COL351 : Analysis and Design of Algorithm
Assignment 1

Gaurav Jain (2019CS10349) & T Abishek (2019CS10407)

1 Minimum Spanning Tree

Let G be an edge-weighted graph with n vertices and m edges satisfying the condition that all the edge weights in G are distinct.

(a) Prove that G has a unique MST.

Solution: Proof by Contradiction:

Let T and T' be two different MSTs of G. Since both T and T' have same number of vertices and are different, there exists at least one edge that belongs to one and not the other. Let the set of such edges be called P.

Let the edge with minimum weight among these edges be e_1. This edge is unique since all edge weights are distinct. Without loss of generality, assume e_1 is in T. If e_1 is added to MST T', it will form a cycle C with e_1 as one of the edges due to the property of tree. Since, T is a tree (acyclic), there must be at least one edge e_2 present in cycle C but not in T because if only edges from tree T is present in cycle C, a cycle wouldn't form.
This e_2 belongs to T' and not T and thus also belongs to set P. As e_1 is the minimum weight edge from set P, wt(e_1) $<$ wt(e_2), as all edge weights are distinct. If e_2 is removed from T' U $\{e_1\}$, (T' U $\{e_1\}$) $\setminus\{e_2\}$ will still be a spanning tree as all paths passing through e_2 can be redirected through e_1.
Thus, (T' U $\{e_1\}$) $\setminus\{e_2\}$ is a spanning tree with cost wt(T') + wt(e_1) - wt(e_2). Since wt(e_1) $<$ wt(e_2), the new weight of this newly formed tree is less than T'. This is a **contradiction**, as we assumed that T' is a MST.

Thus, G has a unique MST. □

(b) If it is given that G has at most n + 8 edges, then design an algorithm that returns a MST of G in O(n) running time.

Observations:

1) Since $m \leq n+8$ and number of edges in a MST is $n-1$, we need to remove atmost 9 edges without disconnecting the graph.
2) Cycles can be detected using DFS which taken $O(m+n)$ time but since $m \leq n+8$ so DFS will taken $O(n)$ time.

Algorithm:

Algorithm 1 MST Algorithm

```
1:  procedure MST(G)                                   ▷ Returns the MST of graph G
2:      n → number of vertices.
3:      m → number of edges and m ≤ n + 8.
4:      while m ≠ n − 1 do
5:          S ← DetectCycle(G)
6:          e ← Maximum edge weight in S
7:          G ← G \ {e}
8:          m ← m − 1
9:      return G                                       ▷ G is the MST of the original graph

10: procedure DetectCycle(G)                           ▷ Returns a set of edges that form a cycle in G
11:     S ← {}
12:     visited ← false for all vertices.
13:     parent ← −1 for all vertices.
14:     start ← −1
15:     end ← −1
16:     for all v ∈ V do
17:         if visited(v) = false then
18:             DFS(v, parent(v))
19:     v ← end
20:     while v ≠ start do
21:         Add v to S
22:         v ← parent(v)
23:     return S                                       ▷ S is the set of edges of a cycle

24: procedure DFS(v, parent)                           ▷ Recursive function to visit vertices
25:     visited(v) ← true
26:     for all u, neighbours of v do
27:         if u = parent then continue
28:         if visited(u) = true then                  ▷ Cycle Detected
29:             end ← u
30:             start ← v
31:             return
32:         parent(u) ← v
33:         DFS(u, parent(u))
```

Correctness of Algorithm:

Claim 1: Let C be any cycle in G, and let edge $e = (v, w)$ be the most expensive edge belonging to C. Then e does not belong to any minimum spanning tree of G.

Proof: Proof by Contradiction
Delete e from T, the MST of G; this partitions the vertices into two components: S, containing node v; and $V \setminus S$, containing node w. Now, the edge we use in place of e should have one end in S and the other in $V \setminus S$, so as to stitch the tree back together.

