Patrick Trinkle
CMSC676
Spring 2009
Project 1 Write-Up

Background

Information Retrieval systems need to index a corpus to provide users the ability to query for documents based on terms. These documents are not necessarily basic text files, but can also be PDF documents, emails, etc. Therefore an intelligent IR system needs to have methods for identifying and handling these varying document formats. This indexing does not need to be extremely fast; as long as it is accurate and precise. An important point in this is that the documents returned should reflect the goal of the query itself. I designed a document parser which strictly tokenizes Hyper Text Markup Language formatted documents, or basic web pages.

Implementation

Ruby is an entirely Object-Oriented programming language. Having no real experience using it; I felt it a good sandbox for working on the document parser. As a modern scripting language it automatically handles certain objects in a convenient way; such as strings and file streams. Although it is Object-Oriented in nature, it does have support for procedural methods.

For this implementation of the tokenizer all tokens are stored in entirely lowercase. Ruby provides casing methods for string objects; therefore downcasing is applied to each line in the process, versus each token. The HTML parser does not tokenize all character groupings or words. Special characters in ampersand-semicolon notation are disregarded as meta character information. Originally the script handled these special cases, but it was determined too costly for the limited benefit.

Tokens are built strictly from letters; ignoring numbers and periods; even if they appear in the string itself. Also any term that is hyphenated is broken into two or more pieces. URLs are not cleanly handled by the parser, and are broken into pieces. Instead of skipping over certain characters, anything not a letter or period is considered an end token character. Also when an HTML tag starts, if we were processing a token it is saved off. Some strangeness can appear in the term list because some terms can be parts of PGP keys, md5 hashes, et cetera. Also there is no error checking logic to throw out terms which are likely useless. By breaking on certain characters in tokens the following case is common: "It's" => "It" + "s". Because we do accept periods into a token, they are deleted once a termination character is reached. Therefore the following conversion occurs: "u.s.a." => "usa". Similarly because of the exclusion of periods as termination characters, the parser incorrectly converts "good.neat" => "goodneat". Therefore any sentences separated by a period without a space are merged.

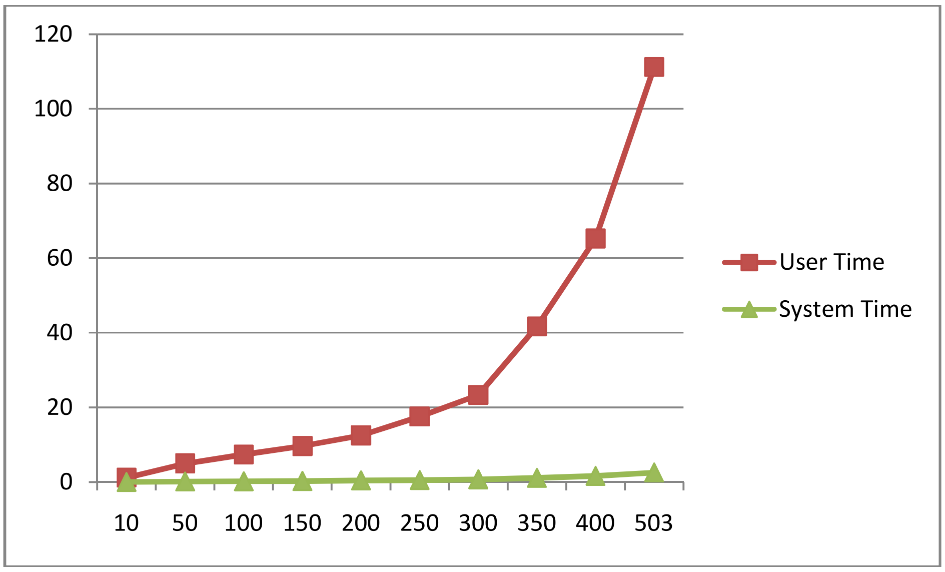
Originally the parser used an object oriented approach whereby each file required instantiating a parser object. This object read in and tokenized the file. This implementation proved rather slow (5 minutes or more) and therefore I switched to a procedural approach. The procedural implementation was still slow. I found that handling special cases as token endings there was a significant slowdown in processing. Also, each line calls block code to handle each byte individually. For all comparisons I had to convert the byte to character. This conversion is not optimized by the interpreter; therefore I had to assign the byte as a character to a character variable and then did the comparisons against the new variable. This detail sliced approximately 90 seconds off the processing time for the 503 input files in the CMSC676 corpus.

Another student approached the problem in a very similar fashion, but with the Python scripting language. In this implementation a stateM object is created to handle parsing of each input file. This object reads in one character at a time. Although a new object is required for each input file, Python is faster. The Python code includes number and characters only and breaks a token on either a whitespace character, new line character or the start of a tag. This can provide better terms if there are meaningful numbers; although since hyphens and periods are not included in terms some values such as IP addresses can lose value as a document term. Also because his tokens just step over non-alphanumeric characters various interesting cases can arise, such as: "It's" => "Its". Many interesting cases provide no detriment to the quality of the term list.

An even faster approach to the HTML tokenizer is to write the processing library in C and then importing into python script. Originally, I wrote the parser in C# and it was very quick as it is entirely in managed system code. This approach provided speed, but was less useful for rapid editing as each incarnation needed compilation.

Conclusion

The other implementation using strictly Python parsed the 503 document corpus in approximately 60 seconds where my Ruby script took 2 minutes. Term frequency was calculated as a simple incrementing of a value in a hash table keyed via the term. Due to the nature of processing in a simple way the time the process takes is directly proportional to the size of the input files; versus the number. This is due to the processing time required per file taking longer than the system IO required for each file.



