Molly Yang, Jose Castillo, Shannon Lytle, Alex Chen
CS51
Checkpoint 1
4/21/13

Progress
Our team has written the autocorrect algorithm, the scraper, and implemented the trie data structure. The autocorrect algorithm utilizes the Levenshtein algorithm, and offers a more efficient way to do so than recalculating the Levenshtein distance for two new strings every node traversal. The algorithm is still in the debugging process because the implementation of the trie and scraper were recently completed (with the autocorrect algorithm being written in parallel). Within the next few days, the autocorrect algorithm will be completely functional, and work on the autocomplete algorithm will begin.

Problems
None

Teamwork
Each team member has taken ownership of one aspect of the project. Molly has implemented the autocorrect algorithm, Jose has implemented the trie data structure, and Shannon has implemented the scraper. Alex will implement the auto-complete alrogithm for the following checkpoint.

Plan
Alex will implement the autocomplete algorithm for the second checkpoint. Shannon, Molly, and Jose will completely debug their portions already written. Shannon will begin working on extensions as well, while the entire team will complete those the final week.

```
#!/usr/bin/python

import sys
import scrape

# import trie of words
trieDict = scrape.DICT

print trieDict

maxDistance = 2

# declare word, counter class
class wordCounter(object):
    def _init_(self, word, counter):
        self.word = word
        self.counter = counter

# traverses trie and calculates distance on each word
# returns a list of words within given distance along w/ their probability
def distance(userWord, trieDict, lastRow, returnList):

    for eachNode in trieDict:
# create a row for this letter, name it currentRow
        currentRow = []
# if at currentRow[0], initialize value as lastRow[0] + 1
        current[0] = lastRow[0] + 1

        trieWord = getNode.getString

        # store last char of string
        currentLetter = trieWord[-1:]
# for each letter in userWord, letter j, go to currentRow[j]
        for i in range(1, (str.count(userWord))):

            if userWord[i] == currentLetter:
                cost = 0
            else: cost = 1

    # currentRow[i] = min of (box to left + 1, box above + 1,
# and box to top diagonal left + cost)
            currentRow[i] = min((currentRow[i-1] + 1), (lastRow[i] + 1),
                (lastRow[i-1] + cost))

    # if at end of userWord
            if eachNode.getIsWord and currentRow[i] <= maxDistance:
# then append word + counter to returnList
                newWord = wordCounter(trieWord, eachNode.getCounter)
                returnList.append(newWord)

                # get new trie
                newTrie = eachNode.edges
            else: distance(userWord, newTrie, currentRow, returnList)
```

```
# calculates most probable word out of words within given distance
# fed a (word, counter) list
# returns word
def mostProbable(returnList):

    if len(returnList) == 1:
      return returnList[0].word

    # iterate thru list
    for i in range(len(returnList)):
# if word1 counter > word2 counter
      if returnList[i].counter > returnList[i+1].counter:
        del returnList[i+1]
        mostProbable(returnList)
# !!! elif returnList[i].counter == returnList[i+1].counter:
      else:
        del returnList[i]
        mostProbable(returnList)


    # !!! random number generator to make this less arbitrary?


# compiler function - autocorrect
def autocorrect():

    #get user word
    userWord = sys.argv[1]

    # initialize first row, from 0 to len(userWord)
    lastRow = list(xrange(len(userWord) + 1))

    # returnList = initialize empty (word, counter) list
    returnList = []

    # initialize most probable list
    probableList = []

    mostProbable((distance(userWord, trieDict, lastRow, returnList)))

print(autocorrect())
```

```
# trie Node
class Node():

# Constructor for node
  def __init__(self,string):
    self.string=string
    self.edge = {}
    self.counter = 0
```

```
    self.isWord = False

# Get is real word
  def getIsWord(self):
    return self.isWord

# Set is real word
  def setIsWord(self, bool):
    self.isWord = bool

# Get frequency counter for a node
  def getCounter(self):
    return self.counter

# Set frequency counter for a node
  def setCounter(self, count):
    self.counter = count

# Get string of node
  def getString(self):
    return self.string

# Get edge of node
  def getEdge(self):
    return self.edge

# Set next node
  def setNext(self,char,node):
    self.edge[char]=node

# Get next node
  def getNext(self,char):
    if (char in self.edge):
      return self.edge[char]
```

```
#Scraper

import re
import trie

def scrape (file):
    f = open(file, 'r')
    return re.findall('[a-z]+', f.read())

def add_all(strings):
    t = trie.Trie(strings)
    return t
    #for s in strings:
      #
        #if isWord (s, t):
        #    setFreq(t, s, 1)
        #else: insert(t, s)
```

```
    #   return t

DICT = add_all(scrape("test.txt"))
```

```
from node import Node
# the trie
class Trie():

# Constructor
  def __init__(self,words):
    self.startNode=Node("")
    for word in words:
      self.insert(word)

# test membership includes substrings of words
# that are not real words
  def isMember(self,string):
    currNode=self.startNode
    for letter in string:
      if(not currNode.getNext(letter)):
        return False
      else:
        currNode=currNode.getNext(letter)
    return True

# test if word is a real word (word that was inserted or initialized)
  def isRealWord(self,word):
    if (self.isMember(word)):
      return self.getNode(word).getIsWord()
    else:
      return False

  # returns the node searched or an empty node not in the trie
  # containing string "Node Not Found"
  def getNode(self,word):
    if (self.isMember(word)):
      currNode=self.startNode
      counter = 0
      for letter in word:
        currNode=currNode.getNext(letter)
        if (counter == len(word)-1):
            return currNode
        counter = counter + 1
    else:
      return Node("Node Not Found")

  # insert a word into the trie
  def insert(self,word):
    currNode=self.startNode
    for char in word:
```

```
    prevNode=currNode
    if(not prevNode.getNext(char)):
      currNode=Node(prevNode.getString()+char)
      prevNode.setNext(char,currNode)
    else:
      currNode=prevNode.getNext(char)
  currNode.setIsWord(True)

# update frequency of a word if in trie
# returns true on success / false if word not found
def setFreq(self,word, freq):
  if (self.isMember(word)):
    currNode=self.startNode
    for char in word:
      currNode=currNode.getNext(char)
      currNode.setCounter(currNode.getCounter() + freq)
    return True
  else:
    return False

# return a list of (letter,frequency) tuples [(letter, freq), ...]
def getFreq(self,word):
  currNode=self.startNode
  freqList = []
  for char in word:
    currNode=currNode.getNext(char)
    freqList.append((currNode.getString(), currNode.getCounter()))
  return freqList
```