THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-08-02

Implementation of PIS

Paul Thomas

June 2008

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical-DOT-Reports-AT-cs-DOT-anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

Implementation of PIS

Paul Thomas, paul.thomas@anu.edu.au

Abstract

PIS, a *p*ersonal *i*nformation *s*earcher, provides a unified "metasearch" interface to any number of search engines so that it is possible, for example, to search email; calendars; websites; and local files simultaneously. It includes modules for a number of search engines as well as the typical metasearch tasks of search engine characterisation and selection, and includes support for user experiments.

This document briefly describes the implementation of PIS, with emphasis on writing any future extensions.

1 Introduction

PIS, a *p*ersonal *i*nformation *s*earcher, is a working personal metasearch tool which has been used in the experiments of Thomas [2008b]. It operates with the hybrid model described by Craswell et al. [2004], acting as a front-end to servers where available and doing its own indexing otherwise. It implements each of the stages in the metasearch process of Thomas [2008b] (§2.2.2), with the exception of server discovery, and includes implementations of several algorithms for server characterisation, server selection, and result merging.

Figure 1 illustrates PIS searching a number of collections and returning a single ranked list, including pieces of email, entries in a library catalogue, BibTeX records, and results from Project MUSE.[1]

PIS represents around 15,000 lines of C# code, plus 42,000 in included third-party libraries, and has been tested on Linux (with the Mono .NET interpreter[2]) and on Windows.

As a hybrid metasearcher, PIS offers a single interface to any number of independent search engines and to any number of collections indexed by PIS itself (Figure 1).

To support embedded comparisons of the type discussed in Thomas and Hawking [2006], PIS can display results in two or more independent panels, each with its own algorithm for server selection and result sorting and merging (Figure 2). It also implements the additional feedback options described by Thomas and Hawking, which allows experimenters to distinguish brilliant success from abject failure and provides confirmation of implicit judgments. These extra feedback features are illustrated in Figure 3.

PIS is available from the author.

[1] http://muse.jhu.edu/

[2] http://www.mono-project.org/

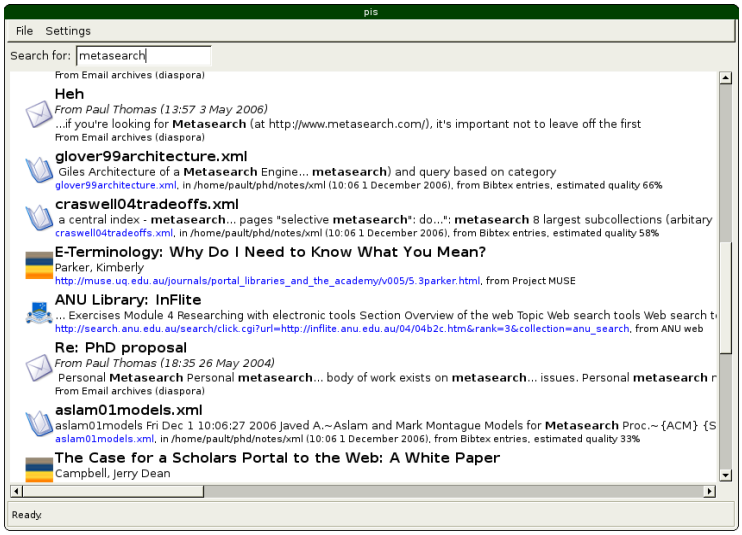


Figure 1: The PIS software.

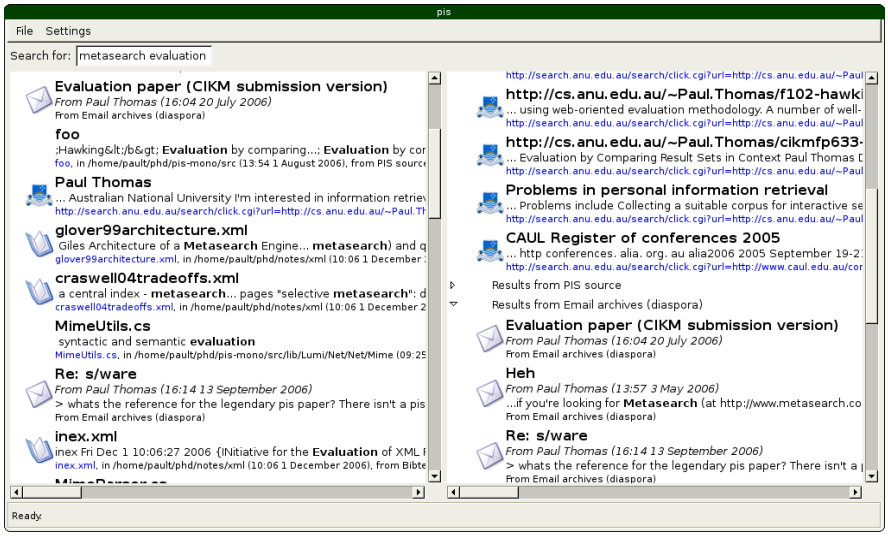
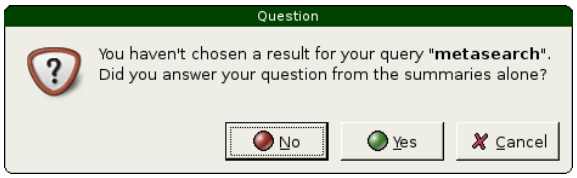
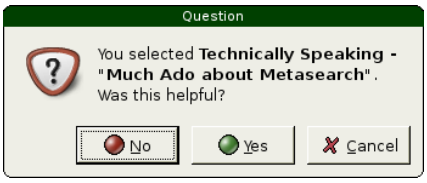


Figure 2: PIS with two panels, for evaluations in the manner of Thomas and Hawking [2006].



