CS6999 Programming Assignment 1

Justin Kamerman 3335272

March 17, 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 from a list of keywords. 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 each 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 and marginal improvements in search times. However, the limits of the experiment platform did not accommodate increasing the pool size to a point where the added complexity and synchronization overhead were justified.

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 visualization 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 start-up. In addition to logging, code was added to generate Graphviz DOT[4] output representing the state machine created. A sample *dot* file for the state machine generated for the four keywords used in [1] (*he, she, his, hers*), is shown below. The *output function* for each state is shown in square braces and unlabelled edges represent the *failure function.* The associated image generated by DOT is shown 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:** a driver class for the Aho-Corasick implementation. It performs the scanning tests and gathers data presented in this report. This class manages the thread pool used to parallelize the scanning process.

- **AhoSearchMain:** a driver class for the Aho-Corasick implementation. It performs the search tests and gathers data presented in this report. This class manages the thread pool used to parallelize the search process.

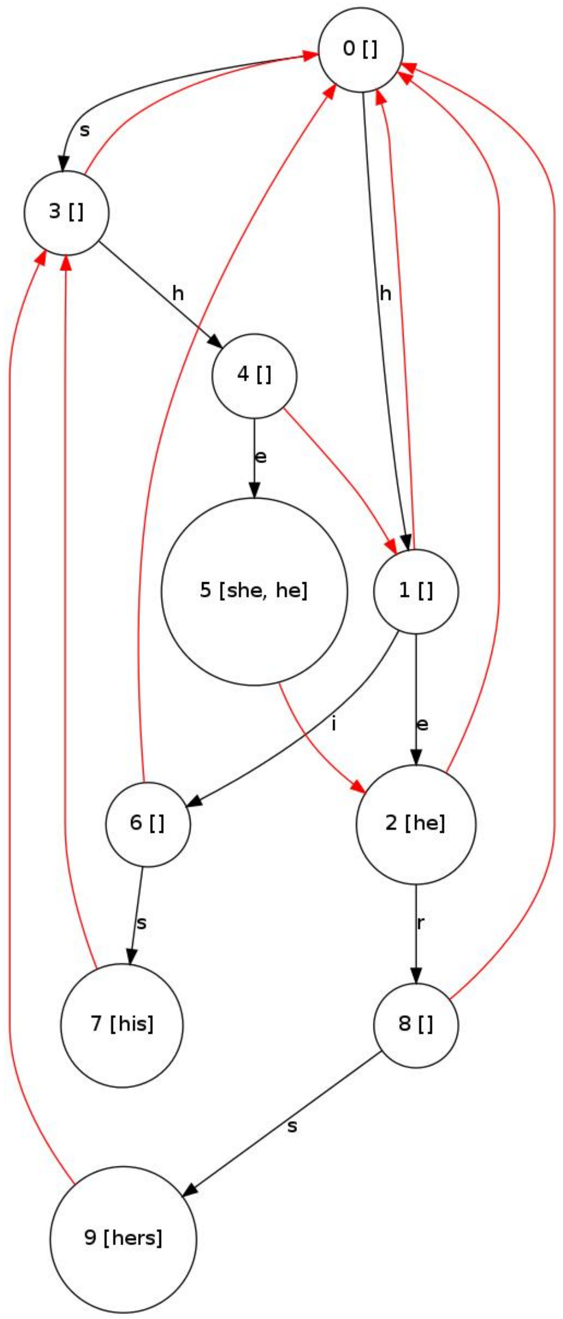

Figure 1: Aho-Corasick state machine visualization generated by DOT

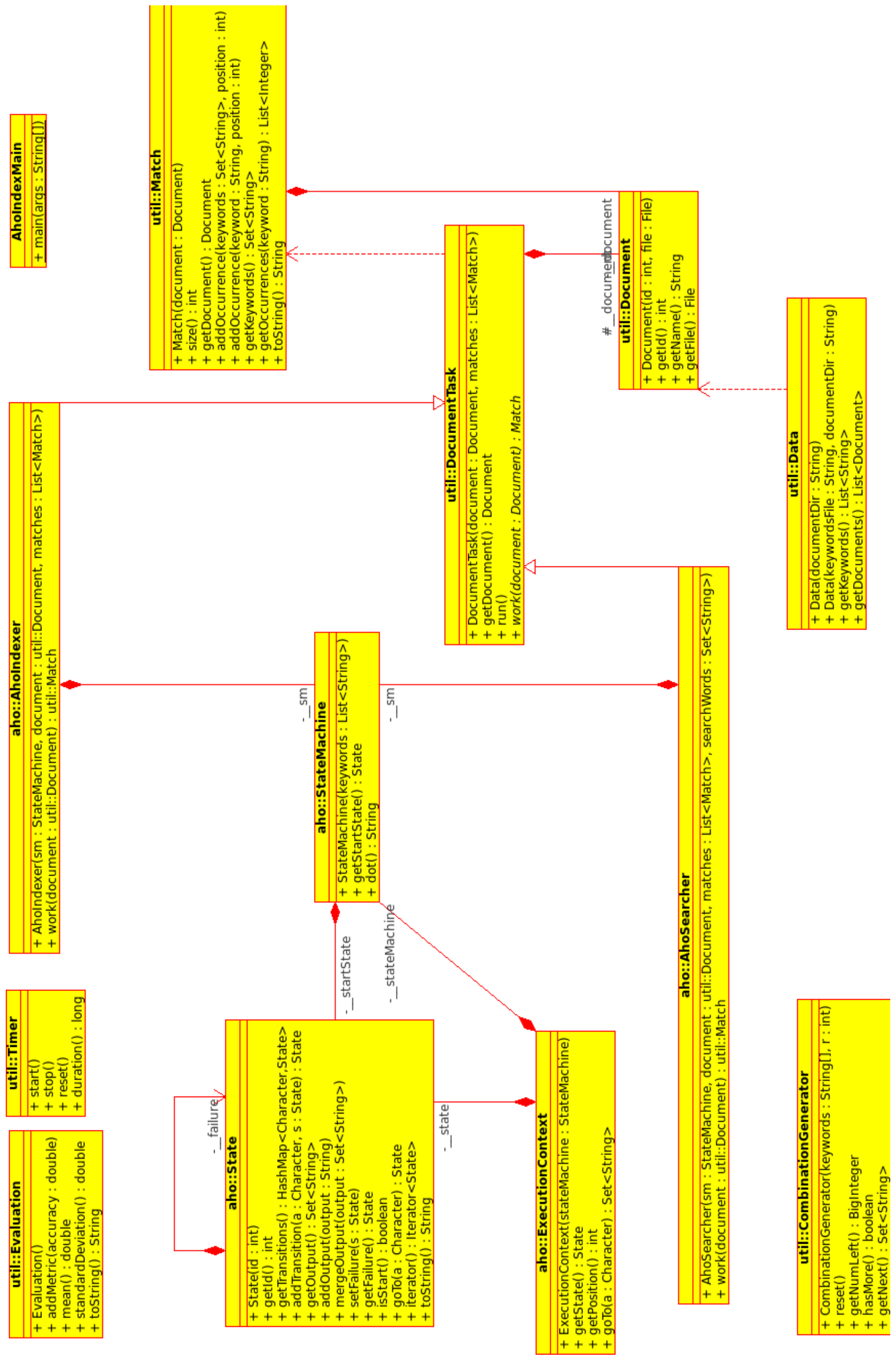
AhoIndexMain
+ main(args : String[])
util::Match
+ Match(document : Document)
+ size() : int
+ getDocument() : Document
+ addOccurrence(keywords : Set<String>, position : int)
+ addOccurrence(keyword : String, position : int)
+ getKeywords() : Set<String>
