# The LikeIt Intelligent String Comparison Facility

Peter N. Yianilos    Kirk G. Kanzelberger

NEC Research Institute Technical Report*
May 1997

**Abstract**

A highly-efficient ANSI-C facility is described for intelligently comparing a query string with a series of database strings. The bipartite weighted matching approach taken tolerates ordering violations that are problematic for simple automaton or string edit distance methods—yet common in practice. The method is character and polygraph based and does not require that words are properly formed in a query.

Database characters are processed at a rate of approximately 2.5 million per second using a 200MHz Pentium Pro processor. A subroutine-level API is described along with an simple executable utility supporting both command-line and Web interfaces. An optimized Web interface is also reported consisting of a daemon that preloads multiple databases, and a corresponding CGI stub. The daemon may be initiated manually or via *inetd*.

**Keywords:** *String Comparison/Similarity, Text/Database Search/Retrieval, Bipartite Matching/Assignment, Edit Distance.*

*Both authors are with the NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. Email: `pny@research.nj.nec.com`.

# 1 Introduction

A decade ago, *databases* were applications used by a relatively small number of users in highly structured, corporate data processing environments. Now, databases are a central application of mainstream computing. This occurred in large part because of the decade's striking advances in connectivity. The mid-eighties' emphasis on local area networks has been replaced with the world Internet. At the same time, the community of users accessing databases has grown from a somewhat homogeneous and geographically localized collection, to a highly diverse group spanning the globe and speaking many languages. This paper describes the LikeIt software facility which addresses a central problem that has emerged as a result of these changes, namely that of *robust semistructured text retrieval* for small and medium size databases.

Semistructured text lies between fully structured databases and unconstrained text streams. A fully structured database might, for example, represent a person's name using as many fields as the name has parts. The semistructured approach might represent a name in a less formal way using a single text field. Other examples of semistructured text fields are company or institution names and addresses (as in a directory of businesses), item names and descriptions (as in an online catalog), author names and book or paper titles (as in a bibliographic database). Several related fields might be combined into one, as in figure 1, which illustrates a bibliographic record implicitly containing author, title, journal, volume, and year fields.

Considerable variation is possible in the description of an item using a semistructured record. A person's last name might be listed first or last. The middle name might be excluded or abbreviated. The ordering of a complex name's parts is not always well determined. Some information may be missing from a given record, or extra information may be present. In principle, policies can be established to try to regularize representations, but in practice, such policies rapidly become complex and confusing. Moreover, any unforeseen exceptional cases, variations, or errors, either in the database itself or (perhaps more commonly) in a user's query, can easily defeat search and retrieval operations that are predicated on an enforced regularity.

The World Wide Web places an enormous amount of information at the disposal of the ordinary user equipped with PC, modem, and Internet connection. Much of this information is unstructured beyond the level of simple paragraphs or list items, and the timely and ephemeral nature of the information makes it unreasonable to expect otherwise. Furthermore, much of this

| | |
|---|---|
| Query: | problmoptimldictionry |
| Record: | Andersson, Optimal Bounds on the Dictionary Problem |
| | LNCS, 401, 1989 |

Figure 1: All three words of this query are misspelled, they occur in the wrong order, and there is no space separating them. Nevertheless the desired record is identified using the LikeIt facility from a listing of 50,360 paper descriptions in the field of theoretical computer science. Author(s) name, paper title, and related information are combined into a single database text field.

information is directly human-generated, so errors are relatively abundant—the data is certainly never as clean as a corporate database.

We address the problems of variation and error in semistructured data by increasing the sophistication of the software that is used to compare queries with semistructured records. The heart of the LikeIt facility is a function that compares two text strings, and returns a numerical indication of their similarity. Typically one of these strings is the user's query and the other is a record from a database. Because this function is very fast, it is possible to compare the query with thousands or even hundreds of thousands of database records while still delivering acceptable response time. Also included is a high-speed heap data structure used to track the *best matches* encountered during a search.

An important benefit of the LikeIt approach is that the queries are simple free-form expressions of what the user is looking for. There is no query language, and the comparison function is rather robust with respect to typical errors, missing or extra information, and overall ordering (see figure 1 based on the database of [13]). Also, the LikeIt facility includes no natural language-specific considerations. It operates on byte strings and as such may be used across languages and perhaps for applications that have nothing to do with language (such as DNA sequence comparison).

The contribution of LikeIt is a particularly simple and time/space-efficient implementation of intelligent string comparison based on bipartite matching. Using a 200Mhz Pentium Pro processor, processing one byte of database information typically requires roughly $0.5\mu S$. So 100,000 records of 30 characters each can be processed in 0.15 seconds. LikeIt is in some sense a fourth generation implementation of the following general approach:

1. Algorithms of the general type used by LikeIt were introduced in [15], and were later used in the commercial spelling correctors of Proximity Technology, Inc., and Franklin Electronic Publishers. The linguistic software components developed by these companies were used under license in word-processing programs from hundreds of publishers, in typewriters, and in tens of millions of hand-held spelling devices.

