COL351 Assignment 1

Rahul Chhabra
2019CS11016

Shrey J. Patel
2019CS10400

September 5, 2021

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.

(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.

Property 1.1 (to be used without proof)
Let S be a spanning tree of G and some e = (x,y) be an edge not lying in S. Let e' ∈ S be an edge on the unique path P between x and y in S. Then $S' = S \backslash \{e'\} \cup \{e\}$ *is also a spanning tree.*

Solution of (a):

Let T = $\{e_1, e_2, ...e_{n-1}\}$ be the MST constructed by the Kruskal's algorithm. Here we assume the correctness of Kruskal's algorithm(Note that we will refer to this tree by T in the entire proof). Now we need to prove that if T' is another possible MST, then T = T'. Note that the edges are in increasing order of weights i.e. $e_1 < e_2 < .... < e_{n-1}$ since the weights are distinct. We will prove this by **induction on the edges of T**, using the following induction hypothesis:

H(i): If T' is some other possible MST, then it must include the first i edges of T or in other words, $\{e_1, e_2, ...e_i\} \subseteq T'$. (Here, $1 \leq i \leq (n-1)$)

- **Base Case(i = 1):**
  From Kruskal's algorithm, e_1 is the edge with the smallest weight in the whole graph. Let T' be any MST distinct from T. There can be two cases:

  Case 1: $e_1 \in T'$. We are done.

  Case 2: $e_1 \notin T'$. We will prove this by contradiction. We assume that T' is an MST which doesn't contain the edge e_1. From property 1.1, if T' is spanning then replacing some edge $e \in T'$ with e_1 should result in a spanning tree, say T". Also, from our assumption, T' is an MST, but because we replace an edge in T' with the smallest edge e_1, $wt(T'') \leq wt(T')$. Thus, T" is an MST. We have a contradiction since we assumed that T' is an MST. So, edge e_1 must be present in any MST.

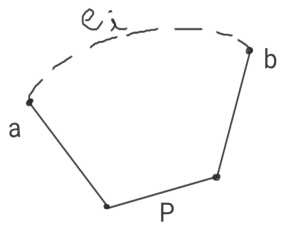
  In any case, $e_1 \in$ any MST. **Case1** $\wedge$ **Case2** $\implies$ **H(1)**.

- **Induction Step:**

  We need to prove that H(i-1) $\implies$ H(i) $\forall\ 1 < i < n$. H(i-1) holds, so for any MST T' (even if it different from T), $\{e_1, e_2, ...e_{i-1}\} \subset T'$. Now we prove that e_i also belongs to any MST.

  Case 1: $e_i \in T'$. We are done.

  Case 2: $e_i \notin T'$. We prove this by contradiction i.e. we assume that T' is an MST which doesn't contain the edge e_i. If e_i = (a,b), then $\exists$ a unique path P in T' joining a and b. From Kruskal's algorithm, we are ensured that the set of edges $\{e_1, e_2, ...e_{i-1}, e_i\}$ does not form a cycle. But,
  P $\cup\{e_i\}$ is a cycle as both P and e_i join a and b and $e_i \notin P$.



  Therefore, $P \nsubseteq \{e_1, e_2, ...e_{i-1}\}$, or in other words $\exists e' \in P$ such that $e' \notin \{e_1, e_2, ...e_{i-1}\}$. In Kruskal's algorithm, the greedy strategy finds the edge (in this case e_i) of minimum weight which doesn't from a cycle with the already constructed set of edges $\{e_1, e_2, ...e_{i-1}\}$. So, $wt(e_i) < wt(e')\ \forall\ e'$ which don't form a cycle with $\{e_1, e_2, ...e_{i-1}\}$.

  Additionally, from property 1.1, if T' is an MST, if we replace such edge e'(in path P), with e_i to obtain T", then T" is a spanning tree. And as argued above, $wt(T'') < wt(T')$, so T" is also an MST. We have a contradiction since we assumed that T' is an MST. So, e_i must be present in any MST.

