COL351-Assignment1

Pratyush Saini 2019CS10444
Prakhar Aggarwal 2019CS50441

5th August 2021

1 Question 1

1.1 Part a

Proof by contradiction:
Assume for the sake of contraction that G contains two Minimum Spanning Trees T1 and T2. Enumerate the edges of T1 and T2 as follows:

$$E(T1) = \{\text{e1, e2, e3, ... , em}\}, \text{ where e1} < \text{e2} < ... < \text{em}$$
$$E(T2) = \{\text{g1, g2, g3, ... , gm}\}, \text{ where g1} < \text{g2} < ... < \text{gm}$$

Let g_k be the minimum edge in $T_2 \setminus T_1$ and Let e_k be the minimum edge in $T_1 \setminus T_2$.
Without loss of generality assume that wt(e_k) < wt(g_k). By the definition of MST, we can say that $T_2 \cup \{e_k\}$ contains a cycle C, which passes through edge e_k. Let u be any edge of cycle C that is not contained in T_1. At least one such edge must exist, because T_1 is a tree. (We may or may not have $u = g_k$). Because $e_k \in T_1$, we can say that u $\neq e_k$ and thus, u $\in T_2 \setminus T_1$. Now consider the spanning tree T = T_2 + $\{e_k\}$ - {u}. wt(T) = wt(T_2) + wt(e_k) - wt(u) $\leq$ wt(T_2). Now, since T_2 is a minimum spanning tree, we conclude that T is also a minimum spanning Tree. It automatically follows that wt(e_k) = wt(u), which contradicts our assumption that all the edges of G are unique.

1.2 Part b

Algorithm 1 Minimum Spanning Tree

2 Question 2

2.1 Part a

Fibonacci sequence follows the relation $F_{n+1} = F_n + F_{n-1}$
We consider the following algorithm for constructing Huffman tree from given Fibonacci Sequence.

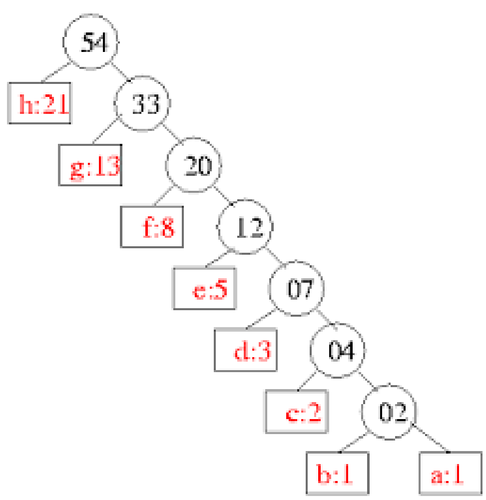
- Create leaf nodes for each character in priority queue(min heap). The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root.
- Extract two nodes with the minimum frequency from the min heap.
- Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
- Repeat the above steps till only one node is left in the min heap.

Claim:

$$\sum_{i=1}^{n} a_k < a_{n+2} \tag{1}$$

Proof: Assume it is true for n, Add a_{n+1} to both the sides, we can show that it is satisfied for n+1.

Claim: There are n internal nodes, all belonging to the set $\{S_1, S_2, \ldots, S_n\}$, where $S_n = \sum_{i=1}^{n} a_k$.
Proof: After j^{th} iteration of algorithm, we have the following frequencies in our queue:
$\sum_{i=1}^{j} a_k$, a_{j+1}, a_{j+2} , ... , a_n. Using the above claim, we have shown that $\sum_{i=1}^{j} a_k$ and a_{j+1} are the lowest frequency nodes in our queue.
So, at the j^{th} iteration, we remove both these frequencies and add the frequency $\sum_{i=1}^{j+1} a_k$ in our queue.
THe resulting huffman tree after all iterations looks like:



Therefore, the encoding of nodes with least frequencies are [(n-2)*"1" + "1"] and [(n-2)*"1" + "0"].
Other node frequencies are [(n-i)*"1" + "0"], i ranging from 3 to n.

2.2 Part b

We can formulate the given problem as :
Given a sequence of characters $\{a_1, a_2, ... , a_{2^k}\}$, where k = 16 in the given question. We have to prove that compression obtained by huffman encoding is same as fixed length encoding.

