Let's Get Greedy

Christopher R. Genovese

Department of Statistics & Data Science

Tue 21 Oct 2025
Session #15

Plan

Hughes Example (cont'd)

Plan

Hughes Example (cont'd)

Greedy Algorithms

Plan

Hughes Example (cont'd)

Greedy Algorithms

A Quick Tour of Zippers

Announcements

- In documents: lecture notes, fpc draft, `Src/lazy`
- **Reading**:
  - Zippers
    - You could have invented Zippers
    - Huet Zippers (optional but interesting)
    - Zippers as Derivatives (optional, for later)
  - Hughes, Why Functional Programming Matters
  - Commentary on Hughes with Python examples
  - Starting sequence on categories
    - What is Category Theory?
    - Definitions and Examples
    - What is a Functor? Part 1
    - What is a Functor? Part 2
    - Fibonacci Functor
    - Natural Transformations
  - Backus, Can Programming be Liberated from the Von Neumann Architecture
- **Homework**:
  - **zippers** assignment due Fri 31 Oct.
  - **classification-tree-basic** assignment due Thu 23 Oct.
  - Push outstanding mini-assignments when complete

Plan

Hughes Example (cont'd)

Greedy Algorithms

A Quick Tour of Zippers

Newton-Raphson Square Roots

Consider the recurrence relation $a_{n+1} = (a_n + x/a_n)/2$ for $x > 0$ and $a_0 = x$. As n increases, $a_n \to \sqrt{x}$.

We might compute with this typically with

```
def sqrt(x, tolerance=1e-7):
    u = x
    v = x + 2.0 * tolerance
    while abs(u - v) > tolerance:
        v = u
        u = 0.5 * (u + x / u)
    return u
```

We will put this in a more modular style with ingredients that can be reused for other problems.

Newton-Raphson Square Roots (cont'd)

```
next : Real -> Real -> Real
next x a = (a + x / a) / 2

iterate f x = x :: (iterate f (f x))     -- a lazy sequence

iterate (next x) init   -- lazy sequence of sqrt approximations

absolute : Real -> Sequence Real -> Real
absolute tol (a0 :: (a1 :: rest))
   | (abs(a0 - a1) <= tol) = a1
   | otherwise             = absolute tol (a1 :: rest)

relative : Real -> Sequence Real -> Real
relative tol (a0 :: (a1 :: rest))
   | (abs(a0/a1 - 1) <= tol) = a1
   | otherwise               = relative tol (a1 :: rest)

a_sqrt init tol x = absolute tol (iterate (next x) init)
r_sqrt init tol x = relative tol (iterate (next x) init)
```

These same primitives apply to give us other approximations, e.g., numerical differentiation, integration, ...

Implementation: Lazy Sequences

`iterate f x = x :: (iterate f (f x))` creates a ***lazy sequence*** of the form $x, f(x), f(f(x)), f(f(f(x))), \ldots$.

Let's look at our implementation of `iterate` as well as functions:

- `cons x lazy_seq` that returns new lazy seq with `x` at the front
- `take n lazy_seq` that returns a list of the first n items from `lazy_seq`.
- `drop n lazy_seq` that returns the tail of `lazy_seq` that drops the first n items.
- `split_at n lazy_seq` that returns a pair: a list with the first n items and the lazy sequence of remaining items.

We can then use this to implement the above.

Numerical Derivatives

```
divDiff f x h = (f (x + h) - f x) / h
halve x = x / 2
derivative f x h0 = map (divDiff f x) (iterate halve h0)

absolute tol (derivative f x h0)

-- sharpen error terms that look like a + b h^n
sharpen n (a :: (b :: rest))
  = ((b * (2**n) - a) / (2**n - 1)) :: (sharpen n (b :: rest))

order (a :: (b :: (c :: rest)))
  = round (log2 ((a - c) / (b - c) - 1))

improve seq = sharpen (order seq) seq

absolute tol (improve (derivative f x h0))

absolute tol (improve (improve (derivative f x h0)))

- ... can improve even further.
```

Plan

Hughes Example (cont'd)

Greedy Algorithms

A Quick Tour of Zippers

Greedy Algorithms

In solving optimization problems, we make choices at each of a sequence of steps. If the number of complexity of the choices is high, finding an optimal solution can be hard, perhaps infeasible. But if we can reduce the number of choices to few – or even *one* – things become considerably easier.

A **greedy algorithm** is one in which we make the choice that ***looks best at each particular moment***.

This is a local strategy in the sense that we make our choice based on the information we have at the moment without concern or consideration for its downstream implications.

A greedy algorithm works when this sequence of *local* choices leads to a *global* optimum.

Question

Consider stepwise regression – a greedy algorithm for variable selection in linear regression.

We start with only an intercept and successively add variables At each stage, we choose the variable to add with the largest "F-to-enter" (the squared regression t-statistic if that variable is added to the current set), or none if all the F-to-enter's are too small.

