ADA HW_3

> 范秉逸 B10902117

1 reference Myself

2 reference Myself

3 reference Myself

4 reference Myself

5 reference

(f), (g) :https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/

6 reference Myself

5

(a)

By Kosaraju Algorithm, we know that if a sequence of starting time of a series number is same as the inverse sequence of finishing time, this series number is in the same SCC <1>
Thus, if vertices a and b are in the same SCC and their starting time is $s_b < s_a$, their finishing time must be $f_a < f_b$
If there is a node between a and b called c, their finishing time is $f_a < f_c < f_b$, and a b is in the same SCC, $s_b < s_c < s_a$ must exist. Thus, by <1>, a, b, and c are in the same SCC.

(b)

Minquan W. Rd. is the station from which taking a ride to the farthest station from it costs the least.

1. Look at the graph of all station and choose the stations in the middle of the graph in my checklist.
2. Look 1 by 1 in the checklist to find the station from which taking a ride to the farthest station from it costs the least in the website.
3. Find the answer.

(c)

Assume there exist k Professor Cheng won't construct any point-cycle.
Try to build a graph with no point cycle :
K professor Cheng which 1 point to 2, 2 point to 3, ...,and k-1 point to k.
Since every point must have exactly one out-degree, k must point to a previous point. Thus, k Professor Cheng must construct at least one point-cycle. Contradiction !
Let n be the minimum index in a point-cycle, n > 0 and k be the length of the point-cycle, k > 0.
$a_k = F_k(n) = n$. Thus $a_{k+k} = F_{k+k}(n) = n$. Thus, $a_{N+T} = a_N$

(d) & (e)

Time Complexity & explain

1. ☐ (start from 1), an array path[N] (start from 1), an array ans[N] (start from 1), an array Length[N - 1] (start from 1), and the tranpose graph.
2. We run dfs in the tranpose graph. For each point, we marl them visited. And use a vector to store the path. If we reach a point with 0 out-degree, we pop the point in the path until we have another road to travers. If the path is empty, start from a point we haven't visit. If we reach to a visited point, this mean our path complete a loop. We store the path in loop(push_back) and clear path. Start from a point we haven't visit. Recursively do this until we visit all the point. Thus, we have all the loop.
3. Let all points unvisited
4. Find the answer for every loop. Start with a point with index c in loop[j]. Let K = K % loop[j].len(). If c - K < 0, K = loop[j].len() - (K - c - 1). And store ans[loop[j][c]] = loop[j][K]. By this we find all answer of points in loops. This is to prevent us from moving out of the array size.
5. Run dfs in the tranpose graph again. This time we deal with points not in loops. If we visit a point, mark it visited. When we reach the points that have no answer, let the index of the point in the path[] be c and create another path[] that start a point in a loop. If c - K < 0, K = K - c - 1. This mean we reach into a loop. Then, we can use the method in 4. to find the answer of path[c], save it in ans[path[c]]. If we reach a point with 0 out-degree, we pop the point in the path until we have another road to travers. If the path is empty, start from a point we haven't visit. If we reach to a visited point, start from a point we haven't visit. Recursively do this until we visit all the point. Thus, we get all i-th point answer in ans[ ].

Time Complexity

1. 2. Run all N points by dfs : O(N)
2. 3. Let all points unvisited : O(N)
3. 4. Run all points in the loops and find answer : O(N) * O(1) = O(N)
4. 5. Run all N points by dfs and find answer : O(N) * O(1) = O(N) Total : O(N) + O(N) + O(N) + O(N) = O(N) Find answer: O(1) //search ans[i]

(f) & (g)

Algorithm

1. Use Kosaraju's algorithm to find all SCC, set the max length of these point of SCC to INF. Find SCC means we found cycle, according to the description, the length of cycle is INF.
2. There will be three kinds of nodes. First one is the node have longest distance. Second is the node that point to a node have largest length. Third one is the node that doesn't meet the condition of first and second one. Since first one has already had the answer, so we skip it.
3. For second case node, pick the max length of the longest path from its out-degree and plus one. This is the answer of the node. Then, we do it until only third case node left.
4. ☐ to store the distance that the node K pass another node. Find longest distance by :
    1. Create a topological order of all vertices that the node K will pass
    2. Do following for every vertex u in topological order.
    3. Do following for every adjacent vertex v of u
    4. if (dist[v] < dist[u] + length(u, v))

    5. dist[v] = dist[u] + length(u, v)
5. Then, we find the largest distance for the node K and go repeatly do 3. and 4. until all node have largest distance.
   Notice that when we find the longest distance, we put this node into first case and the node next to it to second case.

Explain

1. For step 1, we use Kosaraju's algorithm to find SCC first to deal with loop first.
2. For step 2, we spilt all possible condition into three catagories to deal with them one by one.
3. For step 3, because all possible reach distance all add one, choose the longest path from its out-degree must get longesst distance.
4. For step 4, we change Dijkstra's algorithm into finding longest path for a node K to let the node next to it can use step 3 to get the answer.
   Thus, we can get the longest distance of all nodes.