2. The PF474 VLSI chip was a special purpose pipelined processor [16, 11] that implemented the algorithm of [15]. Today's software matches and even exceeds the performance of this device, although the comparison is not entirely fair, since the PF474 was clocked at only 4 Mhz. The same design implemented today would still result in a 1–2 order of magnitude hardware advantage.

3. The Friendly Finder software utility [10, 9], first introduced in 1987 by Proximity Technology, implemented the algorithm together with software accelerations and special treatment for bigrams. The result was that small database could be searched on early personal computers without using the PF474 chip. The computational heart of Friendly Finder was also made available under license, and named "P2."

4. A transition to the bipartite matching viewpoint took place with [1, 2], the algorithms being improved and in some cases simplified. The result is entirely new algorithms that are still of the same family. The LikeIt facility is the first implementation based on these new developments. The algorithms of [1] lead to linear time algorithms for a large class of graph cost functions, including the simple linear costs used by LikeIt. Linear time matching algorithms for this particularly simple special case were first presented in [6].

The LikeIt approach, used in effect by Friendly Finder [10], is to build an optimal weighted matching of the letters and multigraphs in the query, and those in each database record. Words as such receive no special treatment. In this sense it is related to the document retrieval approach of [3, 5].

An alternative approach to string comparison computes *edit distance* [4, 12], i.e., the minimum-cost transformation of one string into another via some set of elementary operations. Most commonly, weighted insertion, deletion, and substitution operations are used, and the edit distance computation is a straightforward dynamic program. However, edit distance suffers from two problems that led to our own approach. First, the dynamic program runs in $O(m \cdot n)$ time, where $m, n$ are the string lengths, whereas LikeIt runs in $O(m + n)$ time. Second, the edit distance approach is highly sensitive to global permutation, e.g., changing word order. Humans frequently are not, and LikeIt deals well with this issue.

The automaton-based approach to fast string matching introduced in [7] deals with exact matches only. A natural generalization relaxes the requirement of exact equality and allows a bounded (and in practice small) number of errors. Each such error is typically restricted to be either an insertion, deletion, substitution, or sometimes a transposition of adjacent symbols. Given a query string, it is then possible to build an automaton to detect it, or any match within the error bounds, within a second string. The recent work of [8, 14] demonstrates that text can be scanned at very high speeds within this framework for comparison. The LikeIt framework, on the other hand, can satisfy queries that do not fall within the practical capabilities of the automaton approach because they are too different from the desired database record.

The LikeIt facility searches by *brute force*. It compares the query with every database record. With today's CPUs, this limits its applications to tens of thousands or perhaps hundreds of thousands of records. We see LikeIt and related approaches as important and effective tools for medium-size textual databases—still small enough to scan in their entirety for each query. Organizing a database so that not every record needs to be considered represents an interesting area for future work. Techniques such as vantage-point trees [17] might be used for this purpose even through the similarity value computed by LikeIt is not strictly a mathematical metric.

Section 2 of this paper describes the ANSI-C subroutine-level interface (API) to the LikeIt facility. The reader interested in quickly applying LikeIt to a problem may read this section and skip directly to appendix A, which describes each file in our standard distribution and explains how to `make` it. Two user-level utility programs are described in section 3, which provide command-line and Web interfaces to the facility. In this way, simple line-oriented databases may be accessed using LikeIt without any programming requirement at all. These simple utilities read the entire database from disk each time they are invoked. A more sophisticated Web solution is described in section 4. Here, a daemon preloads one or more databases and is contacted by a CGI stub program to resolve queries. The daemon may be invoked manually or via *inetd*, and includes a facility for returning HTML links corresponding to matching records. Section 5 gives details of the internal design of LikeIt, and the advanced user may customize certain aspects of its behavior using the interface described in section 6.

## 2 User's View of the Facility

The primary interface consists of a single function:

```
int likeit(unsigned char *query, unsigned char *(*recfunc)(int),
           void *expert, int num_matches, int *matches,
           double *scores );
```

Processing begins by translating the null-terminated `query` string, character by character to an internal alphabet. Default operation collapses ASCII considerably by mapping all non-alphanumeric values to *space* and uppercase letters to their lower-case equivalents. This front end translation may be customized as described in section 6. This translation process and certain other behavioral characteristics are specified via the `expert` parameter. For convenience a `NULL` value for this parameter selects default operation[1].

The query is compared with a sequence $s_1, \ldots, s_n$ of database strings that `likeit()` accesses by calling the user-supplied `recfunc()` slave function. A single integer argument is supplied to select an element of the sequence, and a pointer to that element is returned. The `recfunc()` function must return `NULL` if its argument is less than 1 or greater than $n$, and support random access of the sequence. Using the default `expert` value of `NULL`, the length of queries and database strings should be limited to $1,024$.

