Isearch Internals

Brent Baccala
`baccala@freesoft.org`

August, 2001

1 Introduction

Isearch (`http://www.etymon.com/Isearch/`) is a full-text searching system published under an open source license. It indexes a collection of documents to facilitate rapid searches, for example to find all instances of the word *ethernet* in a gigabyte of data spread over a thousand files. This paper describes the internals of Isearch v1.47, focusing mainly on the structure of the index files on disk. Other documents in the Isearch distribution, particularly the *IsearchTutorial*, discuss the use of Isearch's programs from a user's perspective.

The document collection is a set of ordinary files, residing somewhere on the system, and must be readable during searches, since the Isearch index only stores pointers to the words, not the words themselves. Each document has an associated *document type*, indicating the structure of the document and implying how the file should be parsed into records and fields. Isearch comes with over two dozen document types defined, and provides an extension API to allow new document types to be coded in C and compiled into the Isearch programs.

The collection is modeled as a set of records, each corresponding to a range of bytes within a single file. The results of a search operation are sets of records. In the simplest case, each record corresponds to a single file on disk, but other possibilities exist. For example, the `ONELINE` document type indexes each line of a file as independent records. Each line thus appears independently in the result sets. The `MAILFOLDER` document type parses standard UNIX mail folders and treats each message as a record, producing search results containing individual email messages.

The record can be further parsed into *fields*, for example the title of an HTML document, or the sender of an email message. Fielded search operators allow search terms to be restricted to individual fields. For example, searching for `From/baccala` in a `MAILFOLDER` document matches only those records containing `baccala` in the `From:` header. However, the result set would still indicate the entire email message, since fields are used to restrict search operations, and records are the fundamental entity returned from the searches.

The Isearch index itself consists of a set of files on disk, identified by a common prefix, and using a set of standard file extensions. All of the Isearch utilities accept a `-d` option, which specifies a UNIX path name to reach the index files. No file actually exists by the name specified with `-d`; the software appends the various file extensions to find individual files. For example, specifying `-d index/topics` indicates that `index/topics.mdt` is the Master Document Table, `index/topics.dbi` is the Database Info file, etc.

File Extension	**Description**
`.dbi`	Database Info file
`.dfd`	Data Field Definitions Table
`.inx`	Index
`.inx.`*num*	Index number *num*
`.mdt`	Master Document Table
`.mdk`	MDT Key Index
`.mdg`	MDT GP Index
`.sta`	Database State file
`.`*num*	Data field file *num*

Figure 1: Isearch file extensions

2 Database Info file

The `dbi` file is a small, formatted ASCII file containing overall information about the index. It's structure is simple — each key or value occupies its own line, begun with a "+", and the indentation level indicates the line's position in the hierarchy. The Isearch code refers to it as a "registry", and it's structure is clearly patterned after the Windows registry. Here's a sample `dbi` file:

```
+DbInfo
 +VersionNumber
  +1.47d
 +MagicNumber
  +7
 +DocType
  +
 +DocTypeOptions
  +
 +BigEndian
  +0
```

The most important entries are `VersionNumber`, indicating the version of Isearch that generated the index; `MagicNumber`, indicating the revision of the database format, and thus which versions of the program can read this database; `BigEndian`, indicating the byte ordering in the 32-bit integer values; and `DocType`, the default document type to be used if none is specified on the `Iindex` command line.

3 Master Document Table

The Master Document Table (MDT) contains an entry for every record in the document collection. Stored in a file with extension `.mdt`, the MDT entries currently (Isearch v1.47) are 1380-byte C structures, as shown in Figures 2 and 3.

Isearch indexes documents using a 32-bit index, called a global position (GP), which maps to character positions in the files. Each record maps to exactly the same number of GPs as there are byte in the record, and no overlap is allowed between two record's GP ranges. To resolve a GP, you look it up in the Master Document Table (MDT) which lists initial and final GPs (*GlobalFileStart* and *GlobalFileEnd*) for each record in the collection. After finding the MDT entry, subtract the record's *GlobalFileStart* to get a character offset in the record. The MDT entry will also specify a *LocalRecordStart* — the character offset of the record in the file.

```
typedef UINT4 GPTYPE;

const INT DocumentKeySize  = 16;
const INT DocumentTypeSize = 64;
const INT DocFileNameSize  = 1024;

class MDTREC {
  CHR Key[DocumentKeySize];
  CHR DocumentType[DocumentTypeSize];
  CHR PathName[DocPathNameSize];
  CHR FileName[DocFileNameSize];
  GPTYPE GlobalFileStart;
  GPTYPE GlobalFileEnd;
  GPTYPE LocalRecordStart;
  GPTYPE LocalRecordEnd;
  CHR Deleted;
};
```

Figure 2: Record structure of the `.mdt` file

So, adding the *LocalRecordStart* to the character offset in the record gives the character offset in the file, which can simply be opened and read to find the word.

