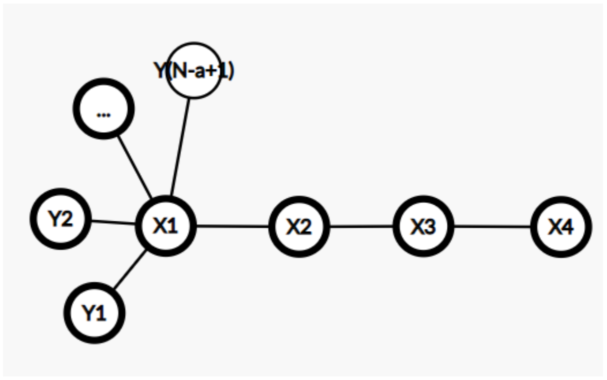
Ethan Wong
COM SCI 180
Discussion 1B
UID 305319001

Homework 2

1. The proposed statement is true. The statement can be proved by contradiction. Say that G is a graph that is not connected. It is split into two distinct sections that are not connected to each other. The first section will be named S1 and the second section will be named S2.
   According to the statement, each node in the graph G has a degree of at least (n / 2). This means that each node in S1 must be connected to a minimum of (n / 2) other nodes. Knowing this, it can then be concluded that S1 must have at least [(n/2) + 1] total nodes in it because any particular node will have an additional (n / 2) nodes that it is connected to. The same logic can be applied to S2. This is problematic because if there are two distinct sections that each have at least [(n/2) + 1] nodes, this means that there are at least (n + 2) total nodes in the graph G because 2 * [(n/2) + 1] = (n + 2). This does not make sense because the statement says there are only n total nodes in G. Thus, the statement is true — for a graph G with n nodes (n is even), if every node of G has degree at least n=2, then G is connected.
   For further proof that the statement is true, a quick example would be as follows. Imagine that n = 2. This means that G has a total of 2 nodes, and therefore one node would belong to S1 and one node would belong to S2. There is only one node in each of the distinct sections. Each node must have a degree of at least one because (n / 2 = 1). However, because there is only one node in each of the sections, this means that there must be an edge between the two nodes to satisfy the requirement of each node being connected to at least (n / 2) other nodes. This proves that the graph G must be connected.
   Note: I am assuming that G is an undirected graph because in lecture, Professor Hsieh said that if not otherwise stated we can assume a graph is undirected for this class.

2. This claim is false and it can be proven by way of an example. There does exist a graph G such that diam(G) ÷ adp(G) > c, contrary to what the problem says. To start off, say that there is a constant number ‘A’. We create a graph of ‘A’ nodes in a straight line so that we have nodes $\{x_1, x_2, \dots x_A\}$. Then we attach n - A + 1 nodes to x_1 so that we have nodes $\{y_1, y_2, \dots y_{n-A+1}\}$. Say for example that A = 4. The graph will look like this:



We can see based on this graph that diam(G) = A. All of the Y nodes have a distance of A=4 when we observe dist(any Y node, X4). Knowing this, we can obtain an upper bound for adp(G).

First, we observe that there are at most (A * n) 2-element sets that contain at least one X node. The maximum distance of any of these pairs is A, because there can not be any path longer than the diameter of the graph. Based on this, we have $A * n = A^2 n$.

Next, we observe that all the other pairs (any pairs without any X nodes) have a maximum distance of 2. Basically, the distance between any of the Y nodes is guaranteed to be 2 because there is the starting node, which connects to X1, which connects to the ending node. This path of three nodes creates a distance of length 2. Based on this we have $2 * \binom{n}{2} = 2\binom{n}{2}$.

