CS:3330 Homework 3, Spring 2017

Due at the start of class on Thu, Feb 9

1. About a year ago, the University of Chicago computer scientist, László Babai posted a paper that was considered a "breakthrough" in the algorithms research community. His paper presented an algorithm for the *Graph Isomorphism (GI)* problem – this was a problem that no one had been able to solve in polynomial time. You don't have to know anything about the GI problem to answer the following questions.

   (a) Babai's paper claimed to have solved GI in time $f_1(n) = 2^{(\log n)^c}$, where n is the input size and $c > 1$ is a constant. Compare the asymptotic growth rate of the function f_1 to the classes of polynomial and exponential functions. Provide explanation for any claims you make. For example, if you claim that f_1 grows faster than any polynomial function, you should be able to argue why.

   (b) Last month Babai posted an update. He had a slightly modified algorithm for the GI problem and roughly speaking, this ran in $f_2(n) = 2^{2^{O(\sqrt{\log n})}}$ time. Compare the asymptotic growth rate of the function f_2 to f_1 and to the classes of polynomial and exponential functions. Provide explanation for any claims you make.

   (c) Based on what you learned from thinking about (a) and (b), do you think we now know how to solve GI in polynomial time? Alternately, do you think it would take an exponential-time algorithm to solve GI?

2. We want to multiply two $n \times n$ matrices A and B. As you know, $A \cdot B$ is an $n \times n$ matrix C, where $c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$. Here $c_{i,j}$ refers to the entry in matrix C that is in row i and column j. Similarly, $a_{i,k}$ refers to the row i, column k entry in matrix A and $b_{k,j}$ refers to the row k, column j entry in matrix B.

   (a) Describe (in pseudocode) an algorithm for multiplying two $n \times n$ matrices.

   (b) State the running time of your algorithm as a function of the input size, using Θ notation. Note that input size need not be n.

3. 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.

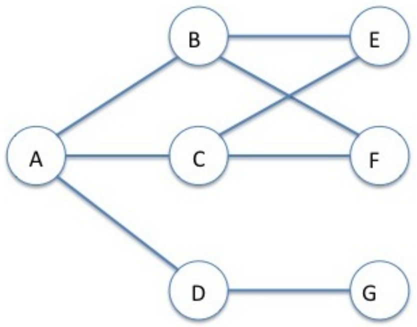
   A well known optimization problem is the *minimum dominating set* problem in which we are given a graph and asked to find a dominating set with fewest vertices. The minimum dominating set problem is often used to abstract problems that arise in wireless networks – routing and saving on battery power.

   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?

4. 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).
