MOKSH SHUKLA
180433

Theoretical Assignment 2: PART 1

ESO207A: Data Structures & Algorithms

2021-03-22

Problem 1

Pseudo Code: Pre-Processing

1. We reverse the direction of edges in the $G = (V, E)$ to obtain a reverse directed graph.
2. Obtain all nodes than can be reached to s, t separately in reverse directed graph using DFS.
3. Find intersection of the edges between the nodes that can be reached from s, t. The intersection gives the nodes that lie on the path from s to t.
4. Find the number of edges passing through the node (found in Step 3) connecting s. Do this for each node.
5. Now if a particular node is articulation point then, number of edges found in previous step for this node is equal to number of edges connecting s, t. If this condition is satisfied, output 1, else output 0.

Problem 2

The approach we use is similar to what we used to calculate inversions in an array during lectures. We recursively split the array into 2 halves: left & right. We then calculate *strong domination* pairs in both the left and right halves separately. Now, when we merge the 2 splitted halves there will also exist some *strong domination* pairs, in which one element is in left half and other in right half. So, while merging we need to calculate the *strong domination* pairs across the halves. Then finally we add these 3 counts to get the result.

Pseudo Code

```
int mergeInversions(A, i, mid, k, C) {
    left = i;
    mid = (left + right) / 2;
    right = mid + 1;
    idx = 0; count = 0;
    while (left <= mid and right <= k) {
        if (A[left] <= 2 * A[right]) {
            left++;
        }
        else {
            count += mid - left;
            right++;
        }
    }
    left = i; right = mid + 1;
    while (left <= mid and right <= k) {
        if (A[left] < A[right]) {
            C[idx] = A[left];
            idx++; left++;
        }
        else {
            C[idx] = A[right];
            idx++; right++;
        }
    }
    while (left <= mid) {
        C[idx] = A[left];
        idx++; left++
    }
    while (right <= k) {
        C[idx] = A[right];
        idx++; right++;
    }

    return count;

}

int countDomination(A, start, end) {
    if (start == end)
        return 0;
    else {
        mid = (start + end) / 2;
        count1 = countDomination(A, start, mid); // count dominations in 1st half of array T(n/2)
        count2 = countDomination(A, mid + 1, end); //count dominations in other half T(n/2)
        C[end - start + 1]; //temp array
        count3 = mergeInversions(A, start, mid, end, C);
        return count1 + count2 + count3;
    }
}
```

Time Complexity

The algorithm makes call to recursive function *countDomination* twice and the loop functions in *mergeInversions* have complexity cn apart from which all other operations in this functions are constant time. So, for the algorithm:

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ T(n) &= cn + c*2*(n/2) + c*2*(2*(n/4)) + ... + c*2^{\log_2 n}*1 \\ T(n) &= c\,\underbrace{(n+n+n....+n)}_{\text{log n terms}} \\ T(n) &= cn\log n \\ T(n) &= \mathcal{O}(n\log n) \end{aligned}$$

Problem 3

We have in P several points along a vertical line and also we are given 2 points s,t to the left and right of line containing P. Now, by simple geometry it can be easily proved that for $dist(s, P_i) + dist(t, P_i)$ to be minimum, point P_i should lie on line joining s, t. So, for this we calculate the intersection point between line joining s, t and the vertical line containing P. Now it is not necessary that this intersection point exists in set of points P. So, to find the nearest points with minimum distance we insert the y-coordinates of all points in P into a ***Red Black Tree***. After inserting in RBT, we subsequently search the predecessor and successor of the found y-coordinate of intersection point. Now for both the predecessor and successor we find $dist(s, P_i) + dist(t, P_i)$ and return the respective point for which the distance is minimum. Using RBT helps, since RBT is a balanced BST and hence query for predecessor and successor can be done in $\mathcal{O}(\log n)$.

Pseudo Code

```
struct node()
{
    int val;
    int x, y;
    node* left; node* right;
    bool color; // stores info about node colors

};

insertRBTree(root,y) //standard insertion discussed in class

int successor(root, x) {
    if (root->val <= x)
        return successor(root->right, x);
    else {
        temp = successor(root->left, x);
        if (temp == NULL)
            return root;
        else
            return temp;
    }

}

int predecessor(root, x) {
    if (root->val > x)
        return predecessor(root->left, x);
    else {
        temp = predecessor(root->right, x);
        if (temp == NULL)
            return root;
        else
            return temp;
    }
}
```

```

euclidDist(x1, y1, x2, y2) {
    return sqrt(pow((x1 - x2), 2) + pow((y1 - y2), 2));
}



minCost_Point(P, s, t, root) {
    for (yCoord in P)
        insertRBTree(root, yCoord); // all y-coordinate values inserted in RBT: Pre-processing
    x = P[0].x;
    yIntersect = s.y + ((t.y - s.y) / (t.x - s.x)) * (x - s.x);
    point1 = successor(root, yIntersect);
    point2 = predecessor(root, yIntersect);
    if (point1 == NULL)
        return (x, point2->val);
    else if (point2 == NULL)
        return (x, point1->val);
    dist1 = euclidDist(x, point1->val, s.x, s.v) + euclidDist(x, point1->val, t.x, t.y);
    dist2 = euclidDist(x, point2->val, s.x, s.v) + euclidDist(x, point2->val, t.x, t.y);
    if (dist1 > dist2)
        return (x, point2->val);
    else
        return (x, point1->val);
}
```

Time Complexity

For pre-processing we use Red Black Tree wherein we insert the y-coordinates of all points in P. Time taken for insertion in RBT is $\mathcal{O}(\log n)$. For each query, our *minCost_Point* called *predecessor* and *successor* functions. For RBT which is a balanced BST, these calls are done in $\mathcal{O}(\log n)$. So, overall time complexity of our *minCost_Point* function is $\mathcal{O}(\log n)$. Therefore,

- Pre-Processing: $\mathcal{O}(\log n)$
- Query: $\mathcal{O}(\log n)$

Space Complexity

Since there are n points in line, Pre-Processing takes $\mathcal{O}(n)$ space.