CS 231
Naomi Nishimura

Module `trees.py`

1 Introduction

Just like the Python list and the Python dictionary provide ways of storing, accessing, and modifying data, a *tree* can be viewed as a way of storing, accessing, and modifying data. Because Python does not have built-in support for trees, I have supplied a module for use in the course.

This document provides background on the types of trees supported by the module as well as specifics of the methods and functions provided. The module is not intended to cover all kinds of trees nor to provide all possible methods and functions. Details of the implementations of the methods and functions are not discussed here; to learn more about such implementations, consider taking CS 234.

At times you may be writing pseudocode that uses tree operations. To use a tree method, simply translate from dot notation, put the name in all capital letters, and capitalize the first letter in each variable name. For example, instead of `tree.set_node_colour(one, new)`, write SET_NODE_COLOUR*(Tree, One, New)*.

2 Tree basics

A tree can be viewed as a special type of graph that contains no cycles (see the document on `graphs.py` for more details on graphs); we will use the term *edge* as is used for graphs, but substitute the term *node* for the term *vertex*. Thus, a *path* in a tree is a sequence of nodes such that there is an edge between each consecutive pair of nodes in the sequence.

In this module, we limit our attention to *rooted trees*, in which each tree has a node specified as the *root*. In a rooted tree, we can view each edge as connecting a node closer to the root (the *parent*) to a node farther from the root (the *child*). Nodes with the same parent are *siblings*.

We distinguish between nodes that each have at least one child (*internal nodes*) and nodes without children (*leaves*).

In a *binary tree*, each node can have at most two children, consisting of at most one *left child* and at most one *right child*. One can also distinguish between trees being ordered (where there is an order imposed on the children of each node) or unordered (where there is no such order). This module supports only unordered trees.

In the course material, we will consider both rooted and unrooted trees, and possibly both unordered and ordered trees.

3 Module `trees.py`

3.1 Objects

The tree module makes use of two different types of objects: Vertex (from `graphs.py`) and Tree. In the trees used in the course, each Vertex object has an ID, a weight, and a colour, where

the weight is an integer and the other attributes are strings; the default values of the weight and colour are `0` and `"white"`. The ID of each Vertex must be unique. Note: Even though the object being used is Vertex (so the same objects can be used in graphs), we will refer to what they support as nodes, to be consistent with tree terminology. Because each node in the tree, except the root, has a unique parent, we can associate the edge between a child and its parent with the child node. An edge can have a weight and a colour; default values are `0` and `"white"`, respectively.

3.2 Creating a tree from a file

The module `trees.py` contains three functions that can be used to form an object of type Tree from a file; `make_simple_tree` uses default values for weights and colours of all nodes and edges, `make_tree` allows the user to specify values of weights and colours for nodes, and `make_full_tree` allows the user to specify values for weights and colours for both nodes and edges. Each of the functions consumes a string (the name of a file) and produces an object of type Tree.

For `make_simple_tree`, a file containing tree data should contain the following information, in this order:

- the number of nodes in the tree (on one line),
- ID, number of children, IDs of children (one line per node, even if there are no children, with the total number of values on a line being the number of children plus two)

For `make_tree`, a file containing tree data should contain the following information, in this order:

- the number of nodes in the tree (on one line),
- ID, weight, colour, number of children, IDs of children (one line per node, even if there are no children, with the total number of values on a line being the number of children plus four)

For `make_full_tree`, a file containing tree data should contain the following information, in this order:

- the number of nodes in the tree (on one line),
- ID, weight, colour, weight of edge to parent (dummy value for root), colour of edge to parent (dummy value for root), number of children, IDs of children (one line per node, even if there are no children, with the total number of values on a line being the number of children plus six)

On the course website you can find examples of text files for use with the functions; the file `sampletree1.txt` is designed to be used with `make_simple_tree`, the file `sampletree2.txt` is designed to be used with `make_tree`, and the file `sampletree3.txt` is designed to be used for `make_full_tree`. In addition, `sampletree4.txt` can be used with `make_tree`.

3.3 Methods

The table below lists the methods that can be used on objects of the class Tree. Pay close attention to the types consumed and produced by each method; sometimes you will be handling objects and sometimes you will be handling string IDs. In all the methods that use them, `one` and `two` are both IDs of nodes in the tree `tree`. The file `treeuse.py` gives an example of the methods being used.

Method	What it does	Cost
`Tree()`	creates a new empty tree	$\Theta(1)$
`repr(tree)`	produces a string of information about nodes in `tree` and their children	$\Theta(n)$
`tree.tree_root()`	produces the ID of the root of `tree`, if any, or `None` if the tree is empty	$\Theta(1)$
`tree.node_weight(one)`	produces the weight of the node with ID `one`	$\Theta(1)$
`tree.node_colour(one)`	produces the colour of the node with ID `one`	$\Theta(1)$
`tree.edge_weight(one)`	produces the weight of the edge from the node with ID `one` (nonroot) to its parent	$\Theta(1)$
`tree.edge_colour(one)`	produces the colour of the edge from the node with ID `one` (nonroot) to its parent	$\Theta(1)$
`tree.is_leaf(one)`	produces True if the node with ID `one` is a leaf and False otherwise	$\Theta(1)$
`tree.children(one)`	produces a list of the IDs of the children of the node with ID `one`	$\Theta(c)$
`tree.parent(one)`	produces the parent of the node with ID `one`, if any, and `None` if `one` is the root	$\Theta(c)$
`tree.add_root(one)`	adds a new node with ID `one` as the root of the empty tree `tree`	$\Theta(1)$
`tree.add_leaf(one, two)`	adds a new node with ID `one` as a leaf that is the child of the node with ID `two`	$\Theta(1)$
`tree.set_node_weight(one, new)`	updates the weight of the node with ID `one` to `new`	$\Theta(1)$
`tree.set_node_colour(one, new)`	updates the colour of the node with ID `one` to `new`	$\Theta(1)$
`tree.set_edge_weight(one, new)`	updates the weight of the edge from ID `one` (nonroot) to its parent to `new`	$\Theta(1)$
`tree.set_edge_colour(one, new)`	updates the colour of the edge from ID `one` (nonroot) to its parent to `new`	$\Theta(1)$
`tree_a == tree_b`	produces `True` if `tree_a` and `tree_b` have the same root IDs, node IDs, and edges (but weights and colours can differ)	see note below

