Pseudocode and Analysis of the Greedy Algorithm for the Minimum Dominating Set problem

CS:3330, Spring 2017, Sriram Pemmaraju

(a) The greedy algorithm in Problem 3 with input adjacency list can be implemented in the following way:

Algorithm 1 Dominate(L)

```
1: Set nonblack be an empty object to host non-black vertices
2: Let ds be an empty set for hosting the dominating set
3: Let color be a length-n array, all of whose slots are initialized to white
4: for each vertex i in the graph do
5:     nonblack.insert(i, L[i].length+1)
6: end for
7: (v, whiteDeg) ← nonblack.getmax()
8: while whiteDeg > 0 do
9:     Save v to ds
10:    if color[v] == white then
11:        for each neighbor j of vertex v do
12:            nonblack.decreaseValue(j, 1)
13:        end for
14:    end if
15:    for each neighbor j of vertex v do
16:        if color[j] == white then
17:            for each neighbor k of vertex j do
18:                nonblack.decreaseValue(k, 1)
19:            end for
20:            color[j] ← gray
21:        end if
22:    end for
23:    color[v] ← black
24:    (v, whiteDeg) ← nonblack.getmax()
25: end while
26: return ds
```

(b) Given the running time of the 3 methods, `getMax`, `insert`, and `decreaseValue`, we can analyze the algorithm's running time complexity as follows. The for-loop (Lines 4-6) executes `insert`(k, v) n times, taking $O(\log n)$ time for each insertion. Thus, this for-loop will run in $O(n \log n)$ time. The while-loop is executed n times because with each execution, one vertex is deleted from **nonblack**. Each execution of `getmax`, takes $O(\log n)$ time and therefore extracting vertices with largest white neighborhood from **nonblack** take $O(n \log n)$ time. After a vertex v is extracted from **nonblack** and added to **ds**, we have to update white neighborhood sizes associated with vertices in `nonblack`. Now note that for each vertex that changes from white to gray or black, we update its neighbors' values in **nonblack**. A vertex changes from white to some other color only once and therefore for each edge we perform this update at most twice. Updates of these values (via `decreaseValue`) take $O(\log n)$ time. Thus the total time to update sizes of white neighborhood sizes is $O(m \log n)$. Thus the total running time of this algorithm is $O((m + n) \log n)$.

(c) The data structure that can fulfill the runtime specifications is a *max-heap* implementation of a priority queue.