(a) When no result is chosen



(b) When a result has been chosen

Figure 3: Extra feedback from PIS.

2 Running PIS

2.1 Requirements

PIS requires a .NET runtime, the GTK+ interface library, and the .NET bindings to GTK+. Under Windows, the last two are available in a bundle from the Mono project. Under Unix, the Mono project provides an interpreter and the GTK+ bindings, and the GTK+ library itself is widely available.

2.2 Invoking PIS

PIS can be started by double-clicking the `pis.exe` icon (Windows) or running `bin/pis` (Unix).

The commandline version accepts the following options:

/conf *filename* Reads configuration details from the named file. By default, PIS looks for `config/config` in an appropriate platform-specific place (e.g. `$HOME/.pis` on Unix).

/debug Logs (verbosely) to standard error.

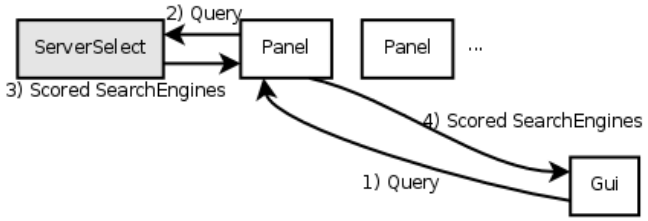
/indexonly Rebuilds any indexes which PIS maintains itself, then exits. PIS will not show a GUI.

2.3 Queries

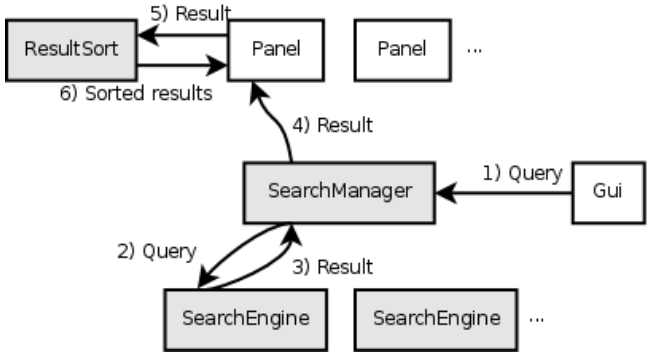
This section uses a simplified grammar to describe queries understood by PIS. A "string" is any sequence of characters except whitespace; a string cannot contain a solidus ("/") or comma (","), where these are used to separate fields.

A query to PIS consists of two parts: first an optional server specification, which allows manual server selection; then one or more query terms. If a server specification is not included, one or more server selection methods will run on the user's behalf.

query → [*server-specification* "/"] *terms*



(a) Server selection



(b) Searching

Figure 4: Rough architecture of PIS

Each server in PIS has a unique name, generally descriptive, such as "email archives on localhost" and any number of "tags": short names such as "email". Tags need not be unique, which gives a shorthand way to name a number of servers at once.

server-specification → *name-or-tag* [", " *server-specification*]
name-or-tag → *server-name* | *server-tag*
server-name → *string*
server-tag → *string*

Whitespace-separated terms in a query are not parsed further, but passed as-is to each selected server.

terms → *term*+
term → *string*

For example, the query "`email/nancy`" searches for "nancy" in all servers named or tagged "email". "`fred astaire`" searches for documents matching "fred", "astaire", or ideally both, in whichever servers are most appropriate. As a final example, "`news, web/sports`" searches for "sports" in servers named or tagged "news" or "web".

3 Overview

The overall architecture of PIS is briefly described below and illustrated in Figure 4. Much of PIS relies on modules loaded at runtime; Section 4, following, describes those modules already implemented and summarises how to add more.

3.1 User interface

The user interface is managed by a *Gui* object (`ui/gui.cs`). The major part of the interface is one or more *Panel*s (`ui/panel.cs`), which are responsible for displaying results from the search engines.

To support embedded comparisons of the type described in Thomas and Hawking [2006], each Panel can perform its own server selection and result sorting. Selection is handled by a per-panel *ServerSelect* object (`select/select.cs`), which returns a score for each search engine for each Query; this is used if the query does not include manual server selection, and only search engines selected by one or more ServerSelect algorithms will be used. Result sorting is performed by a per-panel *ResultSort* object (`sort/sort.cs`), which is called each time a result is received from a search engine's queue.

3.2 Searching

The fundamental task of searching in PIS is carried out by *SearchEngine*s (described in `search/base.cs`), each of which is responsible for retrieving documents from some individual collection. Search engines, each of which runs asynchronously in its own thread, receive *Query* objects (`search/query.cs`) from the application and communicate their results to the rest of PIS via a *ResultQueue* (`search/base.cs`). To facilitate server selection, search engines each include a *Model* (`search/model.cs`) recording real or estimated term frequency data.

PIS maintains a single *SearchManager* (`search/base.cs`), which in turn maintains a list of search engines. Every query includes a set of search engines named in the *server-specification* part of the query text (Section 2.3); the search manager receives these queries, forwards them to the selected set of search engines, and manages search bookkeeping. If no search engines are specified, *ServerSelect* objects from each panel are used.

Results of a search are objects of type *Result* (`search/result.cs`). Typically these are *DisplayResult*s (`search/result.cs`), representing a result which is to be displayed to the user and can be selected from the GUI, or *DocumentResult*s (`search/result.cs`), representing a single document with a URL and an icon. Further subclasses of DocumentResult include *WebResult* (`search/result.cs`), which loads a "favicon" from the Web in a background thread and *PersonResult* (`search/result.cs`), representing a person with details such as name and email address. Search engines implemented in loaded modules can add further result types (for example, *EmailResult* in `search/modules/email/email.cs`).