+ getOccurrences(keyword : String) : List<Integer>
+ toString() : String
aho::AhoIndexer
+ AhoIndexer(sm : StateMachine, document : util::Document, matches : List<Match>)
+ work(document : util::Document) : util::Match
util::DocumentTask
+ DocumentTask(document : Document, matches : List<Match>)
+ getDocument() : Document
+ run()
+ work(document : Document) : Match
util::Document
+ Document(id : int, file : File)
+ getId() : int
+ getName() : String
+ getFile() : File
# _document
-_document
util::Data
+ Data(documentDir : String)
+ Data(keywordsFile : String, documentDir : String)
+ getKeywords() : List<String>
+ getDocuments() : List<Document>
aho::StateMachine
+ StateMachine(keywords : List<String>)
+ getStartState() : State
+ dot() : String
- _sm
- _sm
aho::AhoSearcher
+ AhoSearcher(sm : StateMachine, document : util::Document, matches : List<Match>, searchWords : Set<String>)
+ work(document : util::Document) : util::Match
util::Timer
+ start()
+ stop()
+ reset()
+ duration() : long
util::Evaluation
+ Evaluation()
+ addMetric(accuracy : double)
+ mean() : double
+ standardDeviation() : double
+ toString() : String
- _startState
- _stateMachine
- _failure
aho::State
+ State(id : int)
+ getId() : int
+ getTransitions() : HashMap<Character,State>
+ addTransition(a : Character, s : State) : State
+ getOutput() : Set<String>
+ addOutput(output : String)
+ mergeOutput(output : Set<String>)
+ setFailure(s : State)
+ getFailure() : State
+ isStart() : boolean
+ goTo(a : Character) : State
+ iterator() : Iterator<State>
+ toString() : String
- _state
aho::ExecutionContext
+ ExecutionContext(stateMachine : StateMachine)
+ getState() : State
+ getPosition() : int
+ goTo(a : Character) : Set<String>
util::CombinationGenerator
+ CombinationGenerator(keywords : String[], r : int)
+ reset()
+ getNumLeft() : BigInteger
+ hasMore() : boolean
+ getNext() : Set<String>