We can find such an edge by following the cycle C. The edges of C other than e form, by definition, a path P with one end at v and the other at w. If we follow P from v to w, we begin in S and end up in $V \setminus S$, so there is some edge $e' = (v', w')$ on P that crosses from S to $V \setminus S$.

Now consider the set of edges $T' = T \setminus \{e\} \cup \{e'\}$. We claim that T' is a spanning tree. Clearly (V, T') is connected, since (V, T) is connected, and any path in (V, T) that used the edge $e = (v, w)$ can now be

"rerouted" in (V, T') to follow the portion of P from v to v', then the edge e', and then the portion of P from w to w'.

To see that (V, T') is also acyclic, note that the only cycle in $(V, T \cup \{e'\})$ is the one composed of e' and the path P, and this cycle is not present in (V, T') due to the deletion of e. Moreover, since e is the most expensive edge on the cycle C, and e' belongs to C, it must be that e' is cheaper than e, and hence T' is cheaper than T, a contradiction. □

Proof of Correctness:

Consider any edge $e = (v, w)$ removed by the algorithm. At the time that e is removed it is the most expensive edge on C. Thus by claim 1, e does not belong to any minimum spanning tree.

So if we show that the output (V, T) of the algorithm is a spanning tree of G, we will be done. Clearly (V, T) is connected as only edges present in a cycle are removed by the algorithm. Since number of edges in (V, T) is $n - 1$ as the algorithm terminates here. Thus, (V, T) is acyclic and hence it is a spanning tree. □

2 Huffman Encoding

(a) What is the optimal binary Huffman encoding for n letters whose frequencies are the first n Fibonacci numbers? What will be the encoding of the two letters with frequency 1, in the optimal binary Huffman encoding?

Sketch of Huffman encoding algorithm (taken from notes)

(1) Replace the two letters with least frequency by a new symbol $\tilde{a}$.
(2) Set freq($\tilde{a}$) = $\tilde{f} := f_0 + f_1$.
(3) Solve $\tilde{F} = (F \cup \tilde{f}) \setminus \{f_0, f_1\}$ and find optimal binary tree $\tilde{T}$.
(4) If $\tilde{a}$ is node for $\tilde{f}$, then add children a_0 and a_1 to $\tilde{a}$.

Let the frequency vector be F where $F[i]$ represents the i^{th} Fibonacci number.

Claim 1: $\forall n \leq N, F[n] \leq \sum_{i=1}^{n-1} F[i] < F[n+1]$

Proof: Proof by Induction on n
Base Case:

$n = 2$
$F[2] = 1$
$\sum_{i=1}^{1} F[i] = F[1] = 1$
$F[3] = 2$
Thus,
$F[2] \leq \sum_{i=1}^{1} F[i] < F[3]$

Induction Hypothesis:

$F[n-1] \leq \sum_{i=1}^{n-2} F[i] < F[n]$

Induction Step:

$F[n-1] \leq \sum_{i=1}^{n-2} F[i]$ (Induction Hypothesis)
$F[n-1] + F[n-1] \leq \sum_{i=1}^{n-2} F[i] + F[n-1]$
$\sum_{i=1}^{n-1} F[i] \geq F[n-1] + F[n-1] \geq F[n-1] + F[n-2]$
$\sum_{i=1}^{n-1} F[i] \geq F[n]$ (A)
$\sum_{i=1}^{n-2} F[i] < F[n]$ (Induction Hypothesis)
$\sum_{i=1}^{n-2} F[i] + F[n-1] < F[n] + F[n-1]$
$\sum_{i=1}^{n-1} F[i] < F[n+1]$ (B)
From (A) and (B), $F[n] \leq \sum_{i=1}^{n-1} F[i] < F[n+1]$.

By PMI, $\forall n \leq N, F[n] \leq \sum_{i=1}^{n-1} F[i] < F[n+1]$. □

Claim 2: In k^{th} recursive call, the least two frequency symbols are $\sum_{i=1}^{k} F[i]$ and $F[k+1]$.

Proof: Proof by Induction on k
Base Case:

$k = 1$
Initially, the least two frequencies in F are $F[1]$ and $F[2]$.
So, $\sum_{i=1}^{1} F[i]$ and $F[2]$ are least two frequency symbols.