A *ResultEndMarker* (`search/result.cs`) signals the end of a list of results, and a *ResultError* (`search/result.cs`) can be queued to indicate an error condition at the server.

3.3 Configuration

PIS's configuration is maintained by a *Configuration* object (`ui/configuration.cs`), which is able to read and write an XML file. Details can be edited using the GUI via a *ConfigurationWindow* (`ui/configwindow.cs`), which automatically generates configuration windows for each module.

3.4 Harness

The main loop of PIS (*Harness*, in `ui/main.cs`) runs as follows:

1. Parse the command line (Section 2.2);
2. load any modules (Section 4);
3. set up the GUI (Section 3.1), unless the "`indexonly`" option was given;
4. if the "`indexonly`" option was given, start an indexing run;
5. otherwise, set up a result queue, a search manager, and a GUI; then start the GUI.

4 Modules

PIS uses plug-in modules to provide key metasearch features. The present set of modules are described below.

4.1 General notes

PIS loads all .DLLs in the correct directory except those whose names start with "Pis.", which are assumed to be built-in (see *Harness.LoadModules()*). If a type has a static *OnLoad()* function, this is called when PIS loads the containing module. All types loaded this way can be retrieved via *TypeRegistry.ExtraTypes*, which is used mainly for XML serialisation.

4.1.1 Serialisation

The *Configuration* class writes to and reads from a file containing serialised objects (stored as XML): this includes *SearchEngine*s, *ServerSelect* implementations, *ResultSort* implementations, and *Model*s, as well as their members. *Result* subclasses are also (de)serialised by the external search engine (Section 4.6.5). In each case all public properties are serialised by default.

A registry (*TypeRegistry.ExtraTypes*) is maintained which lists all types which may be encountered in reading or writing XML. Types are added to this registry when their defining modules are loaded, or they can be explicitly added with *TypeRegistry.Add(Type t)*.

The serialisation is handled by standard .NET libraries, and can be controlled with attributes such as *XmlElement* and *XmlIgnore* (q.v.).

4.1.2 Exposing configuration options

Any public properties of search engines, and of the *Configuration* object itself, are by default exposed for user configuration. This can be controlled by *ConfigAttribute*s (`util/configattributes.cs`) attached to each member.

ConfigNameAttribute gives the name which will be exposed by the configuration dialog (otherwise the variable name is used).

ConfigHintsAttribute provides additional tuning. It can be one or more of the following:

- *ConfigHint.Mask*: don't echo the value;

- *ConfigHint.File*: this should be a file name;
- *ConfigHint.FileChoose*: this should name an existing file;
- *ConfigHint.FileSave*: this should name an existing or new file;
- *ConfigHint.FolderChoose*: this should name an existing folder or directory;
- *ConfigHint.FolderCreate*: this should name an existing or new folder or directory;
- *ConfigHint.Icon*: this should name an icon.

For example, from `search/modules/file/file.cs`; the *FileRoot* property specifies the root of the filesystem to index:

```
//
// where to look for files
//
[XmlElement("root")]
[ConfigName("Index files in")]
[ConfigHints(ConfigHint.FolderChoose)]
public string FileRoot { ... }
```

Finally, properties marked with *ConfigIgnoreAttribute* will not be exposed by the configuration dialog.

4.1.3 Logging

Modules can log their actions through the *Log* class (in `util/log.cs`). In that class, the method *Write(String format, params object[] args)* will write to the current log file, with a timestamp and other debugging information. *Log.Writer* is the *TextWriter* logged information gets written to; this is set when PIS starts and should not need to be changed.

The logger emits a *LogEntry* event whenever a message is ready to be written.

4.2 Filters

Filters are responsible for parsing documents of various formats (typically from local files, but possibly from a networked resource) and returning indexable content such as text or title. At present PIS has filters for plain text documents; HTML [Raggett et al., 1999]; XML [Bray et al., 2006]; PDF;[3] several programming languages; and images.

FilterStream() is responsible for parsing an input stream, extracting indexable text, and returning it. The return value is a *FieldCollection*, mapping field names to text. By convention, the "body" and "text" fields are important; the "body" is the text from the stream and is used for generating query-biased summaries and the "text" is all indexable text. (These may be different. For example, the "text" field typically includes file and directory names so these can be searched, but "body" does not.) Commonly, *FilterStream()* is called as *FilterStream(Stream s, string name)*, and the name is added to "text" automatically — therefore unless there are special contents filters can generally just return the same "body" and "text".

[3] `http://www.adobe.com/devnet/pdf/pdf_reference.html`

Class	Extensions	MIME types
CSharpFilter	.cs	text/x-csharp
HtmlFilter	.htm, .html, .asp	text/html
ImageFilter	.jpg, .jpeg	image/jpg, image/jpeg
JavaFilter	.java	text/x-java-source
JavascriptFilter	.js	text/javascript, text/x-javascript, text/ecmascript
PDFFilter	.pdf	application/pdf
PerlFilter	.pl, .pm	application/x-perl, application/x-perl-module
PlainTextFilter	.txt	text/plain
SourceFilter	(abstract)	
XmlFilter	.xml	text/xml

Table 1: Predefined filter classes in PIS

Each field is stored in the index and can be retrieved later via a corresponding *Result*: for example, the *Title* field corresponds to the "title", and the *Snippet* is generated from "body". Details vary with each subtype of *Result* and *SearchEngine*. See also Section 4.6.12.

