## NAME

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

## SYNOPSIS

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

## 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_java* does the same for Java, *sim_pasc* for Pascal, *sim_m2* for Modula-2, *sim_mira* for Miranda, and *sim_lisp* for Lisp. *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 each other. 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.

There are the following options:

−**d** The output is in a diff(1)-like format instead of the default 2-column format.

−**e** Each file is compared to each file in isolation; this will find all similarities between all texts involved, regardless of repetitive text (see '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*.

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

−**P** As −**p** but only the main contributor is shown; implies −**e** and −**s**.

−**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** or **−P** options, sets the threshold (in percent) below which similarities will not be reported; the default is 1, except in *sim_text*, where it is 20.

**−T** A more terse and uniform form of output is produced, which may be more suitable for postprocessing.

**−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 in fact 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 the same. 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 letters such as ä, ø, etc.).
*Sim_text* accepts s p a c e d t e x t as normal text.

The **−s** and **−S** options control which files to compare. Input files are divided into two groups, new and old. In the absence of these control options the programs compare the files thus (for 4 new files and 6 old ones):

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

where the cs represent file comparisons, and the / the demarcation between new and old files.

Using the −**s** option reduces this to:

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

The −**S** option reduces this further to:

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

The programs can handle UNICODE file names under Windows. This is relevant only under the −**R** option, since there is no way to give UNICODE file names from the command line.

## 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 8 identities: `A1=A2`, `A1=A3`, `A1=A4`, `A2=A3`, `A2=A4`, `A3=A4`, `A1A2=A3A4`, and `A1A2A3=A2A3A4`. 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, there are 715 identities, of which at most 19 are meaningful. Reporting all 715 of them is clearly unacceptable.

To remedy this, finding the identities is performed as follows: 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 reported and scanning resumes at the position just after the segment. For the above example this results in the 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 not suitable for obtaining the percentage of a file's content that can be found in another file. This requires comparing in isolation each file pair represented by a `c` in the matrixes above; this is what the **−e** option does. Under the **−e** option a segment `File1:A1`, recognized in `File2`, will again be recognized in `File3` and `File4`. 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
File3:A3=File4:A4
```

for a total of *½N(N-1)* runs.

## 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. If, however, there is not enough memory for the tables, they are discarded in order of unimportance, under which conditions the algorithms revert to their quadratic nature.

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 about 76 bytes for each run found.

## EXAMPLES

The call

```
sim_c *.c
```

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

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

can pinpoint them further.

A call

```
sim_text -e -p -s new/* / old/*
```

compares each file in `new/*` to each file in `new/*` and `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.

A call

```
sim_text -e -n -s -r100 new/* "|" old/*
```

compares the same files, and reports large common segments. (The | can be used as a separator instead of / on systems where the / as a command-line parameter gets mangled by the command interpreter.)

Both approaches are good for plagiarism detection.

## BUGS

Since it uses *lex(1)* on some systems, it may crash on any weird construction in the input that overflows *lex*'s internal buffers, for example an identifier of several thousand letters long.

## AUTHOR

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