The result of each comparison is a numerical indication of similarity, and `likeit()` keeps track of the `num_matches` most similar database strings encountered. Their indices are written into the user-allocated integer vector `matches`. The corresponding `double` similarity values are written into the user-allocated `scores` vector. These are actually nonnegative graph matching costs so that lower values correspond to increased similarity. The `likeit()` function returns the actual number of values written into these vectors—a value less than or equal to `num_matches`.

[1]Default operation is also selected by specifying the constant `LKT_DEFAULT_EXPERT`

## 2.1 User-supplied preprocessing

Our experience is that simple preprocessing of queries and database strings before they are passed to `likeit()` improves results. Leading and trailing white space is deleted, repeated white space is collapsed to a single space character, and finally, a single leading and trailing space character is inserted.[2] Combining these steps with the default translation described above we have, for example:

$\boxed{\texttt{OPtimal␣␣(Dictionary),␣..␣␣}} \rightarrow \boxed{\texttt{␣optimal␣␣dictionary␣␣␣␣}}$

where the repeated spaces in the final result arise from the translation process. We have found this combined processing to be an effective general purpose solution, and it is performed by the utilities described in section 3. Other preprocessing schemes and translation mappings may be more suitable for particular applications.

Ideally preprocessing is performed in advance for each database string so that `recfunc()` can operate by simply returning the address of an already edited string.

## 2.2 Facility Self-Test

It can be difficult to ensure that intelligent functions such as LikeIt are operating correctly. For this reason the facility includes a self-test operation:

```
int likeit_self_test(int test_type);
```

The `test_type` is either 0 or 1, selecting a *little* or a *big* test, respectively. The return is a boolean value indicating success. A little test takes very little time and is performed internally and automatically every time `likeit()` is called. A big test takes much more time, and may be run by calling this function with an argument of 1, or by running the `likeit` utility program described in section 3 with the `-t` command-line flag.

Both tests generate random queries and database records over a fixed size alphabet. The ASCII-based default translation above does not apply. A hash of the information returned by `likeit()` is checked against a recorded value. The test characteristics are:

[2] In the user-level utility programs provided, the leading and trailing single spaces are added to database records at search time, resulting in a small time penalty.

| | Query Length | # Queries | DB record length | # records | Alphabet Size |
|---|---|---|---|---|---|
| Little | $< 8$ | 5 | $< 32$ | 10 | 7 |
| Big | $< 32$ | 100 | $< 128$ | 1000 | 127 |

# 3 Utilities

This section describes a simple multipurpose application program that uses the LikeIt facility to provide both command-line search services, and a common gateway interface (CGI) for Web environments. A single executable serves both purposes—its filename selects the mode. Filename `likeit` provides command-line operation and `likeit.cgi` causes the program to enter CGI mode.

Both modes of the utility read a simple line-oriented database. Vertical bars '`|`' may be used to separate fields within a record, but these separators have no effect on the comparisons made by LikeIt[3]—they only affect the appearance of the output in CGI mode (see below). That is, each line in the database is treated as a single string, so that information in one field can match that in another. The database is first read into an array of structures, and the whitespace preprocessing described in section 1 is performed. A simple `recfunc()` is implemented that delivers records by indexing into this structure.
Command-line usage is:

```
likeit [-n <nmatches>] <database> ´<query>´
```

The optional `-n` argument requests display of a specific number of matches. The default is 25 and the maximum is 1000. The <database> argument names the database file formated as described above. The final argument ´<query>´ provides the search query, and must be enclosed in single or double quotes if it contains whitespace. A facility self-test ("big") may be run by the special usage: `likeit -t`. The following example using the sample database [13] provided with the LikeIt distribution illustrates command-line mode:

```
% likeit -n 2 rochester.lst "optimldictionryproblm"
 6263703.0 Andersson, Optimal Bounds on the Dictionary
```

[3] In fact, they are mapped to *space* characters like other non-alphanumerics.

```
           Problem|LNCS|401|1989
 6272296.0 Li & Probst, Optimal VLSI Dictionary Machines
           Without Compress Instructions|IEEETC|39|1990
%
```

The numeric value at the beginning of each line is the score returned from `likeit()`. It is important to note that these include what amounts to a large positive bias arising from the technical details of our implementation. As such they are not directly proportional to a similarity or distance.

The CGI executable named `likeit.cgi` reads three HTML form parameters: `database`, `nmatches`, and `query`, which are self-explanatory. A horizontal rule separates records in the HTML output. If field separators are used in the database, each field of a record appears on a separate line.

While `likeit.cgi` may be useful for demonstration purposes and small applications, it is far from optimal since the database is reread with each query. In some cases this performance problem is reduced by operating system file system caching. A more sophisticated implementation is described next involving a search daemon to which a very small CGI stub connects to resolve queries.

# 4 A LikeIt daemon

The `likeit.cgi` program described in the previous section is useful for demonstration purposes, for one-time queries, or for small applications. In this section we describe a solution suitable for more general use.