Filters are loaded by the general module system. A filter must implement *FilterStream(Stream s)*, as above; it must also implement *MimeTypes*, which lists the MIME types the filter recognises, and *Extensions*, which lists the file extensions (including the initial dot). At initialisation time, generally in the *OnLoad()* function, filters must also call *RegisterFilter(Filter f)* to maintain the type-to-filter and extension-to-filter maps used by *Filter.ForType()* and similar (see below).

4.2.1 General functions

Filter.ForType(string mime_type) returns a *Filter* for the specified MIME type, or null if there is no registered filter. *Filter.ForExtension(string extension)* is similar for file extensions.

Filter.ForFile(FileInfo fi) and *Filter.ForUrl(Uri url)* are similar, for local files and for URLs respectively. *ForFile()* uses the extension by preference, but otherwise tries to guess the MIME type.

These functions are used to extract text either for indexing or for building language models.

4.2.2 HTML

The *HtmlFilter* (in `filter/modules/html/html.cs`) uses the *Majestic12* library[4] to parse HTML. It puts anything from the HTML `<title>` in the "title" field; ignores comments; and puts the text (not HTML attributes, etc) into "body" and "text". The filter is able to switch encodings based on information in `<meta>` tags.

Filter.ForUrl() returns a *HtmlFilter* by default for remote files, if the type is not otherwise indicated.

4.2.3 Images

ImageFilter (`filter/modules/image/image.cs`) assumes there is no indexable content and returns an empty "body" and "text" field.

[4] `http://www.majestic12.co.uk/` and in `filter/modules/html`

4.2.4 PDF

The *PDFFilter* (`filter/modules/pdf/pdf.cs`) uses the *iTextSharp* library[5] to extract the page-rendering code from PDF files. It uses a simple parser to find the PDF text display commands; this seems to suffice in most cases. Files generated by pdflatex however don't encode the whitespace between words, instead printing each word or even each character separately; as a workaround, the filter also keeps track of the position of each piece of text and inserts whitespace if there's enough change in the x coordinate.

4.2.5 Plain text

`filter/filter.cs` includes a *PlainTextFilter*, which reads plain text from a given stream (assumed to be in the default encoding) and returns this in the "body" and "text" fields. On Unix, *Filter.ForType()* returns a *PlainTextFilter* for local files by default.

4.2.6 Source code

SourceFilter, in `filter/modules/source/source.cs`, implements a simple-minded source-code filter which strips any occurrences of a set of keywords from the stream before copying the remainder to the "text" field. The "body" retains the full contents of the stream as usual.

There are four subclasses of *SourceFilter*: *CSharpFilter*, *JavascriptFilter*, *JavaFilter*, and *PerlFilter*. New subclasses should provide a *Keywords* property, listing the keywords to ignore while indexing.

4.2.7 XML

The *XmlFilter* (defined in `filter/modules/xml/xml.cs`) uses a *XmlReader* (from the .NET standard libraries) to parse XML files. Any text or CDATA nodes are extracted and concatenated; both "body" and "text" fields get this concatenated value.

4.3 Sampling and size estimation

Samplers are started in the main loop (*Harness.Harness()*, in `ui/main.cs`) for any search engine without a language model, in order to estimate collection size and build a language model. At present there is only one sampling plug-in, which implements the multiple queries algorithm [Thomas and Hawking, 2007] with result cut-off k = 10,000 and query terms drawn from a set of common English words.

The only size estimation technique used at present is multiple capture-recapture [Shokouhi et al., 2006], using 30 to 100 samples of up to 10 documents each. The algorithm has been modified slightly to allow non-uniform sample sizes, to request more documents if no samples overlap, and to offer a best guess at collection size in the event that none of the eventual samples overlap.

New modules must implement *SamplerName()*, which returns a human-readable name for the algorithm; and *DoSample(uint docs)*, which returns a set of *Result*s. Modules may also override *StopRunning()* if appropriate.

[5] `http://itextsharp.sourceforge.net/` and in `filter/modules/pdf/itextsharp`

4.3.1 Multiple queries sampler

MQSampler (`sample/modules/mq/mq.cs`) implements the multiple queries algorithm of Thomas and Hawking [2007], which appeared from early experiments to be the most reliable sampler across personal collections. It uses $5 \times$ *sample size* samples of up to 10,000 results each, discarding any which overflow or underflow, and chooses documents from the union of all remaining result sets. Possible query terms are hard-coded and are a list of common English words. Result sets can be cached, although in practice this seems to require more memory than it saves.

4.3.2 Size estimation

Size estimation is handled in *Sampler.StartSampling()*, the main sampling loop. The sampler draws at least 30 samples of 10 documents each, and adds each sampled document to a language model (*SimpleModel*, see below). After each round of sampling, *Sampler.UpdateSizeEstimate(List<Set<Result>> results)* uses a version of Shokouhi et al.'s multiple capture-recapture algorithm [Shokouhi et al., 2006], modified to allow non-uniform sample sizes [Thomas, 2008a], to update the sampler's size estimate.

If *UpdateSizeEstimate()* is unable to estimate a size, because no overlaps are encountered, additional samples are drawn until an estimate is possible or 100 samples have been collected.

4.4 Models

Modelling plug-ins provide techniques for working with unigram language models. Separate plug-ins implement simple models read from a plain-text file; models, useful for collections indexed by PIS itself, which get term occurrence data from a local index; a special model for Wikipedia, which is based on all article titles in an August 2007 snapshot; and a special model for the world-wide Web, which is based on term frequencies from a number of large-scale Web crawls.