Induction Hypothesis:

$\sum_{i=1}^{k-1} F[i]$ and $F[k]$ are least two frequencies in $(k-1)^{th}$ call.

Induction Step:

Applying the algorithm on $(k-1)^{th}$ call, we get a new symbol $\tilde{a}$ with frequency $\tilde{f}$.
Freq $\tilde{f} = \sum_{i=1}^{k-1} F[i] + F[k]$ as these two are least two frequencies (by induction hypothesis).
So, the new frequency vector for next call $\tilde{F_k}$ is

$$\tilde{F_k} = [\sum_{i=1}^{k} F[i], F[k+1], F[k+2] \dots F[N]] \tag{1}$$

Using the first claim,

$$F[k+1] \leq \sum_{i=1}^{k} F[i] < F[k+2] < F[k+3] \cdots < F[n] \tag{2}$$

Thus when the algorithm is called using this frequency vector.

The least two frequencies are $\sum_{i=1}^{k} F[i]$ and $F[k+1]$.

Hence, by PMI, the above claim is true for all recursive calls. □

Binary Encoding of all symbols is determined using claim 2. So, in the last recursive call, the only two frequencies are $\sum_{i=1}^{n-1} F[i]$ and $F[n]$ and they are encoded as '0' and '1' respectively.

$\sum_{i=1}^{n-1} F[i]$ is formed by combining $\sum_{i=1}^{n-2} F[i]$ and $F[n-1]$. Thus, encoding of $\sum_{i=1}^{n-2} F[i]$ and $F[n-1]$ are '00' and '01' respectively.

After expanding all the merged symbols, the final encoding of symbols are:

$F[n] \rightarrow$ '1'

$F[n-1] \rightarrow$ '01'

$F[n-2] \rightarrow$ '001'

...

...

...

...

$F[3] \rightarrow$ '00....001' (n-3 zeros)

$F[2] \rightarrow$ '00....001' (n-2 zeros)

$F[1] \rightarrow$ '00....000' (n-1 zeros)

(b) Suppose you aim to compress a file with 16-bit characters such that the maximum character frequency is strictly less than twice the minimum character frequency. Prove that the compression obtained by Huffman encoding, in this case, is same as that of the ordinary fixed-length encoding.

Property P: maximum frequency $<$ 2*min frequency

Given: 16 bit characters with maximum frequency $<$ 2*min frequency

Claim: The Huffman encoding with n bit characters is reducible to n-1 bit characters by merging 2 characters and representing it as a single character using n-1 bits, following the same property i.e, maximum frequency $<$ 2*min frequency

Proof:
Let N = 2^n and F = [f_1, f_2 ....., f_N] be the frequencies sorted in ascending order.
Merging the least frequencies according to Huffman algorithm.

$\Rightarrow \tilde{f_1} = f_1 + f_2$
Since $f_N < 2 * f_1$

$\Rightarrow f_1 + f_2 \geq 2 * f_1 > f_N$

$\Rightarrow f_1 + f_2 > f_i \ \ \forall i = 1, 2....N$

Thus the new sorted frequency vector is $[f_3, f_4, ...f_N, \tilde{f_1}]$
Recursively merge the least two frequency characters to form a single character of the combined frequency, N/2 times. We get the new frequency vector as,

$\tilde{F} = [\tilde{f_1}, \tilde{f_2}, ...\tilde{f_{N/2}}]$

where $\tilde{f_i} = f_{i*2-1} + f_{i*2}$

max($\tilde{F}$) = $f_{N-1} + f_N$

min($\tilde{F}$) = $f_1 + f_2$

Since $f_N < 2 * f_1 \ and \ f_{N-1} \leq f_N < 2 * f_1 \leq 2 * f_2$

So, $f_N + f_{N-1} < 2f_1 + 2f_2$

$\Rightarrow$ max($\tilde{F}$) $<$ 2min($\tilde{F}$)

Thus, $\tilde{F}$ represents frequency vector of 2^{N-1} characters or n-1 bit characters with same property. □

Proof: Proof by Induction on number of bits
Base Case:

$n = 1$

Number of characters = 2^1

Huffman encoding is '0' and '1'

Huffman encoding is '0' and '1'

Induction Hypothesis:
Huffman encoding for n-1 bit characters which follow property P is same as fixed length encoding of length n-1.