  In any case, $e_i \in$ any MST, and knowing that $\{e_1, e_2, ...e_{i-1}\} \subset$ any MST, we have $\{e_1, e_2, ...e_i\} \subset$ any MST. **Case1** $\wedge$ **Case2** $\wedge$ **H(i-1)** $\implies$ **H(i)**.

Thus, H(i) holds $\forall\ 1 \leq i < n \implies$ H(n-1) holds. So, if T' is any MST, then $\{e_1, e_2, ...e_{n-1}\} \subset T' \implies$ T=T'. **So, every graph G has a unique MST.**

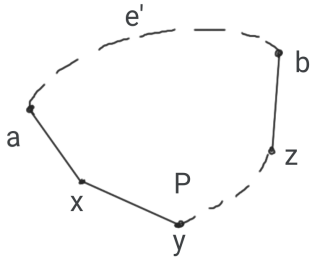
Solution of (b):

Strategy:

- Normally, any MST can be obtained by construction from the edges of the given graph. But here, because the number of edges is at most, we will **remove edges(at most 9)** to obtain a tree.
- We require a spanning tree, so the graph should remain connected throughout the edge removal process. So, the **removed edge must not be a bridge edge**.

- So, we need to remove non-bridge edges. The existence of such edges is trivially true. Because otherwise, G would be a tree, which isn't. But there can be multiple non-bridge edges, then which one to choose to obtain an MST?

  Heuristic for choosing optimal non-bridge edge:
  Consider a non-bridge edge e' = (a,b). By definition of a non-bridge edge, vertices a and b are connected in G \ e'. Or in other words, ∃ P connecting a and b such that e' ∉ P. So, e' ∪ P forms a cycle.

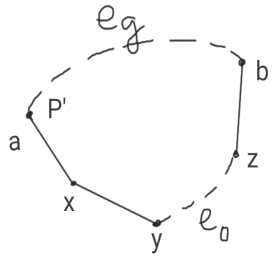


In fact, the same argument holds for all edges of P as well i.e. e is a non-bridge edge ∀ e ∈ {e'} ∪ P. Also, a tree contains no cycles, so it is essential to remove at least one edge from this cycle. But which edge of the cycle is the best to remove first?

Claim: It is always optimal to **remove the non-bridge edge with the greatest weight from any cycle**.

Proof of Claim:
Let e_g be the edge with the greatest weight from a cycle, say C. We prove that removing e_g from C is the optimal move. First of all, it is safe to remove e_g since it is a non-bridge edge.

We will prove this by contradiction. So, we assume that it is not optimal to remove the greatest weight edge from any cycle of the graph. Let's say that for some cycle, it is optimal to remove an edge, say e_0, such that e_0 is not the greatest weight edge for that cycle. Let's assume that T is the final MST.



We have already removed the edge e_0, so $e_0 \notin$ T. From the figure, e_0 = (y,z). T is an MST. So, ∃ a unique path P' joining y and z in T, such that $e_0 \notin P$.

Now, $e_0 \cup P$ is a cycle in the original graph. And by our assumption e_0 is not the greatest weighted edge. Say, the greatest weighted edge for that cycle be e_g. Then, $e_g \in P$, and from property 1.1, we can replace the edge e_g with e_0 to obtain another spanning tree, say T'. Also note that since wt(e_0) < wt(e_g), wt(T') < wt(T). We have a contradiction, since we assumed that T is an MST. So, we see that it is **optimal to remove the edge of greatest weight for every cycle**.

From this, we have derived a greedy strategy to obtain an MST by removal of edges.

Algorithm:

1. First find a cycle as a sequence of edges in the graph. This can be done by traversing the graph using DFS and parent pointers.
2. Then find the edge with the greatest weight from this sequence of edegs and remove it.
3. Repeat the first two steps until the number of edges are equal to *n-1*.