New modules must subclass *Model* (defined in `search/model.cs`), and must implement *df(string term)*, *tf(string term)*, *NumTerms()*, *Terms()*, and *Docs()*. A new implementation may also wish to override *CalcSums()*, if it has a more efficient way of knowing $\sum_t$ *tf*(t) and $\sum_t$ *df*(t).

4.4.1 Simple

The *SimpleModel* class (`search/model.cs`) is a simple *Model* implementation which must be maintained by the programmer. It can save and load files in a human-readable format (*Read(), FromFile(), Write()*).

4.4.2 Lucene

The *LuceneModel* class (`search/model.cs`) provides a language model view of a Lucene index, which is useful for those search modules which do their own indexing. *df()* and *tf()* are implemented in terms of *IndexReader.DocFreq()* and *IndexReader.Freq()* respectively.

4.4.3 Web

WebModel (also in `search/model.cs`) reads a pre-defined model, in `web.model`. This is intended as a model of the WWW; terms have been extracted from several large-scale web crawls including the .GOV collection from the TREC Web Track and a crawl of the `.au` top-level domain [Ackland et al., 2007].

4.5 Server selection

Four modules perform server selection. Two provide baseline algorithms: a trivial "select all" technique and random scoring. A third implements Kullback-Leibler divergence, including smoothing, and the fourth implementation is of vGlOSS [Gravano and García-Molina, 1995], using the $\text{SUM}(0)$ scoring formula and with weights as described by Gravano and García-Molina.

New algorithms must subclass *ServerSelect* (`select/select.cs`), and must implement *Score(Query q, SearchEngine e)*. Higher scores are assumed to represent more promising servers, although scores need not be consistent between algorithms.

Before servers are scored, the method *Start(IEnumerable<SearchEngine> engines)* is called; the default implementation does nothing, but this can be used to initialise any per-query data. (A separate *ServerSelect* instance is used for each panel.)

4.5.1 Kullback-Leibler

`select/modules/kl/kl.cs` defines *KLSelect*, which uses the Kullback-Leibler divergence between a query and a language model to score servers [Xu and Croft, 1999, Si et al., 2002]. Term frequencies are acquired from each server's language model, and a global model is used for smoothing and mixed in with $\lambda = 0.5$.

4.5.2 Random

The *RandomSelect* class (in `select/modules/random/random-select.cs`) returns a random number from 0 to 1 for every call of *Score()*, ignoring the query and search engine.

4.5.3 Trivial (all)

The *AllSelect* method (`select/modules/trivall/all.cs`) simply returns 1.0 for every call of *Score()*, again ignoring the query and search engine.

4.5.4 vGlOSS

The VGlOSS algorithm of Gravano and García-Molina [1995] is implemented in the *VGlOSSSelect* class (`select/modules/vgloss/vgloss.cs`). It uses the model provided by each search engine to determine term and document frequencies.

4.6 Search

A number of modules provide search capabilities. PIS assumes a hybrid model, so modules can either communicate with a search engine or provide their own indexing and searching. Modules provide search for:

- *Addressbooks*, in standard (vCard) format,[6] and for the KDE[7] addressbook in particular;
- *Calendars*, in standard iCalendar format [Dawson and Stenerson, 1998] and the KDE calendar in particular;
- Local *email*, in maildir,[8] mbox [Hall, 2005], nnml,[9] or babyl[10] format;
- Remote *email*, via IMAP [Crispin, 2003];
- *LDAP directories* [Zeilenga, 2006];
- The *Web*, via Google[11] and Yahoo!;[12]
- *Local files*, in any format for which there is a filter (Section 4.2);
- PostgreSQL *databases*;[13]
- *Bibliographic* and *reference sources*, including Wikipedia,[14] the library of the Australian National University,[15] Project MUSE, and WorldCat;[16]
- Any application with a web interface. Built-in examples are del.icio.us[17] and the public websites of the Australian National University[18] and the CSIRO.[19]
- The output of any external search software, in an XML-based interchange format.

Any new search module must subclass *SearchEngine* (`search/base.cs`), and implement at least *EngineName()* (which returns a human-readable name for the search engine) and *DoSearch(Query query, ResultQueue queue)*.

Given a query and a queue for any results, *DoSearch()* does all the actual search work of PIS. Modules can use the functions *AddResult(Result r, ResultQueue queue)* and *AddResult(DocumentResult r, ResultQueue queue)* to add results to the queue as they become available; *EndResults(ResultQueue queue)* signals the end of a result set. Errors can be signalled with *SendError(Exception e, ResultQueue queue)*. *DoSearch()* runs in its own thread, so most modules will not need to worry about thread-related issues.

Search engines are created at runtime either from the configuration file—so *SearchEngine* instances must be (de)serialisable — or by the user from menu commands. Modules can add to the menu with *AddSample(string[] name, ConstructEngine ce)*, which adds a "sample" search engine users can customise. For example, the following

[6] `http://www.imc.org/pdi/vcard-21.txt`
[7] `http://www.kde.org/`
[8] `http://www.qmail.org/man/man5/maildir.html`
[9] `www.gnu.org/software/emacs/manual/html_node/gnus/Mail-Spool.html`
[10] `http://quimby.gnus.org/notes/BABYL`
[11] `http://code.google.com/`
[12] `http://developer.yahoo.com/search/`
[13] `http://www.postgresql.org/`
[14] `http://www.wikipedia.org/`
[15] `http://library.anu.edu.au/`
[16] `http://www.worldcat.org/`
[17] `http://del.icio.us/`
[18] `http://www.anu.edu.au/`
[19] `http://www.csiro.au/`

code (from `search/modules/email/mbox.cs`) adds a sample when the containing module is loaded; the menu item is "Search email in mbox or nnml file(s)" and lives in the "Email" submenu.

```
//
// register a sample
//
public static new void OnLoad() {
  AddSample(new string[] {
    "Email", "Search email in mbox or nnml file(s)"
  }, delegate {
    MboxSearch m = new MboxSearch();
    return m;
  });
}
```

Any number of samples are allowed (see for example `search/modules/web/web.cs`, which registers several).