Induction Step:
Using the claim proved above, n bit characters are merged to form n-1 bit characters with same property P. By induction hypothesis, these n-1 bit characters has fixed length encoding and forms a complete binary tree.
Expand the merged symbols in n-1 bit characters i.e, expand each leaf node present at height n-1 to get 2 children which is present at height h. These resulting nodes represent the n bit characters that were merged to form the n-1 bit Huffman tree and thus the resulting tree is the Huffman tree for n bit characters.
As the height of all leaf nodes in the resulting tree is n, it too forms a complete binary tree. Thus, they have fixed length encoding as all the characters require n bits to represent them.

Hence, By PMI, compression obtained by Huffman encoding of 16 bit characters is same as fixed length encoding of length 16. □

3 Graduation Party of Alice

(a) Alice wants to throw a graduation party and is deciding whom to call. She has n people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick a largest subset of n people, subject to two constraints: at the party, each person should have at least five other people whom they know and five other people whom they don't know. Present an efficient algorithm that takes as input the list of n people along with the list of pairs who know each other and outputs the best choice of party invitees. Give the running time in terms of n.

Algorithm:

Algorithm 2 Selection Algorithm

```
1:  procedure SELECT(list of people, list of pairs)        ▷ Returns the maximal set of people possible
2:      G → Undirected Graph produced using every person as a vertex
3:      n → number of vertices/people
4:      m → number of edges. An edge is added between two people if they know each other.
5:      change ← true
6:      current ← n
7:      D ← array of buckets where vertices are arranged by their degree
8:      while change = true do
9:          change ← false
10:         for v ∈ D with deg(v) < 5 and current > deg(v) > current − 5 do
11:             change ← true
12:             current ← current − 1
13:             REMOVEVERTEX(G, v)
14:     return the set of unmarked vertices

15: procedure REMOVEVERTEX(G, v)        ▷ Reduces degrees of vertices connected to v
16:     Mark v                                          ▷ v is removed from invitees list
17:     D[deg(v)].REMOVE(v)
18:     for j from 1 to deg(v) do
19:         u ← adj(v)(j)
20:         if u is marked then
21:             continue
22:         deg(u) ← deg(u) − 1
23:         D[deg(u) + 1].REMOVE(u)
24:         D[deg(u)].INSERT(u)
```

Runtime Analysis:
The graph is stored in adjacency list as a HashMap mapping each vertex to an unordered set of connected vertices. The time taken to make the graph is $O(m)$. Degree of all vertices is also stored separately as a HashMap to give $O(1)$ access to degrees of each vertex and updated accordingly.

The degree of all the vertices is then sorted using bucket sort with each bucket being an unordered set(Hash) of vertices which allows us to insert and remove in $O(1)$ time. The buckets themselves are stored as an array of buckets of size N, with the index of the bucket representing the degree of the vertices stored in them. The initial sorting takes $O(m)$ as it is done through a single pass.

The inner FOR loop inside the WHILE loop in the Select procedure, is also run only $O(n)$ times as each iteration, a vertex is removed, and the maximum no. of vertices that can be removed is n.

Since the number of edges are m so the procedure REMOVEVERTEX's FOR loop runs atmost $2 * m$ over all the calls as the procedure is called for each vertex only once, and the for loop is run degree(v) times. Sum of the degrees of all the vertices is $2 * m$. So the overall running time of the algorithm is $O(m + n)$. Since $m \leq n^2$ so the time complexity is $O(n^2)$.

Correctness of Algorithm:
In each iteration of the algorithm, vertices or people which can never be part of the optimal invitation list are removed as such people know less than 5 or such people don't know less than 5 people. At each iteration, we must remove at least one person, so this algorithm terminates.