Figure 2: Aho-Corasick Implementation UML class diagram

- **AhoIndexer:** manages the scanning of documents by the state machine. It presents an interface suitable for use by the thread pool.
- **AhoSearcher:** manages the searching of documents by the state machine. It presents an interface suitable for use by the thread pool.
- **CombinationGenerator:** this class generates the $\binom{n}{k}$ search term combinations required for the search evaluation. It borrows heavily from source code published by Michael Gilleland at Miriam Park Software.
- **Evaluation:** a utility class for aggregating general test results and calculating means and standard deviations.
- **ExecutionContext:** externalizes an execution path through a `StateMachine` so that the machine can be used concurrently by multiple threads.
- **StateIterator:** utility class for iterating through all states of a `StateMachine`. Used in generating DOT visualization output.
- **StateMachine:** represents the finite state machine created by the Aho-Corasick string matching algorithm[1].
- **Data:** utility class for reading blog and keyword files from the filesystem.
- **Document:** encapsulates document attributes.
- **DocumentTask:** abstract class which enforces an interface suitable for use by the thread pool. Subclasses include `AhoSearcher` and `AhoIndexer`.
- **LogFormatter:** is a helper class to format log messages.
- **Match:** maps a `Document` to a set of keywords that occur in the document and their position with the document.
- **Timer:** utility class for collecting execution times.

Results

Various tests were conducted to characterize the performance of the Aho-Corasick search algorithm:

- The time taken to construct the Aho-Corasick state machine was measured for different numbers of keywords. The results of this test is shown in figure 3 and construction time can easily be considered linear with respect to the number of keywords. This is consistent with [1] which proves that the state machine construction algorithm is linearly proportional to the sum of the lengths of the keywords used to construct the state machine.

- The time taken for the Aho-Corasick state machine to scan different sized corpora was measured. The test was repeated for various state machines, constructed using a 100, 200, and 400 keyword set. The results of this test are shown on figure 4 and indicate that scan time is nearly linear with respect to the size of the corpus and independent of the number of keywords used to generate the state machine. This last result is consistent with [1] which proves that the number of state transitions involved in processing an input string is independent of the number of keywords used to construct the state machine.

- The time taken to search for different numbers of keywords was measured over different sized corpora. A set of ten 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 and indicate that search time is independent of the number of terms used in the search and linear with respect to the size of the corpus. This first result is surprising given that the search process exists as soon as a single instance of each search term is found. One would expect the number of documents scanned completely to increase as the number of search term increases. The small average document size and the fact that the search hit rate is low may account for this phenomenon.

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 to construct a term index each document. The scanner is configured via a set of regular expression macros to ignore a list of 430 stop words. The term occurrences for each document are post-processed to construct an inverted index mapping each term to a list of documents containing the term.

