FRS 117: Google and Ye Shall Find???
Fall 2007
Assignment 4
Due Friday, October 19 at 5PM

Notice: There will not be an assignment due Friday, October 26.
Final paper topic description due Monday, November 5 at 5pm.

Blog entry: Read the blog entry posted just after yours for the blog assignment last week; if yours is the most recent entry, read the earliest blog entry for the blog assignment last week. Using the blog commenting mechanism, write a comment that adds some additional information or analysis to the entry of your classmate. ***Examples*** of what you might do are: try the query your classmate used and make another observation; add another thought to the analysis your classmate did; remark on similarities or differences between what you observed and what your classmate observed. Your comment should ***add value*** to the blog of your classmate, not critique it. It need be only a couple of sentences.

Note that the earliest blog entries for last week are not on the front page of the blog. The earliest entry is October 11th, 2007 by GoogleRocks. GoogleRocks will write a comment on the entry October 11th, 2007 by swoosh615; swoosh615 will write a comment on the entry October 12th, 2007 by Anthony Loring; etc.

Technical Exercises:

Problem 1: *This problem reviews the algorithm used for crawling that we discussed in class.*

Crawling the Web is an application of the abstract problem of *visiting the nodes of a graph*. To visit the nodes of a graph, one starts at one (or more) nodes, and uses the edges (links) out of that node to find more nodes to visit. When we "visit" a node, we find all the edges out of the node and do other processing that depends on the application. For Web crawling, when we process a node we index the contents of the Web page represented by the node. We repeatedly use edges found when visiting a node to find new nodes until no more new nodes can be found.

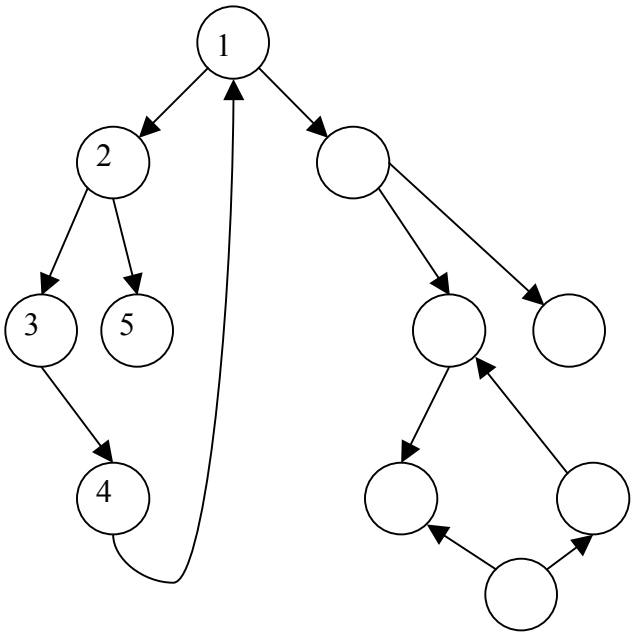
Consider a simple instance of visiting the nodes of a graph: We have only one starting node. We keep one list of edges that we have not followed yet. We take an edge from the front of the list, follow it to a node and process the node. When we follow an edge to a node we have already seen, we just ignore the node and go back to our list for a new edge.

The order in which we visit nodes depends on how we manage the list of edges. When we visit a node, we add all the edges out of the node to the list (in the order we find them). Where on the list we add the edges makes a big difference:

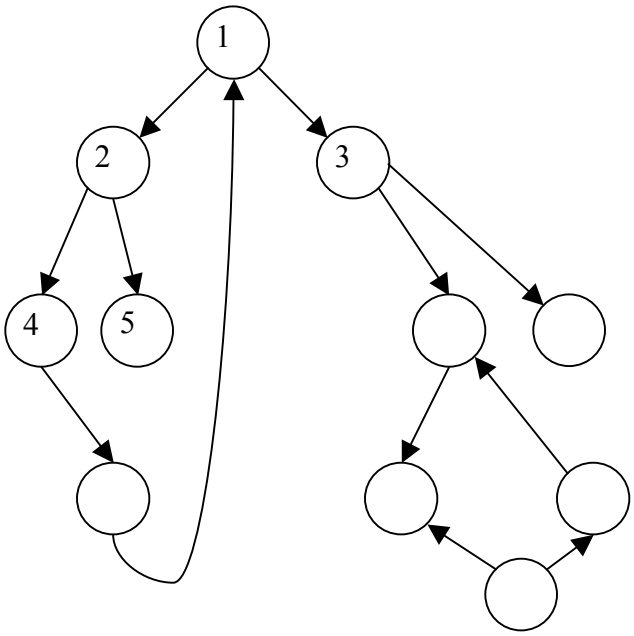
A. If we add the edges to the front of the list, we visit neighbors of a node right away. This is "*last in, first out*" order for the list.
B. If we add the edges to the back of the list, we postpone visiting neighbors of a node. This is "*first in, first out*" order for the list.

The figure below shows two copies of the same graph with numbers indicating the order in which nodes are visited: in one copy *last in, first out* is used and in the other, *last in, first out* is used. Only the first 5 nodes visited are numbered.

Using *last in, first out* order (A):



Using *first in, first out* order (B):



Part i: Complete the numbering of this graph using each of the orders. Can you see how "last in, first out" yields what is called "depth first" visting and "first in, first out" yields what is called "breadth first" visiting?

Part ii: Part of this graph is invisible regardless of which ordering is used. Which part of the graph is invisible and why?

Problem 2: *This problem asks you to think about the details of the efficiency gained by sorting inverted index entries.*

We have seen that an inverted index is built so that it is sorted on the words in the index, each word entry (occurrence list) is sorted by document IDs (or maybe document PageRank with ties broken by document ID), and each list of occurrences of a word within a document is sorted by position. All of this makes processing queries more efficient. Consider the algorithm for processing the two-term query "cat dog":

Algorithm using sorted occurrence lists:

1. retrieve occurrence list for "cat"
2. retrieve occurrence list for "dog"
3. find first document in "cat" list – call it Dcat1
4. look down "dog" list until find Dcat1 or a document which comes later than Dcat1 in sorted order; remember where now reached on "dog" list
5. if found Dcat1 for "dog" too, merge two position lists and save for ranking
6. repeat for each remaining document. on "cat" list in order:
    i. starting where left off on dog list, look down "dog" list until find current "cat" list doc or a document which comes later in sorted order; remember where now reached on "dog" list
    ii. if found current "cat" list document for "dog" too, merge two position lists and save for ranking

The list for "cat" and the list for "dog" are each read exactly once using this algorithm.

Now suppose that the documents in occurrence lists are ***in no particular sorted order.*** That is, the occurrences of a word in a single document are listed together, but documents can appear in different orders in the occurrence lists for different words.

Part i: Write an algorithm in the style above to process query "cat dog" when the documents are not in any sorted order within the occurrence lists.

Part ii: For your algorithm, how many times is the list for "cat" read? How many times is the list for "dog" read?