Relationship Identification System

End Term Project

By

Candidates

Section: K18RD

Roll no. 45,47



Lovely Faculty of Technology and Sciences

Lovely Professional University

Student Declaration

This is to declare that this report has been written by us. No part of the report is copied from other sources. All information included from other sources have been duly acknowledged. We ever that if any part of the report is found to be copied, we are shall take full responsibility for it.

Name: Aman Chauhan

Roll no. 47

Section: K18RD

Name: Shreedhar N

Section: K18RD

Roll no. 45

Place: Lovely Professional University

Date: 08-April-2020

Table of Contents

* Background
* Relationship Identification System
* Objectives
* Description
* Algorithm
* Code

Background

In recent years, informal, online communication has transformed the ways in which we connect and collaborate with friends and colleagues. With millions of individuals communicating online each day, we have a unique opportunity to observe the formation and evolution of roles and relationships in networked groups and organizations. Yet a number of challenges arise when attempting to infer the underlying social network from data that is often ambiguous, incomplete and context-dependent.We specifically address the challenge of relationship identification where the objective is to identify relevant communications that substantiate a given social relationship type.

Functionality and Implementation

In this project we have basically created a specific program which is used to identify if or any relationship between two people respectively. The program has a different classes created to contain the function of entities which are used to define a relation between said person and helps identify it through the execution of the said code. The project being on a small level is not very complex and has a basic functionalitywhich is enough for this project.

Objective

The objective of this project is to use the given data and relation function defined in the classes to find the relationship of people which are defined in the classes. This gives us a subtle idea to clear things at a smaller level but also help us to envision bigger projects in the future. Relationship identification system will essentially be used to identify said relation between two entities.

Description

In this code we have created different classes like for example a member class and a family class, each of the classes have definite functions such a in the case of member class we have created get\_parent function which is used to return the parent member node of the particular member and then the relation is printed by using the values which are pre-defined in the code.

Algorithm

|  |
| --- |
|  |
|  |  |
|  | **This is a type of traversing the tree to get specific information. All information originally provided except cousin method.** |
|  |  |
|  | **Input:** |
|  | **Class Member is a class that represents a single person in the family, and Class Family represents the whole family tree.** |
|  |  |
|  |  |
|  | **Output:** |
|  | **Write code for the method cousin of the class Family according to the docstring in FamilyTree.py and the definitions for degree removed and cousin type** |
|  |  |
|  | **A** |
|  | **| |** |
|  | **B C** |
|  | **| | | |** |
|  | **D E F G** |
|  | **| | | | | | | | | |** |
|  | **H I J K L M N O Q R** |
|  |  |
|  | **Degree removed = How many levels different the nodes were on (e.g. node A is zeroth level and D is 2nd level they are two removed)** |
|  |  |
|  | **Cousin type = The node level that is the closest to a common parent/ancestor node. In general, i'th cousins have a grandparent or ancestor that is i levels up from their parents. (e.g. B & C are zeroth cousins, D & G are 1st cousins, H & M are 2nd cousins)** |
|  |  |
|  | **Combined = D & M are 1st cousins 1 removed, B & L are zeroth cousins 2 removed.** |

Code

class Member(object):  
 def \_\_init\_\_(self, founder):  
 *"""   
 founder: string  
 Initializes a member.   
 Name is the string,  
 parent is None, and no children  
 """* self.name = founder  
 self.parent = None   
 self.children = []   
  
 def \_\_str\_\_(self):  
 return self.name   
  
 def add\_parent(self, mother):  
 *"""  
 mother: Member  
 Sets the parent of this node to the `mother` node  
 """* self.parent = mother   
  
 def get\_parent(self):  
 *"""  
 Returns the parent Member  
 """* return self.parent   
  
 def is\_parent(self, mother):  
 *"""  
 mother: Member  
 Returns: Boolean  
 """* return self.parent == mother   
  
 def add\_child(self, child):  
 *"""  
 child: Member  
 Adds another child Member node to this Member  
 """* self.children.append(child)   
  
 def is\_child(self, child):  
 *"""  
 child: Member  
 Returns: Boolean  
 """* return child in self.children   
  
  
class Family(object):  
 def \_\_init\_\_(self, founder):  
 *"""   
 Initialize with string of name of oldest ancestor  
 Keyword arguments:  
 founder -- string of name of oldest ancestor  
 """* self.names\_to\_nodes = {}  
 self.root = Member(founder)   
 self.names\_to\_nodes[founder] = self.root   
  
