Traversal Redux

Christopher R. Genovese

Department of Statistics & Data Science

24 Oct 2024
Session #16

Plan

Recap: Trees

Plan

Recap: Trees

Graphs

Announcements

- On practice
- Survey link https://docs.google.com/forms/d/e/1FAIpQLSd5OfhtlpNVznFKUKFxEPQNM6ZOXh2y8tOseyTc2xs8eANxIA/viewform
- `fpc` code, python 3.12
- **Reading**:
  - F examples 2.18, 2.19, 5.11
- Democracy Day
- **Homework**: `migit-2` due Tuesday 05 Nov, `kd-tree` Exercises #1 and #2
- Coming topics: Dynamic Programming, Functional Thinking Part 3, Parsing, Monte Carlo, Databases, Optimization Part 2

Plan

Recap: Trees

Graphs

Tree Flavors

```
type BinaryTree a = Tip | Branch (BinaryTree a) a (BinaryTree a)

type RoseTree a = Node a (List (RoseTree a))

-- Heterogenous version of Rose Tree, different types on Leaf and Branch
type HTree b l = Leaf l | Branch b (NonEmptyList HTree b l)

type Trie k m a = Trie { data : Maybe a
                       , children : Map k (Trie k m a)
                       , annotation : m      -- m will be a Monoid
                       }

-- One way to think about unrooted trees; there are others.
-- Fin n is the type of integers 1..n. OrderedPairs a is like Pair a a
-- but represents pairs (x, y) where x < y.

type UnrootedTree a = forall n.
         Pair(Injection Fin(n - 1) (OrderedPairs (Fin n)), Fin n -> a)
```

Recap: Unfolding

Last time, you implemented unfold

```
-- Binary case
unfold : (b -> (a, Maybe b, Maybe b)) -> b -> BinaryTree a

-- Rose case
unfold : (b -> (a, List b)) -> b -> RoseTree a
```

Notice how the flexibility in the *type* constrains the possible implementations. We used that to our advantage.

Debrief and discussion.

```
def complete_btree(depth: int) -> BinaryTree[int]:
    "Returns a complete binary tree of given depth with integer data."
    def generate(k):
        if k < 2 ** depth - 1:
            return (k, 2 * k + 1, 2 * k + 2)
        return (k, Tip, Tip)

    return BinaryTree.unfold(generate, 0)
```

Traversing, Mapping, and Folding Trees

Four common operations:

- A **traversal** of a structure is an organized visit to every part of the structure exactly once.
- A **map** of a structure converts it to another structure of the *same shape*, applying a function at each point. (A structure you can map over is a functor.)
- A **fold** of a structure converts that structure so a single value, reducing the structure systematically.
- A **homomorphism** of a structure maps to another structure that respects the relations constituing the structure.

During traversals of a tree, we often do an ***effectful*** computation at each node, e.g., input/output, database queries, modifying global state.

As we will see, we can represent effectful computation with a special type of Functor, and `traverse` has type

```
traverse: (a -> f b) -> t a -> f(t b)
```

For instance, for a computation that does input/output we would have

```
traverse: (a -> IO b) -> t a -> IO(t b)
```

Let's unpack this.

Folding a Tree

We've seen how to map and unfold a tree, now sketch a basic version of a fold.

```
type MonoidalFold v m r = record MonoidalFold where
                                   monoid: m
                                   wrap : (v -> m)
                                   done : (m -> r)

foldTree : MonoidalFold v m r -> Tree v -> r
```

Let's build a simple version of this. Assume that Monoid objects have two fields: `munit: m` and `mcombine: m -> m -> m`.

Class Coding Challenge: Traversing a tree by level

Last time, you implemented at least one tree traversal of the form

- Preorder: The root of a subtree is visited before its children.
- Postorder: The children of a subtree are visited before the root.
- Inorder (Binary case): Left-Root-Right order

Discussion and debrief. Other orders are possible and useful.

Now, consider the following challenge for a rooted binary or rose tree:

> *Given a tree with n nodes, create a tree of the same shape but with the leaves numbered* 1..*n level by level from the root and left to right.*

What is the difficulty here? What functions do we need? What are their types? What data structures do we need? Can you do it without global state?

Let's do it. This exercise will give us insight into the graph case.

Aside: A Beautiful, Lazy Solution

From Jones and Gibbon (Osaki, 2000), translated to TL1:

```
-- Given the list of next available index at each level
-- Produce a tree numbered with these at each level
-- and return the updated list

bfn : (List Int, BinaryTree a) -> (List Int, BinaryTree Int)
bfn (inds, Tip) = (inds, Tip)
bfn (Cons ind inds, BinaryTree left _ right) =
  (Cons (ind + 1) inds'', BinaryTree left' ind right')
  where (inds', left') = bfn (inds, left)
        (inds'', right') = bfn (inds', right)



-- When done, the next available index at one level is the first available
-- for the next level.  And poof, like magic

bfnum : BinaryTree a -> BinaryTree Int
bfnum t = t'
  where (ks, t') = bfn (Cons 1 ks, t)
```

After Hours: Printing Trees

Write a function `to_string` that converts a tree (of whichever form you like) to a printable string that represents the structure of the tree.

Demo

Consider a divide and conquer approach.

- What information do you need to maintain at any point?
- How does recursion figure in to this? How does the recursion reflect the structure of the tree?
- Do you see a Monoid structure here? (You don't have to use it today.)

Class Work: Searching with Tries

Consider a simple form of the "Trie," a special type of rose tree that can be used for efficiently associating values with a set of strings.

We consider a simplified form here

```
type Trie a = Trie { data : Maybe a
                   , children : Map Char (Trie a)
                   }
```

where `Map k v` is an associative map (e.g., dictionary) from key type `k` to value type `v`.

When a string like "foobar" (and associated value) are inserted into the trie, we start at the root node (associated with the empty string) and add a child for the first character (f), then moving to that child, add a child of that for the second character (o), and so on until the string is done. We store the value in the final node. (Picture)

We have an interface

```
empty  : Trie a
insert : String -> Trie a -> Trie a
lookup : String -> Trie a -> Maybe a
```

Let's make a simple trie implementation together.

Plan

Recap: Trees

Graphs

Graphs

A **graph** is a collection of nodes and edges that describe pairwise relationships (edges) between entities (nodes).

There are many, many flavors of graphs. Particularly important features:

- Directed versus Undirected – do the edges have a direction?
- Simple versus multi – can there be more than one edge between two nodes?
- Loops versus No Loops – can there be an edge from a node to itself?
- Labeled (edge or node) versus unlabeled – is there extra information associated with nodes and/or edges?

Function and Functorial representations.

- Graph representations (adjacency list and matrix)
- Traversals: simple breadth first and depth first
- Getting more out of the traversal.

Let's build it.

Depth-First versus Breadth First Traversal

- **Breadth-First Search** (BFS) :: visit all neighbors of the current node before visiting any of *their* neighbors.
- **Depth-First Search** (DFS) :: visit all neighbors of the next visited node before visiting the other neighbors of the *current* node.
- **Priority Search** :: visit nodes in priority order, adding neighbors at each stage

Aside:

- Stack: push, pop, isEmpty, peek
- Queue: enqueue, dequeue, peek, isEmpty
- Priority Queue: enqueue(obj, priority), dequeue, peek, isEmpty
- Deque: pushFront, popFront, pushBack, popBack, peekFront, peekBack, isEmpty

Traversal Powerup

See code in `documents/Src/graphs`.

- A traversal template
- Tracking traversal state
- Configuring the traversal: actions and stores
- Applications

The End