15-110: Principles of Computing, Spring 2018

Programming Assignment 6

Due: Tuesday, March 6 by 9PM

Note: You are **responsible for protecting your solutions** to the following problems from being seen by other students both physically (e.g., by looking over your shoulder or verbal discussion) and electronically. In particular, since the lab machines use the Andrew File System (AFS) to share files worldwide, you need to be careful that you do not put files in a place that is publicly accessible.

If you are doing the assignment on the Gates-Hillman Cluster machines we use in the lab or on `unix.andrew.cmu.edu`, please remember to have your solutions inside a `private` folder (which is under your home directory). Our recommendation is that you create a `pa6` folder under `~/private/15110` for this assignment. That is, the new directory `pa6` is inside the directory named `15110`, which is inside the `private` directory.

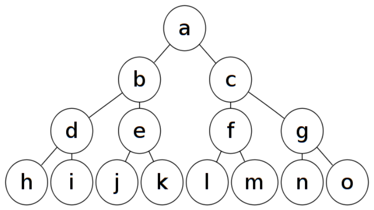
Overview

For this assignment, you will create a Python file for each of the problems below. You should save all of these files in a folder named `pa6`. Once you have every file, you should zip up the `pa6` folder and submit the zipped file on Autolab.

This assignment will help you understand how to use lists to represent trees and graphs to hold data. You learn how to write the corresponding Python code to specific algorithms. In addition to writing the code, you should test your code on multiple inputs to check its correctness.

Problems

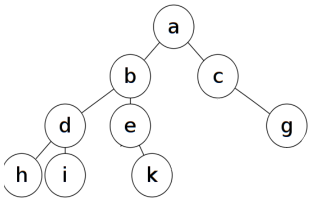
1. [3 points] Recall that, one way to represent the nodes of a complete binary tree is with a list, where the first element contains the root, the next two elements contain the next level of the tree (the children of the root), the next four elements contain the next level of the tree (two containing the children of the root's left child and then two containing the children of the root's right child), and so on, depending on how many levels the tree has.



The binary tree at left with nodes labeled "a" through "o" would be represented by the Python list:

```
[None,"a","b","c","d","e","f","g","h","i","j","k","l","m","n","o"]
  0    1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
```

Note in this representation, we do not use index 0 of the list. (This will be helpful.)



This representation of a tree using a list can be extended to incomplete trees by using None to represent the "missing" nodes that would need to be added to make a complete tree. This is the case for the tree to the left, which can be represented by the Python list:

```
[None,"a","b","c","d","e",None,"g","h","i",None,"k"]
```

Observe that:

- The left child of the node at index 1 (labeled "a") is the node at 2 (labeled "b").
- The left child of the node at 4 ("d") is the node at 8 ("h").
- The right child of the node at 3 ("c") is the node at 7 ("g").
- The right child of the node at 5 ("e") is the node at 11 ("k").
- The parent of the nodes at 4 ("d") and at 5 ("e") is at 2 ("b").
- The parent of the nodes at 8 ("h") and at 9 ("i") is at 4 ("d").

Do you see a pattern? There are simple formulas that can be used to calculate the indices of a node's left child, right child, and parent from that node's index.

(a) (1 pt) Define a function `left(tree, index)` (in `tree_functions.py`) that requires a list representing a binary tree and the index of one of its nodes. It should return (not print) the data of that node's left child. If there is no left child, it should return `None` instead. You may assume that the index given is valid for the given tree.

(b) (1 pt) Define a function `right(tree, index)` (in `tree_functions.py`) that requires a list representing a binary tree and the index of one of its nodes. It should return (not print) the data of that node's right child. If there is no right child, it should return `None` instead. You may assume that the index given is valid for the given tree.

(c) (1 pt) Define a function `parent(tree, index)` (in `tree_functions.py`) that requires a list representing a binary tree and the index of one of its nodes. It should return (not print) the data of that node's parent. If there is no parent, it should return `None` instead. You may assume that the index given is valid for the given tree.

Example usage: (Note: All three functions must compile correctly for Autolab to test.)

```
python3 -i tree_functions.py
>>> tree = [None,"a","b","c","d","e",None,"g","h","i",None,"k"]
>>> left(tree, 4)
'h'
>>> right(tree, 1)
'c'
>>> parent(tree, 8)
'd'
>>> parent(tree, 9)
'd'
>>> left(tree, 5)
>>> right(tree, 7)
>>> parent(tree, 1)
>>>
```

2. [2 points] In a Binary Search Tree, each node has a value that is greater than that of all of the nodes reachable through its left child and that is less than that of all of the nodes reachable through its right child. (We will assume that the tree does not hold two nodes with the same value.) A Binary Search Tree can be stored using a list just as we did in the previous problem.

