How WCopyfind and Copyfind Work.

User Interface versus Core Code

Both programs share a common core code for loading, hashing, and comparing documents. They differ only in their user interfaces. WCopyfind provides a windows-style graphical user interface. Copyfind provides a non-graphical command-line user interface.

Outline of the Document Matching Process

The core code proceeds in two steps. First, it loads and hash-codes all of the documents. Second, it compares pairs of those hash-coded documents and reports those that share common prose.

Loading and Hash-Coding Documents

During the loading and hash-coding step, each document is read one word at a time. Depending on the comparison settings, letter case, punctuation, numbers, and other word characteristics are removed or changed. Each word is then converted into a 32-bit hash code—a simplified representation of that word that allows for efficient storage and comparison. Each word, regardless of length or character set is converted into a 32-bit number or "hash code." Hash codes are not unique—two different words can map to the same 32-bit hash code. However there are over 4 billion different 32-bit hash codes, so the chances that two different words in a given language will map to the same hash code are quite small.

Actually, each "word" may also include punctuation and numbers that are located between or adjacent to the letter characters. Thus the sentence "**He wrote paper1, however, he didn't write paper2.**" contains the following eight word-items: "**He**" "**wrote**" "**paper1,**" "**however,**" "**he**" "**didn't**" "**write**" "**paper2.**". Each of those word-items will be converted to a hash code. The hash code for "**paper1,**" will be different from the hash code for "**paper2.**".

If you tell the program to ignore all punctuation, outer punctuation, numbers, or letter case, then each word item will be stripped of the requested characteristic prior to hash coding. For example, ignore outer punctuation will case "**paper1,**" to become "**paper1**" prior to hash coding. Ignore all punctuation will cause "**didn't**" to be converted to "**didnt**" prior to hash coding. Ignore numbers will cause both "**paper1,**" to become "**paper,**" and ignore letter case will cause "**He**" to become "**he**". Turning on multiple ignores will cause multiple conversions, such as turning "**paper1,**" and "**paper2.**" both into "**paper**".

In this manner, each document is converted into a document-ordered list of 32-bit hash codes, one code for each word in the document and listed in the order that those words appear in the document.

But the program needs a second list of hash codes, one that is in numerical-order rather than document-order. So the document-ordered list of hash codes is duplicated and each hash code has its document-order word number attached to it. This second list of hash codes is then sorted numerically to obtain the numerical-ordered list of 32-bit hash codes, one code for each word in the document, but listed in the numerical order of those hash codes themselves. Even though this list is not in the document-order, each hash code in this list has its document-order word number attached to it. Having that numerical-ordered list of the hash codes in the document is equivalent to having an alphabetized lists of the words in the document. Either of those logically ordered lists allows for very efficient comparisons between documents.

Comparing Documents

Once all the documents have been loaded and hash coded, the comparison step occurs. This step takes each pair of documents that needs comparison and works through it carefully, looking for matching phrases.

The comparison process focuses primarily on the two numerical-ordered lists of hash codes. The program has a pair of counters, one for each document's numerical-ordered list of hash codes. The program advances those counters through the numerical-ordered lists until it comes to a pair of identical hash codes—representing a matching pair of words. Since the hash code lists are in numerical order, the program finds those matching pairs quickly and efficiently. Again, having those numerically ordered lists of hash codes is equivalent to having alphabetized lists of the words in the documents. The program makes only one pass through each of those numerical-ordered lists and doesn't have to hunt around through those lists in order to find matching pairs.

Whenever the program finds a pair of matching hash code (words) in its numerical-ordered lists, the program then shifts its attention over to the document-ordered list of hash codes Attached to each hash code in the numerical-ordered list is the document-order word number, so the program knows exactly where the matching pair of hash codes (words) is in each document. The program then searches forward and backward through the document-order list of hash codes to see if there are matching phrases around those matching hash codes (words).

Depending on the comparison settings, the program may step over imperfections in the matching and thereby extend the matching phrases even further backward and forward through the document-ordered lists of hash codes (words). It will only step over flaws that are not more than "Most Imperfections to Allow" words long. It's ability to handle complicated flaws is limited; if the flaw lengths differ by more than one word between the two documents, it probably won't find the continuing phrase at the other ends of the flaw. Also, each time it encounters a flaw while looking for a continuing phrase, the program checks to see if the percentage of perfectly matching words has dropped below the "Minimum % of Matching Words value. It finds the percentage of perfectly matching words in the sum of perfectly matching words plus flaws, and stops looking for the continuing phrase if that percentage drops below the minimum.

Once it has found the longest available matching phrases, it checks to see if the number of perfectly matching words in those phrases (non-matching words in those phrases are not counted) is at least as large as the "Shortest Phrase to Match" value. If so, it records data about those matching phrases in the document-ordered lists of hash codes. It also excludes the words in those matching phrases from further matching in this pair of documents.

When there are multiple copies of a certain word in one or both of documents, the program checks for matching phrases around each possible pairing of those duplicate words. Because words with three or fewer characters (the, a, and, but) are so common in a typical document, they are not used as the starting points for phrase matching. That rarely causes problems because, as long as there is at least one word of four or more characters in a matching phrase, the phrase match will be found based on that longer word.

The program thus works its way through all of the matches pairs of words in the numerical-ordered lists of hash codes, checking each matching word pair to see if it is part of a matching phrase pair. The program continues work until it reaches the ends of the two numerical-ordered lists of hash codes. The comparison of the document pair is then complete.

Having finished the comparison, the program then checks to see if it has found enough matching words to warrant generating a report. If that report is warranted, the two documents are read in again, one word at a time, and data recorded in the document-ordered lists of hash codes during the comparison process are used to generate properly marked up versions of those two documents.

When all of the pairs of documents have been compared and the appropriate reports generated, an overall comparison report is assembled and presented to the user. The computer memory allocated to the lists of hash coded words and the comparison process is deallocated and the program either awaits a new comparison request or exits.