A daemon is invoked which preloads one or more databases. HTML queries are communicated to it via a small CGI stub program `likeitstub.cgi`. The daemon responds with a list of matching records which the stub formats as HTML and returns to its client. The preloaded records are fully preprocessed by the daemon, including the addition of leading and trailing space characters. The result is that search times are slightly faster than those reported elsewhere in this paper. Three arguments `database`, `nmatches`, and `query` are used. The value of the first is a zero-based integer selecting one of the preloaded databases. The meaning of the others is clear.

Communication between the stub and daemon is line-oriented. After the connection is established a simple handshake verifies that the versions of these two programs agree. The integral database identifier, requested number of matches, query length, and query are then communicated to the daemon – each on a separate line.

The daemon responds with a series of lines. The preamble contains an integral database type indicator. Currently two types, *line* and *line-url*, are supported. These correspond to indicators 0 and 1 respectively. The first selects a database format identical to that described earlier in this paper. Vertical bar symbols '`|`' mark line breaks in the HTML output. In the second format a URL is provided on a separate line following each database record. The HTML output then displays the record as a link to this URL. A blank URL is allowed.

The daemon then responds with the number of matches returned (may differ from the number requested). Next the matching records are returned. Each begins with a score. The stub ignores this in the present implementation. The matching record is then returned followed by a URL (for *line-url* database).

All HTML formating is performed by the stub, so the appearance of the returned document may be altered by editing its source code.

The daemon may be invoked manually or via *inetd*. The executable's name differs in these two cases and must be The `likeitd` or `in.likeitd` respectively. In both cases the daemon accepts a single argument naming a configuration file. Each line of this file specifies a database and consists of two white space delimited fields. The first is either `line` or `line-url` and selects type. The second gives the filename.

# 5 Internals

The LikeIt facility reads database records and outputs those that it regards as the most similar matches to the query provided. A weighted bipartite graph matching approach is taken to the problem of quantifying *similarity*. The query string is imagined to be positioned above the database string and the matching problem is to correspond features using edges of minimum total weight. In the simplest case the features consist of single letters, and the weight of an edge is the distance (in units of string position) between occurrences in the top and bottom string.

Our own sense of string similarity fairly clearly depends on higher level features such as digraphs, trigraphs, and ultimately entire words. The LikeIt facility captures this effect by efficiently constructing several matching problems—one for single letters, another for letter pairs (digraphs), etc. Its sense of similarity is then a composite of these solutions.

The particular alignment of the query above the database string clearly affects the cost of a matching. For this reason LikeIt treats this alignment as a variable and attempts to minimize total matching cost over it. That is, the query is imagined to slide left or right until a position resulting in minimum total matching cost is found.

The result is a rather complicated process, and despite our emphasis on efficiency, a time consuming one. For this reason the LikeIt facility is implemented as a three stage filter (figure 2) in which the computation above is the final stage designated $F3$. The two earlier stages $F1$ and $F2$ approximate the similarity judgment of $F3$ using far less time. The $F2$ stage crudely approximates the optimization over query alignment as described later in this section. The first stage $F1$ approximates the matching process itself by counting matching polygraphs—taking no account of their position.

Each of these stages acts as a filter outputting fewer records than are input. The number of records output from the first filter is denoted $Y$ and the number delivered by the last is denoted $X$. The LikeIt facility sets $Y$ to the greater of $10 \cdot X$ and 1000. The effect of this tapered filter approach is that the final output is (in practice) as though the full algorithm, $F3$ were applied to every database record. The records output from each filter stage are maintained in a binary heap. Our implementation is simple and fast. As a result heap operations represent a negligible portion of the overall CPU time.

## 5.1 Front End Automaton

Each of the three filter stages operate on the query and database strings as a series of polygraphs of various lengths (single letters are 1-polygraphs). Matching edges can exist only between polygraphs that occur in both the query and the database string under consideration. Thus, all other data record polygraphs may be disregarded.

Our approach is to build a finite state machine (FSM) based on the query which detects all polygraphs (up to some fixed length) in a database record that also occur in the query. The machine changes state as each database record character is presented. It's states correspond to the longest trailing polygraph that is also present in the query.