Does this greedy algorithm give an optimal solution? Why or why not?

A Framework for Analyzing Greediness

Most optimization problems have a variety of ***candidates*** that can be selected at each step. Imagine that we have a function `candidates` that takes a default initial candidate and some generic list of problem components and returns *all* candidates for the solution.

```
candidates : Candidate -> List Component -> List Candidate
```

Then, we can solve the problem by choosing the best according to the problem's cost function:

```
exhaustive : Candidate -> List Component -> Candidate
exhaustive c_0 = minWith cost . candidates c_0
```

where `minWith cost` selects an item from the input list with minimum cost

```
minWith : Ord c => (a -> c) -> List a -> a
minWith f = foldRight1 (smallerBy f)
  wherre smallerBy f x y = if f x <= f y then x else y
```

Unfortunately, while `exhaustive` search will find an optimal solution, it is not usually efficient, so we want something better if possible.

Fortunately, by equational reasoning from this idealistic code, we can derive a generic *greedy* algorithm and the conditions that make it work.

Deconstructing Exhaustive Search

We can write the `candidates` function as

```
candidates : Candidate -> List Component -> List Candidate
candidates c_0 components = foldRight step [c_0] components
  where step comp cands = concatMap (extend comp) cands
```

where

```
concatMap : (a -> List a) -> List a -> List a
extend : Component -> Candidate -> List Candidate
```

For example, if `Candidate`'s were permutations of a list, then `c_0` would be the empty list and `extend comp` would be a list of all the ways that `comp` could be inserted into a given permutation. E.g.,

```
extend 1 [3, 6, 9] = [[1, 3, 6, 9], [3, 1, 6, 9], [3, 6, 1, 9], [3, 6, 9, 1]]
```

The Fusion Law

We want to decompose `exhaustive` to construct a greedy algorithm that uses the best candidate at each step.

The key to doing this is the **fusion law of folding**. We have

```
exhaustive comps
= minWith cost (candidates c_0 comps)
= minWith cost (foldRight step [c_0] comps)
```

Letting

```
post = minWith cost
init = [c_0]

exhaustive c_0 comps = post (foldRight step init comps)
```

The fusion law says that

```
post (foldRight step init comps) = foldRight gstep ginit comps
```

when `ginit = post init` and for all `c`, `a`

```
post (step c a) = gstep c (post a)
```

If this holds and we can find `gstep`, then `ginit = minWith cost [c_0] = c_0` and

```
greedy c_0 comps = foldRight gstep c_0 comps
```

will return an optimal solution!

The Proof

```
  minWith cost (step x cands)
= { definition of step }
  minWith cost (concatMap (extend x) cands)
= { by a "distributive" property below }
  minWith cost (map (minWith cost . extend x) cands)
= { define gstep x = minWith cost . extend x }
  minWith cost (map (gstep x) cands)
= { the "greedy condition"}
  gstep x (minWith cost cands)
```

The distributive property states that we can divide up the problem into pieces (in order), solve the pieces, and take the best piece solution to get the same overall solution.

```
minWith f (concatMap g xs) = minWith f (map (minWith f . g) xs)
```

This is fairly straightforward to prove.

The **greedy condition** is the essential piece to make this work. *It does not always hold!*

Note in particular that if there are multiple solutions, we pick one arbitrarily, so order effects can make the greedy condition fail.

(There is a way to weaken this condition substantially!)

Example: Making Change

Problem: Given a monetary value `amount` in cents and a set of coin denominations `coins`, also in cents, generate a way to make change that requires the smallest number of coins.

Write this as a function `best_change_for` (or `best-change-for` or `bestChangeFor` depending on your language conventions). You can assume that the smallest denomination is 1.

For example:

```
best_change_for(17, [1, 5, 10, 25])  #=> [10, 5, 1, 1]
```

Questions:

1. What do the candidates look like? Can we find them all?
2. What is the cost function here?
3. Is the greedy condition satisfied here?
4. How would you implement this?

Making Change: Candidates

```python
from itertools import chain

def change_for(amount, coins, current=None):
    assert coins is not None, "Coin denominations required"

    coins = set(coins)
    current = [] if current is None else current

    if amount == 0:
        yield current
    elif amount < 0 or len(coins) == 0:
        return # cannot make change
    else:
        reduced_coins = coins.copy()
        next_coin = reduced_coins.pop() # removes arbitrary coin
        next_amount = amount - next_coin
        sequel = current[:]
        sequel.append(next_coin)

        ways = chain(change_for(next_amount, coins, sequel),
                     change_for(amount, reduced_coins, current))
        for change in ways:
            yield change
```

Making Change: Candidates

Try it!

```
ways = change_for(47, [1, 5, 10, 25])
ways #=> <generator object change_for at 0x7fa596866c50>
next(ways) #=> [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...]

for way in ways:
    print(way)
```

Making Change: Candidates

This can be put in the form we saw earlier, giving a greedy algorithm that looks like:

```
changeFor amount coins = fst (foldRight gstep ([], amount) (reverse coins))
  where
    gstep denom (change, remains) = let c = remains div denom
                                    in
                                      (snoc change c, remains - c * denom)
```

The greedy condition does not work with cost equal to the number of coins.