Assume for the sake of simplicity that the given sequence is present in sorted order of their frequencies, i.e.

$$f_1 \leq f_2 \leq ... \leq f_{2^k} \tag{2}$$

Claim : The huffman tree obtained by the given sequence is full binary tree. **Proof:**

3 Question 3

3.1 Part a

3.1.1 Algorithm

We are given n people and a list of pairs of people who know each other. Mathematically, we can formulate the problem as:
Given: A graph G with n vertices and m edges, we need to find the largest subset S(V,E) of G satisfying the following contraints:

- $\forall u \in S(V), \exists$ at least 5 v $\in S(V)$ such that (u,v) $\in S(E)$
- $\forall u \in S(V), \exists$ at least 5 w $\in S(V)$ such that (u,w) $\notin S(E)$

Algorithm 2 Yet to be decided

```
procedure ADJACENCY_LIST(Edges, N)
    adjList ← [{}, {}, ... {}]                 ▷ list of unordered_set (length N)
    for i from 1 to size(Edges) do
        adjList[Edges[i][0]].insert(Edges[i][1])
        adjList[Edges[i][1]].insert(Edges[i][0])
    end for
    return adjList
end procedure
```

Algorithm 3: Yet to be decided1

```
procedure PEOPLETOINVITE(Edges, N)             ▷ People who know each other
    adjList ← ADJACENCY_LIST(Edges, N)
    RemPeople ← N
    GuestToRemove ← True
    PeopleList ← [1, 1, ....., 1]              ▷ length N

    while GuestToRemove = True do
        GuestToRemove = False
        NewlyRemoved ← []                      ▷ Will store the list of people removed in 1 iteration
        for i from 1 to N do
            if PeopleList[i] = 1 and (size(adjList[i]) < 5 or size(adjList) > RemPeople - 5) then
                GuestToRemove ← True
                PeopleList[i] ← 0
                RemPeople ← RemPeople −1
                NewlyRemoved.insert(i)
            end if
        end for

        if GuestToRemove = True then
            for i from 1 to N do
                if PeopleList[i] = 0 then
                    continue
                end if
                for j from 1 to size(NewlyRemoved) do
```

```
                if adjList[i].find([NewlyRemoved[j]]) != adjList[i].end() then
                    adjList[i].erase(NewlyRemoved[j])              ▷ O(1) (unordered_set)
                end if
            end for
        end for
    end if

end while

GuestList ← []                                                     ▷ Empty List

for i from 1 to N do
    if PeopleList[i] = 1 then
        GuestList.insert(i)
    end if
end for

return GuestList                                                   ▷ Party Invitees
end procedure
```

3.1.2 Time Complexity/Space Complexity

Time Complexity of the algorithm = Time Complexity of the Adjacency_List procedure + Time Complexity of the While loop + Time Complexity of the for loop

- Time Complexity of the Adjacency_List procedure is O(length(Edges)) i.e. O(n^2) (considering input to be a dense graph)
- Time Complexity of the last for loop is O(n)
- While Loop
  - The while loop will run atmost n times, and, in each iteration, the first for loop runs O(n) times: n*O(n) -> O(n^2)
  - For every person added to the **NewlyRemoved** list, the algorithm runs a loop over the **adjList** list (size n) and removes the edge(if present) incident on the person in the **NewlyRemoved** list
  - The loop over the **adjList** helps to remove all such edges and it takes O(1) time (unordered_set) to remove 1 such edge
  - So, we can insert atmost n persons to the **NewlyRemoved** list, and, for each such person, we will loop over the **adjList** (size n), removal of an edge takes O(1) time
  - Therefore, Time Complexity of the while loop will be: O(n^2) + n*O(n)*O(1) -> O(n^2)
- Overall Time Complexity: O(n^2) + O(n) + O(n^2) -> **O(n^2)**

Space Complexity of the algorithm: **O(n^2)** (For storing the adjacency List), other lists can be stored in O(n)

3.1.3 Proof of Correctness

Let G be the graph at the beginning of the while loop, and, G' be the graph at the end of while loop. Let opt(G) denote the optimal subset S $\subseteq$ G (i.e. of maximum possible size) satisfying the mentioned constraints

Claim:

$$opt(G) = opt(G') \tag{3}$$

Proof:

- To obtain G' from G, we remove only those vertices or people who do not satisfy the constraints of the problem
- So, we remove only those vertices whose degree is less than 5 (we need to know atleast 5 people) or whose degree is greater than size(G(V)) - 5 (we need atleast 5 people whom we don't know)
- This implies the above algorithm always produces the largest subset of vertices or people who can be invited to the party
- Also, any vertex which is a solution for G, will also be a solution for G' i.e. it will not be removed (cause, if removed from going from G to G', then it won't be a solution for G also)
- This implies, if u $\in$ opt(G) then u $\in$ opt(G') i.e. opt(G) $\subseteq$ opt(G')
- Also, as G' is a subset of G, this implies opt(G') $\subseteq$ opt(G)
- Hence, opt(G) = opt(G')

Proof of termination: At each iteration, we are removing atleast 1 guest from Alice's invitee list (If there is no such guest, then our algo terminates). As the number of people(n) are finite, this guarantess the **termination of the algorithm**.