Algorithm 1 Pseudocode for finding MST by removal of edges

```
procedure MST(G)
  numEdges ← m
  while numEdges >= n do
    Choose some vertex, say v
    parent ← an array of size n                ▷ Parent array to store the previous vertex during DFS
    parent[v] ← v
    u ← DFS(v,parent)
    w ← parent[u]
    e ← edge(u,w)
    while w != u do
      if wt(e) < wt(edge(w,parent[w])) then
        e ← edge(w,parent[w]
      end if
      w ← parent[w]
    end while
    remove e from G
    numEdges ← numEdges-1
  end while
procedure DFS(x , parent)          ▷ Returns a vertex that is a part of a cycle, null otherwise
  for all neighbours y of x do
    if parent[y] = null then
      parent[y] ← x
      if DFS(y,parent) != null then
        return DFS(y,parent)
      end if
    else
      parent[y] ← x
      return x
    end if
  end for
  return null
```

Time Complexity Analysis:

Note that here $m \leq (n + 8)$ and so m = O(n). Finding cycle through DFS takes O(m+n) = O(n) time. Finding the greatest weight edge from this sequence of edges takes O(n) time i.e. the maximum length of a cycle. Since number of edges are at most n+8, number of iterations ≤ 9. **Total time complexity = O(n)**

2. Huffman Encoding

(a) What is the optimal binary Huffman encoding for n letters whose frequencies are the first n Fibonacci numbers? (For example: The frequency vector F is [1, 1, 2, 3, 5, 8, 13, 21] for n = 8). What will be the encoding of the two letters with frequency 1, in the optimal binary Huffman encoding?

(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.

Solution for (a):

The frequency vector F = {1,1,2,3,5...} i.e. f_n = fib(n). Here we define fib(n) as:

$$\textit{fib(x)} = \begin{cases} 1 & x = 1, 2 \\ fib(x-1) + fib(x-2) & otherwise \end{cases}$$

We use prefix encoding scheme to obtain an optimal encoding for this frequency vector. We first prove a useful claim regarding the fibonacci numbers:

Claim 2.1: $\sum_{k=1}^{i} fib(k) + 1 = \textit{fib(i+2)} \; \forall \; i \geq 1$

Proof of Claim 2.1:
Proof by induction on i.

- **Basis:** For i=1, LHS = fib(1) + 1 = 2. And, RHS = fib(3) = 2. So, LHS=RHS. So, the hypothesis holds for i=1.
- **Induction Step:** We assume that the hypothesis holds for some positive integer i, i.e. $\sum_{k=1}^{i} fib(k)$ + 1 = *fib(i+2)*. Now we prove the same for i+1.

  $\sum_{k=1}^{i} fib(k) + 1 = \textit{fib(i+2)} \implies$
  $\sum_{k=1}^{i} fib(k) + 1 + \textit{fib(i+1)} = \textit{fib(i+2)} + \textit{fib(i+1)} \implies$
  $\sum_{k=1}^{i+1} fib(k) + 1 = \textit{fib(i+3)}$

  Thus, the hypothesis also holds for i+1.

Therefore, from the above claim, $\sum_{k=1}^{i} fib(k) < \textit{fib(i+2)}$. During Huffman encoding, we consider the frequencies which are at the same depth in the encoding tree, which in the case of prefix encoding corresponds to the two lowest frequencies. The initial ordering on frequencies if $f_1 \leq f_2 < f_3 < f_4 .... < f_n$ since f_i = *fib(i)*. For constructing the encoding tree however, we require the lowest two frequencies at every iteration of the Huffman theorem. Consider the following claim:

Claim 2.2: After **i** iterations of Huffman theorem, the two lowest frequencies are *fib(i+2)* and $\sum_{k=1}^{i+1} fib(k)$

Proof of Claim 2.2:
Proof by induction on i.

- **Basis:** After i=0 iterations, i.e. initially, fib(2) and fib(1) are the lowest frequencies trivially.

- **Induction Step:** Let's assume that the hypothesis holds after i iterations i.e. *fib(i+2)* and $\sum_{k=1}^{i+1} fib(k)$ are the two smallest frequencies. Thus, in the next iteration i.e. the $(i+1)^{th}$ iteration, these two frequencies will be added to decrease the depth of the tree by 1.