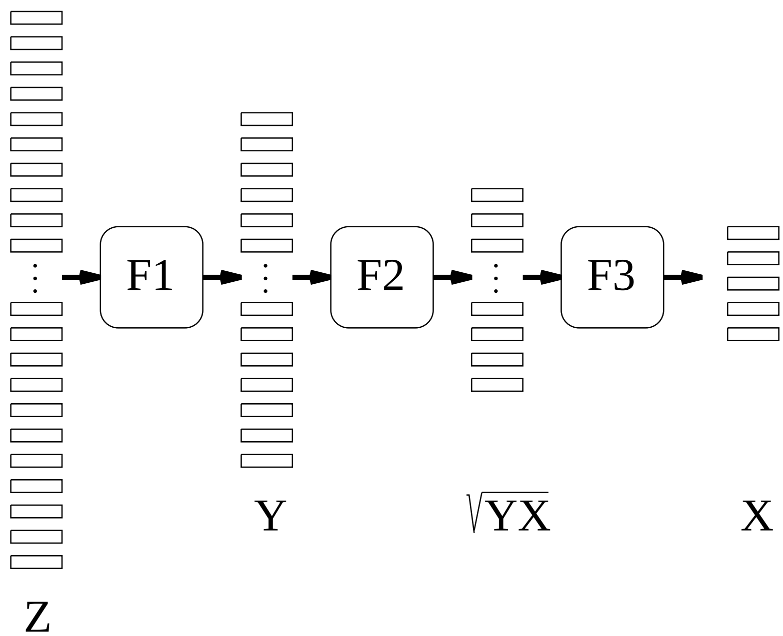


Figure 2: The LikeIt process consists of three stages (F1, F2, and F3) that filter an input database of $Z$ records down to the desired number $X$ of maximally similar output records. Each stage computes an increasingly effective but also increasingly CPU-intensive notion of similarity. Stage F1 outputs $Y$ records where $Z \gg Y \gg X$. The size of stage F2's output interpolates between $Y$ and $X$.

The machine's construction is straightforward but involved, and is described in the distribution file `fsm.doc`. We considered processing the database once using this machine, and saving the result. But because far fewer records are considered by $F2, F3$ than by $F1$, and because the machine is very fast when compared with $F2, F3$, we instead re-process the records for each filter stage.

## 5.2 Matching

The matching filters $F1, F2, F3$ operate on polygraphs identified by the FSM. Default operation limits attention to polygraphs of lengths 3–6 for filter $F1$, and lengths 1–6 for $F2, F3$.

In all filters a form of normalization is required, so that matching scores are comparable in the presence of variable-length queries and database strings. This normalization may be regarded as *padding* both query and database string to some large length $L$ that in the default case is 1024. In all cases, the effect of padding on the matching score is easily computed, and the padding is never actually performed.

Filter $F1$ counts matching polygraphs. Initialization identifies all polygraphs in the query within the requested range (3–6 by default). The count of each polygraph within the query is recorded. As the FSM processes database string characters and polygraphs are identified, $F1$ tallies *matches* up to the limit imposed by each polygraph's multiplicity in the query. For example, if "ing" occurs three times in the query, then only the first three occurrences in the database string contribute to the match tally. Database-string polygraphs that do not occur in the query are unmatched by definition. Unmatched polygraphs also include those that were not counted because they exceeded the query multiplicity limit.

Filter $F1$ takes no account of the relative position of matching polygraphs. It assigns a constant cost $0.5L$ to matching polygraphs, and cost $L$ to each pair that does not match. As such, it can be thought of as a relaxation of the later matching stages to trivial constant cost functions. Because position is irrelevant, the mutual alignment of query and database record is not an issue for $F1$. The final scores computed by $F1, F2, F3$ combine scores for each polygraph length and weight them linearly, i.e., polygraph lengths $1, \cdots, 6$ receive weights $1, \cdots, 6$, respectively.

The next stage, $F2$, begins with a left-aligned query, and decomposes the matching problem into subproblems for each *level* as defined in [1]. Each such level consists of polygraph occurrences that alternate between the query and database string. If the number of occurrences is even, the matching is uniquely defined. If odd, then LikeIt approximates the optimal matching by omitting either the first or last occurrence. The entire process is implemented without actually recording the matching edges—only costs are propagated online as database characters are processed. Investing only a little additional time yields the optimal matching [2]—also in an online fashion—but this is not implemented in LikeIt.

Having produced a matching, a single approximate realignment step is performed. This is accomplished by keeping track of the average edge length during processing, and mathematically repositioning the query so that the average length is as near as possible to zero. It is important to note that the matching itself is unchanged—edges are simply expanded or contracted to account for the realignment. For this reason we refer to this as a *free realignment*.

The final filter $F3$ begins as does $F2$ but after each realignment a new matching solution is constructed. This realignment/rematching step is performed three times, or until the score fails to improve. The mean-length approach to realignment taken by LikeIt is easily implemented, but we remark that the proper computation instead focuses on median length [2].

# 6 A More Detailed Interface

The operation of LikeIt is customized via the `expert` parameter to the `likeit()` function. This argument is of type `struct likeit_expert *`, and it is by filling in this structure that customization is effected:

```
struct likeit_expert {
    int (*char_map)(int);   /* Character mapping function */
    int max_record_len;     /* Max record len */
    int num_filters;        /* Number of filters */
                            /* Sequence of filters */
    struct likeit_filter filters[LKT_MAX_FILTERS];
};
```

The header file `likeit_expert.h` is included by applications that perform such customization. Passing `NULL` (or `LKT_DEFAULT_EXPERT`) for `expert` selects default operation.

