CS6999 Programming Assignment 1

Justin Kamerman 3335272

March 16, 2011

Assignment

1. Implement the Aho-Corasick string matching algorithm[1] and test its performance for:

   - 1000 blogs and 100 keywords
   - 2,000 blogs and 100 keywords
   - 4,000 blogs and 100 keywords
   - 8,000 blogs and 100 keywords
   - 16,000 blogs and 100 keywords
   - 32,000 blogs and 100 keywords

   Repeat the experiments for 200 and 400 keywords.

2. Build an inverted index for 10, 000 blogs. Use 10 keywords to query the index to:

   - Locate each keyword and retrieve their corresponding blogs.
   - Show the intersection of every two keywords. For example, retrieve the document ID of the blogs that keywords 1 and 2 have occurred together at least once. Similarly, for keywords 1 and 3, ..., keywords 2 and 3, keywords 2 and 4, ...
   - Similarly, show the intersection of every 3 keywords, 4 keywords, etc.

3. Do a comparative analysis (time to build the index and time to retrieve) of the two methods.

1 Aho-Corasick Algorithm

The Aho-Corasick string matching algorithm[1] is a kind of dictionary matching search algorithm that constructs a finite state machine to scan for a given set of keywords. It is, in effect, a reduced grammar regular expression parser described in [2]. In our search implementation, the finite state machine is constructed given a list of keywords and then the documents of the test corpus are read from secondary storage and fed through the finite state machine. As soon as a document is found to contain at least one instance of a set of search terms, parsing of that document ends.

The documents of the corpus are each stored in a disk file, named for the identifier of the original blog extract. In an attempt to exploit the fact that reading documents from disk is slower than state machine scanning, the scan process uses a thread pool to read and parse multiple documents concurrently. By expanding the thread pool, the implementation is able to achieve higher CPU utilization but the limits of the experiment platform did not accommodate increasing this limit to a point where the thread pool and context switching overhead could be effectively ammortised.

Implementation

The Aho-Corasick algorithm are implemented by a Java program. The only external dependency is on the Apache commons-cli library for parsing commend line options. To that end, the program is operated from the command line, taking options listed in table 1.

Option	Description
-d arg	Document directory
-k arg	Keywords file
-p arg	Thread pool size. Default is 10
-g	Generate DOT visulaization of state machine
-h	Print help message

Table 1: Command line options for Aho-Corasick implementation

The program implements various mechanism to facilitate debugging. Throughout the code, log statements have been added using the Java logging framework. The logging output is controlled for individual classes via the *logging.properties* file which the program read on startup. In addition to logging, code was added to generate Graphviz DOT (described in [4]) output representing the decision tree created. This output can be rendered using Graphviz dot tool. An sample *dot* file for the four keywords used in [1] (*he, she, his, hers*), is shown below. The associated image generated by dot in Figure 1.

```
digraph G {
0  [label="0 []", shape=circle];
0 -> 3 [label="s"];
0 -> 1 [label="h"];
```

```
1  [label="1 []", shape=circle];
1 -> 2 [label="e"];
1 -> 6 [label="i"];
1 -> 0 [color="red"];
6  [label="6 []", shape=circle];
6 -> 7 [label="s"];
6 -> 0 [color="red"];
7  [label="7 [his]", shape=circle];
7 -> 3 [color="red"];
2  [label="2 [he]", shape=circle];
2 -> 8 [label="r"];
2 -> 0 [color="red"];
8  [label="8 []", shape=circle];
8 -> 9 [label="s"];
8 -> 0 [color="red"];
9  [label="9 [hers]", shape=circle];
9 -> 3 [color="red"];
3  [label="3 []", shape=circle];
3 -> 4 [label="h"];
3 -> 0 [color="red"];
4  [label="4 []", shape=circle];
4 -> 5 [label="e"];
4 -> 1 [color="red"];
5  [label="5 [she, he]", shape=circle];
5 -> 2 [color="red"];
}
```

The implementation classes and their relationships are represented in a UML class diagram in Figure 2. Following is a brief description of each class:

- **AhoIndexMain**
- **AhoSearchMain**
- **AhoIndexer**
- **AhoSearcher**
- **CombinationGenerator**

