Trees

COMPSCI 230 — Discrete Math

March 22, 2016
Overview

A Simple Spelling Checker
Tree Nomenclature

Tree Traversal Orders
Depth-First Traversal Orders
Breadth-First Traversal Order
A Simple Spelling Checker

• A spelling checker highlights incorrectly spelled words as you type
  Thas is an example ov poorl
• Note the incomplete word in red at the cursor
• Needs to compare against a large dictionary, say \( \approx 100k \) words
• Should use minimal computation resources
• How can we store the dictionary for rapid access?
A Simple Spelling Checker

A Small Sample Dictionary

<table>
<thead>
<tr>
<th>be</th>
<th>begin</th>
<th>weeks</th>
<th>ward</th>
</tr>
</thead>
<tbody>
<tr>
<td>bear</td>
<td>begins</td>
<td>word</td>
<td>warden</td>
</tr>
<tr>
<td>bears</td>
<td>we</td>
<td>words</td>
<td>was</td>
</tr>
<tr>
<td>bee</td>
<td>wear</td>
<td>work</td>
<td>wasp</td>
</tr>
<tr>
<td>bees</td>
<td>wearer</td>
<td>worker</td>
<td>wasps</td>
</tr>
<tr>
<td>beg</td>
<td>week</td>
<td>war</td>
<td>waste</td>
</tr>
</tbody>
</table>

- Many words have prefixes in common
- Want to check for each typed character if the string typed so far is a word
- Use a *rooted tree*
A Simple Spelling Checker

The Dictionary Tree

COMPSCI 230 — Discrete Math  Trees  March 22, 2016  5 / 22
The Root

- A tree is *rooted* when one of its nodes is designated as the root.
- Any node could be the root, put the root at the top.

```
    1
   / \    
  2   5
 / \  /  /
3  4 6
```

- [This tree is *binary*: it has a *branching factor* of at most 2]
- [It would be *strictly binary* if the b.f. were always either 2 or 0]
- The same tree, but rooted at node 2 (no longer binary!)

```
    2
   / |
  3 4
```

- These are different as *rooted trees*, the same as *free trees*.
A Possible Implementation

- Each node in the dictionary tree has one child per letter of the alphabet.
- The *depth* of a node is the number of edges between it and the root.
- The *branching factor* of each node is the length of the alphabet.
- All but two of the children of the root are empty subtrees.
- We know the letter from the position among the siblings.
- No need to store the letters anywhere.
- All nodes look the same, except for which children are empty and whether the node *value* is True or False.
What Letter Corresponds to Child \( n \)?

- It is convenient to have child 0 for "not a letter"
- Case insensitive: child 1 is either 'a' or 'A'
- Make a "lookup string"
  
  ```python
  alphabet = ' ' + ' '.join(map(chr, \
      range(ord('a'), ord('z') + 1)))
  ```
- Does a given number correspond to a letter?
  
  ```python
  def isLetterNumber(n):
      return 1 <= n and n < len(alphabet)
  ```
- Convert integers to a lowercase character or the empty string
  
  ```python
  def character(n):
      return alphabet[n] if isLetterNumber(n) \ 
      else ''
  ```
- Reverse map uses "ASCII arithmetic"
  
  ```python
  def number(c):
      n = ord(c) - (ord('A') if c.isupper() \ 
      else ord('a')) + 1
      return n if isLetterNumberNumber(n) else 0
  ```
A Node in Python

class node:
    def __init__(self, isWord=False):
        self.isWord = isWord
        self.child = [None]*length(alphabet)

• Could have more information in a node
• The information in a node is the node’s value
• A tree is just a node with nodes as children
• Since children have children in turn, each child can be viewed as the root of a subtree
• Recursive definition of tree: A tree is a (possibly empty) value and a (possibly empty) list of trees
• The trees in the list are the subtrees rooted at the children
• Once you implement a node, you have the whole tree
The Spelling Checker: General Idea

• Assume that the dictionary tree is built, call it tree
• Set current to tree initially, so current points to the root
• If you read a white space (one of ', . ; ; ! ? ( ) [] & \ - / @ # \ n \ r "'), reset current to tree
• [No numbers allowed, for simplicity]
• Otherwise, it’s a letter: descend to the corresponding child of current
• Return current.isWord
• [Additional code will recolor the current word appropriately]
A Complication

