NAME

sim − find similarities in C, Java, Pascal, Modula-2, Lisp, Miranda, 8086 assembler code or in text files

SYNOPSIS

sim_c [−**[adefFiMnOpPRsSTuv]** −**r** *N* −**t** *N* −**w** *N* −**o** *F*] file ... [[/ |] file ...]
sim_text ...
sim_c++ ...
sim_java ...
sim_pasc ...
sim_m2 ...
sim_lisp ...
sim_mira ...
sim_8086 ...

DESCRIPTION

Sim_c reads the C files *file ...* and looks for segments of text that are similar; two segments of program text are similar if they only differ in layout, comment, identifiers, and the contents of numbers, strings and characters. If any runs of sufficient length are found, they are reported on standard output; the number of significant tokens in the run is given between square brackets.

Sim_c++ does the same for C++, *sim_java* for Java, *sim_pasc* for Pascal, *sim_m2* for Modula-2, *sim_mira* for Miranda, *sim_lisp* for Lisp, and *sim_8086* for 8086 assembler code. *Sim_text* works on arbitrary text and it is occasionally useful on shell scripts.

The program can be used for finding copied pieces of code in purportedly unrelated programs (with −**s** or −**S**), or for finding accidentally duplicated code in larger projects (with −**f** or −**F**).

If a separator / or | is present in the list of input files, the files are divided into a group of "new" files (before the / or |) and a group of "old" files; if there is no / or |, all files are "new". Old files are never compared to other files. See also the description of the −**s** and −**S** options below.

Since the similarity tester needs file names to pinpoint the similarities, it cannot read from standard input.

The similarity tester takes ASCII or UTF-8 text as input, and produces a sorted list of runs in text form (default or with the **-d** or **-n** options) or in percentage form (with the **-p** option). Input in other formats, e.g. *.pdf* or *.doc* needs to be converted to ASCII or UTF-8 by preprocessing. Aggregated similarity results can be obtained by doing postprocessing on one of the forms of output.

There are the following options:

−**a** All new files are compared to all files. See the section 'Calculating Percentages' below.

−**d** The output is in a diff(1)-like format instead of the default 2-column format. Recommended for text in languages with non-Latin alphabets.

−**e** Each file is compared to each file in isolation. This will find all similarities between all texts involved, regardless of repetitive text, but may be slow for large numbers of files. See also 'Calculating Percentages' below.

−**f** Runs are restricted to segments with balancing parentheses, to isolate potential routine bodies (not in *sim_text*).

−**F** The names of routines in calls are required to match exactly (not in *sim_text*).

−**i** The names of the files to be compared are read from standard input, including a possible separator / or |; the file names must be one to a line. This option allows a very large number of file names to be specified; it differs from the @ facility provided by some compilers in that it handles file names only, and does not recognize option arguments.

−**M** Memory usage information is displayed on standard error output.

−n Similarities found are summarized by file name, position and size, rather than displayed in full.

−o F The output is written to the file named *F*.

−O The option settings used are shown at the beginning of the output.

−p The output is given in similarity percentages; see 'Calculating Percentages' below; implies **−s**.

−P When reporting percentages, only the main contributor for each file is shown.

−r N The minimum run length is set to *N* units; the default is 24 tokens, except in *sim_text*, where it is 8 words.

−R Directories in the input list are entered recursively, and all files they contain are involved in the comparison.

−s The contents of a file are not compared to itself (−s for "not self").

−S The contents of the new files are compared to the old files only − not between themselves.

−t N In combination with the **−p** option, sets the threshold (in percents) below which similarities will not be reported; the default is 1, except in *sim_text*, where it is 20.

−T Suppresses the printing of information about the input files.

−u The output is not buffered and not sorted (only when reporting percentages).

−v Prints the version number and compilation date on standard output, then stops.

−w N The page width used is set to *N* columns; the default is 80.

−− (A secret option, which prints the input as the similarity checker sees it, and then stops.)

The **−p** option results in lines of the form

```
F consists for x % of G material
```

meaning that `x` % of `F`'s text can also be found in `G`. Note that this relation is not symmetric; it is quite possible for one file to consist for 100 % of text from another file, while the other file consists for only 1 % of text of the first file, if their lengths differ enough. The **−P** (capital P) option shows the main contributor for each file only. This simplifies the identification of a set of files `A[1] ... A[n]`, where the concatenation of these files is also present. A threshold can be set using the **−t** option. Note that the granularity of the recognized text is still governed by the **−r** option or its default.

