Path:
sparky!uunet!munnari.oz.au!yoyo.aarnet.edu.au!sirius.ucs.adelaide.edu.au!spam!ross
From: ro...@spam.ua.oz.au (Ross Williams)
Newsgroups: comp.compression
Subject: An algorithm for matching text (possibly original).
Summary: An algorithm for matching text (possibly original).
Keywords: data compression differences super diff text matching algorithm string
searching
Message-ID: <1276@spam.ua.oz>
Date: 27 Jan 92 15:10:05 GMT
Sender: news@spam.ua.oz
Followup-To: comp.compression
Organization: Statistics, Pure & Applied Mathematics, University of Adelaide
Lines: 176

Dear Compressor Heads,

Here is a description of an idea I just had for text matching. It may
have some compression application (see the end of the description of
the idea). I don't know if it is original. Whether it is or not, it
could be of interest to compressor heads and those involved with large
text databases and text matching.

Mainly though, I'm just posting this idea here publicly to impede
anyone who might independently have had the idea from patenting it. If
you like this idea and you want it to be in the public domain, do
something to make it even clearer that the idea has been exposed
publicly (and is therefore more clearly prior art).

Ross Williams
ro...@spam.adelaide.edu.au
27-Jan-1992.

(Header for my Guru text database program).
--<Guru>--
D=27-Jan-1992
F=superdiff.mes
S=An idea for a super differences algorithm/program.
K=differences super differences text matching algorithm string searching
--<Guru>--

IDEA FOR SUPERDIFFERENCES ALGORITHM/PROGRAM
==========

Author : Ross Williams.
Date : Monday 27-Jan-1992, 10:40pm CST Australia (when I had the idea).

The Problem

Quite often one finds that because of various backup activities and so
on, that one has multiple large directory trees full of files, many of
which are identical or share large identical chunks. From this grows
the need for a superdifferences program, that not only compares files,
but also directories of files.

To construct a program to find all identical files in a file system is
easy --- just compute a table of checksums, one for each file, and
then perform full comparisons on files that share the same checksum.
(If you wish, for "checksum" read "hash value").

A harder problem is identifying files that are NEARLY identical or which share large slabs of text. For example, two .C files being nearly the same versions of a program will share most of their text. The problem proposed is to construct an algorithm/utility that will find such shared slabs of text in the file system. The result would be a very useful utility.

A Brute Force Solution

An extremely thorough brute force approach would be to take every subsequence of N characters (e.g. with N=1000 say) at every alignment (i.e. commencing at every byte) in every file and form a huge table of checksums in the same manner as the table of checksums at the file level. The table would identify all common strings of 1000 characters in different files and these connections could be used to explore and identify longer matches.

The only problem with this idea is that the table would be about four times the size of the filesystem (because there would be (say) a 4-byte checksum for each byte (being the checksum for the string of N characters commencing at that byte)).

To cut down on the number of checksums required, we might think of only recording a checksum every M bytes (where M=50 say). If M<<N this should pose no problem. Unfortunately, this scheme fails totally because of alignment problems. Two files sharing a span of text may not have those two spans aligned at MOD 50 boundaries.

Example:
File1:
!This is some shared text.

File2:
This is some shared text.

In the above, although the two files share a slab of text, the two slabs are not aligned (because one has the exclamation mark). If checksums are performed every M bytes, the match will not be detected. Only checksums on every byte boundary will do the trick.

The Good Part (My Idea)

My idea provides a stochastic solution that does not guarantee to find all matches, but will most likely find most matches. It could certainly be used to construct a practical tool.

The idea is this: Run through each file computing a checksum of C bits (e.g. C=32) of all N-byte strings on ALL byte boundaries as with the brute force approach. HOWEVER, only store a (checksum,file,position) tuple in the checksum table if the bottom B bits (e.g. with (say) B=10) of the checksum are zero (or some other B-bit constant).

B can be chosen to taste, with the size of the table and the probability of finding each match being inversely related to B. A value of B=0 corresponds to the brute force approach which will find all matches, but produce a massive table. A value of B=C will produce a table of at most one entry and will most likely find no matches. The number 1/(2^B) is the probability that the checksum commencing at any

particular byte will be entered into the checksum table.

A sliding window checksum could be used so that only about two arithmetic operations are required per byte.

Discussion

The technique that the algorithm uses is to collect a "random" sample of checksums of N-byte sequences from all of the files, but to define "random" deterministically so that the same "random" strings will tend to be collected from the different files. Thus, if the algorithm scans over a span of identical text in two different files, it may not pick up any checksums from the texts (although if B is not set too high this should not happen too often), but if it does pick up any checksums, it will pick them up at the SAME points in the strings - and so a match will be detected!

The idea behind this approach was inspired by the way that humans might perform the task. A human, asked to roughly compare several documents would not compare every substring against every other, but would visually scan each document for "interesting" features and these would be used as reference points to see if large slabs of the document have been seen before. For example, if you put "BISHOP IN SEX SCANDAL" in a comment in the middle of one of your Pascal source files, a human reader would probably consider that string a relatively "interesting" feature of that file, and, if the same feature was seen in another file, the reader would instantly recognise it and try to compare that file against the earlier one seen with the same feature. Note that the reader's definition of "interesting" means that the reader (whom we will temporarily think of as a semi-automaton) does not remember "ISHOP IN SEX SCANDAL ", or " BISHOP IN SEX SCANDA", but "BISHOP IN SEX SCANDAL". The interestingness or otherwise of the information automatically, deterministically, creates a set of alignments at which the various scraps of the different files can be compared.

To get the computer to do the same thing, we merely need to define "interesting". A simple definition, and the one I have chosen to use in the above algorithm is "random" - just carve out a subspace of the checksum space! The algorithm then stores these points as "interesting". So long as the total number of interesting things recorded is eclipsed by the size of the reduced checksum space, this algorithm forms an efficient method for identifying documents sharing large slabs of text. B, which determines the density of interesting things can be set so an interesting feature appears only every couple of thousand bytes or so. And remember that we can set N (the number of bytes used in the checksum) up high too (e.g. 1000).

Once the algorithm has identified different files containing text slabs having the same checksum, it can explore around the shared parts of the files to see just how long the match is. The result could be a comprehensive report outlining (nearly all of) the shared texts in one's file system.

It would be easy to calculate the performance (e.g. failure to match, size of tables etc) probabilities for this system.

A Compression Application

A file system that ran this idea continuously could identify large tracts of shared text in the file system and replace them by references to their copies. This would be substring macro compression on a grand scale. The files being compared would have to be in uncompressed form though, because ordinary compression scrambles everything, making comparisons of substrings difficult.

Conclusion

It is possible that this is an original idea. It would be fun to write a utility that uses this algorithm! The utility would be extremely useful for locating old backup copies of programs in one's file system or any other number of searching tasks.

--<End of Idea>--