Thomas Haddy
Com S 311
Section 19
TA: Trent Muhr
4/26/18

Homework 5

1. In addition to graph G, let H be a max heap sorted by a v ∈ V's outdegree. So, the root of the max heap has the largest degree, and when it gets removed heapify will take O(log(n)). So, this algorithm assumes that the vertex with the largest degree is most likely to be a dominating set, so it checks that vertex first.

   Dominating Set Algorithm:

   ```
   Input G = (V, E)
   Let H be a max heap with a comparator of v ∈ V's outdegree
   LinkedList D is empty
   While H is not empty do
       Let v be H's root node
       Remove v and perform maxHeapify on H
       Add v to D
       Remove v and all vertices in N(v) from G (and thus from V)
   Output D
   ```

   So, by having the max heap, searching for v with the largest degree takes O(log(n)) time. This makes the total runtime O((m^2)log(n)) because it needs to remove the current vertex and all of its edges and their vertices.

2. Independent Subset Algorithm:

   ```
   Input: G = (V, E), S = { 1, 2, …, n }
       Let visited be a boolean array with size G's vertices and set to false
       For each ( vertex s in S )
           Mark s as visited
           For each ( edge e in s )
               Mark e as visited
               If (e or s is already visited)
                   Output false

       Output true
   ```

   Runtime: In the worst case, this algorithm will go through all of G's vertices n times, where n is the size of S. It’s like BFS because it’s searching for a vertex x and vertex y whose edges are not connected at all. So it takes O(V + E).

Maximal Independent Algorithm:

```
                Input: G = (V, E)
O(1)            let S be an empty LinkedList
O(1)            let cantAddToS be an empty balanced AVL tree by V’s number of edges
O(Vlog(V))      let Q be a queue that sorts(V) by V’s smallest 2nd-neighbor-outdegree in queue
O(V)            For each ( vertex v in V )
O(log(V)                if (cantAddToS does not contain v)
O(1)                            add v to S
O(log(V)                        add v to cantAddToS
O(E)                            for each ( edge e in v )
O(1)                                    add e to cantAddToS
O(1)                                    if (cantAddToS size equals V size)
O(1)                                            break out

O(1)            Output S
```

Runtime: This algorithm is greedy in that assumes that a maximal independent set is more likely to be a vertex’s smallest 2^{nd}-neighbor-outdegree. By using an AVL tree, searching for a vertex becomes logarithmic time instead of linear. In the worst case, the algorithm still has to go through each vertex V and edge E in G.

$1 + 1 + V\log(V) + V*[\log(V) + 1 + \log(V) + E(1 + 1 + 1)] + 1$

$= V(\log(V) + \log(V) + E) \in O(EV)$

3. Max-cost Cut Algorithm:

```
Input: Matrix M[n][m],
Let maxCuts be an integer matrix of size M
For ( go through M’s j-th rows )
        For ( go through M’s i-th cols )
                If (the col is 0)
                        maxCuts[i][j] equals M[i][j]
                else
                        maxCuts[i][j] equals max{ maxCuts[col-1][row-1],
                        maxCuts[col-1][row], maxCuts[col-1][row+1] }

output max value from maxCuts last col
```

Recurrence: The recurrence in this algorithm comes from looking for the Max{ maxCuts[col-1][row-1], maxCuts[col-1][row], maxCuts[col-1][row+1] }

Runtime: Filling the maxCuts array will take n * m operations to do. Also, outputting the max value takes at most n time. So the runtime will be O(nm) because the algorithm takes longer filling maxCuts than selecting the max once maxCuts is filled.

4. Equal Subsets Algorithm:

    Input: Set S = { $x_1, x_2, \ldots, x_n$ }
    Let N be the sum of S's elements
    Let n be S's size
    Construct a 2-D array answer with size[N/2][n]
    For (j from 0 to answer's rows)
        For (i from 0 to answer's cols)
            If (i equals 0)
                Answer[i][j] equals 1
            Else if (j equals 0 and i does not equal 0)
            Answer[i][j] equals 0
            Else
                answer[i][j] = answer[i-1][j] OR answer[i – S[j]][j-1]
    output bottom right element in answer

Runtime: So, this algorithm will calculate the 2-D array of size[N/2][n] so the runtime and output the bottom right element. So filling this array takes O(Nn) time.