CMPSC 465 – Spring 2021 — Solutions to Homework 4

Josiah Kim, SID 948500821, `juk483`

February 18, 2021

1. Getting started

(a) I did not work in a group.

(b) I did not consult with any of my group members.

(c) I did not consult any non-class materials.

2. Divide-and-Conquer

(a)
```
function find_majority(array A):
        if length of array is 1:
                return A[1]
        midpoint = (length of A)/2
        A_left = A[:midpoint]
        A_right = A[midpoint:]
        left_majority = majority(A_left)
        right_majority = majority(A_right)
        if no majority in A_left and A_right:
                return none
        if left_majority is the same as right_majority:
                return left_majority
        else:
                return none
```

(b) If both halves have the same majority element, x, then we can say that x is the majority element of the entire array. If both arrays have majority elements, x and y, then the entire array does not have a majority element. If only one (either) half of the array has a majority element, then the entire array does not have a majority element. If none of the halves have majority elements, then the entire array does not have a a majority element.

(c) Since the function is recursively called twice: $T(n) = 2$

When the function is called recursively, only half of the original items are included each time: $T(n) = 2 * T(\frac{n}{2})$

The basic comparisons are done in lenear time: $T(n) = 2 * T(\frac{n}{2}) + O(n)$

This is the form for $O(nlogn)$ time complexity.

3.Reverse Graph

(a) Before any traversal happens, the algorithm will initialize an empty array that will contain the adjacency list (linked lists) for G^R.

The algorithm would select the first linked list to traverse of which the first element will be a vertex, u. u will then be pushed into the beginning (zero index) of the empty array with a pointer pointing to $NULL$.

Then, the algorithm will traverse the same linked list to the next vertex, v. v will then be pushed into the beginning of the new array with an incoming pointer from u and an outgoing pointer to $NULL$. The algorithm will traverse the linked list, push the vertex into the new array, and change pointers until the end of the linked list or a $NULL$ has been reached.

Once a linked list has fully been traversed, the algorithm will move onto the next linked list and the process will begin again.

This algorithm is $O(|V| + |E|)$ because every vertex and edge is visited once. Therefore, the time complexity is solely dependent on the number of vertices and edges in the graph.

(b) Before any traversal happens, the algorithm will initialize an empty array that will contain the adjacency list (linked lists) for G^R.

The first linked list will be traversed starting at the first vertex, head node, u. The algorithm will then search each linked list to see if u has an pointers pointing to it. If not, then u will be appended to the new array with a pointer pointing to $NULL$. This means that in G^R the linked list with head node, u, is an empty linked list.

However, if the algorithm finds a pointer pointing to u, then it will append u into the new array along with vertex, v, that is pointing to u with a pointer pointing from u to v. The algorithm will keep searching until it gets to the end of the last linked list.

This algorithm is $O(|V|^2)$ because for each vertex, every vertex is visited once. For each vertex, the alogrithm goes through every vertex to see if it will be an empty linked list or if it has adjacent verticies in the reverse direction in the new array.

4.Graph Basics

(a) $A \to B \to E$

$B \to D \to G$

$C \to H \to I$

$D \to E$

$E \to D$

F

$G \to D \to F$

$H \to I$

I

(b) In the extreme case when we're looking for the maximum amount of edges that an undirected graph can have, we can assume that each vertex has an edge to every single vertex in the graph even to itself. This means that each vertex has $|V|$ edges. This means $|V|^2$.

However, this would include parallel edges. To prevent counting parallel edges, each vertex should be counted as having one less edge than the vertex before it. We know that the first vertex will always have $|V|$ edges since there are no parallel edges that exist. Therefore, the number of edges would be $|E| = |V| + (|V| - 1) + (|V| - 2) + ... + (|V| - |V|) = \sum_{i=0}^{|V|} |V| - i = \frac{|V|(|V|-1)}{2}$.

(c) Let's take this adjacency list for an undirected graph:

$1 \to 2 \to 4$

$2 \to 1 \to 3 \to 4$

$3 \to 2$

$4 \to 1 \to 2$

$5 \to 6$

$6 \to 5$

For the case above, $|V| = 6$ and $|E| = 5$.

$\sum_{i=1}^{|V|} d_i = \sum_{i=1}^{6} d_i = d_1 + d_2 + d_3 + d_4 + d_5 + d_6 = 2 + 3 + 1 + 2 + 1 + 1 = 10$

By counting each incident edge, we are effectively counting each edge twice since every edge can only be connected by two vertices. In other words, the adjacency list is symmetric. Therefore, $\sum_{i=1}^{|V|} d_i = 2 * |E|$.

By definition of even numbers, $\sum_{i=1}^{|V|} d_i$ will always be even.

(d) The first algorithm's runtime is dependent not only on the number of vertices but also the number of edges. However, we know that the maximum number of edges that an undirected graph can have is greater than the number of vertices but less than the number of vertices squared (4b).

Therefore, by replacing number of edges with the number of vertices squared, we can assume that the first algorithm's runtime is less than the number of vertices cubed. The second algorithm's runtime is solely dependent on the number of vertices. We can see that the runtime shares a common factor which is the number of vertices, so by factoring it out we get $|V|log|V|$.

Now, we can compare $|V|log|V|$ to $|V|^2$ since we replaced $|E|$ with it earlier. We know that a polynomial with power greater than 1 grows faster than a logarithm multiplied by a polynomial with power 1. Therefore, the second algorithm will run faster.

(e) Both algorithms are dependent on the number of edges. However, the first algorithm is also dependent on the number of vertices while the second algorithm is ONLY dependent on the number of edges.

We know that the maximum number of edges that an undirected graph can have is greater than the number of vertices but less than the number of vertices squared. By replacing the the number of edges in the second algorithm to the number of vertices squared, we get $|E|log|V|^2$.

Now, we can compare this to the first algorithm $|E|log|V|$. Since they share a common factor, the number of edges ($|E|$), we can ignore it. Therefore, by comparing $log|V|$ from the first algorithm to $log|V|^2$ from the second algorithm, we know that a logarithm of a polynomial with power greater than 1 grows faster than a logarithm of a polynomial with a power 1. Therefore, the first algorithm will be faster.