The front-end character mapping is specified by `char_map`. This function maps input character codes to the values (range 1–255) that will be used by `likeit`. Common uses are to fold case, ignore most punctuation, and correspond foreign symbols with English letters. A sample implementation that folds upper to lower case and reduces all non-alphanumeric characters to *space* would be the following:

```
int likeit_default_char_map( int c )
{
    if (!isalnum(c))
        return ' ';
    else
        return tolower(c);
}
```

If `char_map` is set to `LKT_DEFAULT_CHAR_MAP`, a default character mapping function is used which adds to the above functionality the mapping of accented ISO Latin-1 characters to their unaccented lower case form.[4]

The `max_record_len` value should be set to a value larger than the longest string one expects to encounter. Value `LKT_DEFAULT_MAX_RECORD_LEN` defined as 1024 is assigned in the default structure. The result of processing strings longer than `max_record_length` is not specified.

The filter structure of LikeIt may be customized via the `num_filters` and `filters` structure elements. If `num_filters` is set to `LKT_DEFAULT_FILTERS` the default filter configuration described earlier is selected and `filters[]` need not be specified. If `num_filters` is set to a positive integer, then as many entries must be present in vector `filters[]`. The default structure specifies 3 corresponding to $F1, F2, F3$. Each vector entry is a `struct likeit_filter`:

```
struct likeit_filter {
    int type; /* Filter type */
    int gmin; /* Min polygraph length */
    int gmax; /* Max polygraph length */
    int hmax; /* Max # of records on filter output heap */
};
```

Structure element `type` is set to one of:

[4] This default character mapping function is actually implemented as a table lookup.

```
enum {
    LKT_FILTER_POLYCOUNT=0,
    LKT_FILTER_QCONVEX_1,
    LKT_FILTER_QCONVEX_2,
    LKT_FILTER_NUM_TYPES
};
```

corresponding to the three filter types described in section 5. For each, the `gmin` and `gmax` values are set to limit the polygraphs considered. The default configuration sets `gmax` to 6 for all three filter stages, `gmin` to 1 for the second two ($F2, F3$), and to 3 for the first ($F1$). Entry `hmax` determines the size of the corresponding filter output heap. Typically the final stage is configured to output exactly the number of matches requested by the user, and the output size of each stage is less than its predecessor's.

In addition to supporting customization using the filter types included with the distribution, this architecture anticipates the creation of new types.

# 7 Timings

We measure the facility's time performance by applying the `likeit` program to the database of [13]. This database is a listing of 50,360 papers in theoretical computer science. Each line gives the authors, title, journal or conference, and date. We reordered fields to match this description and added '`|`' between them. The resulting file is 4,278,967 bytes.

Our timings are made using an Intel Pentium Pro 200MHz processor with a 512K L2 cache under Linux version 2.0.27. The distribution was compiled using gcc -O3 version 2.7.2.1.

We focus on the time required to process a single character of database text, since this statistic enables application designers to quickly compute response time. This time, however, is not constant; the primary variables affecting it are the query's length and the number of best matches requested. Experiments verify that there is in practice much less sensitivity to the query's particular value, or the database's specific content.

We report results in table 1 for three queries of increasing length and requests for between 1 and 500 best matches. For example, 413ns per database character are required for query Q2 applied to our test database, where 25 best matches are requested. The response time for this query is then

$413ns \times 4,278,967 \approx 1.77$ seconds. It is also convenient to compute the processing rate $1/413ns \approx 2.4$ million characters per second.

The table also gives times for the three filter stages in the LikeIt process (described in section 5). The patterns evident in this table are consistent with the algorithm's design and we now remark on the qualitative nature of the timings. Notice that the F1 time is essentially constant for each query and varies little between them. We expect this because the role of F1 amounts to counting polygraphs in database records and there is very little dependency on the query. Filter 2 time depends rather strongly on query length since very similar processing takes place for each character of the database and query strings. For a fixed query F2 is essentially constant through 100 requested matches—but has increased considerably at 500. This is explained by our choice to set the number of matches output by F1 to the greater of ten times the number of requested matches, and 1000. So up to 100 requested matches, F1 always outputs 1000 candidates for consideration by F2. Requesting 500 matches forces F1 to output 5000 candidates, thus increasing almost linearly the F2 time required. For a fixed query we expect F3 time to increase with the number of candidates output by F2. For table values of 1, 5, 25, 100, and 500, filter F2 outputs 31, 70, 125, 316, and 1581 records, respectively, as determined by figure 2. Analysis of F3 time is complicated by the variable number of realignment steps performed. We expect, however, fewer such steps to be necessary as query length approaches the length of the database string. The table's F3 times are consistent with these two observations.

In summary, for each query the time varies by roughly 2:1 as the number of requested matches ranges from 1 to 500. The variation is somewhat less than this within columns. The corner-to-corner variation is just above 3:1. Excluding the 500 matches column the variation is much smaller. We therefore suggest that application designers can approximate performance well, at least for budgetary estimation purposes, by simply assuming:

> Each database character requires $\approx 400ns$ to process, corresponding to a rate of 2.5 MB/second.

