ESO207: Theoretical Assignment 2 - Part 1

Debaditya Bhattacharya

March 22, 2021

Name: Debaditya Bhattacharya **Roll No:** 190254
Email id: debbh@iitk.ac.in **Hackerrank Id:** debbh922

Problem 1

Description:

To solve this question we first traverse the graph from s to t to obtain a path P. This path is guaranteed to pass through the 1-connected edges. We then construct an graph $G' = (G \backslash P) \cup P^{-1}$. G' is a graph with its edges where the path P is reversed. The graph G' is now converted into a graph which is disjoint into groups at the 1-connected edges in the ordinal graph. We then try to reach t from s in G'. We traverse groups and when we are unable to reach t we traverse the path to find the 1-connected edge where this fails. We add this edge into a array B with index as starting position and value as the finish position. (any given node can have at max one 1-connected edge). We repeat this procedure until we reach t. We then return B.

To handle the queries, we simply check if $B[v_1] == v_2$ Where $e = (v_1, v_2)$.

Algorithm:

```
#Function to find an arbitrary path from s to v with a depth first search
def find_path(V, E, s, t)
    visited <- array of zeros of size V
    path <- head of linked list based stack #Path can be stored in a linked stack.
    stack.append(s) #Stack to implement depth first search

    #Depth first search
    while stack != NULL
        current <- stack.pop()
        path.append(current) #add current node to path
        if current == t: #If we reach t, return the path
            return path
        adjacent <-E[current]
        visited[current] <-1

        while adjacent!=NULL:
            if visited[adjacent.value] !=1:
                stack.push(adjacent.value)
            adjacent <- adjacent.next
        path.pop() #remove current node from path as does not lead to t.
    return path

#Make a new edge map with the edges in path reversed.
def invert_path_edges( E, path):
    x <- path
    E_0 <- copy E #Copy E

    #Invert edges
    while x.next.next != NULL:
        E_0[x].delete(x.next)
        E_0[x.next].append(x)
        x <- x.next
    return E_0

#Build the data structure given V and adjacency list E, starting point s, and ending point t.
def find_bridges_data_structure(V, E, s, t):
    path <- find_path(V, E, s, t) # Find an arbitrary path

    E_0 <- invert_path_edges(E, path) # Invert edges on path
    first = True # variable to ensure s enters the queue.
    B <- array of size V with elements -1 #Array of size v with entries -1
    group <- array of size V initialized to 0 #stores group number (groups separated by bridges)

```

```
    while group[t] == 0: #Run until we set a group for t
        current_group <- 1
        y <- path
        if first:
            queue.append(s)
            first = False
        else:
            while(group[y.value] != 0) #find last node in path which does not have a group
                x <- y
                y <- y.next
            B[x.value]<-y.value #edge from x-> y must be a bridge as one cannot reach t from s without traversing it
            queue.append(y) #restart traversal in y, after incrementing current group
            current_group++

        #run a breadth first traversal to reach try to reach t by using E_0. Places where it cannot are identified as edges.
        while !queue.isEmpty():
            current <- queue.pop()
            adjacent <- E_0[current]
            while adjacent != NULL:
                if group[adjacent.value] != 0:
                    queue.append(adjacent.value)
                    group[adjacent.value] <- current_group;
                adjacent <- adjacent.next
    return B

#Query function given Bridges and edge
def 1-connectivity(B, e):
    v_1, v_2 <- e
    if B[v_1] == v_2: #There can be at max one bridge from a node.
        return 1
    else:
        return 0
```

Listing 1: 1-connectivity

Time and Space Complexity Analysis: The worst time complexity of the traversing the graph for a path is $T = \mathcal{O}(n+m)$.
The worst case time complexity of producing the inverted graph is $T = \mathcal{O}(m)$.
The worst case time complexity of traversing the nodes to find t from s is $T = \mathcal{O}(m+n)$
The worst case time complexity of incrementing y is $T = \mathcal{O}(m+n)$.
Hence the overall time complexity of the pre-processing step is $T = \mathcal{O}(n+m)$

The space complexity of the array B is $S = \mathcal{O}(n) = \mathcal{O}(n+m)$. The query time is $T = \mathcal{O}(1)$.

Problem 2

Description:

This problem is a modification of the count inversions problem we have discussed in class (solved via a modified merge sort). The modification is that we pass over the array in the merge ($T(n) = \mathcal{O}(n)$) once again to count the strong dominations.