For example, assume `index.htm` is 400 bytes long and maps to global indices 100001-100400. Then global index 100200 corresponds to the 200th byte in `index.htm`

This snippit of Perl will read and unpack entry `$num` from an MDT:

```
if ($DBI{BigEndian}) {
    $pack_template = 'Z16Z64Z255Z1024xNNNNCxxx';
} else {
    $pack_template = 'Z16Z64Z255Z1024xVVVVCxxx';
}

seek(MDT, 1380 * $num, 0);
read(MDT, $mdtentry, 1380);

my ($dockey, $doctype, $pathname, $filename,
    $gfstart, $gfend, $lrstart, $lrend,
    $deleted) = unpack($pack_template, $mdtentry);
```

Since the MDT can become fairly large, searching it quickly can be a problem. To speed searches, the `.mdt` file has two auxiliary files associated with it, each containing the same number of records as MDT, but with just a few fields, and sorted. The GP index (extension `.mdg`) is sorted by global position, and the key index (extension `.mdk`) is sorted by document key. Because they are relatively small, the `.mdg` and `.mdk` files are typically read into memory; further disk accesses needed only to read the full entries from the `.mdt` file. Figure 4 shows the structure of the key records.

For example, Figure 5 shows a directory listing of these files for one of my databases. The document collection contains 2926 files, so each file contains 2926 records, of 12 bytes each (MDG), 20 bytes each (MDK) and 1380 bytes each (MDT). Finding the MDT entry corresponding to a given GP (a fairly common operation during searches) requires a binary search on the in-memory copy of the `.mdg` file, which yields the *Index* of the entry in the `.mdt` file.

Field Name	**Description**
`Key`	Key number of the document. Each document has a unique decimal key, used to uniquely identify it, during delete requests for example.
`DocumentType`	One of the defined document types
`PathName`	UNIX-style pathname leading to document file. Usually absolute (with a leading slash), but can be relative (without one).
`FileName`	Filename of the document, without any leading path.
`GlobalFileStart` `GlobalFileEnd`	The range of global positions (see Introduction) referred to by this MDT entry. The length of the record (usually the length of the file) is $GlobalFileEnd - GlobalFileStart + 1$
`LocalRecordStart` `LocalRecordEnd`	`LocalRecordStart` and `LocalRecordEnd` indicate the limits of the record, measuring in bytes from the start of the file, the first byte counted 0. $LocalRecordEnd - LocalRecordStart + 1$ is the length of the record in bytes, and always equals $GlobalFileEnd - GlobalFileStart + 1$
`Deleted`	A single byte value, non-zero if the document has been deleted from the collection.

Figure 3: Fields in the Master Document Table

```
class GPREC {
public:
        GPTYPE GpStart;
        GPTYPE GpEnd;
        GPTYPE Index;
};

class KEYREC {
public:
        CHR Key[DocumentKeySize];
        GPTYPE Index;
};
```

Figure 4: Record structure of the `.mdg` and `.mdk` files

```
-rw-rw-r--    1 baccala  baccala     43656 Jun 27 17:09 htmlrfcs.mdg
-rw-rw-r--    1 baccala  baccala     72760 Jun 27 17:09 htmlrfcs.mdk
-rw-rw-r--    1 baccala  baccala   5020440 Jun 27 17:09 htmlrfcs.mdt
```

Figure 5: UNIX directory listing of MDT and related files

4 Index Files

Isearch indices are contained in one or more index files. If a single index file is present, its extension is simply `.inx`. Multiple index files are indicated using a `.num` file, which contains a single ASCII decimal number, the number of index files. Each of these index files has extension `.inx.`, followed by the number of the file. For example, if the `.num` file contains `3`, then the three index files have extensions `.inx.1`, `.inx.2`, and `.inx.3`. If multiple index files are present, then any `.inx` file that may be present is ignored.