Due to the way that a tree is implemented, a list produced by a method may not have the items appear in a predictable order. Please see information on the module `equiv.py` for functions to use in such situations.

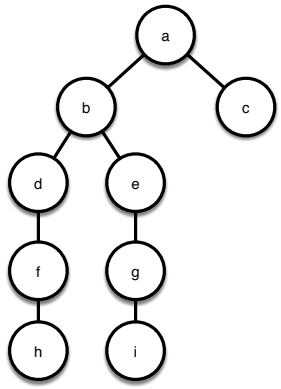

Figure 1: Sample tree 1

You need to ensure that IDs are distinct for all nodes; the code will not check for you.

Because the module is designed to allow you to implement code with trees without considering the details of how the tree is implemented, the worst-case costs listed in the table are not intended to reflect the actual costs of this particular implementation. When writing an algorithm for trees, one often chooses among various options with differing costs for operations. The costs listed here are not the best possible, but a reasonable choice that you can use for analysis. Here we use n to denote the number of nodes in `tree` and c to denote the number of children of node `one`; note that $c \in O(n)$.

For your convenience, the module also allows you to check for equality of trees (use this only for tests, please), where root IDs, node IDs, and edges much match but weights and colours can differ.

4 Using the module to write code

The module uses the modules `graphs.py` and `equiv.py`, so make sure that you have downloaded both before using `trees.py`.

4.1 Copying trees

If you wish to make a copy of a tree, import the `copy` module and use `copy.deepcopy`.

5 Sample trees

Sample trees have been provided for you in the files `sampletree1.txt`, `sampletree2.txt`, `sampletree3.txt`, and `sampletree4.txt`. For your convenience, they have been illustrated here. Code that you write for assignments should work for any tree, not just the samples provided. The nodes in the illustrations are labeled with the node IDs and, if differing from the defaults, weights and colours, in that order; similarly, edges are labeled with weights and colours, in that order.

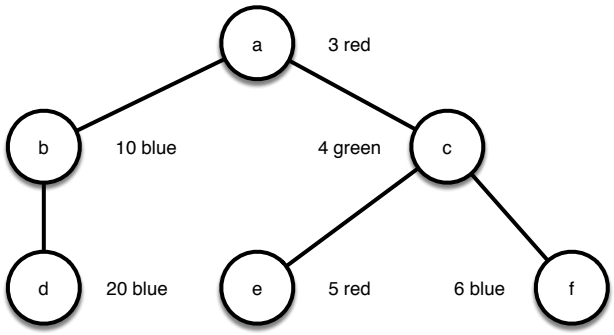

Figure 2: Sample tree 2

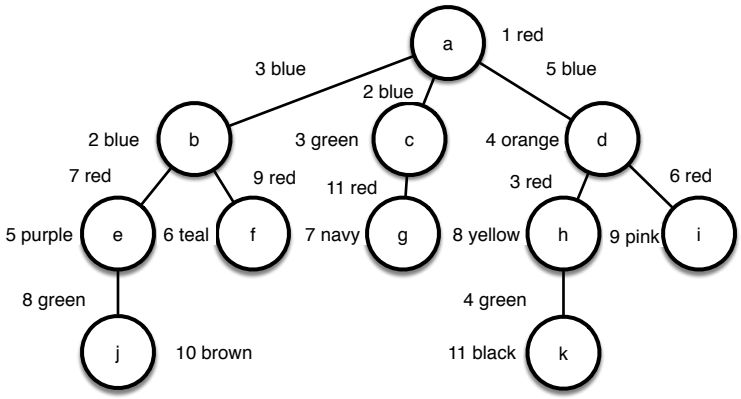

Figure 3: Sample tree 3

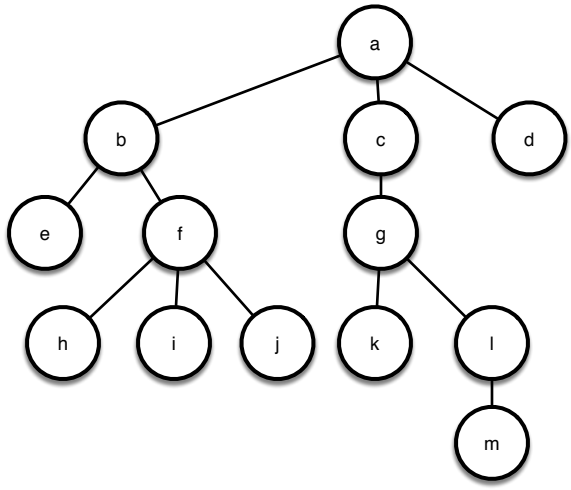

Figure 4: Sample tree 4