Question 3

Geneology of blood relations

December 2016

Introduction
A geneology is a directed graph connecting parents to their children. Assume strict exogamy operates and that none of the spouses are part of the family tree. Our geneology only tracks the those related by blood, and ditches the spouses. So if A is the son of B, A and B are in the geneology, but As mother is skipped. It is also assumed that people in the geneology never have children with each other, but only with others outside the geneology. So each node in our geneology can trace its descent form the original ancestor vai a unique path.

Q1. Define a data structure to store the information in the geneology. People are nodes in this graph, while edges are parent child relationships. The value of the edge is the age of the parent when child was born. Nodes contain age of the individual when he/she died (and an Id and their names).

Ans.

```
typedef struct //node
  {
    char name[15];
    char death_age[3];
  }Person;

  Person P[N];  //stores required information about each
      individual

  int c[N][no_of_children_max],edge[N][no_of_children_max];
       /*c[i][j] stores the index of  jth child and edge
      stores the age of i when jth child was born*/
```

Here, the ID is assumed to be the array index of P[]. The nodes are read from the input file and stored in the array P[]. Later, the parent child relations are read and are accordingly filled into C[][] , p[][] and edge[][]. This has been done while reading the input file.

Q2. Will a binary tree be the appropriate structure?

Ans. Binary tree cannot be used since a person might have more than two children. Hence we wont be able to accomodate all the data in a binary tree in an efficient way. Even if we do manage to do so, it would take up a lot of memory space while making replica of parents having more than 2 children and traversal through the tree would be difficult too. Hence a binary tree is not preferred for the above geneology unless a condition is stated that no parent can have more than 2 children.

Q3. What about a 2 3 tree?

Ans. A 2-3 tree wont work either because of similar reasons as that of the binary tree case. It wont hold if an individual in the geneology has more than 3 children. Hence a 2-3 tree is not preferred for the above geneology unless a condition is stated that no parent can have more than 3 children.

Q4. Or should it be a general tree or graph?

Ans. I have considered a general graph for my algorithm. It has been stated that : people in the geneology never have children with each other, but only with others outside the geneology. So each node in our geneology can trace its descent form the original ancestor vai a unique path. Hence, a unique path can be traced from a node to its descendant . Hence, a node might have more than 1 ancestor as it could be a part of two different trees originating from two different ancestors. That is, consider a node born of two parents belonging to two different geneology, then the node will become part of 2 geneologies. That node might marry into another geneology and hence have children that have 3 ancestors now and it may go on. There will also be trees from ancestors that are disconnected from from the graph under consideration.

I have taken such a situation under consideration and made the algorithms. And hence, it is more of a general algorithm for family trees that are not

necessarily trees and hence I have taken it as a graph, that maybe connected or disconnected. In the example input given, Sarah stands out as she has neither descendants nor ancestors .Hence only one tree exists starting from James.The formulated algorithms are for general cases and will work in case of multiple geneologies.
Ex :