 def set\_children(self, mother, list\_of\_children):  
 *"""  
 Set all children of the mother.   
 Keyword arguments:   
 mother -- mother's name as a string  
 list\_of\_children -- children names as strings  
 """* mom\_node = self.names\_to\_nodes[mother]   
   
 for c in list\_of\_children:   
   
 c\_member = Member(c)   
   
 self.names\_to\_nodes[c] = c\_member   
   
 c\_member.add\_parent(mom\_node)   
   
 mom\_node.add\_child(c\_member)   
   
 def is\_parent(self, mother, kid):  
 *"""  
 Returns True or False whether mother is parent of kid.   
 Keyword arguments:   
 mother -- string of mother's name  
 kid -- string of kid's name  
 """* mom\_node = self.names\_to\_nodes[mother]  
 child\_node = self.names\_to\_nodes[kid]  
 return child\_node.is\_parent(mom\_node)   
  
 def is\_child(self, kid, mother):  
 *"""  
 Returns True or False whether kid is child of mother.   
 Keyword arguments:   
 kid -- string of kid's name  
 mother -- string of mother's name  
 """* mom\_node = self.names\_to\_nodes[mother]   
 child\_node = self.names\_to\_nodes[kid]  
 return mom\_node.is\_child(child\_node)  
  
 def cousin(self, a, b):  
 *"""  
 Returns a tuple of (the cousin type, degree removed)   
 cousin type is an integer that is -1 if a and b  
 are the same node or if one is the direct descendent   
 of the other. Otherwise, cousin type is 0 or greater,  
 representing the shorter distance to their common   
 ancestor as described in the exercises above.  
 degree removed is the distance to the common ancestor  
 Keyword arguments:   
 a -- string that is the name of a  
 b -- string that is the name of b  
 """* a\_node = self.names\_to\_nodes[a]  
 b\_node = self.names\_to\_nodes[b]  
  
 def create\_branch(node):  
 branch = [node]  
 parent = node.get\_parent()  
  
 while parent:  
 branch.append(parent)  
 parent = parent.get\_parent()  
 return branch  
  
 if a\_node.name == b\_node.name:  
 return (-1, 0)  
 elif a\_node.is\_child(b\_node) or b\_node.is\_child(a\_node):  
 return (-1, 0)  
  
 a\_branch = create\_branch(a\_node)  
 b\_branch = create\_branch(b\_node)  
  
 b\_parent\_index = 0  
 for a\_parent\_index, node in enumerate(a\_branch):  
 try:  
 b\_parent\_index = b\_branch.index(node)  
 break  
 except ValueError:  
 pass  
  
 cousin\_type = max(a\_parent\_index, b\_parent\_index)  
 degree\_removed = abs(a\_parent\_index - b\_parent\_index)  
 return (cousin\_type, degree\_removed)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
   
 f = Family("a")  
 f.set\_children("a", ["b", "c"])  
 f.set\_children("b", ["d", "e"])  
 f.set\_children("c", ["f", "g"])  
  
 f.set\_children("d", ["h", "i"])  
 f.set\_children("e", ["j", "k"])  
 f.set\_children("f", ["l", "m"])  
 f.set\_children("g", ["n", "o", "p", "q"])  
  
 words = ["zeroth", "first", "second", "third", "fourth", "fifth", "non"]  
  
 ## These are your test cases.   
  
 ## The first test case should print out:  
 ## 'b' is a zeroth cousin 0 removed from 'c'  
 t, r = f.cousin("b", "c")  
 print "'b' is a", words[t],"cousin", r, "removed from 'c'"  
  
 ## For the remaining test cases, use the graph to figure out what should   
 ## be printed, and make sure that your code prints out the appropriate values.  
  
 t, r = f.cousin("d", "f")  
 print "'d' is a", words[t],"cousin", r, "removed from 'f'"  
  
 t, r = f.cousin("i", "n")  
 print "'i' is a", words[t],"cousin", r, "removed from 'n'"  
  
 t, r = f.cousin("q", "e")  
 print "'q' is a", words[t], "cousin", r, "removed from 'e'"  
  
 t, r = f.cousin("h", "c")  
 print "'h' is a", words[t], "cousin", r, "removed from 'c'"  
  
 t, r = f.cousin("h", "a")  
 print "'h' is a", words[t], "cousin", r, "removed from 'a'"  
  
 t, r = f.cousin("h", "h")  
 print "'h' is a", words[t], "cousin", r, "removed from 'h'"  
  
 t, r = f.cousin("a", "a")  
 print "'a' is a", words[t], "cousin", r, "removed from 'a'"