• “Descend to the corresponding child of current”
• What if the child is None?
• Messy solution: check for that condition and handle it appropriately
• More elegant solution: Replace None with an infinite subtree with isWord equal to False everywhere

```
                  F
                 /
                F...
               /    
              F      F
             /
            ...
```
A Very Small Infinite Tree

class terminal:
    def __init__(self, isWord=False):
        self.isWord = isWord
        self.child = [self]*length(alphabet)

Only works if all nodes are identical
Only One Infinite Tree

class terminal: # Infinite tree
def __init__(self, isWord=False):
    self.isWord = isWord
    self.child = [self]*length(alphabet)

term = terminal() # Single instantiation

class node: # Regular node
def __init__(self, isWord=False):
    self.isWord = isWord
    self.child = [term]*length(alphabet)

def isLeaf(n): # Not used during lookup!
    return all(c is term for c in n.child)
The Spelling Checker

```python
whitespace = ' ' ,.:;!?()[]"'-/@\#\n\r''

def isWhitespace(c): return whitespace.find(c) >= 0

current = tree

def check(c):
    global current
    current = tree if isWhitespace(c) \n        else current.child[number(c)]
    return current.isWord
```

Loading the Dictionary

# words is a list of words, perhaps from a file

def makeTree(words):

    def insert(word, tree):
        nd = tree
        for letter in word:
            k = number(letter)
            if nd.child[k] is term:
                nd.child[k] = node()
            nd = nd.child[k]
        nd.isWord = True

    tree = node()
    for word in words: insert(word, tree)
    return tree
Tree Traversals

- **Traversing** a tree means visiting each node exactly once
- There are several ways to traverse a tree
  - Depth-first traversal: pre-order and post order
  - Breadth-first traversal
- Depth-first is most naturally done recursively
- Breadth-first is most naturally done iteratively
Depth-First, Pre-Order Traversal

```python
def printDepthFirst(tree, word=''):  
    if tree.isWord: print(word)  
    for k in range(1, len(tree.child)):  
        if tree.child[k] is not term:  
            printDepthFirst(tree.child[k], 
            word + character(k))
```

- **word** behaves like a *stack*:
  - Initially empty
  - *Push* one more character onto **word** when going one level deeper
  - **word** contains the letters on the *path* from the root to the current node
  - *Pop* a character from **word** when returning from `printDepthFirst`

- What does this print?

```
be  
bear  
bears  
bee  
bees  
beg  
begin  
begins  
war  
ward  
warden  
was  
wasp  
wasps  
waste  
we  
wear  
wearer  
week  
weeks  
word  
words  
work  
worker
```
Alphabetical Order

This is a lexicographic sorting method
def printDepthFirst2(tree, word=' '):
    for k in range(1, len(tree.child)):
        if tree.child[k] is not term:
            printDepthFirst2(tree.child[k], \
                             word + character(k))
        if tree.isWord: print(word)

• Only change: move the print from first to last line
• Prints on its way back up the tree, rather than on its way down
• What does this print?
• How to print in reverse alphabetical order?
How to Print in Reverse Order?

```python
def printDepthFirst3(tree, word=' '):
    for k in range(len(tree.child) - 1, 0, -1):
        if tree.child[k] is not term:
            printDepthFirst3(tree.child[k],
                word + character(k))
    if tree.isWord: print(word)
```

- Do print last
- and visit children in reverse
- Prints on its way back up the tree and from right to left
- This is still a depth-first traversal
How to Print by Increasing Length?

- First print words at depth 1, if any, then at depth 2, ...
- Depth-first had a natural recursive implementation, because you first print the current root, then all the subtrees (smaller problems)
- Not so for printing one level at a time
- Best done iteratively
- Called *breadth-first traversal*
- Idea
  - Put the root (if not terminal) in a queue
  - While the queue is not empty
    - pop the first node from the queue
    - print the corresponding word if appropriate
    - put all the children of that node at the end of the queue
Breadth-First Traversal

```python
def printBreadthFirst(tree):
    if tree is not term:
        queue = []
        queue.append((tree, ''))
    while len(queue) > 0:
        (node, word) = queue.pop(0)
        if node.isWord: print(word)
        for k in range(1, len(node.child)):
            child = node.child[k]
            if child is not term:
                queue.append((node.child[k],
                              word + character(k)))
```

- The queue ensures that any given level is printed first, then the children, left to right
- What does this print?
- Increasing lengths, and alphabetical for each length