The **−r** option controls the number of "units" that constitute a run. For the programs that compare programming language code, a unit is a lexical token in the pertinent language; comment and standard preamble material (file inclusion, etc.) is ignored and all strings are considered equal. For *sim_text* a unit is a "word" which is defined as any sequence of one or more letters, digits, or characters over 127 (177 octal), to accommodate full Unicode (UTF-8).

The programs can handle Unicode (UTF-16) file names under Windows.

Sim_text accepts s p a c e d t e x t as normal text.

Once *sim* has read, stored and preprocessed the input, it will no longer run out of memory. If memory is short it will change automatically to unbuffered, unsorted output (while issuing a warning message).

HOW SIMILARITY IS RECOGNIZED

Since computers cannot test for similarity, only for equality, all units in the input files are replaced by 16-bit tokens, such that all units that are regarded as similar are reduced to the same token. For example in *sim_c* all identifiers are replaced by the token IDF and all strings are replaced by the token STR. The secret option −− can be used to see the resulting token sequence.

In *sim_text* each word is reduced to a 16-bit token using a hash function. There is a chance of 1 : 65536 that two different words get the same token value, but because recognized runs of tokens

are usually several tokens long, the chances for accidental similarities are very low.

The sequence of tokens obtained this way is then processed as follows.

The default operation cycle of *sim* starts at the beginning of the token sequence of the first input file or at a position X in file F at which the previous cycle has left off. *Sim* then finds the longest segment S such that 1) S is equal to the segment starting at X; 2) S is situated somewhere between position X in F and the end of all files; 3) S does not overlap with the segment starting at X. If the segment is at least of minimum run size, it is recorded, and the cycle starts again just after the segment at X; otherwise it starts again at $X + 1$.

So if the token sequence at X reads `abcabcadefabdabcz`, the cycle finds S to be the `abc` just before the end; `abca` at $X + 3$ would be longer but overlaps with the `abca` at $X + 0$. The cycle then starts at $X + 3$, and will find another match with the `abc` near the end. Finally the `ab` after the `f` will be matched with the `ab` just before the `cz`. So the following matches are found:

$X[0{:}2] = X[13{:}15] =$ `abc`
$X[3{:}5] = X[13{:}15] =$ `abc`
$X[9{:}10] = X[13{:}14] =$ `ab`

This way best matches for the text in a file are found in material to the right of it, until the end of all files. The results are asymmetric: given files *F1*, *F2*, *F3*, *F4*, no matches for *F3* are reported from *F1* or *F2*, for example. As explained below under "Limitations", this avoids duplicate reports of similarity and helps to keep *sim* fast.

WHAT IS COMPARED TO WHAT

The area that is searched by *sim's* cycle is called the *range*. The default range, which as we have seen above runs from the file under observation to the end of all files, is excellent for finding similarities in program files, and, when doing percentages, for getting an impression of which files are related to which files, but sometimes more control is needed. The following modifications to the range are available:

The **−a** option includes *all* text in the range by not stopping the search at the end of the files but rather looping back to the beginning of the files and continuing to the point where the search started. Now matches are also found in files before the present one and the results are symmetric: given files *F1*, *F2*, *F3*, *F4*, matches for *F3* will also be reported from *F1* or *F2*, if present. But matches may be reported twice, once for file *Fa* versus file *Fb*, and once for file *Fb* versus file *Fa*. The **−a** option allows a more accurate determination of similarity percentages.

The **−a** option is the only way to obtain symmetrical results, with information about both *F1* vs. *F2* and *F2* vs. *F1*.

The **−S** option removes the new files from the range, so files are only compared to the old files.

The **−s** option removes the file itself from the range, so a file will not be compared to itself. This is the default when reporting percentages.

In normal operation the whole range is searched as one unit. The **−e** option divides up the range into the separate files, and causes *sim* to compare a file to each of the other files separately. This produces the most detailed information when reporting text similarities, and the best possible results when reporting similarity percentages, but can be quite slow.

