My Simple Search

Designer’s Manual

Table of Contents

Introduction

This document records the design and working of the 'MSS' software. MSS stands for **My Simple Search**. It is a simple software that can be used to search within text files on a system.

Software Platform:	GNU/Linux
Development Platform:	Red Hat Linux
Programming Language:	C++
Libraries used:	C++ Standard Library

Overview

This document attempts to provide a broad overview of the design of the MSS software. It starts by describing the various components that make up the system and how they interact with each other. This is followed by a description of some of the important data structures used. The organization of the code and pointers to how change it for improvements/bug fixes is described next. Some of the limitations found in the current design are then described. Bugs that were discovered during the coding and not yet corrected are described thereafter. Finally, some possible improvements that have been thought of for future are described. This section also describes some design designs that were discussed during the development of the software.

What this document does not discuss, is the code itself. Please refer to the Source code manual for this.

Specification

Following is a summary of the requirements from the Requirements Specification

1. **Keyword search.** Search for a given keyword and return the set of documents containing the keyword. Rank the query results based on how frequently the keyword has appeared in the documents.

2. **Case-insensitivity.** Keyword searches should be case insensitive.
3. **Logical operators.** Define logical operators AND and OR that can be used to compose a complex query.
4. **Phrase matching.** Phrases that are enclosed in quotes should match exactly.
5. **Stemming.** An asterisk (*) at the end of a keyword should match all endings of the word.
6. The **result** of a query should be a set of path names ordered in descending order of the number of occurrences of the keywords.

All the specified requirements have been met. Infact, the software does more than this.

- The logical operators can be cascaded to any level
- Phrases of any length are matched
- Any or all of logical operators, phrase matching and stemming can all be used in the same search allowing for very varied and powerful searches.
- The number of occurrences of the search string in a path can optionally be given along with the paths.
- Different indexes can be created for different directories and any of them can be used for a search.

<u>Logical Operators</u>:

MSS understands two logical operators: AND and OR. Both of these are binary and act on two search strings as input. Note that, a word in itself is a search string.

<u>AND</u>:

The AND operator, when used, limits the search results to the paths having occurrence(s) of both the search strings used.

mss word1 AND word2:

returns paths of files containing both the words *word1* and *word2*.

This is simply the intersection of the set of results for the first and second search strings. Infact, this is how the result is computed when AND operator is used.

<u>OR</u>:

The OR operator, when used, gives the search results as paths to the files having occurrence(s) of either or both or the search strings used.

mss word1 OR word2

returns paths of files containing either or both of *word1* and *word2*.

This is simply the union of the set of results for the first and second search strings. Infact, this is how the result is computed when OR operator is used.

<u>Phrase matching</u>:

When two or more words are given within the search string in double quotes, search is done to match the occurrence of this exact phrase.

mss “word1 word2”

returns paths of files containing the exact phrase *“word1 word2”*

<u>Stemming</u>:

Stemming refers to deriving words from a given word by appending something to it.

mss ‘word’*

returns paths of files containing words like *word*, *words*, *wording*, *worded* etc. i.e. all words that start with *‘word’*.

<u>Cascading</u>:

The logical operators can be cascaded to any extent i.e. the result of any of the logical operations can be used as an operand to any logical operator. To properly state the meaning of a search string containing more than one logical operator, we state the following:

- *AND and OR both have the same precedence*
- *AND and OR are both left associative*

<u>What this means</u>:

word1 AND word2 AND word3

is equivalent to

((word1 AND word2) AND word3)

word1 OR word2 OR word3

is equivalent to

((word1 OR word2) OR word3)

word1 AND word2 OR word3

is equivalent to

((word1 AND word2) OR word3)

word1 OR word2 AND word3

is equivalent to

((word1 OR word2) AND word3)

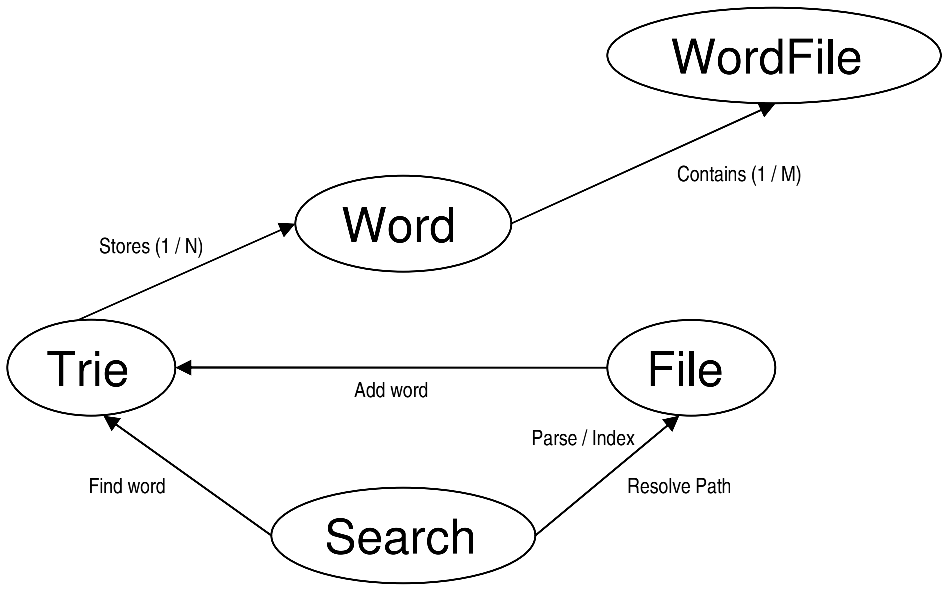
The system

Going through all the files for each search would be very inefficient and quite useless for practical purposes. So, MSS constructs an index of the files and then uses this index to quickly search through the documents. The index creation is a one time job. Then, for all subsequent searches, this index is used.

