Assignment 3 - Analysis

Daniel Sundstrom
daniel@sykewarrior.com

I believe in rough consensus and running code.

March 17, 2012

1 Overview

The implementations are done in Java and Ruby respectively, mostly due to time constraints and familiarity with languages. I contemplated using Erlang and Objective-C instead, but external time constraints and ambiguity in the specifications[1] led to my main language Ruby and Java.

- Ruby - 4,5h (Of which 1,5 hours was spent chasing down a rogue reference where a pure object was required)
- Java - 5h

Note however that this is time spent actually implementing code, the author spent a few days thinking about the problem and

2 Implementation

The algorithm is a three-step algorithm - indexing, mapping and reducing the results. The index is a number-keyed lookup table using the longest common subsequence of the dictionary content converted using the provided map. This decision was based on the excellent growth graph of a keyed index of that sort - since the larger the dictionary the more common subsequences. This was verified with running the indexer with various dictionaries, and even on a dictionary with 350000 words generate acceptable sizes on the lookup table.

The map-algorithm steps through the number, checking for existence of the subsequence. If a match is found in the dictionary, it stores the result, splits at the current pivot character and calls itself recursively with the remaining substring. Note that the result stack is kept outside the recursive loop since there's no need to initialize it for all empty results. Results are only added to the list if the total combined subsequence matches the number in length.

Finally, the reducer takes the generated result-list, generates a cartesian product of the found subsequences and builds a map of the result. This map could be avoided but is included for the sake of readability in the code.

[1] I contemplated building a distributed solution in Erlang but was unsure if that was allowed

3 System Language

Java was quite suitable to the problem at hand, mainly due to its parameterized vectors and Iterables. At first the algorithm was using a threaded design, hence the choice of vectors over ArrayLists (since operations on vectors are synchronized), this was changed late in the development process.

Generally speaking were a statically typed language suitable for this task since the program is short enough to have a short compile time and thus a short REPL. This was very useful in the early stages of development. The parameterized lists and maps removes the need for error checking and type-checking in the reducer-step of the algorithm.

However, the program displays one of the main problems with Java - its verbosity. A rough estimate gives about 40-50% of the LOC as boilerplate code for either reading files, initializing variables or curly braces. One might question the need to instantiate four difference classes in order to iterate over the lines in a textfile.

4 Dynamic Language

Ruby has one drawback to this problem, otherwise it was very well suited. The main drawback is performance. Prototyping, tweaking, development and testing was very smooth using Ruby, but when the performance testing began it became apparent that Ruby doesn't have the performance required. Performance testing was done using a dictionary of 500000 words, generated from a variety of english words (to guarantee a natural variation on word length) and a list of 2000000 phone numbers of a length up to 16 digits. While Ruby still finishes the tasks in about 2 minutes[2], that is still almost twice as long as the compiled jar-file from the Java-implementation.

File-handling in Ruby feels very natural, where everything is derived from the basic IO-class and thus provides the same API for reading and writing to. Another one of the strengths of Ruby is string manipulation, which was one of the stated goals of Matz when creating Ruby. It is very neat to be able to handle strings as arrays of characters without having to do explicit conversion.

5 Approach and Design

The problem was first explored in Ruby, both due to being the authors natural weapon of choice and due to working with a small subset of the final data so performance was not an issue. Some of the first implementations used a regular suffix-tree for the index, but size rapidly became a problem (even though searching was ridiculously fast). After that the index was instead built as a trie

[2]On a 1.6 Ghz Core 2 Duo Macbook Air with 4 GB of RAM.

with custom edge generation in order to handle the case of multiple words sharing the same prefix but having a very small difference in the end (which proved very common when using a natural dictionary). During this step, Ruby was invaluable due to both having pre-build implementations of these structures, but allowing for redefinition of the leaf-generation using open classes.

In the end, the author decided to go for a simple lookup table instead of using a more complex data type due to the restrictions placed by the systems language. In order to implement the specific leaf-generation algorithm for the trie in Java, major development work would have been required and tests on natural lists of data showed that the problem with memory consumption while still higher than the author would like was within reasonable limits.

The map-reduce concept, while implemented in a non-standard way and not strictly speaking being a true map-reduce was a neat way of approaching the problem - which is definitely a product of exploring the problem space with Ruby before dropping into Java. Due to its strict typing and comparatively bad support for polymorphism the pattern of map/reduce does not naturally fit into Java best practices and style of coding.

5.1 Language Choice

If the author were to solve the same problem again, Java would probably the language of choice now that a working solution has been designed. It would, however, be interesting to implement solutions to the problem in a weakly-typed functional language other than Lisp - perhaps Javascript in order to implement a true map/reduce-based version.

6 Notes

The *validate.rb* file provided has some minor problem which caused some confusion during the start of development. Firstly, it assumes the result is in lower case when the specification explicitly states that output must be printed as written in the dictionary. Secondly, it runs hideously slow on larger datasets (many, many times slower than the actual generation). Thirdly, it falsely reports duplications of phone numbers from the file as errors. The specification does not state that the list of phone numbers is unique - it should only report numbers duplicated if the same number is printed more than it appears in the list of phone numbers.