Testing your program

To verify that your program is working well, I built a set of six small websites connected as follows:



The content of these sites is as follows :

- a.html: Algorithms are fun
- b.html: Algorithms are great
- c.html: Algorithms are great
- d.html: Authority about algorithms
- e.html: Algorithms are difficult
- f.html: This is a deadend about algorithms

If you start your internetTraversal from the website

<http://www.cs.mcgill.ca/a.html>, then your algorithm should learn the graph above.

After computing the pageRank of each site, you should obtain the following numbers:

<http://www.cs.mcgill.ca/a.html>=0.645...

<http://www.cs.mcgill.ca/b.html>=0.580...

<http://www.cs.mcgill.ca/c.html>=0.725...

<http://www.cs.mcgill.ca/d.html>=1.233...

<http://www.cs.mcgill.ca/e.html>=0.580...

<http://www.cs.mcgill.ca/f.html>=1.116...

Thus, on a query “algorithms”, your program should return site D. On query “are”, you should return site C.

When you run your program starting from <http://www.cs.mcgill.ca/index.html>, here is an example of queries:

Query: “colloquium” returns <http://www.cs.mcgill.ca/index.html> , p.r. = 8.95

Query: “computer” returns <http://www.cs.mcgill.ca/webadmin/index.html>, p.r.=20.2534

Query: “algorithms” should return <http://www.cs.mcgill.ca/news/index.html>, p.r.= 0.872

Query: “google” should return <http://www.cs.mcgill.ca/socsinfo/index.html> p.r. = 1.014

Query: “hockey” should return no site.

Avoiding denial-of-service (DoS) attacks

Suppose you have a bug in your program and during the internetTraversal phase, you have an infinite loop that keeps asking for the content of a given URL. This may be dangerous as it may eventually bring down the web server, in particular if many students had such an infinite loop running. This would be the equivalent of a denial-of-service attack like those that often happen against Microsoft and others: If millions of malicious

programs are requesting web pages from the same web server at a very high pace, the web server will be overloaded and may crash.

To prevent this from happening with the cs.mcgill.ca web server, the content of the URLs you will access will actually be already stored locally on your computer. This is all done for you in the htmlParsing class. It would be very easy to modify the getContent and getLinks methods so that they actually get stuff over the internet instead of from a file, BUT I AM ASKING YOU NOT TO DO THAT, because it may cause trouble to our web server.

How much work does this represent?

This programming project looks quite complicated but in the end your code will probably take about 150 lines of code. What will take you time is to get used to playing with LinkedList and HashMap. Don't wait until the last moment to ask questions!

Evaluation:

Correctness of internetTraversal:	25%
Correctness of pageRank calculation:	25%
Correctness of best web page reported:	20%
Style, simplicity, reuse, comments:	30%

Project #2: Sudoku solver (45 points + potential bonus)

A Sudoku is a type of puzzle that has become extremely popular these days. You will find the rules of the Sudoku puzzle at: <http://www.sudoku.com/>

Your goal is to write a program that will attempt to solve Sudoku puzzles. We are giving you a simple java class to represent a Sudoku puzzle, and to read or write a Sudoku from and to a file. You are asked to write the solve() method, which modifies the Sudoku object on which it is called and should fill all the empty squares in the Sudoku to produce a valid solution.

The code is at: <http://www.mcb.mcgill.ca/~blanchem/250/sudoku.html>

You will also find there a set of sudoku puzzles, sorted in increasing order of difficulty. To get full credits for this project, your program should be able to solve all “easy” puzzles in less than one minute of running time. To run your program from the command line, simply give as argument the file of the Sudoku you want to solve:
java Sudoku easy1.txt

To run your program within Eclipse, simply specify in the

There are many ways your program could attempt to solve Sudokus. It is totally up to you to decide which approach to use. I would suggest that you try to solve a few Sudokus manually before trying to write a program. One classical way to approach this kind of problem is to use a backtracking search as described in class for the 8-queens problem. What will determine how quickly your program will find a solution is the criteria you use to decide in what order the backtracking search should be performed. This project is totally open-ended. Use your imagination!