When this algorithm terminates, by definition the subset of invitees is valid. Since we only remove people we have deduced could not possibly be invited, we always produce the largest possible number of invitees. □

(b) Suppose finally Alice invited n_0 out of her n friends to the party. Her next task is to set a minimum number of dinner tables for her friends under the constraint that each table has a capacity of ten people and the age difference between members of each dining group should be at most ten years. Present a greedy algorithm to solve this problem in $O(n_0)$ time assuming the age of each person is an integer in the range [10, 99].

Algorithm 3 Minimum tables Algorithm

```
1:  procedure MINIMUM(G)
2:      n_0 ← number of selected friends.
3:      Tables ← {}
4:      S ← SORT(G)                                   ▷ sorted in ascending order of Age
5:      while SIZE(S) ≠ 0 do
6:          start ← FRONT(S).Age
7:          T ← NULL
8:          T.INSERT(S.POP_FRONT)
9:          k ← 1
10:         while (SIZE(S) ≠ 0 and k < 10 and FRONT(S).age ≤ start + 10) do
11:             T.INSERT(S.POP_FRONT)
12:             k ← k + 1
13:         Tables.INSERT(T)
14:     return Tables
```

Runtime Analysis:
Sorting of people by age is possible in $O(n_0)$ time complexity using bucket sort as the range of age is fixed between 10 and 99. Every person present in the sorted list is only visited once because as soon as a vertex is visited, it is popped out of the list. Thus, the overall time complexity of the above algorithm is $O(n_0)$.

Correctness of Algorithm:
Claim: Let P be the set of people that have to be assigned tables. Let a_1 be the age of the youngest person. Let T be a table that seats the youngest person in P in optimal arrangement. Let T' be a table that seats the youngest person in P by applying the greedy algorithm. Then $size(T') \geq size(T)$ and $youngest((P \setminus T')) \geq youngest((P \setminus T))$

Proof:
Case 1: The no. of people with age a_1 is more than 10.
In this case, our greedy algorithm would form a table with all the 10 people of the same youngest age a_1. $Size(T') = 10$ which is the maximum allowed. $youngest((P \setminus T')) = a_1$ as there is more than 10 people with that age. No matter what arrangement T has, $youngest((P \setminus T)) = a_1$ as there is more than 10 people with that age. So, the claim is proved.

Case 2: The no. of people with age a_1 is less than or equal to 10.

Case 2.1: $T = T'$

Claim is proved.

Case 2.2: $T \neq T'$

Proof by contradiction

Assume $size(T') < size(T)$
$=>$ $\exists\, p_1$ present in T but not present in T' and size(T') < 10.
But because $p_1 \nexists\, T' => p_1 > a_1 + 10$
$=> p_1 \nexists\, T$ as a person with age a_1 is present in T.
Contradiction arises. So, Assumption is false.

Assume $youngest((P \setminus T')) < youngest((P \setminus T))$
All persons with age less than $youngest((P \setminus T))$ is present in T, but only persons with age less than $youngest((P \setminus T'))$ are present in T'.
$=> size(T') < size(T)$, due to the assumption, which is false as proved above.
Contradiction arises. So, Assumption is false.

Claim is proved. □

Proof of Correctness:

At each step of the algorithm, where a table is assigned for the youngest person of the remaining people, it is shown by the above claim that both size of the assigned table by the algorithm is equivalent to optimal arrangement and the youngest person left to be assigned also is equivalent.

As, the youngest person's age not assigned is greater than or equal to the optimal arrangement, this implies that the people not assigned a table at each step is minimised optimally in the greedy algorithm, as all the people below that age have been assigned a table. This shows that the greedy algorithm is working equivalent to an optimal algorithm, and in fact stays ahead of the optimal solution always. As this algorithm continues executing until all people are assigned a table, and at each step, at least 1 person is assigned a table, the algorithm terminates.

The correctness of the above greedy algorithm is thus proved. □