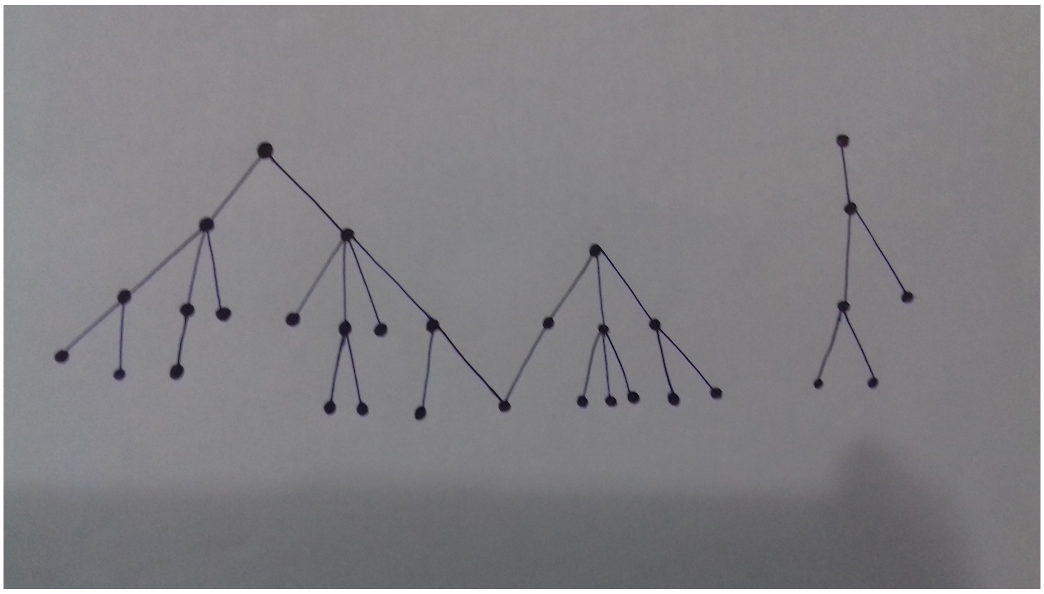

Figure 1: Here is an example

I have written codes for both DP as well as recursive algorithms and hence the two program files. The DP method is much more preferred as it has much lesser time complexity, though recursion has lesser space complexity. I have explained the DP approach in the below explanations. The recursive algorithm has been attatched in the zip folder (well commented) along with a brief set of answers.

Q5. Also develop the algorithm that will read the input file and store the graph with its relationships.

Ans.

```
for(i=0;i<N;i++)  //reading in the name and death age of
    each individual from the file
```

```
{
  fscanf(fin,"%s",P[i].name);
  fscanf(fin,"%s",P[i].death_age);
  p[i][0]=-1;   //initialising paret indices as -1
  p[i][1]=-1;
  cno[i]=0; //initialise no. of children as 0
}

while(!feof(fin)) //reading in the relations
{
  fscanf(fin,"%s",str1);  //read in parent name
  fscanf(fin,"%s",str2);  //read in child name
  fscanf(fin,"%d",&E);  //read in edge value
  for(i=0;i<N;i++)
    if(strcmp(str1,P[i].name)==0) //find index of parent
      index_p=i;
    else if(strcmp(str2,P[i].name)==0)  //find index of
       child
      index_c=i;
  c[index_p][cno[index_p]]=index_c; //storing the index of
     the child for the parent
  edge[index_p][cno[index_p]]=E;    //storing the edge
     value to the child from the parent
  (cno[index_p])++;     //increment the no. of children
  if(p[index_c][0]==-1) //if no parent till now
  {
    p[index_c][0] = index_p;  //parent index stored
  }
  else        //if one parent was already found before
    p[index_c][1] = index_p;
}
```

Here, the parents information are also stored as it helps in finding out the ancestors in O(n), where n is the no. of nodes. This is necessary as the rest of the code depends on this factor. The whole algorithm has its base once the ancestors are accurately found.

Q6. Determine the original ancestor.

Ans. Original ancestors are identified by the fact that they dont have any parents. Note that though there are multiple ancestors, their heirarchial levels may not be same.

```
void find_ancestor()     //3a answer. To find the ancestors
{
  int i,j=0;
  printf("\nThe ancestors are : ");
  for(i=0;i<N;i++)
  {
    if(i<10)
      lev[i]=-1;  //initialise lev[] to -1

    if(p[i][0]==-1 && p[i][1]==-1)  //if they have no
        parents at all, then they are ancestors.
    {
      printf("\n\t%s",P[i].name);   //print the name of the
          ancestors
      if(cno[i]==0)          //if isolated node
        printf("  (but no descendants)");

      lev[j]=i; //store the indices of ancestors in lev[]
      j++;  //increment j as one more ancestor index has
          been stored

    }
  }
}
```

Assigning the heirarchial levels for the nodes

Before computing the number of descendants, since we are using a DP approach, it is necessary to know the heirarchial level of each node.The function used for the purpose is :

```
int max_level()
{
  ....
}
```