Each index file is nothing more than a list of 32-bit global positions, stored using the byte ordering indicated by the DBI file. Each GP points to the first byte of a word in the document collection, and the GPs in each index file are sorted by the ascending alphabetical order of their corresponding words. The command `od -t 'd4'` can be used to conveniently print the contents of an index file. For example, here's a simple one line document, with each GP marked, and its corresponding 24 byte index file:

```
This is my web page
^    ^  ^  ^   ^

0    5  8  11  15

Index file:    5   8   15   0   11
```

Each index file is sorted independently of the others, and any search is performed on all the index files, with the result sets or'ed together. This allows documents to be incrementally added to the collection, simply by creating another index file. Of course, the more index files there are, the less efficient the searches become. Isearch provides an optimization function to combine multiple index files together, using a merge sort, with the `.inx` file (remember, it's ignored if there are multiple index files) as output. The various index files are scanned from beginning to end, taking always the smallest term and writing it to the output. This continues until all the index files are exhausted. Then they are deleted, along with the `.num` file, and Isearch begins using the newly created `.inx` file.

5 Search Algorithm

Isearch finds words using binary search on the index file(s). Some of the items in the search index may be invalid, if they point to deleted data, and are thus unable to be used in a comparison operation. Thus, Isearch's binary search algorithm has been modified to deal with the possibility of invalid entries.

The primitive search operation is `MatchMid`, which is passed a range `[low, high]` and the search term to compare against. `MatchMid`, in its simplest form, takes the term at the midpoint of `[low, high]`, compares it to the search term, and returns equal, less than, or greater than. Since each entry in the index is a global position, finding the corresponding term requires a binary search on the MDT to find the file corresponding to the GP, opening the file, seeking to the location, reading the term, then comparing it to the search term. If there are no invalid terms present, then `MatchMid` simply performs a comparison on the midpoint of `[low, high]`.

However, `MatchMid` also must handle an invalid midpoint, in which case it does linear searches away from the midpoint, looking for the first valid term in both directions. In the general case, `MatchMid` returns a range, `[midl, midr]`, the endpoints being valid terms, and everything between them invalid. As a special case, the linear search may reach all the way to the original `low` or `high` limit without finding a valid term, in which case it terminates and returns a special marker to indicate this case.

To search for a term, we binary search on a range `[low, high]`, initially the entire range of the index file. `MatchMid` is called and returns a range `[midl, midr]` near the midpoint of `[low, high]`.

Figure 6: MatchMid

Consider the following cases:

- If either `midl` or `midr` matched equal to the search term, the binary search succeeded and we immediately return
- If `midl` is greater than the search term, set the `high` limit of the search to `midl-1` and repeat, since `midr` is always greater than `midl` and therefore the entire `[midl, midr]` range is greater than the search term
- If `midr` is less than the search term, set the new `low` limit of the search to `midr+1` and repeat since `midl` is always less than `midr` and therefore the entire `[midl, midr]` range is less than the search term
- Otherwise, return an empty result. `midl` must be either less than the search term, or have reached the lower search limit finding nothing but invalid terms, since both the equality and greater than cases have already been addressed for `midl`. Likewise, `midr` must be either greater than the search term, or have reached the upper search limit, since its equality and less than cases have already been addressed. So, `midl` is less than the search term (or limit reached), and `midr` is greater than the search term (or limit reached), so any hits must lie in the range between `midl` and `midr`, which contains nothing but invalid entries, so there are no valid results

Repeat the binary search algorithm until either the first or last case triggers an immediate return, or until `high > low`, in which case return an empty result. Since the iteration steps each set `low` (or `high`) to one more (less) than `midr` (`midl`), the algorithm always makes forward progress, until `low = high` and the last step increments `low` past `high`, or decrements `high` past `low`.

Isearch needs to find all the matching terms, so once a hit is found, two additional binary searches are performed to bracket the range of matching terms, using the location of the hit, and the `[low, high]` values from the last iteration. First, a binary search is performed on `[low, hit]`, looking for the beginning of the matching range, then another binary search is performed on `[hit, high]`, looking for the end of the matching range.

Consider the search for the beginning of the range. Call `MatchMid` on `[low, hit]` to get `[midl, midr]`, then apply the following cases. The upper limit of our search (`hit`) compares equal to the search term, and we'll maintain this as an invariant. Also, we'll never get a comparison result greater than our search term, only less than or equal.

- If `midl` matched equal to the search term, set `hit` to `midl` and repeat
- If `midr` matched less than the search term, set `low` to `midr+1` and repeat
- Now, `midl` must be either less than the search term, or have reached the low limit finding nothing but invalid entries, and `midr` must be equal to the search term, since `hit` is always valid, so `midr` can't have reached the search limit. Furthermore, everything between `midl` and `midr` is invalid, therefore `midr` is the lowest matching entry, so return it as the result of our search

Repeat until `low = hit-1`. The algorithm will make forward progress until this point, since `midl` will always be less than `hit` until this point, where it may stall (if `midl = midr = hit`). Compare `low` to the search term. If it is equal, return `low` as the result, otherwise return `hit`. A similar algorithm is used to find the upper limit of the search range.