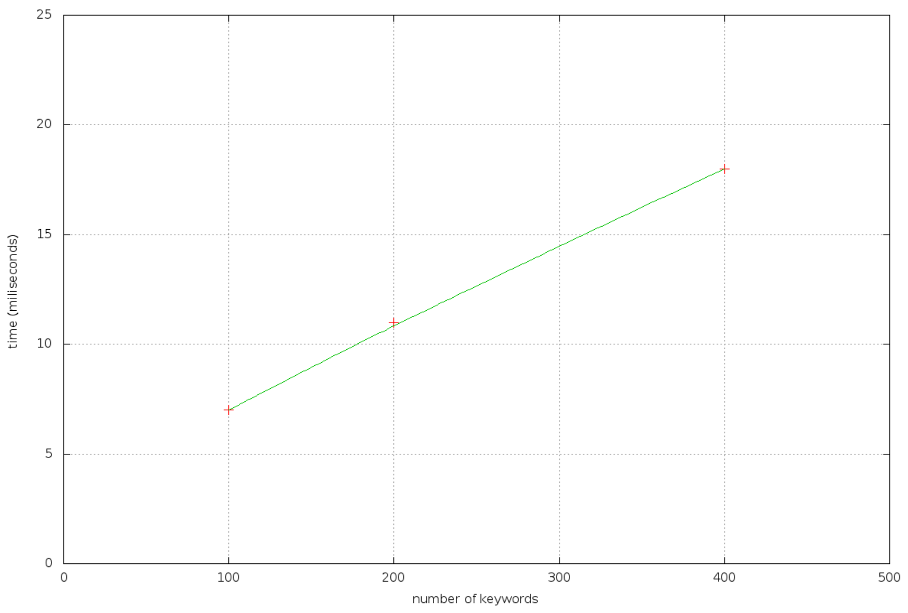

Figure 3: Aho-Corasick state machine construction

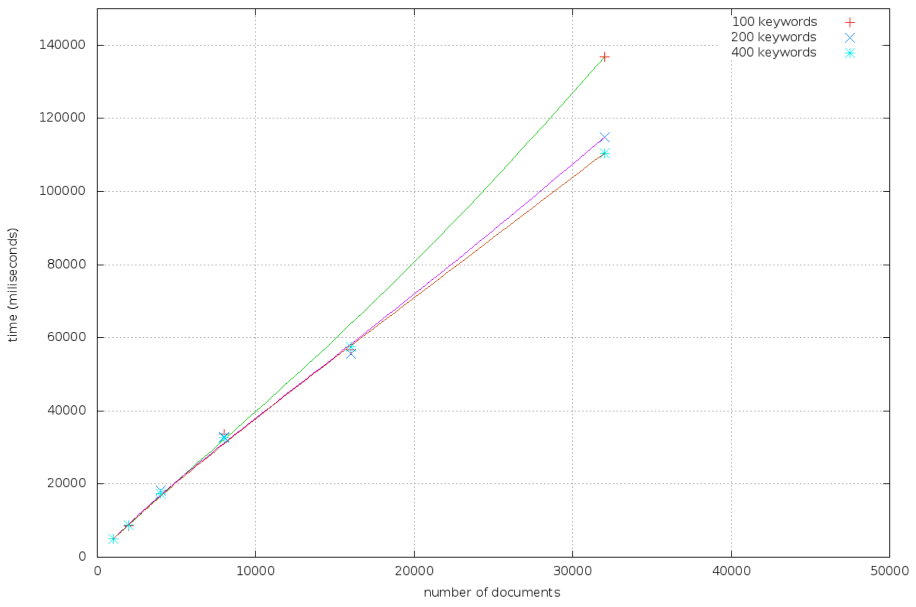

Figure 4: Aho-Corasick state machine scanning

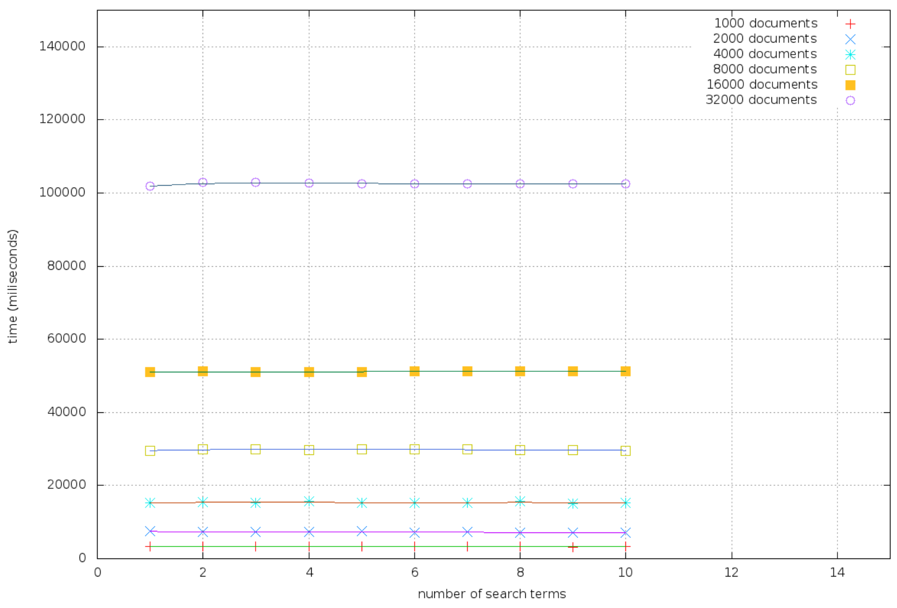

Figure 5: Aho-Corasick state machine search

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 index creation 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 and marginal improvements in index construction times. However, the limits of the experiment platform did not accommodate increasing the pool size to a point where the added complexity and synchronization overhead were justified.

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.

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

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 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:

- **CombinationGenerator:** this class generates the $\binom{n}{k}$ search term combinations required for the search evaluation. It borrows heavily from source code published by Michael Gilleland at Miriam Park Software.

- **Evaluation:** a utility class for aggregating general test results and calculating means and standard deviations.

- **InvIndexMain:** a driver class for the inverted index implementation. It performs the scanning tests and gathers data presented in this report. This class manages the thread pool used to parallelize the scanning process.

- **InvIndexer:** manages the scanning of documents. It presents an interface suitable for use by the thread pool.

- **Data:** utility class for reading blog and keyword files from the filesystem.

- **Document:** encapsulates document attributes.

- **DocumentTask:** abstract class which enforces an interface suitable for use by the thread pool. `InvIndexer` is a subclass.

- **LogFormatter:** is a helper class to format log messages.

- **Match:** maps a `Document` to a set of keywords that occur in the document and their position with the document.

- **Scanner**: lexical scanner class generated by JLex based on regular expression macros defining the stop word list and acceptable word boundaries.