dp[N][2] : dp[i][j] stores the heirarchial level of node i at j=1 and no. of descendants at j=0.
Hence, the code used for the purpose has been given below :

```
void find2(int pos, int cnt)    //recursive function for
    finding the lowest descendant count
{
  int i;
  if(cno[pos]!=0)   // if the node at pos has children
  {
    cnt++;    //level counted
    for(i=0;i<cno[pos];i++)
    {
      find2(c[pos][i],cnt); //recursive function call to
          calculate the lowest descendents level
      if(cnt>max)
        max=cnt;  //to get the maximum depth of a descendant
            from the oldest ancestor inorder to decide the
            heirarchial levels
    }
  }
}
void assign(int pos,int cnt)  //recursive function to assign
    levels to the descendant nodes of node at pos
{
  int i;

  if(cno[pos]!=0)   // if the node at pos has children
  {
    cnt--;    //level counted
    for(i=0;i<cno[pos];i++)
    {
      dp[c[pos][i]][1]=cnt; //assign level to the child node
          (oldest ancestor at the heighest level)
      assign(c[pos][i],cnt);  //recursive function call to
          assign levels to the children nodes
    }
  }

}

void assign_p(int pos,int cnt)  //function to assign levels
    to parent nodes that have children with nodes that are
    part of the biggest tree : connected trees and hence
    referred to as graphs
{
  int i;
  cnt++;  //increment level as compared to the child

  if(p[pos][0]!=-1 && dp[p[pos][0]][1]==0 ) //for parent 1,
      if the level has not been yet assigned as it wasn't
      part of the largest tree
  {
    dp[p[pos][0]][1]=cnt; //assign level to the parent node
    assign_p(p[pos][0],cnt);  //recursive function call to
       assign levels to further parent nodes
  }
  if(p[pos][1]!=-1 && dp[p[pos][1]][1]==0)  //for parent 2,
     if the level has not been yet assigned as it wasn't
     part of the largest tree
  {
    dp[p[pos][1]][1]=cnt; //assign level to the parent node
    assign_p(p[pos][1],cnt);  //recursive function call to
       assign levels to further parent nodes
  }

}
int max_level()
{
  int i=0,index=-1,level_max=-1;  //level_max holds the
     deepest depth from the ancestor at index

  while(lev[i]!=-1) //loop to calculate the oldest ancestor
     level from all the found ancestors
  {
    find2(lev[i],0);
    if(max>level_max) //max is a global variable, check if
       the depth= max found is greater than the one
       previously found
    {
     level_max=max;  //new maximum depth is stored in
        level_max
     index=lev[i]; //index of the oldest ancestor found as
        of now is stored in index
    }
    i++;  //increment i to access the next ancestor
  }
  dp[index][1]=level_max+1; //assign the highest level no.
     to one of the oldest ancestors (in case more than one
     ancestor holds the same heighest heirarchial level)
  assign(index,level_max+1);  //assign levels to all the
     descendants of the oldest ancestor

  for(i=0;i<N;i++)  //loop to assign levels upto the rest of
     the other ancestors such that each ancestor receives
     their heirarchial location with respact to one another
  {
    if(dp[i][1]!=0) //node that has been assigned a level
    {
     assign_p(i,dp[i][1]); //if parents have not been
        assigned levels, then does a recursive loop to do
        so. This is in case of common children between
        people belonging to different geneology
    }

  }

  i=0;
  while(lev[i]!=-1) //loop to assign levels to the rest of
      the ancestors and their descendants (connected or
      disconnected from the main graph)
  {
    if(lev[i]!=index && dp[lev[i]][1]!=0) //if the ancestor
        tree is connected with the other ancestor trees via
        some marriage and common descendant
      assign(lev[i],dp[lev[i]][1]);   //assign heirarchial
          levels to all descendants of such trees
    else if(lev[i]!=index && dp[lev[i]][1]==0)  //if the
        ancestor tree is disconnected with the other ancestor
        trees
    {
      dp[lev[i]][1]=level_max+1;     //assign  the maximum
          level to ancestor
      assign(lev[i],dp[lev[i]][1]); //assign descending
          levels to the descenants of the ancestor
    }

    i++;  //finding the rest of the ancestors and doing the
        same
  }
  return(level_max+1);  //return the max level
}
```

Q7. Use the graph to determine the number of descendents of each member of the geneology. What algorithm will you use?