A Tabular Representation

Input files are divided into two groups, new and old. In the absence of control options *sim* com-

pares the files thus (for 4 new files and 6 old ones):

```
                        n e w    /     o l d       <- second file
                       1  2  3  4 / 5  6  7  8  9 10
                     |------------/------------
               n  1 | c  c  c  c / c  c  c  c  c  c
               e  2 |    c  c  c / c  c  c  c  c  c
               w  3 |       c  c / c  c  c  c  c  c
                  4 |          c / c  c  c  c  c  c
      first       / / /  /  /  / / /  /  /  /  /  /
      file  ->    5 |            /
               o  6 |            /
               l  7 |            /
               d  8 |            /
                  9 |            /
                 10 |            /
```

where a `c` indicates that the first file is compared to the second file, and the `/` represents the demarcation between new and old files. The comparison range of the first files is clearly visible.

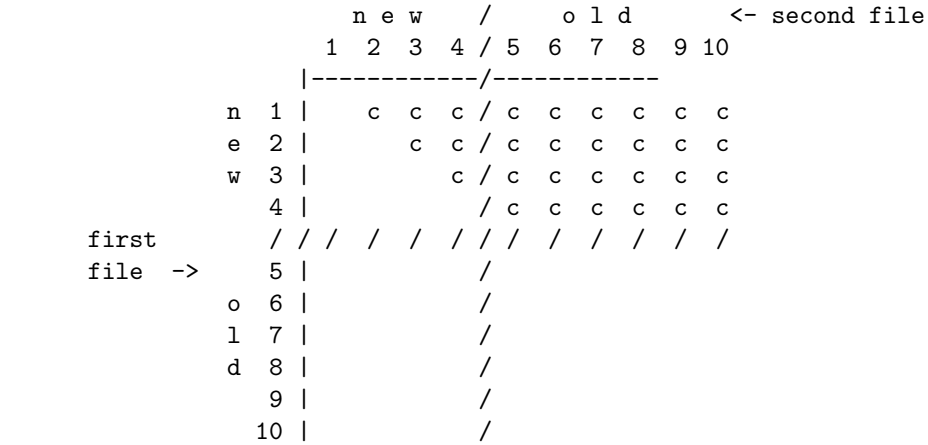
Using the **−a** option extends this to

```
                        n e w    /     o l d       <- second file
                       1  2  3  4 / 5  6  7  8  9 10
                     |------------/------------
               n  1 | c  c  c  c / c  c  c  c  c  c
               e  2 | c  c  c  c / c  c  c  c  c  c
               w  3 | c  c  c  c / c  c  c  c  c  c
                  4 | c  c  c  c / c  c  c  c  c  c
      first       / / /  /  /  / / /  /  /  /  /  /
      file  ->    5 |            /
               o  6 |            /
               l  7 |            /
               d  8 |            /
                  9 |            /
                 10 |            /
```

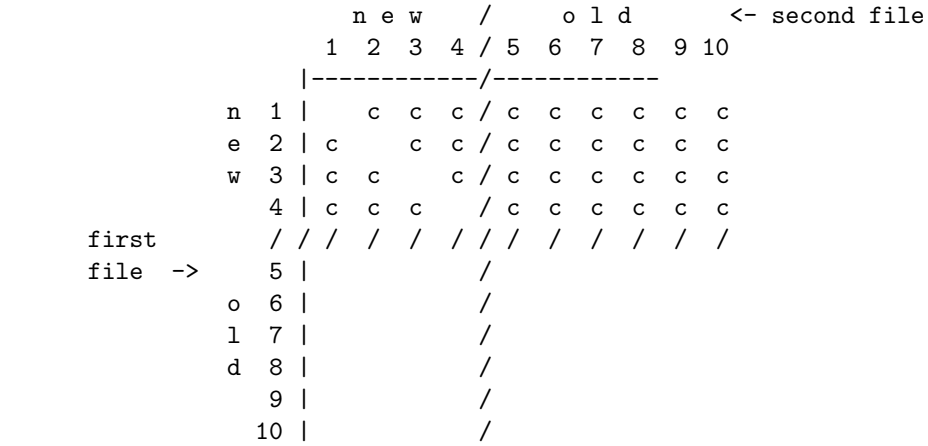
Using the **−S** option instead reduces this to

```
                        n e w    /     o l d       <- second file
                       1  2  3  4 / 5  6  7  8  9 10
                     |------------/------------
               n  1 |            / c  c  c  c  c  c
               e  2 |            / c  c  c  c  c  c
               w  3 |            / c  c  c  c  c  c
                  4 |            / c  c  c  c  c  c
      first       / / /  /  /  / / /  /  /  /  /  /
      file  ->    5 |            /
               o  6 |            /
               l  7 |            /
               d  8 |            /
                  9 |            /
                 10 |            /
```

