MIT OpenCourseWare
http://ocw.mit.edu

6.006 Introduction to Algorithms
Spring 2008

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

Lecture 1: Introduction and the Document Distance Problem

Course Overview

- Efficient procedures for solving problems on large inputs (Ex: entire works of Shakespeare, human genome, U.S. Highway map)
- Scalability
- Classic data structures and elementary algorithms (CLRS text)
- Real implementations in Python $\Leftrightarrow$ Fun problem sets!
- β version of the class - feedback is welcome!

Pre-requisites

- Familiarity with Python and Discrete Mathematics

Contents

The course is divided into 7 modules - each of which has a motivating problem and problem set (except for the last module). Modules and motivating problems are as described below:

1. Linked Data Structures: Document Distance (DD)
2. Hashing: DD, Genome Comparison
3. Sorting: Gas Simulation
4. Search: Rubik's Cube $2 \times 2 \times 2$
5. Shortest Paths: Caltech $\rightarrow$ MIT
6. Dynamic Programming: Stock Market
7. Numerics: $\sqrt{2}$

Document Distance Problem

Motivation

Given two documents, how similar are they?

- Identical - easy?
- Modified or related (Ex: DNA, Plagiarism, Authorship)

- Did Francis Bacon write Shakespeare's plays?

To answer the above, we need to define practical metrics. Metrics are defined in terms of word frequencies.

Definitions

1. *Word*: Sequence of alphanumeric characters. For example, the phrase "6.006 is fun" has 4 words.

2. *Word Frequencies*: Word frequency $D(w)$ of a given word w is the number of times it occurs in a document D.

   For example, the words and word frequencies for the above phrase are as below:

   $$\begin{array}{rcccccc} Count: & 1 & 0 & 1 & 1 & 0 & 1 \\ Word: & 6 & the & is & 006 & easy & fun \end{array}$$

   In practice, while counting, it is easy to choose some canonical ordering of words.

3. *Distance Metric*: The document distance metric is the inner product of the vectors $\boldsymbol{D_1}$ and $\boldsymbol{D_2}$ containing the word frequencies for all words in the 2 documents. Equivalently, this is the projection of vectors $\boldsymbol{D_1}$ onto $\boldsymbol{D_2}$ or vice versa. Mathematically this is expressed as:

   $$\boldsymbol{D_1} \cdot \boldsymbol{D_2} = \sum_w \boldsymbol{D_1}(w) \cdot \boldsymbol{D_2}(w) \tag{1}$$

4. *Angle Metric*: The angle between the vectors $\boldsymbol{D_1}$ and $\boldsymbol{D_2}$ gives an indication of overlap between the 2 documents. Mathematically this angle is expressed as:

   $$\theta(\boldsymbol{D_1}, \boldsymbol{D_2}) = \arccos\left(\frac{\boldsymbol{D_1} \cdot \boldsymbol{D_2}}{\parallel \boldsymbol{D_1} \parallel * \parallel \boldsymbol{D_2} \parallel}\right)$$

   $$0 \leq \theta \leq \pi/2$$

   An angle metric of 0 means the two documents are identical whereas an angle metric of $\pi/2$ implies that there are no common words.

5. *Number of Words in Document*: The magnitude of the vector $\boldsymbol{D}$ which contains word frequencies of all words in the document. Mathematically this is expressed as:

   $$N(\boldsymbol{D}) = \parallel \boldsymbol{D} \parallel = \sqrt{\boldsymbol{D} \cdot \boldsymbol{D}} \tag{2}$$

So let's apply the ideas to a few Python programs and try to flesh out more.

Document Distance in Practice

Computing Document Distance: docdist1.py

The python code and results relevant to this section are available here**.** This program computes the distance between 2 documents by performing the following steps:

- Read file
- Make word list [“the”,“year”,. . .]
- Count frequencies [[“the”,4012],[“year”,55],. . .]
- Sort into order [[“a”,3120],[“after”,17],. . .]
- Compute θ

Ideally, we would like to run this program to compute document distances between writings of the following authors:

- Jules Verne - document size 25k
- Bobsey Twins - document size 268k
- Lewis and Clark - document size 1M
- Shakespeare - document size 5.5M
- Churchill - document size 10M

Experiment: Comparing the Bobsey and Lewis documents with docdist1.py gives $\theta = 0.574$. However, it takes approximately 3 minutes to compute this document distance, and probably gets slower as the inputs get large.
What is wrong with the efficiency of this program?
Is it a Python vs. C issue? Is it a choice of algorithm issue - $\theta(n^2)$ versus $\theta(n)$?

