CSCE 629: Analysis of Algorithms
Home Work - 3

Shabarish Kumar Rajendra Prasad
UIN - 228000166

October 2019

1 Question 1

Consider that there m sequences on which the MakeSet-Union-Find operations are to be performed and there are n nodes in the considered set S. Since there are no Find operations appearing before the corresponding MakeSet-Union operation, we shall consider that the Makeset and Union operations take constant time and is bounded by $O(m)$. Now as we know, a node is a *boundary node* if the node is child of the root of T and a node is an *internal node* if the node is not a child of the root of T. Now, we say that, since there are at most m find operations, the number of boundary nodes is also bounded by m. We can also say that, for every node in set S, that is labelled a, there is at most one internal node labelled a. For example, if there an internal node labelled a in the Find-path, then the node will become a child of a root r after the find operation and thus becoming a boundary node. Since there are no Union operations after the Find operation, the root r always remains the root and thus the node labelled a, always remain the child of the root and never becomes an internal node again. Thus, since there are n nodes in total in the set S, the number of internal nodes s bounded by n.

Thus the sum of the lengths of all the Find-paths, which is equal to the sum of total number of boundary nodes and the internal nodes is bounded by $m + n$. Since in this case, all the Find operations take constant time to execute, the total time complexity to compute all the Find-paths is in the order $O(m+n)$. Thus the total time complexity of the program can be expressed as $O(m+n)$.
‘

2 Question 2

The algorithm is shown below:

Find_Distance ($G = \{V, E\}, s, w$)

```
for each v in V:
    D[v] = 0

for each {i=1 to i=|V|-1}:
    for each (edge(u,v) in E):
        if D[v] > D[u] + w(u,v):
            D[v] = D[u] + w(u,v)

for each (edge(u,v) in E):
    if D[v] > D[u] + w(u,v):
        return ("Negative cycle found")

return D
```

The output returned by the algorithm D[v] = $\delta(v)$

Proof of correctness

We know that D[v] $< \delta(v)$ is not possible. This can be proved by induction on the number of times the if branch is executed. Before the if branch is executed at all, we have D[v] $= 0 \geq \delta(v)$ (because of the first for loop), for every vertex v. If this is true, then after every execution of the if branch, we have:

$$D[v] = D[u] + w(u,v) \geq \delta(u) + w(u,v) \geq \delta(v)$$

Thus D[v] $\geq \delta(v)$, remains true even after all the executions of this statement for a node v by induction.

Now we just have to prove that D[v] is not greater than $\delta(v)$. In the Graph G, let $dist[u,v]$ denote the shortest distance from u to v in G. Now let u be a node such that $dist[u,v] = \delta(v)$ (The existence of such a node in G is ensured, since G has no negative cycles). Let $P =< u = u_0, u_1, ..., u_k = v >$ be the shortest path with weight $\delta(v)$ from u to v in G. We can prove that after k executions of the outer for loop, D[v] $= \delta(v)$ of path P, by induction. When $k = 0$, then $u = v$ and the weight $\delta(v)$ of the path P is also equal to 0. Similarly in the algorithm, before the excution of the outer for loop, when $k = 0$, D[v] $= 0$ (Because of the first for loop which sets all values of D to 0) and thus induction holds for $k = 0$.

Now when $k > 0$ and consider that the path P has at least two nodes u and v. Consider u_{k-1} is also in the path P. The existence of the path $P =< u = u_0, ..., u_{k-1} >$ implies that $\delta(u_{k-1}) \leq \delta(v) + w(u_{k-1}, v)$. If $\delta(u_{k-1}) > \delta(v) + w(u_{k-1}, v)$, then the path would give a weight such that it is strictly smaller than $\delta(v)$, which is a contradiction to the definition of $\delta(v)$. Thus $\delta(u_{k-1})$ is the weight of the path to the node u_{k-1} and of length $k-1$. Thus by inductive hypothesis, after $k-1$ iterations of the for loop, we would have D[u_{k-1}] $= \delta(u_{k-1})$ and in the kth iteration of the for loop, the edge(u_{k-1}, v) will be examined and the value of D[v] gets changed as D[v] $= \delta(u_{k-1}) + w(u_{k-1}, v) = \delta(u_{k-1}) + w(u_{k-1}, v) = \delta(v)$. After the kth iteration, the value of D[v] will stay the same, because there is no value smaller than $\delta(v)$.

The length of the shortest path from some vertex to v can be a maximum of $|V| - 1$ in the worst case (in the case where there is a linear path from one vertex to another vertex through all the other vertices), and that is the reason that the outer loop executes for a maximum of $|V| - 1$ times.

Time Complexity

The time taken to execute the first for loop is bounded by $O(n)$, since there are n vertices in the graph. In the second sequence, there is a for loop and inside that, there is a nested for loop. The outer loop executes for almost n times. The inner for loop executes up to m times every time. Thus the time complexity of this sequence can be described as $O(nm)$. Finally the time taken to run the final for loop is bounded by $O(n)$. Thus the time complexity of the entire program can be described as $O(nm)$.