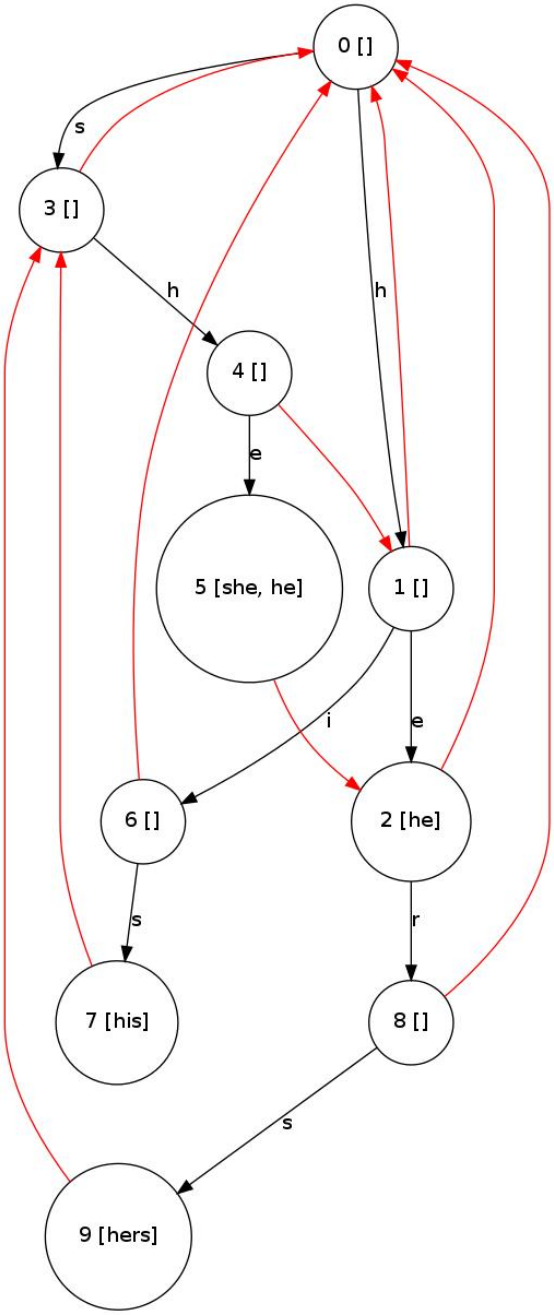

Figure 1: Aho-Corasick state machine visualization generated by DOT

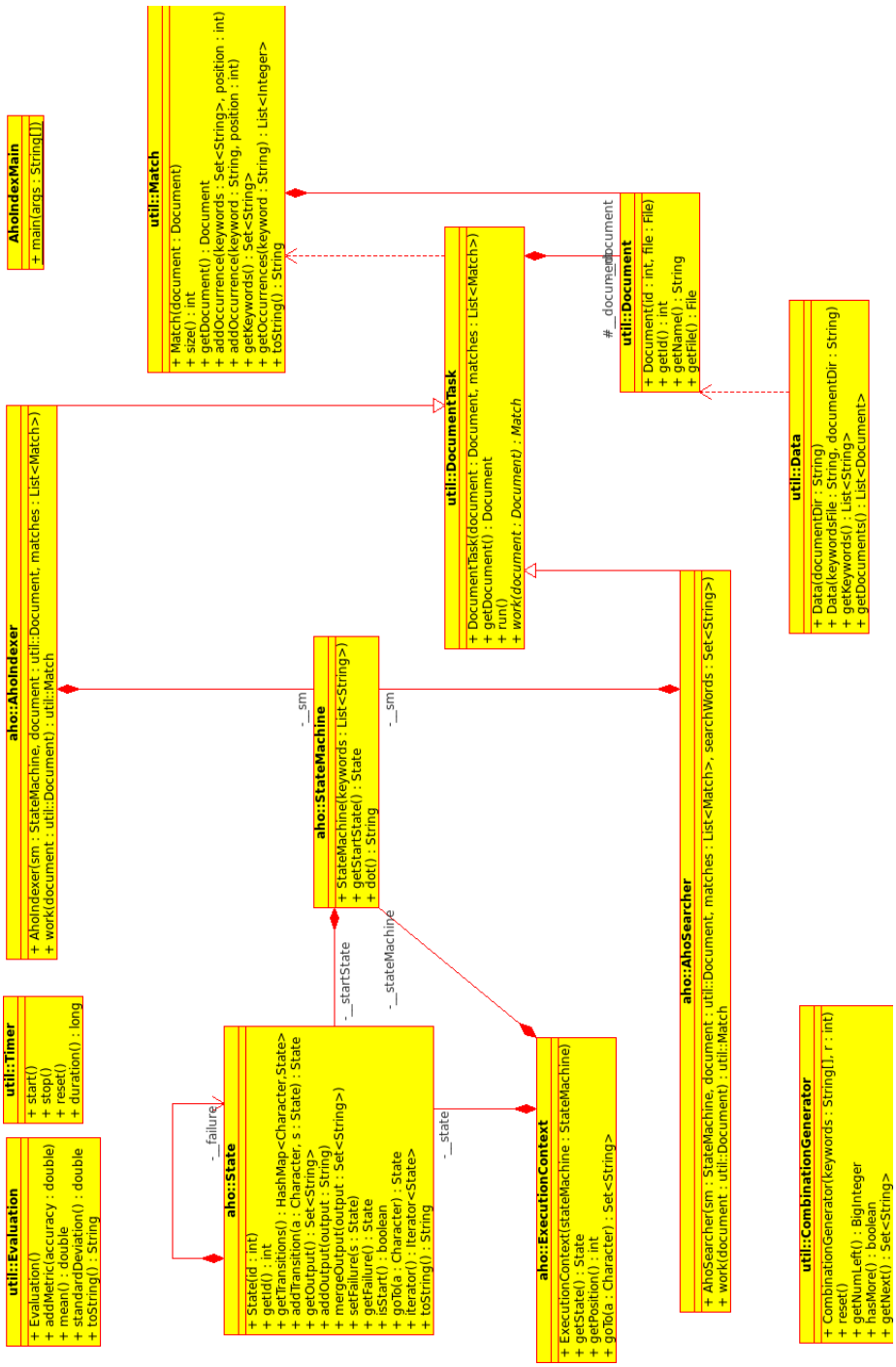

Figure 2: Aho-Corasick Implementation UML class diagram

- **Evaluation**
- **ExecutionContext**
- **StateIterator**
- **StateMachine**
- **Data**
- **Document**
- **DocumentTask**
- **LogFormatter** is a helper class to format log messages.
- **Match**
- **Timer**

Results

Varous tests were conducted to characterise the performance of the Aho-Corasick search algorithm:

- The time taken to construct the Aho-Corasick statre machnie was measured for different numbers of keywords. The results of this test is shown in figure 3. TODO
- The time taken for the Aho-Corasick state machine to scan different sized corpora was measured. The test was repeated for various state machines, contructed using a 100, 200, and 400 keyword set. The results of this test are shown on figure 4. TODO
- The time taken to search for different numbers of keywords was measured over different sized corpora was measured. A set of 10 keywords was selected randomly and the the corpora searched for $\binom{n}{k}$ enumerated combinations thereof to obtain an average for a particular keyword set size. The results of this test are shown in figure 5. TODO

Figure 3: Aho-Corasick state machine construction

Figure 4: Aho-Corasick state machine scanning

2 Inverted Index

