

UE5 Fundamentals of Algorithms

Lecture 10: Trees

Ecole Centrale de Lyon, Bachelor of Science in Data Science for Responsible Business

Romain Vuillemot



Outline

- Definitions
- Data structures
- Weighted trees

Trees

Tree is a hierarchical data structure with nodes connected by edges

- A non-linear data structures (multiple ways to traverse it)
- Nodes are connected by only one path (a series of edges) so trees have no cycle
- Edges are also called links, they can be traversed in both ways (no orientation)

Example of trees:

- Binary trees, binary search trees, N-ary trees, recursive call trees, etc.
- HOB (Horizontally Ordered Binary), AVL (Adelson-Velskii and Landis, self-balancing trees), ...
- B-trees, forests, lattices, etc.

Definitions on trees

(similar to the ones for the binary trees)

Nodes - a tree is composed of nodes that contain a **value** and **children**.

Edges - are the connections between nodes; nodes may contain a value.

Root - the topmost node in a tree; there can only be one root.

Parent and child - each node has a single parent and up to two children.

Leaf - no node below that node.

Depth - the number of edges on the path from the root to that node.

Height - maximum depth in a tree.

Definitions on trees (cont.)

N-ary Tree - a tree in which each node can have up to N children. Binary trees is the case where $N = 2$.

Weight - a quantity is associated to the edges.

Degree - the number of child nodes it has. Binary tree is the case where degree is 2.

Subtree - a portion of a tree that is itself a tree.

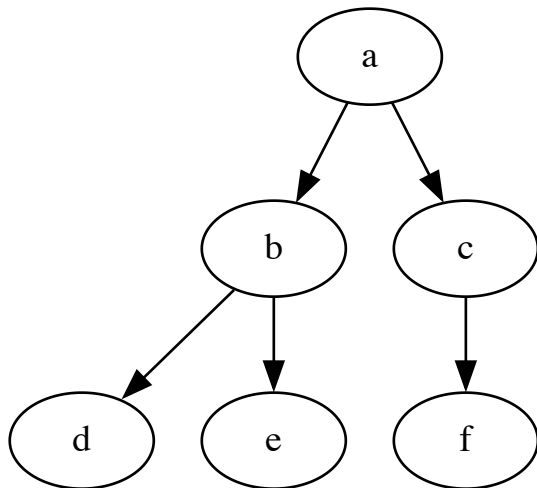
Forest - a collection of trees not connected to each other.

Data structures (dicts + lists)

A simple way is the adjacency list using a dictionary `dict` type.

```
In [147]: tree = {  
    "a": ["b", "c"],  
    "b": ["d", "e"],  
    "c": ["f"],  
    "d": [],  
    "e": [],  
    "f": []  
}
```

```
In [148]: draw_tree(tree)
```



Data structures (dicts + named lists)

- A variation is to use a named variable for the list.

```
In [168]: tree = {  
    "a": {"neighbors": ["b", "c"]},  
    "b": {"neighbors": ["d", "e"]},  
    "c": {"neighbors": ["f"]},  
    "d": {"neighbors": []},  
    "e": {"neighbors": []},  
    "f": {"neighbors": []}  
}
```

```
In [169]: tree["a"]["neighbors"]
```

```
Out[169]: ['b', 'c']
```

Data structures (sets)

- The children are not ordered
- Children names are unique

```
In [115]: tree = {  
    "a": set(["b", "c"]),  
    "b": set(["d", "e"]),  
    "c": set(["f"]),  
    "d": set(),  
    "e": set(),  
    "f": set()  
}
```


Data structures (lists of lists)

- Each node is an entry in the list
- Childre are sub-lists

```
In [122]: tree_list = [  
    ['a', ['b', 'c']],  
    ['b', ['d', 'e']],  
    ['c', ['f', 'g']],  
    ['d', []],  
    ['e', []],  
    ['f', []],  
    ['g', []]  
]
```

Data structures (tuples)

- Each node is the first tuple
- Children are additionnal tuply entries

```
In [119]: tree = ("a", [  
    ("b", []),  
    ("c", [  
        ("d", [  
            ("e", [])  
        ])  
    ])  
])
```