Search engine implementations may also want to overload *Start()*, *StopRunning()*, *CleanUp()*, or *ForceIndex()*.

Two subclasses may be useful for future extensions. *IndexerSearchEngine* adds convenience functions for hybrid models, which do their own indexing on a periodic basis. Implementations of this class must implement *Reindex()*. Finally, *LuceneSearch* adds convenience functions for indexing with lucene. Users of this class can call *Writer(bool create)* to write to the index, and *Reader()* to read from it.

4.6.1 Addressbook

`search/modules/addressbook/addressbook.cs` defines the *AddressBook* engine, which indexes an addressbook in the standard vCard format. It uses a *LuceneSearch* to build an index each time the file changes, and returns *PersonResult*s. `search/modules/addressbook/vcard.cs` defines a simple vCard parser.

4.6.2 Calendar

The very similar *Calendar* class (`search/modules/calendar/calendar.cs`) indexes and searches calendars in the iCalendar format [Dawson and Stenerson, 1998]. Results are instances of *AppointmentResult*, which extends *DocumentResult*s with summaries, start and end times, and locations.

`search/modules/calendar/ics.cs` defines classes for reading iCalendar files. To-do items are not currently handled.

4.6.3 Database

PIS includes modules for connecting to databases and making query results available as search results. The *Database* class (`search/modules/database/database.cs`) defines routines for issuing database commands and returning PIS *Result* instances. Named columns from the database are copied into the title, snippet, and URL fields; any extra columns are copied to named "extra" fields. Any new implementation must define *MakeConnection()* and *MakeCommand(Query query, IDbConnection conn)*.

SqlDatabase extends *Database* with a query parameter, an SQL query, and implements *MakeCommand()*.

The *Postgres* class (in `search/modules/database/postgres.cs`) further extends *SqlDatabase* to speak to PostgreSQL database, using the *Npgsql* library.[20]

4.6.4 Email

Various classes in `search/modules/email` handle parsing, indexing, and searching email repositories in different formats.

`search/modules/email/email.cs` defines several utility classes used elsewhere. *EmailResult* is the basic result type for email searches: it includes sender and recipient addresses, date, and subject (the latter overriding the inherited *Title*). This class also generates appropriate markup for result display.

LuceneEmail extends *LuceneSearch* to provide a class to handle indexes of email messages. It includes *AddMessage(IndexWriter writer, Stream s, Document doc)*, which reads an email message in RFC 2822 format from a stream and adds it to an index. The message may be in MIME format, in which case *AddMessage()* will try to find text parts to index. It does not (yet) use filters to interpret message parts.

IMAP *ImapSearch*, in `search/modules/email/imap.cs`, implements a simple IMAP Crispin [2003] client. It subclasses *IndexerSearchEngine* to handle period re-indexing, and uses *LuceneEmail* to handle indexing. Results use `imap://` URLs, which can be opened at least on KDE.

Since the *Lumisoft* library (`lib/Lumi`) unfortunately ignores the UIDVALIDITY given by the IMAP server, *ImapSearch* uses a combination of UID and size to identify objects and will re-index any objects whose size changes. The module can also either index, or ignore, unsubscribed folders.

mbox *MboxSearch* (`search/modules/email/mbox.cs`) subclasses *FileSearch* to index the standard "mbox" email format [Hall, 2005], commonly used on Unix and by several non-Unix mail tools. Mboxes include several messages in one file, so the *AddFile()* implementation looks for the "From:" lines which start each message.

A *LuceneEmail* instance handles the details of parsing messages, as well as indexing and searching.

MboxSearch returns *MboxResult*s, which subclass *EmailResult*s. *MboxResult*s include the file the message came from and the range of line numbers; *MboxResult.Activate()* extracts this line range to a temporary file before opening a viewer.

maildir *MaildirSearch*, in `search/modules/email/maildir.cs`, is a *FileSearch* derivative which handles the "maildir" repository format where each message is contained in its own specially-named file. It includes any messages in version two of the format, marked by ":2" in the filename, with the exception of messages marked for deletion; and will recurse into subdirectories as needed. Again, a *LuceneEmail* instance handles parsing, indexing, and searching.

[20] `pgfoundry.org/projects/npgsql/` and in `search/modules/database/Npgsql1.0`

BABYL BABYL, used mainly by rmail and gnus, is a similar format to mbox. *BabylSearch* (`search/modules/email/babyl.cs`) is implemented as a subclass of *MboxSearch*, with modifications in *AddFile()* to look for BABYL-style EOOH lines and message dividers.

nnml This is another gnus format. It is essentially identical to mbox and is handled in the same way by *MboxSearch*.

4.6.5 External

External (`search/modules/external/external.cs`) allows PIS to use arbitrary external search engines based on an XML interchange format. External programs are fed queries on standard input, in plain text and followed by a newline; they are expected to return a well-formatted *ResultList*. External programs may accept multiple queries, or can be re-started for each query, as appropriate.

Models for *External* search engines are not presently well-defined.

4.6.6 Fake

Fake (`search/modules/fake/fake.cs`) returns invented "search results" which do not correspond to any real document; this may be a useful stub for new search modules and is occasionally useful for testing the GUI. *DoSearch()* simply returns a set of "documents" with random titles and URLs.