The program builds an *inverted index* of a given corpus for boolean search. The implementation is based on the techniques and methods described in [5]. A lexical scanner generator, *JLex*[3], is used to generate a scanner which is used to construct a term index each document. The scanner is configured to ignore a list of 430 stop words. The term indices for each document are post-processed to construct an inverted index mapping each term to a list of documents containing the term.

The documents of the corpus are each stored in a disk file, named for the identifier of the original blog extract. In an attempt to exploit the fact that reading documents from disk is slower than scanning, the scan process uses a thread pool to read and parse multiple documents concurrently. By expanding the thread pool, the implementation is able to achieve higher CPU utilization but the limits of the experiment platform did not accommodate increasing this limit to a point where the thread pool and context switching overhead could be effectively ammortised.

Implementation

The *inverted index* is generated by a Java program. The only external dependency is on the Apache commons-cli library for parsing commend line options. To that end, the program is operated from the command line, taking options listed in table 2.

The program implements various mechanism to facilitate debugging. Through-

Figure 5: Aho-Corasick state machine search

Option	Description
-d arg	Document directory
-p arg	Thread pool size. Default is 10
-h	Print help message

Table 2: Command line options for inverted index implementation

out the code, log statements have been added using the Java logging framework. The logging output is controlled for individual classes via the *logging.properties* file which the program reads on startup.

