CS:3330 Homework 3, Fall 2015

Due in class on Tue, Oct 20

---

1. For any vertex v in a graph, the *neighborhood degree* of v is the sum of the degrees of v and all its neighbors. Your problem is to design an algorithm that takes as input a graph $G = (V, E)$ represented in some way (i.e., using an adjacency matrix or an adjacency list) and returns an array `nbdDegree` of neighborhood degrees of all vertices in the graph.

   (a) First, assume that the graph is represented using an adjacency list. Describe an algorithm for the problem that runs in $O(m + n)$ time, where m is the number of edges and n is the number of vertices of the input graph.

   (b) Argue that the running time of your algorithm is indeed $O(m + n)$.

   (c) Now make any modifications necessary to the algorithm described in (a) so that the algorithms runs correctly assuming that the input graph is represented using an adjacency matrix. (My expectation is that you'll have to make very few modifications.) Restate your algorithm and analyze its running time. Express your answer as a function of m and n using the O-notation.

2. A set $D \subseteq V$ of vertices of a graph $G = (V, E)$ is a *dominating set* if for every vertex $v \in V$, either v is in D or has a neighbor in D. Think of D as a set of "dominators," each of which "dominates" itself and all of its neighbors; thus every vertex is "dominated" by a dominating set.

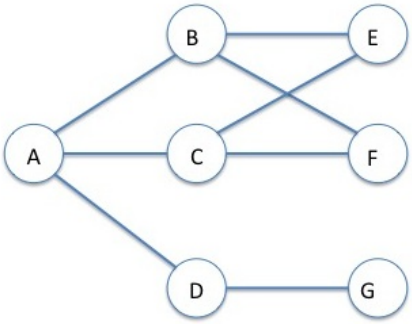
   A well known optimization problem is the *minimum dominating set* problem in which we are given a graph and asked to find a dominating set with fewest vertices. The minimum dominating set problem is often used to abstract problems that arise in wireless networks – routing and saving on battery power.

   A simple, natural *greedy* algorithm for the problem would be to repeatedly pick a vertex (as a dominator) that can dominate the most, as yet, undominated vertices. (Note: a vertex dominates itself.) Here is pseudocode that implements this idea.

   1. Initialize all vertices in the graph to have the color `white`
   2. **while** the graph contains `white` vertices **do**
   3. pick a vertex v that dominates the highest number of `white` vertices
   4. color v `black`
   5. color `gray` any neighbor of v that is not black

   Here is a brief explanation of this algorithm. At any point in the algorithm, the vertices have one of three colors: white, gray, and black. The black vertices are in the solution (the dominating set). The gray vertices are not in the solution, but they have been "dominated" by some vertex in the solution. Finally, the white vertices are the ones that have not yet been dominated and need to be. So at every step in the algorithm, we make a natural greedy choice of picking a vertex (to add to the solution) that dominates the maximum number of white vertices.

   (a) Show the execution of the above-described greedy algorithm on the following input graph. Make sure to clearly show the color of each vertex after each iteration of the **while**-loop. When there are ties (i.e., two or more vertices dominating the most number of white vertices) then your algorithm should pick a vertex whose name appears earliest in alphabetical order. Finally, write down the dominating set chosen by your algorithm.



(b) As you might expect, the greedy algorithm described above does not always return a minimum dominating set. Here I describe a family of graphs G_n for $n = 1, 2, \ldots$. Start with subsets of vertices $L_1, L_2, \ldots, L_n$, where $|L_i| = 2^i$ for each $i = 1, 2, \ldots, n$. Now add vertices v_i, $i = 1, 2, \ldots, n$ to the graph and connect each v_i to all the vertices in L_i. Then connect all the n v_i's to each other. Next add two vertices v_R and v_B to the graph. Connect v_R to v_B and all the v_i's; similarly, connect v_B to all the v_i's. Finally, for each $i = 1, 2, \ldots, n$, pick half the vertices in L_i and connect to v_R and pick the other half and connect to v_B.

Carefully draw G_3. Describe the execution of the above-described greedy algorithm on G_3. What is the size of the dominating set returned by your algorithm? And what is the size of a minimum dominating set on G_3?

(c) In general, what is the size of a minimum dominating set in G_n? What is the size of the dominating set returned by the greedy algorithm on G_n?

(d) Suppose that someone claimed that the greedy algorithm is a 10-approximation algorithm for the minimum dominating set problem. What graph would serve as a counterexample to this claim?

3. To implement the above greedy algorithm efficiently, we need a data structure that permits us to efficiently (and repeatedly) pull out a vertex that dominates the maximum number of white vertices. This needs to be done as the number of white vertices falls in each iteration. For now imagine that we have a data structure that maintains a collection of $(key, value)$ pairs (with no two elements having the same key) supporting the following operations:

- `getMax()`: This deletes and returns a $(key, value)$ pair with the maximum $value$ among all elements in the collection.
- `insert(`k`, `v`)`: This inserts an element (k, v) into the collection, assuming that there are no elements already in the collection with key equal to k.
- `decreaseValue(`k`, `d`)`: The modifies the $(key, value)$ pair with key equal to k, replacing $value$ by $value - d$.

Now here is the problem.

(a) Restate the greedy algorithm from Problem 2 using pseudocode, but now in a way that allows us to do a run-time analysis of the pseudocode. Specifically, assume that the graph is represented as an adjacency list. Also, maintain a collection of non-black vertices and the number of white vertices they dominate as a collection of $(key, value)$ pairs using the data structure mentioned above. Thus, your pseudocode will be making repeated calls to the operations `getMax()`, `insert(`k`, `v`)`, and `decreaseValue(`k`, `d`)` described above.

(b) Suppose that there is a way implement the above-mentioned data structure so that `getMax()` runs in $O(1)$ rounds and `insert(`k`,` v`)` and `decreaseValue(`k`,` d`)` run in $O(\log s)$ time each, where s is the number of elements in the collection. Given this, what is the running time of your implementation in (a), as a function of m (number of edges) and n (number of vertices) of the input graph. As usual, I am looking for an asymptotic running time.

(c) Think back to your class on data structures. What is the name of the data structure that can be implemented in a manner satisfying what we've supposed in (b).

4. A *number maze* is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board. Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution.
(This problem is from Jeff Erickson's notes.)

5. I have posted on the course website a text file containing a graph described as a list of edges. As mentioned in the comments at the top of this file, the graph has 23133 vertices and 186936 edges. Your task is to implement an efficient program in your favorite language that answers the following questions (see (a) and (b) below) about this graph. What we have learned about graph traversals in class should be more than adequate for you to write this program. The interaction between your program and the user should be extremely simple – the program just reads the file (no need to prompt the user for anything) and produces the two answers as output. Note that the answer to the first question is two numbers and the answer to the second question is three numbers.

   Submit your program via an ICON dropbox. Make sure your program follows all the standard guidelines for good coding, including meaningful documentation and variable names, modularity, etc. The actual answers to the questions below and the paragraph description of your algorithms (part (c)) can be submitted with the rest of the homework.

   (a) What is the size (i.e., number of vertices and edges) of the largest connected component in this graph?

   (b) What is the *diameter* of this graph and what are two vertices that are this distance apart from each other? The *diameter* of a graph is the maximum distance between any pair of vertices in the graph.

   (c) Write a paragraph briefly describing what algorithms you implemented to answer questions (a) and (b). Also, write down the asymptotic running time of your implementations that answer each of these questions.
