CSC265 Homework 5

Note: This assignment was rushed in a few hours and poorly written

Question 1 (Mark: 6.5/8)

In this question, we will take use of the, combined union-by-rank and path-compression heuristic disjoint set, which is proved to runs in $O(m \log^*(n))$ time for m disjoint-set operations on n elements.

In the following code, we use the array T to record which vertices added into the traversal. For the sake of brevity, after each MAKE-SET(u), we treat u as both a vertex and an object in the disjoint set. $\mathcal{S}$ will be maintained and represented by the set of all FIND-SET(u) for which u that we MAKE-SET.

```
1  Initialize Array D[1...n]
2  Initialize Array T[1...n] where each entry is False
3  D[1] = 0
4  DISTANCE(1)
5  TRAVERSE(1, NULL)
6
7  DISTANCE(u):
8      for v in A[u]:
9          D[v] = D[u] + w(u,v)
10         DISTANCE(v)
11
12 TRAVERSE(u, p):
13     MAKE-SET(u)
14     for v in A[u]:
15         TRAVERSE(v, u)
16     T[u] = True
17     if p != NULL:
18         UNION(u,p) and exchange u and p to make p the root if necessary
19     for v in B[u]:
20         if T[v] is True:
21             w = FIND-SET(v)
22             δ(u,v) = D[u] + D[v] - 2D[w]
```

We first prove the correctness:

For the DISTANCE function, we conducted a pre-order traversal, resulting in D[u] for each $u \in T$ is defined. The correctness is established through induction. Initially, we set D[1] $= 0$, as it represents the distance from the root to itself. Assuming D[u] is correct, we then set D[v] for each child v of u to be D[u] + w(u, v), where w(u, v) is the weight associated with the edge between u and v. Since T is a rooted tree, the distance from a node to the root equals the sum of the distance between the node and its parent and the distance between the parent and the root. Therefore, by induction, D[1...n] is correct after the execution of line 4.

According to the definition of the TRAVERSE function, p always represents the parent of u, and MAKE-SET(p) is executed if p is not NULL. We establish the maintenance of $\mathcal{S}$ through induction. Lines 12-16 demonstrate that TRAVERSE follows a post-order traversal. When the first node is incorporated into the traversal, it has no children. While traversing down, each node on the path undergoes MAKE-SET, resulting in all nodes being singletons. Therefore, the property of $\mathcal{S}$ is initially fulfilled.

As this node is singleton, line 18 makes p as the root of the new set. This sets the stage for the next node to be added to the traversal. Since p will be on its path to the root, and u is a descendant of p that has been previously processed and is not included in the descendants of any other child of p, the algorithm may traverse down from p before the next node is added, adding more singleton sets S_i to $\mathcal{S}$. Alternatively, the next node added might be p. In either case, at the moment the new node is included in the traversal, the property of $\mathcal{S}$ is maintained.

Assuming that when u is added to the traversal, the property of $\mathcal{S}$ is maintained, following the identical rationale as presented earlier, the algorithm readies itself for the next node, ensuring that at the moment of its addition, the property of $\mathcal{S}$ holds.

The traversal terminates when the root is added to the traversal, with p being NULL. Consequently, in this iteration, no operation is performed on $\mathcal{S}$. Therefore, by induction, the property of $\mathcal{S}$ consistently holds immediately after a node is added to the traversal.

(-1.5 Marks, flawed proof) Considering each edge $(u, v) \in P$, one of them must be added later than the other. Without loss of generality, let's assume that u is added first. Then, when v is added, the algorithm iterates through u in line 17, and T$[v]$ is true. According to the invariant of set $\mathcal{S}$, v belongs to S_i for some i. Following line 18 in our algorithm, the root of S_i must always be u_i on the path from u to $u_0 = 1$. Consequently, both u and v are descendants of w. To demonstrate that w is the deepest common ancestor of u and v, we assume the existence of u_j on the path that is deeper than w. Then, u is a descendant of u_j. However, the path $u - u_j - w - u$ forms a cycle, contradicting the fact that T is a tree. Therefore, the path $u - w - v$ is simple. Since T is a tree, the path from w to the root 1 is unique, as well as the paths u to w and w to v. Consequently, the distance between u and v is correctly calculated in line 22.

We now analyze the time complexity:

Initializing two arrays takes $O(n)$. Calling DISTANCE(1) preforms a pre-order traverse through T, with constant number of array access and calculation, which contributes to a complexity of $O(n)$.