   The following is an iterative algorithm for searching a list called *bst* encoding a binary search tree to determine whether it contains a node with the value *key*.

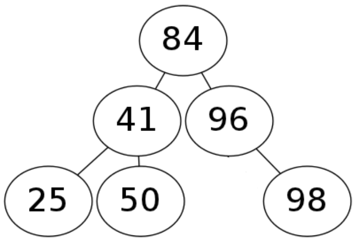
```
I.   Set index to the index of the root node of the tree stored in the list bst.
II.  While index is valid and the value of the node at index is not None, do the
     following:
       A. Set value to the value of the node at index in bst.
       B. Return True if value is equal to key.
       C. Set index to the index of the left child of the node at index if key is less
          than value.
       D. Set index to the index of the right child of the node at index if key is
          greater than value.
III. Return False.
```

   HINT: For this algorithm the index is valid if is less than the length of the *bst* list.

Define a function `bst_search(bst,key)` (in `bst_search.py`) that uses this algorithm to determine whether the value `key` occurs in `bst`, the list representing a binary search tree.

Example usage:

```
> python3 -i bst_search.py
>>> bst = [None, 84, 41, 96, 25, 50, None, 98]
>>> bst_search(bst, 50)
True
>>> bst_search(bst, 51)
False
>>> bst_search(bst, 41)
True
>>> bst_search(bst, 90)
False
```

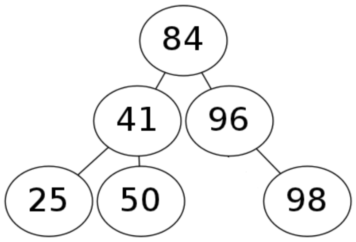


3. [2 points] We can also search a binary search tree recursively. Here is a recursive algorithm to perform the same search as in problem 2 starting at a given index in the tree.

    I. If *index* is not valid, return False.
    II. Set *value* to the value of the node at *index* in *bst*.
    III. If *value* is None, return False.
    IV. If *key* is equal to *value*, return True.
    V. If *key* is less than *value*, return the result of this function on the index of the left child of the node at *index*. Otherwise, return the result of this function on the index of the right child of the node at *index*. (Use recursion here. You will return the results of a call to `bst_search2` with the same `bst` and `key` but with a new index: e.g. `return bst_search2(bst, key, ?????)`)

    Define a function `bst_search2(bst,key,index)` (in `bst_search2.py`) that uses this recursive algorithm to determine whether the value `key` occurs in `bst`, the list representing a binary search tree, starting from the given `index`. When you test this function, you will always use an index of 1 in the initial function call, as shown below.

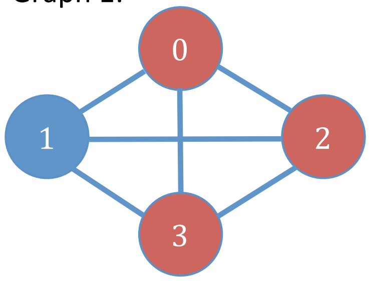
```
> python3 -i bst_search2.py
>>> bst = [None, 84, 41, 96, 25, 50, None, 98]
>>> bst_search2(bst, 50, 1)
True
>>> bst_search2(bst, 51, 1)
False
>>> bst_search2(bst, 41, 1)
True
>>> bst_search2(bst, 90, 1)
False
```



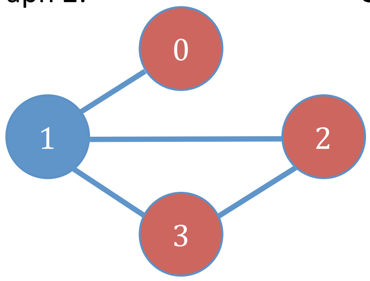
4. [3 points] In a social network, each node represents a person and two nodes are directly connected if the two people are friends. One thing to measure for a person is the cluster coefficient, which measures how connected the person's friends are with each other. If a person's friends are all directly connected to each other, then the person's cluster coefficient is 1.0. If the person's friends are not directly connected to each other at all, then the person's cluster coefficient is 0.0.

Consider the following graphs representing small social networks. In each graph, the person we're analyzing is in blue, and the friends of this person are in red.