Having said this, we must remark that this assumption breaks down in extreme cases such as databases consisting of very short records. Here, per-record overhead dominates.

| Query | Filter | # Matches Returned 1 | 5 | 25 | 100 | 500 |
|---|---|---|---|---|---|---|
| Q1 | F1 | 285 | 283 | 283 | 280 | 287 |
| | F2 | 40 | 42 | 42 | 42 | 168 |
| | F3 | 2 | 5 | 12 | 23 | 164 |
| | Total | 327 | 330 | 337 | 345 | 619 |
| Q2 | F1 | 301 | 299 | 299 | 301 | 306 |
| | F2 | 68 | 70 | 70 | 68 | 301 |
| | F3 | 12 | 21 | 44 | 108 | 285 |
| | Total | 381 | 390 | 413 | 477 | 892 |
| Q3 | F1 | 337 | 337 | 334 | 339 | 346 |
| | F2 | 98 | 98 | 96 | 98 | 437 |
| | F3 | 14 | 26 | 54 | 108 | 285 |
| | Total | 449 | 461 | 484 | 545 | 1068 |

| | |
|---|---|
| Q1: | Optimal |
| Q2: | Optimal Dictionary Problem |
| Q3: | Andersson, Optimal Bounds on the Dictionary Problem |

Table 1: Processing time (nanoseconds) per character of database text for three queries of increasing length and requests for between 1 and 500 best matches. The total time as well as times for each filter stage are shown. Measurement error is approximately $\pm 2$ ns for each filter.

## 8 Future Work

Because the similarity values computed by LikeIt arise from a graph matching computation it should be possible to visualize the system's judgment. The edges themselves might be shown, or, matching letters and polygraphs somehow highlighted. We view this as an important aspect of intelligent user interfaces because the user who better understands the machine's judgment will, we believe, be better equipped to employ it.

We also intend to modify $F3$ to exactly solve each subsidiary matching problem, and use median-based realignment as described in [2].

Two interesting areas for theoretical work are evident. First, an additional filter stage before $F1$ might be added to prune the search without examining every record. Second, online forms of our matching-based approach suggest a system similar to LikeIt for searching text databases that are not record-delimited. Some progress towards this goal is described in [2]. We remark, however, that the LikeIt system as described in this paper can be used effectively for such databases by simply introducing record divisions periodically, or at appropriate locations such as sentence or paragraph boundaries.

## References


[1] S. R. Buss and P. N. Yianilos, *Linear and $o(n \log n)$ time minimum-cost matching algorithms for quasi-convex tours*, in Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 65–76. To appear SIAM Journal on Computing.

[2] ——, *A bipartite matching approach to approximate string comparison and search*, tech. rep., NEC Research Insitute, 4 Independence Way, Princeton, NJ, 1995.

[3] M. Damashek, *Gauging similarity with n-grams: Language-independent categorization of text*, Science, 267 (1995), pp. 843–848.

[4] P. A. V. Hall and G. R. Dowling, *Approximate string matching*, Computing Surveys, 12 (1980), pp. 381–402.

[5] S. Huffman and M. Damashek, *Acquaintance: A novel vector-space n-gram technique for document categorization*, in Proceedings Text RE-

trieval Conference (TREC-3), Washington, D.C., 1995, NIST, pp. 305–310.

[6] R. M. Karp and S.-Y. R. Li, *Two special cases of the assignment problem*, Discrete Mathematics, 13 (1975), pp. 129–142.

[7] D. E. Knuth, J. J. H. Morris, and V. R. Pratt, *Fast pattern matching in strings*, SIAM Journal on Computing, 6 (1977), pp. 323–350.

[8] U. Manber and S. Wu, *GLIMPSE: A tool to search through entire file systems*, in Proceedings of the Winter 1994 USENIX Conference, 1994, pp. 23–32.

[9] M. J. Miller, *First look – friendly program doesn't need exact match to find database search objects*, INFO WORLD, (1987).

[10] Proximity Technology, Inc., *Friendly finder*. Commercial Software for the IBM-PC, 1987.

[11] S. Rosenthal, *The PF474 – a coprocessor for string comparison*, BYTE Magazine, (1984).

[12] D. Sankoff and J. B. Kruskal, *Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, 1983, 1983.

[13] J. Seiferas, *A large bibliography on theory/foundations of computer science*. ftp://ftp.cs.rochester.edu, 1996-7.

[14] S. Wu and U. Manber, *Fast test searching allowing errors*, Communications of the ACM, 35 (1993), pp. 83–91.

[15] P. N. Yianilos, *The definition, computation and application of symbol string similarity functions*, Master's thesis, Emory University, Department of Mathematics, 1978.

[16] ——, *A dedicated comparator matches symbol strings fast and intelligently*, in Electronics Magazine, McGraw-Hill, December 1983.

[17] ——, *Data structures and algorithms for nearest neighbor search in general metric spaces*, in Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93), 1993, pp. 311–321.

# A The Distributed Software

After unpacking the distribution, you will need to make sure that the following parameters are set properly in the `Makefile`: `CC`, `CFLAGS`, `RANLIB`, `OS_TYPE`, `SERVERADDR`, `SERVERPORT`, and `TIMEOUT_VALUE`. See the associated explanatory comments in the `Makefile`. The last three of these parameters are only critical if you plan to run the LIKEIT daemon and CGI stub.