Ans. The algorithm used makes use of the heirarchial levels of the nodes. The logic behind forming the DP is the fact that the no. of descendants of a parent node = no. of descendants of its children + no. of children. Hence for each lower level, the parent nodes are updated.

dp[N][2] : dp[i][j] stores the heirarchial level of node i at j=1 and no. of descendants at j=0

```
void no_of_decendants2()     //function used to calculate the
    no. of descendents of each node.
{
  int m = max_level();  //find the maximum level and assign
      heirarchial levels to each node
  int i,j;
  for(j=1;j<=m;j++) //levelwise adding the no. of
      descendants from bottom-up approach (note: ignoring
      level 1. ie, the bottom-most level as their
      descendants=0)
  {
    for(i=0;i<N;i++)  //traversing the array to find the
        required level j
    {
      if(dp[i][1]==j) //if the level is found, increment the
          no. of descendants of the parents
      {
        if(p[i][0]!=-1) //for parent 1
        {
          dp[p[i][0]][0]+=dp[i][0]+1; //increment the no. of
              descendants = descendants of current node + the
              current node
        }
        if(p[i][1]!=-1) //for parent 2
        {
          dp[p[i][1]][0]+=dp[i][0]+1; //increment the no. of
              descendants = descendants of current node + the
              current node
        }
      }
    }
  }
}
```

Q8. How is edge information stored?

Ans. Edge information is stored in a seperate matrix edge[N][no_of_children_max]. edge[i][j] stores the age of i when j^{th} child was born.

Q9. What is the time complexity of your algorithm to obtain the desired information, if the number of generations below the node is k?

Ans. For finding the no. of descendants of each individual, the time complexity is : O(mn)
where n = total no .of nodes in the graph.
m = maximum heirarchial level (ie, heirarchial level of one of the oldest ancestors(if there are more than one oldest ancestors in case of connected graphs))
For the specified node having k generations beneath it , the time complexity would be O(kn). In order to find the no. of descendants of node having k generations below it, change m to k.

Q10. For each node, how will you obtain its list of great-grandchildren? Determine how many individuals in the geneology lived to see their great-grandchildren. How will you traverse the graph?

Ans. The list of great grandchildren is formulated by traversing down the ancestor nodes. That is, first great grandchildren of the ancestor nodes are found, further travelling one level below, we obtain the great grandchilden of the children of the ancestor and so on. The information obtained is stored in a matrices g2[i][j] and g3[i][j] where g2 holds indices of the j^{th} great grand child of node at i and g3 holds the indices of the j^{th} great grandchild seen by node at i.

In fun2() used to find and store all the great grand children details (once the great grandchildren of the ancestor is identified) , pos0 to pos3 has been used instead of pos[4] because arrays are always passed by reference, and in my implementation, pos0 to pos3 holds unique values in each recursive function call and the value in one call shouldnt be altered by changes happening during a recursive call to the same function.

pos0 to pos3 holds the indices of the nodes in heirarchial line to the current node , ie, it holds the indices of its parents and ancestors upto to 4 nodes above the level.