Time in Seconds by Number of Input Files

Appendix A - Ruby Script

```
#!/usr/bin/env ruby

# Course: CMSC676 - Information Retrieval
# Semester: Spring 2009
# Professor: Dr Nicholas
# Student: Patrick Trinkle <tri1@umbc.ed>
# Date: 20090204
# Title: HTMLParser Object Definition
# Note: This was far trickier than doing it in C# or Perl or C, but has provided a good learning experience
#       Also this method is rather slow.  Doing it as a procedural script might be faster as it won't have to
#       instantiate objects, etc.  But since speed wasn't a primary concern; I felt this approach was
#       appropriate.

#       Okay all special characters within a token (either in the middle or the end) are
#       disregarded: be&#146s => bes
#       Numbers are also ignored.
#       Tokens are terminated by anything that isn't a letter or a '.'; unless it's in a special character code
#       Also note: "good.neat" => "goodneat"

require 'fileutils'

# Ruby doesn't have quite the notion of an enumeration
class HTMLParserState
  InsideTag = 1
  InsideToken = 2
  InsideSpecial = 3
end

def HTMLParserTokenize( filename )
  state = 0
  templine = ""
  termList = Hash.new( 0 )

  file = File.open( filename )

  file.each {
    |line|

    line.downcase.each_byte {
      |c|
      b = c.chr

      case b
        when '<':
          if state == HTMLParserState::InsideToken then
            # end current token
            if templine.size > 0 then
              templine.delete!( '.' )
              termList[templine] += 1
            end

            templine = ""
            state = HTMLParserState::InsideTag
          end
          if state != HTMLParserState::InsideTag then
            state = HTMLParserState::InsideTag
          end
        when '>':
          if state == HTMLParserState::InsideTag then
            state = HTMLParserState::InsideToken
          end # end if insidetag
        when '&':
          # we only go into specialstate if we are inside a token and by that i
          # mean in the middle/end of a token
          if state == HTMLParserState::InsideToken && templine.size > 0 then
            state = HTMLParserState::InsideSpecial
          end # end if state == insidetoken
        when ';':
          if state == HTMLParserState::InsideSpecial then
            # we go back to regular token because we had to have been in this state before special
            state = HTMLParserState::InsideToken
          end # end if state == insidespecial
        else
          if state == HTMLParserState::InsideToken then
            if (b > 'z' || b < 'a') && b != '.' then
              if templine.size > 0 then
                templine.delete!( '.' )
                termList[templine] += 1
              end
              templine = ""
            elsif (b >= 'a' && b <= 'z') then # we currently ignore urls
              templine.concat( b )
            end # end if b.chr == ...
          elsif state == HTMLParserState::InsideSpecial
            if (b == '\n' || b == ' ' || b == '\t' || b == '\r') then
              templine.delete!( '.' )
              termList[templine] += 1
            end
          end #end if state == HTMLParserState::InsideToken
      end #end case
      } #end each byte
    } #end each line

  file.close
  # we now have all the terms for that file

  return termList

end #end tokenize

# Main Execution
# Parse input parameters
directory = ""
index = ""

if ARGV.size != 2 then
  puts "usage: <input file directory> <index directory>"
  exit
else
  directory = ARGV[0]
  index = ARGV[1]
end

# Is there an index directory?
if File.exists? index then
  puts "Index Directory Already Exists"
else
  FileUtils.mkdir( index )
end

files = Array.new

# Glob up the HTML Files in the Directory
dir = Dir.open( directory )
files = dir.to_a
dir.close

# Tokenize each file
i, j = 0, 0
totalTokens = Hash.new( 0 )
temporaryList = Array.new

# We only want .html files
while i < files.size do
  if files[i] !~ /html$/ then
    files.delete_at( i )
    i -= 1
    else
      shortname = String.new( files[i] )
      shortname =~ /(.*?)\.html/
      temporaryList = HTMLParserTokenize( (files[i].insert 0, directory) ).to_a.sort
      outputFile = File.new( index + $1 + ".txt", "w" )
      j = 0
      while j < temporaryList.size do
        totalTokens[temporaryList[j][0]] += temporaryList[j][1]
        outputFile.puts temporaryList[j][0] + " : " + temporaryList[j][1].to_s
        j += 1
      end
      outputFile.close
    puts "Finished: " + files[i] + " Found: " + temporaryList.size.to_s + " Tokens"
   end
  i += 1
end

puts "Total Tokens: " + totalTokens.size.to_s

# Print out Total Terms List sorted by term
puts "Writing out Complete Term List by Term"
outputFile = File.new( index + "TermListSortedByTerm.txt", "w" )
temporaryList = totalTokens.to_a.sort
i =0
while i < temporaryList.size do
  outputFile.puts temporaryList[i][0] + " : " + temporaryList[i][1].to_s
  i += 1
end
outputFile.close

# Print out Total Terms List sorted by frequency
puts "Writing out Complete Term List by Frequency"
outputFile = File.new( index + "TermListSortedByFrequency.txt", "w" )
# reverse sort code: .sort {|a,b| -1*(a[1]<=>b[1]) }
temporaryList = totalTokens.sort {|a,b| a[1] <=> b[1]}.to_a
i = 0
while i < temporaryList.size do
  outputFile.puts temporaryList[i][0] + " : " + temporaryList[i][1].to_s
  #puts temporaryList[i].to_s
  i += 1
end
outputFile.close
```