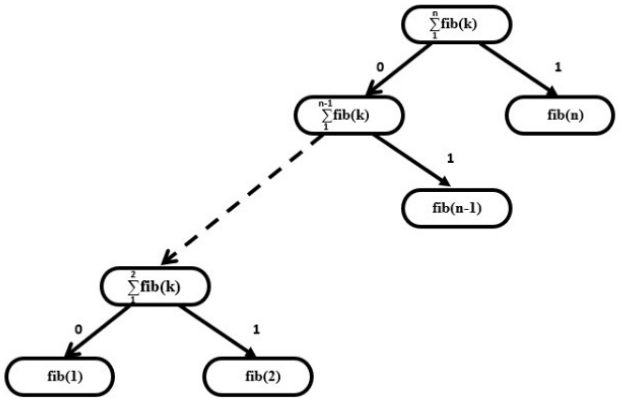
  f = *fib(i+2)* + $\sum_{k=1}^{i+1} fib(k) = \sum_{k=1}^{i+2} fib(k)$

  Note that from claim 2.1, $f <$ *fib(i+4)*, and from the initial ordering *fib(i+3)* < *fib(i+4)*. So, both f and *fib(i+3)* are smaller than *fib(i+4)* and therefore constitute the two smallest frequencies.

  In other words, after (i+1) iterations, *fib((i+1)+2)* and $\sum_{k=1}^{(i+1)+1} fib(k)$ are the lowest two frequencies. So, the hypothesis also holds for i+1.

From claim 2.2, we can say that the **summation keeps on accumulating on increasing depth**. Additionally, the term fib(n) is added to the summation the last because of highest value and therefore it should also be the one with the least depth i.e. 1. Similarly the frequency fib(n-1) should be at one depth higher and so on. However, the depth of the last two frequencies i.e. fib(1) and fib(2) should be same according to the property of the encoding tree. Th depth relation can be encoded using the following function:

$$depth(x) = \begin{cases} n-1 & x = 1, 2 \\ depth(x-1) - 1 & otherwise \end{cases}$$



Final Encoding:

fib(n) → "1"
fib(n-1) → "01"
fib(n-2) → "001", and so on

In general, fib(x) → depth(x-1) zeroes followed by a one.

So,
fib(1) → "0000....1"
fib(2) → "0000....1"
n-2 zeroes followed by a 1, where n is the length of frequency vector.

Solution for (b):

Let the frequency vector be F = $\{f_1, f_2, f_3, ..., f_n\}$ in non-decreasing order of frequencies. It is given that $f_n < 2 * f_1$. The encoding is 16-bit, so the total number of characters, n = 2^{16}. We will follow the prefix encoding scheme and the Huffman's theorem, i.e. the two lowest frequencies will be removed from F and their sum will be added to F, thus increasing the depth of these frequencies by 1.

Claim 2.3:
Let the F vector be represented as a **queue** which is sorted in non-decreasing order. We will remove the two lowest frequencies from the front of the queue and add their sum to the end of the queue. Then F remains sorted, and if f is the lowest frequency and f' is the highest frequency, then *f' < 2*f.*

Proof of Claim 2.3:
We prove this by induction on the number of iterations(i) of the above procedure.

- **Base Case:** When i=0(initially), F is trivially sorted by assumption and it is given that 2 * lowest frequency(f_1) > highest frequency(f_n). So, the base case holds.
- **Induction Step:** Let's say that after i iterations, F = $\{f_1, f_2, f_3, ..., f_n\}$. We assume that the hypothesis holds for this F i.e. F is sorted in non-decreasing order and $f_n < 2 * f_1$. In the next iteration i.e. (i+1), we will remove the lowest two frequencies, which are f_1 and f_2 (since F is sorted) and add their sum ($f_1 + f_2$) at the end of F to obtain F'. We need to prove two things:

  1. **F' is sorted:** It is sufficient to show that the sum is larger than the highest frequency to prove that F' is sorted i.e. $f_n < f_1 + f_2$.
     $f_1 < f_2 \implies 2 * f_1 < f_1 + f_2 \implies f_n < f_1 + f_2$
     So, F' is sorted.
  2. **Highest Frequency in F' < 2 * Lowest Frequency in F'**
     Lowest Frequency = f_3, Highest Frequency = $f_2 + f_1$
     $(f_1 < f_3) \wedge (f_2 < f_3) \implies f_1 + f_2 < 2 * f_3$
     So, the claim holds.