ed0 to ed2 holds the corresponding edge values with respect to pos0 to pos3 and is used for determining whether the node at pos0 lived to see the node at pos3 .That is, sum of the edge lengths upto the current node(pos3) should be less than or equal to the death age of the great-grandparent at pos0.

```
void fun2(int count, int pos0,int pos1, int pos2, int pos3,
    int ed0, int ed1, int ed2)  //count is a dummy variable
    used to identify that its a great grandchild
{ //po0 to pos3 holds 4 nodes starting from great
    grandparent to great grandchild and ed0 to ed2 holds the
    edge values.
  int i,gpos;
  count--;
  if(count==0)  //if great grandchild
  {
    gpos=pos0;  //gpos = position of great grandparent
    g2[gpos][0]++;     //incrementing the no. of great
        grandchildren of node at gpos
    g2[gpos][g2[gpos][0]]=pos3; //store the great-grand
        child index in the next storage place after the
        previous one if any.
    if(ed0+ed1+ed2<=atoi(P[gpos].death_age))  //checks if
        the great-grandparent lived to see the great
        grandchild under consideration
    {
      g3[gpos][0]++;  //increment the no. of
          great-grandchildren seen
      g3[gpos][g3[gpos][0]]=pos3; //store the index of the
          great grand child seen
    }
    if(cno[pos3]!=0)  //further children of great grandchild
        of node at gpos exists
    {
      pos0=pos1;  //shift the positions of the heirarchial
          descendant nodes by eliminating the great
          grandparent
      pos1=pos2;
      pos2=pos3;

      ed0=ed1;  //shift the edge values under consideration
      ed1=ed2;

      for(i=0;i<cno[pos2];i++)  //iterative loop through the
          children of node at pos2(great grand child) to
          check for great grandchildren of node at pos0
      {
        pos3=c[pos2][i];  //pos3 stores the new node ie,
            child of pos2
        ed2=edge[pos2][i];  //ed3 holds the edge value
            between pos2 and pos3
        fun2(count+1,pos0,pos1,pos2,pos3,ed0,ed1,ed2);
            //recursive call to find further
            greatgrandchildren to the nodes
      }
    }
  }
```

```
}

void fun3(int count,int pos[],int ed[], int gpos, int n)
    //function to find the great grandchildren of the
    ancestors
{
  int i;
  count--; //decrement count and execute the current
      function till it reaches a great grand child
  if(cno[pos[n]]!=0 && count!=0)  //not great grandchildren
      but further children of node at pos[n] exists
  {
    n++; //increment the depth
    for(i=0;i<cno[pos[n-1]];i++)  //iterative loop through
        the children of node at pos[n-1] to check for great
        grandchildren of node at gpos
    {
      pos[n]=c[pos[n-1]][i];  //add the current child to the
          chain of nodes from the ancestor to great grandchild
      ed[n-1]=edge[pos[n-1]][i];  //add the edge between the
          current node and child considered
      fun3(count,pos,ed,gpos,n);  //recursive call to
          function
    }
  }
  else if(count==0) //if great grandchild found
    fun2(count+1,pos[0],pos[1],pos[2],pos[3],ed[0],ed[1],ed[2]);
        //call fun2 to find the great grand children of all
        descendants of gpos
}

void great_grandkids2() //function to find the great
    grandkids and also the great grandkids individuals lived
    to see
{
  int i,j;
  int pos[4]={-1,-1,-1,-1}; //array used to hold a chain of
      great grandparent to great grandchild
  int ed[3]={-1,-1,-1};    //array to hold the edge values to
      later calculate if the individual lived to see the
      great grand child

  i=0;
  while(lev[i]!=-1) //iterate through the ancestors found
  {
    pos[0]=lev[i];  //pos[0] holds the great grand parent
        index
    fun3(4,pos,ed,lev[i],0);  //recursive function to find
        the great grandchildren of node at i and also the
```

```
            ones node i lived to see
        i++;            //increment to next ancestor
    }
}
```

The graph is traced down once from ancestor to the last great grandparent and this is done for each ancestor. This is how the required data is acquired, by traversing the graph once for each ancestor.

Q11. How will you determne if there was an overlap?

Ans. In the above code, for DP implementation, the graph is traversed down once for each main ancestor. And hence, there wont be an overlap unless two geneologies are connected by common offsprings. Since in the input file provided, there is only one main tree, there wont be an overlap since we are traversing the graph only once from top ancestor to last node.

Q12. Obtain the time complexity to obtain this information in terms of the nodes in the graph, n, and the number of generations, k.

Ans. For finding the great grand children, The time complexity will be p*time complexity for traversing down k generations in an N-ary tree. Here , p is the no. of ancestors and k is average no. of generations below an ancestor. For a binary tree,

$$\begin{aligned} T(n) &= 2 * T(n/2) \\ &= 2 * [\ 2 * T(n/4)\ ] \\ &\vdots \\ &= 2^k * T(n/2^k) \end{aligned}$$

But T(1)=1
Therefore, n=2^k and k=$log_2(n)$

$$\begin{aligned} T(n) &= log_2(n) * T(1) \\ &= n \end{aligned}$$

Therefore, time complexity = O(n) where n is the total no. of nodes in the binary tree.

The same would be true if it was a ternary tree or an N-ary tree. Hence, the time complexity is O(n) where n is the total number of nodes in the tree. But since our case has connected and disconnected geneologies, the time complexity cannot be accurately calculated since common nodes belonging to more than one geneology may get counted more than once. Hence those repeats would also have to be accomodated in n. If that was estimated , then the time complexity would be O(n).

Another way of proving it is,
Let n/k = m be the average no. of children per parent.
Then,

$$\begin{aligned}
T(n) &= m * T(n/m) \\
&= m * [\ m * T(n/m^2)\ ] \\
&\ \ . \\
&\ \ . \\
&\ \ . \\
&= m^p * T(n/m^p)
\end{aligned}$$

Now, T(1)=1
Therefore,
$n/m^p=1$ or $n=m^p$ or $p=log_m(n)$

$$\begin{aligned}
T(n) &= m^{log_m(n)} * T(1) \\
&= n
\end{aligned}$$

Therefore, time complexity = O(n)