Similarly, TRAVERSE(1, NULL) preforms a post-order traverse through T, a total of n nodes (line 14-15). We first analyze line 13 and 16-18. For each node, other than constant number of array access and comparison, one MAKE-SET and one UNION is preformed, contributing to a total of $2n$ operations along with $O(n)$ for other operations. For line 19-22, it traverse through all edges twice in B, however, for each edge, only once the if condition of line 20 is true. Thus m FIND-SET are executed and along with $O(m)$ for operations in line 22.

Therefore we preformed $2n + m$ operations on this disjoint set of m elements, which has a complexity of $O((2n + m)\log^*(n)) = O((n+m)\log^*(n))$. Summing all complexity mentioned above, we have $O(n)+O(n)+O(n)+O(m)+O((n+m)\log^*(n)) = O((n+m)\log^*(n))$ as desired.

Question 2 (Mark: 9/12)

Part a. (Mark: 2/2) Bad implementation of BFS, have typos

```
Initialize Matrix D[1...n][1...n] such that each entry is ∞

For i = 1 to n:
    Initialize a linkedlist L
    For j = 1 to n skipping i:
        If A[i][j] = 1:
            L.append(j)
            D[i][j] = 1
    For j = 2 to n-1:
        Initialize a linkedlist L'
        For k in L:
            For z = 1 to n skipping i:
                If M[k][z] = 1 AND D[i][z] = ∞:
                    L'.append(z)
                    D[i,z] = j
        L = L'
```

We first prove the correctness:

Consider an arbitrary i from 1 to n. In lines 4-8, vertices $j \neq i$ that are adjacent to i are identified, and their distance is set to 1, stored in L. Thus, L now contains vertices at a distance of 1 from i.

Let's establish an invariant through induction. Assume that for $j \in 2, \ldots, n-1$, L holds vertices at exactly a distance of $j-1$ from i. Line 11 iterates through such vertex k, and line 12 identifies vertices z not reached by i at a distance $\leq j-1$, setting their distance to k to 1. These vertices z are stored in L' and have a distance of j to i, making line 15 correct. Setting L to L' concludes the correctness of the invariant.

As there are only n vertices and the maximum distance is $n-1$, the for-loop in line 9 indeed finds all distances for i and stores them in D. Since i is chosen arbitrarily, we conclude that the algorithm correctly generates the distance matrix D.

Now we analyze the time complexity:

Let's analyze lines 9-16. For each j, L contains vertices at exactly distance j from i. Iterating from $j = 2$ to $n-1$, line 11 is executed at most $n-1$ times since i cannot be in L. Line 12 takes $n-1$ iterations, each with a constant number of matrix accesses, comparisons, and linked list appends, all taking constant time. Thus, lines 11-15 take $O(n^2)$ during the entire for-loop in line 9. Since lines 10 and 16 each take constant time, we conclude that lines 9-16 take a total of $O(n^2)$.

Line 4 takes constant time, and the for-loop in line 5 has $n-1$ iterations, each with a constant number of matrix accesses, comparisons, and linked list appends, contributing to $O(n)$. Summing all together, lines 4-16 take $O(n^2)$.

As initializing D takes $O(n^2)$ and line 3 has n iterations, the overall time complexity is $O(n^2) + O(n \times n^2) = O(n^3)$ as desired.

Part b. (Mark: 3/4) -1 Marks, lots of typos and wrong sign at line 5

```
1 COMPUTE_D(D_sq, P)
2     Initialize Matrix D[1...n][1...n] such that each entry is 0
3     For i = 1 to n:
4         For j = 1 to n:
5             D[i][j] = 2S_sq[i][j] + P[i][j]
6     return D
```

The initialization step takes $O(n^2)$, and the nested for-loop iterates n^2 times, each with a constant number of matrix accesses and calculations. Overall, this contributes to a time complexity of $O(n^2)$ as desired.

To establish correctness, we derive a formula for D_{sq} in terms of D and P. Let the distance between u and v be d; then the shortest path from u to v has d edges.

As the squared graph introduces a new edge for vertices at distance 2, we can pair each two adjacent edges in the path into one. Therefore, if d is even, the length of the new path is $d/2$, and if d is odd, the new path length is $(d+1)/2$. We assert that this new path is the shortest; if there exists a path of length $\leq (d+p)/2 - 1$, where p is the parity, it implies that the distance between every adjacent vertex on the path is ≤ 2 by the definition of a squared graph. However, the path obtained by joining the shortest paths between those vertices in D has a length of $d + p - 2 \leq d$, contradicting the fact that d is the distance between u and v.