  Thus, F' also follows the induction hypothesis.

We need to prove that the fixed length encoding is optimal in this case, which is same as proving that the all leaves are at the same level of the optimal binary tree. From claim 2.3, at any stage of the algorithm, the two lowest frequencies add up to obtain a frequency which is greater than any other frequency. Note that, the **frequency vector at any point contains the frequencies of the leaves of the current binary tree**.

Example: Initially, F = $\{f_1, f_2, f_3, ..., f_n\}$, where f_i $\forall$ i are leaves of the encoding tree. Then, f_1 and f_2 add up to form $f_{1,2}$ which is the leaf of the smaller tree, and simultaneously F' = $\{f_3, f_4, f_5, ..., f_n, f_{1,2}\}$ and so on.

Apart from this, we also use the property of the encoding tree that if the frequency order is $f_1 \leq f_2 \leq f_3 ... f_n$ then depth(f_1) $\geq$ depth(f_2) $\geq$ ....depth(f_n).

Claim 2.4:
If at any point of the algorithm, F contains the leaves f_1, f_2,.... f_n, then the maximum difference in the height of these leaves is 1, i.e. depth(f_1) - depth(f_n) $\leq$ 1.

Proof of Claim 2.4:
We prove this by induction on the depth of the current binary tree.

- **Base Case:** When depth = 0, F is a singleton set, which means the corresponding tree only contains the root node whose depth = 0. So, it trivially follows the hypothesis.
- **Induction Step:** Assuming that hypothesis holds for the binary tree of depth i. Assume that F = $\{f_1, f_2, f_3, ..., f_n\}$ for this binary tree. Then f_1 is at the highest depth, being of lowest freq., so depth(f_1) = i.

  For next step, we split the leaf with highest frequency(i.e. f_n) to obtain two new leaves, say f and f', such that $f + f' = f_n$. Also, depth(f) = depth(f') = depth(f_n) + 1. From induction hypothesis, we have that depth(f_1) $\geq$ depth(f_n) $\geq$ depth(f_1)-1.

  From claim 2.3, f and f' are the new lowest frequencies and f_{n-1}, so it is sufficient to prove that depth(f) - depth(f_{n-1}) $\leq$ 1.

  Case 1:
  depth(f_n) = depth(f_1) = i $\implies$
  depth(f_{n-1}) = i since f_n has the lowest depth
  and depth(f) = depth(f_n) + 1 = i + 1 $\implies$
  depth(f) - depth(f_{n-1}) = 1 $\leq$ 1.

  Case 2:
  depth(f_n) = depth(f_1) - 1 = i - 1 $\implies$
  (depth(f_n) = i-1) $\leq$ depth(f_{n-1}) $\leq$ (depth(f_1) = i) $\implies$
  and depth(f) = depth(f_n) + 1 = i $\implies$
  depth(f) - depth(f_{n-1}) $\leq$ 1.

  The claim also holds for the next step. So, the hypothesis holds for all depths of the binary encoding tree.

From claim 2.4, we proved that **all the leaves of the optimal binary encoding tree in this case lie at the depth difference of at most 1**. If given number of characters = n, then from this claim,
$2^{k-1} < n \leq 2^k$ (property of binary tree), where k is the depth of the tree. If n is a power of 2 (= 2^k), then we can prove that all the leaves are at the same level, which we need to prove.