But it does work for the lexicographically sorted coin tuples (maximizing), which **holds only for some sets of denominations**.

```
[10, 2, 1, 1] > [7, 3, 2, 1, 1]
```

Example: Huffman Coding

We have a stream of tokens drawn from some distribution. We want to encode that stream to compress the data. Specifically, we assign a binary codeword (a string of 0's and 1's) to each token such that more frequent tokens have shorter code words.

We want to come as possible to the *information bound* on data compression for this source.

Note: We will consider only **prefix codes**, in which no code word is a prefix of any other code word. This makes decoding efficient and (it turns out) costs us nothing in effectiveness.

We will represent binary prefix codes by binary trees, with left branches corresponding to 0 and right branches corresponding to 1, and with tokens at the leaves.

```
data BTree t = Leaf t | Branch (BTree t) (BTree t)
  deriving (Display, Functor)
```

Aside: Making a Code

How do we build a code from a tree? Try a functional approach or use `fp-concepts`, if you can.

```
makeCode : BTree a -> NonEmptyList (a, Text)
```

Aside: Making a Code

In TL1:

```
makeCode : BTree a -> NonEmptyList (a, Text)
makeCode = go ""
  where go sofar (Leaf x) = [(x, sofar)]
        go sofar (Branch l r) = go (sofar <> "0") l ++ go (sofar <> "1") r
```

Using `fp-concepts`:

```
def make_code[A](bt: LeafyBinaryTree[A]) -> List[tuple[A, str]]:
    tagged = bt.imap(lambda index, value: (value, "".join(index)))
    return tagged.foldMap(identity, Collect)
```

Now, given a code, how would you write `encode` and `decode`, where both have type `Code -> Text -> Text`, where `Code` is a data structure derived from the result of `make_code`/`makeCode`.

Implementation

We start with a forest, where each tree is a singleton node for one token (and its frequency). We then greedily combine them. The greedy algorithm is linear-logarithmic in general.

Consider how you would implement this.

Here's a simple example to use:

```
|-------+------+------+------+------+------+------|
| token |    a |    b |    c |    d |    e |    f |
| freq  | 0.45 | 0.13 | 0.12 | 0.16 | 0.09 | 0.05 |
|-------+------+------+------+------+------+------|
```

The types:

```
huffman : (Numeric f, Ord f) => NonEmptyList (f, a) -> NonEmptyList (a, Text)
huffman = makeCode . huffmanTree

huffmanTree : (Numeric f, Ord f) => NonEmptyList (f, a) -> BTree a

makeCode : BTree a -> NonEmptyList (a, Text)
```

Implementation

In TL2, this looks like

```
function huffman(tokens, freqs):
    q = PriorityQueue()
    for entry in zip(freqs, tokens):
        q.insert(entry)

    n = len(tokens)
    for iteration in range(1,n):
        freq1, min1 = q.pop_min()
        freq2, min2 = q.pop_min()
        new_freq = freq1 + freq2
        q.insert((freq1 + freq2, Tree(new_freq, left=min1, right=min2)))
    return q.pop_min()[1]
```

Implementation

In TL1, this looks like

```
huffman : (Numeric f, Ord f) => NonEmptyList (f, a) -> NonEmptyList (a, Text)
huffman = makeCode . huffmanTree

huffmanTree : forall f a. (Numeric f, Ord f) => NonEmptyList (f, a) -> BTree a
huffmanTree tokens =
  let queue = PriorityQueue.fromList $ map (fst `fork` Leaf . snd) tokens
  in
    snd $ PriorityQueue.findMin $ until PriorityQueue.isSingle update queue
  where
    update : PriorityQueue f (BTree a) -> PriorityQueue f (BTree a)
    update pq = let ((freq1, best1), pq') = PriorityQueue.popMin pq
                    ((freq2, best2), pq'') = PriorityQueue.popMin pq'
                    freq' = freq1 + freq2
                in
                  PriorityQueue.insert freq' (Branch best1 best2) pq''

makeCode : BTree a -> NonEmptyList (a, Text)
makeCode = go ""
  where go sofar (Leaf x) = [(x, sofar)]
        go sofar (Branch left right) = go (sofar <> "0") left ++
                                       go (sofar <> "1") right
```

Plan

Hughes Example (cont'd)

Greedy Algorithms

A Quick Tour of Zippers

Zippers for Trees

Zippers are a data structure that enable the exploration and modification of other data structures (like trees) in a functional way.

Goals: Move freely through a tree, allow local modifications that can maintain the original tree while efficiently sharing structure.

Metaphor: Moving through the directory/folder tree on your computer.

The End