COMP 50 Lab: Faster trigrams

December 4–5, 2013

In class we showed that the binary-search trees for the trigrams project are far out of balance. In this lab, you will try two different ways to balance them.

Fundamentals

Any inorder listing of a binary-search tree can be converted back to a binary-search tree using *generative recursion* as follows:

- The empty list is converted to the empty tree.
- For a nonempty list, you can apply the following procedure:

  a) Choose *any* key-value pair to be the root of the tree.
  b) Make the left subtree using all the pairs that *precede* the chosen pair in the listing.
  c) Make the left subtree using all the pairs that *follow* the chosen pair in the listing.

  The algorithm is guaranteed to terminate because by taking away one pair, you are guaranteed that the remaining list of pairs is smaller than the problem you started with.

Balance by node count

For this part you will use three measures of quality:

- The *path ratio* as computed in class
- The *average path length* to a `node`
- The *time taken to classify* `http://www.nytimes.com/`

The first and third codes, you have implemented.

1. *Design a function* to compute the average path length to a node in a *nonempty* tree. The signature of this function should be `(bst X) -> number`.

   Like other functions that compute averages, this function can't be implemented using simple structural recursion.
2. Record the goodness of your models on these three measures.
3. *Design a function* that takes a list of key-value pairs and produces a *balanced* binary-search tree. The first step should be to sort the pairs by key so that you have an inorder listing. You will find `take` and `drop` useful.

   The signature of your function should be `(alist string X) -> (bst X)`.
4. *Balance the trees* used in your models, and record their goodness on each of the three measures above.

Balance by frequency (aka weight)

It is possible to do better then a perfectly balanced binary-search tree. The key is to recognize that because some trigrams are more common than others, their nodes will be visited more frequently than others. By putting the more frequently visited nodes in smaller trees, we get them closer to the root and save time overall.

And we already have the data! In the trigrams project, the value associated with a node is exactly equal to the number of times that node is visited during training. So we can use that data as an assumption.

5. Design a function to the compute the *weighted* average path length in a *nonempty* tree. Its signature should be `(bst number) -> number`.

   - Assume that each node *n* is visited a number of times that is equal to `(node-value` *n*`)`.
   - The path length to a node is the number of string comparisons needed to reach that node—so the root has a path length of 1.
   - The weighted average path length is the sum total of all of the path lengths times the number of visits. So if a node with path length 1 is visited 3 times and a node with path length 2 is visited 10 times, the total path length is 23 and when visited by 13 visits the average weighted path length is 23/13.

6. *Design a function* that takes a list of key-value pairs and produces a *weight-balanced* binary-search tree. As before, the first step should be to sort the pairs by key so that you have an inorder listing.

   To balance the weights, the idea is that you choose the root node such that as closely as possible, the left and right subtrees have the same total weight. You will find it useful to define functions `take-weight` and `drop-weight`. Craft their purpose statements carefully.

   The signature of your function should be `(alist string number) -> (bst number)`.

   Note: this problem is *not* like the fish-distribution problem: *you have to preserve the order* of the nodes, or the result won't be a binary search tree.

7. *Weight-balance the trees* used in your trigram models, and record their weighted average path length and the time needed to classify `http://www.nytimes.com`.

Submitting the lab

Five minutes before the end of the lab, put the following text at the beginning of a DrRacket file. You may use an empty file or the source code you have been working on:

```
#|
What I did during this lab:
   (you fill in this part)

What I learned during this lab:
   (you fill in this part)

|#
```

Finally, *submit this file through the handin server* as `lab-final`. **Submit it as lab, not as homework.** You will need to submit it using *two* usernames connected by a + sign, as in

```
Jane.Doe+Richard.Roe
```

You submit using Jane Doe's password.