Class object

- The object contains a value and an unrestricted list of children

```
In [176]: class Node:
    def __init__(self, value, children = []):
        self.value = value
        self.children = children

    def get_all_nodes(self):
        nodes = [self.value]
        for child in self.children:
            nodes += child.get_all_nodes()
        return nodes

    def get_all_nodes_iterative(self):
        nodes = []
        stack = [self]
        while stack:
            current_node = stack.pop()
            nodes.append(current_node.value)
            stack += current_node.children
        return nodes
```

```
In [177]: root = Node("a", [
            Node("b", [
                Node("d"),
                Node("e"),
            ]),
            Node("c", [
                Node("f"),
            ]),
        ])

# or using root.children
```

```
In [178]: root.get_all_nodes()
```

```
Out[178]: ['a', 'b', 'd', 'e', 'c', 'f']
```

```
In [175]: root.get_all_nodes_iterative()
```

```
Out[175]: ['a', 'c', 'f', 'b', 'e', 'd']
```

Weighted trees

Trees with a quantity associated to the edges

- Since we have a tree a way to store weights is using nodes values
- Root node weight is 0

Data structures (dicts for edges)

- To encode values in edges we need to add an extra value

```
In [134]: tree = {'a': [{ 'b': 0}, { 'c': 0}],  
                  'b': [{ 'd': 0}, { 'e': 0}],  
                  'c': [{ 'f': 0}],  
                  'd': [],  
                  'e': []  
                }
```

```
In [135]: tree = {  
    'a': [( 'b', 0), ( 'c', 0)],  
    'b': [( 'd', 0), ( 'e', 0)],  
    'c': [( 'f', 0)],  
    'd': [],  
    'e': []  
}
```

Weighted trees as classes

```
In [136]: class Node_weight:
            def __init__(self, data, weight=0):
                self.data = data
                self.children = []
                self.weight = weight

            tree = Node_weight(1)
            child1 = Node_weight(2, weight=5)
            child2 = Node_weight(3, weight=7)
            tree.children = [child1, child2]
```

Exercise: Calculate the total weight of a tree

Go through all the nodes..

Exercise: Calculate the total weight of a tree

Go through all the nodes..

```
In [137]: def get_tree_edges(root):
            edges = []
            stack = [(root, None)]

            while stack:
                node, parent_data = stack.pop()

                for child in node.children:
                    stack.append((child, node.data))
                    edges.append((node.data, child.data, child.weight))

            return edges
```

```
In [138]: tree = Node_weight(1)
            child1 = Node_weight(2, weight=5)
            child2 = Node_weight(3, weight=7)
            tree.children = [child1, child2]
            get_tree_edges(tree)
```

```
Out[138]: [(1, 2, 5), (1, 3, 7)]
```

```
In [139]: sum(tpl[2] for tpl in get_tree_edges(tree))
```

```
Out[139]: 12
```

Exercise: Calculate the total weight of a tree

A recursive version:

```
In [142]: def calculate_total_weight(node):  
            total_weight = node.weight  
            for child in node.children:  
                total_weight += calculate_total_weight(child)  
            return total_weight
```

```
In [143]: calculate_total_weight(tree)
```

```
Out[143]: 12
```

An Edge class for edges

- To consider edges as objects

In [145]:

```
class Edge:
    def __init__(self, source, target):
        self.source = source
        self.target = target

class Node:
    def __init__(self, label):
        self.label = label
        self.children = []

class Tree:
    def __init__(self, root_label):
        self.root = Node(root_label)
        self.edges = []
```

Visualize a tree

```
In [146]: from graphviz import Digraph
          from IPython.display import display

          def draw_tree(T):
              dot = Digraph(format='png')

              def add_nodes_and_edges(tree, parent_name=None):
                  for parent, children in tree.items():
                      dot.node(parent, parent)
                      if parent_name:
                          dot.edge(parent_name, parent)
                      add_nodes_and_edges({child: [] for child in children}, parent)

              add_nodes_and_edges(T)

              display(dot)
```