4.6.7 File

The *FileSearch* search engine, in `search/modules/file/file.cs`, is possibly the most complex of the pre-defined modules. It watches a directory and all its contents for any changes, such as additions, deletions, or renamings; each changed or added file will be added to a queue and processed in a separate indexing thread.

For directories, this indexing thread simply recurses; for a new or modified file, an appropriate filter is found and the filtered contents are added to a *LuceneSearch* index. Files are identified in the index by a ⟨ full name, size, last modified ⟩ triple.

By redefining *ShouldIncludeFile(string name)* and/or *ShouldIncludeDirectory(string name)*, subclasses can implement additional filtering. The implementation here uses two regular expressions, *Include* and *Exclude*; any file or directory which matches the former and not the latter is processed.

Results are returned as *FileResult*s, which records typical file information such as path and time last modified.

4.6.8 Google

The *GoogleAPI* engine (in `search/modules/google/google.cs`) makes use of Google's old SOAP API to search the web, via the *GoogleSearchService* library. This API is unfortunately no longer supported, although the Yahoo! API was always more reliable and up-to-date.

SiteRestrictedGoogle adds a "site:" clause to each query.

4.6.9 LDAP

The *LDAP* class (`search/modules/ldap/ldap.cs`) uses Novell's LDAP library to search an LDAP directory [Zeilenga, 2006]. It returns results of type *LDAPResult*, which subclasses *PersonResult*.

4.6.10 Web

Anything with a web front-end can be searched in PIS via the *Web* module (`search/modules/web/web.cs`), which acts as a screen scraper. The scraper can masquerade as a different user agent, and can use a specified username and password where required by restricted sites. Any HTTP errors are returned via *SendError()*.

Different services can be searched by specifying three parameters. First, the URL from which to scrape: the token "{0}" will be replaced with the query text, reformatted appropriately for HTTP (*Web* can only use the GET method at present). The second, a regular expression describing which parts of the returned HTML to accept as search results, is typically complex. It uses .NET "named groups" to specify which parts of the matched string are title, URL, and snippet. These are marked with the sequence "(?<name>...)".

The final parameter specifies which matches from the previous regular expression should in fact be excluded.

For example, one sample added by *Web.OnLoad()* describes Wikipedia in the following way:

```
 Web w = new Web();
 w.Url = "http://en.wikipedia.org/wiki/Special:Search \
?search={0}&fulltext=Search";
 w.Include = "<a href=\"(?<url>/wiki/.*?)\" \
title=\"(?<title>.*?)\">";
 w.Exclude = "(Special|Help|Wikipedia):";
```

which points at "Special:Search" for searching, with the "search" parameter set to the user's query; takes the URL from any "a href" sequences starting with "/wiki"; takes the title from the same link; and removes several special Wikipedia links.

Web includes several samples, including Wikipedia; Project Muse; WorldCat; and services from CSIRO as well as the ANU.

4.6.11 Yahoo

The *YahooAPI* class (`search/modules/yahoo/yahoo.cs`) uses the Yahoo! REST API to search the web. It uses the .NET web and XML libraries to handle (de)serialisation of the query and results.

Details of the Yahoo! API are online at `http://developer.yahoo.com/search/`.

4.6.12 Results

The *Result* class (`search/result.cs`) represents results from search engines: both documents to present to the user and "results" in a broader sense, such as errors.

A *DisplayResult* (also in `search/result.cs`) is anything we can display; each has some text (*Markup*) and optionally an icon. *DisplayResult*s can also have some

actions (each a *ResultAction*), which are exposed in the UI — for example, a result representing a person may have actions such as "email this person" or "add to address-book". *Activate()* is called when the result is selected in the GUI.

*DisplayResult*s can emit a *ResultChanged* signal when they change, which will be noticed in the GUI. This is useful for, e.g., lazily loading details in a background thread.

In most cases results will be *DocumentResult*s or subtypes; this is a single result from a search engine. *DocumentResult*s each may have a URL; a URL for display (which may be different, e.g. in the presence of clickthrough tracking); a title; a snippet, possibly query-baised; a note; and a server-assigned score. Markup is generated automatically from these attributes unless it is separately defined. A *DocumentResult* may also include an arbitrary set of other attributes, indexed by string (as `a_result["key"]`). *DocumentResult.Activate()* tries to launch a viewer for the given URL.

There are further specialisations of *DocumentResult* which may be useful (also in `search/result.cs`):

- *WebResult* lazily loads a favicon in a background thread and simply emits *ResultChanged* when it's done.
- *PersonResult* includes useful attributes like name, location, telephone number, etc.

And for communicating between search engines and the search manager:

- *ResultEndMarker* signals the end of a set of results.
- *ResultError* encapsulates an exception, to report an error in the search.

A *ResultList* is a list of results, useful for (de)serialising.

New result types may need to override *OpenStream()*, if the result represents a readable document (although note that *DocumentResult.OpenStream()* can handle most web-accessible documents); and *Activate()*, if appropriate.

4.7 Sorters

Finally, five different algorithms are available for sorting and merging the results from the selected search engines. Trivial algorithms provide first-in-first-out "sorting" and random "sorting"; PIS also includes an algorithm which sorts results first by collection then by server-assigned score, and an implementation of Rasolofo et al.'s SM-TSS algorithm [2003]. A final algorithm removes any results from a specified collection before passing the remainder to another sorter.

New sort modules must subclass *ResultSort*, and since they are saved in the configuration they must be serialisable.

InsertResult(TreeStore store, DisplayResult r) is responsible for sorting: given a result *r* and a tree *store* of existing results (possibly empty), it must insert the result at an appropriate place. The tree can be modified as needs be.