Profiling: docdist2.py

In order to figure out why our initial program is so slow, we now “instrument” the program so that Python will tell us where the running time is going. This can be done simply using the *profile* module in Python. The *profile* module indicates how much time is spent in each routine.
(See this link for details on *profile*).
The *profile* module is imported into docdist1.py and the end of the docdist1.py file is modified. The modified docdist1.py file is renamed as docdist2.py
Detailed results of document comparisons are available here**.**

More on the different columns in the output displayed on that webpage:

- tottime per call(column3) is tottime(column2)/ncalls(column1)
- cumtime(column4)includes subroutine calls
- cumtime per call(column5) is cumtime(column4)/ncalls(column1)

The profiling of the Bobsey vs. Lewis document comparison is as follows:

- Total: 195 secs
- Get words from line list: 107 secs
- Count-frequency: 44 secs
- Get words from string: 13 secs
- Insertion sort: 12 secs

So the get words from line list operation is the culprit. The code for this particular section is:

```
word_list = [ ]
for line in L:
    words_in_line = get_words_from_string(line)
    word_list = word_list + words_in_line
return word_list
```

The bulk of the computation time is to implement

```
word_list = word_list + words_in_line
```

There isn't anything else that takes up much computation time.

List Concatenation: docdist3.py

The problem in docdist1.py as illustrated by docdist2.py is that concatenating two lists takes time proportional to the sum of the lengths of the two lists, since each list is copied into the output list!
$L = L_1 + L_2$ takes time proportional to $\mid L_1 \mid + \mid L_2 \mid$. If we had n lines (each with one word), computation time would be proportional to $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \theta(n^2)$

Solution:

```
word_list.extend (words_in_line)
[word_list.append(word)] for each word in words_in_line
```

Ensures $L_1.extend(L_2)$ time proportional to $\mid L_2 \mid$

Take Home Lesson: Python has powerful primitives (like concatenation of lists) built in. To write efficient algorithms, we need to understand their costs. See Python Cost Model for details. PS1 also has an exercise on figuring out cost of a set of operations.

Incorporate this solution into docdist1.py - rename as docdist3.py. Implementation details and results are available here. This modification helps run the Bobsey vs. Lewis example in 85 secs (as opposed to the original 195 secs).

We can improve further by looking for other quadratic running times hidden in our routines. The next offender (in terms of overall computation time) is the count frequency routine, which computes the frequency of each word, given the word list.

Analysing Count Frequency

```
def count_frequency(word_list):
    """
    Return a list giving pairs of form: (word,frequency)
    """
    L = []
    for new_word in word_list:
        for entry in L:
            if new_word == entry[0]:
                entry[1] = entry[1] + 1
                break
        else:
            L.append([new_word,1])
    return L
```

If document has n words and d distinct words, $\theta(nd)$. If all words distinct, $\theta(n^2)$. This shows that the count frequency routine searches linearly down the list of word/frequency pairs to find the given word. Thus it has quadratic running time! Turns out the count frequency routine takes more than 1/2 of the running time in docdist3.py. Can we improve?

Dictionaries: docdist4.py

The solution to improve the Count Frequency routine lies in hashing, which gives constant running time routines to store and retrieve key/value pairs from a table. In Python, a hash table is called a dictionary. Documentation on dictionaries can be found here.

Hash table is defined a mapping from a domain(finite collection of immutable things) to a range(anything). For example, D[‘ab’] = 2, D[‘the’] = 3.

Modify docdist3.py to docdist4.py using dictionaries to give constant time lookup. Modified count frequency routine is as follows:

```
def count_frequency(word_list):
    """
    Return a list giving pairs of form: (word,frequency)
    """
    D = {}
    for new_word in word_list:
        if D.has_key(new_word):
            D[new_word] = D[new_word]+1
        else:
            D[new_word] = 1
    return D.items()
```

Details of implementation and results are here. Running time is now $\theta(n)$. We have successfully replaced one of our quadratic time routines with a linear-time one, so the running time will scale better for larger inputs. For the Bobsey vs. Lewis example, running time improves from 85 secs to 42 secs.

What's left? The two largest contributors to running time are now:

- Get words from string routine (13 secs) — version 5 of docdist fixes this with translate
- Insertion sort routine (11 secs) — version 6 of docdist fixes this with merge-sort

More on that next time ...