Algorithm:

```
    #Merge and count function
def merge_and_count_strong_domination(A, L, mid, R, count):
    p <- L #index for left array
    j <- mid + 1 #index for right array
    copy <- empty array of size R-L+1
    r <- 0 #index for copy array

    #Pass over the array to count the number of strong dominations.
    while p < mid and j <= right:
        if A[p] > 2 * arr[j]: #Strong domination
            count += mid - p
            j++
        else:
            p++

    #p should be in between L and mid, j should be in between mid and R.
    while p < mid and j <= R:
```

```

        #A[p] <= [j] -> Not an inversion. Copy value at p.
        if (A[p] <= A[j]):
            copy[r] <- A[p]
            r++
            p++

        #A[p] > A[j] -> Inversion. Copy value at j.
        else:
            copy[r] <- A[j]
            j++
            r++

    #Copy the rest of values in left subarray
    while p <= mid:
        copy[r] <- A[p]
        r++
        p++

    #Copy the rest of the values in the right subarray
    while j <= L:
        copy[r] <- A[j]
        r++;
        j++;

    #Overwrite A with the temporary copy array
    for x from 0 to R-L:
        A[i+x] <- copy[x]


#Recursive sort and count algorithm
def sort_and_count_strong_domination(A, L, R, count)

    #Base case
    if L == R:
        return

    else:
        mid <- (L + R) / 2 #dividing the array (for divide and conquer)
        #sort and count left half
        sort_and_count_strong_domination(A, L, mid, count)
        #sort and count right half
        sort_and_count_strong_domination(A, mid+1, R, count)
        #merge and count cross terms
        merge_and_count_strong_domination(A, L, mid, R, count)
    return

#Driver function
def count_strong_domination(A):
    N <- length(A) #Get the length
    count <- 0
    sort_and_count_strong_domination(A, 0, N-1, count)
    return count
```

Listing 2: Count strongly dominant

Time Complexity Analysis: The algorithm is a modified version of the merge sort algorithm. The `sort_and_count_strong_domination` algorithm runs in $T(n)$. and the `merge_and_count_strong_domination` function runs in $T = \mathcal{O}(n)$. The overall relation is can be expressed as

$$T(1) = c \text{ for some constant c}$$

$$T(n) = an + 2T(n/2) \text{ for some constant a}$$

This is the same expression as for merge sort, and hence by inspection we can write $T = \mathcal{O}(n \log n)$.

Problem 3

Description: To solve this problem, we first note that the shortest distance between two points is a line. Given our two points s and t we wish to find the points from the set P whose points lie on the line $x = x_o$. Hence the minimum 3 point distance will be achieved by one of the closest neighbours of the point (x_o, y_o) which is defined as the point of intersection of the line between s and t and $x = x_o$. (y_o can be calculated as $y_o = y_s + \frac{(x_o - x_s)(y_t - y_s)}{x_t - x_s}$). We build a red black tree containing the y values from the points in P. The red-black property will ensure that the tree is a bst and we can easily find the floor and the ceil (which are the closest neighbours) and proceed to calculate the distances and chose the minimum and return the corresponding point.

Algorithm:

```
#Insert a value x in a redblack tree with given root. T = O(logn)
def insert_rbt(root, x):
    ...
    #This code has been discussed in class.

#Funciton to find the value smaller than or equal to x. T = O(logn)
def find_floor(root, x):
    temp <- root
    floor <- NULL
    while temp != null:
        if temp.value <= x:
            floor <- temp
            temp <- temp.right
        else:
            temp <- temp.left
    return floor

#Funciton to find the value greater than to x. T = O(logn)
def find_ceil(root, x):
    temp <- root
    ceil <- NULL
    while temp != null:
        if temp.y > x:
            ceil <- temp
            temp <- temp.right
        else:
            temp <- temp.left
    return ceil

#Preprocessing algorithm. (Generate a red-black tree with n nodes) T = O(nlogn)
def min_cost_point_generate_DS(P):

    root <- NULL (Empty red black tree)

    for point in P:
        if root == NULL:
            root <- point.y
            root.colour <- black #Set first node to be black colour
        else:
            insert_rbt(root, point.y)
    return root

#Query operation given root to red black tree
def query(root , x_o, s, t)

    (x_s,y_s) <- s #Extract coordinates from s
    (x_t,y_t) <- t #Extract coordinates from t

    y_o <- y_s + (x_o - x_s)*(y_t - y_s)/(x_t - x_s) #Calculate y_o

    #Find bounding values
    a <- find_floor(root, y_o) # T = O(logn)
    b <- find_ceil(root, y_o)  # T = O(logn)
    if (a == NULL): #Edge case
        a <- b
    if (b == NULL): #Edge case
        b <- a

    #Calculate distances
    d1 <- distance_3_point(s, a, t) #T = O(1)
    d2 <- distance_3_point(s, b, t) #T = O(1)

    #Return point with minimum distance.
    if d1 < d2:
        return a
    else:
```

```
            return b

    #Driver code
    def min_cost_point(P, s, t):
        rbt <- min_cost_point_generate_DS(P)
        x_o <- P[0].x
        return (x_o, query(root, x_o, s, t)
```

Listing 3: Min Cost Point

Time Complexity Analysis:

Inserting n nodes into a red black tree will take time $T = \mathcal{O}(n \log n)$. Hence the time complexity to build the data structure (which is a red-black tree containing y values from P) will take $T = \mathcal{O}(n \log n)$

The query operation comprises of a few calculation steps like calculating y_o and the distances $d1$ and $d2$, and takes $T = \mathcal{O}(1)$ time. Finding the floor / ceil from the red black tree, each takes time $T = \mathcal{O}(\log n)$. Hence the overall time complexity of the query operation is $T = \mathcal{O}(1) + 2\mathcal{O}(\log n) = \mathcal{O}(\log n)$