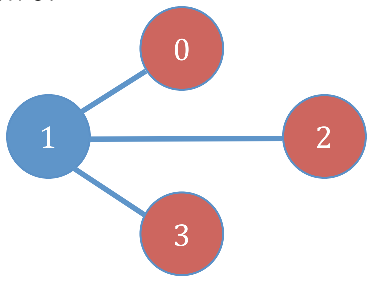
Graph 1:



Graph 2:



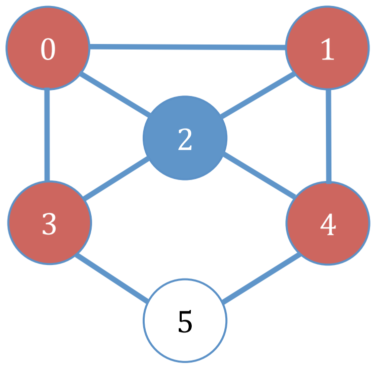
Graph 3:



In graph 1, person 1 has three friends (0, 2, and 3) who all are connected with each other. This is the strongest network of friends and the cluster coefficient is 1.0.
In graph 2, person 1 has three friends (0, 2, and 3), but only 2 and 3 are friends with each other. This is a weaker network of friends and the cluster coefficient is 0.33333.
In graph 3, person 1 has three friends (0, 2 and 3), but none of these friends are friends with each other. This is the weakest network of friends, and the cluster coefficient is 0.0.

Here is another example of a social network, with 6 people, and we're analyzing person 2. Person 2 has four friends (0, 1, 3 and 4). Person 2 and person 5 are not friends, so 5 is shown in white.

Graph 4:



In graph 4, person 2 is directly connected to four friends (in red). Of these four friends, there are 3 connections (0 with 1, 0 with 3, and 1 with 4). Four people can have at most 6 connections, so the cluster coefficient for person 2 is 3 / 6 = 0.5.

We can represent each graph as a list of lists as described in class, where the cost to go between two directly connected nodes is 1, the cost from a node to itself is 0, and the cost between two nodes that are not directly connected is infinity, i.e. `float("inf")`.

(a) In the file `cluster.py`, write four functions `create_graph1()`, `create_graph2()`, `create_graph3()` and `create_graph4()` that return each graph above as a list of lists, using the value `float("inf")` for infinity. Do not use the string `"inf"` by itself. The colors don't mean anything for this problem; they're just for you to see what a cluster is.

Sample usage:

```
python3 -i cluster.py
>>> graph1 = create_graph1()
>>> graph1
[[0, 1, 1, 1], [1, 0, 1, 1], [1, 1, 0, 1], [1, 1, 1, 0]]
>>> graph2 = create_graph2()
>>> graph2
[[0, 1, inf, inf], [1, 0, 1, 1], [inf, 1, 0, 1], [inf, 1, 1, 0]]
>>> graph3 = create_graph3()
>>> graph3
[[0, 1, inf, inf], [1, 0, 1, 1], [inf, 1, 0, inf], [inf, 1, inf, 0]]
>>> graph4 = create_graph4()
>>> graph4
[[0, 1, 1, 1, inf, inf], [1, 0, 1, inf, 1, inf], [1, 1, 0, 1, 1, inf],
[1, inf, 1, 0, inf, 1], [inf, 1, 1, inf, 0, 1], [inf, inf, inf, 1, 1, 0]]
>>>
```

(b) The following algorithm computes the cluster coefficient for a graph *G* for a given node *n*.

```
I.    Set numnodes equal to the number of nodes in the graph G.
II.   Set neighborhood equal to the empty list.
III.  For i in the range 0 to numnodes−1 inclusive, do the following:
      A.  If node n is directly connected to i in the graph G,
          append i to the neighborhood list.
IV.   Set numlinks equal to 0.
V.    For each j in the neighborhood list, do the following:
      A.  For each k in the neighborhood list, do the following:
          1.  If j is not equal to k, and if j and k are directly connected in graph G,
              add 1 to numlinks.
VI.   Set m equal to the length of the neighborhood list.
VII.  Set maxlinks equal to m times m-1.
VIII. Return numlinks divided by maxlinks. (Use regular division.)
```

HINTS:

- If node *a* is directly connected to node *b* in the graph *G*, then *G[a][b]* is equal to 1.
- To compute the number of nodes in the graph, remember that it is a list, so compute its length.

First, test to make sure your functions from part (a) work correctly! Then in the same file `cluster.py`, write another function `cluster_coeff(G, n)` that computes and returns the cluster coefficient for node `n` in graph `G` using the algorithm above. You may assume that `n` is a valid node number and that G is a list of lists representing a graph as defined above.

Example usage:

```
python3 -i cluster.py
>>> graph1 = create_graph1()
>>> graph2 = create_graph2()
>>> graph3 = create_graph3()
>>> graph4 = create_graph4()
>>> cluster_coeff(graph1, 1)
1.0
>>> cluster_coeff(graph2, 1)
0.3333333333333333
>>> cluster_coeff(graph3, 1)
0.0
>>> cluster_coeff(graph4, 2)
0.5
```

Submission

You should now have the `pa6` folder that contains the following four Python files:

a. `tree_functions.py`
b. `bst_search.py`
c. `bst_search2.py`
d. `cluster.py`

Zip up the folder and submit the zipped file named as `pa6.zip` on Autolab. Be sure to check your submission to see that you submitted the correct code.