We prove this by contradiction. Assume that,
Number of leaves at depth k-1 = x $\implies$
Number of internal nodes at depth k-1 = 2^{k-1} - x $\implies$
Number of leaves at depth k = 2 * (2^{k-1} - x) = 2^k - 2x $\implies$
Total number of leaves = 2^k - x = 2^k(given) $\implies$ x = 0

We have a contradiction. Thus, there are no leaves at depth = k - 1. Thus, all leaves are at the same depth(=16 in this case) in the optimal binary encoding tree. So, the fixed length encoding is the optimal encoding. Hence Proved.

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

(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].

Solution for (a):

Let's formulate the problem in terms of a graph. Let G = (V,E) where V denotes the set of n people that Alice can choose to call from. E is the set of edges where an edge (u,v) $\in$ E if and only if u and v know each other. For each vertex v_i, the degree of vertex denotes the number of people v_i knows.

Algorithm: We will use a greedy strategy to reduce the problem size by removing those vertices which are definitely not part of the optimal solution. In particular, we will remove all those vertices v_i from V which fulfill one of the below criteria:

1. deg(v_i) < 5 i.e. *Such vertices represent people who know less than 5 people among the current potential candidates.*
2. |V| - deg(v_i) - 1 < 5 i.e. *Such vertices represent people for whom the number of unknowns are less than 5, where |V| represents the number of potential candidates.*

The algorithm will terminate when there is no such vertex to be removed.

Claim: The subgraph G' of G obtained as a result of the removal of all such vertices will be the optimal graph i.e. the set S returned by above algorithm is optimal.

Proof of claim:
Let S_i denote the state of set S after i vertices have been removed by the above algorithm. Let S' be any compatible set of vertices required by Alice. We will show that H(i) = S' $\subseteq S_i$ holds $\forall i >= 0$

Base Case: i = 0 $\implies$ No element has been removed from the set of vertices, i.e. S_0 = V(G) and S' is a subset of vertices of given graph i.e. $S' \subseteq V(G) \implies S' \subseteq S_0$

Induction Hypothesis: Let H(i-1) holds i.e. any compatible set of vertices S' $\subseteq S_{i-1}$ for some i>1

Induction Step: Let v_i be the vertex that is removed at i^th step of the execution of our algorithm
$\implies S_i = S_{i-1}/\{v_i\}$
By our Algorithm the edge v_i is removed if and only if :

Algorithm 2 Pseudocode to obtain the maximally optimal set of people S

```
procedure getBestChoice(Graph G, int n)
    isPresent ← a boolean array of size n with all entries initially true
    compatible ←[{degree_vi,v_i} : ∀v_i ∈ V ]        ▷ A list of pairs where each pair is of form {degree_vi,v_i}
    num ← n
    i ← getIdx(compatible, isPresent, num)
    S ← {v_i : v_i ∈ V(G)}
    while i ≠ −1 do
        S.remove(compatible[i].second)
        k ← 1
        while k<=n do
            if isPresent[k] and {compatible[i].second, compatible[k].second} ∈ G.Edges then
                compatible[k].first ← compatible[k].first − 1
            end if
            k ← k + 1
        end while
        num ← num-1
        i ← getIdx(compatible, isPresent, num)
    end while
    return S
procedure getIdx(compatible, isPresent, num)
    i ← 1
    while i ≤ compatible.size() do
        if isPresent[i] then
            if compatible[i].first < 5 then
                isPresent[i] = false
                return i
            else if num - compatible[i].first - 1 < 5 then
                isPresent[i] = false
                return i
            end if
        end if
        i ← i + 1
    end while
    return -1
```

1. $\deg(v_i) < 5$ i.e. the person denoted by v_i represents a person who knows less than 5 people
2. |V| - $\deg(v_i)$ - 1 < 5 i.e. there are less than 5 people whom the person denoted by v_i don't know .

Since S' is a compatible set, all the vertices in S' should obey the two conditions Alice states but v_i doesn't obey the above conditions $\implies$ v_i cannot belong to S'
$\implies$ $v_i \notin S'$
$\implies$ $S' \subseteq S_{i-1} \backslash \{v_i\}$
$\implies$ $S' \subseteq S_i$
H(i-1) $\implies$ H(i) so by PMI, H(i) holds $\forall i >= 0$

This proves that any compatible set of vertices will be a subset of the set S returned by our algorithm , so set S is the maximal compatible set.
Hence Proved.