Evaluation

The program succeeds at solving “easy” cases correctly in less than 1 minute.	50%
Originality of the solution.	30%
Style and documentation	20%

Sudoku tournament

To identify a winner for the Sudoku tournament, each program will be given one minute to solve each Sudoku puzzle on our computer. The ranking of the programs will be established based on the number of Sudokus they succeeds at solving, from a set of 10 problems ranging from very easy to very difficult. The actual problems that will be used for the tournament will not be released before the tournament, but a set of equally difficult problems will be given for training. To keep the competitive spirit going, teams will have the possibility of using webCT to report which problems they are able to solve.

1st place winner: 20 bonus points
2nd place: 10 bonus points
3rd place: 5 bonus points

Question 2. (15 points) Dynamic Programming

- a) (10 points) You are leaving on a hiking expedition and you want to fill your backpack with useful objects. You have n different type of objects to choose from. Each object i has a integer weight $W(i)$ and an integer value $V(i)$. Your backpack can carry at most a total weight of T . Write a dynamic programming algorithm to find the optimal total value you can put in your backpack subject to that constraint. Assume that you have infinite supply of each type of object, so you can put as many copies of each object as you want. However, the items are not divisible; you have to take an integral number of each (either zero or more).

For example, for $n = 4$ and

$W(0) = 20, V(0) = 9$

$W(1) = 15, V(1) = 7$

$W(2) = 11, V(2) = 5$

$W(3) = 8, V(3) = 1$

Then if $T = 34$, the optimal solution is to take three item2 and zero of all others, for a total weight of $33 \leq T$ and a total value of 15. If $T = 35$, the optimal solution is to choose one item0 and one item1, for a total weight of $35 \leq T$ and a total value of 16. Your algorithm has to work for any choice of W and V .

Algorithm BackpackValue($W[0 \dots n-1], V[0 \dots n-1], T$)

Input: An array W of weights and an array V of values, both of them of size n . The total weight allowed T .

Output: Returns the optimal value of the backpack. You don't need to output what choice of items actually realizes that value.

/* Complete this pseudocode */

- b) (5 points) Write the pseudocode that actually prints what items should be chosen in what quantity to achieve the optimal solution.

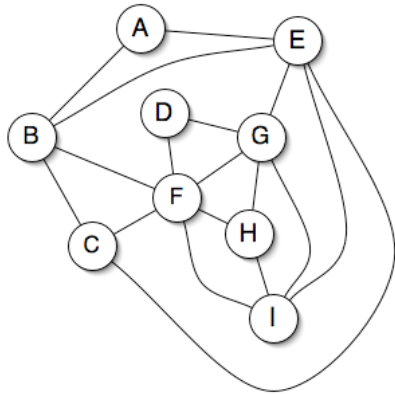
Question 3. (15 points) Sorting again!

a) (10 points) Consider the following sorting problem: You are given an array of n integers. The integers are all between 1 and $2n$ and they are all distinct. Write an algorithm that sorts this array and that runs in time $O(n)$ in the worst case. (Hint: this is really easy once you see it. I could have said that the numbers are all between 1 and $10n$, or between 1 and $9394923n$ and it would make no difference, except that the constant hidden in the big-Oh notation would get larger.)

b) (5 points) Why can't your algorithm be used efficiently to sort an arbitrary set of integers?

Question 4. Graphs and NP-completeness (25 points)

Consider the graph below:



- a) (5 points) Find the largest clique present in this graph.
- b) (5 points) Find the minimum number of colors needed to color the vertices of this graph. Argue (in one or two lines) why it is impossible to color it with fewer colors.
- c) (5 points) Given a graph G , an *independent set* is a subset X of the vertices of G such that no pair of vertices in X is connected by an edge in G . Find the largest independent set in the graph above
- d) (5 points) The k -INDEPENDENT-SET decision problem is defined as follows:
Given: an undirected graph G and an integer k
Question: Does G contain an independent set of size at least k ?

Prove that k -INDEPENDENT-SET is in NP. To do so, you must prove (by providing a simple algorithm) that the correctness of a potential solution provided to you in the form of a list of vertices can be *verified* in polynomial time.

- e) (5 points) Prove that k -INDEPENDENT-SET is NP-complete. To do so, take one known NP-complete problem (maybe SAT, k -CLIQUE, or 3-COLORABILITY) and show that this problem can be encoded as a graph that will contain an independent set of size at least k if and only if the answer to the original problem is YES. This should take about five lines. Congratulations, you have just written your first NP-completeness proof!