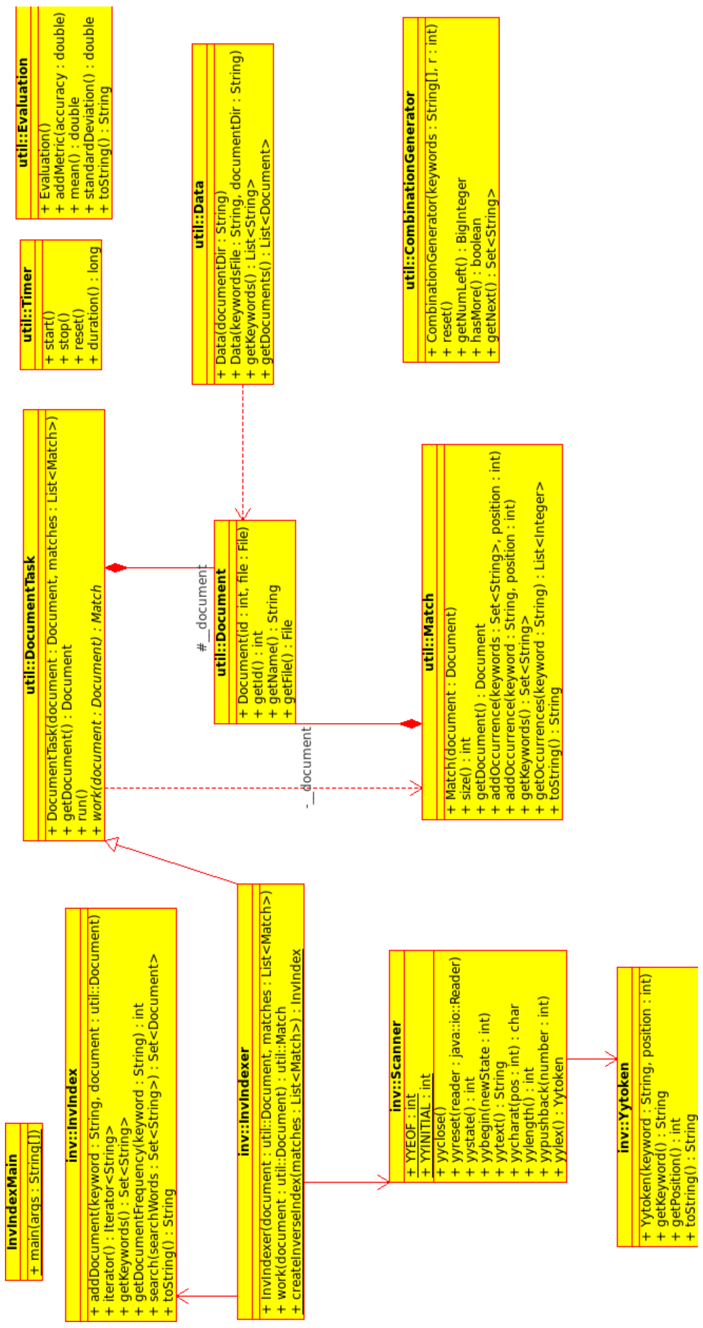
util::Evaluation
+ Evaluation()
+ addMetric(accuracy : double)
+ mean() : double
+ standardDeviation() : double
+ toString() : String
util::Timer
+ start()
+ stop()
+ reset()
+ duration() : long
util::Data
+ Data(documentDir : String)
+ Data(keywordsFile : String, documentDir : String)
+ getKeywords() : List<String>
+ getDocuments() : List<Document>
util::CombinationGenerator
+ CombinationGenerator(keywords : String[], r : int)
+ reset()
+ getNumLeft() : BigInteger
+ hasMore() : boolean
+ getNext() : Set<String>
util::DocumentTask
+ DocumentTask(document : Document, matches : List<Match>)
+ getDocument() : Document
+ run()
+ work(document : Document) : Match
#_document
util::Document
+ Document(id : int, file : File)
+ getId() : int
+ getName() : String
+ getFile() : File
-_document
util::Match
+ Match(document : Document)
+ size() : int
+ getDocument() : Document
+ addOccurrence(keywords : Set<String>, position : int)
+ addOccurrence(keyword : String, position : int)
+ getKeywords() : Set<String>
+ getOccurrences(keyword : String) : List<Integer>
+ toString() : String
InvIndexMain
+ main(args : String[])
inv::InvIndex
+ addDocument(keyword : String, document : util::Document)
+ iterator() : Iterator<String>
+ getKeywords() : Set<String>
+ getDocumentFrequency(keyword : String) : int
+ search(searchWords : Set<String>) : Set<Document>
+ toString() : String
inv::InvIndexer
+ InvIndexer(document : util::Document, matches : List<Match>)
+ work(document : util::Document) : util::Match
+ createInverseIndex(matches : List<Match>) : InvIndex
inv::Scanner
+ YYEOF : int
+ YYINITIAL : int
+ yyclose()
+ yyreset(reader : java::io::Reader)
+ yystate() : int
+ yybegin(newState : int)
+ yytext() : String
+ yycharat(pos : int) : char
+ yylength() : int
+ yypushback(number : int)
+ yylex() : Yytoken
inv::Yytoken
+ Yytoken(keyword : String, position : int)
+ getKeyword() : String
+ getPosition() : int
+ toString() : String