Finally, using the −**s** option changes the default ranges to



and the −**a**-extended ranges to



LIMITATIONS

Repetitive input is the bane of similarity checking. If we have a file containing 4 copies of identical text,

```
A1 A2 A3 A4
```

where the numbers serve only to distinguish the identical copies, there are 7 non-overlapping identities: `A1=A2`, `A1=A3`, `A1=A4`, `A2=A3`, `A2=A4`, `A3=A4`, and `A1A2=A3A4`. Of these, only 3 are meaningful: `A1=A2`, `A2=A3`, and `A3=A4`. And for a table with 20 lines identical to each other, not unusual in a program text, there are 715 non-overlapping identities, of which at most 19 are meaningful. Reporting all 715 of them is clearly unacceptable.

This is remedied by *sim's* search cycle: for each position in the text, the largest segment is found of which a non-overlapping copy occurs in the text following it. That segment and its copy are then reported and scanning resumes at the position just after the segment. For the above example this results in the two identities `A1A2=A3A4` and `A3=A4`, which is quite satisfactory, and for N identical segments roughly $2 \log N$ messages are given.

This also works out well when the four identical segments are in different files:

```
File1: A1
File2: A2
File3: A3
File4: A4
```

Now combined segments like `A1A2` do not occur, and the algorithm finds the runs `A1=A2`, `A2=A3`, and `A3=A4`, for a total of *N-1* runs, all informative.

Calculating Percentages

The above approach is unsuitable for obtaining the exact percentage of a file's content that can be found in another file, although indicative results can be obtained. Obtaining exact percentages requires comparing each file pair in isolation; this is what the **−ae** options do. Under the **−ae** options a segment `File3:A3`, recognized in `File4`, will again be recognized in `File1` and `File2`. In the example above it produces the runs

```
File1:A1=File2:A2
File1:A1=File3:A3
File1:A1=File4:A4
File2:A2=File3:A3
File2:A2=File4:A4
File2:A2=File1:A1
File3:A3=File4:A4
File3:A3=File1:A1
File3:A3=File2:A2
File4:A4=File1:A1
File4:A4=File2:A2
File4:A4=File3:A3
```

for a total of *N(N-1)* runs.

When the **−e** option is used alone. *sim* will find the following runs:

```
File1:A1=File2:A2
File1:A1=File3:A3
File1:A1=File4:A4
File2:A2=File3:A3
File2:A2=File4:A4
File3:A3=File4:A4
```

for a total of *½N(N-1)* runs, thus missing half the percentage contributions; in fact, `File4` is found to have 0% in common with the other files.

If, however, the **−a** option is used alone. *sim* finds the following runs:

```
File1:A1=File2:A2
File2:A2=File3:A3
File3:A3=File4:A4
File4:A4=File1:A1
```

for a total of *N* runs. This setting misses many of the percentage contributions, but finds something for every file.

TIME AND SPACE REQUIREMENTS

Care has been taken to keep the time requirements of all internal processes (almost) linear in the lengths of the input files, by using various tables.

The time requirements are quadratic in the number of files. This means that, for example, one 64 MB file processes much faster than 8000 8 kB files.

The program requires 6 bytes of memory for each token in the input; 2 bytes per newline (not when doing percentages); and 80 bytes for each run found.

EXAMPLES

The call

```
sim_c *.c
```

highlights duplicate C code in the directory. (It is useful to remove generated files first.) A call of

```
sim_c -f -F *.c
```

can pinpoint the duplicate code further.

A call

```
sim_text -peu -S new/* "|" old/*
```

compares each file in `new/*` to each file in `old/*`, and if any pair has more that 20% in common, that fact is reported. Usually a similarity of 30% or more is significant; lower than 20% is

probably coincidence; and in between is doubtful.

The `u` in `-peu` causes the output to be unbuffered (and unsorted), so if the program is stopped for running out of time, any results already found are not lost.

For large data sets, using `-pu` rather than `-peu` may do the job much more quickly, but less accurately.

The `|` can be used as a separator instead of `/` on systems where the `/` as a command-line parameter gets mangled by the command interpreter.

These calls are good for plagiarism detection.

BUGS

Unbuffered, unsorted output is not available for text output, only for percentage output.

AUTHOR

Dick Grune, Vrije Universiteit, Amsterdam; dick@dickgrune.com.