Finding Files Fast

James A. Woods
Informatics General Corporation
NASA Ames Research Center
Moffett Field, California 94035

January 15, 1983

ABSTRACT

A fast filename search facility for UNIX is presented. It consolidates two data compression methods with
a novel string search technique to rapidly locate arbitrary files. The code, integrated into the standard
find utility, consults a preprocessed database, regenerated daily. This contrasts with the usual mechan-
ism of matching search keys against candidate items generated on-the-fly from a scattered directory
structure.

The pathname database is an incrementally-encoded lexicographically sorted list (sometimes referred to
as a “front-compressed” file) which is also subjected to common bigram coding to effect further space
reduction. The storage savings are a factor of five to six over the standard ascii representation. The list
is scanned using a modified linear search specially tailored to the incremental encoding, typical “user
time” required by this algorithm is 40%-50% less than with naive search.

Introduction

Locating files in a computer system, or network of systems, is a common activity. UNIX users
have recourse to a variety of approaches, ranging from manipulation of cd, ls, and grep commands, to
specialized programs such as U. C. Berkeley’s wheres and fleece, to the more general UNIX find.

The Berkeley fleece is unfortunately restricted to home directories, and whereis is limited to eke-
ing out system code/documentation residing in standard places. The arbitrary

find / -name "*<filename>*" -print

will certainly locate files when the associated directory structure cannot be recalled, but is inherently
slow as it recursively descends the entire file system to mercilessly thrash about the disk. Impatience
has prompted us to develop an alternative to the “seek and ye shall find” method of pathname search.

Precomputation

Why not simply build a static list of all files on the system to search with grep? Alas, a healthy
system with 20000 files contains upwards of 1000 blocks of filenames, even with an abbreviated hu (vs.
/usr) adopted for user home prefixes. Grep on our unloaded 30-40 block/second PDP 11/70 system
demands half a minute for the scan. This is unacceptable for an oft-used command.

Incidently, it is not much of a sacrifice to be unable to reference files which are less than a day
old—either the installer is likely to be contactable, or the file is not quite ready for use! Well-aged files
originated by other groups, usually with different filesystem naming conventions, are the probable can-
didates for search.

Compression

To speed access for the application, one might consider binary search or hashing, but these
schemes do not work well for partial matching, where we interested in portions of pathnames.
Though fast, the methods do not save space, which is often at a premium. An easily implementable
space saving technique for ordered data, known as incremental encoding, has been adapted for the similar task of dictionary compression [Morris/Thompson, 1974]. Here, a count of the longest prefix of the preceding name is computed. For example,

```
/usr/src
/usr/src/cmd/aardvark.c
/usr/src/cmd/armadillo.c
/usr/tmp/zoo
```

transforms to

```
0 /usr/src
8 /cmd/aardvark.c
14 armadillo.c
5 tmp/zoo
```

If we choose to delimit the pathname residue with parity-marked count bytes, decoding can be as simple as (omitting declarations):

```c
fp = fopen ( COMPRESSED_FILELIST, "r" );
while ( (count = (getc ( fp ) & 0177)) != EOF ) {
    for ( p = path + count; (*p++) = getc ( fp ) ) < 0200; )
        /* overlay old path with new */
    ungetc (*--p, fp );
    *p- = NULL;
    if ( match ( path, name ) == YES )
        puts ( path );
}
```

where `match` is a favorite routine to determine if string `path` contains `name`.

In fact, since the coded filelist is about five times shorter than the uncoded one, and the decoding is very easy, this program runs about three to four times as fast as the efficient `grep` on the expanded file.

**Speedier Yet**

Useful as it is, there is still room for improvement. (Aside: this code is best inserted into the distributed `find`. There is no need to burden UNIX with another command [and manual page] when we can improve an existing similar program. Conveniently, there is no two-argument form of `find` so we can fill the vacuum with an undated

```
find name
```
to perform the function.)

Notice that the above code fragment still searches through all the characters of expanded list, although in main memory instead of disk. It turns out that this can be avoided by matching the name substring backwards against a reversed pathname, until the boundary delineated by the repetition count. Assuming `namend` points to the final character of a NULL-byte prefixed `name`, then replace `match` by

```c
for ( s = p, cutoff = path + count; s >= cutoff; s-- ) {
    if ( *s == *namend ) { /* quick first char check */
        for ( p = namend - 1, q = s - 1; *p != NULL; p--, q-- )
            if ( *q != *p )
                break;
    if ( *p == NULL )
        puts ( path );
    break;
    }
}
```
This is more easily understood by considering three cases. If the substring lies wholly to the right of the cutoff, the match will terminate successfully. If there is an overlap, the cutoff becomes "soft" and the match continues. If the substring lies completely to the left of the cutoff, then a match would have been discovered for an earlier pathname, so we need not search these characters! Technically, cutoff must be re-anchored to path immediately after matches. This condition is omitted above for the sake of clarity. Statistics on overlap have not been garnered, but a 40-50% speedup is consistently observed.

The author has not discovered this refinement in the literature.

**Two Tier Technique**

Shell-style filename expansion without undue slowdown can be had by first performing the fast search on a metacharacter-free component of name, then applying regular expression syntax "globbing" to these selected paths via the slower recursive amatch function internal to find. Ergo,

\begin{verbatim}
puts ( path );
\end{verbatim}

becomes

\begin{verbatim}
if ( globchars == NO | amatch ( path, name ) )
    puts ( path );
\end{verbatim}

where globchars is set if name contains shell glob characters. Using wildcarding, a primitive man command might be

\begin{verbatim}
vtree -man 'find *man*"$1". [1-9]'
\end{verbatim}

**Diminishing Returns**

Production find code at Ames exacts a further 20-25% space compression (entropy reduction) by assigning single non-printing ascii codes to the most common 128 bigrams. ".e" and ".P" figure prominently. Room for these codes is made by reserving only 28 count codes for the likeliest "differential" counts (the interline difference between one prefix count and the next), along with a "switch" code for out-of-range counts (remember the possible 1024 byte pathnames, courtesy BSD 4.2). Printable ascii comprises the filename residue. We will not dwell on this rather ad hoc means, which barely reduces search time.

Other algorithms to address the time-space complexity tradeoff such as Huffman or restricted variability coding [Reghbati, 1981] do not look promising—they only change an I/O-bound process to a compute-bound one. Some experiments were done with the inverted file programs inv and hunt. Here, process startup overhead (the grep call to disambiguate "false drops") and space consumption (full pathnames plus an index) make inv invocations noncompetitive. Boyer-Moore sublinear search [Boyer, 1977] or macro model methods [Storer/Szymanski, 1982] might be employed, but must concern typically short 4-10 character patterns and equally short post-compression pathname content, for all their added complexity.

To conclude, we are content to scan 19000 filenames in several seconds using 180 blocks and two extra pages of C code.

**REFERENCES**