Figure 6: Inverted Index Implementation UML class diagram

- **Timer:** utility class for collecting execution times.
- **YyToken:** class returned by the `Scanner` when a token is found. This implementation bridges the gap between the JLex tokens and the `Match` class used to generate the inverted index.

Results

Various tests were conducted to characterize the performance of the inverted index search algorithm:

- The time taken to construct the inverted index was measured for different sized corpora. The results of this test is shown in figure 7 and, as expected, indicate a linear relationship between index construction time and the size of the corpora.
- The time taken to search for different numbers of keywords was measured over different sized corpora. A set of ten 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 8. If the results for one and ten search words are ignored, search times appear to be independent of the number of search terms and close to linear with respect to the size of the corpus. The unusual results for one and ten keywords warrant further investigation.

Algorithm Comparison

All tests were run on a Dell laptop with an Intel Core Duo 2GHz processor, 1GB RAM, running a 32 bit Linux 2.6.37 kernel. The Java Virtual Machine used was version 1.6.0-24.

Comparing corpus scan times for Aho-Corasick and inverted index implementations, figures 4 and 7 respectively, indicates that the inverted index scanner is significantly faster than the Aho-Corasick state machine. Given that the grammar supported by the JLex-generated scanner is more expressive than that of the Aho-Corasick state machine, one would expect results to more comparable. Reviewing the source code of the generated scanner indicates a compact and efficient implementation using packed data structures to represent the internal state machine. In comparison, the Aho-Corasick implementation is relatively unoptimized, closely following the pseudo-code in [1].