Runtime Analysis: In every iteration of the algorithm, we find a vertex which violates either of the two conditions as mentioned above (i.e. degree < 5 or number of non-neighbours < 5), and remove that vertex and the edges incident on that vertex as that vertex cannot be a part of the optimal solution. After removing one such vertex, we need to update the degree of the rest of the vertices. The total time taken for each of these is:

- To find such a vertex, we may need to check the degrees of at most n vertices = *O(n)*.
- Number of edges to be removed = degree of that vertex = *O(n)*.
- Number of degree updates = Number of edges removed = *O(n)*.

So, total time per iteration is *O(n)*. And in the worst case, we might need to remove all vertices. So, total time of the algorithm = O(n) * O(n) = $\mathbf{O}(n^2)$.

Solution for (b):

Algorithm:
We will maintain an array of size 100 where arr[i] will contain the count of number of Alice' friends having age equal to i. We will start from 10 which is the minimum possible age of Alice' friends and iterate over the array till the end. While traversing the array we will maintain the count of number of people on the current table, the minimum age of person on current table and number of tables taken till now . We will introduce a new table if and only if one of the following conditions is satisfied:

1. Number of people on current table exceeds 10
2. The difference between age of current person and the person with minimum age exceeds 10

Let any table be represented by $\{minAge_i$, $maxAge_i\}$ where $minAge_i$ represents the age of youngest person on table i and $maxAge_i$ represents the age of oldest person in table i. A table is compatible only if number of people on table i<=10 and $maxAge_i - minAge_i <= 10$. We call two tables i and j disjoint if $minAge_i \leq maxAge_i \leq minAge_j \leq maxAge_j$ and since these tables are compatible $\implies$ $maxAge_k - minAge_k <= 10$ $\forall$ k and $numPeople_k <= 10$ for $k \in \{i, j\}$

To prove that this algorithm indeed produces an optimal distribution of tables , we will prove that any optimal distribution D' can be reordered to get the distribution D returned by our algorithm and during

Algorithm 3 Pseudocode for above algorithm

```
procedure minimumTables(guestAges)
    countArray ← generateCountArray(guestAges)
    countTables ← 0
    i ← 1
    j ← 1
    curCount ← 0
    numTables ← 0
    while j <= 100 do
        if j-i>10 then
            numTables ← numTables + 1
            curCount ← 0
            i ← j
        else if curCount + countArray[j] > 10 then
            countArray[j] ← curCount + countArray[j] - 10
            numTables ← numTables + 1
            curCount ← 0
            i ← j
        else
            curCount ← curCount + countArray[j]
            j ← j +1
        end if
    end while
    return numTables
procedure generateCountArray(guestAges)
    countArray ← new int[100]                    ▷ Empty array of size 100 with all entries 0
    i ← 1                                        ▷ Assuming 1 based indexing
    while i <= guestAges.size() do
        countArray[guestAges[i]] ← countArray[guestAges[i]] + 1
        i ← i +1
    end while
```

reordering the number of tables do not increase $\implies$ |D| $\leq$ |D'|

Claim 3.1: People on any two non-disjoint tables can be reordered to get at most two disjoint tables.

Proof of claim:
Let i and j be two non disjoint tables. Without loss of generality let's assume that $minAge_i \leq minAge_j$. Since the two tables are non-disjoint $\implies$ $maxAge_i > minAge_j$. We can keep swapping the youngest person on table j with oldest person on table i until $maxAge_i <= minAge_j$.

Claim 3.1.1: The configuration of the two tables is compatible after a finite number of above operations.

Since for both of the tables at each step we are removing one person and adding one person, the number of persons on any table doesn't change. So, the assumption that their sizes are at most 10 still holds.