Time Complexity

1. Kosaraju's algorithm: O(V+E)
2. O(V) since there is only V node
3. O(E) since there is only E edge need to consider
4. Because the third case only appears when a graph is disjoint to another and there is only V node, we only do this in total V+E. Dijkstra's algorithm: O(V+E)
   Total: O(V+E)+O(V)+O(E)+O(V+E) = O(V+E)

6

(a)

algorithm:

stack[] to store our stack

1. create a array sum[N]. In each sum[i] store the sum of value from 0-th to i-th, so we can get the sum of top x node by sum[stack.size] - sum[stack.size - x] which is O(1). By this we can solve dd.
2. pop and push just like normal stack. But also need to maintain our sum. If is pop, pop the sum[stack.size] too. If is push, sum[stack.size] = sum[stack.size - 1] + stack[stack.size]. By this we can solve PUT and TAKE.
3. For cc we pop x node out and save it in the array O[]. We use quick select algorithm to save $\lceil x/k \rceil$ (call it outNumber)-th biggest value as Min
4. Then we push the node to the stack (use PUT) in origin order but only if the value of the node is smaller than Min. For the node >= Min, we add them together. By 3. and 4. we can solve cc.

Accounting method for stack

1. Guess per-op amortized costs:

Operation	Actual cost	Amortized cost	Credit change

PUT	1	k+2	**存 k+1 元存在帳戶裡**
TAKE	1	0	**從 popped object 的帳戶領 1 元**
CC x k	k+1	0	**從每個 deleted object(最多x個) 的帳戶領 k + 1 元**
DD x	1	1	**不變動**

1. Validity check: Show that every object has credit ≥ 0
    1. push: the pushed object is deposited $1 credit
    2. pop and CC: use the credit stored with the popped object
    3. There is always enough credit to pay for each operation If we put T items in the stack, we will have T*(k+1) dollar. Since we can only TAKE x times and CC y items $(x + y \le T, 1 \le k \le k)$. Thus, $(x*(k+1)+y) \le T*(k+1)$. Prove 3.
2. Per-op amortized costs for PUT is O(k) and TAKE, CC is O(1), so total cost is o(kM) < o($Mk^{1.1101420}$)

(b)

Potential method

1. Guess $\Phi(T_i) = \frac{3+\sqrt{5}}{2} * T(i).sum - \frac{1+\sqrt{5}}{2} * T(i).size$, where T(i).sum is the number of inserted objects and T(i).size is the table size with i-th op
2. Validity check:
    1. Φ(T_0) = 0, because T(0).sum and T(0).size is initially 0
    2. Φ(T_i) ≥ 0, because T(i).size must $\geq$ T(i).sum to put thing s into the table
3. Compute the amortized cost of INCREMENT:
    1. Insertion without resize
       actual cost: 1 (insert)
       T(i).size is same, T(i).sum = T.sum + 1
       $ C_i = \frac{3+\sqrt{5}}{2}*(1)-\frac{1+\sqrt{5}}{2}*(0) + 1 = \frac{5+\sqrt{5}}{2} $
    2. Insertion without resize
       actual cost: $F_{n-1} + 1$ (copy and insert)
       T(i).size = T(i-1).size + $F_n - H_{n-1}$ is same, T(i).sum = T.sum + 1
       $ C_i = \frac{3+\sqrt{5}}{2}*(1)-\frac{1+\sqrt{5}}{2}*(F_n-F_{n-1}) + F_{n-1} + 1 = \frac{3+\sqrt{5}}{2}*(1)-\frac{1+\sqrt{5}}{2}*(F_{n-2}) + F_{n-1} + 1 \approx \frac{5+\sqrt{5}}{2} + F_{n-1} - F{n-1} = \frac{5+\sqrt{5}}{2}$
4. All operations have $\frac{5+\sqrt{5}}{2}$ = O(1) amortized cost, so the total cost of n operations is O(n)

(c)

algorithm:

1. When we get (l_i, r_i, c_i), find the node j that j.r $\geq r_i$, copy the value of it as (j.l, j.r, j.c) and delete it from the tree.
2. Find the node k that k.l $\leq l_i$, copy the value of it as (k.l, k.r, k.c) and delete it from the tree.
3. Take out the node between j and k, and delete them from the tree.
4. Create three node, first is the left node with (k.l, $l_i - 1$, k.c), middle node with (l_i, r_i, c_i), and right node with ($r_i + 1$, j.r, j.c)

5. Push these three node into the balanced binary search tree.

Explain:

For step 1, 2, and 3, we are modifying the node into the pattern we want. We find that if the nodes between j and k is very large the time complexity may exceed O(logN), but if we think carefully, we find that after the operation we will delete (k - j + 1 - 3) nodes. Thus, the number of node need to remove at later operation will cost last. We can think this with amortized analysis. In the end, we will get the time complexity as below.

Time complexity:

To search node: O(logN)

(d)

Since for each u belongs to child(v) and for each v belongs to child(v). By the question, when u and v, $x.w * y.w$ will be add to sum. But when the value of u and v is swap and the value of x and y swap again, $x.w * y.w$ will be add to the sum again.
Thus, a pair (x, y) will be found twice. So, total multiplication will be 2*C{V取2}=V*(V-1)
Total time complexity after running Algorithm 1 on all vertices is O(V^2)
Thus, for each node, the amortized time complexity is $\frac{O(V^2)}{V} = O(V)$