3 Question 3

Consider that each vertex in the graph is labelled as a unique integer. The algorithm is shown below:

Algorithm: Linear time path($G = \{V, E\}$)

```
S = Null

for v in V:
    for e(v,u) in E:
        S.append(v,u,w)      #where w is the weight of the edge

S = radixSort(S)    #perform a sort such that S is sorted u in every v

G1 = Null
```

```
G1.V1 = G.V
v1, u1, w_max = S[0].v, S[0].u, S[0].w

for each [a,b,c] in S:
    if a == v1 and b == u1:
        if c > w_max:
            w_max = c
    else:
        G1.E1.output(v1,u1,w_max)
        v1, u1, w_max = a, b, c

return G1
```

Explanation

The first step in the algorithm is that we are intializing a collection S to $Null$. We assume that each vertex is denoted by an integer. In the first for loop, we append each edge to the collection along with their weights, such that the collection is sorted in v (This is possible, since the graph is given in an adjacency list. We can add all the edges from one vertex to the collection and move on to the next vertex). But at this step, we cannot guarantee that all the edges between the same set of vertices are right next to each other in the collection. That depends whether the adjacency list is sorted, which is not mentioned in the question.

To ensure that all edges between the same set of vertices are grouped together, we then perform Radixsort (linear time sorting algorithm, chosen because the question demands the entire algorithm to be in linear time) on the collection, such that it sorts the collection to ensure that either $v_1 < v_2$ or $v_1 = v_2$ and $u_1 \leq u_2$. Thus we have all edges between the set of vertices grouped together in the collection. But the weights between these vertices may not be sorted in the collection.

Finally, the last for loop in the program scans through collection and picks the edge with maximum weight in every set of vertices and appends it to a new graph. We finally return the new graph, which would now have only one edge every from the set of edges between every pair of vertices from the original graph and the weight of that edge would the maximum edge weight from the original graph.

Time Complexity

The first statement would take time $O(1)$. Then the outer for loop in the nested loop sequence would execute n times, as it executes for all the vertices. The inner for loop would execute for a maximum of m times, all for all executions of the outer for loops combined, since there are only m edges in the graph. Thus the total time complexity of the nested loop sequence would be $O(n+m)$. The sorting algorithm used is a linear time sorting algorithm, and thus the time complexity of the statement will be in the order of the length of the collection, which is m (since, the length of the collection would be equal to the total number of edges in the original graph). Thus, the time complexity of this statement will be in the order $O(m)$. All the following statements before the for loop will execute in $O(1)$ time. The number of times the final for loop is executed is bound by the length of the collection which is m. Both the if branch and the else branch inside the loop can be executed in time $O(1)$. Thus the total time complexity of the for loop is $O(m)$.

Thus the total time complexity of the program is $O(1) + O(n + m) + O(m) + O(m)$ or it can simply be described as $O(n + m)$. Thus this algorithm, executes in a linear time.

4 Question 4

The algorithm is given below:

Algorithm: Median_Quicksort(arr, start, end)

```
if (start<end):
    #finding pivot
    pivot = index(median(arr, start, end))      #linear time median finding

    #parition
    for (i=start to i=end):

        val = arr[i]
        if (val<arr[pivot]):
            arr[i] = arr[pivot+1]
            arr[pivot+1] = arr[pivot]
            arr[pivot] = val
            pivot++

    #recursion
    Median_Quicksort(arr, start, pivot-1)
    Median_Quicksort(arr, pivot, end)
```

The algorithm given above is the well known and familiar quicksort algorithm, with the only exception that it uses a linear time median finding algorithm, to find the pivot every time it is called.

Proof

We see that the program has three parts in execution, namely:

1. Finding pivot
2. Partition
3. Recursion

We can see that the partition is performed in $O(n)$ time, since it executes for the length of the array from *start* to *end*. Since, we have assumed that the median finding algorithm also take time $O(n)$, all the other two parts of the program except for *Recursion* is executed in $O(n)$ time. Thus this algorithm has to satisfy the recurrence relation $T(n) = 2T(n/2) + O(n)$. We know that the solution to this recurrence relation is $O(n \log n)$. Thus the time complexity of this algorithm is $O(n \log n)$, even in the worst case.

Why this algorithm is not preferred

We have mentioned that the time complexity of the median finding algorithm is of the order $O(n)$. But in most cases, the hidden constant in the $O(n)$ can be quite large. Even if the value of this constant is smaller for most instances, many small values can contribute to a much larger value in the recurrence relation, thus making the hidden constant in $O(n \log n)$ larger. And for larger constant in $O(n)$ complexities also contribute to much higher constants in the $O(n \log n)$ relation. The original quicksort algorithm, where the pivot is chosen randomly might take $O(n^2)$ in the worst case. But such worst-case scenarios are rare occurrences. In other occurrences, even if the pivot does not split the array into two equal haves, in most cases, splits the array into two halves that are somehow balanced sizes. Thus the hidden constant in the $O(n \log n)$, average execution time of the typical quicksort algorithms, tend to be lower than the constant in this modified algorithm in most cases. Thus in pratice, this modified quicksort algorithm is not generally preferred.