We have to map each word to a list of files where that word is found. This is called an inverted index. Doing it the other way (mapping files to words in them), called forward index, would lead to a very inefficient solution because this is almost equivalent to reading all the files and looking through them.

Normally, to build the inverted index, the forward index is first built and the inverted index is constructed from it. But this means that the index creation will be a two step process and would be quite slow. So, we decided to build the inverted index directly and skip building of the forward index.

Here is a simple overview of the system:



Trie:

The *Trie* takes care of storing *Word* objects and finding them. This also has the logic to store them on the disk. One instance of this class is created for running the application.

It exposes the following functions:

File:

This *File* exposes static functions that aid in storing and retrieving from file index. It also parses the files for words in them and adds them to the *Trie*.

The following static functions are available:

Word:

The *Word* has information about one word. It contains the word itself and a set of *WordFile* objects.

WordFile:

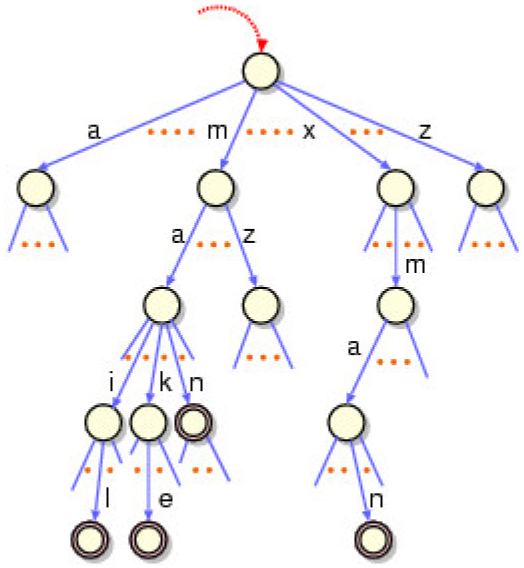
The *WordFile* has information about the occurrence of a word in a file. It contains the file ID and a set of positions in the file.

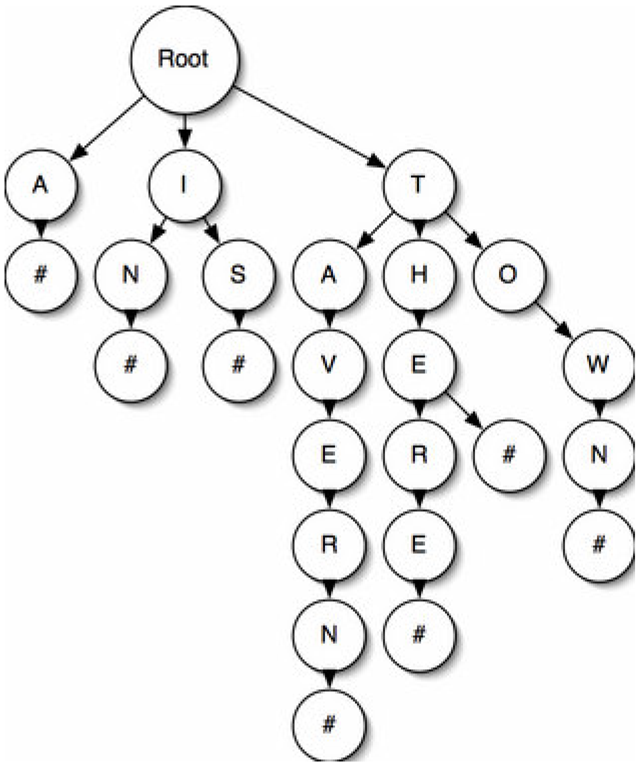
Data structures

The index is stored in files and in what format to do this is a major design decision. The index should be such that it can be created really fast and more importantly, the searches are really fast. Occupying less disk space is a lesser important factor. A *trie* data structure has been used in MSS.

The origin of the name trie is from the middle section of the word "retrieval", and this origin hints on its usage: information retrieval systems. A *trie*, is a multi-way tree structure useful for storing strings over an alphabet.

A trie (general structure):





A simple trie storing
A, IN, IS, TAVERN, THE, THERE, TOWN

We need not store the rest of the pointers.

Though a trie has a higher space complexity, its time complexity is very less (of the order of O(1)). In fact, the search time is of the order of O(l) where l is the length of the word being indexed. Insertion is also O(l) complexity.

Limitations

- The current design limits the possible wild card searches to *stem**.
- The entire index is constructed in memory and then written to disk. For a very large index, this may cause problems.
- The space complexity of the index is quite high.

Bugs

As of the time of writing this document, no bugs were discovered in the software. If you do find any, please inform us about them. And if you manage to fix them, we would be very much happy to accept your fixes.

Improvements

There are possible improvements to the current design that were discussed during the design. Some of these are enumerated here:

- provide for updating the index
- use PATRICIA tries instead of normal tries to improve space efficiency
- store pointer to a word *word1*'s data in both *word1* and *1drow*. This will allow for searches of the type **word1* in addition to *word1**.
- approximate phrase matching can be implemented in the tries.