The `Makefile` is configured for Linux. Solaris users will need to edit the definitions at the top of the `Makefile`. Now type `make`. This makes targets `liblikeit.a`, `likeit`, `likeit.cgi`, `likeitd`, `in.likeitd`, and `likeitstub.cgi`. Library `liblikeit.a` is derived from the following distributed files:

| | |
|---|---|
| `likeit.c` | Top-level module implementing the `likeit()` routine |
| `likeit_sys.h` | Private top-level header file |
| `fsm.c` | Search query automaton |
| `fsm.h` | Private header file associated with `fsm.c` |
| `invtab.c` | Inverted table of query polygraph occurrences |
| `invtab.h` | Private header file associated with `invtab.c` |
| `heap.c` | Fast binary heap |
| `heap.h` | Private header file associated with `heap.c` |
| `polycount.c` | Polygraph counting filter |
| `qconvex.c` | Quasi-convex filters |
| `charmap.c` | Default character mapping function |
| `array.c` | Multi-dimensional arrays, used by `fsm.c` |
| `array.h` | Private header file associated with `array.c` |
| `common.h` | Private header file containing common definitions |
| `mem.c` | Storage allocation routines |
| `timer.c` | Timer routine |
| `utils.c` | Miscellaneous utilities |

Users of `liblikeit.a` include:

| | |
|---|---|
| `likeit.h` | Contains the `likeit()` function prototype |
| `likeit_expert.h` | Definitions for expert configuration |

The `likeit` and `likeit.cgi` programs are derived from:

| | |
|---|---|
| `cgi.c` | Top-level CGI search code |
| `data.c` | Database management routines |
| `display.c` | Match display routines |
| `likeit_prog.c` | Main program module |
| `likeit_prog.h` | Top-level header file |
| `state.c` | Routines for modifying program state |

The `likeitd`, `in.likeitd`, and `likeitstub.cgi` programs are derived from:

| | |
|---|---|
| `svr_main.c` | Main daemon program module |
| `svr_data.c` | Database management routines |
| `cli_main.c` | Main stub program module |
| `clisvr.h` | Common definitions for daemon and stub |
| `systemtype.h` | OS type-dependent definitions |
| `ipc.c` | Socket communication routines |
| `ipc.h` | Socket communication header file |
| `ipcutils.c` | Communications utilities |
| `error.c` | Error reporting routines |
| `error.h` | Error reporting header file |

To try the Web-based demonstration software, install `likeit.cgi` in the typical fashion, i.e., copy it into the `cgi-bin` directory of your local HTTP server. You'll need to edit `likeitform.html` so that the absolute pathnames for files `rochester.lst` and `catalog.lst` are correct for your system, and so that `likeit.cgi` is properly pointed to. Two sample databases `rochester.lst` and `catalog.lst` are provided.

To install the daemon and CGI stub programs, copy `likeitstub.cgi` into the `cgi-bin` directory of your local HTTP server. The HTML form `likeitd.html` is used to invoke the stub, and you will need to edit it so that it points at the stub program correctly. Note that the option values corresponding to the database selections are integers rather than pathnames. These integers correspond to the order in which the databases are listed in the daemon configuration file `likeitd.config`. Make sure that the pathnames for these databases are correct, *absolute* pathnames in the configuration file.

If you wish to start the daemon manually, type

```
likeitd <path>/likeitd.config &
```

where the path is the *absolute* pathname of the configuration file. The daemon detaches from the controlling terminal, so its execution can only be terminated with `kill`. Make sure that the daemon is running on the host you specified in the `Makefile` via the `SERVERADDR` parameter!

If you wish the *inetd* Internet superserver to start the LikeIt daemon, you must have your system administrator modify two systems files. First, a line similar to the following must be added to the file `/etc/services`:

```
likeit 5000/tcp # likeit daemon
```

In place of the number `5000`, use the port number you decided on for the `SERVERPORT` parameter in the `Makefile`. This is the port the LikeIt daemon will listen on.

Next, a line similar to the following must be added to the file `/etc/inted.conf`:

```
likeit stream tcp wait <user> <path>/in.likeitd <path>/likeitd.config
```

The LikeIt daemon `in.likeitd` will run as `user` Paths should be correct for the executable `in.likeitd`, and the daemon configuration file `likeitd.config`. These must be *absolute* pathnames. The *inetd* superserver will start the LikeIt daemon at the first attempt of the stub program to connect. The daemon will read in the databases specified in the configuration file, and then begin to accept queries. Error messages generated by stub and daemon are written to the files `/tmp/likeitstub.console` and `/tmp/likeitd.console`, respectively.

File `likeit.ps` contains Postscript for this paper and `LICENSE` gives the noncommercial license under which the distribution is made. File `fsm.doc` provides details regarding the finite state machines used to detect polygraphs in the LikeIt filters. A simple `README` file directs the reader to this paper.