Performance. Since the algorithm reverts to linear search when confronted with invalid entries, its performance depends strongly on the density of such entries. In the worst case scenario, when the entire search array is invalid, the algorithm will require linear time to return an empty result. On the other hand, in the degenerate case with no invalid entries, the algorithm acts as a traditional binary search and requires `log(N)` time. Clearly, this algorithm is suited only for indices with few deleted entries. Isearch's optimization function, which eliminates deleted entries, should be used as often as possible when deleted entries may be present.

6 Fields

Fields, as mentioned above, are byte ranges within records. Normally, a search term will match anywhere within a record, but can be limited to a specific field. Each field has an entry in the Data Field Definitions Table (`.dfd` file), which is a small ASCII file listing field names, an associated file number, and one or more attributes, identified by OIDs.

Each field has a file number associated with it, and this file number is used as an extension (`.`*num*) to name the field's associated Data Field file. These files contain pairs of 32-bit GPs — the starting and ending GPs of each field. The GP pairs are sorted in ascending order, allowing a binary search to determine if any GP is contained in the field in question. A fielded search thus begins with a normal search, which returns a list of GPs, which are each checked against the Data Field file and rejected unless within one of its ranges.

Numeric fields are supported, and are handled specially. The numbers are parsed by the document type handler and stored, each one, as a GP/double pair in the Data Field file. This special format of Data Field file is sorted by the double (not by the GP), allowing binary searches for numbers directly on the Data Field file without using the standard index files at all, though the numbers are also indexed there, in the conventional manner, for non-fielded searches. Dates, for example, are converted to numbers and stored in this fashion.

7 Parsing and Scoring

When a search query is presented to Isearch, it first scans the query into tokens. Tokens are single words, separated by whitespace, with several special character sequences that parse into unique tokens — the two parenthesis, `&&`, `&!`, and `||`. Double-quoted strings are grouped as single tokens without regard for whitespace or special characters, then the double quotes are stripped away.

Next, the query is parsed according to one of several grammars (standard, and, infix, and RPN) selected by command line options. In a standard query, every token is regarded as a search term, with the results logically or-ed together. An *and* query is almost the same, but the result sets from each term are logically and-ed. Infix notation is converted into RPN, and both use a set of operators to construct more complex queries — `AND` or `&&` (logical and), `OR` or `||` (logical or), `ANDNOT` or `&!` (logical andnot), `NEAR`, and the parenthesis for grouping. I hope logical *and* and *or* are self-explanatory. Logical *andnot* is an *and* with the following search term inverted, but since inverted terms tend to generate enormous result sets, logical *not* isn't provided as a primitive. `NEAR` matches its two arguments within 50 characters of each other.

Each search term within the query is parsed like `(.+)(*)?(/.+)?(:-?[0-9]+)?`, i.e, the search term itself, followed by an optional asterisk to indicate a wildcard, followed by an optional slash and field name to restrict the search, followed by an optional colon and numeric weight (possibly negative). For example, `ethernet*/title:3` searches for any word beginning with `ethernet` in the `title` field, weighing this term three times more than normal. If double quotes were used, an entire phrase can be submitted as a search term.

For each search term, the search algorithm described above is used to find a range of index entries that match the search term in each index file. Each GP in the range(s) is read, possibly rejected if not present in a required field (which would require the GP to searched for in the proper Data Field file), and looked up in the MDT to find its matching document. The hit count for that document is incremented. There is no need to open the document files on disk at this point, but each index entry triggers a binary search on the MDT, and possibly the Data Field file, if a field was specified. This is the most time consuming part of a search.

Next, the hit counts are normalized by summing the squares of the hit counts for each document, and dividing through by the square root of the sum. The score is multiplied by the weighting factor associated with this term in the search query, default 1. This produces a set of document scores for a single term.

$$score_n = weight \frac{hits_n}{\sqrt{\sum hits^2}}$$

The full search query is now constructed from the weighed, normalized result sets from each term. A logical *and* produces a result set containing only documents that appeared in both of its input result sets; the scores from the two input sets are added together to get the scores on the output result set. A logical *or* produces the union of documents in its input result sets; again the scores are added together for any document appearing in both sets. The logical *andnot* removes from the result set of its first argument any documents in the result set of it's second argument; the scores of the remaining documents are unchanged. For `NEAR`, the scores from the left-hand result set are discarded; the scores from the right-hand result set are used, multiplied by the number of hits within 50 characters of each other.

Finally, having produced a result set for the entire search query, the document list is sorted into descending order of scores, the final scores are normalized so that the highest score is 100, and the list is presented to the user.