$(maxAge_i - minAge_i \leq 10) \wedge (maxAge_i > minAge_j) \implies$
$maxAge_i - minAge_i > minAge_j - minAge_i \implies$
$minAge_j - minAge_i \leq 10$
So, when the youngest person (i.e. $minAge_j$) of table j is transferred to table i, table i is still compatible.

$(maxAge_j - minAge_j \leq 10) \wedge (maxAge_i > minAge_j) \implies$
$maxAge_j - minAge_j > maxAge_j - maxAge_i \implies$
$maxAge_j - maxAge_i \leq 10$
So, when the oldest person (i.e. $maxAge_i$) of table i is transferred to table j, table j is still compatible.

Thus, both tables remain compatible after any number of above operations until both the tables remain intersecting.

Note that if numPeople(i) + numPeople(j) <= 10 and $maxAge_j - minAge_i <= 10$ we can merge these two tables to get a single compatible table, in which case claim 3.1 still holds.

Hence, claim 3.1 holds.

Claim 3.2: Any non-disjoint distribution of tables D' can be converted to a disjoint distribution of tables D such that |D| <= |D'|.

Proof of Claim 3.2
We can do this conversion by taking any two non-disjoint tables and rearranging people on them to make these tables disjoint, according to Claim 3.1. Since for k tables we can have atmost ${}^kC_2$ possible pairs, we can convert D to D' in finite number of iterations and at each reshuffling the number of tables either remains same or decreases by one, so number of table(D) <= number of table(D'). So, |D| <= |D'|.

Claim 3.3: Let D be the distribution returned by our algorithm and $\tilde{D}$ be any optimal disjoint distribution obtained by claim 3.2, then |D| <= $|\tilde{D}|$

Proof of Claim 3.3
Let n = |D| and m = $|\tilde{D}|$. Further let $a_1, a_2, a_3, ..., a_n$ be the number of people in distribution D in

which each tables are sorted according to age of youngest person on that table and let $b_1, b_2, b_3, ..., b_m$ be the corresponding order of number of people in distribution $\tilde{D}$ sorted according to age of youngest person.

Let k be the minimum index such that a_k != b_k then there can be two cases:

- **Case 1:** $a_k < b_k$
  Let i_1, i_2..... be the ages in sorted order on table k in Distribution D and j_1, j_2..... be the ages in sorted order on table k in distribution $\tilde{D}$. Since upto k-1 tables we had a similar distribution in D and $\tilde{D}$ and the above orders are sorted, so $\forall$ l <= a_k, $i_l = j_l$. But, if j_{a_k+1} - j_1 <= 10 then it would have been included in table k by our algorithm but because the size of table is a_k, j_{l+1} - j_1 > 10 which contradicts the fact that $\tilde{D}$ is a compatible distribution. So, this case is not possible.

- **Case 2:** $a_k > b_k$
  Since a_k <= 10 and $b_k < a_k \implies b_k < 10$, so we can move the youngest person from table k+1 to table k until we get $a_k = b_k$ since our algorithm has selected a_k persons carefully considering the constraints given by Alice, so the youngest person on $(k+1)^{th}$ table in $\tilde{D}$ distribution should obey those constraints. In this process, either the number of tables remain same or if the $(k+1)^{th}$ table becomes empty after the removal of youngest person at some iteration, the number of tables decrease by 1.

By repeatedly applying above transformation we can convert $\tilde{D}$ to D such that |D| $\leq |\tilde{D}|$.

Hence, D is the most optimal disjoint set distribution as proved by the claim 3.3.So any optimal distribution of tables can be converted into a disjoint distribution obtained by our algorithm without increasing the number of tables . So the distribution returned by our algorithm is optimal

Running Time Analysis: In our Algorithm, we first generate a count array from the given list of ages of Alice' guest. To generate the count array we have to to iterate the guest array fromt start to end so this will take O(n_0) times since Alice has n_0 number of guests. Now we have to iterate the count array ewhich is of size 100, so iterating the count array takes constant time.
$\implies$ T(n) = O(n_0) + C
$\implies$ Time Complexity of above algorithm: **O(n_0)**