The implementation classes and their relationships are represented in a UML class diagram in Figure 6. Following is a brief description of each class:

- **InvIndexMain**
- **InvIndexer**
- **AhoSearcher**
- **Data**
- **Document**
- **DocumentTask**
- **LogFormatter** is a helper class to format log messages.
- **Match**
- **Timer**

Algorithm Comparison

All tests were run on a personal computer with an AMD Athlon 64 2GHz Processor, 1GB RAM, running a 32 bit Linux 2.6.31 kernel. The Java Virtual Machine used was version 1.6.0-18. TODO

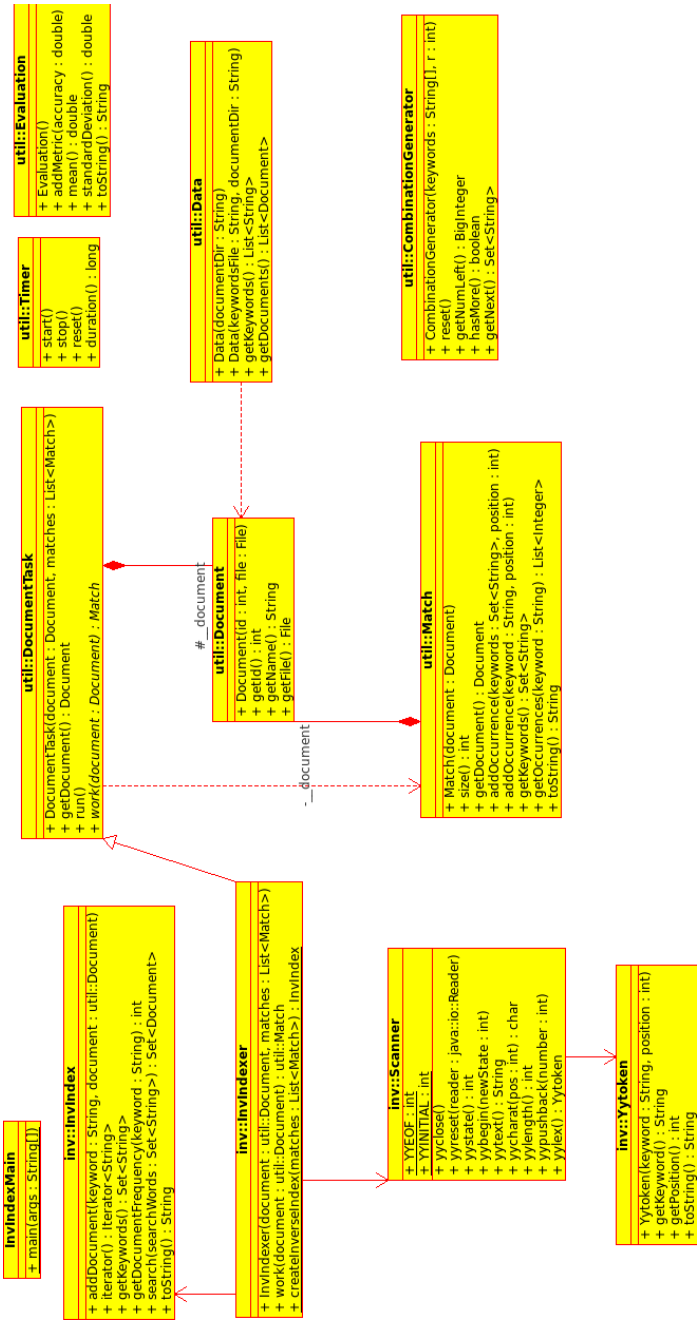

Figure 6: Inverted Index Implementation UML class diagram

Data Set	**Naïve Bayes**	**ID3**	**NBTree 5%**	**NBTree 3%**
car	0.841449	0.938377	0.882783	0.925797
ecoli	0.684776	0.384776	0.674328	0.668657
mushroom	0.952007	1.000000	0.951850	0.996921
letter-recognition	0.737670	0.760870	0.819500	
breast-cancer-wisconsin	0.972662	0.945899	0.972806	0.971223

Table 3: Comparison of Naïve Bayes, ID3, and NBTree accuracy over benchmark data sets. NBTree results for both 3% and 5% split utility thresholds are shown

Bibliography

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun.ACM*, vol. 18, no. 6, pp. 333–340, June 1975.

[2] A. A.V., S. R., and U. J.D., *Compilers. Principles, techniques, and tools*, 1986. [Online]. Available: http://adsabs.harvard.edu/abs/1986cptt.book.....A

[3] E. Berk, *JLex: A Lexical Analyzer for Java.* Department of Computer Science, Princeton University, 2003. [Online]. Available: http://www.cs.princeton.edu/ appel/modern/java/JLex/

[4] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz - open source graph drawing tools," *Graph Drawing*, pp. 483–484, 2001.

[5] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval.* New York, NY, USA: Cambridge University Press, 2008.