3.2 Part b

3.2.1 Algorithm

Given: N people and their respective age, we need to find minimum number of tables to accomodate them such that:

- Each table has a capacity of 10 people
- Age difference between members of the same table should be atmost 10

Approach: We will store the count of persons with a particular age in a list (AgeList), we will have a variable currAge which will start from MinAge (10 in this case). We will increment currAge until we find non-zero value of AgeList[currAge]. Starting from currAge, we will try to go to currAge + 9 (admissible age on a table) and try to fill the table.

Algorithm 4: Yet to be decided2

Yet to be decided

```
procedure MinTables(PersonAge, N)                                  ▷ Replacing n_0 with N
    Tables ← 0
    MaxAge ← 99
    MinAge ← 10
    AgeGap ← 10
    TableCap ← 10
    RemPeople ← N
    AgeList ← [0, 0, ....., 0]                              ▷ length = MaxAge - MinAge + 1
    for i from 1 to N do
        AgeList[PersonAge[i] - MinAge] ← AgeList[PersonAge[i] - MinAge] + 1
    end for

    currAge ← MinAge

    while currAge ⩽ MaxAge and RemPeople ⩾ 0 do

        while currAge ⩽ MaxAge and AgeList[currAge] = 0 do
            currAge ← currAge + 1
        end while

        if currAge > MaxAge then
            return Tables
        end if

        startAge ← currAge
        lastAge ← currAge
        peopleCount ← 0

        while peopleCount ⩽ TableCap and lastAge - startAge ⩽ AgeGap do
            peopleOfThisAge ← min(TableCap - peopleCount, AgeList[lastAge - MinAge])
            AgeList[lastAge - MinAge] ← AgeList[lastAge - MinAge]) - peopleofThisAge
            peopleCount ← peopleCount + peopleOfThisAge
            if AgeList[lastAge - MinAge] = 0 then
                lastAge ← lastAge + 1
            end if
```

```
            if lastAge > MaxAge then
                break
            end if
        end while

        RemPeople ← RemPeople - peopleCount
        Tables ← Tables + 1
        currAge ← lastAge
    end while

    return Tables                                         ▷ Minimum Tables
end procedure
```

3.2.2 Time Complexity/Space Complexity

Time Complexity of the algorithm = Time Complexity of the for loop + Time Complexity of the While loop

- Time Complexity of the for loop is O(n)
- While Loop
  - In each iteration, either we decrement the remaining people by 10 i.e. RemPeople ← RemPeople - 10 or we will increment the currAge by atleast 10
  - This is because, for a given table, either we fill it completely(if possible) i.e. decreasing the remaining people by 10, else, our currAge will be incremented by 10 because we would have looked at all admissible ages at this table which is [currAge, currAge + 9]
  - So, time complexity of the while loop is O((n/10) + (MaxAge - MinAge)/10) i.e. $O((n/10)+$ 9.9)
  - Therefore, Time Complexity of the while loop will be: O(n)
- Overall Time Complexity: O(n) + O(n) -> O(n)

Space Complexity of the algorithm: O(n) (considering the input space), otherwise, space complexity will be O(MaxAge - MinAge).

3.3 Proof of Correctness

Let L be the person's list at the beginning of the while loop, and, L' be the person's list at the end of the while loop. Let opt(L) denote the minimum number of tables required to satisfy the mentioned constraints **Claim:**

$$opt(G) = opt(G') + 1 \tag{4}$$

Proof:

- Let k = opt(L). In a particular iteration, the algorithm accomodates only those people (atmax 10 (table capacity)) on a table which satisfy the age constraint (difference at most 10)
- After the iteration, we obtain L' and have a valid solution of k-1. This implies, opt(L') $\leqslant$ k - 1 i.e. opt(L') $\leqslant$ opt(L) - 1
- Now, suppose p = opt(L'). L' is obtained by removing certain number of people from L and assigning them a single table. Inserting all those elements back to L' gives us L and a valid solution for L of size p+1

- This implies, opt(L) $\leqslant$ p+1 i.e. opt(L) $\leqslant$ opt(L') + 1. This gives us opt(L') + 1 $\leqslant$ opt(L) $\leqslant$ opt(L') + 1
- Hence, opt(L) = opt(L') + 1

Proof of termination: At each iteration, we are assigning a table to atleast 1 guest. As the number of people(n) are finite, this guarantess the **termination of the algorithm**.