Problem 1

(a) Consider the following algorithm with input being an adjacency list representation $L$ of the given graph:

Algorithm 1 NeighborhoodDeg($L$)

1: Initialize $\text{degree}$ to be an array of size $n$
2: Initialize $\text{nbdegree}$ to be an array of size $n$
3: for each vertex $i$ in the graph do
4:   $\text{degree}[i] \leftarrow 0$
5:   for each neighbor $j$ of vertex $i$ do
6:     $\text{degree}[i] \leftarrow \text{degree}[i] + 1$
7:   end for
8: end for
9: for each vertex $i$ in the graph do
10:   $\text{nbdegree}[i] \leftarrow \text{degree}[i]$
11:   for each neighbor $j$ of vertex $i$ do
12:     $\text{nbdegree}[i] \leftarrow \text{nbdegree}[i] + \text{degree}[j]$
13:   end for
14: end for
15: return $\text{nbdegree}$

(b) For the algorithm above, the overall running time can be calculated as follows. The first loop (Lines 3-8) computes the degree of each vertex in the graph and stores it in the $\text{degree}$ array. Each vertex $i$ is considered in the outer for-loop (Line 3) and then each neighbor $j$ of vertex $i$ is considered. Thus, for a vertex $i$, the inner loop runs in time $\Theta(1) + \Theta(\text{degree}(i))$. Summing up over all vertices $i$, we see that the total running time of the first loop is $\Theta(m + n)$. The second loop (Lines 9-14) sums up the degrees of all the neighbors of each vertex (along with the degree of that vertex) and stores this in an array $\text{nbdegree}$. The structure of this for-loop is essentially the same as the structure of the first for-loop and therefore the running time of this loop is also $\Theta(m + n)$. Thus the total running time of this algorithm is $\Theta(m + n)$.

(c) Consider a new implementation of the algorithm with input being an adjacency matrix $A$ instead of an adjacency list $L$.

The major differences between the two implementations is that in an adjacency matrix representation, examining all neighbors of any vertex $i$ takes $\Theta(n)$ time independent of the number of neighbors that vertex $i$ has. This means that the first loop (Line 3-7) executes in $\Theta(n^2)$ time and similarly, the second loop (Lines 8-15) executes in $\Theta(n^2)$ time. Thus the total running time of this new implementation is $\Theta(n^2)$.

Problem 2

(a) Initially, all vertices are white. After each iteration of the while loop, the results are as below:

1) white: $E, F, G$; grey: $B, C, D$; black: $A$
2) white: $G$; grey: $C, D, E, F$; black: $A, B$
3) white: None; grey: $C, D, E, F, G$; black: $A, B, D$

The final dominating set is $A, B, D$. 

Problem 2 continued on next page...
Algorithm 2 NewNeighborhoodDeg(A)
1: Initialize degree to be an array of size $n$
2: Initialize nbdegree to be an array of size $n$
3: for each vertex $i$ in the graph do
4:     for each vertex $j$ in the graph do
5:         degree[$i$] ← degree[$i$] + $A_{ij}$
6:     end for
7: end for
8: for for each vertex $i$ in L do
9:     nbdegree[$i$] ← degree[$i$]
10:    for each vertex $j$ in the graph do
11:        if $A_{ij} == 1$ then
12:            nbdegree[$i$] ← nbdegree[$i$] + degree[$j$]
13:        end if
14:    end for
15: end for
16: return nbdegree

![Figure 1: A bad example for greedy algorithm for Minimum Dominating Set.](image)

(b) The execution of the greedy algorithm will repeatedly pick the vertex with the maximum number of white neighbors. In the beginning, $v_3$ has 8 white neighbors from $L_3$ plus $v_1$, $v_2$, $v_R$, $v_B$, which are white as well. Thus $v_3$ has a white neighborhood size of 13 (including itself). The remaining vertices have the following white neighborhood sizes: $v_2$: 9, $v_1$: 7, $v_R$: 12, $v_B$: 12, and every vertex $x \in L_1 \cup L_2 \cup L_3$ has white neighborhood size equal to 3. Thus $v_3$ will be picked first and colored black. Once $v_3$ is colored black, the white neighborhood sizes become: $v_2$: 4, $v_1$: 2, $v_R$: 3, $v_B$: 3 and every vertex $x \in L_1 \cup L_2$ has white neighborhood size equal to 1. Thus $v_2$ will be picked next. Now the white neighborhood sizes are $v_1$: 2, $v_R$: 1, $v_B$: 1 and every vertex $x \in L_1$ has white neighborhood size equal to 1. So $v_1$ is picked in the last iteration. In this way, the dominating set created by the algorithm is the set \{v_1, v_2, v_3\}, but the minimum dominating set is easily seen as \{v_R, v_B\}.

(c) The minimum dominating set in $G_n$ has size 2. The greedy algorithm returns a dominating set \{v_1, v_2, \ldots, v_n\} of size $n$ in $G_n$.

(d) The graph $G_{21}$ serves as a counterexample to the claim that the greedy algorithm is a 10-approximation. This is because the greedy algorithm produces a solution of size 21 which is strictly more than 10 times the size of a minimum dominating set.
Problem 3

(a) The greedy algorithm in Problem 2 with input adjacency list can be implemented in the following way:

Algorithm 3 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.
Problem 4

This problem is a graph modeling problem. We can convert the number maze $M$ into graph and use BFS on this graph to find a solution. First, we make each number in the number maze a vertex. Thus there are a total of $n^2$ vertices corresponding to an $n \times n$ maze. Then for each vertex $M_{i,j}$, we read its value $k$ in the number maze and connect $M_{i,j}$ to $M_{i+k,j}$, $M_{i-k,j}$, $M_{i,j+k}$, $M_{i,j-k}$ (provided these vertices exist). Once the graph is completely constructed, we can use BFS to find a shortest path from the vertex corresponding to the top-left square to the vertex corresponding to the bottom-right square.