Now we can assemble all this information to create an upper bound for adp(G). Based on the formula provided in the problem, we get this inequality:

$$apd(G) \leq \frac{2\binom{n}{2} + A^2 n}{\binom{n}{2}} = \frac{2(n + A^2 - 1)}{n-1} \leq 2 + \frac{2A^2}{n-1}$$

With this, it is finally time to choose a value for n, as we did not do so earlier. If we say that $n - 1 > 2A^2$, based on the above inequality $apd(G) < 3$. Then if we say that $k > 3c$we are given this inequality:

$$\frac{diam(G)}{apd(G)} = \frac{A}{apd(G)} > \frac{3c}{3} = c$$

And thus the claim is false because:

3. a) $\frac{diam(G)}{apd(G)} > c$

Pseudocode for Algorithm:

```
findDiameter(G, N):
  G = tree — given as input to algorithm
  N = root node (if no root is given, choose any node) — given as input to algorithm
  F = node returned from calling A(N)
  S = distance returned from calling A(F)
  P = node returned from calling A(F)
  Return S as the diameter of the tree
```

Proof of Correctness:

This has a constant number of calls to A(·) as there are only two calls. The logic behind this function lies in the fact that the diameter of any given tree will always be two leaf nodes. More precisely, the diameter of a tree is the largest distance between two leaf nodes. This is because if the starting and ending nodes are not leaf nodes, then the distance could be extended by adding on the nodes that lead to the leaf node. Calling A(·) on the root node means it will return the leaf node that is furthest from N — in other words, it will be the “deepest” leaf node in the tree. This node, F, will represent the starting point of the diameter’s path. Calling A(·) on F will yield the furthest leaf node from F and the distance between these two nodes. The distance between these two nodes is guaranteed to be the diameter of the tree. This is because the path of the diameter must travel all the way from P to N, and then from N to F. The algorithm is able to find the diameter with only two calls to A(·).

b) It would be best to use some variation of the DFS algorithm in order to implement the A(·) algorithm. This is because the node v that is returned by A(·) is guaranteed to be a leaf in the tree that is also a corner of DFS. The for loop in the algorithm for part a is constant time because we already know how many times it will run — it will run len(leaf) many times. This means that the calls to the A(·) algorithm will dictate the overall time complexity. Because A(·) is implemented by DFS, the overall time complexity for this diameter computing algorithm for trees will be O(n + m).

4. A way to solve this problem is by using a directed graph G. For every person P_i, their recorded day of birth (B_i) and their recorded day of death (D_i) will be added into the graph as nodes. At this point the graph is just a collection of nodes with no edges connecting them. The directed edges will then represent when one of the events precedes another. For example, there should be edges added in between each person's birth date and death date because every person must be born before they can die.

In the case of the first fact, P_i died before P_j was born. Since the death of P_i precedes the birth of P_j, then an edge from D_i to B_j will be added. In the case of the second fact, the lifespans of P_i and P_j overlapped. This means that an edge from B_i to D_j will be added and an edge from B_j to D_i will be added. After all the nodes and edges are added, then G is complete.

A sure sign of inconsistency would be if the directed graph contains a cycle. This would indicate that each node precedes the other in time, and this would continue in an infinite loop. This is not allowed because there needs to be a singular event that came before all the others. The absence of such an event would mean that some part of the data is incorrect.

An absence of a cycle in the graph indicates that the data is all reasonable and makes sense. The directed graph can then be used to create a topological ordering that produces the proposed dates of birth and death for each of the n people.

Helper functions that uses DFS concept — used to help detect if there is a cycle in a graph:

```
dfsHelper(G, n):
G = undirected graph — given as input to algorithm
n = starting node in G — given as input to algorithm
s = recursion stack
mark n as visited
add n to s
for i in the nodes of G:
  if i has not been visited:
    if dfsHelper(G, i) == true:       //use recursion (just like in DFS)
      return True
  else if i is in s:
      return True
remove n from s
return False

checkCycle(G):
G = undirected graph — given as input to algorithm
for i in the nodes of G:
  if i has not been visited:
    mark i as visited
    if dfsHelper(G, i) == true:
      return True
return False
```