Thus, we have proven that the distance between u and v in the squared graph is $(d+p)/2$, precisely corresponding to line 6 in the algorithm. Hence, our algorithm correctly generates the matrix D.

Part c. (Mark: 3/4) -1 Marks, includes typo and flawed proof

```
COMPUTE_P(D_sq, T)
    Initialize Matrix P[1...n][1...n] such that each entry is 0
    For i = 1 to n:
        For j = 1 to n skipping i:
            If T[i,j] < D_sq[i][j]:
                P[i][j] = 1
    return P
```

The initialization step takes $O(n^2)$, and the nested for-loop iterates n^2 times, each with a constant number of matrix accesses and comparisons. Overall, this contributes to a time complexity of $O(n^2)$ as desired.

Consider $u \neq v$, let w be adjacent to v. (Existence of w can be shown by choosing the second vertex on the path from v to w). Let the distance between u and v be d in G. By definition of distance, the distance between u and w is either $d-1$ or $d+1$. The existence of w shows that at least one vertex adjacent to v that $\delta(u,w) = d-1$.

If d is odd, then for w such that $\delta(u,w) = d-1$, $D_{sq}[u][w] = D_{sq}[u][v] - 1$. And for w such that $\delta(u,w) = d+1$, $D_{sq}[u][w] = D_{sq}[u][v]$. By definition, $T[u,v]$ is the average of $D_{sq}[u][w]$ for all such w, and at least one w that $\delta(u,w) < D_{sq}[u,v]$. Therefore $T[u,v] < D_{sq}[u,v]$.

If d is even, then for w such that $\delta(u,w) = d-1$, $D_{sq}[u][w] = D_{sq}[u][v]$. And for w such that $\delta(u,w) = d+1$, $D_{sq}[u][w] = D_{sq}[u][v] + 1$. By definition, $T[u,v]$ is the average of $D_{sq}[u][w]$ for all such w, and at least one w that $\delta(u,w) = D_{sq}[u,v]$. Therefore $T[u,v] \geq D_{sq}[u,v]$.

Thus, $P[i][j] = 1 \iff d \text{ is odd} \iff T[i,j] < D_{sq}[i][j]$. This implies that our algorithm is correct.

Part d. (Mark: 3/4)

```
1  D = D_MATRIX(A, 1)
2
3  D_MATRIX(A, order)
4      If order ≥ n:
5          Initialize D[1...n][1...n] such that all entries are 1 except 0 on diagonal
6          return D
7      else:
8          A^2 = SQUARE(A)
9          D_sq = D_MATRIX(A^2, 2×order)
10         T = COMPUTE_T(D_sq, A)
11         P = COMPUTE_P(D_sq, T)
12         D = COMPUTE_D(D_sq, P)
13         return D
```

(-1 Marks, insufficient justification) Let $order = 2^k$ for some $k \in \mathbb{N}$, distinct vertices u and v are adjacent in G^{order} if $\delta(u,v) \leq order$ in G, which can be shown by induction on k. In our algorithm, $order$ is always a power of 2.

We first prove the correctness by induction. The variable $order$ tracks that the algorithm is calculating distance matrix for G^{order}. For the base case where $order \geq n$, every pair $u \neq v$ satisfies $\delta(u,v)$ in G since it's a connected graph. Thus every pair $u \neq v$ is adjacent, which implies that our D in line 5 is correct.

Assume DMATRIX is correct for $order' = 2^k$ and now let $order = 2^{k-1}$ then D is correctly generated since every function in line 8-13 is assumed to be correct. Thus by induction, DMATRIX is correct for $order = 2^k$ for all $k \in \mathbb{N}$. Then line 1 is correct, and our algorithm correctly generates D as desired.

Since each recursion we multiply $order$ by 2 until $order \geq n$, the depth of the recursion is $O(\log n)$. For the last iteration, initializing D takes $O(n^2)$. And for each other iteration, we add the complexity of line 8, 10-12 and get a total of $O(n^\omega) + O(n^\omega) + O(n^2) + O(n^2) = O(n^\omega)$. Then each iteration, including the last iteration, have time complexity of $O(n^\omega)$.

Finally, we conclude that the algorithm takes $O(\log n) \times O(n^\omega) = O(n^\omega \log n)$ as desired. $\square$