New implementations may also wish to overload *Start(TreeStore store, Query q)*, which is called before each set of *InsertResult()*s, and *Finish(TreeStore store)*, which is called afterward.

The subclass *ScoreSort* (`sort/score.cs`) sorts results into a flat list by score, which suffices for many modules. Implementations of this subclass need to implement *GetScore(DisplayResult r)*.

4.7.1 By source FIFO

BySourceFIFO (in `sort/modules/bysourcefifo/bysourcefifo.cs`) sorts results in a two-level tree: first by source search engine, then in order of arrival. It defines a *DisplayResult* subclass, *HeaderLine*, to provide the search engine headers.

4.7.2 Drop

The *DropResults* "sorter" (`sort/modules/drop/drop.cs`) simply drops all results from a specified search engine before passing the remainder to another sorter. It has been used to support experiments in server selection by ensuring results from certain servers do not appear.

4.7.3 FIFO

The very simple *FlatFIFO* sorter (in `sort/sort.cs`) simply appends results to a flat list as they arrive, without regard to the source or to anything else.

4.7.4 Rank

RankSort (`sort/modules/rank/rank.cs`) uses *ScoreSort* and a simple scoring formula to interleave results such that the first result from all seach engines appear first; then the second-ranked results; and so on.

4.7.5 Random

In `sort/modules/random/random-sort.cs`, *RandomSort* uses *ScoreSort* and a PRNG to sort results randomly.

4.7.6 SM-TSS

RasolofoSMTSS (`sort/modules/rasolofo/rasolofo.cs`) implements the SM-TSS algorithm of Rasolofo et al. [2003]. This algorithm counts query words in the title or snippet, and falls back to considering the rank assigned by the relevant search engine.

5 Support for experiments

PIS includes some support for experiments, and can record user interface actions and periodically send a log back to researchers. The *ExperimentSupport* class, in `util/expt.cs`, includes the *Log(string format, params object[] args)* function which records a message; and *SendData()*, which is periodically called and sends the accumulated data over HTTP to a CGI script running on the researchers' machine.

6 Utility classes

A small set of utility classes might be useful in extending PIS, or writing further modules.

- `util/compare.cs` includes *Backwards*<*T*>, an *Icomparer* which reverses the default sort order;

- *Format* (`util/format.cs`) includes static functions for formatting dates and times, and various numeric types;
- `util/pango.cs` defines the *Pango* class, which includes functions for escaping characters such as "<" which would otherwise be interpreted as formatting instructions for Pango;[21]
- *Platform* (`util/platform.cs`) includes functions for detecting the desktop environment (KDE, Gnome, "other Unix", or Windows); getting the MIME type of a file; opening a URL; and various GUI and file manipulation utilities;
- *Set* (in `util/set.cs`; by Rüdiger Klaehn) is a simple set class;
- `util/syncqueue.cs` includes *SyncQueue*, a generic queue class which enforces mutual exclusion when necessary;
- the *TypeRegistry* in `util/types.cs` maintains the list of types needed for XML (de)serialisation;
- and `util/words.cs` includes functions for extracting words from strings, optionally removing duplicates, and for collapsing repeated whitespace.

References

R. Ackland, A. Spink, and P. Bailey. Characteristics of .au websites: An analysis of large-scale web crawl data from 2005. In *Proc. 13th Australasian World Wide Web Conference (AusWeb07)*, 2007.

T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0. `http://www.w3.org/TR/2006/REC-xml-20060816/`, 2006.

N. Craswell, F. Crimmins, D. Hawking, and A. Moffat. Performance and cost tradeoffs in web search. In *Proc. Australasian Database Conference*, 2004.

M. Crispin. Internet message access protocol — version 4rev1. RFC 3501, 2003.

F. Dawson and D. Stenerson. Internet calendaring and scheduling core object specification (icalendar). RFC 2445, 1998.

L. Gravano and H. García-Molina. Generalizing GlOSS to vector-space databases and broker hierarchies. In *Proc. VLDB*, 1995.

E. Hall. The application/mbox media type. RFC 4155, 2005.

D. Raggett, A. L. Hors, and I. Jacobs. HTML 4.01 specification. `http://www.w3.org/TR/1999/REC-html401-19991224/`, 1999.

Y. Rasolofo, D. Hawking, and J. Savoy. Result merging strategies for a current news metasearcher. *Information Processing and Management*, 39(4), 2003.

M. Shokouhi, J. Zobel, F. Scholer, and S. M. M. Tahaghoghi. Capturing collection size for distributed non-cooperative retrieval. In *Proc. ACM SIGIR*, 2006.

[21] `http://www.pango.org`

L. Si, R. Jin, J. Callan, and P. Ogilvie. A language modeling framework for resource selection and results merging. In *Proc. CIKM*, 2002.

P. Thomas. Generalising multiple capture-recapture to non-uniform sample sizes. In *Proc. ACM SIGIR*, 2008a. Poster.

P. Thomas. *Server Characterisation and Selection for Personal Metasearch*. PhD thesis, Australian National University, 2008b.

P. Thomas and D. Hawking. Evaluation by comparing result sets in context. In *Proc. CIKM*, 2006.

P. Thomas and D. Hawking. Evaluating sampling methods for uncooperative collections. In *Proc. ACM SIGIR*, 2007.

J. Xu and W. B. Croft. Cluster-based language models for distributed retrieval. In *Proc. ACM SIGIR*, 1999.

K. Zeilenga. Lightweight directory access protocol (LDAP): Technical specification road map. RFC 4510, 2006.