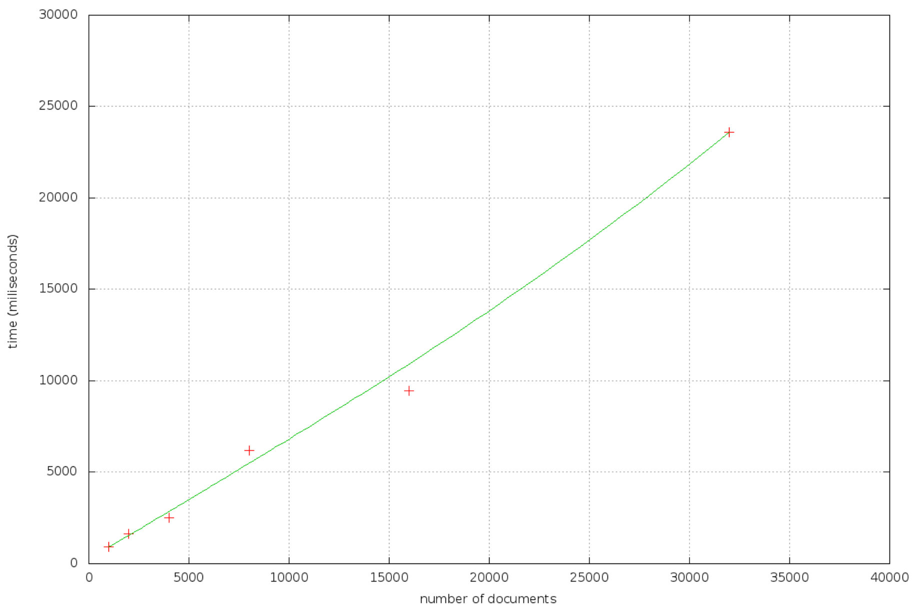

Figure 7: Inverted index creation

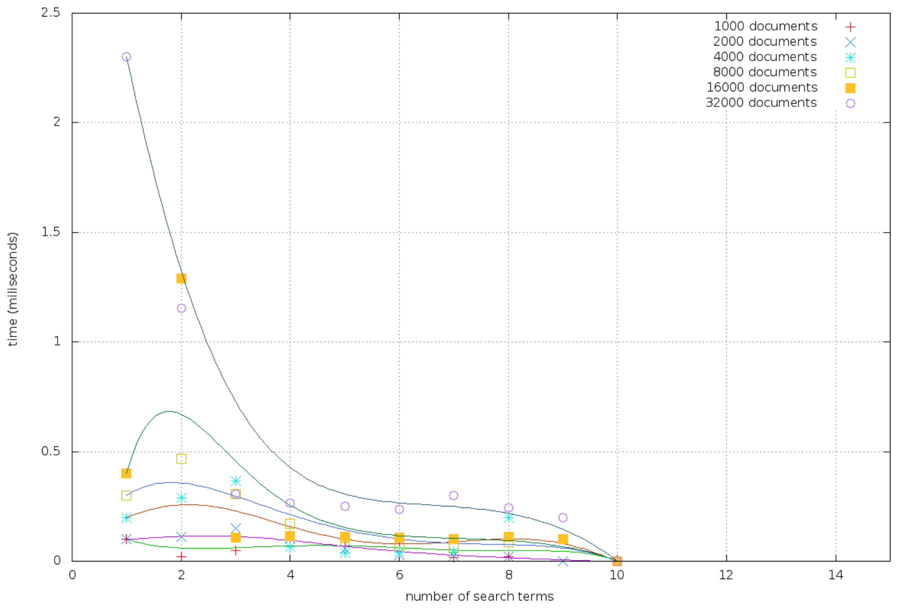

Figure 8: Inverted index search

Comparing search times for Aho-Corasick and inverted index implementations, figures 5 and 8 respectively, indicates that the inverted index implementation has an advantage of several orders of magnitude. The initial cost of creating the inverted index is significantly larger than the creation of the Aho-Corasick state machine however this cost is amortized over each subsequent search. The Aho-Corasick implementation rescans the entire corpus for each search, as opposed to a scan of the inverted index alone. This exercise is a classic demonstration of the impetus behind using indexed search techniques for large corpora.

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.