Pseudocode for Algorithm:

```
helpEthnographers(n. m):
G = initially empty undirected graph
n = array containing birth date and death date for each of the people — given as input
m = array of facts (i.e. $P_i$ died before $P_j$ , lifespans of $P_i$ and $P_j$ overlapped) — given as input
for i in n:
  create node $B_i$ for birth date of person i
  create node $D_i$ for death date of person i
  create an edge between $B_i$ and $D_i$
for j in m:
  if j corresponds to fact 1 ($P_i$ died before $P_j$):
    create an edge between $D_i$ and $B_j$
  if j corresponds to fact 2 (lifespans of $P_i$ and $P_j$ overlapped):
    create an edge from $B_i$ to $D_j$
    create an edge from $B_j$ to $D_i$
if checkCycle(G) == true:       //a cycle was found in the graph
  print("The facts collected are not consistent.")
  return;
if checkCycle(G) == false:
  answer = topologicalSort(G)       //using topological sort algorithm from lecture notes on (4/22)
  print(answer)
  return;
```

The time complexity of this algorithm is O(n + m). This is because both of the for loops used to create nodes and the edges run in O(n) time. The checkCycle function runs in O(n + m) time as it utilizes a modified version of the DFS algorithm that was covered in time. The DFS algorithm runs in O(n + m) time, and therefore checkCycle has the same time complexity. In the case that there is no cycle and a topological ordering must be created, the topologicalSort function also has O(n + m) time complexity because it must process m facts in at most n iterations. Therefore, the overall time complexity of my solution to this problem is O(n + m).

5. Pseudocode for the algorithm is below — it is loosely similar to C++ code:
Note: sub stands for subsequence, which is denoted as S' in the problem description
 sequence is the same as S in the problem description
 subLen is the same as "m" and seqLen is the same as "n" in the problem
 I renamed these variables because I felt they were clearer to understand this way

```
bool subsequence(string sub, string sequence) {
  int subLen = length of sub;
  int seqLen = length of sequence;
  int counter = 0;
  if( subLen > seqLen)
     return false;          //subsequence should not be longer than entire sequence

  for( int i = 0; i < seqLen; i++){
     if( counter == subLen)
        break;                    //every element of subsequence has been checked, stop here
     if( seq[i] == sub[counter])
        counter := counter + 1;           //element of sub matches element of sequence
  }
  if( counter == subLen)        //all elements of sub were found in order within sequence
      return true;
  return false;
  }
```

The main idea of what this algorithm is doing is actually quite straightforward. First, it checks if the subsequence is longer than the actual sequence. This should not be possible because the subsequence needs to be contained within the sequence. If this is the case, the algorithm simply returns false and stops there.
Next the algorithm iterates through the sequence one element at a time. It will check to see if the element matches the first element of the subsequence. If it does not match, the next element of the sequence is checked. If it does match, the counter variable is iterated — this makes sure that the algorithm checks the next element of the subsequence and does not just repeatedly check the first element. If this continues and eventually the counter variable is the same value as the length of the subsequence, this means that every element of the subsequence has already been checked and found. The algorithm will then return True. If by the end of the for loop counter is not equal to the length of the subsequence, that means that not all elements of the subsequence were found within the sequence — the algorithm returns False.

The time complexity of this algorithm is O(n) because of the for loop. This is slightly better than the proposed O(n +m) time complexity, but I am hoping that this faster algorithm is acceptable. All the code outside of the for loop should run in constant time so they do not add anything to the overall time complexity. The for loop will run for at most "n" iterations as it may need to go through the entire length of the sequence.

S’ is a subsequence of S if there is a way to delete a certain number of the events from S so that the remaining events, in order, are equal to the sequence S’

Proof by contradiction:

- Let’s say that S’ is a subsequence of S but there is no way to delete a certain number of the events from S so that the remaining events, in order, are equal to the sequence S’
- According to the algorithm, “counter” will be incremented every time S’[counter] == S[i]
- By the time the algorithm is finished running, counter should have the same value as “m” (m is the length of S’) since S’ was said to be a subsequence of S
- However, in this case, counter < m because there is no way to derive S’ from S by deleting a certain number of events from S. Basically, not all of the elements of S’ were found in order within S.
- Because counter < m, there is a contradiction — there is no way that S’ is actually a subsequence of S. S’ would only be a subsequence of S if counter == m
- Therefore, this algorithm is guaranteed to correctly be able to tell whether or not S’ is a subsequence of S